DESIGN AND APPLICATION OF NETWORK-ON-CHIP VIRTUAL PROTOTYPING PLATFORM

LIM ZHEN NING

MASTER OF ENGINEERING SCIENCE

FACULTYOF ENGINEERING AND GREEN TECHNOLOGY
UNIVERSITI TUNKU ABDUL RAHMAN
JANUARY 2015

DESIGN AND APPLICATION OF NETWORK-ON-CHIP VIRTUAL PROTOTYPING PLATFORM

By

LIM ZHEN NING

A Dissertation submitted to the Department of Electronic Engineering,
Faculty of Engineering and Green Technology,
Universiti Tunku Abdul Rahman,
in partial fulfilment of the requirements for the degree of
Master of Engineering Science
January 2015

DEDICATION

Dedicated to my family and friends

ABSTRACT

DESIGN AND APPLICATION OF NETWORK-ON-CHIP VIRTUAL PROTOTYPING PLATFORM

The progressive growth in the complexity of Multiprocessor System-on-Chip (MPSoC) designs to meet demands on low power, speed, performance as well as functional features has increased the level of complexity of component and system level modelling for design verification. In this research, a reconfigurable and scalable verification environment for Network-on-Chip (NoC) systems to allow early verification and validation is proposed.

Various component level verification environments based on the Universal Verification Methodology (UVM), such as the Advanced High-performance Bus (AHB), the Advanced Peripheral Bus (APB), and the AHB-to-NoC Bridge, have been developed. Generic *Sequences* which, can be extended to model various peripherals such as memory controller, GPIO, SPI, and timer are used within the corresponding environments. To coordinate between multiple *Sequences*, *Virtual sequencers* are used. *Analysis ports* are used to collect transfers from different hierarchies of the verification environment for analysis. These component level environments are used to build up the sub-system and system level environment. The system level environment will be used to verify the designs from specifications to gate level implementation.

Configuration Objects are used to provide re-configurability of the verification environment in various hierarchical levels, from component to system. Generally, this allows the models to be configured based on the specific configurations such as mapping the slave models on different memory addresses, and defining the number of routers that is used so that a basic verification framework can be generated.

To demonstrate scalability and re-configurability, verification environments have been set up to verify a few NoC architectures. *Scoreboards* and *checkers* are implemented to verify the correctness of the transactions. Functional and code coverage measurements are taken to ensure the design has been tested thoroughly.

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr. Loh Siu Hong and my Co-supervisors, Dr. Yap Vooi Voon and Prof. Lee Sze Wei, and Mr. Tang Chong Ming for their invaluable advices, guidance and enormous patience throughout the development of the research.

My sincere gratitude to Mr. Ng Mow Song for his contributions in this research. This project would not be that successful without his guidance and insight perspectives in verification.

.

In addition, I would also like to express my gratitude to my loving parent who had helped and given me encouragement and fellow postgraduate teammates, Dicky Hartono, Felix Lokananta, and Arya Wicaksana in the discussions.

Last but not least, the gratitude also to all my other friends who give ideas and suggestions. Without all of them, this project would not be as successful as it is. Thank you once again for all the ideas, suggestions, guidance, efforts and skills that have been given to me.

APPROVAL SHEET

This dissertation/thesis entitled **"DESIGN AND APPLICATION OF NETWORK-ONCHIP VIRTUAL PROTOTYPING PLATFORM"** was prepared by LIM ZHEN NING and submitted as partial fulfilment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

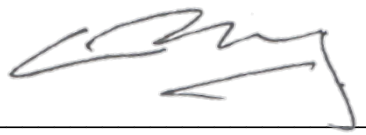
Approved by:

_____

(Dr. Loh Siu Hong)
Date:……..
Supervisor
Department of Electrical and Electronic Engineering
Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

_____

(Dr. Yap Vooi Voon)
Date:……..
Co-supervisor
Department of Electronic Engineering
Faculty of Engineering and Green Technology
Universiti Tunku Abdul Rahman



_____

(Mr. Tang Chong Ming)
Date:12-01-2015
Co-supervisor
Department of Electronic Engineering
Faculty of Engineering and Green Technology Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 15 Jan 2015

SUBMISSION OF DISSERTATION

It is hereby certified that ***Lim Zhen Ning*** (ID No: ***12AGM00015***) has completed this dissertation entitled "***DESIGN AND APPLICATION OF NETWORK-ON-CHIP VIRTUAL PROTOTYPING PLATFORM***" under the supervision of Dr. Loh Siu Hong (Supervisor) from the Department of Electronic Engineering, Faculty of Engineering and Green Technology , Dr. Yap Vooi Voon Co-Supervisor from the Department of Electronic Engineering, Faculty of Engineering and Green Technology and Mr. Tang Chong Ming Co-Supervisor from the Department of Electronic Engineering, Faculty of Engineering and Green Technology .

I understand that University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

_____

(*LIM ZHEN NING*)

DECLARATION

I hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

Name Lim Zhen Ning

Date 15 Jan 2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF Figures

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
AHB	Advanced High-performance Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus, including pipelined access, bursts, split and retry operations.
AMBA	Advanced Microprocessor Bus Architecture. Bus system defined by ARM Technologies for system-on-chip architectures.
APB	Advanced Peripheral Bus. Peripheral bus definition within the AMBA 2.0 specification. The Bus is used for low power peripheral devices, with a simple interface logic.
ASIC	Application Specific Integrated Chip
AVM	Advanced Verification Methodology
BFM	Bus Functional Model
BIST	Build In Self-Test
CDV	Coverage Driven Verification
DUT	Design Under Test
eRM	E Reuse Methodology
ESL	Electronic System Level
HDL	Hardware Description Language
HW	Hardware
IP	Intellectual Property
NoC	Network-On-Chip
OOP	Object-Oriented Programming
OVM	Open Verification Methodology
RTL	Register Transfer Level. Description of hardware at the level of digital data paths, the data transfer and its storage.
SoC	System-On-Chip. A highly integrated device implementing a complete computer system on a single chip.
STA	Static Timing Analysis
SV	System Verilog
SW	Software
TLM	Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.
URM	Universal Reuse Methodology
UVC	Universal Verification Component
UVM	Universal Verification Methodology
VIP	Verification IP
VMM	Verification Methodology Manual

CHAPTER 1

INTRODUCTION

1.1 *Background and Motivations*

The semiconductor industry as we know it today is still very much on an unrelenting pursuit of Moore's Law. Moore's Law predicts that device density in digital CMOS integrated circuits doubles approximately every two years. This prediction still holds true today to a great extent because the law is widely used in the semiconductor industry to guide long-term planning and set targets for research and development. We, therefore, witness exponential improvement in performance as well as the miniaturization of any product that uses mainly microelectronic chips. It also opens up the possibilities of higher levels of intellectual property (IP) integration, even multiple systems, in a single chip increasing significantly the functional capabilities and performance within the same die area.

SoC companies are able to leverage these technology advancements to innovate and differentiate their market offerings which includes single core to multicore system solutions. These design possibilities leads to various improvements such as high speed and performance as well as bigger embedded storage within the same chip size. Complex systems are now able to be embedded into Multi-Processor System-On-Chip (MPSoC).

However, MPSoC inherit board level interconnect issues whereby interconnecting the IPs with conventional busses is not possible anymore as the number of cores increases. This is because conventional bus structures take up routing resources, making the chip not routable and it is not cost effective to increase the chip size. Hence, research on the interconnection between the processor cores has also been rapidly evolving from the conventional bus interconnect in MPSoC to use terrestrial network based concept as interconnect. On-chip networks, or Network-On-Chip (NoC), will allow the architecture to be scalable and adaptive as each core will be connected to a node in the network. A few core, thus connected forms a local network cluster and several clusters can be similarly linked to form a global network of clusters.

The challenge now is in the verification of such an MPSoC design. It is common knowledge in SoC design circles that design verification efforts easily take up to 80% of the time and resources in a chip design project. Every additional IP core that is integrated into the MPSoC multiplies the state-space and the level of complexity of component and system level modelling for design verification. An MPSoC consists of multiple functional blocks, processors, protocols, interfaces and peripherals. In order to obtain a satisfactory functional verification coverage, the interactions between the functional block as well as the functional block's operating mode have to be tested and exercised thoroughly. Now add to this the chip design imperative to be

able to detect every defect that will be present in silicon wafers, the enormity of the task becomes gravely apparent. A strict methodology and disciplined is called for. The ability to scale and re-use any verification components and models created in the effort is highly desirable.

1.1.1 Research Objectives and Approach

In this research, a scalable, reconfigurable, reusable verification platform for the verification of an MPSoC design with on-chip network communication is proposed. The followings are the research objectives of the research:

- Acquire ASIC design skills in such design, especially in "First-Time-Success" ASIC Design Methodology
- Apply industry standard tools for the verification methodology
- Design the scalable verification environment for another concurrent research on the SoC architecture.
- Demonstrate the scalability of the verification environment for architecture exploration

Due to the high cost of masks and wafer fabrication in current deep submicron technology ASIC design failure is not an option. A first-time-success ASIC Design Methodology is strictly adhered to in designing our MPSoC and the verification platform so that the design

process is systematic, deliberate and schedules manageable. In terms of verification, the process also consumes a lot of resources and computational power in designing and developing the test scenarios. Therefore, an industrial standard methodology known as the Universal Verification Methodology (UVM) is adopted in designing the verification platform. In addition, the platform has to be reusable, reconfigurable and scalable so that it can be used not only in another concurrent MPSoC design project with on-chip network interconnects but also for other future projects.

Functional design verification starts with directed test cases that model numerous scenarios to exercise the functionality of the design. All corner cases must be carefully and exhaustively considered. The increasing number of cores in the multicore system design further complicates matters. These cores working independently running its own firmware add on to the existing scenario modelling complexity. Functional coverage is used as a mean of measurement to track the functional verification progress.

As the crystalline nature and the manufacturing process of silicon wafers are bound to have defects, complete fault coverage is essential. Each node of the system is injected with stuck-at-1 and stuck-at-0 fault, the fault should be able to be observed at the output of the system during each simulation. If the fault cannot be observed, new test patterns will be generated or changes to the design is made. In a complex

design with millions of gates, these fault simulations are extremely time consuming and difficult. Code coverage is the best effort alternatives for this situation. Code coverage reflects how thoroughly the Hardware Description Language (HDL) code has been exercised. Code coverage tools usually provide line coverage, arc coverage for state machines, expression coverage, event coverage, and toggle coverage. Toggle coverage gives an approximation of the fault coverage and the quality of the test patterns. Coverage closure defines the measurement of the quality of the verification suite in generating stimulus to exercise the Design Under Test (DUT).

Constrained Random Verification is used to speed up the coverage closure process. Using a constrained random approach, stimulus within the constraints are autonomously generated. Describing the stimulus this way is more concise, easier to review and more productive. By utilizing constraint random solver in formal tool such as Synopsys VCS, all possible stimuli within the valid stimulus space that has been defined in the constraints, which may not be otherwise anticipated are generated to exercise the DUT. This stimulus is critical to cover unexpected cases during the verification process.

If the stimulus is randomly produced, there is a high probability that the generated stimulus is repeated introducing redundancy of the stimulus. This situation will result in difficulties in archiving the expected coverage closure. To further understand the problem, consider

the “Coupon Collector’s Problem” from probability theory (Ballance, 2009). The subject to the problem is the number of trials required to collect a full set of coupons from a limitless uniformly distributed random collection. In the early stage, the coupon collection slot can be easily filled up as the probability of a new coupon selection is high compared to the previously selected coupon. As the coupon collection reaches its complete collection, the probability of getting the duplicated coupon is higher. From this theory, the estimated number of trials needed to completely collect the set of coupon is estimated to be O (n*In (n)). Consider a collection that consists of 50 coupons, then, about 196 random trials are needed to archive the full collection. Thus, a uniformly distributed random stimulus will result in 10-30% of efficiency. In contrast, if the redundant stimulus is removed, then, the coverage closure could be archived 5-10 times faster. Rules or constraints are added to the stimulus generation to remove the redundancy, allowing constrained random stimulus to be generated.

The stimulus generation or tests and verification environment that interface directly to the Design Under Test (DUT) is separated. This is to ensure that the same verification environment can be reused with multiple tests and enable the tools to merge the coverage report from these tests.

1.1.2 Dissertation Organization

The remaining part of the dissertation, Chapter 2 is organized with a literature review on designing the scalable and reconfigurable virtual prototyping platform, including surveys on current verification languages and methodologies adoption trends. An introduction to the AMBA bus is then followed by an overview of our chip specifications. Chapter 3 discusses the ASIC Design Methodology and the Universal Verification Methodology (UVM) which has been adopted. Chapter 4 introduces to the readers our proposed hierarchically reusable and reconfigurable verification platform architecture that we have developed. In Chapter 5, the results from our proposed platform will be discussed. This chapter will also discuss the various measurements taken to ensure the chip is fully verified. Finally, Chapter 6 concludes and summarizes the dissertation and the recommendations for further works.

CHAPTER 2

LITERATURE REVIEW

2.1 *ESL Design levels*

In the exploratory stage of a VLSI design project, it is essential to evaluate the trade-off between simulation accuracy and speed because early architectural exploration of the design could lead to better decision in designing the SoC and reducing major architectural changes late in the project timeline.

2.1.1 Evolution of the ESL design flow

The system architects require a fast and accurate simulation of an MPSoC that are capable of running real application softwares. The current MPSoC design flow has changed, considering the complexities involved (Ghenassia, 2005). Modelling at a higher abstraction level, the Electronic System Level (ESL) focuses on the functionality rather than its implementation allows faster yet reasonably accurate simulation results (David et al. , 2009). A technique, Transaction Level Modeling (TLM) has evolved to aide this task.

2.1.2 TLM Overview

TLM can be used to model the functional specifications from the customers by describing the system exchanging transfers in the form of transactions over the channels. TLM interfaces are implemented within the channels to encapsulate the communication protocols. Communications are established by accessing these interfaces through the module ports. This allows the descriptions to be abstract, not encumbered by implementation details like clocks, drive strengths, signal delays, data flow and so on (Ashwin et al. , 2013). These models are refined as the design process evolves. Along with the ESL introduction, there is also a need to have languages that are able to support the features. Currently, only SystemC and System Verilog supports TLM.

2.2 *Verification Languages*

When the Verilog Hardware Description Language was created in the mid-1980s, the typical design was of the order of five to ten thousand gates. The size and complexity of hardware designs and verification has quickly outgrown the capabilities of Verilog and VHDL because the amount of codes required becomes significantly larger which was becoming unmanageable and inefficient.

2.2.1 Early Verification Languages

Verisity launched the Specman e language which built upon object-oriented programming (OOP) to deliver aspect-oriented programming (AOP) and soon it became a main verification language. AOP approach allows new functionality to be added to the existing code in a non-invasive manner. AOP also addresses same feature in various sections of the codes concern by allowing the existing structs to be extended to add additional functionalities.

Synopsys also introduced its own OpenVera. These commercial tools are costly. In some cases, companies resorted to the test benches written using C or C++ and would drive the DUT through Programming Logic interface (PLI). PLI allows C or C++ functions to be invoked from Verilog.

2.2.2 System Verilog Language

System Verilog which is a significant enhancement from its predecessor, Verilog includes major extension into abstract design, testbench, and C based APIs has emerged. This extension is an integration of the features from SUPERLOG, VERA, C, C++, and VHDL along with OVA and PSL assertions (Chris & Greg , 2012). It allows System Verilog to be effectively used as Hardware Description and Verification Language (HDVL) because it provides synthesizable construct, capabilities of hardware modelling at RTL, system and

architectural levels and various verification features such as classes, constrained random stimulus and coverage. (Mentor & Cadence, 2007). From the functional verification study done by Mentor Graphics under Wilson Research Group, it is notable that System Verilog is widely used in the industries, especially for designs in the multi-million gate region (Foster, 2013).

2.2.3 SystemC Language

While System Verilog can be seen as a bottom up approach, extending Verilog with OOP features to allow the representation of the system level design in a more abstract manner and access to systems described using high-level descriptions such as C or C++, and SystemC. The approach builds upon the designers' familiarity with C or C++ to provide the libraries needed for HDL modelling. This gives a top down approach in the SoC system design. (Donatella et al. , 2004). It has been the researcher's and EDA vendors' interest to seamlessly synthesize C/C++/SystemC codes to targeting hardware implementation. The ability to combine System Verilog and SystemC in a single hardware/software co-verification platform let the designers leverage the fast simulation speed while providing a platform for concurrent hardware/software development. SystemC model can be used as a golden reference model to the intended design modelled using HDL while System Verilog can be used to develop the automated verification platform since the verification features such as constraint random stimulus and coverage are more prominent in System Verilog (Black,

2013). The language itself can provide useful constructs for the users, but, designers need to go beyond just knowing them. They require some guidelines or methodology to fully utilize the language (Ruggiero, 2009).

2.3 *Verification Methodologies*

Despite the richness in language itself that are able to provide the needs to implement complex verification environment, best practices and shared understanding between engineers within the workplace, and code reusability has called for a standard methodology (Bromley, 2013). Some EDA Tool vendors provides user guides in terms of more detailed code examples while illustrating the basic concept of their own methodology. These become a problem when the designer or verification engineers' tries to use tools from different vendors as the vendors based their tools upon their own verification methodology (Anderson, 2010).

2.3.1 Early Verification Methodologies

One of the early verification methodology is Verification Advisor (vAdvisor), a comprehensive collection of best practices and advisory for users of the e verification language developed by Verisity Design. This leads to the development of the e Reuse Methodology (eRM).

The methodology provides guidelines on the naming conventions to avoid interference between verification components. The methodology also introduces the concept of *Sequences* to enable e Verification Component (eVCs) to generate and synchronize complex multi transaction scenarios in the verification environment. These *Sequences* are passed to the BFM through the *Sequence Driver*. Functional partitioning of the testbench is another concept introduced in the methodology. Bus Functional Model (BFM) is used to drive the DUT while the *Monitor* monitors the DUT. The *Monitors, Sequence Drivers,* BFMs and *Configuration Objects* are encapsulated within the *Agent*. These *Agents* with a global *Configuration Objects* are encapsulated within the environment (Shvartz, 2003). Figure 2-1 shows the overall architecture of this methodology.

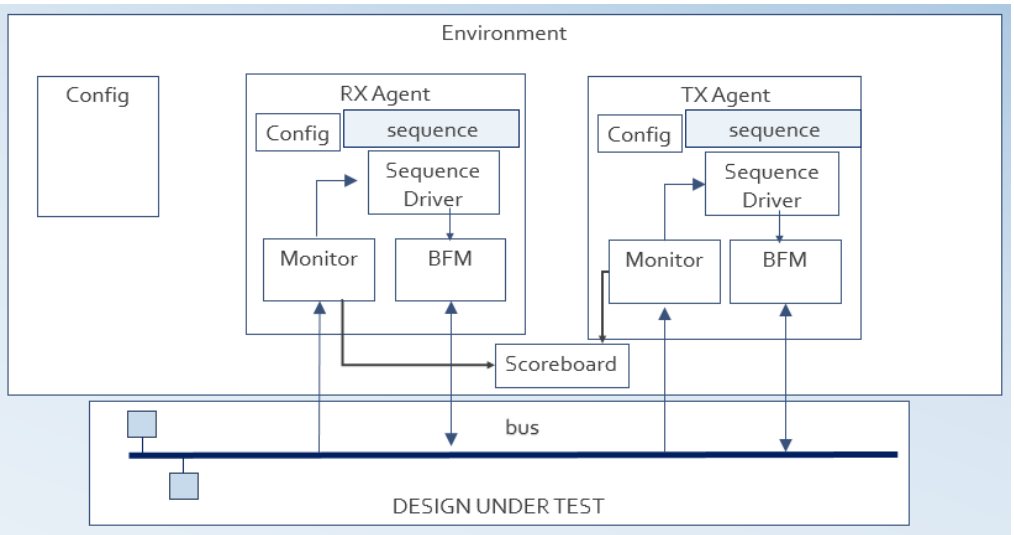


Figure 2-1: eRM functional partitioning testbench (Verisity, 2004)

This verification methodology was used in developing a reusable verification environment for an Ethernet IP core (Swarna et al. , 2008). It was already evident that the concept of reusability of the verification component and *Sequences* had been the focus in verification (Krolikoski, 2011).

Shortly thereafter, Synopsys introduced the Reference Verification Methodology (RVM) based on its own OpenVera verification language. Based upon RVM, Synopsys published the Verification Methodology Manual VMM). The methodology provides guidelines in creating a layered verification architecture to allow the reusability of the components. Interfaces are used to connect the verification environment with the DUT. This provides pin name abstractions that can be used with different DUTs as well as different model implementation of the same DUT (Janick et al. , 2005). A verification platform based on this methodology has been developed to verify the Yak SoC (Lu et al. , 2009).

Mentor Graphics also introduces their own verification methodology, Advanced Verification Methodology (AVM) for SystemC and System Verilog. The methodology provides a framework for component hierarchy and TLM communication to provide a standardized use of the model in SystemC and System Verilog verification environment.

Cadence then acquired Verisity and transformed its eRM into the Universal Reuse Methodology (URM) which supports System Verilog, e as well as SystemC.

2.3.2 Open Verification Methodology (OVM)

Although the EDA vendors' methodologies had their success with their own customer base which ran on its own simulators, there is no attempt at cross vendor support or any form of standards. Cadence and Mentor Graphics later took the initiative by published an open source verification methodology, Open Verification Methodology (OVM). This methodology became widely adopted because of its reusability of various verification components at different hierarchical levels of the design and different projects, and also reusability of the verification components with different tests whereby the stimulus generation or tests are separated from the verification environment (Malik et al. , 2013). It also supports the development of Multilanguage verification environment with System Verilog, SystemC and e, and is able to interoperate between different tools from different vendors (Mentor & Cadence , 2007).

2.3.3 Universal Verification Methodology (UVM)

Accellera Systems Initiative, a standards organization that supports a mix of users and EDA vendors decided to create, support and advance system-level design, modelling, and verification standards for use by worldwide electronic industries. To tackle the verification standardization, Accellera uses OVM as baseline and includes key features from VMM methodology resulting in the now widely adopted Universal Verification Methodology (UVM).

The UVM builds upon SystemC and System Verilog Object-Oriented Programming (OOP). The UVM components use Transaction level communication (TLM) between object communications. UVM is based on a hierarchical testbench organization. The dynamically-generated objects allows the tests and testbench architecture to be specified without recompiling and the separation of the testbench stimulus or *Sequences* from the testbench structure. (Salemi, 2013). Universal Verification Component (UVCs) is a reusable verification IP developed based on the UVM Methodology (Aynsley, 2012) that also includes the interface protocol. This promotes the reusability of the modules and allows designers to work together independently. *Virtual Sequences* can be used to coordinate the *Sequences* or test cases across multiple modules. This further enhances reusability (Yun et al. , 2011). A robust AHB verification environment has been derived based on the methodology (Bhaumik & Jaydeep, 2013). Other works include a

reusable verification environment for ethernet (Sridevi & Dr. Krishnamurthy, 2013).

An independent study undertaken by Mentor Graphics under Wilson Research Group on the various methodologies that has been adopted in 2010 and 2012 revealed that the adoption of the UVM methodology since its introduction in 2010 was widespread. It indicated that half of the designs which were over 5 Million gates used UVM in 2012 (Foster, 2013). Figure 2-2 shows the overall evolution of the verification methodology.

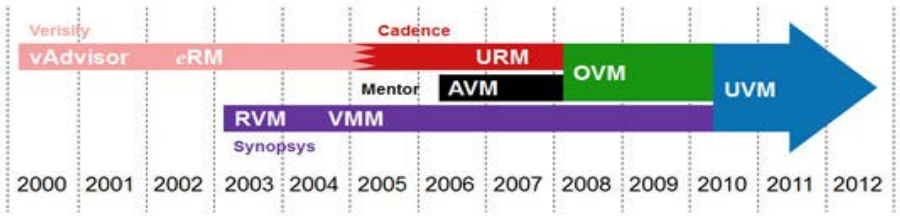


Figure 2-2: Verification Methodology Evolution

2.4 *NoC Architecture*

2.4.1 Overview

We used the ARM Cortex-M0 core in all of our works. This core only has the computation logic. The instructions and data needed by the core to perform the computations are stored in the external memory. ARM implements the AHB-Lite bus to interface the core to the memory. Peripheral devices for IO and special purpose processing that complement the core are abstracted as registers mapped into the memory space. In this regard, the Processing Element (PE) in our work consists of an ARM Cortex-M0, memory, and peripheral devices connected with the AHB-Lite bus.

The AHB-Lite bus is a single-master subset of the full AHB bus (ARM, 2006). As such, the core is the bus master to all the devices on the bus. In a true multi-core system, a PE should be able to initiate access on the interconnect regardless of the other PEs at any time. Clearly, the single-master AHB-Lite bus is not directly applicable as the interconnects between the PEs.

In the following sections, we first describe the architecture of the PE based on the Cortex-M0 and AHB-Lite bus. This is followed by the description of a multi-core system that interconnects the PEs using a traditional multi-layer matrix technique. This system is used as the reference system for benchmarking in a corresponding work in our group. As shown in one of our research projects, The Design and

Implementation of a Scalable Multi-Processor System-on-Chip Using Network Communication for Parallel Coarse-Grain Data Processing, the complexity in such system may grow to an unmanageable scale. Our research into scalable multi-core architecture has resulted in a ready-for-tapeout NoC base system. Hence, in the last section, this specific architecture, together with several other network topologies investigated in our works, are described. The subsequent chapters in this thesis shall build on the expositions here to demonstrate the design of a scalable, reconfigurable and reusable verification platform used in this work and The Design and Implementation of a Scalable Multi-Processor System-on-Chip Using Network Communication for Parallel Coarse-Grain Data Processing.

2.4.2 ARM Cortex-M0 based PE

AHB-Lite addresses the requirement for a high performance bus. It supports single AHB master and provides high bandwidth operation. The features of AHB-Lite for high performance and high clock frequency system includes burst transfers, single-clock edge operation, non-tristate implementation and wide data bus configurations. AHB-Lite Bus consist of a decoder and a multiplexer. The decoder is used to decode the address (HADDR) from the AHB Master to generate the slave select signal (HSEL). The multiplexer uses the HSEL signal to channel back the corresponding slave responses (ARM, 2006). These are shown in Figure 2-3. We refer the interested readers on the AHB-Lite bus to AMBA™ 3 APB Protocol v1.0 Specification (ARM, 2006). Figure 2-3 shows the single master AHB architecture.

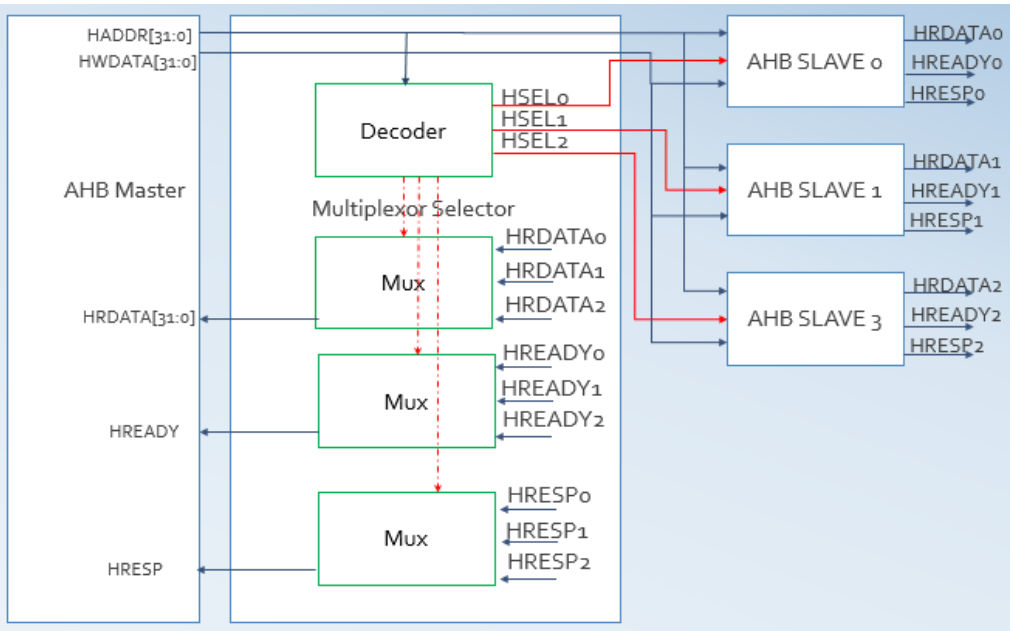


Figure 2-3: Single master AHB-Lite Bus System (ARM, 2006)

In our system, Cortex-M0 communicates with the flash controller, SRAM controller, Parallel port, GPIO, AHB2APB Bridge, Advanced Encryption Standard (AES) core and AHB2NOC adapter through the AHB-Lite Bus. Figure 2-4 shows our special core implementation. The purpose of this system is to control the I/O interfaces such as GPIO and parallel port. In AHB normal core system as shown in Figure 2-5, it consist of an AES core. This AHB system which will perform the AES cross-grain encryption. In both of the system, AHB2APB Bridge is used to add peripherals such as timer and SPI. The Advanced Peripheral Bus (APB) is designed for low bandwidth control accesses. This system is also used to read from the 32kB FLASH through the FLASH controller.

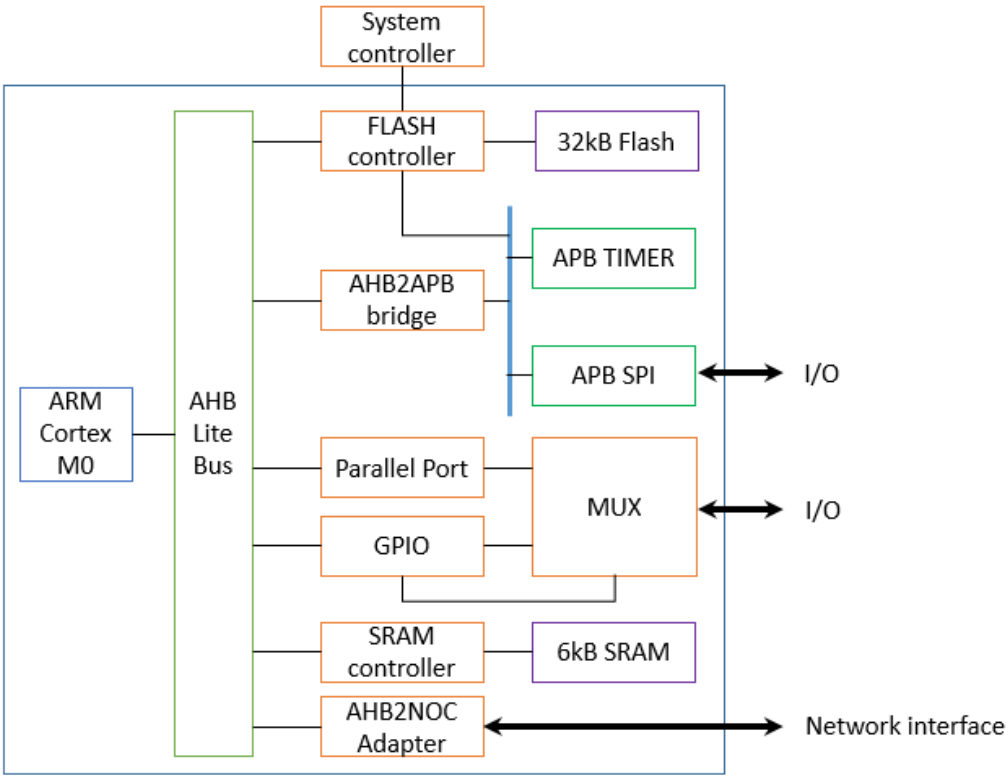


Figure 2-4: AHB Special core system

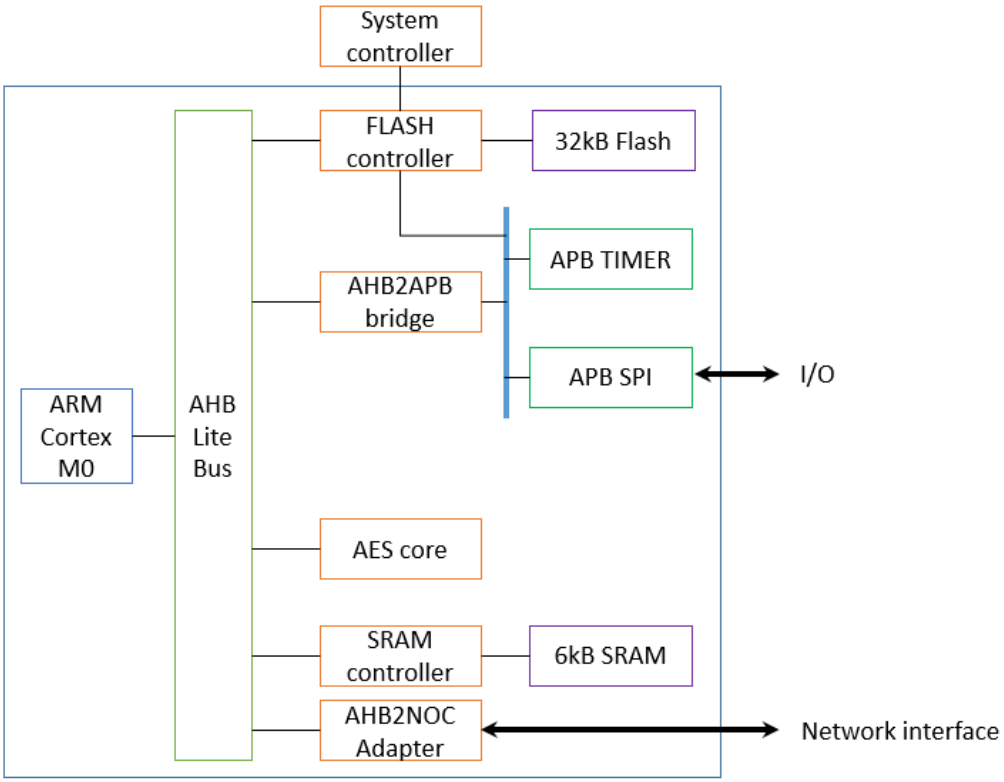


Figure 2-5: AHB Normal Core System

2.4.3 AHB-Lite Bus based Multicore System

The AHB-Lite is a single master bus interface. In AHB-Lite based multicore system, multilayer matrix layers has to be added to isolate the AHB masters from each other but allowed the slaves to be accessed by the processors.

The operation of the multi-layer matrix is best described with an example. In Figure 2.6, slave arbitration is performed by the multilayer matrix so that when AHB master 1 is accessing slave 1, AHB master 2 is not allowed to have access to that slave. Figure 2-6 illustrates AHB master 1 and master 2 each have access to slaves 1, 2, and 3. Slaves 4

and 5 is a local slave to AHB master 2 so only AHB master 2 have access to those slaves. In order for AHB master 1 and AHB master 2 or the processor cores to communicate, mailbox is one of the mechanism that can be used by having it as a common slave to the processor cores. This allows processor core 1 to write to the mailbox. Processor core 2 can then retrieve from the mailbox and vice versa.

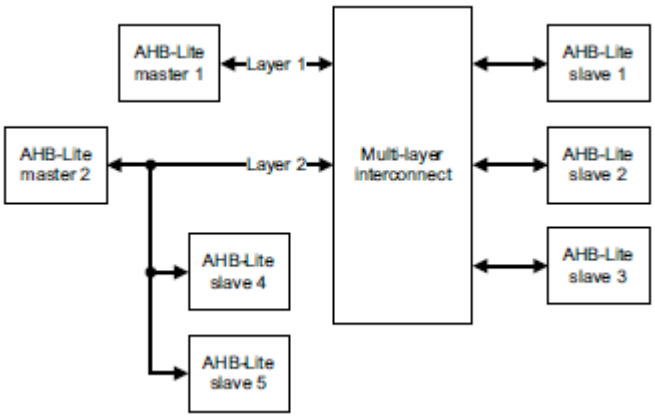


Figure 2-6: AHB Multicore system using Multiplayer Matrix

In another study in Design and Implementation of a Scalable Multi-Processor System-on-Chip Using Network Communication for Parallel Coarse-Grain Data Processing, a bus based AHB-Lite multicore system has been developed as a comparison for the network based multicore system. The implementation of the architecture is as shown in Figure 2-7.

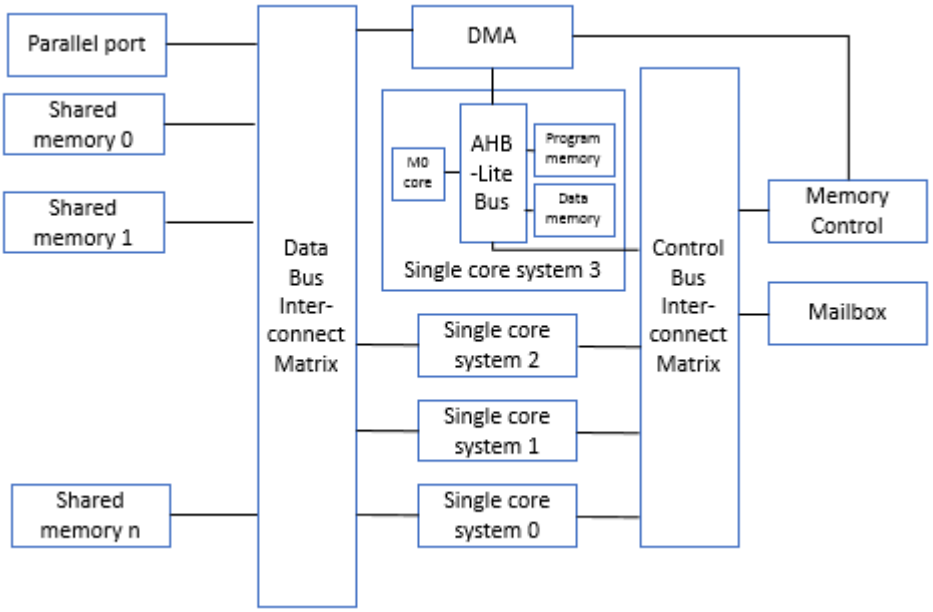


Figure 2-7: Bus based AHB-Lite multicore system implementation

Once the parallel port contains transfer, interrupt is triggered to single core system 3. Single core system 3 then requests the memory control through the DMA. The memory control establishes connection the Shared memory 0 and Parallel port by configuring the data bus interconnect matrix. After the connection is established, the transaction from the parallel port is transferred to the shared memory 0 through the DMA. When the transaction has finished, the DMA sends request to the memory controller to break the parallel port and shared memory 0 connections.

The single core system 3 sends message to the mailbox through the control bus interconnect matrix. Each mailbox buffer corresponds to a single core system. Let's say the message is for single core system 0, this message is directly passed to the system. The single core system 0

requests the memory controller to form the connection to the shared memory 0 to retrieves the content of shared memory 0 for processing. The processed transfer is sent back to the shared memory 0. This overrides the shared memory 0 content. Once the transfer has finished, the single core system 0 requests the memory controller to break the connection between the system and shared memory. A message is sent to the single core system 3 through the mailbox. To retrieve the processed data in shared memory 0, single core system 3 sends request to form the connection between the parallel port and the shared memory 0 though the memory controller. After the transfer has finished, the single core system 3 sends another request to the memory controller to break the connection.

The system level verification challenge is to develop a system level verification environment which is able to monitor the various different transfer types and the environment can also be configured with different monitors to monitor from various points based on the number and type of PE.

2.4.4 Network-based Multicore System

The increasing of the PE, controllers and peripherals in the Multi-Processor System-on-Chip (MPSoC) can lead to a very large and complex multilayer matrix design. This will make the overall design size to be significantly larger which will increase the die area and fabrication cost. A network can be used to replace the multilayer matrix.

2.4.4.1 Network Topologies

A network topology consists of links and nodes. A link is the communication path between 2 communicating nodes. The nodes are the endpoints of any branch in the network and are essentially the PEs. These nodes can have different arrangements and are linked to form a network. There are different topologies such as ring, star, line, and mesh. The challenge is to develop a verification platform that is able to adapt based on these different architecture needs.

The ring architecture shown in Figure 2-8. The data packet travels from one node to the next. When the data packet reaches a node, the node checks if the destination address predefined in the packet is the same as the node address. If it is the same, the data packet is meant for that node. If not, the data packet continues to travel until its designated node is reached. This topology can be easily extendable, but the network diameter increases linearly with the number of nodes. (Chen et al. , 2011)

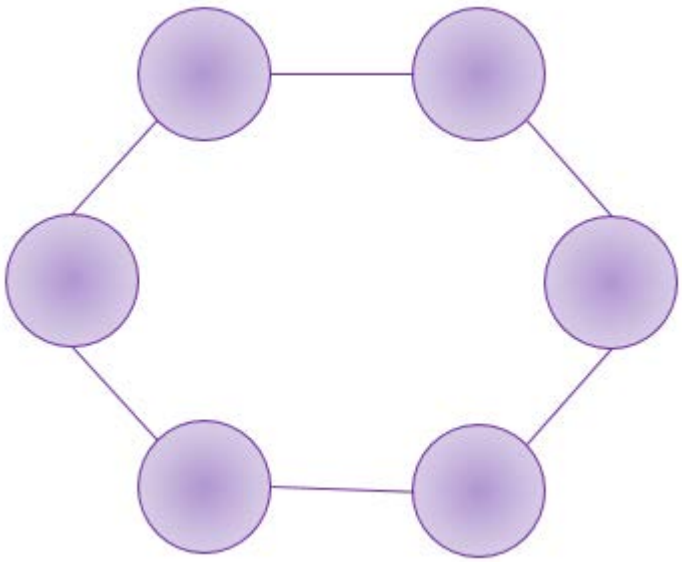


Figure 2-8: Ring Network Topology

In a star topology as illustrated in Figure 2-9, the nodes communicate across the network though the centre hub. The data packet is broadcast from the central hub to the corresponding nodes. This topology also allows additional nodes to be added with ease. But, if the central hub failed, the network will fail to operate.

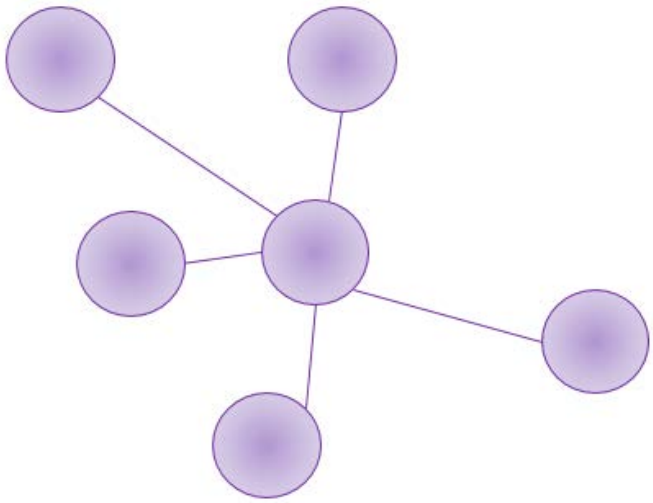


Figure 2-9: Star Network Topology

A network topology is considered mesh when more than 2 nodes are connected to each other. Figure 2-10 shows a partially connected mesh topology. Part of the nodes are not connected to each other. The nodes which are connected to a few other nodes would need a significantly large amount of I/O interface. This would reduce the scalability of the system.

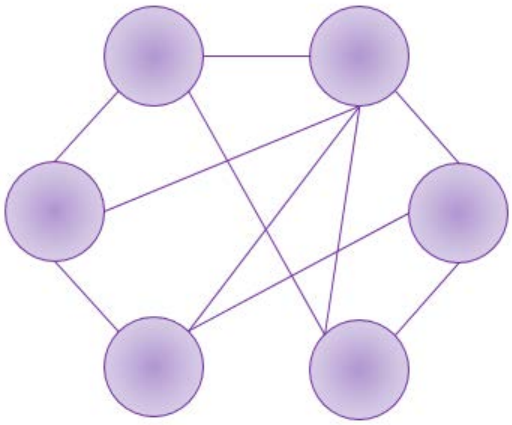


Figure 2-10: Partially Connected Mesh Network Topology

In a fully connected mesh shown in Figure 2-11, each node is connected to all the other available nodes in the network. Large number of I/O on the nodes are required as a result (Pandya, 2013). This reduces the network congestion by using the available alternative paths to the designated node directly.

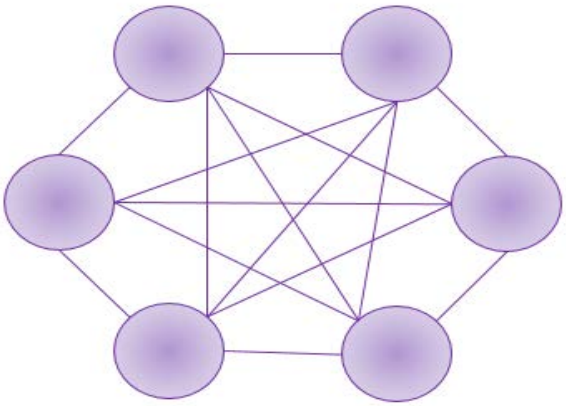


Figure 2-11: Fully Connected Mesh Network Topology

In a bus or line topology in Figure 2-12, all nodes are connected to one common link. This topology is commonly used as interconnect in SoC as well as MPSoC systems. An example of this interconnect would be an AHB-Lite Bus which has been described earlier. This topology grows linearly with the system which reduces the scalability of the system.

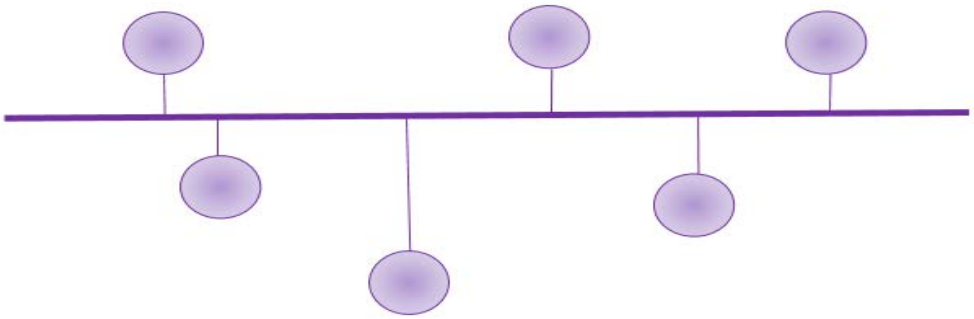


Figure 2-12: Line Network Topology

2.4.4.2 The UTAR NOC

The network topologies used in our systems are generated by utilizing the CONfigurable Network Creation Tool (CONNECT) (Michael Papamichael, 2012). The network uses credit-based flow control. This means that the network clients need to control the credits with the routers that is connected. In order to send the flit, the client enables the EN_putFlit and flit is loaded to pufFlit register to send to the network. The getCredits is set to the maximum selected Flit Buffer Depth during the network generation. Once the flit has been sent out into the network, the client decrements the credit's availability for the particular virtual channel to keep track of the remaining credits. The Send Flit protocol is shown in Figure 2-13.

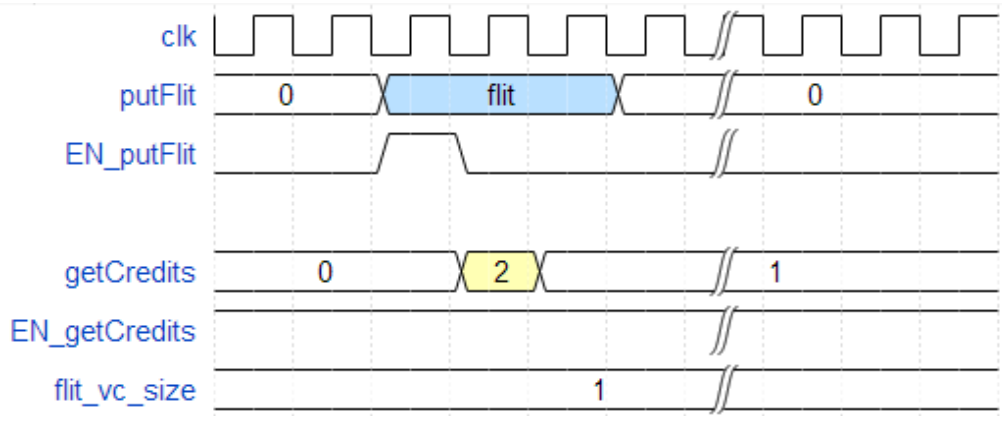


Figure 2-13: CONNECT Network Send Flit Protocol

For the client to retrieve the flit from the router, the client needs to maintain incoming flit buffers for each of the virtual channels to store the flit. These buffers are sized accordingly to the selected flit buffer depth. When the client de-queues a flit from its incoming flit buffers, a credit corresponding to the virtual channel is sent to the router. This is to inform the router about the availability of the new space. Figure 2-14 illustrates the receiving flit protocol.

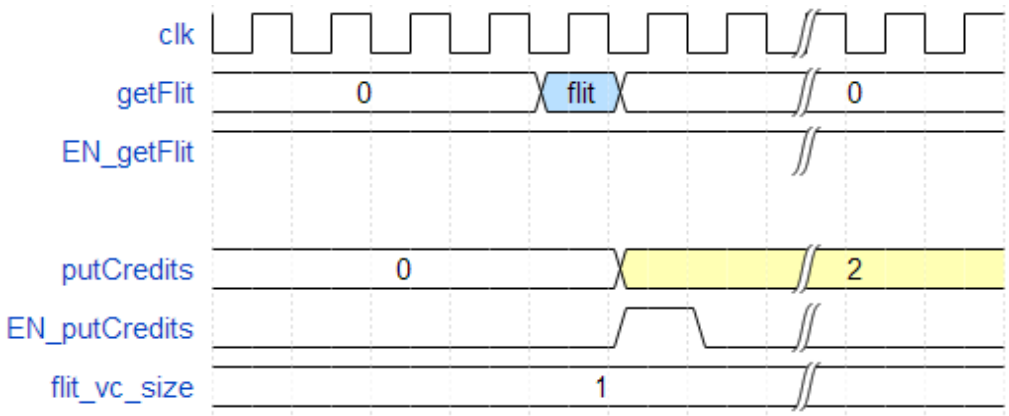


Figure 2-14: CONNECT Network Receive Flit Protocol

Figure 2-15 shows the fields of the routers' flit that defines the network protocol of the generated router.

valid bit	is_tail	destination	virtual channel	source	data
1-bit	1-bit	2-bit	1-bit	2-bit	32-bit

Figure 2-15: CONNECT NoC Flit

Valid bit indicates the validity of the flit to be sent. It is set to 1 to indicate that flit is valid. In order to identify the last flit in the transfer, Is_tail field is used. The destination field defines the transfer from one router to its designated destination. The width of this field depends on the number of endpoints in the network. Virtual channels are used to transmit and receive the flits are indicated in the virtual channel field and the width depends on the number of available virtual channels that is defined during the router generation. The data field contains the data for the transfer. Various data sizes can also be defined during the router generation. In our case, the data size is defined to be 32 bits incoherent with the AHB-Lite bus size. An additional source field is added to the CONNECT flit to add the address of the flit sender as illustrated in Figure 2-16. This field is used by the AHB2NOC adapter for buffering the data transfer.

valid bit	is_tail	destination	virtual channel	source	data
1-bit	1-bit	2-bit	1-bit	2-bit	32-bit

Figure 2-16: Modified CONNECT NoC Flit

The AHB2NOC adapter that we have developed in Figure 2-17 allows the AHB system to communicate with the network routers. For the processor to retrieve the flit transfer, the adapter converts the flit to AHB transfer. This can be done through polling the buffer empty status register flag or wait for the interrupt when the flit is received. The adapter wraps the AHB transfer into packet flits and send them to the network router.

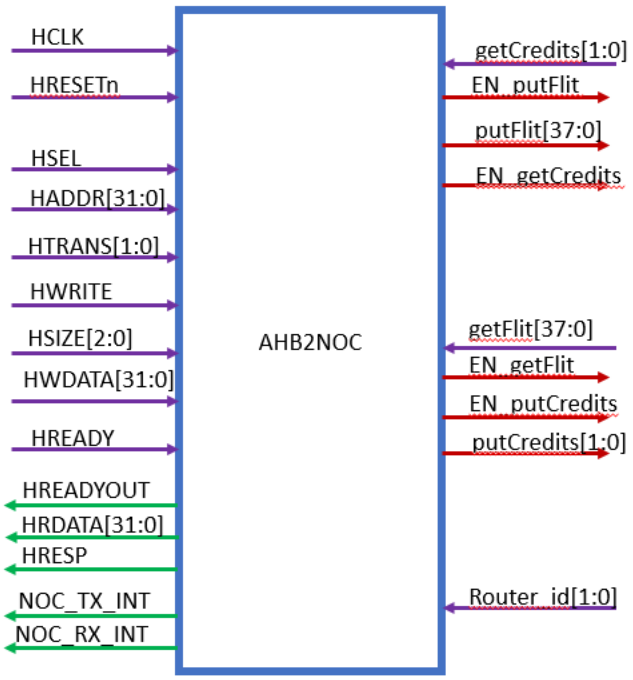


Figure 2-17: AHB2NOC Block

One node of the router is connected to the special core. This core has access to the NoCs' I/Os through GPIO or parallel port, and SPI to allow the system to capture the data and to send the processed data out to another system. The other 3 nodes are connected to 3 normal cores. These cores will perform the AES encryption. The overall system is shown in Figure 2-18.

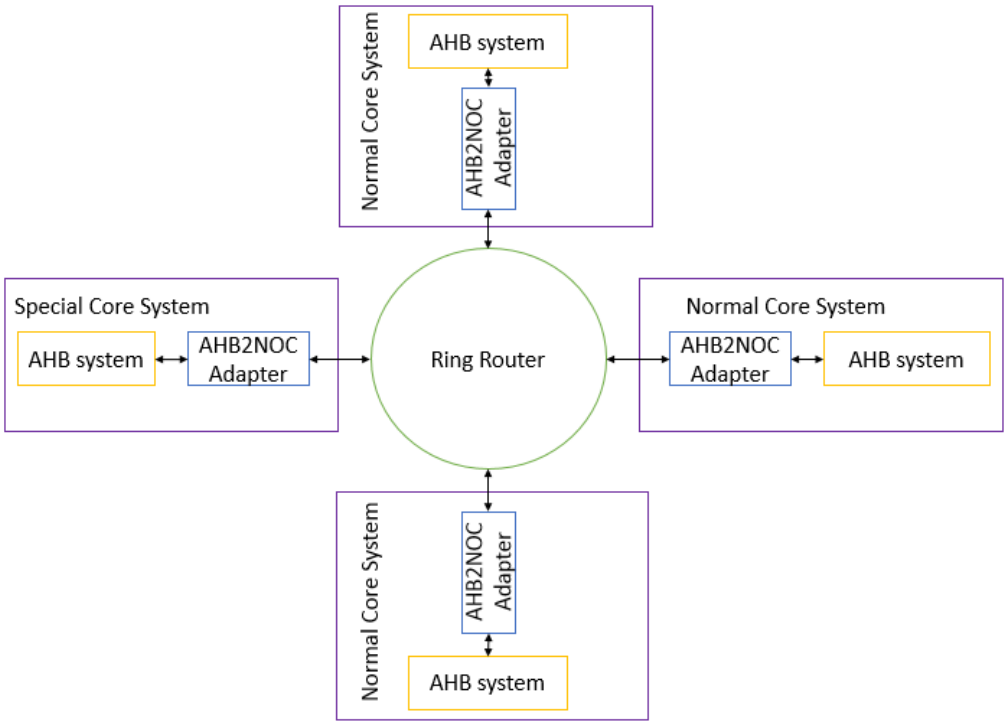


Figure 2-18: NoC Architecture

The AHB Systems are connected to the ring network through the AHB2NOC adapter in the normal core and special core modules. Similarly, the ring router can be replaced by other network topologies.

2.5 *Summary*

This chapter has discussed on the ESL Design flow using the TLM modelling. The discussion followed by various verification languages that has been developed. It should be notable that System Verilog and SystemC has evolved from the traditional verification languages to cope with the new verification challenges. System Verilog has gained much interest among EDA vendors. This has led to the development of various verification methodologies for System Verilog such as AVM, URM, OVM and UVM. Among them, UVM is the standard verification methodology that has agreed upon among the EDA community. The final part of this chapter discussed on the NoC architecture of our UTAR NoC which utilizes the CONNECT generator to generate the network router. The following chapter will discuss on the methodologies involved in ASIC design in particular the UVM methodology that is adhered to develop our verification environment.

CHAPTER 3

ASIC DESIGN AND VERIFICATION METHODOLOGIES

3.1 ***ASIC Design Methodology***

ASIC Design is a very costly process, especially fabricating wafers in submicron technology. A set of the photolithographic masks is extremely expensive and the lead-time for the fabrication process is long thereby influencing a product's time-to-market. Therefore, absolute discipline in the design process is critical and the adherence to a proven methodology is necessary to ensure success in the first attempt. Tools to assist the design process further enforce the methodology. Usually major EDA vendors offer integrated tools for every phase of the design flow. However, sometimes it may be necessary to integrate tools from different EDA vendors in areas where they are better supported.

Since failure in ASIC design is not an option, thorough design verification would feature predominantly in any design methodology. Coupled with the inherent defects in silicon wafers extremely high levels of fault coverage are demanded.

The following subsections of this chapter will discuss the ASIC design methodology and the Universal Verification Methodology (UVM).

3.1.1 Front End Design

3.1.1.1 Specifications

A chip design starts with defining the system specifications that describe the functional features, electrical and speed performance, and technology options. This involves the evaluation of the various fabrication technology nodes that are able to accommodate the performance requirements along with the cost trade-off that are involved. The overall project budget should also be taken into consideration.

3.1.1.2 Electronic System Level Designs

The electronic system should be described at the behavioural level. At this level, there are several ways to model its behaviour. Transaction Level Modelling (TLM) models are one of the methods that can serve this purpose. In TLM, the model description is in terms of transactions that are transferred over the channel. The TLM concepts and interfaces shall be discussed in Section 3.2.1. TLM allows the implementation of the design to be optimized as early as possible in the designing stage. Various innovative implementation methods can then be effectively explored to differentiate the performance and time-to-market of the finished product from the competitors.

The system architecture can be defined by mapping the system requirements to hardware and software components in such a way as to

meet design objectives. From TLM, conventionally, designers manually translate the model into Register Transfer Level (RTL) models, which are prone to errors. However, with the High Level Synthesis (HLS) tools, TLM models could ease the design flow by being able to synthesize the design intent to the lower Register Transfer level. (Brown, 2009).

3.1.1.3 Register Transfer Level (RTL) Design

The designs are usually described using Hardware Description Language such as Verilog or VHDL at Register Transfer Level (RTL). RTL design involves describing the design behaviour as transfer is occurring between registers at every clock cycle. This introduces the concept of timing and allows the speed performance of the chip to be defined.

3.1.1.4 Verification

In ASIC Design, the verification process is a discipline in itself and will take up a major share of the time and labour resources of an entire design cycle. It is a common misconception that an ASIC design that has been verified functionally with an FPGA is ready for tape out. This is not the case because an FPGA is a finished product at an already determined process point, guaranteed to work within the specified supply voltage and temperature range. In the case of an ASIC design, this is the stage where this guarantee is ensured by performing thorough design verification within all corners of process, temperature and supply

voltage variations. The functions and design intents defined in the specification must work within these environmental boundaries.

The verification plan contains descriptions of the functional features that need to be exercised and the techniques in developing the test cases especially for all boundary cases. This verification plan will serve as an overall scoreboard for functional coverage.

A verification environment models the actual system in which the Design Under Test (DUT) works. This environment consists of various models to generate the input stimulus and check for the responses. These models depend on the interfaces that the DUT is connected and the DUT's functionality. The Verification process is done each time when changes are made to the design and when the design is synthesized from one abstraction level to another.

Since ASIC Design verification is a highly discipline process, a standard methodology is strictly adhered to. In our case, the Universal Verification Methodology (UVM) is used in developing our reusable, reconfigurable and scalable verification environment. The details of the methodology will be discussed in Section 3.2 while the implementation of the verification platform will be reviewed in Chapter 4 of this dissertation.

3.1.1.5 Design For Test (DFT)

Manufacturing defects and crystalline imperfections in silicon wafers will manifest as faults at random locations in a fabricated wafer. These faults will cause the chip not function according to its design intent. Therefore, these faults must be detected at test. These special tests, or Design for Test (DFT) features, must be built in to achieve near one hundred percent fault coverage.

Achieving full functional, line, code and toggle coverage in verification will give a high degree of fault coverage, but will almost certainly be insufficient due to the limited controllability and observability for the finite number of input and output pins of the chip. It is therefore necessary to implement structured testability features. These are special DFT structures designed into the chip with the goal of detecting every fault.

The most favoured structured DFT features adopted are Built-In-Self-Test (BIST) for regular structures like memories and Scan Chaining for logic. For memories, it is common for BIST hardware design to include Built-In-Self-Repair (BISR) where redundant memory rows or banks can be swapped for bad ones. Structures can also be implemented in hardware to support the structured fault simulation.

Although Scan Chaining of internal logic and BIST will allow a very high degree of fault coverage, the chip input and output (I/O)

buffers and associated functions are not easily covered. The JTAG (IEEE 1149.1) Boundary Scans implementation takes care of this. JTAG Boundary Scan together with BIST will allow fully automated testing of a number of ASICs in a system board.

3.1.1.6 Synthesis

The RTL codes are synthesized into a gate level netlist according to the imposed timing constraints using synthesis tools. Since the logic gate, designs are specific to the manufacturer and the technology the target cell library must be provided for design synthesis. The gate level netlist can be optimized for area, speed performance and testability through the design constraints.

3.1.1.7 Static Timing Analysis (STA)

Static timing analysis is performed on fully synchronous designs to validate the timing performance of the ASIC. The speed or timing constraints of the design must be provided as input to the synthesis tool with considered timing margins to allow for physical design variations.

A good wire load model from the physical chip floor plan is also beneficial during synthesis and STA in order to achieve early timing closure after placement; routing and parasitic extraction are back annotated into the netlist. STA checks all possible paths for timing violations under worst-case scenarios. This process is performed on the

gate level netlist. Timing closure with a well-accepted STA tool is very often sufficient to sign off for tape out.

The synthesized gate level netlist also needs to be ensured that the RTL design intent is not modified by performing the equivalence checking. The process from defining the specification to design synthesis is considered the front-end design flow in designing SoC. The next section will discuss on using this generated gate level netlist and transform it into the layout of the chip.

3.1.2 Back End / Physical Design

3.1.2.1 Floorplan

The physical implementation of the ASIC can begin with a gate level netlist, which has sufficient timing margins. Macro cells like SRAM, and other pre-designed Intellectual Properties (IP) blocks, and IO locations are placed to minimize the die area and maximize routability so that the overall chip cost can be reduced.

Core utilization is set to the appropriate value to give margins for routability taking into consideration the extra space requirements for power networks and clock tree routing during the first attempt. It usually takes a few iterations to get a better core utilization value. It is important to weight the trade-off between the chip size and routability of the chip. Setting the core utilization value too high may result in either an un-

routable chip or timing inability to close timing for critical paths due to long or roundabout routing. Any changes in the chip size to accommodate more routing resources would mean re-starting the floor-planning process due. This increases the use of valuable resources and time-to-market. If the core utilization value is set too low, the chip size grows, causing the overall cost of the chip will increase.

3.1.2.2 Power Network Synthesis

There must be sufficient power and ground supply pins to anchor the chip firmly to its operating voltage, prevent power, and ground noise from interfering. Placement constraint script is used to place these power pads strategically on each side of the chip, especially where there are many simultaneously switching output drivers. Ground bounce simulation can be done to determine that sufficient pairs of power pads have been added.

If there are more than one power domain in the design the power distribution within these power domains features significantly in the floor planning and placement phase. The power distribution networks for the chip are synthesized based on the power budget specifications for each domain. The width of the power buses must be sufficient to prevent a drop in supply voltage below the designed range. This IR drop is analysed and displayed by the tool. Adjustments can be made to the floorplan and power networks until the IR drop requirement is met.

3.1.2.3 Clock and Buffer Tree Synthesis

Clock tree will be synthesized to be as close to an ideal clock as possible as this is set at design synthesis for a fully synchronous design. The Clock Tree synthesis tool will buffer the clock to sufficiently fast rise and fall edges and at the same time try to balance the clock branches so that the clock skew is controlled when it arrives at all the flip-flops to ensure no register data shoot through situations. Skew control is not as critical in the buffer tree generation.

3.1.2.4 Place and Route

The rest of the standard cells fill up the core area. Where they are placed is determined by the routing weight of the cells. Before these cells are routed, it is prudent to examine the congestion map, which will indicate whether there are sufficient routing resources available. Placement optimization can be done to improve routability. The tool will try to find all possible routes for the signals, irrespective of whether design rules are violated or not. Many iterations of the search and repair process are then performed to fix design rules as well as timing violations. If these violations cannot be fixed after rounds of placement optimization and re-routing it may be necessary to increase the chip size. Parasitic RC extraction is done using a high precision 3-D extraction tool. A delay-timing calculator is used to convert these RC delay values to the Standard Delay Format (SDF) to be back annotated to the post layout netlist for precise timing simulation and STA.

3.1.2.5 Chip Finishing

Additional steps are taken to prepare the chip for tape out. The length of the routed interconnects may be too long in the same metal layer. This will cause damage to the devices it connects to due to the collected charges on the metal layer during the fabrication. Antenna rules, which are provided by the foundry, are used by the tool during the search and repair process. Antenna fixing is done automatically by the tool by breaking these long same level interconnects using the next higher-level metal. For the top-level metal, which do not have a higher metal layer to split the wires, protection diodes, can be added to the device for protection.

The remaining unused areas in the standard cell rows filler cells are added to establish continuity of the N- well, the P-well and the power buses. Extra metals and polys fills are added to meet the density requirements by the foundry for manufacturability.

Empty spaces may be left after the corresponding IO cells are added. IO fillers are added to fill up these areas to complete the IO power ring of the chip, which also provides some ESD protection.

Redundant vias are inserted wherever there are spaces to avoid a possible manufacturing issue where some vias become too narrow and may not form proper connections.

3.1.2.6 Post layout verification

The post layout netlist is extracted from the GDSII physical design database. LVS is the process where the equivalence of the physical layout is checked against the pre-layout gate level netlist to ensure that logical nothing is changed during Clock Tree Synthesis, buffering and optimization in the backend process. Design Rule Check (DRC) is performed to ensure geometrical design rules determined by the technology and manufacturability considerations are not violated.

Minor changes or corrections can still be done to the layout through the Engineering Change Order (ECO) processes provide for by the layout tool. DRC and LVS are done again each time the physical is altered.

3.1.2.7 Tape out

The signoff STA reports and timing simulations are reviewed to ensure that the design is ready for tape-out. The foundry will review that design have met the manufacturability guidelines provided by them before proceeding to order the masks. Figure 3-1 shows the overall ASIC Methodology.

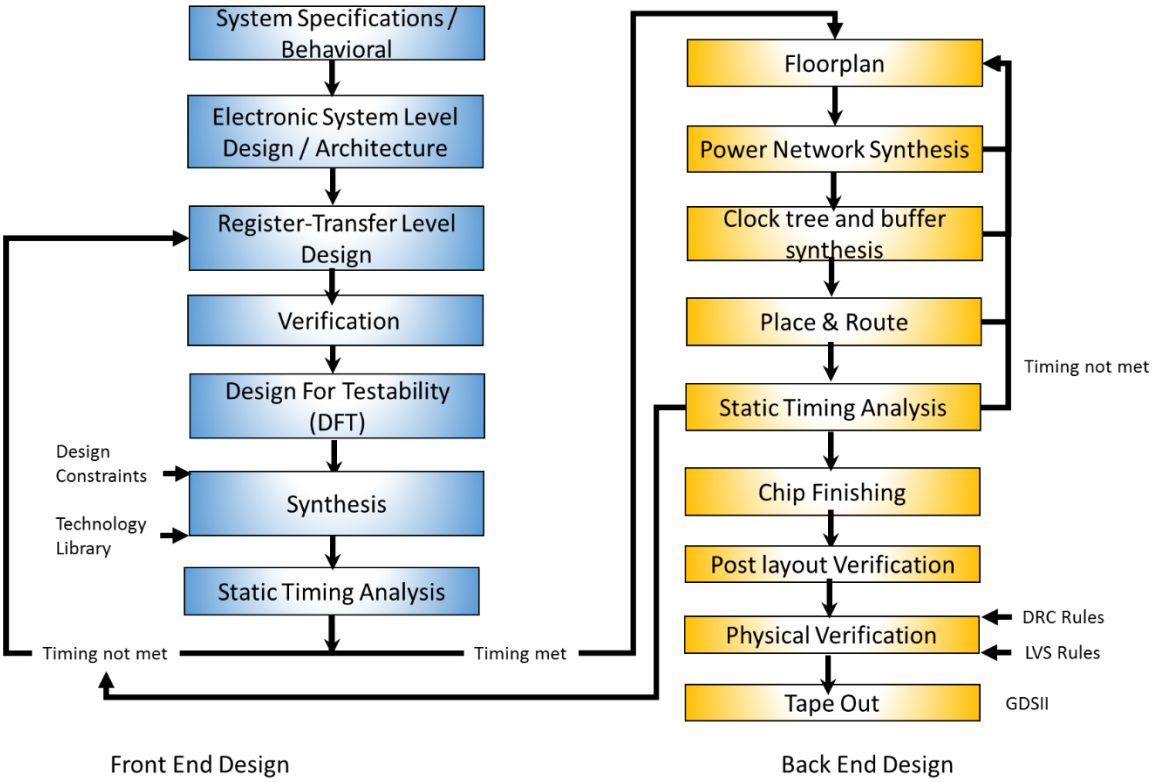


Figure 3-1: ASIC Design Methodology

3.2 *Universal Verification Methodology (UVM)*

The increasing complexity of ASIC design verification has called for a standard methodology to be strictly adhered to. UVM is the current industrial standard verification methodology (Glasser, 2011).

UVM is based on the TLM and OOP (Object Oriented Programming) concepts. It provides a set of TLM communication interfaces and channels to allow the OOP classes to communicate at transaction level. The TLM and OOP concepts will be introduced in next two sub-sections to allow a clearer understanding of the UVM discussion.

3.2.1 Transaction Level Modelling (TLM) Concepts

The TLM is a transaction based modelling used for developing abstract models of components and systems. The transaction is an OOP class object that includes the necessary information such as variables and constraints to model the communication between two components. The amount of information that is encapsulated within the transaction indicates an abstraction level of the model. The basic transaction could be extended to include additional properties and constraints. For example, to fully specify the bus transaction, including Objects such as transfer delays or latency will better model the bus operation.

3.2.1.1 TLM Communication

TLM components initiates transactions by 3 basic communication mechanism – put, get and broadcast. *put* is when a producer component puts information to a consumer component. *get* is when a consumer gets information from the producer. A third mechanism allows a producer to broadcast information and consumers may subscribe to these broadcasted information (Glasser, 2009).

The TLM communication interface is defined by the *port* and *export* pair. A TLM *port* specifies the methods to be used for a particular connection. These methods are implemented by the TLM *export*, so that it can be executed when the *port* initiates the transaction. TLM interface allows each of the components to be isolated from each other. The

components at the higher level can instantiate its sub-components and connects them together independent of any knowledge of the implementation. Accessing the sub-components can be achieved by making its interfaces visible at its higher-level component allowing the whole component to be visible as a single component with a set of interfaces regardless of its internal implementations.

3.2.1.2 Analysis Communications

The TLM *put* and *get* ports would require at least one export connected. In the case of passive components such as *Monitors*, which collect transactions and broadcast them to the other components, a third port, the *Analysis Port*, which may be left either unconnected or connected to any number of components, is needed. The components that subscribe to the broadcast are connected to the *Analysis Port* via their respective *Analysis Exports*.

The next subsection will introduce the concepts of the object oriented verification environment, which makes it configurable, scalable, and reusable across projects (Sharon, and Kathleen, 2013).

3.2.2 Object Oriented Verification Environment

The verification environment is used to fully exercise the DUT by modelling devices the DUT connects to. A Hardware Description Language (HDL) can be used to model these devices. However, it will be a challenge to model the verification environment of complex designs as in the case of most SoC with HDL and its limited features. For example complex predictors or checkers need to be implemented to allow self checking especially during regression testing.

Extensive input stimulus needs to be designed to fully exercise the DUT for complete functional coverage. Manually defining each of the stimulus scenarios would take take an extraordinary amount of time and effort and some cases might still be missing. Here constraint random stimulus generators are needed to generate the stimulus within the defined constraints for valid, invalid as well as abnormal transfers.

HDL, without additional aids to facilitate complex modelling requirements, will be unable to deliver the complete functional verification coverage and the desirable features of configurability, scalability and reusability. The system Verilog language has evolved from Verilog HDL and incorporates OOP features to overcome this (IEEE, 2009).

In Object Oriented Language, class contains data elements, and methods which can be instantiated. An object refers to an instance of a

class. These objects can be created and destroyed dynamically, allowing the verification environment to be configured. Each of these classes performs a specific task.

The *Scenario Generator* generates the stimulus based on constraints or rules provided by the verification engineer to model the behavior of the test cases. The DUT uses virtual interfaces to form the connections between the DUT and the verification environment. Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same *Driver* to operate on different portions of the DUT and to dynamically control the set of stimulus associated with the *Driver*. Changes to the underlying design under test do not require the code using virtual interfaces to be rewritten. The other classes that need to communicate with the DUT using the same communication protocol can then reuse the same virtual interface. These virtual interfaces can be passed to the different classes to make the connections between the DUT and verification environment.

The *Driver* class interprets and drives the *Transactions* to the virtual interface. The *Monitor* class is used to monitor the response from the DUT through the virtual interface and extract transaction information for comparison by the Checker. Figure 3-2 illustrates the various classes in an OOP based verification environment.

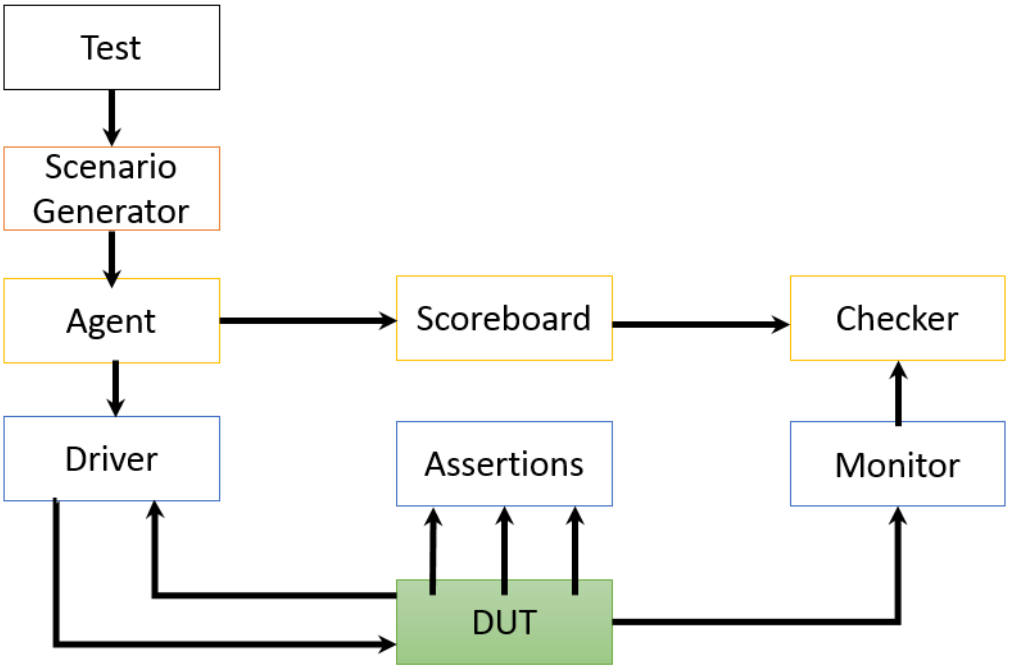


Figure 3-2: Layered OOP Verification Environment

The communication between these components can be simplified by encapsulating the transfer using objects or as transactions instead of passing multiple signal Objects. Data analysis and manipulation can then be done easily by accessing the corresponding objects. When a similar type of transfer is needed, the same transaction class can be reused. As examples, the types of transfer classes that has been developed for the use of our verification environments are AHB, APB, NoC, and SPI.

Inheritance is another concept in OOP to facilitate reusability. It allows different behaviours for the same method in the derived class to override the behaviour in the base class. This can be done through virtual functions. Virtual functions are used to support polymorphism where

multiple classes with different behaviours can be used interchangeably. As an example of the use of inheritance, the lower level layers such as protocol layer can be modelled using the base class. More complex models can utilize these protocol models. The base classes can be reused directly or used to derive other classes thus increasing the verification productivity.

We have introduced the importance of using object oriented concepts in the verification environment development. This concept has been adopted in recent Verification Methodologies such as Verification Methodology Manual (VMM) (Janick et al. , 2005), Open Verification Methodology (OVM) (Glasser, 2009) and Universal Verification Methodology (UVM) (Sharon, and Kathleen, 2013). The next section shall discuss on the generic verification environment in the Universal Verification Methodology (UVM) which has adopted the various OOP classes.

3.2.3 Generic UVM Verification Environment

The stimuli are abstracted as *Transactions*. They are generated in the *Sequence* and randomized according to specific rules and constraints. The *Sequence* sends the stimulus packets to the *Sequencer*. The *Sequencer* coordinates these *transactions* from multiple *Sequences*. When the *Driver* is ready, it grabs the *Transaction* from the *Sequencer's* TLM *get_port*. The *Driver* converts the *Transactions* into timed input stimulus to drive the DUT.

In order to verify the correctness of the DUT operation, we need to sample the stimulus, analyse the responses and keep score of the coverage. The *Monitor* acts as a service provider. It packages the sampled data and sends to its service subscribers such as the *Scoreboard* and Coverage Analysis Objects through the TLM *analysis_port*. The *Scoreboard* then checks for the correctness of the received transfer.

The *Monitor*, *Driver, Configuration Objects* and *Sequencer* can be grouped as *Agent* so that they can be reused as a basic block or a Universal Verification Component (UVC). These *Agents* as well as in some cases the *Configuration Objects* will be grouped to form the *Environment*. This forms another UVC. In this way, the *Environment* can also be reused and re-configured based on the requirement of a design. It allows multiple *Agents* to be instantiated. The virtual interfaces that connect to the DUTs can be passed to each of the *Agents*

and its sub components through the *Environment* allowing each of the sub-environments to be instantiated to verify various modules in the system. This module level environment can be built hierarchically to form a system testbench.

The basic verification environment using UVM is shown in Figure 3-3.

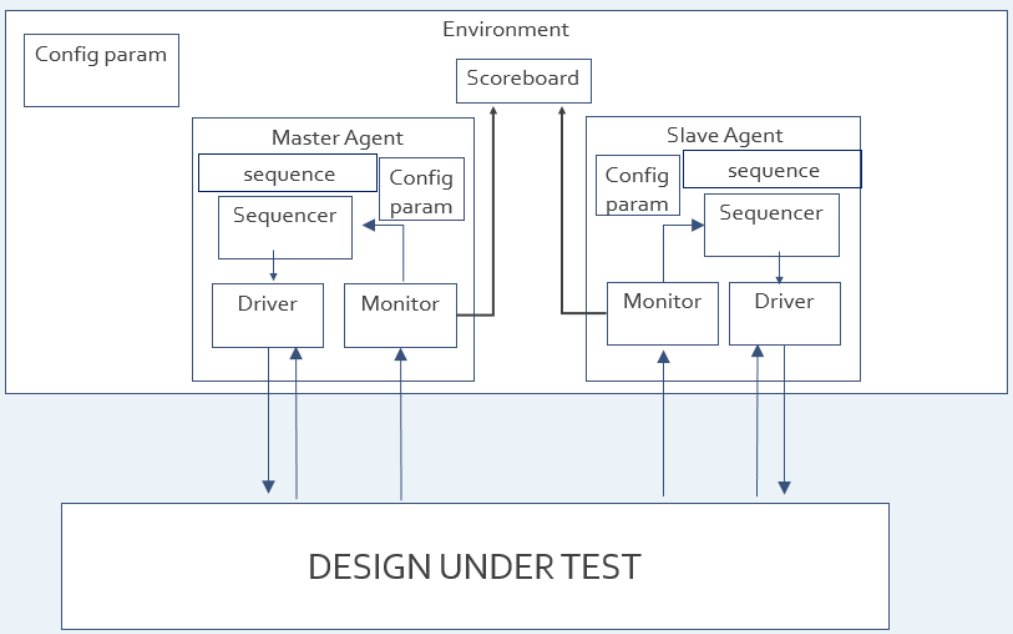


Figure 3-3: Generic UVM Verification Environment Architecture

This discussion shall continue with the discussion of the methods to build test cases or scenarios for the DUT.

3.2.4 Scenario Generation

In UVM, *Sequences* are used to model test scenarios. These *Sequences* can be layered to model scenarios that are more complex. (Janick et al. , 2013).

The basic *Sequence* or the lowest level *Sequence* models the protocol communication layer and the read write access. These *Sequences* are generic and often implement as an Application Specific Interface (API) layer for the *Driver*. For instance, these have been developed as our UVC for the AHB, APB, GPIO, and SPI protocols.

Higher level *Sequences* to model more complex scenarios can utilize these basic *Sequences* as shown in Figure 3-4. These layered *Sequences* are used to model components such as AHB memory which utilises the basic AHB *Sequences*.

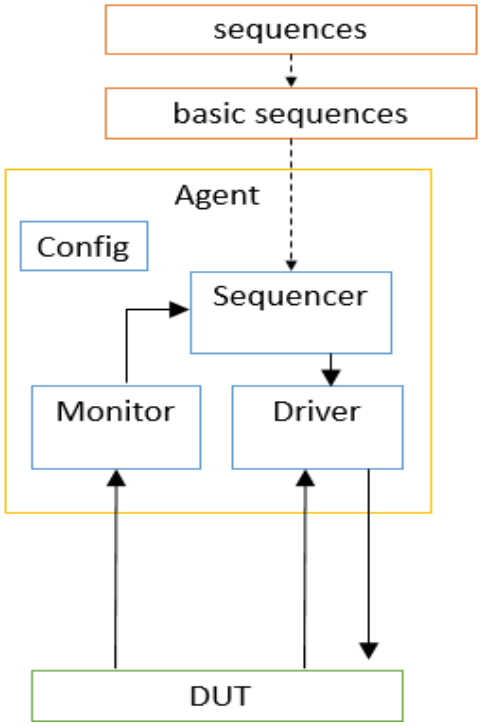


Figure 3-4: *Sequence* Layering

3.2.5 Factory concept in UVM

The UVM *Factory* is an advanced implementation of the *Factory* design pattern in OOP software, which provides features to allow the subtype of the object to be decided during run time. All the components are registered with this *Factory*. When the *Factory* creates this object, it will search for the instance or type override. The components, *Transactions* and *Sequences* can be overridden from the higher-level components. If none exists, it will create the existing instance and type. For a complete exposition, refer to Cummings, 2012.

As an illustration, we want to send the bad frame UART stimulus instead of the good frame stimulus that has been implemented in the *Driver*. The same verification environment can still be used as long as

the good frame UART *Driver* component is replaced by the bad frame components. Figure 3-5 shows a bad frame UART *Monitor* and *Driver* replacing the existing *Driver* through the *Factory* overriding method. The method has been used in developing our verification components. This will allow custom components to override the existing components in our current verification environment.

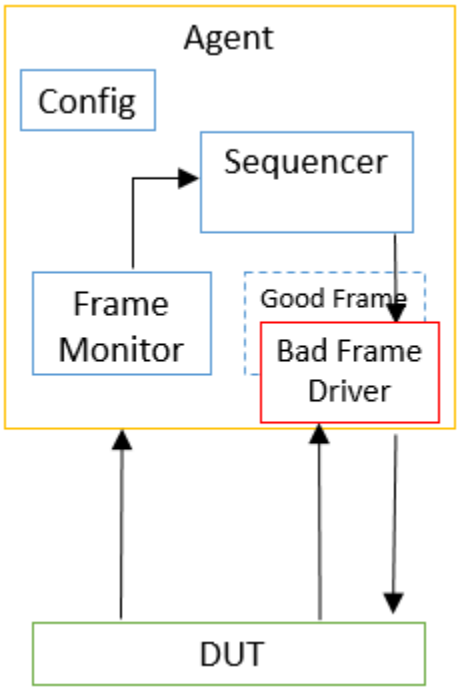


Figure 3-5: Factory overriding

3.2.6 Configuration Objects

Since the verification teams need to be able to run thousands of tests on a design, it would be ideal to compile the whole verification environment once and then run thousands of tests using the same environment. UVM allows the implementation of this dynamically reconfigurable verification environment.

The *Configuration Objects* serve this purpose. It is a mechanism in UVM to allow lower level component variables to be configured by its higher-level component. Objects such as strings, integers, objects and virtual interfaces can be defined in a common class. This class, which contains the Objects, is passed from the highest hierarchy of the verification to its sub components. These can include the number of *Agents*, the starting and ending addresses of the slaves and the type of components.

As an example, the number of *Agents* can be instantiated based on these Objects. This allows the verification environment structure to be maintained and reused while some of its components are replaced.

The next section shall discuss on how the UVM simulation is executed.

3.2.7 UVM Simulation Phases

In order to have a consistent verification environment execution, UVM uses phases. Using the predefined phases allows verification components to be developed in isolation but still interoperable. This is because there is a common understanding of the events that should happen in each phase.

The simulation runs in phases. It starts by building the UVM environment root component, which is the DUT in its test environment. The UVM phasing is then instantiated. The build phase constructs the testbench and the various child components and configures them. The components are constructed from top to bottom of the testbench hierarchy by using the Objects from the higher-level components in the hierarchy to properly construct the lower level components.

All these components are connected during the connect phase. This phase makes the TLM connections between components and works from the bottom of the hierarchy upwards. At the *end of elaboration* phase, final configuration, topology, connection and other integrity checks are performed.

The start of the simulation phase is used to display banners, verification environment topology and configuration information. This phase occurs right before the time consuming part of the simulation begins.

The run phase is where the main body of the test executes. This phase is implemented as a task and consumes simulation time. Each of the UVM components is in the run phase at the same time and the tasks are executed in parallel.

The extract phase is used to retrieve and process information captured by the scoreboards and functional coverage monitors through the analysis components after the simulation time no longer advanced. It computes coverage statistics and summaries. The results of the simulation are displayed or written to a file.

The check phase is used to validate the transaction that has been collected in the extract phase and determine the overall simulation outcome. It is used to check that no unaccounted-for data remains.

The report phase reports the result of the test. It may also be used to write to a file. In Figure 3-6, the basic phases in UVM are illustrated.

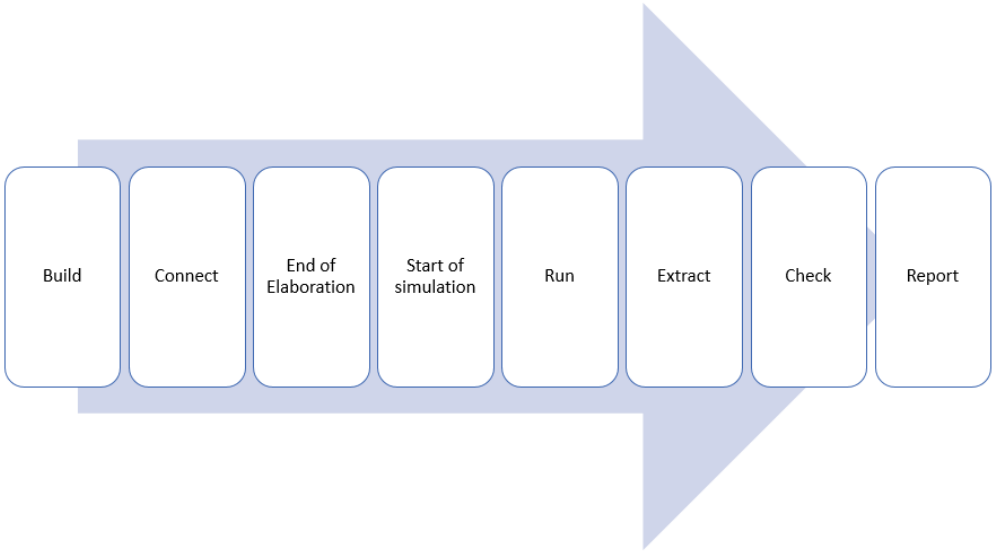


Figure 3-6: UVM Phases

3.3 *Summary*

This chapter emphasized the importance of methodologies in ASIC design and verification. Verification, being a very significant part of the ASIC design effort, calls for an absolute discipline and methodology of its own. Both the ASIC design methodology and the Universal Verification Methodology have been introduced. The advantages of using an OOP based verification environment to facilitate the creation of reusable and scalable verification components were discussed. The discussion continues by introducing the generic UVM based verification environment. The reusability and re-configurability is further enhanced by using *Factory* methods, TLM, and *Configuration Objects* in UVM. In Chapter 4 the objective of this work, which is the implementation of a verification platform using UVM for Net-on-Chip (NoC) verification, is presented.

CHAPTER 4

VERIFICATION PLATFORM ARCHITECTURE

4.1 *Overview*

The hardware design of a Multiprocessor System-on-Chip (MPSoC) using an on chip network (NoC) has been described in Chapter 2 Section 2.4. The NoC design can be implemented using various architectures such as mesh, ring and line which can be easily scaled. The objective of this work is to build a verification platform which is reconfigurable, scalable and reusable to accommodate these various architectures and design scalability. This Chapter presents the implementation of this verification platform using UVM. This platform will be built hierarchically. The component level verification environment will be discussed next.

4.2 *Component level Testbench*

The component level testbench is the most primitive verification environment. In UVM, it is one of the Universal Verification Components (UVCs). This environment consists of basic *Sequences* that model the communication protocols such as the AHB, APB, SPI, GPIO, network and parallel port. The basic *Sequences* can then be extended to model other complex scenarios such as memory read write model using the specific communication protocols. This enhances the reusability of the

basic *Sequences*. The component level testbench is used to verify each component individually in our system.

4.2.1 NoC Environment

The NoC environment is used to verify the generated CONNECT routers that have been described in Chapter 2 Section 2.4. To begin with, here is a recap on the protocol to send and receive a flit. In order to send a flit, the enable *putflit*, ***EN_putFlit*** has to the asserted. The credit availability is decremented by the client once the flit has been sent into the network. To retrieve the flit from the network, incoming flit buffers are used to maintain the flit for each of the virtual channels. The enable put credit signal, ***EN_putCredits*** is asserted when a flit is received by the client.

Figure 4-1 shows the overall NoC verification environment. This environment consists of a pair of *Send* and *Receive Agent* connected to the router.

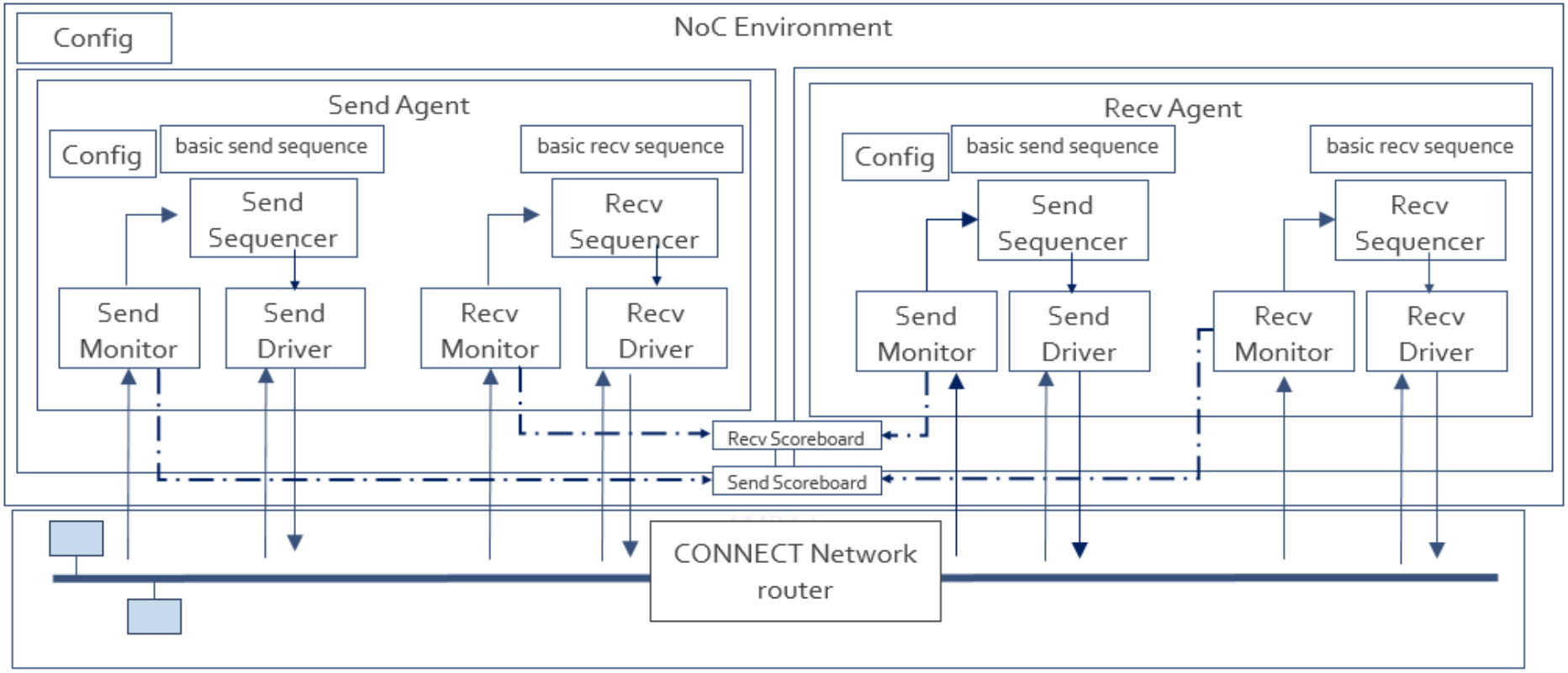


Figure 4-1: NoC Verification Environment

The number of *Agents* instantiated depends on the parameter ***num_of_routers***. This parameter is defined in the *noc_config* class and it is passed to all the components using *uvm_config_object*. When each of these *Agents* is created, an *id* associated with it is also created. This *id* is passed to all the *Agents* sub-components using *uvm_config_int*. The implementation is shown in Figure 4-2.

```
38        for(int i = 0; i < cfg.num_of_routers; i++) begin
39                string sname,sname_with_star ;
40                sname = $psprintf("agents[%0d]", i);
41                sname_with_star = $psprintf("agents[%0d]*", i);
42                uvm_config_int::set(this, sname_with_star, "id",
  i);                          // distribute master_agent index
43                uvm_config_object::set(this, sname_with_star, "cfg",
  cfg.master_configs[i]); // distribute master_config
44                        agents[i] = noc_agent::type_id::create(sname, this);
45
46                        if (agents[i] == null)
47                                `uvm_info(get_type_name(), $psprintf("agents[%
  0d] is null", i), UVM_LOW)
48                        else
49                                `uvm_info(get_type_name(), $psprintf("agents[%
  0d] is not null", i), UVM_LOW)
50                end
```

Figure 4-2: NoC Environment Implementation

In the Figure 4-3, part of the *Agent* implementation is shown. The *is_active* is another important parameter set in the *Configuration Objects* class. This parameter can be set to *UVM_ACTIVE* or *UVM_PASSIVE*. *UVM_ACTIVE* means that this UVC is activated and is going to be used to generate the stimulus. As an active component, the *Driver* and *Sequencer* will be created apart from the *Monitor*. If the component is set to passive using *UVM_PASSIVE*, then only *Monitor* will be created. In this NoC verification, a Send and Recv pair of *Driver*, *Monitor* and *Sequencer* are instantiated.

```
41 function void noc_agent::build_phase(uvm_phase phase);
42         super.build_phase(phase);
43
44         if(cfg == null) begin   // check whether tb set the config
45         if (!uvm_config_db#(noc_config)::get(this, "", "cfg", cfg))
   begin   // try to get config from database
46                 `uvm_error("NOCFG",{"config must be set for: ",get_full_name
   (),".cfg"});    // no config found
47                 end
48         end
49         `uvm_info(get_type_name(),$psprintf("cfg = %s", cfg.sprint()),UVM_LOW)
50         if(cfg.master_configs[id].is_active == UVM_ACTIVE) begin
51                 `uvm_info(get_type_name(), "noc config is active, create driver
   and sequencer", UVM_LOW)
52
53             s_sequencer= noc_send_sequencer::type_id::create("s_sequencer",this);
54             r_sequencer= noc_recv_sequencer::type_id::create("r_sequencer",this);
55
56             s_driver = noc_send_driver::type_id::create("s_driver", this);
57             r_driver = noc_recv_driver::type_id::create("r_driver", this);
58         end
59
60     s_monitor = noc_send_monitor::type_id::create("s_monitor",this);
61     r_monitor = noc_recv_monitor::type_id::create("r_monitor",this);
62 endfunction: build_phase
```

Figure 4-3: NoC *Agent* Implementation

The *flit* transaction is defined using a class. This has been used in the *Sequence* to contain the stimulus, the *Driver* packs this *flit* and drives to the *VSif* virtual interface that connects to the DUT. The implementation of the *Driver* is shown in Figure 4-4.

```
68 //drive transfer task
69 task noc_send_driver::drive_transfer(noc_flit flit);
70
71         `uvm_info(get_type_name(), "drive flit", UVM_HIGH)
72         `uvm_info(get_type_name(), $psprintf("flit: %s",flit.sprint()), UVM_HIGH)
73         VSif.EN_putFlit = 1;
74         VSif.putFlit = {flit.valid, flit.is_tail, flit.dst, flit.vc,
   flit.data};
75         @(posedge VSif.clk);
76         VSif.EN_putFlit <= 0;
77 endtask : drive_transfer
```

Figure 4-4: NoC Send *Driver* Implementation

The *Recv Monitor* then monitors the flit. It waits until a valid *flit* is monitored. The corresponding fields in the *flit* are packed into *flit_temp*. Once all the values of the virtual interface have been recorded into the *flit_temp*, the *flit_temp* is passed to the subscribers that subscribe to the *Monitor*. The Figure 4-5 shows the implementation of the *NoC Recv Monitor*.

```
82 task noc_recv_monitor::wait_flit();
83         forever begin
84                 @(posedge this.VRif.clk);
85
86                 while (this.VRif.getFlit[(`FLIT_DST_SIZE+`FLIT_VC_SIZE
   +`FLIT_DATA_SIZE+1)] == 0) //is valid flit..
87                 begin
88                         @(posedge this.VRif.clk);
89                 end
90                 `uvm_info(get_type_name(), "recv mon flit", UVM_LOW)
91                 flit_temp = noc_flit::type_id::create("flit_temp"); //storage
   for incoming flit
92
93                 if (VRif.getFlit[(`FLIT_DST_SIZE+`FLIT_VC_SIZE+`FLIT_DATA_SIZE
   +1)] == 'bx)
94                         `uvm_error(get_type_name(),"Valid Flit return x value.")
95                 flit_temp.valid = VRif.getFlit[(`FLIT_DST_SIZE+`FLIT_VC_SIZE
   +`FLIT_DATA_SIZE+1)];
96

117                //$display("%3d %3d %3d %3d %3d is credit = %
   3d",flit_temp.valid, flit_temp.is_tail, flit_temp.dst, flit_temp.vc,
   flit_temp.data, flit_temp.is_credit);
118                recv_port.write(flit_temp);
119
120                flit_ready = 1; //Flag to send flit to sequencer
121        end
122 endtask : wait_flit
```

Figure 4-5: NoC Recv *Monitor* Implementation

The *Send Scoreboard* and *Recv Scoreboard* are instantiations of the same *Single Channel NoC Scoreboard*. These *Scoreboard*s are used to compare and check if the send flit is received by the right endpoint of router as illustrated in Figure 4-6.

```
152                 if (send_flit.compare(recv_flit)) begin
153                         flit_valid++;
154                         `uvm_info(get_type_name(),$psprintf("FLIT SEND:\n %0s,
    \n RECV: \n %s", send_flit.sprint(), recv_flit.sprint()),UVM_LOW)
155                 end
156                 else begin
157                         flit_invalid++;
158                         `uvm_info(get_type_name(),$psprintf("Sender: %0d
    Receiver %0d",sender,receiver),UVM_LOW)
159                         `uvm_error(get_type_name(),$psprintf("FLIT SEND:\n %0s,
    \n RECV: \n %s", send_flit.sprint(), recv_flit.sprint()))
160                 end
161                 num_flit++;
```

Figure 4-6: Flit Compare

Similarly, the *Driver* at the receiving end sends another flit back. The *Recv Monitor* at the sending end now *monitors* the flit. The *Recv Scoreboard* checks the responses.

4.2.2 AHB Environment

In our proposed NoC architecture, each of the ARM M0 core has several peripherals connected to it using the ARM Advanced High-performance Bus (AHB). The AHB protocol consists of address and data phase working in pipeline as shown in Figure 4-7. During the address phase, the AHB master sends the control signals which includes address

(*HADDR*), and transfer type (*HWRITE*) to the AHB slaves through the AHB-Lite bus. The data (*HRDATA*) is send or read during the data phase at the cycle following the address phase.

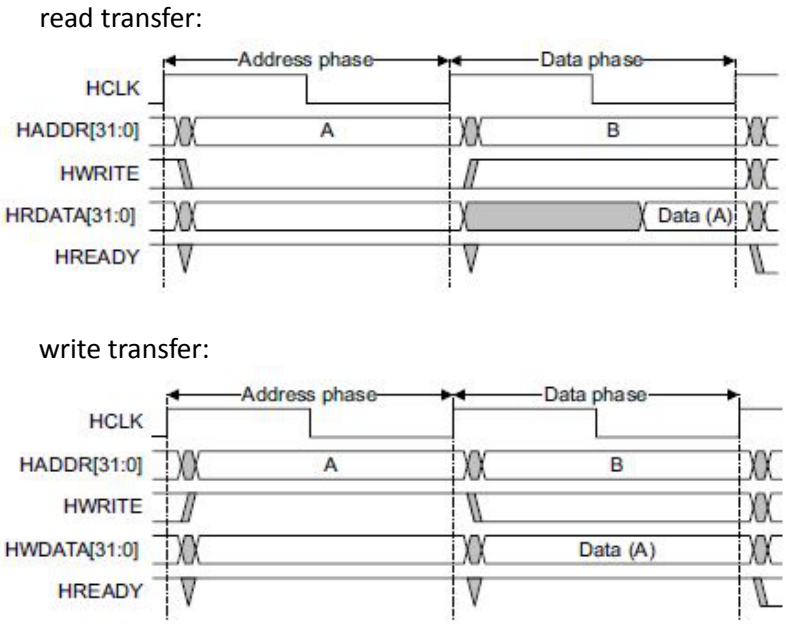


Figure 4-7: AHB Read Write Transfer (ARM, 2006)

If additional cycles are required, *HREADY* signal can be set to 0 by the AHB slaves until the slave is ready to receive the data from the AHB master during the write transfer or to have the data ready for the AHB master to read during the read transfer shown in Figure 4-8. (ARM, 2006)

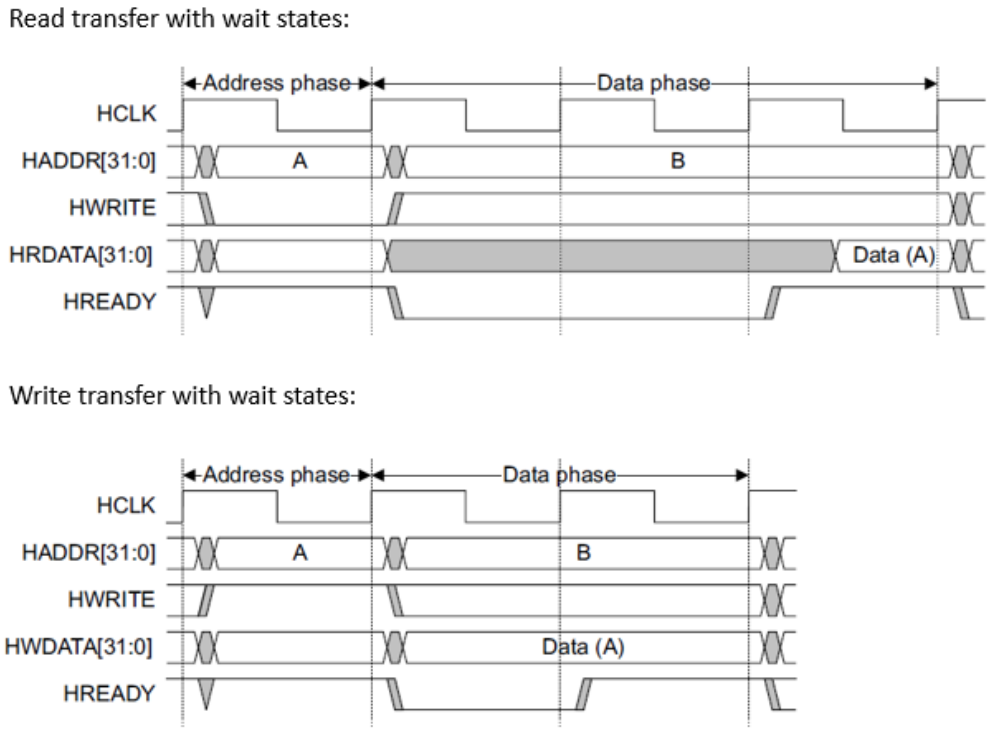


Figure 4-8: AHB Read Write Transfer with Wait States (ARM, 2006)

From the NoC architecture perspective, each core and its peripherals is a component. Thus, the basic building block of the verification environment is the AHB environment. A component level verification of an AHB system consists of an AHB master and two AHB slaves is shown in Figure 4-9.

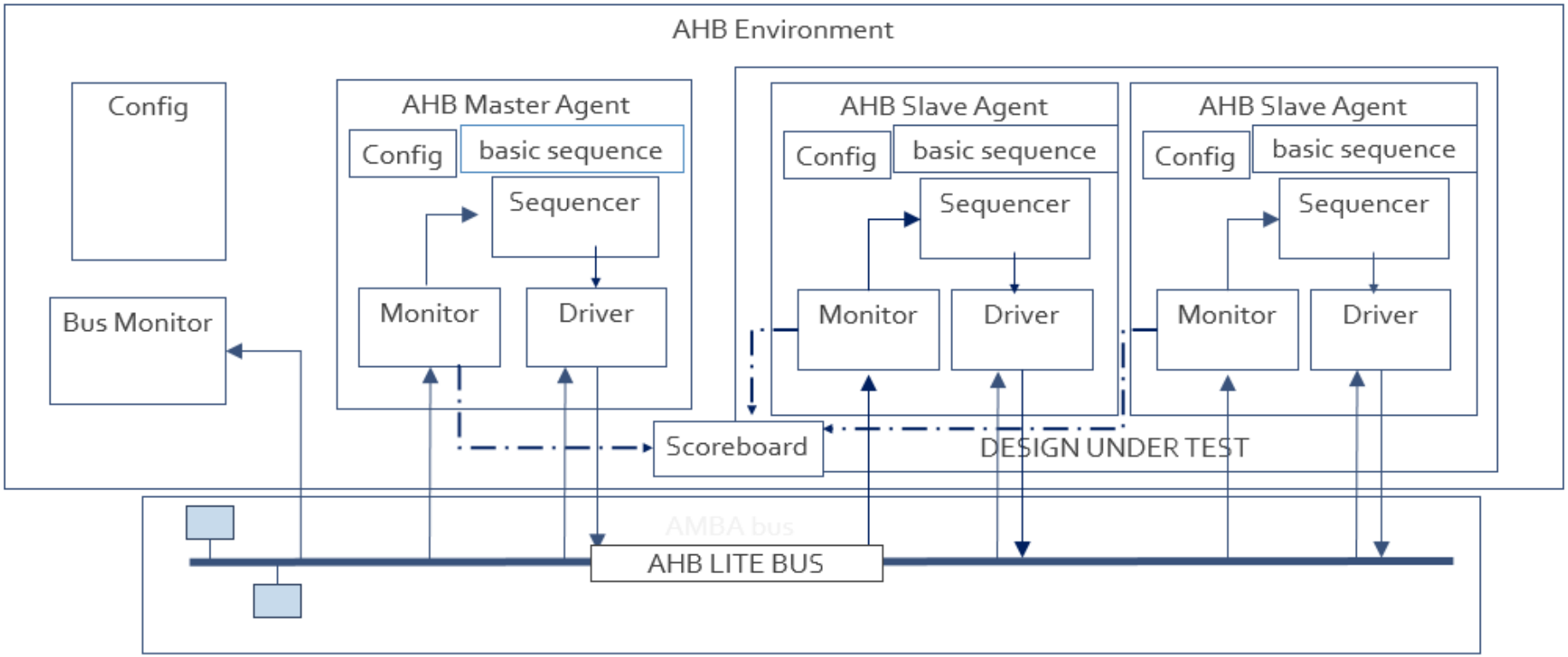


Figure 4-9: AHB Environment

It should be noted that the AHB slave is a generic model which can be further extended to model specific components, such as memory controller, GPIO, SPI, I^2C, USART, ADC, RTC, Timer etc. The basic *Sequences* will be discussed in Chapter 5 Section 5.2. In our work, the AHB master *Sequence* will model write transactions to access both AHB memory slave models through the AHB master *Driver*.

Figure 4-10 shows the implementation of the AHB master *Driver*. During the start of the simulation, the default values are driven to the DUT through the *Vif* interface. The *wait_reset_deassertion()* task is used by the AHB master waits for the system to reset. After the system reset has been detected, the *get_and_drive_pipelined()* task performs the address and data phases. In the event the reset is asserted when the system is running, the *monitor_reset_assertion()* task is used to monitor this event. The unfinished transaction that occurs due to the asynchronous reset is managed by the *manage_unfinished_trans()* task.

```
61        this.vif.MASTER_HADDR   <= 0;     // drive IDLE transaction during the
  start of the simulation
62        this.vif.MASTER_HWRITE  <= READ;
63        this.vif.MASTER_HSIZE   <= BYTE;
64        this.vif.MASTER_HBURST  <= SINGLE;
65        this.vif.MASTER_HTRANS  <= IDLE;  // IDLE
66        this.vif.MASTER_HWDATA  <= 0;
67        this.vif.MASTER_HLOCK   <= 1'b0;    // No locked transfer
68        this.vif.MASTER_HPROT   <= 4'b0001; // Always drive data (We do not know
69                                    // instruction encoding
70
71        forever begin
72        wait_reset_deassertion(); //synchronously wait the reset deaserrted
73        fork
74                get_and_drive_pipelined();  //invoke two thread to drive
  pipeline transaction
75                monitor_reset_assertion();  //asynchronously check the reset
76        join_any
77
78        disable fork;
79        //progress_lock.put();  //release all lock
80        pipeline_lock.put();
81        manage_unfinished_trans();  // finished current transaction
82        end
```

Figure 4-10: AHB Master *Driver* Implementation

These transactions are also monitored by the AHB master *Monitor*. The implementation is shown in Figure 4-11.

```
69        forever begin
70                wait_reset_deassertion();       //synchronously wait the reset
  deaserrted
71                fork
72                        monitor_transactions(); //invoke two thread to monitor
  pipeline transaction
73                        monitor_reset_assertion();      //asynchronously check
  the reset
74                        monitor_error();
75                        monitor_ready();
76                join_any
77
78                disable fork;
79                progress_lock.put();    //release all lock
80                pipeline_lock.put();
81                reset_detected = 1;             //if the sequence call the get
  method,
82                //this will cause the get method to return
83        end
```

Figure 4-11: AHB Master Monitor Implementation

In the *Monitor*, *monitor_transaction(), monitor_reset_assertion(), monitor_error()* and *monitor_ready()* tasks are used to indicate different transaction types. The *monitor_reset_assertion()* task monitors the system resets while the *monitor_transaction()* task is used to monitor the address and data phases. When the data phase and address phase has completed, *monitor_ready()* is used to inform the base *Sequence* the transaction is ready. *monitor_error()* is used to indicate any error transaction. These transactions are sent to the subscribers of the *Monitor*.

On the other hand, the AHB slaves will record the write transactions from the master through the AHB slave *Monitor* and loop back the transactions using the AHB slave *Driver* for the AHB master to read.

To verify the correct functionalities of the master and slaves, the *Scoreboard* gathers the transactions from AHB master and slave *Monitor*s to compare for the correctness of the transactions. This is shown in Figure 4-12.

```
48                wait(slave_trans && master_trans) begin
49                        if(slave_transfer.size == master_transfer.size &&
slave_transfer.burst == master_transfer.burst
50                                && slave_transfer.trans ==
master_transfer.trans && slave_transfer.direction == master_transfer.direction
51                                && slave_transfer.address ==
master_transfer.address && slave_transfer.data == master_transfer.data) begin
52
53                                `uvm_info(get_type_name(), $psprintf
("Transaction matched"), UVM_HIGH)
54                                /*`uvm_info(get_type_name(), $psprintf("Size\t:
%d", master_transfer.size), UVM_HIGH)
55                        `uvm_info(get_type_name(), $psprintf("Burst\t: %d",
master_transfer.burst), UVM_HIGH)
56                                `uvm_info(get_type_name(), $psprintf("Trans\t: %
d", master_transfer.trans), UVM_HIGH)
57                                `uvm_info(get_type_name(), $psprintf("Dir\t: %
d", master_transfer.direction), UVM_HIGH)
58                                `uvm_info(get_type_name(), $psprintf("Addr\t:
h'%0h", master_transfer.address), UVM_HIGH)
59                                `uvm_info(get_type_name(), $psprintf("Data\t:
h'%h0h\n", master_transfer.data), UVM_HIGH)*/
60                        end
61
62                        else begin
63
64                                `uvm_info(get_type_name(), $psprintf
("Transaction NOT matched"), UVM_HIGH)
```

Figure 4-12: AHB Scoreboard Implementation

The *Configuration Objects* are added to allow the slaves model to be mapped to different regions in the memory map using the *add_slave()* function as shown in Figure 4-13.

```
138 //------------------------------------------------------------------
139 // ahb_config - Creates and configures a slave agent config and adds to a queue
140 //------------------------------------------------------------------
141 function void ahb_config::add_slave(string name, bit [`BUS_SIZE:0]  start_addr,
    bit [`BUS_SIZE:0]  end_addr,
142         int hsel_indx, uvm_active_passive_enum is_active = UVM_ACTIVE, int
    hready_low_dur = 16);
143         ahb_slave_config tmp_slave_cfg;
144         num_slaves++;
145         tmp_slave_cfg = ahb_slave_config::type_id::create("slave_config");
146         tmp_slave_cfg.name = name;
147         tmp_slave_cfg.start_address = start_addr;
148         tmp_slave_cfg.end_address = end_addr;
149         tmp_slave_cfg.hsel_index = hsel_indx;
150         tmp_slave_cfg.is_active = is_active;
151         tmp_slave_cfg.hready_low_dur = hready_low_dur;
152
153         slave_configs.push_back(tmp_slave_cfg);
154 endfunction : add_slave
```

Figure 4-13: AHB add_slave() function

4.2.3 APB Environment

For completeness of the system, we have also developed the APB verification environment using similar structure as described above (ARM, 2003). We can use the Advanced Peripheral Bus (APB) to connect to components that have lower bus bandwidth requirements. When there is no transfer, the APB stays in the default state that is the IDLE state. Once the transfer occurs, the bus state changes from IDLE to SETUP. The appropriate select signal, *PSELx* is asserted. The bus remains in this state for 1 clock cycle. During the next clock cycle, the bus state changes to ENABLE state. In this state, PENABLE is asserted which also last for 1 clock cycle. If there is no further transfer, the bus state returns to IDLE. The bus changes from ENABLE to SETUP when there is another transfer. Figure 4-14 illustrates the overall APB transfer process.

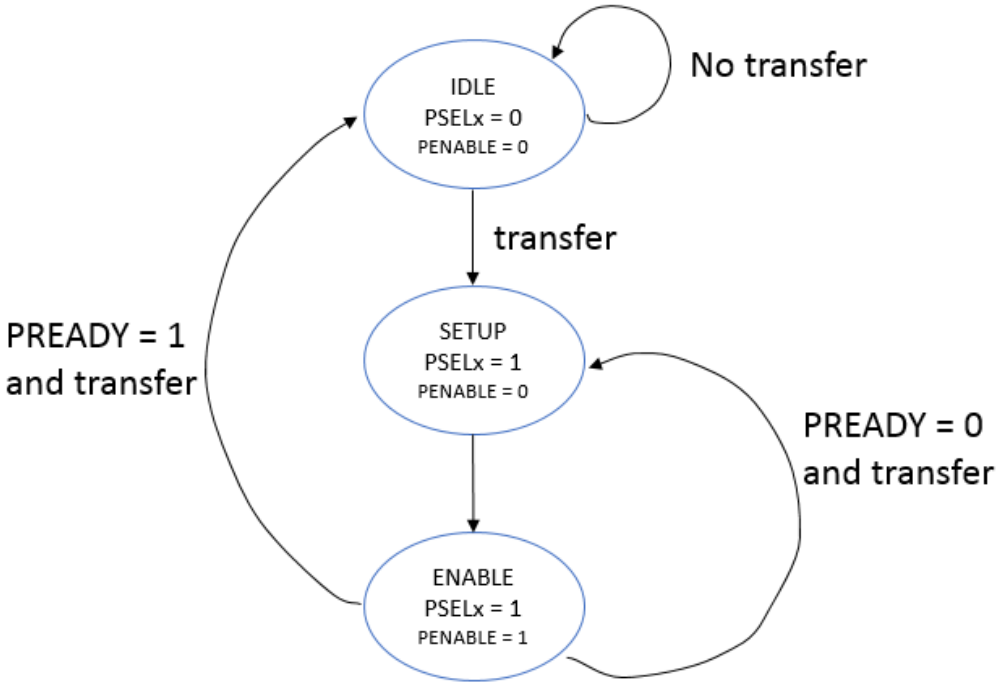


Figure 4-14: APB transfer state machine

During the write transfer, the address (*PADDR*), data (*PRDATA*) and select (*PSEL*) signal is asserted during the SETUP cycle. The enable signal, PENABLE is asserted during the next clock cycle, indicating the ENABLE cycle is taking place. The address, data, and control signals remain valid throughout the cycle. Once the transfer is completed, the enable signal is de-asserted. The timing diagram in Figure 4-15 shows the write process.

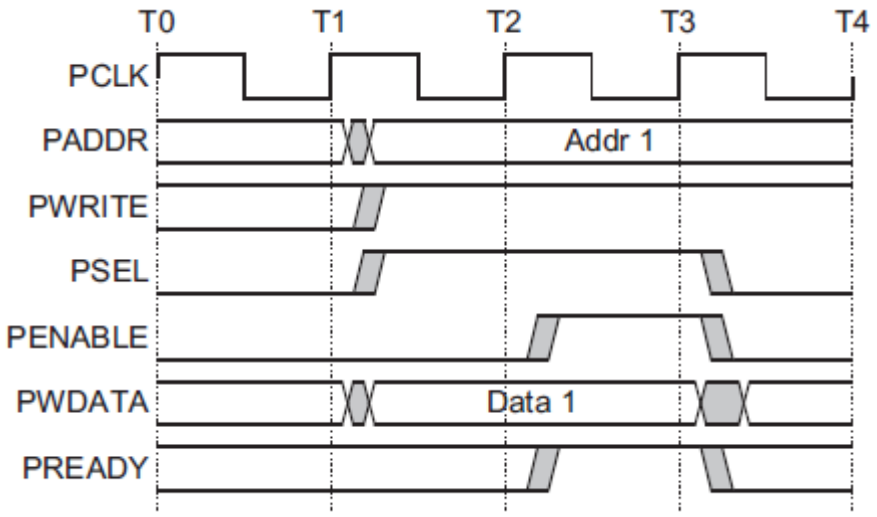


Figure 4-15: APB Write timing diagram (ARM , 2003)

Figure 4-16 shows the read process for the APB bus. Similarly, the address, write and select signal is set during the setup cycle. In this case, the write signal is de-asserted. The APB slave is required to provide the data once the enable signal is asserted. The data will be sampled on the rising edge of the clock at the end of the ENABLE cycle.

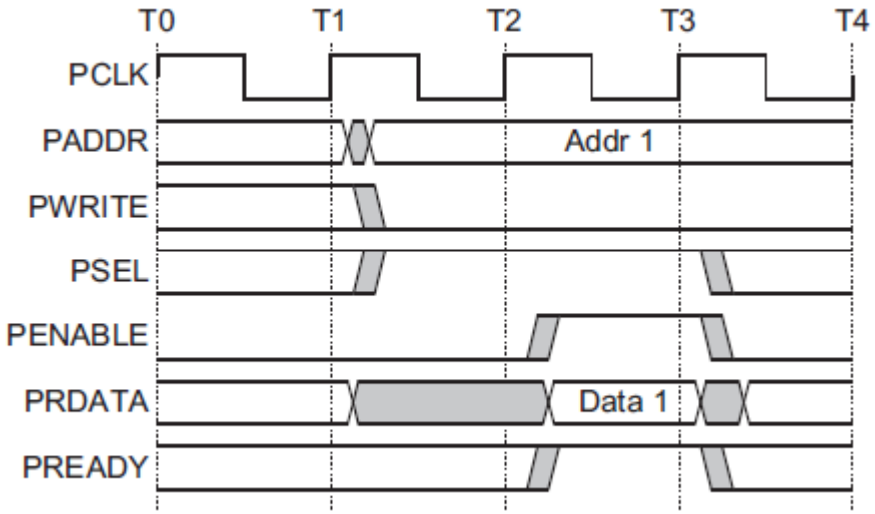


Figure 4-16: APB Read timing diagram (ARM , 2003)

If additional time is required, the APB can de-assert the ready signal when the enable signal is asserted. This holds the APB in the SETUP state until the ready signal is asserted indicating the slave is ready to receive or provide the data. When the ready signal is asserted, the state changes from SETUP to enable and performs the transfer as shown in Figure 4-17.

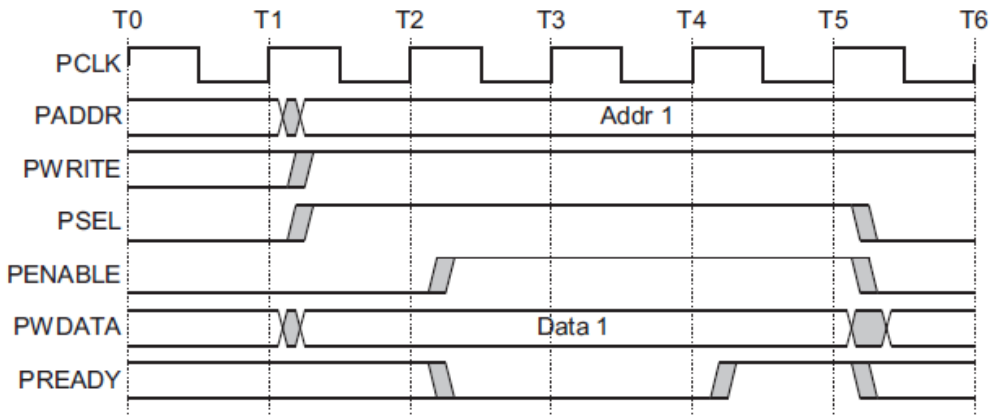


Figure 4-17: APB timing diagram with wait state (ARM , 2003)

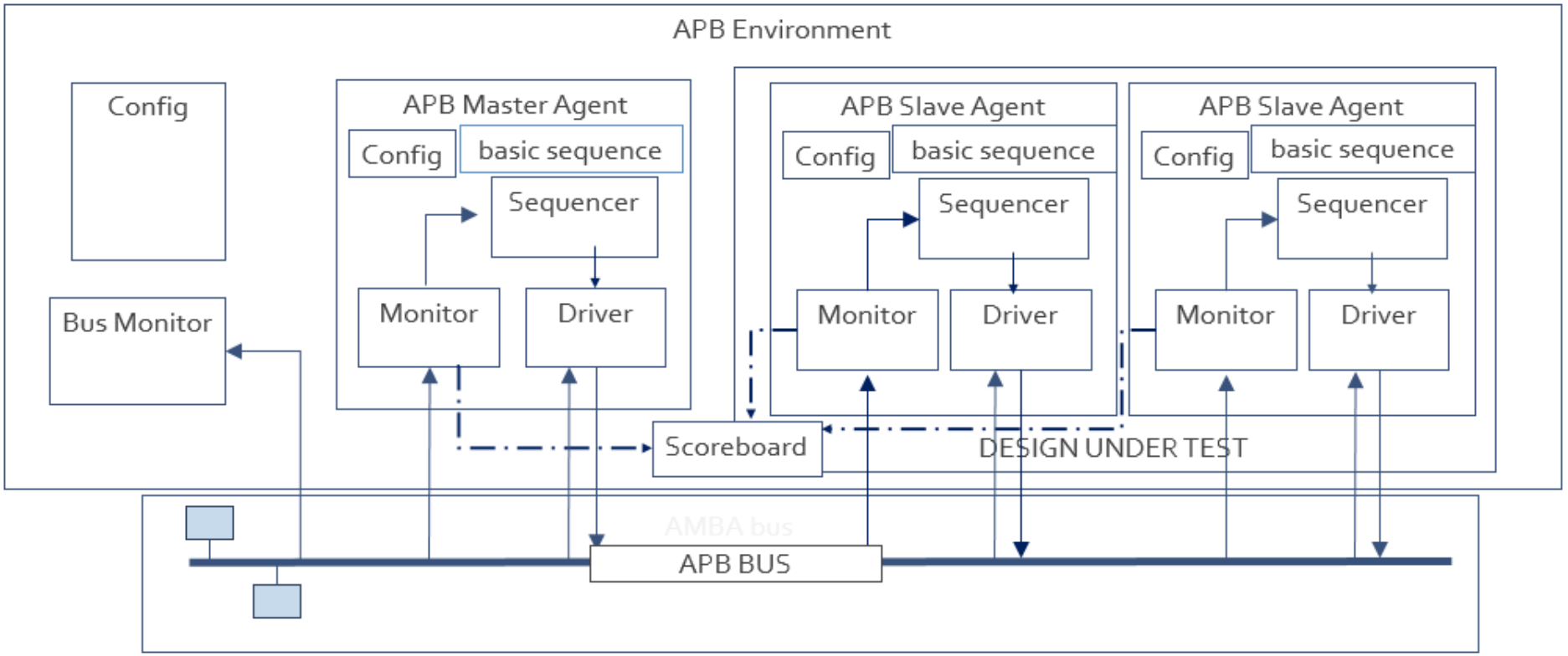


Figure 4-18: APB Environment

The APB environment consists of APB Master and 2 APB slaves modelled as memory is shown in Figure 4-18. The APB master is connected to the APB slaves through the APB bus. The APB master will perform a write read transfer to access these memories. The APB master *Driver* waits for the system to reset. It waits for new transaction from the *Sequence* using *get_next_item()*. The transaction obtained will be driven to the *Vif* virtual interface. Once the transaction is completed, it uses *item_done()* to indicate to the *Sequence* that the transaction has been successfully driven. Figure 4-19 shows the implementation of this APB master *Driver*.

```
85      reset();
86      fork
87        @(negedge vif.preset)
88          // APB_MASTER_DRIVER tag required for Debug Labs
89          `uvm_info("APB_MASTER_DRIVER", "get_and_drive: Reset dropped", UVM_MEDIUM)
90        begin
91          // This thread will be killed at reset
92          forever begin
93            @(posedge vif.pclock iff (vif.preset))
94            seq_item_port.get_next_item(req);
95            drive_transfer(req);
96            seq_item_port.item_done(req);
97          end
98        end
99        join_any
100       disable fork;
```

Figure 4-19: APB *Driver* Implementation

Figure 4-20 shows the implementation of the APB master *Monitor*. This *Monitor* is used to monitor the transaction driven by the APB master *Driver* and also the responses from the APB slaves. If the read transaction is monitored, the *Monitor* will gather the slave responses. When a write transaction is detected instead, the *Monitor* gathers the transaction driven by its master *Driver*.

```
117     trans_collected.addr = vif.paddr;
118     trans_collected.master = cfg.master_config.name;
119     trans_collected.slave = cfg.get_slave_name_by_addr(trans_collected.addr);
120     case (vif.prwd)
121       1'b0 : trans_collected.direction = APB_READ;
122       1'b1 : trans_collected.direction = APB_WRITE;
123     endcase
124       //@(posedge vif.pclock);
125     //if(trans_collected.direction == APB_READ)
126     //  trans_collected.data = vif.prdata;
127     if (trans_collected.direction == APB_WRITE)
128       trans_collected.data = vif.pwdata;
129     -> addr_trans_grabbed;
130     @(posedge vif.pclock);
131     if(trans_collected.direction == APB_READ) begin
132         if(vif.pready != 1'b1)
133           @(posedge vif.pclock);
134       trans_collected.data = vif.prdata;
135       end
136     this.end_tr(trans_collected);
137     item_collected_port.write(trans_collected);
138     `uvm_info(get_type_name(), $psprintf("Transfer collected :\n%s",
139               trans collected.sprint()), UVM MEDIUM)
```

Figure 4-20: APB Monitor Implementation

The *Checker* or the *Scoreboard* compares the intended transfer that is sent to each memory is received by the correct memory slave as shown in Figure 4-21.

```
61  function void apb_demo_scoreboard::memory_verify(input apb_transfer transfer);
62    int unsigned data, exp;
63    string op;
64    for (int i = 0; i < transfer.size; i++) begin
65      // Check to see if entry in associative array for this address when read
66      // If so, check that transfer data matches associative array data.
67      if (m_mem_expected.exists(transfer.addr + i)) begin
68        if (transfer.read_write == READ) begin
69          op = transfer.read_write.name();
70          data = transfer.data[i];
71          `uvm_info(get_type_name(),
72            $psprintf("%s to existing address...Checking address : %0h with data : %
    0h", op, transfer.addr, data), UVM_LOW)
73          if(m_mem_expected[transfer.addr + i] != transfer.data[i]) begin
74            exp = m_mem_expected[transfer.addr + i];
75            `uvm_info(get_type_name(),
76              $psprintf("Read data mismatch.  Expected : %0h.  Actual : %0h", exp,
    data), UVM_NONE)
77          end
78          num_init_reads++;
79        end
80        if (transfer.read_write == WRITE) begin
81          op = transfer.read_write.name();
82          data = transfer.data[i];
83          `uvm_info(get_type_name(),
84              $psprintf("%s to existing address...Updating address : %0h with
    data : %0h",
85                op, transfer.addr + i, data), UVM_LOW)
86          m_mem_expected[transfer.addr + i] = transfer.data[i];
87          num_writes++;
88        end
89      end
```

Figure 4-21: APB Scoreboard

The behaviour of the stimulus is modelled in the basic *Sequence* which can be extended to model other peripherals such as, in this case, APB memory. The implementation of the basic *Sequence* is discussed in Chapter 5 Section 5.2.

4.2.4 SPI Environment

The SPI module is attached to the APB bus in our system. This module is to allow each of the Cortex M0 to send status information such as debug messages. For the SPI UVC environment, basic *Sequences* have been created to generate the SPI transfer. The UVC can be configured to generate and capture various SPI modes. Figure 4-22 shows the 8 bit SPI transfer with the transmitting and receiving at the negative edge of the clock. The 1st bit also can be configured as the most significant bit (MSB).

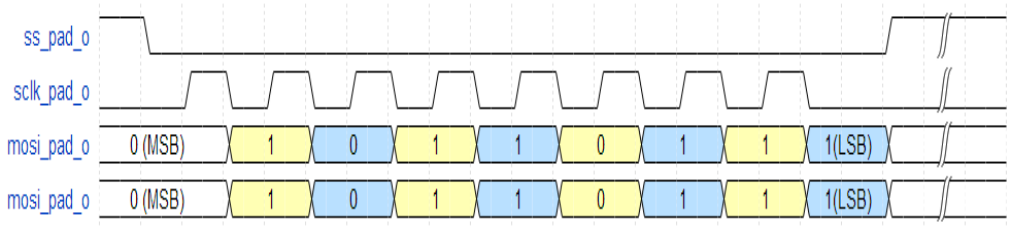


Figure 4-22: 8bit MSB SPI transfer at negative clock edge

Another example is shown in Figure 4-23. The transfer is configured with the 1st bit as least significant bit (LSB). The transmitting and receiving of the transfers occurs at the positive edge of the generated clock, *sclk*.

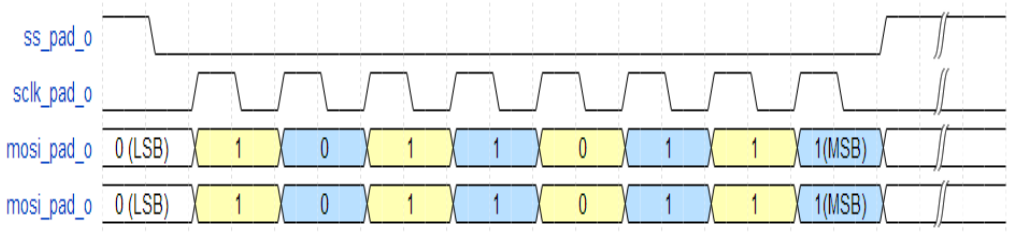


Figure 4-23: 8bit LSB SPI transfer at positive clock edge

The SPI *Driver* implementation is shown in Figure 4-24. In this UVC, the C*onfiguration Objects* have been used to determine the SPI transfer Objects such as data transfer size, data type indicating the first bit the Most Significant Bit (MSB) or the Least Significant Bit (LSB) and, the transmit and receive clock edges. Before the SPI drives the SPI virtual interface, *spi_if*, the SPI *Driver* waits until the SPI *Monitor* detects an SPI slave is selected. It then drives each bit of the transaction until it reached the configured transaction data size. The *Driver* utilizes the SPI *Monitor* to determine the SPI clock edges that the transaction should be sent.

```
153     if (csr_s.mode_select == 1) begin       //DUT MASTER mode, OVC SLAVE mode
154         `uvm_info(get_type_name(),"MODE SELECT == 1", UVM_HIGH);
155         @monitor.new_transfer_started;
156         `uvm_info(get_type_name(),"New transfer spi driver_monitor", UVM_HIGH);
157         for (int i = 0; i < csr_s.transfer_data_size-1; i++) begin
158                 @monitor.new_bit_started;
159                 //`uvm_info(get_type_name(),"CSR data size ", UVM_HIGH);
160                 spi_if.sig_n_so_en <= 1'b0;
161                 spi_if.sig_so <= trans.transfer_data[i];
162         end
163         spi_if.sig_n_so_en <= 1'b1;
164         `uvm_info("SPI_DRIVER", $psprintf("Transfer sent :\n%s", trans.sprint
    ()), UVM_HIGH)
165     end
```

Figure 4-24: SPI *Driver* Implementation

Figure 4-25 shows the implementation of the SPI *Monitor*. It waits until an SPI slave is selected and packs each bits collected from the SPI virtual interface to be sent to its subscribers. If the *tx_clk_phase* in the SPI *Configuration Parameter* is set to zero, the transaction will be collected during the negative phase of the clock and vice versa.

```
164     if (csr_s.tx_clk_phase == 0) begin
165         `uvm_info("SPI_MON", $psprintf("CSR_S is %s", csr_s.sprint()), UVM_HIGH)
166         for (int i = 0; i < csr_s.transfer_data_size; i++) begin
167                 @(negedge spi_if.sig_sclk_in);
168                 -> new_bit_started;
169                 if (csr_s.mode_select == 1 || csr_s.spi_model) begin     //DUT
    MASTER mode, OVC Slave mode
170                         if(csr_s.lsb == 1) begin
171                                 trans_collected.receive_data[i] = spi_if.sig_si;
172                                 `uvm_info("SPI_MON", $psprintf("LSB is 1
    received data[%0d] is %0h",i, trans_collected.receive_data[i]), UVM_HIGH)
173                                 `uvm_info("SPI_MON", $psprintf("data_size = %0d
    i = %0d", csr_s.transfer_data_size, i), UVM_HIGH)
174                         end
175                         else begin
176                                 trans_collected.receive_data
    [csr_s.transfer_data_size-i-1] = spi_if.sig_si;
177                                 `uvm_info("SPI_MON", $psprintf("LSB = 0.
    received data[%0d] is %h",csr_s.transfer_data_size-i-1,
    trans_collected.receive_data[csr_s.transfer_data_size-i-1]), UVM_HIGH)
178                                 `uvm_info("SPI_MON", $psprintf("data_size = %0d
    i = %0d", csr_s.transfer_data_size, i), UVM_HIGH)
179                         end
180                 end
181                 else begin
182                         trans_collected.receive_data[i] = spi_if.sig_mi;
183                         `uvm_info("SPI_MON", $psprintf("received data in
    mode_select 0 is %0h", trans_collected.receive_data), UVM_HIGH)
184                 end
185         end
186     end
```

Figure 4-25: SPI Monitor Implementation of tx negative clock phase

4.2.5 Parallel port Environment

In our system, the parallel port module is used as an input for the AES encryption. The parallel port UVC is developed to verify the parallel port module. During the read transfer as shown in Figure 4-26, the parallel port drives *pp_av_n* low, indicating there is data available in the transmit buffer. When the host device detected the *pp_av_n* is low, it drives the *pp_strobe_n* low to initiate the read transfer. *pp_wrn_n* signal is driven high, indicating a read transfer. The parallel port asserts the *pp_wait_n* signal. At the same time, the data are available at the parallel port interface. The host reads the data and drives the *pp_strobe_n* signal high. This indicates the end of the transfer. The parallel port drives the *pp_wait_n* low. If there are still data remaining in the buffer, the *pp_av_n* signal remains low. The *pp_av_n* is asserted once the buffer is empty.

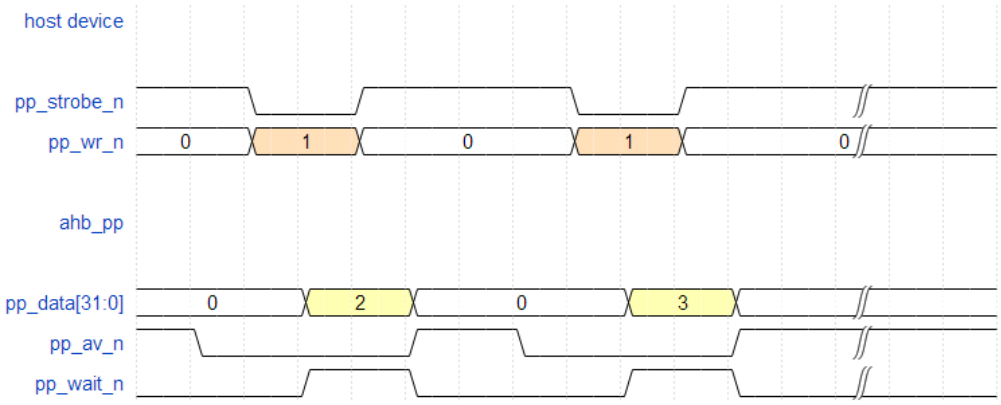


Figure 4-26: Parallel port read timing diagram

Figure 4-27 shows the protocol for the parallel port write. The host device will de-assert the *pp_strobe_n* signal to initiate the write transfer. At the same time, the *pp_wr_n* is also driven low to indicate write transfer type. The valid data is driven by the host device. Once the parallel port is ready, it reads the data and assert the *pp_wait_n* signal. To end the transfer, the host device drives the *pp_strobe_n* signal high. In the next cycle, *pp_wait_n* is driven low by the parallel port.

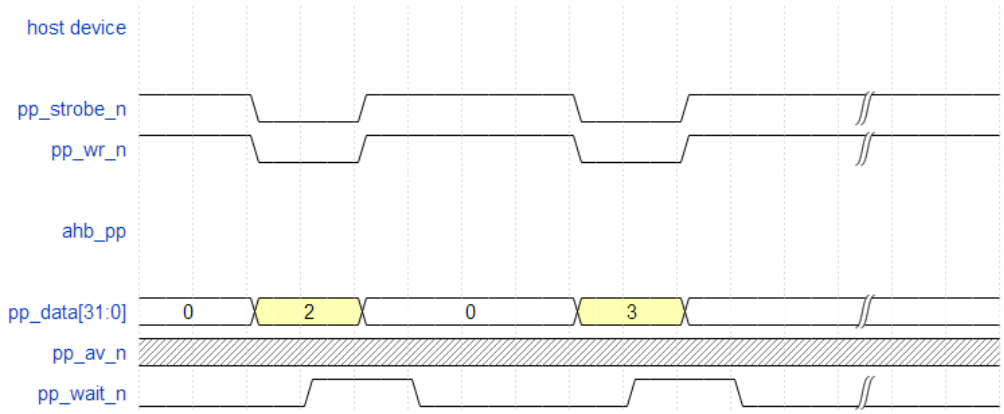


Figure 4-27: Parallel port write timing diagram

The read, write protocol of the parallel port modules has been modelled using *Sequences* in the parallel port UVC. This forms the basic *Sequence* for this UVC. The parallel port transfer stimulus generated by the UVC is injected to the parallel port RTL module for processing. The read *Sequence* enables the parallel port UVC to gather the transfer from the RTL.

Coupled with these basic read, write *Sequence*, the Parallel Port *Driver* in Figure 4-28 implements the Parallel Port write protocol. Similarly, the Parallel Port UVC drives the default values to the virtual

interface and waits for the system to reset using *wait_reset_deassertion()* task. Once the system has been reset, the UVC uses the get_and_drive() task to drive the available transactions from the *Sequence*. In the event that if the system reset occurs, *monitor_reset_assertion ()* is used to monitor this event.

```
 82        parallel_port_if.pp_strobe_n    <= 1;
 83                parallel_port_if.pp_wr_n                <= 1;
 84
 85                //foreach(parallel_port_if.pp_data[i])
 86                        parallel_port_if.pp_data                <= 'bz;
 87
 88                //foreach(parallel_port_if.i_pp_data[i])
 89                        parallel_port_if.i_pp_data              <= 'bz;
 90
 91        forever begin
 92
 93                wait_reset_deassertion();
 94                fork
 95                        begin: driving
 96                                get_and_drive();
 97                        end
 98                        monitor_reset_assertion();
 99                join_any
100
101                disable driving;
102        end
```

Figure 4-28: Parallel Port *Driver* Implementation

Figure 4-29 shows the implementation of the Parallel Port *Monitor*. The *trans_collected* is an instantiation of the parallel port transfer, which consist of the *pp_wr_n*, *pp_data* and *pp_data_width*. This object is used to collect the transaction from the *parallel_port_if* virtual interface.

```
140         while(parallel_port_if.pp_strobe_n == 1)
141                 @(posedge parallel_port_if.clk);
142
143         void'(this.begin_tr(trans_collected));
144
145                 trans_collected.pp_wr_n = parallel_port_if.pp_wr_n;
146
147                 @(posedge parallel_port_if.clk);
148
149         while(parallel_port_if.pp_wait_n == 0)
150                 @(posedge parallel_port_if.clk);
151
152                 trans_collected.pp_data = parallel_port_if.pp_data;
153
154                 `uvm_info(get_type_name(), $psprintf("parallel_port monitor
    finish collecting data: 0x%0h from parallel port interface: 0x%0h,
    pp_data_width: %0d", trans_collected.pp_data, parallel_port_if.pp_data,
    trans_collected.pp_data_width), UVM_HIGH)
155
156         endtask : collect_transfer
```

Figure 4-29: Parallel Port Monitor Implementation

4.2.6 GPIO Environment

The GPIO is added to our system as another means of I/O. It allows our system to send out transfer to another system. It also enables our system to use it to gather transfers. To verify this module, a basic UVC has been developed. The basic *Sequences* in the GPIO UVC allow us to perform basic GPIO read write transfer. It allows us to generate GPIO transfer for our system and *Monitor* the activity on the GPIO bus.

The GPIO *Driver* waits for the transaction delay which can be set in the GPIO basic *Sequence*. This can be used to model the delay between the GPIO transfers and constraints can be set to randomize the delay between the transfers. This can be used to model the delay between the GPIO transfers and constraints can be set to randomize the delay between the transfers. The *GPIO_DATA_WIDTH* parameter corresponds to each

of the GPIO pins. For each of the pins, if gpio_pin_oe is set to zero, the GPIO UVC will drive the transactions to the DUT.

```
 83  // drive_transfer
 84  virtual protected task drive_transfer (gpio_transfer trans);
 85    if (trans.transmit_delay > 0) begin
 86         repeat(trans.transmit_delay) @(posedge gpio_if.pclk);
 87    end
 88    drive_data(trans);
 89  endtask : drive_transfer
 90
 91  virtual protected task drive_data (gpio_transfer trans);
 92    @(posedge gpio_if.pclk);
 93        `uvm_info("GPIO_DRIVER", $psprintf("GPIO PIN OE:%0h",
    gpio_if.n_gpio_pin_oe), UVM_HIGH)
 94    for (int i = 0; i < `GPIO_DATA_WIDTH; i++) begin
 95      if (gpio_if.n_gpio_pin_oe[i] == 0) begin
 96        trans.transfer_data[i] = gpio_if.gpio_pin_out[i];
 97        gpio_if.gpio_pin_in[i] = gpio_if.gpio_pin_out[i];
 98      end else
 99        gpio_if.gpio_pin_out[i] = trans.transfer_data[i];
100    end
101    `uvm_info("GPIO_DRIVER", $psprintf("GPIO Transfer:\n%s", trans.sprint()),
    UVM_HIGH)
102  endtask : drive data
```

Figure 4-30: GPIO *Driver* Implementation

The GPIO *Monitor* implementation is illustrated in Figure 4-31. The variable *transfer_data* is used to collect the transaction when the GPIO UVC drives the transaction to the DUT. When the GPIO UVC receives the transaction from the DUT, *monitor_data* variable is used instead.

```
116    void'(this.begin_tr(trans_collected));
117    trans_collected.transfer_data = gpio_if.gpio_pin_out;
118    trans_collected.monitor_data  = gpio_if.gpio_pin_in;
119    trans_collected.output_enable = gpio_if.n_gpio_pin_oe;
120    if (!last_trans_collected.compare(trans_collected))
121      `uvm_info(get_type_name(), $psprintf("Transfer collected :\n%s",
    trans_collected.sprint()), UVM_MEDIUM)
122    last_trans_collected = trans_collected;
123    this.end_tr(trans_collected);
```

Figure 4-31: GPIO *Monitor* Implementation

4.3 *Subsystem level Testbench*

The subsystem verification environment builds up from multiple component level verification environment modules discussed in Chapter 4 Section 4.2. In this case, multiple *Sequences* targeted for these multiple environments are used. *Virtual Sequencer* can be used to coordinate these *Sequences*. For the subsystem *Scoreboard* to gather the transfer from its lower analysis components such as *Monitors, Analysis Port* can be used. The basic concepts *Analysis Port* has been discussed in Chapter 3 Section 3.2.1.2. The *Monitor* from the component level verification environment will broadcast the transfers that it has gathered through this port.

4.3.1 AHB2NoC Environment

Once we have the basic modules of the system and verification environment set up. We can reuse the modular components to develop our subsystem verification environment. The AHB2NoC subsystem verification environment uses the AHB and NoC UVC. The AHB master *Agent* generates the AHB transfer. This transfer is converted to flit by the AHB2NoC adapter. The NoC receive *Agent* is used to capture the flit from the AHB2NoC adapter and sends it to the *Scoreboard* to check for the correctness of the flit. The NoC send *Agent* is then used to generate a flit and send through the AHB2NoC adapter. This is to ensure that the content of the flit is converted to the corresponding AHB transfer. The environment is shown in Figure 4-32.

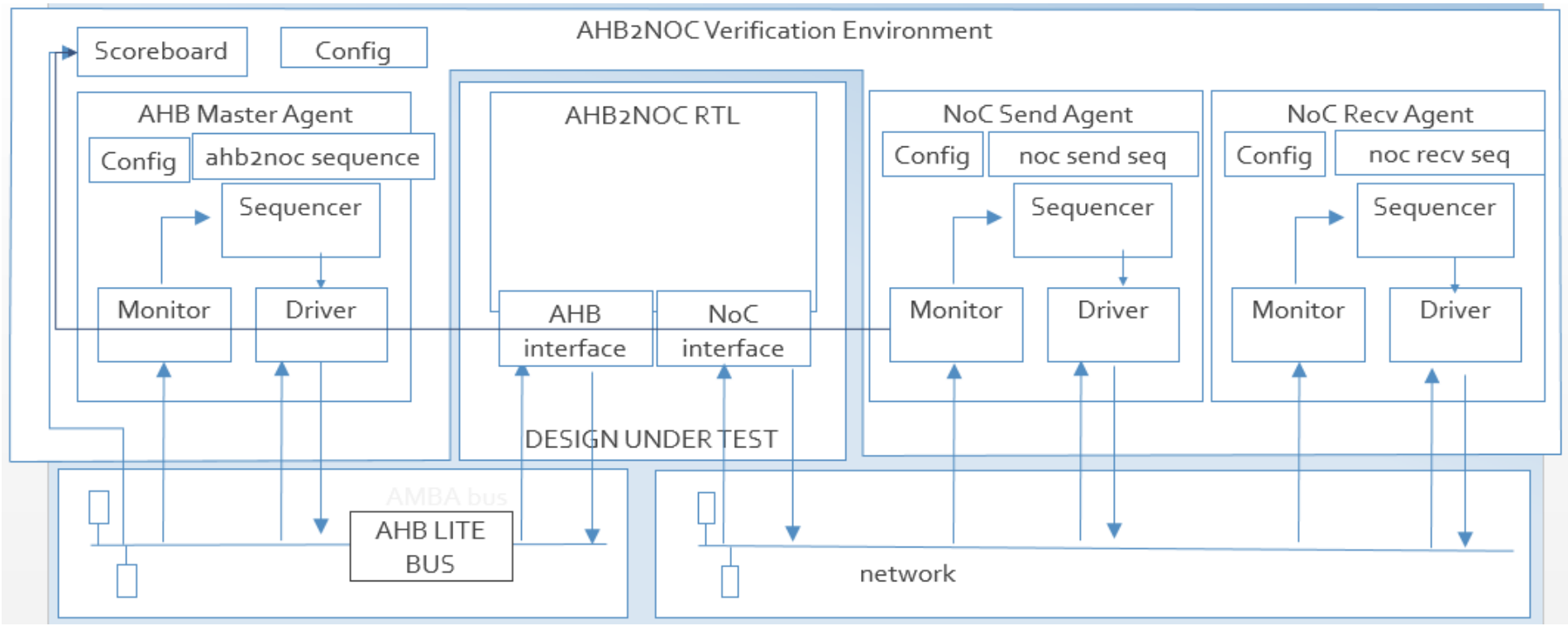


Figure 4-32: AHB2NoC Adapter Verification Environment

Since there are 2 different UVCs involved, the *Virtual Sequencer* has been used to coordinate the *Sequences* running. The *Sequencer* coordinates the *Sequence* for the AHB UVC to send the transfer and NoC UVC to monitor the transfer and vice versa. This is shown using Figure 4-33.

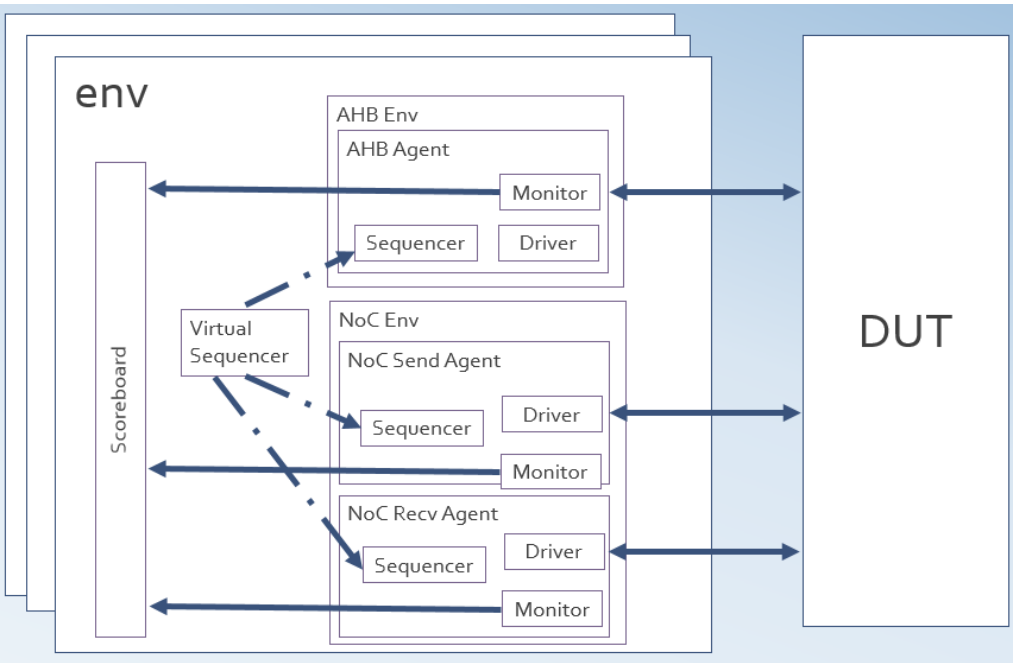


Figure 4-33: *Virtual Sequencer* and *Analysis port* for AHB2NoC

Figure 4-34 shows the implementation of our *Virtual Sequencer*. This *Sequencer* consist of both the AHB master *Sequencer* and two NoC slave recv and send *Sequencer*.

```
17 class ahb2noc_special_virtual_sequencer extends uvm_sequencer;
18
19         ahb_master_sequencer ahb_seqr;
20         noc_slave_recv_sequencer noc_slave_recv_seqr;
21         noc_slave_send_sequencer noc_slave_send_seqr;
22
23         function new (string name, uvm_component parent);
24                 super.new(name, parent);
25         endfunction : new
26
27         `uvm_component_utils_begin(ahb2noc_special_virtual_sequencer)
28         `uvm_component_utils_end
29
30         // build_phase
31         virtual function void build_phase(uvm_phase phase);
32                 super.build_phase(phase);
33         endfunction : build_phase
34
35         function void connect_phase(uvm_phase phase);
36                 super.connect_phase(phase);
37                 // Assign interface connection
38
39         endfunction : connect_phase
40
41 endclass : ahb2noc_special_virtual_sequencer
```

Figure 4-34: AHB2NoC *Virtual Sequencer* Implementation

As an example, during the creation of the test as shown in Figure 4-35, the *p_sequencer* which is the AHB2NoC *Virtual Sequencer* is registered with the factory. This allows different *Sequence* which runs on different *Sequencer* utilise the same *Virtual Sequencer*.

```
36                 fork
37                         `uvm_do_on(ask_flit_send_credit_seq,
   p_sequencer.noc_slave_recv_seqr);
38                 join_none
39
40                 fork
41                         `uvm_do_on(send_seq, p_sequencer.ahb_seqr);
42                         `uvm_do_on(send_flit_ask_credit_seq,
   p_sequencer.noc_slave_send_seqr);
43                 join
```

Figure 4-35: AHB2NoC Test Example

4.3.2 AHB Parallel Port Environment

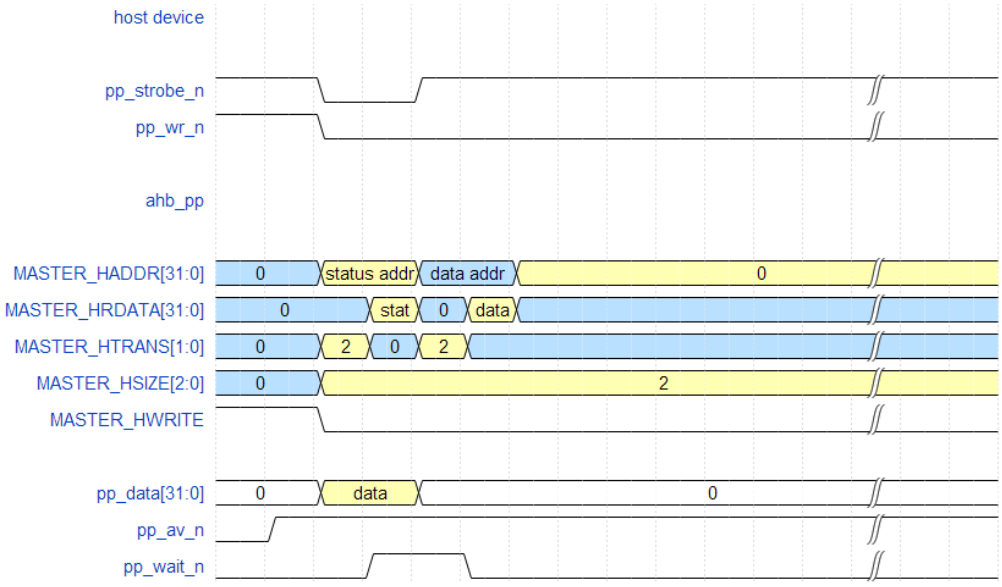


Figure 4-36: AHB Parallel Port read timing diagram

To write to the parallel port, the Cortex M0 begins by writing the address and transfer type. The transfer data is sent out during the next cycle. When the host device is ready to accept the transfer, the host device de-asserts *pp_strobe_n* signal. The parallel port reads the data and assert the *pp_wait_n* for the next 2 cycle. To indicate the end of the transfer, the host device drives the *pp_strobe_n* signal to high. *pp_wait_n* is driven low by the parallel port.

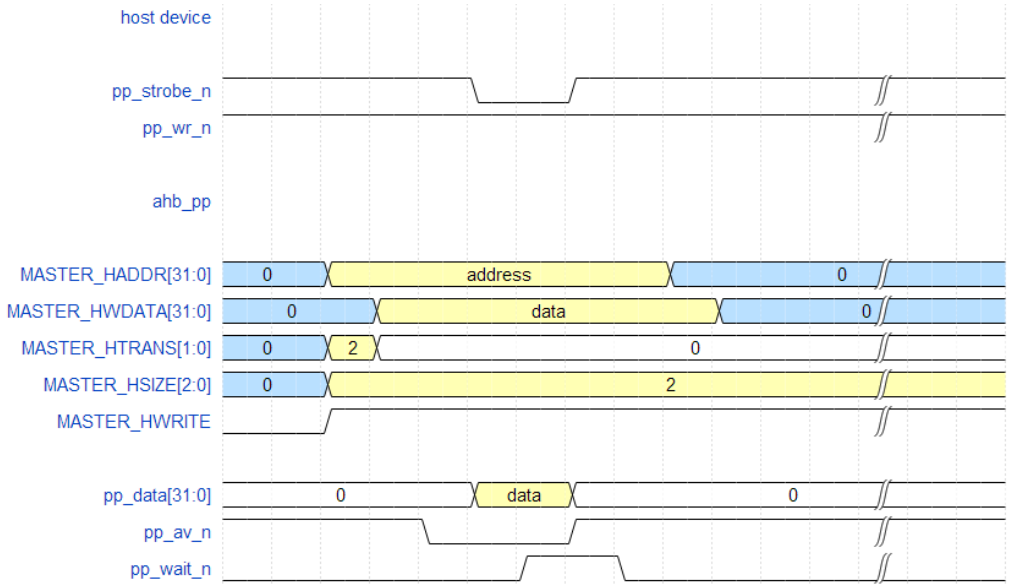


Figure 4-37: AHB Parallel Port write timing diagram

Figure 4-38 shows the setup of our AHB2ParallelPort verification environment. The AHB2ParallelPort RTL module is the design under test in this case. The AHB side of AHB2ParallelPort module connects to the AHB-Lite Bus while the other end connects to our parallel port interface. To verify this module, the AHB sends transfer to the parallel port. This transfer is also monitored by the AHB UVC. The parallel port transfer from the RTL module is monitored by the parallel port UVC. The *Scoreboard* has been setup to check for the correctness of the transfer by comparing the AHB with the parallel port transfer.

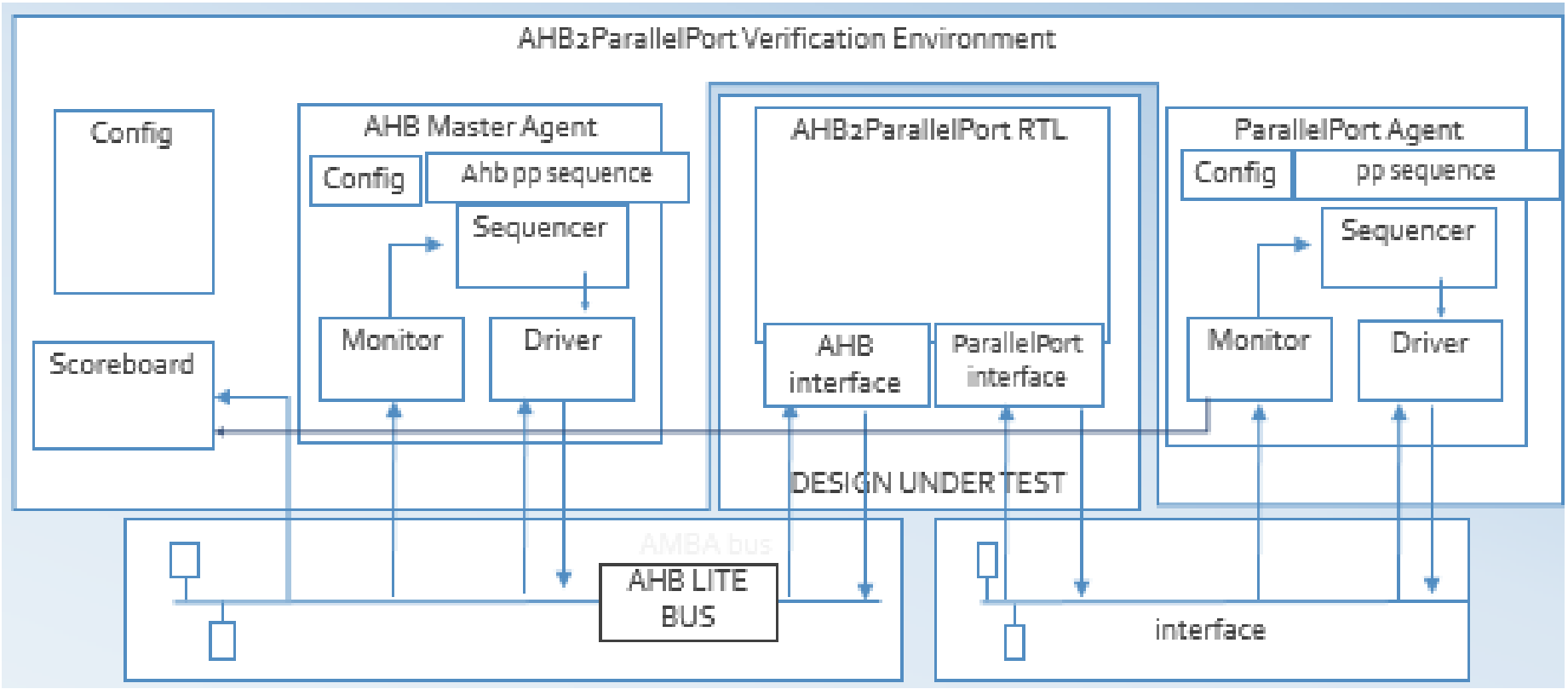


Figure 4-38: AHB2ParallelPort Verification Environment

The coordination between the AHB and parallel port UVC *Sequences* is done through the subsystem *Virtual Sequencer* as shown in Figure 4-39. This module is muxed with the GPIO module.

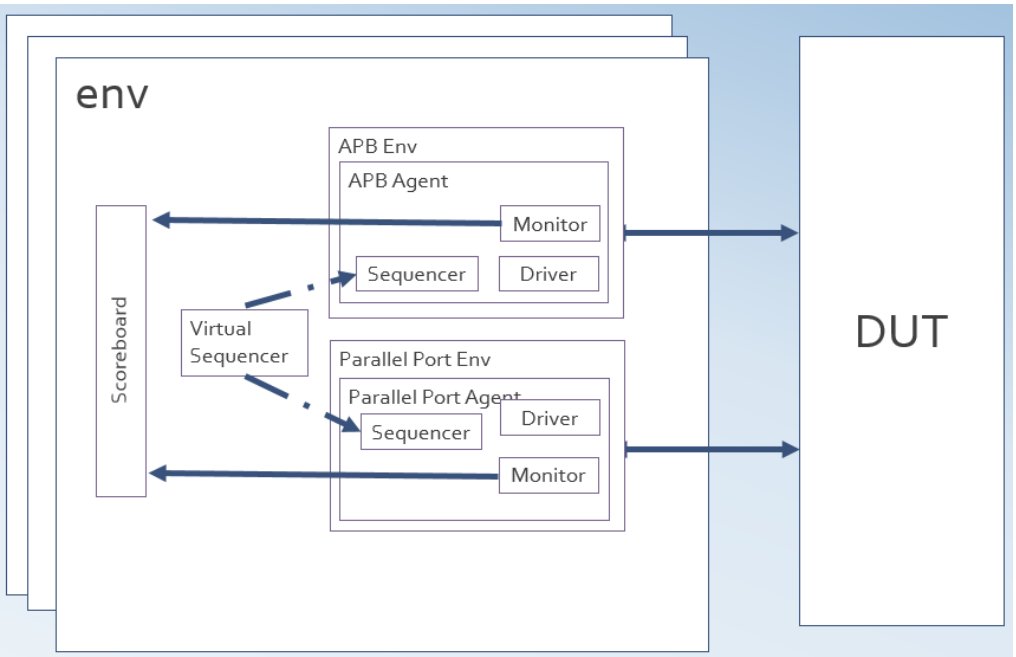


Figure 4-39: ParallelPort Verification Environment *Virtual Sequencer*

Similarly, instead of having the NoC slave *Sequencer*, the AHB Parallel Port Virtual *Sequencer* consist of an AHB master *Sequencer* and a parallel port *Sequencer* shown in Figure 4-40.

```
17 class ahb_pp_vir_sequencer extends uvm_sequencer;
18
19         ahb_master_sequencer ahb_seqr;
20         parallel_port_sequencer pp_seqr;
21
22         // The virtual interface used to drive and view HDL signals.
23         virtual interface ahb_pp_int_if vif;
24
25         function new (string name, uvm_component parent);
26                 super.new(name, parent);
27         endfunction : new
28
29         `uvm_component_utils_begin(ahb_pp_vir_sequencer)
30         `uvm_component_utils_end
31
32         //UVM connect_phase
33         function void connect_phase(uvm_phase phase);
34                 super.connect_phase(phase);
35
36                 if (!uvm_config_db#(virtual ahb_pp_int_if)::get(this, "", "vif",
   vif))
37                         `uvm_error("NOVIF",{"virtual interface must be set for:
   ",get_full_name(),".vif"})
38         endfunction : connect_phase
39
40 endclass : ahb_pp_vir_sequencer
```

Figure 4-40: AHB Parallel Port *Virtual Sequencer*

4.3.3 AHB GPIO Environment

In the AHB GPIO verification environment, the AHB UVC sets the register in the GPIO to send GPIO transfer. This transfer also writes to the configuration environment which will set the respective *Configuration Objects* of the GPIO transfer. The GPIO UVC is used to capture this transfer and compare it with the AHB transfer in the *Scoreboard*. These transfers are sent to the *Scoreboard* to check for correctness of the transfer. Figure 4-41 shows our AHB GPIO verification environment setup. The DUT is attached to the verification environment through the AHB interface on one end. On the other end, the DUT is attached to the GPIO interface.

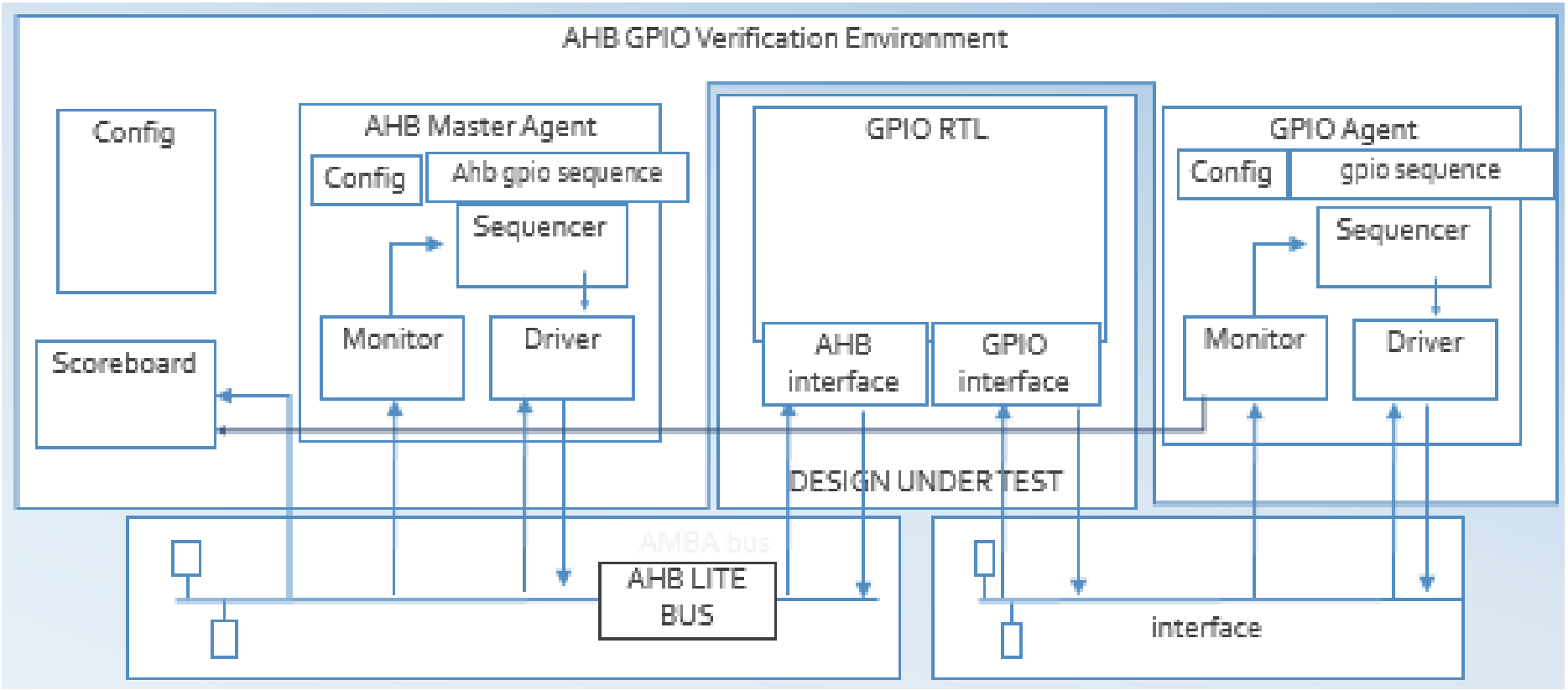


Figure 4-41: AHB GPIO Verification Environment

Since we need the AHB and GPIO UVC to work together, the verification environment uses its *Virtual Sequencer* to load the AHB and GPIO *Sequences* to the corresponding *Sequencers* in the AHB and GPIO UVC as shown in Figure 4-42.

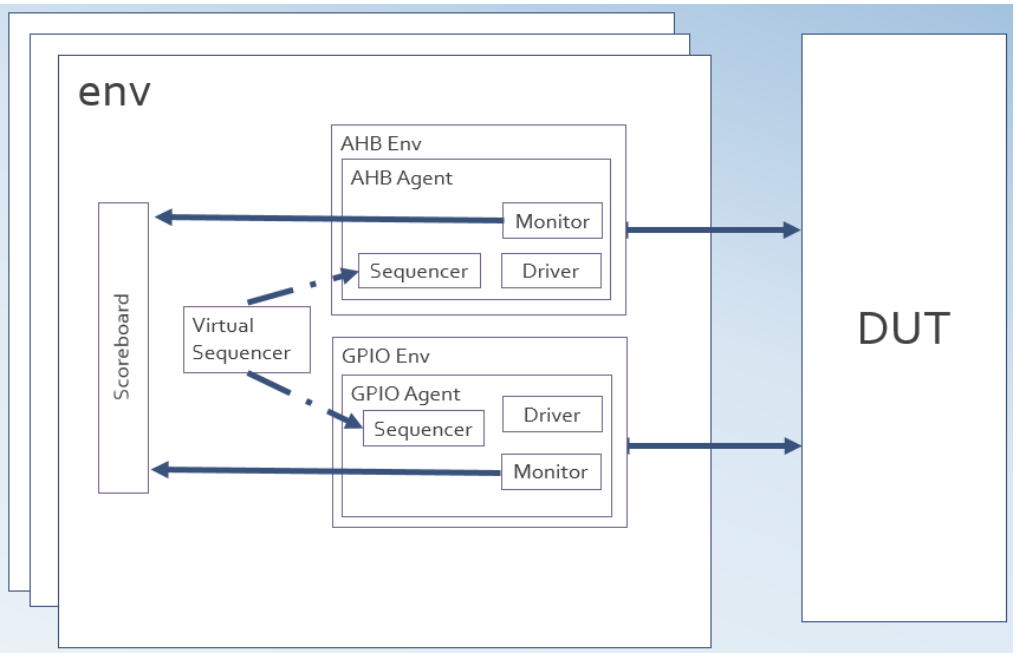


Figure 4-42: *Virtual Sequencer* and *Analysis port* for AHB GPIO

The implementation of the AHB GPIO *Virtual Sequencer* is similar to the *Virtual Sequencer* that has been discussed for the AHB2NoC verification environment in Section 4.3.1 as well as for the AHB Parallel Port in Section 4.3.2.

4.3.4 AHB APB Subsystem Environment

The AHB and APB bus is connected using the AHB2APB Bridge. This bridge converts the AHB-Lite bus to the APB bus signals. Due to the bus conversion from a pipeline transfer in the AHB-Lite to a non-pipeline bus for the APB bus, a minimum of 3 cycle is required for the APB bus. In Figure 4-43, we have an overview of our AHB-APB subsystem verification environment which uses the AHB and APB environments.

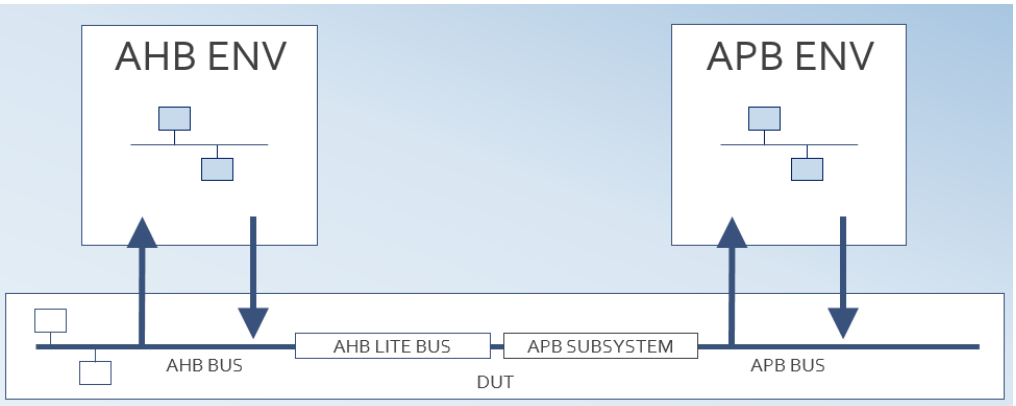


Figure 4-43: AHB APB Bridge Verification Environment Overview

The AHB2APB bridge verification environment is shown in Figure 4-44. The DUT in this case is the AHB2APB Bridge

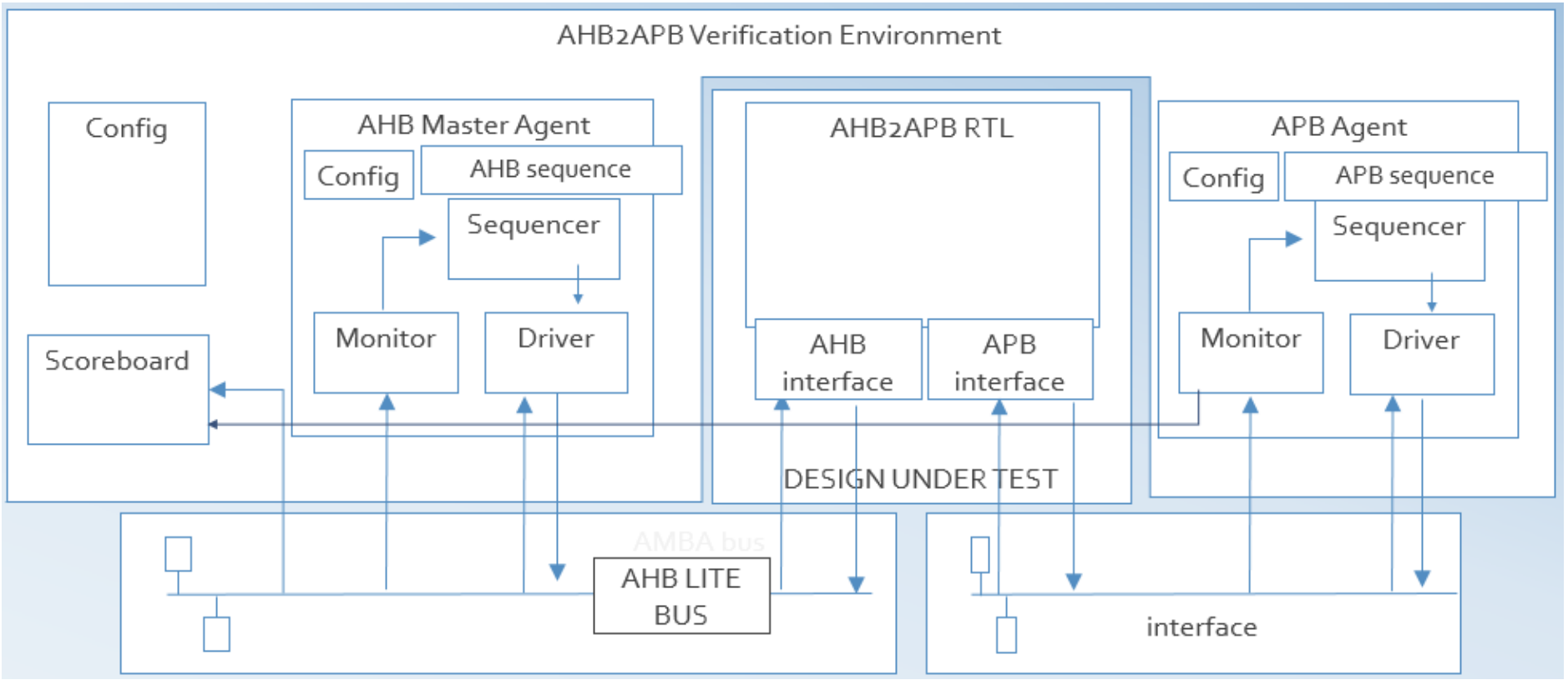


Figure 4-44: AHB2APB Bridge Verification Environment

In this verification environment, the AHB master will send write transactions to the APB subsystem through the AHB APB Bridge. The AHB master will try to write to the APB memory slave. The Figure 4-45 shows the timing for the AHB to write to the APB bus.

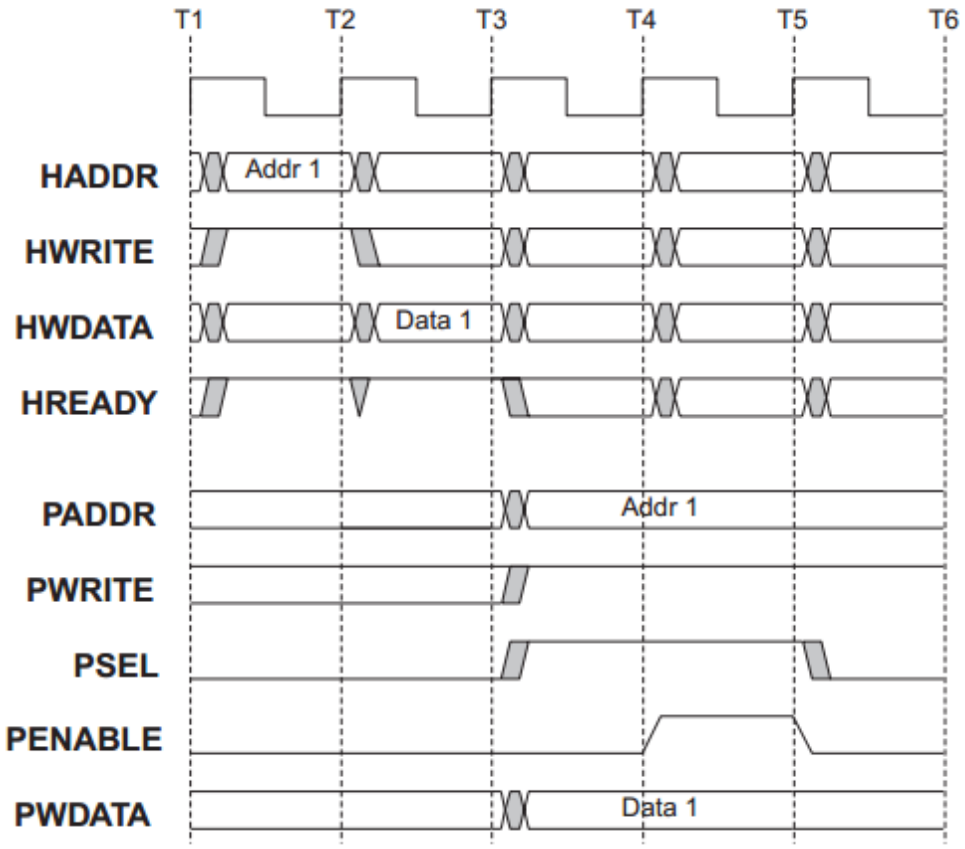


Figure 4-45: AHB APB transfer timing diagram

The APB slaves will then loop back the previous write transactions when the AHB master generates a read transaction as illustrated in Figure 4-46. The APB slave data in *PRDATA* can be seen in the *HRDATA* during the read transfer.

.

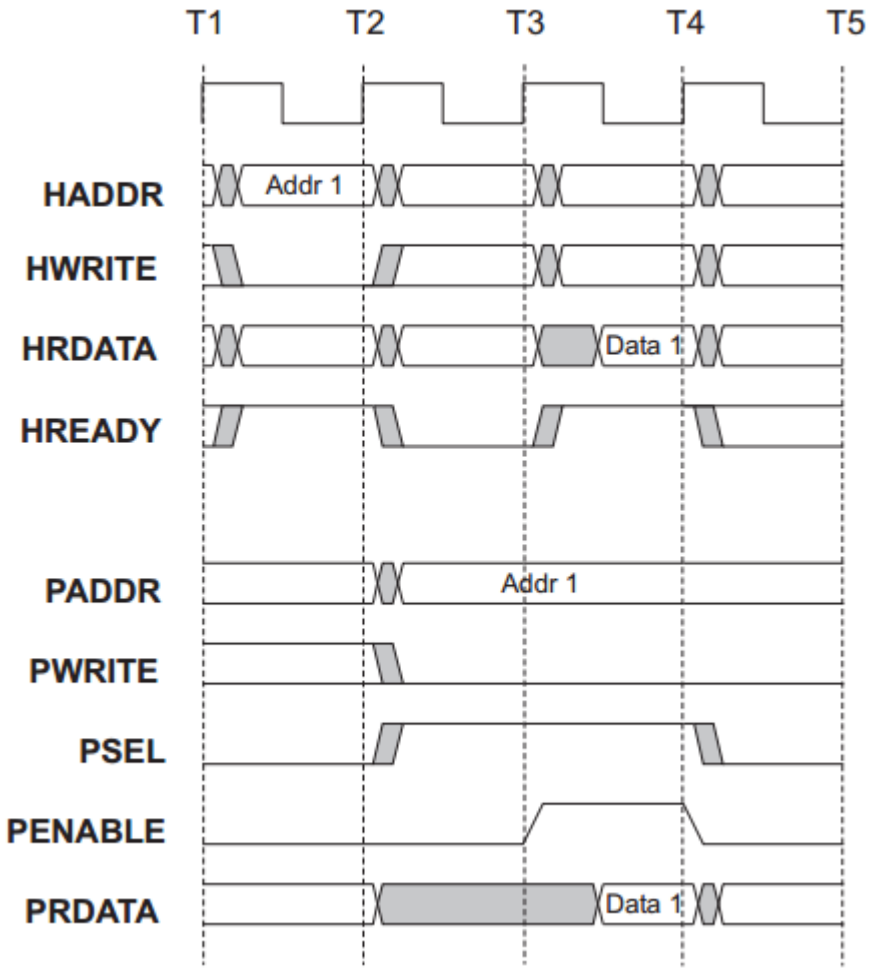


Figure 4-46: AHB APB read timing diagram

The AHB and APB environment as well as the AHB-Lite bus and APB subsystem has previously been verified using the corresponding modular verification environment. In the sub-system level, a *Virtual Sequencer* is used to coordinate AHB and APB *Sequence* as in Figure 4-47. Thus, component level environment can be reused without modification.

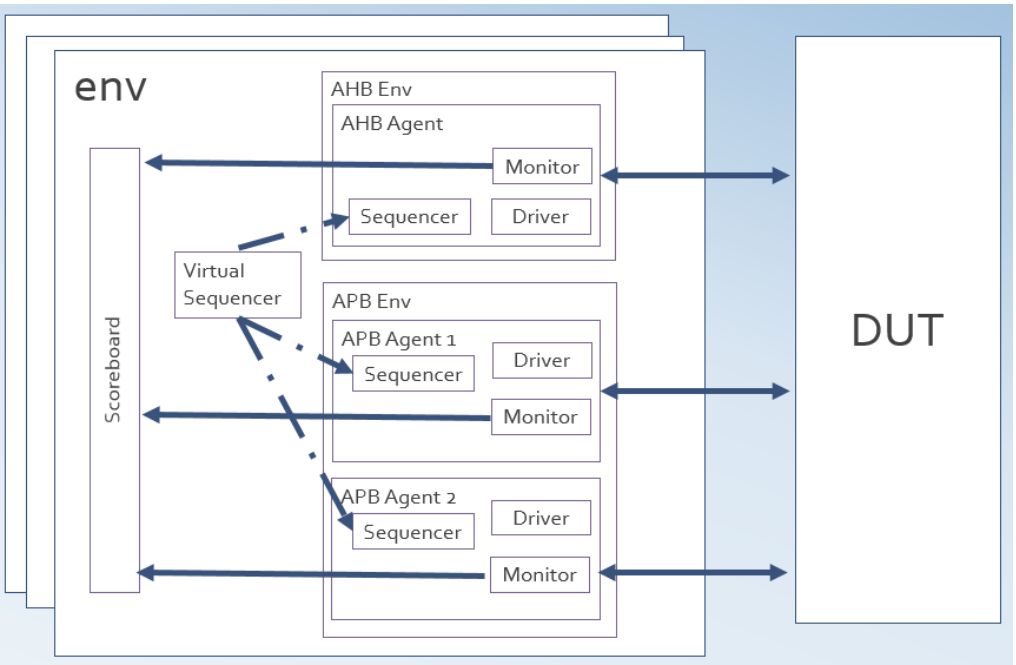


Figure 4-47: *Virtual Sequences* and *Analysis port* for AHB APB Bridge Verification Environment

4.3.5 APB SPI Environment

The APB SPI verification environment consist of the APB and SPI verification environment. The APB master *Agent* UVC is used to configure configuration registers in the SPI RTL module. This transfer will also be used to configure the *Configuration Objects* in the SPI UVC. The module will generate the SPI transfer according to the transfer configuration. The APB transfer, which is used to configure the configuration register is also used to configure the configuration of the SPI UVC. This is to ensure that the SPI RTL and the SPI UVC are using the same SPI protocol. Figure 4-48 shows the APB SPI verification environment.

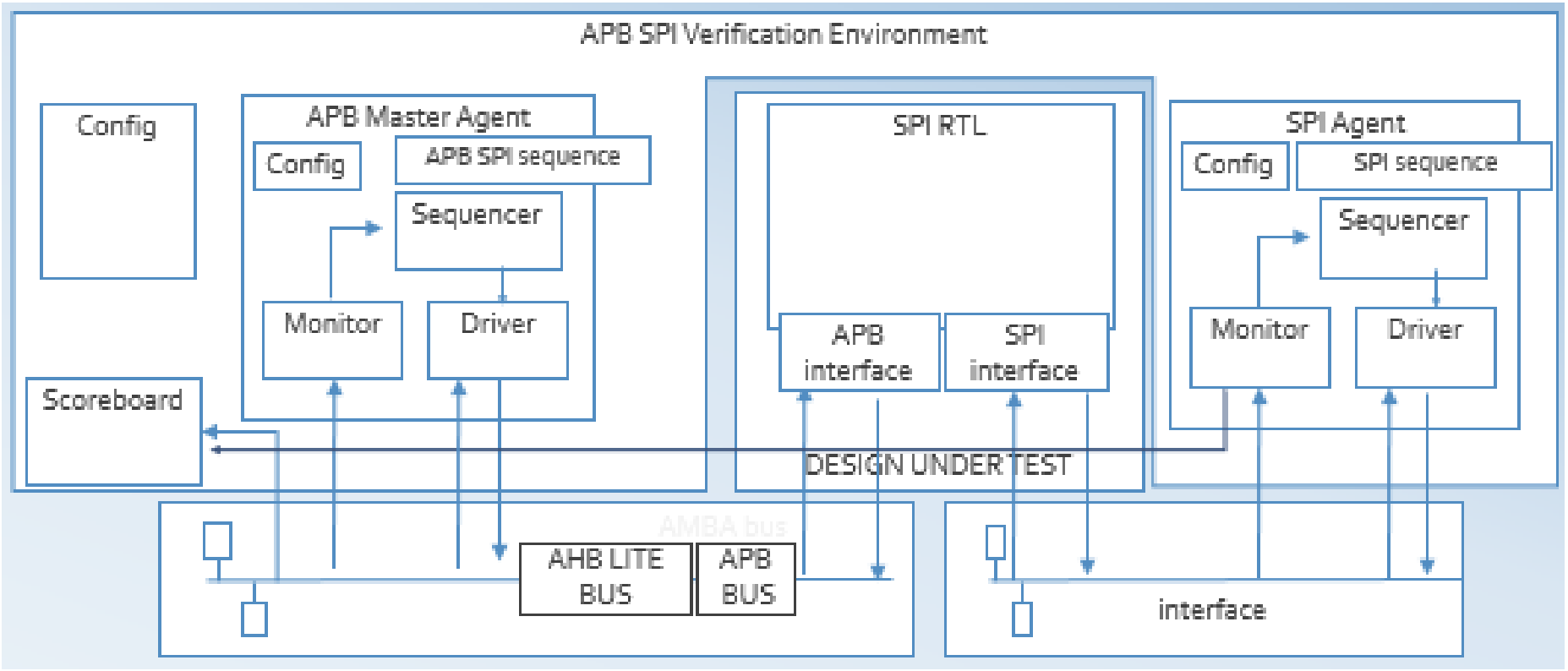


Figure 4-48: APB SPI Verification Environment

Virtual Sequencer is setup in this verification environment as shown in Figure 4-49. This is to allow the coordination between the APB and SPI *Sequences*. The *Monitor* in the APB and SPI environment monitors the activity on the bus connected to the DUT. These data gathered is sent to the subsystem *Scoreboard* to be compared.

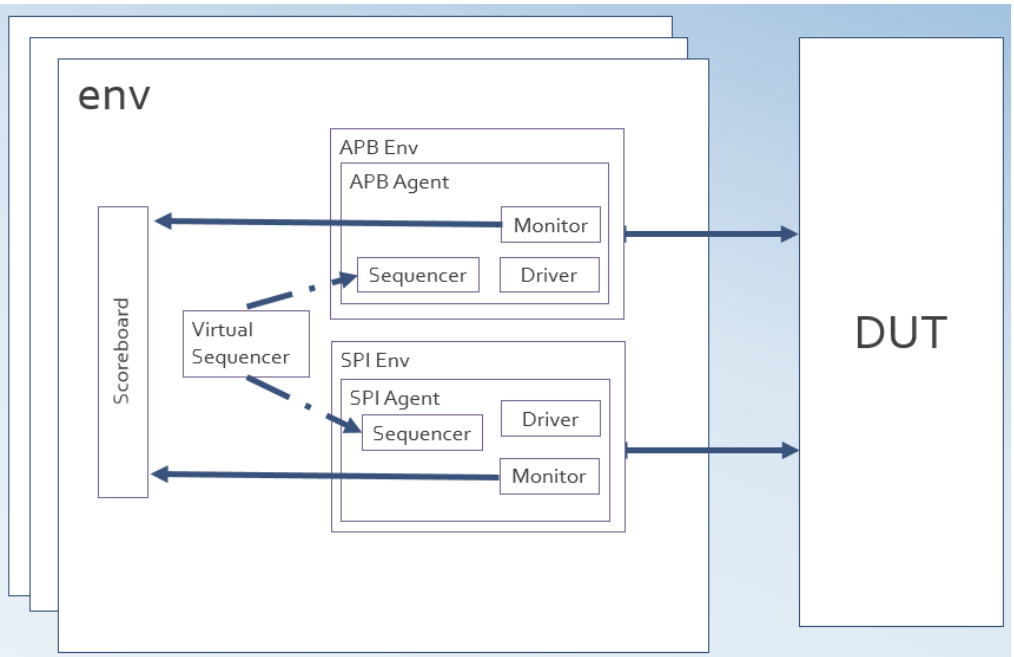


Figure 4-49: *Virtual Sequencer* and *Analysis port* for APB SPI

4.4 *System level Testbench*

The system level verification environment can then be built upon the multiple component and subsystem level verification environment. This section will discuss the setup of the system level verification environment using UVC and followed by implementation with Cortex M0 attached to our NoC system. This section will also elaborate on the firmware loading and the functionalities of the NoC system.

4.4.1 NoC System Level Environment

After the modular component of the system has been designed and verified correctly, these components will be integrated to form a system. To ensure that this system is working correctly, a system level testbench has to be designed. In our case, we reuse the components that we have developed from component and subsystem level. Depending on the network architecture in the System level verification, multiple NoC environment can be used to generate and receive transfers over the network. Furthermore, the *Configuration Objects* in each environment and its sub components allow each of them to be configured independently according to the needs of the verification. This allows the verification environments to be adaptive to the network architecture and scalable to the architecture as the network grows.

We have used CONNECT network generator to generate different network architectures for our simulations. For a 4-endpoint architecture, our NoC Env is configured to consist of 4 AHB environments, 1 parallel port environment, 1 APB environment and 1 NoC environment. This is illustrated in Figure 4-50.

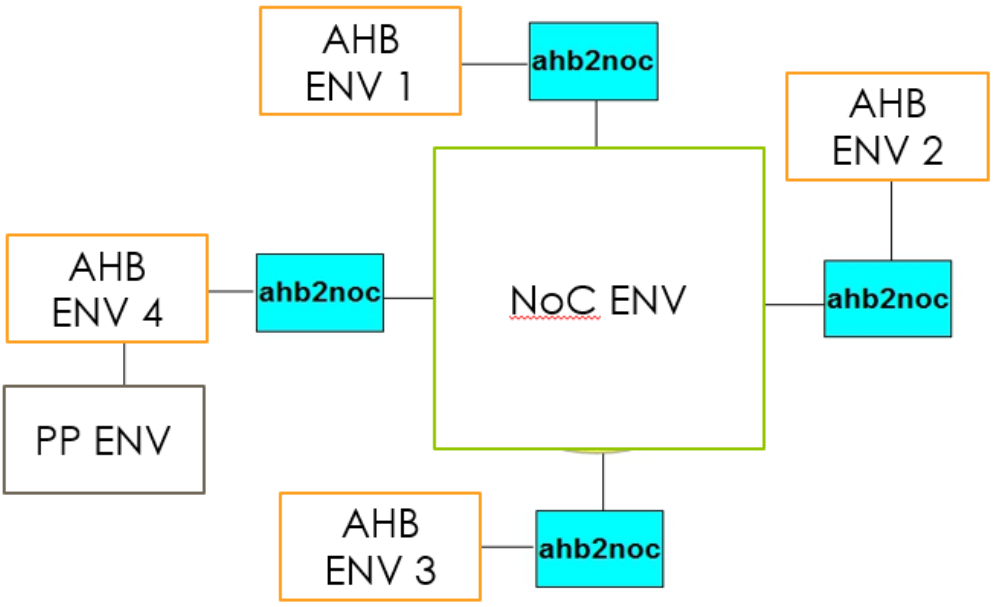


Figure 4-50: NoC System simulation using *Sequences*

The Parallel Port Master in a parallel port environment UVC is used to inject the plain text and key for the AES encryption to the NoC System. One of the AHB masters in AHB environment UVC will be used to read the data from the parallel port and write the plain text to one of the other 3 AHB masters to perform AES encryption. Once, the plain text is encrypted, the corresponding AHB master sends the encrypted data back to the AHB master which is attached to the parallel port to write the cipher text out.

To check for the correctness of the AES encryption, a golden reference model of the AES using the AES algorithm is modelled. The values generated from the AES golden reference model is compared with the values read from the AES RTL Design when the encryption has finished. Figure 4-51 shows the input implementation for the AES model. When the AHB transaction occurs to the corresponding AHB

addresses which corresponds to the AES module registers, the values are stored in its respective variable and *AES_encryption()* task is called.

```
109                                 AES_KEY0 = input_temp[0];
110                                 AES_KEY1 = input_temp[1];
111                                 AES_KEY2 = input_temp[2];
112                                 AES_KEY3 = input_temp[3];
113                                 AES_PLAIN0 = input_temp[4];
114                                 AES_PLAIN1 = input_temp[5];
115                                 AES_PLAIN2 = input_temp[6];
116                                 AES_PLAIN3 = input_temp[7];
117                                 AES_encryption();
118
119                                 cipher = {text[0][0],text[1]
    [0],text[2][0],text[3][0],text[0][1],text[1][1],text[2][1],text
    [3][1],text[0][2],text[1][2],text[2][2],text[3][2],text[0]
    [3],text[1][3],text[2][3],text[3][3]};
```

Figure 4-51: AHB Plain Text Input

The implementation of this task is shown in Figure 4-52. The task will get the key and plain text for encryption. These are arranged in 2D matrix or *state* for the AES encryption by making use of the *encrypt_block()* task.

```
284 task ahb_aes1_system_scoreboard::AES_encryption;
285         for(int i = 0; i < 4; i++)begin
286                 text[i][0] = AES_PLAIN3[(31-(((i+1)*8)-1)) +:
    8 ];
287                 text[i][1] = AES_PLAIN2[(31-(((i+1)*8)-1)) +:
    8 ];
288                 text[i][2] = AES_PLAIN1[(31-(((i+1)*8)-1)) +:
    8 ];
289                 text[i][3] = AES_PLAIN0[(31-(((i+1)*8)-1)) +:
    8 ];
290         end
291         for(int i = 0; i < 4; i++)begin
292                 key[i][0] = AES_KEY3[(31-(((i+1)*8)-1)) +: 8 ];
293                 key[i][1] = AES_KEY2[(31-(((i+1)*8)-1)) +: 8 ];
294                 key[i][2] = AES_KEY1[(31-(((i+1)*8)-1)) +: 8 ];
295                 key[i][3] = AES_KEY0[(31-(((i+1)*8)-1)) +:
    8 ];
296         end
297
298         encrypt_block();
299         cipher = {text[0][0],text[1][0],text[2][0],text[3]
    [0],text[0][1],text[1][1],text[2][1],text[3][1],text[0][2],text
    [1][2],text[2][2],text[3][2],text[0][3],text[1][3],text[2]
    [3],text[3][3]};
300 endtask: AES_encryption
```

Figure 4-52: AHB AES Reference Model

This task performs the AES encryption algorithm. The number of iterations required to perform the encryption depends on the number of bits for the AES. The 128-bit AES uses 10 rounds of iterations. In the *add_roundkey()* task, the plain text is combined with the key using the bitwise XOR function. The *sub_bytes()* task is used to perform substitution on each of these bytes using the *Rijndael_Sbox* lookup table.

```
263 task ahb_aes1_system_scoreboard::encrypt_block;
264         int count;
265         add_roundkey();
266         for (count = 0 ; count < 9 ; count++) begin
267                 key_schedule(count);
268                 sub_bytes();
269                 shift_row();
270                 mix_column();
271                 add_roundkey();
272
273                 `uvm_info(get_type_name(), $psprintf
    ("\n******************Round %d*************************\n",count+1), UVM_HIGH)
274
275         end
276         key_schedule(count);
277         sub_bytes();
278         shift_row();
279         add_roundkey();
280
281         `uvm_info(get_type_name(), $psprintf("\n******************Round %
    d*************************\n",count+1), UVM_HIGH)
282 endtask:encrypt block
```

Figure 4-53: AHB AES Encryption Block

For AES encryption, the first row remains unchanged. Depending on the row, n, the elements in that particular row is shifted left circularly by n-1 bytes using the *shift_row()* task. These new elements are then multiplied by a fixed matrix:

$$\boldsymbol{M} = \begin{bmatrix} \mathbf{2} & \mathbf{3} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \mathbf{3} & \mathbf{1} & \mathbf{1} & \mathbf{2} \end{bmatrix} \qquad \textbf{(Equation 4.1)}$$

Multiplication of the element by 1 means no change to the location of that element. If multiplication by 2 is done, this means the element shifts to the left. When multiplication by 3 is performed, these elements should be shifted to the left and XOR is performed with the initial un-shifted value. If the shifted values are larger than 0xFF, then, a conditional XOR with 0x1B is performed.

At every round of iteration, a new *subkey* is produced. These *subkeys* are combined with the partially encrypted plain text using bitwise XOR until the encryption is completed. This is implemented in the *add_roundkey()* task. The *mix_column()* task is omitted during the last iteration to simplify the decryption process.

After the environment has been setup and verified to be able to monitor the correctness of the transfer, the M0 core is attached to the system. The verification environments which previously is used to generate the stimulus for the DUT is not removed. Instead, they are used to monitor the system for the correctness of the transfer. The M0 core will generate the stimulus according to the firmware it is running.

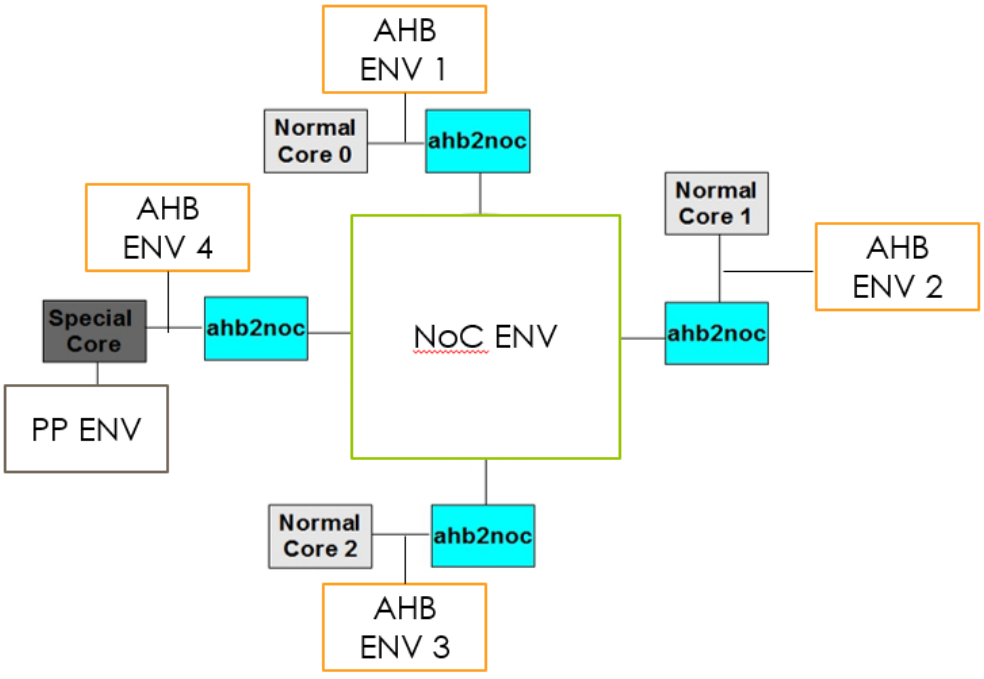


Figure 4-54: Cortex-M0 replacing the *Sequences*

The firmware for the Cortex-M0 is written in C and compiled using CodeSourcery. The compiled firmware is loaded into the program memory for each M0 through the verification environment. The encrypted cipher text from the AES that is sent out through the parallel port is monitored by the parallel port UVC. The result is logged into a debug log file.

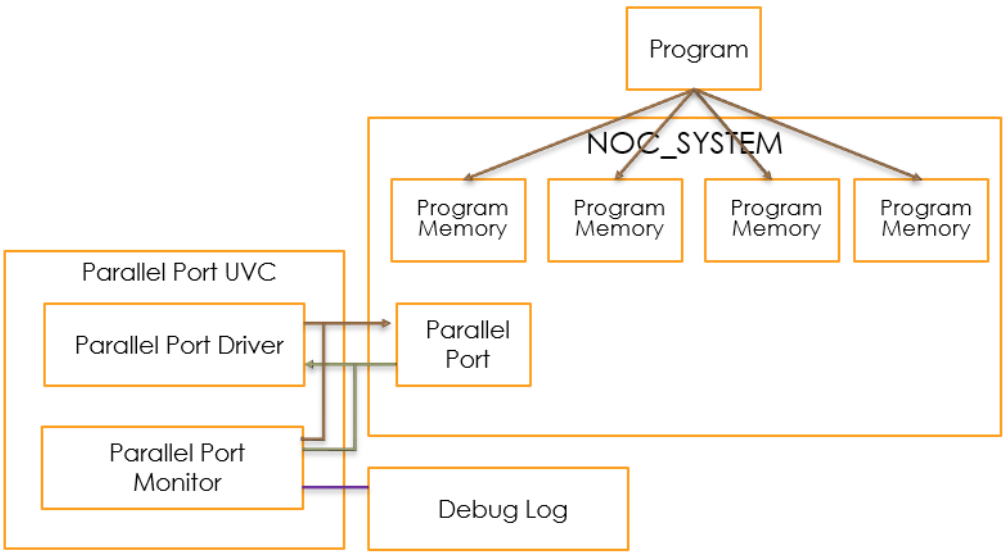


Figure 4-55: Loading firmware to the Cortex-M0

4.5 *Summary*

This chapter has discussed on the architecture of our verification environment which we have built hierarchically. We started off by designing the component level verification environment for AHB, APB, GPIO, SPI, and CONNECT network. We utilize these primitive environment in our subsystem environments such as AHB GPIO, AHB2APB Bridge, AHB2NoC and APB SPI. The system level

verification environment can then be formed from multiple subsystem level and component level verification environment. The following chapter shall discuss on the results from the exploration of our different NoC architectures.

CHAPTER 5

RESULTS AND DISCUSSIONS

This Chapter shall discuss the results of our verification platform for our NoC architecture exploration. The platform can be separated into fixed and variable components. The fixed components are the AHB, and APB system modules in the architecture as well as the basic *Sequences* for stimulus generation. For the variable components, it consists of the router architectures and the number of modular verification environments that can be instantiated in the system level environment. In the following section, the verification strategies for verifying the various UTAR NoC architectures using the platform will be discussed. This includes the verification plans for the modules of the UTAR NoC, the simulation results and the self-checking verification environment scoreboards that we have developed to assist in the verification process.

5.1 *Verification Plans*

The foremost purpose in any design verification is to achieve full and complete functional coverage. A verification plan is necessary to serve as a guideline and checklist in developing our test cases for verifying the corresponding modules. In order to maintain clarity and not cloud things with the overwhelming details only the functions of the AHB2NOC Bridge, various router architectures and APB subsystem modules and their verification plan will be discussed in the following subsections, as illustrations. The discussion for the rest of the modules in the system is included in APPENDIX A.

5.1.1 AHB2NOC Bridge

The AHB2NoC Bridge converts an AHB data transfer to a network data flit and also from the network flit to the AHB transfer. For each router node a dedicated flit transmit and receive buffer is created. The transfer is sent through the AHB to the corresponding transmit buffer. The transfer or flit is received by another AHB2NoC bridge at the destination. If the transmit interrupt is enabled, the transmit interrupt will be triggered upon sending the flit. Upon receiving the flit, the new transfer flag is set. The received interrupt can also be triggered if the received interrupt is enabled. The bridge also allows the bridge to receive multiple flits before triggering the interrupt. The received flit will be stored in the receive buffer location that denotes the sender router. When the buffer in the router and the bridge buffers are full, the transfers that follow will be discarded.

The Table 5-1 summarizes the AHB2NoC Verification plan. The first test scenario is to ensure the AHB2NoC Bridge is able to cast the transfer from the AHB to flit and vice versa. Another test is developed to ensure that the bridge only generates the interrupt after the number of transfers defined. Since the bridge also contains transmit and receive buffers, it is also necessary to verify that the buffer full flag is set and the transfers following does not override the buffer content. Polling method is also used as a test to gather the new transfer from the bridge.

Table 5-1: AHB2NoC Verification Plan

Tests	***Test Descriptions***	***Verification Criteria***
ahb2noc_special_test	Enable transmit and receive interrupt. Send eight flit from AHB to router and receive eight credit from the router and compare the transfer	Expects the AHB2NoC bridge is able to send ahb and flit correctly and vice versa
ahb2noc_n_flit_special_test	Enable transmit and receive interrupt. Enable multiple receive interrupt. Send sixteen flit from AHB to router and receive sixteen credit from the router and compare the transfer	Expects that the receive interrupt is set after n_flit transfer.
ahb2noc_transmit_buffer_overrun_test	Transmit 34 bit flit until transmit buffer overrun and check for buffer full flag	Expects that the data in the transmit buffer is not overwritten by the new transfer after the transmit buffer is full
ahb2noc_receive_buffer_overrun_test	Receive 34 bit flit until receive buffer overrun and check for the buffer full flag	Expects that the data in the receive buffer is not overwritten after the receive buffer is full
ahb2noc_polling_test	Check for buffer receive flit flag and compare the transfer	Expect that the flit receive flag is triggered upon receiving a new flit
ahb2noc_random_test	Randomly transmit and receive flit with random interval delay	Expects the correct transfer that is send and receive within different time intervals

5.1.2 NOC Router

For the NoC Router, we will use the ring architecture as an illustration.

5.1.2.1 Ring architecture

In Table 5-2, the NoC Ring Router verification plan is summarized. The router routes the transfer from one of the core to another. The ***noc_ring_34_test*** sends the flit stimulus from one node to another node of the router. This is used to verify the send and receive of the router. To ensure that collision does not occur in the network, ***noc_ring_34_collision_test*** has been setup. This test starts sending from all the router nodes simultaneously and monitors the incoming flit.

Table 5-2: NoC Verification Plan

Tests	***Test Descriptions***	***Verification Criteria***
noc_ring_34_test	34 bit data transfer is sent from one node to another in the 4 node ring router	Expects the router to send and receive the flit correctly.
noc_ring_34_collision_test	Send 34 bit transfer from all 4 node of the router simultaneously	Expects the flits send by the router does not collide with the flit that is sent from another router
noc_ring_34_random_transfer_collision_test	Send 34 bit transfer from random routers for collision detection	Expects no collision to occur when randomly send and receive from the routers

5.1.3 APB Subsystem Environment

The APB Subsystem provides a lower bandwidth bus for slower peripherals. In our system, only the SPI module is attached to the APB bus. Hence, the main focus is to verify the APB SPI module in this case. The SPI module is able to generate maximum of 128bit of SPI transfer.

The basic test case for the SPI is to read and write through the SPI correctly is shown in Table 5-3. Other test cases include generating various speed and transfer size of the SPI transfer. Illegal range of speed and data sizes test has also been setup to ensure that the invalid transfers are not driven to the interface.

Table 5-3: APB Subsystem Verification Plan

Tests	**Test Descriptions**	**Verification Criteria**
apb_spi_read_test	Set the SPI module to output SPI transfer 128bit and the UVC to read the transfer	Expects correct SPI transfer from the module
apb_spi_write_test	Set the SPI to output SPI transfer 128bit and the UVC to loopback the transfer.	Expects the module to read the correct SPI transfer
apb_spi_random_test	Set the SPI module to send constrained random transfer size and transfer mode	Expects correct transfer with various sizes from the module
apb_spi_random_div_test	Set the SPI module to send constrained random transfer speed and the delay between each transfer	Expects the module to send the SPI transfer with various speeds
apb_spi_illegal_transfer_size_test	Set illegal SPI transfer size	Expects the module not to drive any transfer
apb_spi_illegal_speed_test	Set SPI divisor register beyond the range	Expects the module not to drive any transfer

The next section shall discuss the stimulus generation mechanism that we have developed.

5.2 *Verification environment stimulus generations*

The *Sequences* or stimulus generation mechanism described in this section form the basis test template of each of the verification environments. These *Sequences* has been used and extended to generate the test cases that has been described in the verification plan subsection earlier. Some of these *Sequences* which models a more complex scenario consist of tasks so that it can be utilized when it is used in the higher level *Sequences*. During the discussion later, a few examples are given to illustrate the usages of some of these *Sequences*.

5.2.1 NoC Sequences

Table 5-4 summarizes the NoC Master Basic *Sequences* that model the client for the network. The ***send_flit*** *Sequence* can be used to generate the flit for the router. In this *Sequence,* the parameters that can be constrained are ***valid***, ***is_tail, dst, vc*** and ***data***. This ***valid*** bit is used to indicate that the generated flit is valid. The ***is_tail*** parameter is used to indicate that this is the last flit while ***dst*** allows the flit to be sent to different locations. The ***vc*** is the virtual channel that the flit is used to send through the network. Since the network is credit based, the ***send_credits*** *Sequence* is created to send credit to the router. In this *Sequence* the ***valid*** and ***vc*** bit can be constrained. This ***valid*** bit is used

to indicate a valid credit while the ***vc*** bit indicates the virtual channel for the incoming credit. The ***ask_credit*** *Sequence* can be used to request credit from the router. In order to ease the use of the basic *Sequences*, the ***send_flit_ask_credit*** *Sequence* combines the functionality of the ***send_flit*** and ***ask_credit*** *Sequences*.

Table 5-4: NoC Master Basic *Sequence*

NoC Master Basic *Sequence*	**Description**
send_flit	*Sequence* to generate flit to the router
ask_flit	*Sequence* to request flit from the router
send_credits	*Sequence* to send credits to the router
ask_credit	*Sequence* to ask credit from the router
send_flit_ask_credit	*Sequence* that sends flit and request credit from the router

To illustrate the usage of these *Sequences*, Figure 5-1 shows part of the ping_pong_agent_seq. From the figure, the ***send_flit_seq*** *Sequence* is an instance of ***send_flit.*** The parameters inside this *Sequence* can be constrained. For this example, the flit is set to valid and this flit is the last flit of the transfer. The data for this flit is 1.

```
65          send_flit send_flit_seq;
66          send_credits send_credits_seq;
67          ask_flit ask_flit_seq;
68          ask_credit ask_credit_seq;
69
70          rand logic select_seq;
71          rand int num_of_routers;
72          int recv_data;
73          int id;
74
75          virtual task body();
76                  id = p_sequencer.id;
77                  `uvm_info(get_type_name(), $psprintf("ID: %0d", id), UVM_LOW)
78
79                  if (select_seq == 1) begin      // initial flit
80                          `uvm_do_on_with(send_flit_seq, p_sequencer.s_sequencer,
81                                  {       valid   == 1;
82                                          is_tail == 1;
83                                          dst     == (id+1)%num_of_routers;
84                                          vc              == 0;
85                                          data    == 1;}); // sending flit for the
   first time.
86
87                          fork
88                                  `uvm_do_on(ask_credit_seq,
   p_sequencer.s_sequencer); //waiting for credit after sent the flit.
89                          join_none
90                  end
```

Figure 5-1: Part of ping_pong_agent_seq

The NoC Slave Basic *Sequences* models the router is listed in Table 5-5. The ***slave_send_flit*** *Sequence* is used to send flit to the client. Similarly, the ***slave_send_credits*** *Sequence* sends the credit to the client and the ***slave_ask_credits*** *Sequence* request the credit from the client. The ***slave_send_flit_ask_credit*** *Sequence* allows the flit to be sent to the client and request the credits from the client. This *Sequence* merges the ***slave_send_flit*** *Sequence* with ***slave_ask_credit*** *Sequence*.

Table 5-5: NoC Slave Basic *Sequence*

NoC Slave Basic *Sequence*	**Tasks**	**Description**
slave_send_flit		*Sequence* to generate the flit for the router client
slave_send_credits		*Sequence* to generate the credit for the router client
slave_ask_credit		*Sequence* to ask the credit for the router client
slave_send_flit_ask_credit	noc_slave_send_flit(valid, is_tail, vc, data, delay)	*Sequence* to generate the flit for the router client and request credit
slave_ask_flit_send_credit	send_credits(flit, num_valid, int num_vc, int num_credit_delay)	*Sequence* to request the flit from the router client and send the credit to the router client
	noc_slave_ask_flit(int num_flit, int num_valid, int num_vc, int num_credit_delay)	

An ***ahb2noc_special_slave_send_flit*** *Sequence* is extended from one of the NoC Slave Basic *Sequences* as shown in Figure 5-2. The task ***noc_slave_send_flit()*** has been utilized to generate the flit. In addition, constraints are added to the destination and source bits in the flit so that the source and destination is limited to sixteen.

```
15 class ahb2noc_special_slave_send_flit extends slave_send_flit_ask_credit;
16
17         rand logic [`FLIT_DST_SIZE-1:0] dst;
18         rand logic [`FLIT_DATA_SIZE-1:0] data;
19         rand int delay;
20
21         constraint src {data[`FLIT_DATA_SIZE-1:`FLIT_DATA_SIZE-`FLIT_DST_SIZE]
   inside {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}; }
22         constraint flit_dest {dst inside {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}; }
23
24         function new(string name="ahb2noc_special_slave_send_flit",
25                 uvm_sequencer_base sequencer=null,
26                 uvm_sequence parent_seq=null);
27                 super.new(name);
28         endfunction : new
29
30         // Register sequence with a sequencer
31         `uvm_object_utils(ahb2noc_special_slave_send_flit)
32         `uvm_declare_p_sequencer(noc_slave_send_sequencer)
33
34         virtual task body();
35                 //valid, is_tail, dst,vc, data, delay
36
37                 noc_slave_send_flit(1,  1,dst,  0,  data,  delay);
38
39         endtask: body
40
41 endclass: ahb2noc special slave send flit
```

Figure 5-2: ahb2noc_special_slave_send_flit *Sequence*

5.2.2 AHB

Table 5-6 shows the AHB Master Basic *Sequence* that allows the AHB Master transfer to be modelled. These *Sequences* provides various types of AHB transfers – bytes, half word, word, double word, four word, un-pipelined, un-aligned.

Table 5-6: AHB Master Basic *Sequence*

AHB Master Basic *Sequence*	**Tasks**	**Description**
ahb_seq	wait_response(ahb _transfertx)	The task to wait for the response from the slave
	wait_response_mul (ahb_transfertx, bit last_trans, int index, ref bit[`BUS_SIZE-1:0] data_array[])	The task to wait for multiple response from the slave
	monitor_reset_erro r()	The task to monitor reset and error on the bus
ahb_single_seq		*Sequence* to generate idle and non-sequential AHB transfer
ahb_wait_read_data_seq		*Sequence* to wait for the data ready to be read
ahb_single_unpipelined_seq		*Sequence* to send non pipelined AHB transfer
ahb_doubleword_seq		*Sequence* to send AHB double word size transfer
ahb_fourword_seq		*Sequence* to send AHB four word size transfer
ahb_unalined_seq		*Sequence* to send unaligned AHB transfer
ahb_incr_seq		*Sequence* to incremental AHB transfer
ahb_incr_n_seq		*Sequence* to send multiple incremental AHB transfer
ahb_wrap_n_seq		*Sequence* to send four, eight or sixteen incremental AHB transfer
ahb_valid_seq		*Sequence* to send random valid AHB transfer
ahb_unpipelined_seq		*Sequence* to send non pipelined AHB transfer
basic_multiple_seq		*Sequence* that combines the various sequences that generates the AHB transfers

As an example to use the AHB Master Basic *Sequence*, the ***ahb_pp_single_seq*** is extended from the ***basic_multiple_seq*** *Sequence.* Part of this *Sequence* is shown in Figure 5-3. ***single_non_seq*** *Sequence* in the ***basic_multiple_seq*** *Sequence* has been used in this example. The ***pp_direction, pp_address, pp_data***, and ***pp_size*** parameters can be used to constraint the *Sequence*. The ***pp_direction*** parameter allows either a read or a write transaction to be generated. The ***pp_addess*** parameter allows the transaction to be generated for the corresponding address that is defined while the data for that transaction can be defined in the ***pp_data*** parameter. The size of each transfer can be defined using ***pp_size***.

```
18 class ahb_pp_single_seq extends basic_multiple_seq;
19
20         rand bit               pp_direction;
21         rand bit [31:0] pp_address;
22         rand bit [31:0] pp_data;
23         rand bit [2:0]  pp_size;
24
25         function new(string name="ahb_pp_single_seq",
26                 uvm_sequencer_base sequencer=null,
27                 uvm_sequence parent_seq=null);
28                 super.new(name);
29         endfunction : new
30
31         // Register sequence with a sequencer
32         `uvm_object_utils(ahb_pp_single_seq)
33         `uvm_declare_p_sequencer(ahb_master_sequencer)
34
35         virtual task body();
36                 `uvm_info(get_type_name(), $psprintf("ahb ahb_pp_single seq
   running, pp_direction: %0d, pp_address: 0x%0h, pp_size: %0d, pp_data: 0x%0h",
   pp_direction, pp_address, pp_size, pp_data), UVM_HIGH)
37
38                 //-------------------------------------------------------------
39                 // one transfer
40                 //-------------------------------------------------------------
41                 if(pp_direction == WRITE)
42                         single_nonseq(pp_direction, pp_address, pp_size,
   pp_data);
```

Figure 5-3: Part of ahb_pp_single_seq *Sequence*

The AHB Slave Basic *Sequences* listed in Table 5-7 can be used as a generic AHB slave model. One of the functions in this *Sequence* is to wait for the slave data to be ready using the ***wait_data_ready()*** task. The AHB control signals for the from the AHB master can be retrieved by using ***wait_control_ready()*** task. The ***driving_driver()*** allows the model to drive AHB transfers with errors and also extends the transfer by de-asserting the HREADY signal through the ***hready_duration*** parameter.

Table 5-7: AHB Slave Basic *Sequence*

AHB Slave Basic *Sequence*	**Tasks**	**Description**
ahb_slave_basic_seq	wait_data_ready(ahb_transfer rx, ahb_transfer prev_rx)	Task in the *sequence* to wait for the slave data
	wait_control_ready	Task in the *sequence* to wait for the AHB control signals
	driving_driver(int hready_duration, int error, logic [31:0] hrdata)	Task to drive transfers to the AHB bus

Figure 5-4 shows the ***ahb_lite_bus_sw_ahb_slave_seq1*** which is used as a model for one of the AHB slaves. The ***driving_driver()*** task has been utilized. There are 3 input parameters to this task: ***hready_duration, error,*** and ***hrdata***. The ***hready_duration*** parameter allows the transaction to be delayed for a defined cycles before driving the transaction to the DUT. In order to drive an error transaction, the

error parameter must be set. The data can be constrained using the ***hrdata*** parameter.

```
170 class ahb_lite_bus_sw_ahb_slave_seq1 extends ahb_slave_basic_seq;
171
172         function new(string name="ahb_lite_bus_sw_ahb_slave_seq1",
173                 uvm_sequencer_base sequencer=null,
174                 uvm_sequence parent_seq=null);
175                 super.new(name);
176         endfunction : new
177
178                 randc logic [31:0] hrdata;
179         // Register sequence with a sequencer
180         `uvm_object_utils(ahb_lite_bus_sw_ahb_slave_seq1)
181         `uvm_declare_p_sequencer(ahb_slave_sequencer)
182
183         task body();
184                 forever begin
185                         assert(this.randomize());
186                         driving_driver(0,0, hrdata);
187                         driving_driver(0,1, hrdata);
188                         driving_driver(0,0, hrdata);
189                         driving_driver(0,0, hrdata);
190                 end
191         endtask: body
192 endclass: ahb_lite_bus_sw_ahb_slave_seq1
```

Figure 5-4: ahb_lite_bus_sw_ahb_slave_seq1 *Sequence*

5.2.3 APB

The APB Master Basic *Sequences* in Table 5-8 can be used to generate various APB transfers. There are *Sequences* to perform byte and word read write *Sequence*. The ***read_after_write_seq*** allows a *Sequence* of write read transfer. By using the ***multiple_read_after_write_seq***, multiple write read transfer can be done. This *Sequence* extends the ***read_after_write_seq.***

Table 5-8: APB Master Basic *Sequence*

APB Master Basic *Sequence*	**Description**
read_byte_seq	*Sequence* to read a byte
write_byte_seq	*Sequence* to write a byte
read_word_seq	*Sequence* to read a word
write_word_seq	*Sequence* to write a word
read_after_write_seq	*Sequence* to perform write and read
multiple_read_after_write_seq	*Sequence* to perform multiple read and write

The APB Slave Basic *Sequences* is used to model the APB slave. The ***Simple_response_seq*** can be used to send simple transfer to the APB master. ***Mem_response_seq*** can also be used. This allows the write transfer from the APB master to be stored and the content can be read back during the read transfer. Table 5-9 summarizes these *Sequences*.

Table 5-9: APB Slave Basic *Sequence*

APB Slave Basic *Sequence*	**Description**
Simple_response_seq	*Sequence* that checks for the valid APB slave access and if the address is within the range, response to the transfer
Mem_response_seq	*Sequence* that loopback the APB data that is written to the slave model

5.2.4 SPI Sequence

In the SPI environment, the ***spi_base_seq*** implements the basic SPI read write *Sequence*. The ***Spi_incr_payload_seq*** allows the generation of continuous SPI transfer. Table 5-10 summarizes the list of SPI Basic *Sequences*.

Table 5-10: SPI Basic *Sequence*

SPI Basic *Sequence*	**Description**
Spi_base_seq	*Sequence* that generates basic SPI transfer
Spi_incr_payload_seq	*Sequence* that generates multiple SPI transfer

5.2.5 GPIO Sequence

The GPIO Basic *Sequence* is developed to generate the GPIO read write transfer. These *Sequences* is listed in Table 5-11. This implementation is done within the GPIO environment. ***Gpio_multiple_simple_trans*** extends the ***Gpio_simple_trans_seq*** to be able to generate multiple read write transfer.

Table 5-11: GPIO Basic *Sequence*

GPIO Basic *Sequence*	**Description**
Gpio_simple_trans_seq	*Sequence* that generates GPIO read write transfer
Gpio_multiple_simple_trans	*Sequence* that generates multiple write transfer

The performance evaluation for our various architectures will be discussed in the following subsections. The discussion compares the simulation results between the M0 cores and the AHB UVC model. The result from four, eight and sixteen router system will also be discussed.

5.3 *Performance Evaluation*

Using the UVC environment that we have setup, the *Sequence* to generate the traffic and estimate the performance of our system is designed. A simple scenario where transactions are initiated from the special core to one router followed by two and three router. This setup has been done using the four router system. Figure 5-5 shows the results from the simulation. It is notable that the latency for the initial transfer is 29 cycles for transfer to 1 and 3 routers. When transferring to 2 router the latency is 33 cycles. In 2 router system, it is expected that the latency will increase initially because at this time before the 1st transfer is read by the special core, the acknowledge flit for the first normal core is being read. For a 3 router system, then, the latency increase at every 4th transfer. There is no spike in between the 3 router system because the special core is programmed to send to all the 3 routers before reading the acknowledge flit.

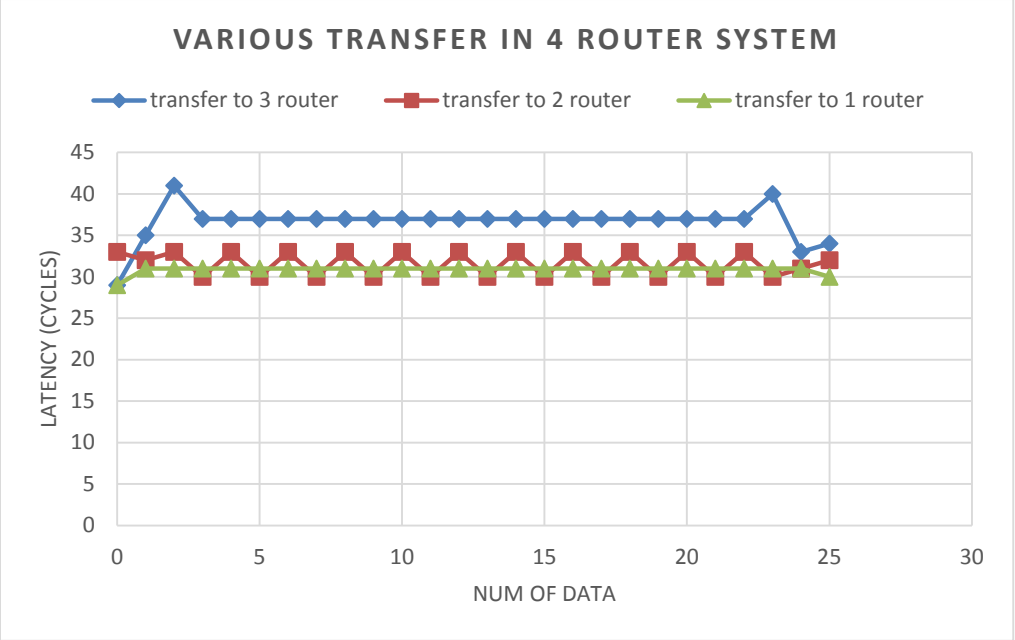


Figure 5-5: Various transfer in 4 router system

This simulation has been extended to evaluate the performance of the eight and sixteen router system. For the sixteen router system, the latency increases drastically after the 10^{th} data is sent. This is because at this stage, the data is already in the special core's buffers but the special core is receiving the acknowledge flit for the last transfer to be sent. The 8 router system on the other hand received the last acknowledge flit during its 19^{th} flit. Hence, there is a peak on the graph.

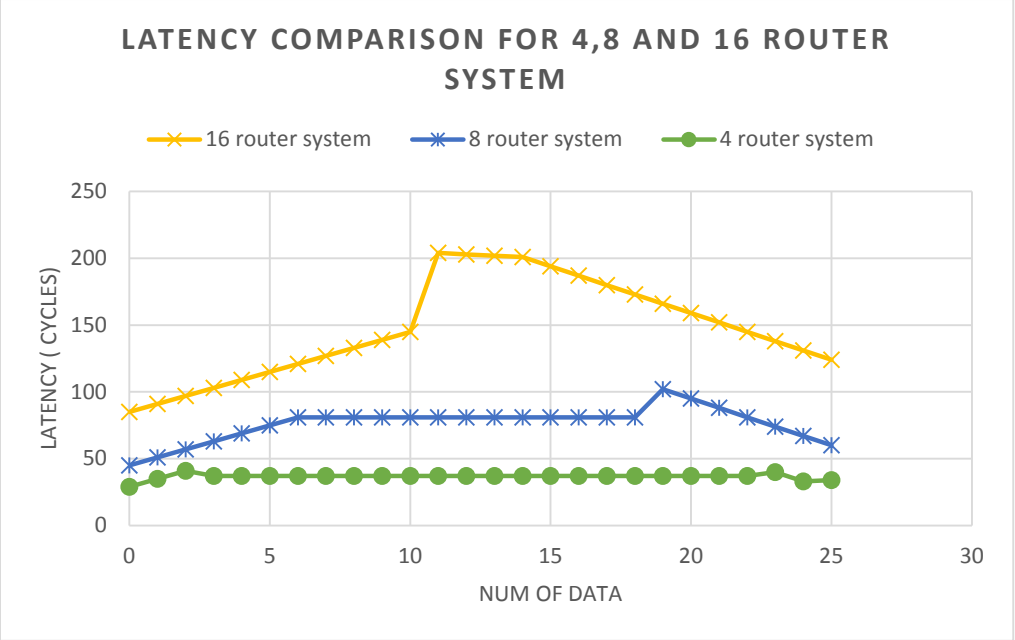


Figure 5-6: Latency comparison for 4, 8, and 16 router system

Figure 5-7 illustrate the results from configuring the Ahb2NoC Bridge in the 4 router system with buffer depth of 1, 2 and 5. As the buffer size increases the system throughput also increases. The system response with the Ahb2NoC Bridge configured with 5 buffers begins to stabilize after the 15^{th} data. This is because this fills up the 5 buffer depth in the corresponding Ahb2NoC Bridge which interfaces to the 3 normal cores. The system would be least efficient when the number of data to be encrypted is less than the buffer size of the system.

.

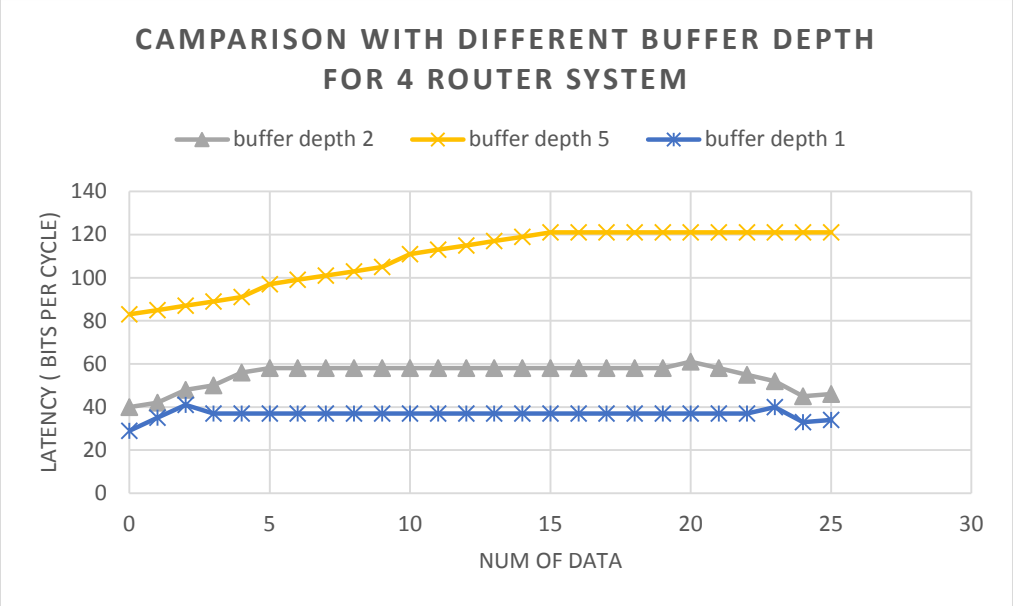


Figure 5-7: Various Buffer Depth for 4 router system

The throughput comparison for four router system with buffer depth 1, 2 and 5 is shown in

Table 5-12. The buffer depth 2 system has higher throughput compared to the buffer depth 5 system because the buffer depth 5 system reaches its steady state throughput after 15th data while the system with buffer depth 2 already stabilize during its 6th data. Once the amount of input data is higher than the time the system takes to stabilize then the throughput of the system with buffer depth 5 will be more than the system with buffer depth 2. The following section will discuss on the result of the system with 1kB, 2kB, 4kB and 8kB data size.

4 router system architecture	Throughput(bit per cycle)
Buffer depth 1	2.7826
Buffer depth 2	3.3147
Buffer depth 5	2.869

Table 5-12: Throughput for 4 router system with various buffer depth

The effect of the system with various data sizes has also been studied.

Figure 5-8 shows the throughput performance result when the system of different buffer depth is injected with various data sizes. The throughput difference between systems with buffer depth 5 and buffer depth 2 gives about additional throughput of 0.5 bit per cycle. However, increasing the buffer depth further to 10 results in about 0.2 bit per cycle.

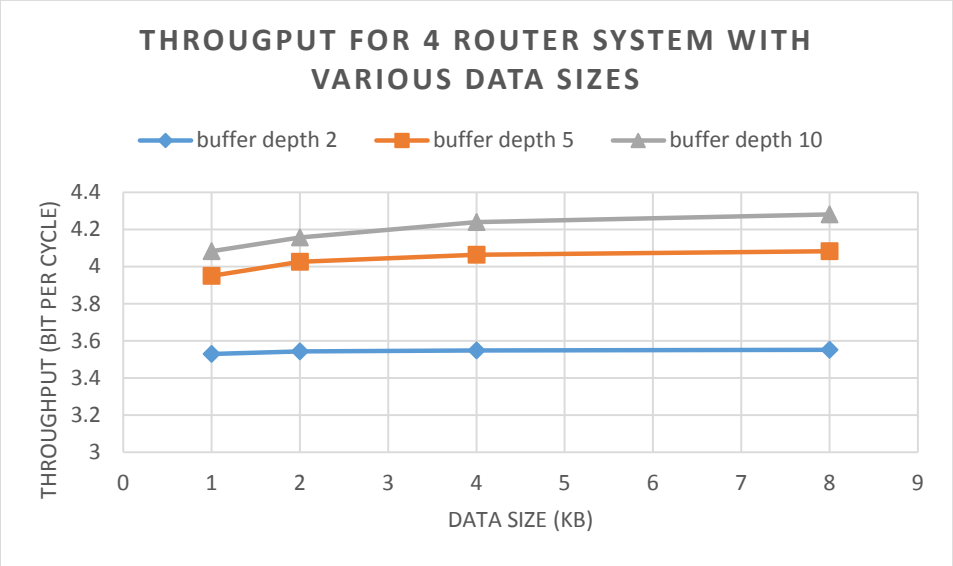


Figure 5-8: Throughput for 4 router system with various data sizes

The result of the latency of the system is shown in

Figure 5-9. By increasing the buffer depth from 2 to 5, the latency increases about 50 cycles. Increasing from buffer depth 5 to 10 further results in an increase of 100 cycle's latency. The system with buffer depth 5 in this case would be the most efficient because increasing the system buffer depth to 10 would result in insignificant throughput performance compared to the buffer depth 5 system performance. Furthermore, this will increase the latency and cost of the system as

bigger buffer size would contribute to bigger chip size and higher power consumption.

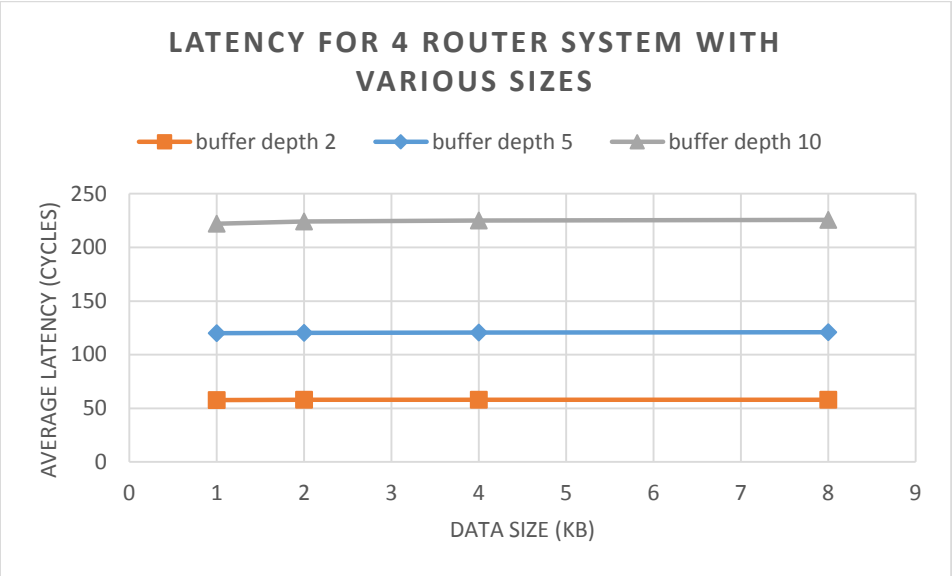


Figure 5-9: Latency for 4 router system with various data sizes

Figure 5-10 shows the result of the effects of various buffer depth on an eight router system. With buffer depth 1, the systems latency stabilizes after 8th transfer. However, it is notable here that the system with buffer depth 5 and 10 increases the latency until the 20th data. The system with buffer depth 5 stabilizes around 35th transfer which is the total amount of data required to fill up the buffers. On the other hand, buffer depth 10 system increases in latency before 10th data and more rapidly from the 15th to 20th data and decline slowly after the 20th data. As the buffer size increases, the latency increases as well. This is because with more buffers, the system have to wait for the buffers to get filled up before initiating the transfer.

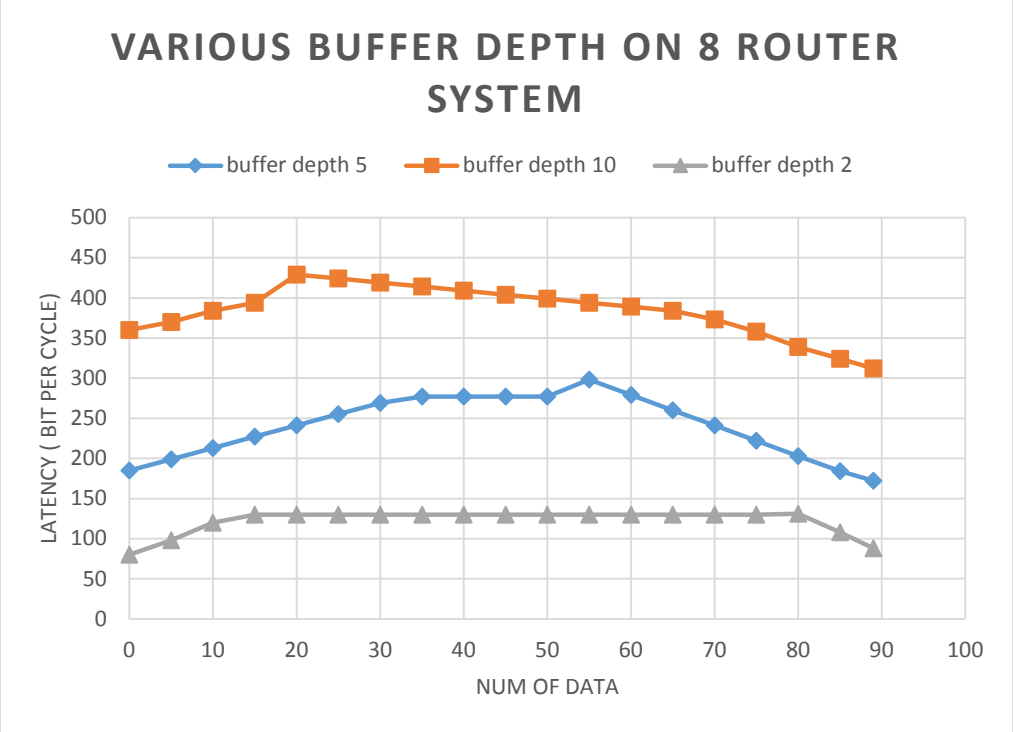


Figure 5-10: Various Buffer Depth for eight router system

Further investigation on the effect of various data sizes is perform to evaluate the performance of the system. The result is illustrated in Figure 5-11. With data size less than 2kB, a system with buffer depth 5 outperformed a system with buffer depth 10. This means that increasing the buffer depth at this stage would reduce the throughput performance instead. When the data size is 2kB, the throughput performance of a system with buffer depth 10 is similar to a buffer depth 5 system. From 4kB data size onwards, the throughput difference between a buffer depth 5 and 10 system is about 0.1 bit per cycle.

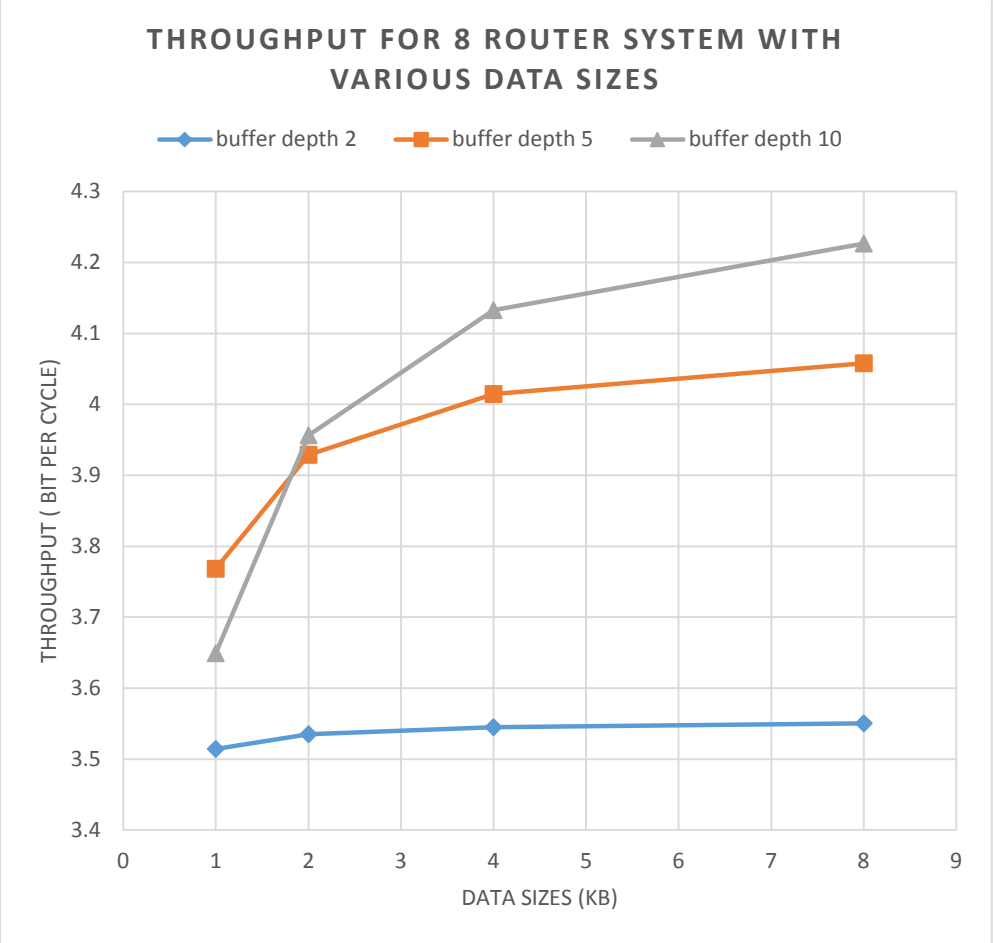


Figure 5-11: Throughput for eight router system with various data sizes

Figure 5-12 shows the latency of the eight router system with various data sizes. The latency between system with buffer depth 2 and 5 is about 140 cycles while the difference of latency between buffer depth 5 and 10 system is 230 cycles. In this system, it is also evidently that system with buffer depth 10 offers insignificant throughput improvement but increases the latency instead. The insignificant improvement is due to the fact that beyond 5 buffer depth, the network and normal resources are already fully utilized.

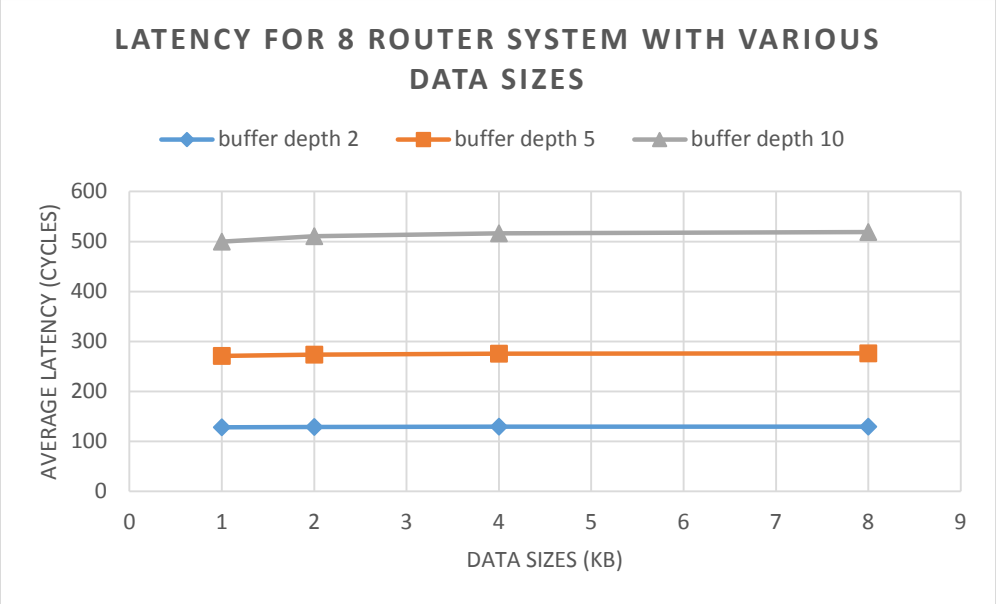


Figure 5-12: Latency for eight router system with various data sizes

For sixteen router system, the latency increases for the system with buffer depth 2, 5 and 10 with 160 data as shown in Figure 5-13. This is because the router size is significantly larger compared to 4 and eight router system which affects the time for the system to reach a steady throughput response.

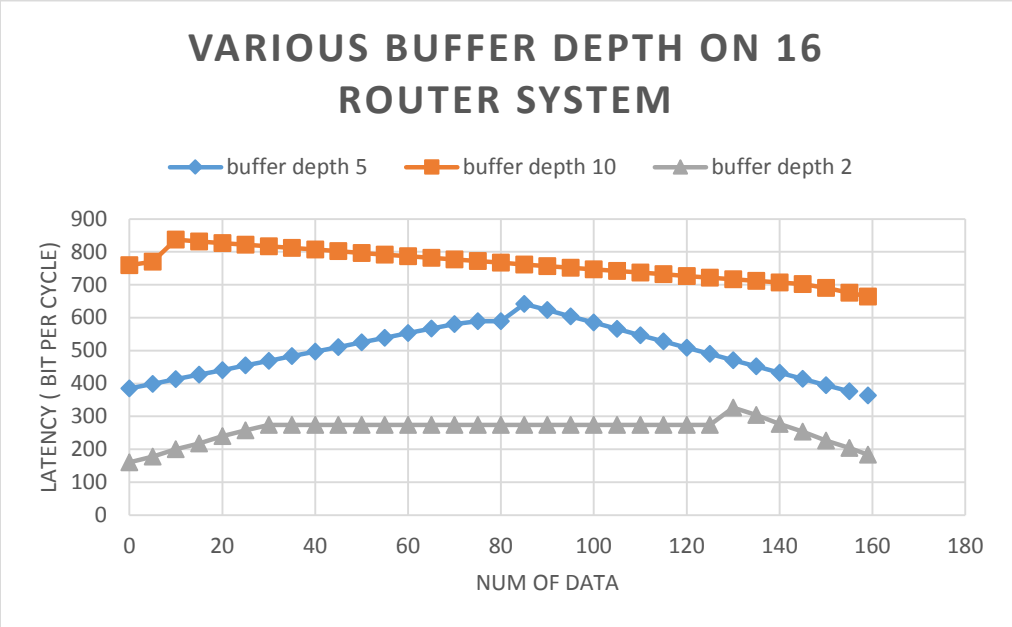


Figure 5-13: Various Buffer Depth for sixteen router system

Similarly, the study of the response of various data sizes on this system has been performed. Figure 5-14 shows the throughput result response of the system. When the data size is lesser than 2kB, the system with buffer depth 10 would have the lowest throughput of 0.5 bit per cycle lesser compared to the system with buffer depth 2 and 5. At this point, the system would be more efficient if a buffer depth 2 is used instead. As the data size increases to 4kB, the system with buffer depth 5 and 10 offers about 0.5 bit per cycle of throughput improvement. This makes a system with buffer depth 5 a better choice compared to buffer depth 10 system. When the data size is more than 4kB, system with buffer depth 10 offers only 0.1 bit per cycle of throughput improvement to the system. Again, the insignificant improvement is due to the fact that beyond 5 buffer depth, the network and normal resources are already fully utilized.

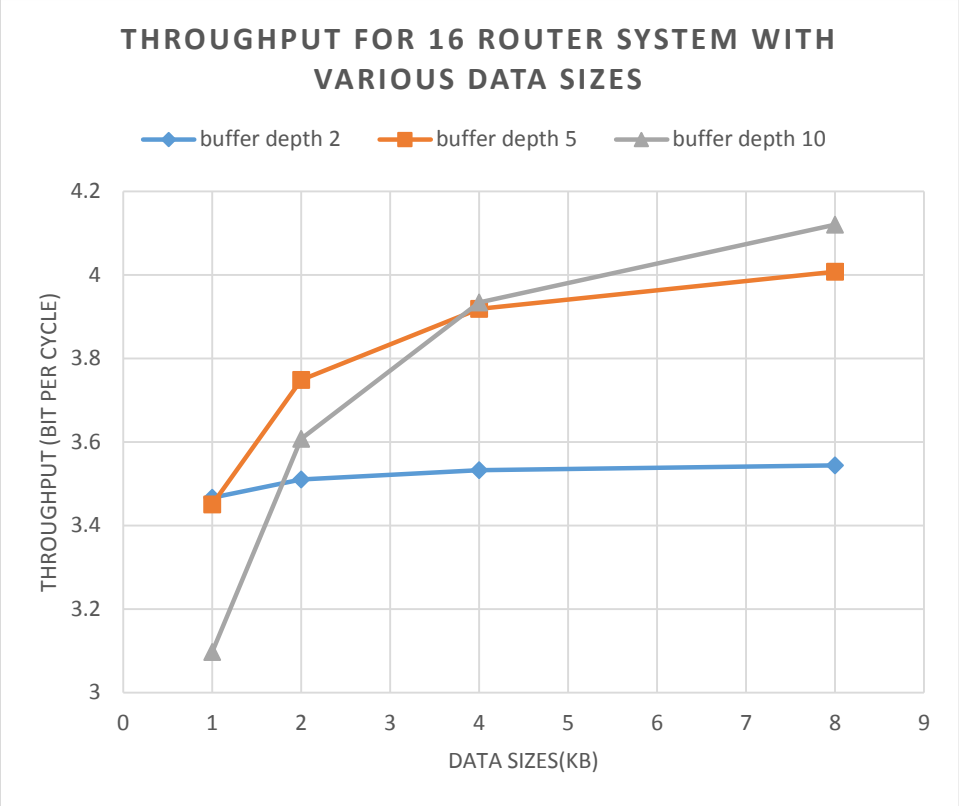


Figure 5-14: Throughput for 16 router system with various data sizes

The latency response of the sixteen router system can be seen in Figure 5-15. The latency of a buffer depth 5 system is 300 cycles more than the buffer depth 2 system. The buffer depth 10 system has 450 cycles latency more than a buffer depth 5 system. Overall, system with buffer depth 5 is the most efficient in this case as increasing the buffer depth further offers insignificant improvement of the throughput.

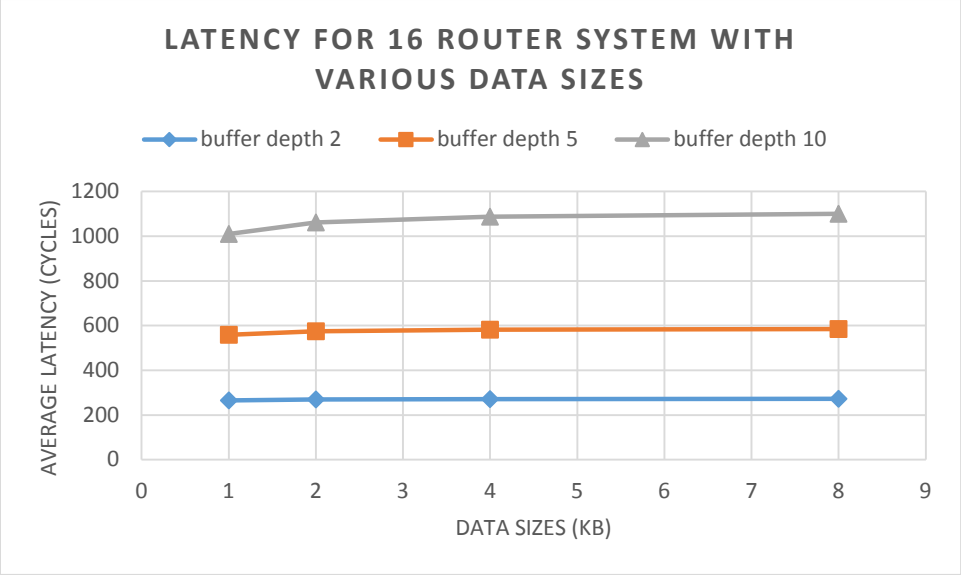


Figure 5-15: Latency for 16 router system with various data sizes

The comparison between the router systems and buffer depth is performed. Figure 5-16 shows the result of these comparisons. In this case, the four router system has a better throughput performance compared to the eight and sixteen router system. The maximum throughput of these systems are about 3.55 bit per cycle. This is because the special core has more buffers to monitor in a larger system.

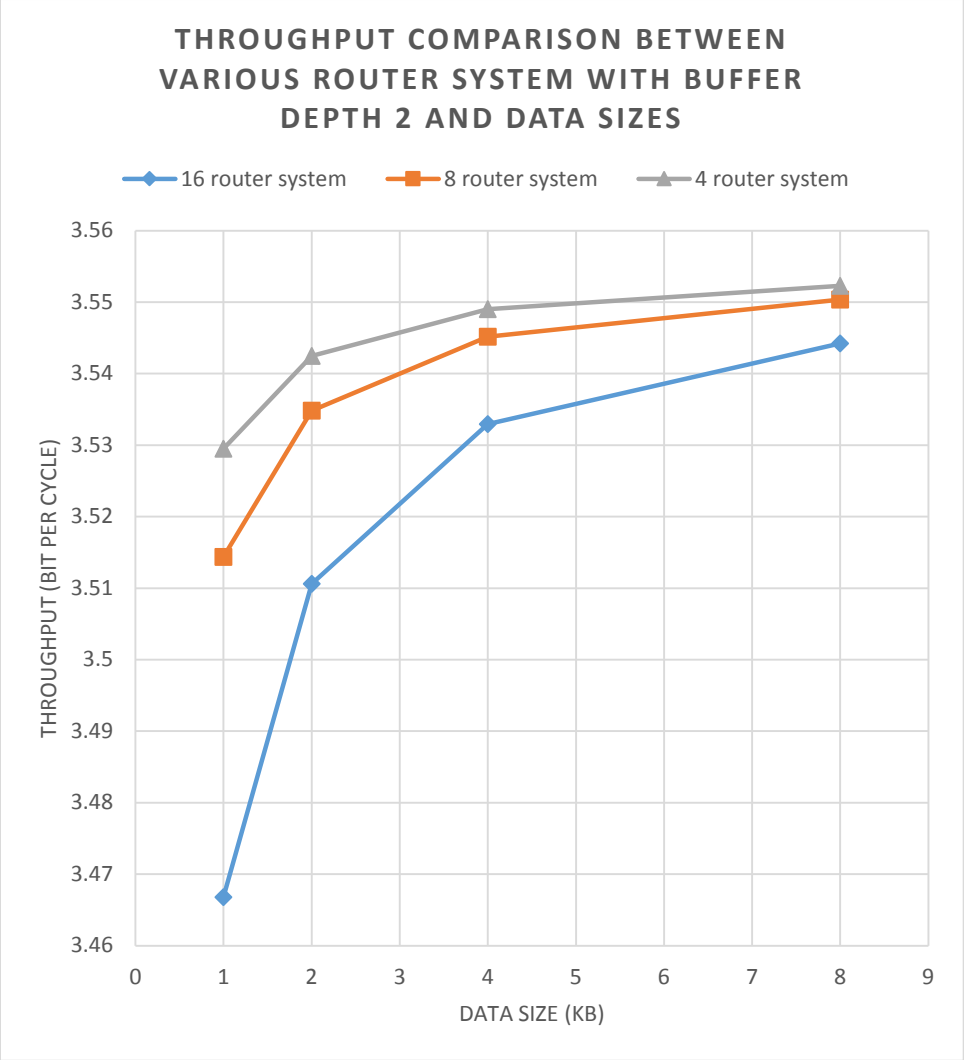


Figure 5-16: Throughput comparison between various router system with buffer depth 2 and data sizes

The latency for comparison for the systems with buffer depth 2 is shown in Figure 5-17. It is expected that the sixteen router system will have the highest latency since the overall network size is larger compared to the 4 and eight router system.

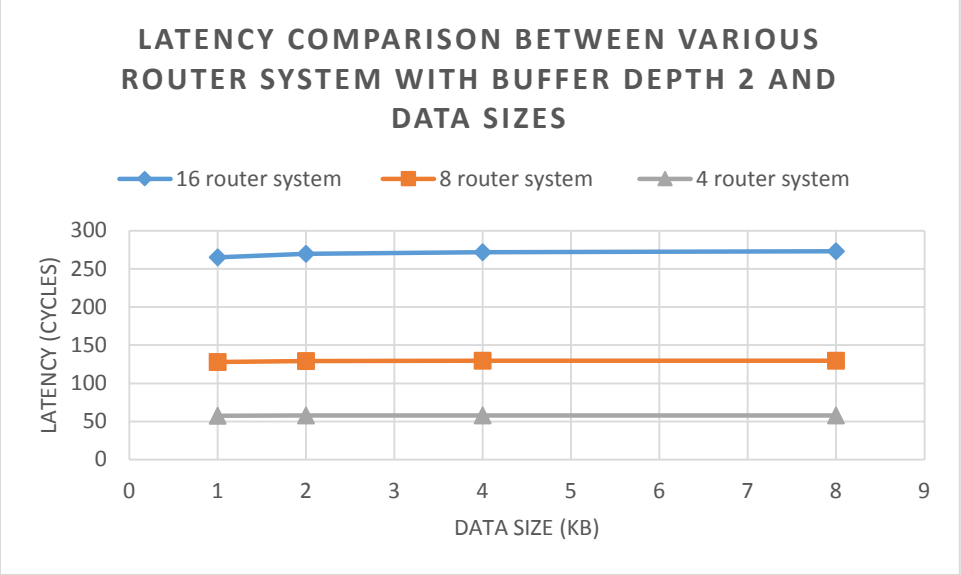


Figure 5-17: Latency comparison between various router system with buffer depth 2 and data sizes

Figure 5-18 shows the throughput comparisons between systems with 5 depth buffers. The throughput for a 4 router system is larger than the throughput of the eight and sixteen router system. From this result, it can be predicted that when the router system becomes larger, the throughput of the system will reduce. The maximum throughput of these systems are about 4.1 bit per cycle. Comparing the buffer depth 2 and the buffer depth 5 systems, the maximum throughput has increased by 0.55 bit per cycle from 3.55 to 4.1 bit per cycle.

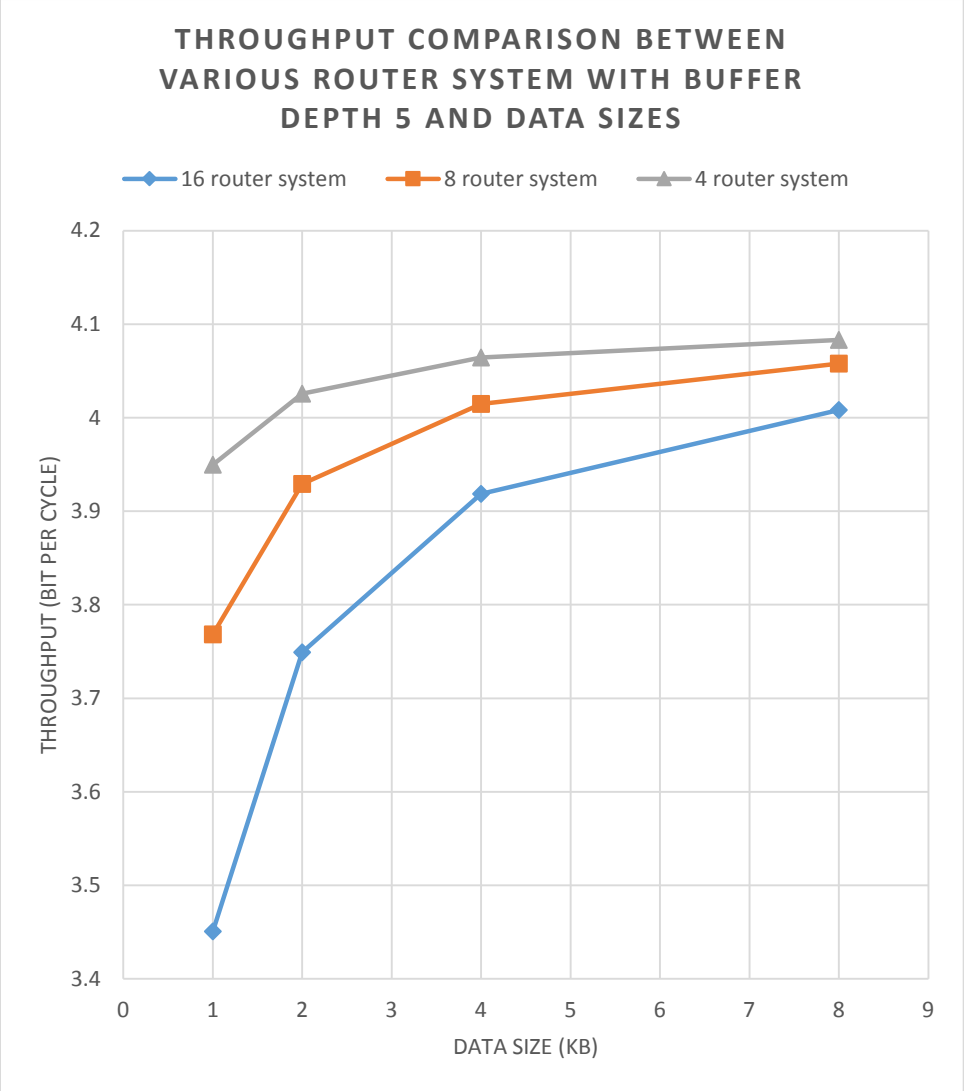


Figure 5-18: Throughput comparison between various router system with buffer depth 5 and data sizes

The latency between the systems with 5 depth buffer is shown in Figure 5-19. The 4 router system has the least latency in the comparison. This is because the size of the 4 router system is smaller compared to the size of a sixteen router system. From the result, it can also be seen that when the size of the router doubled, the latency is also doubled.

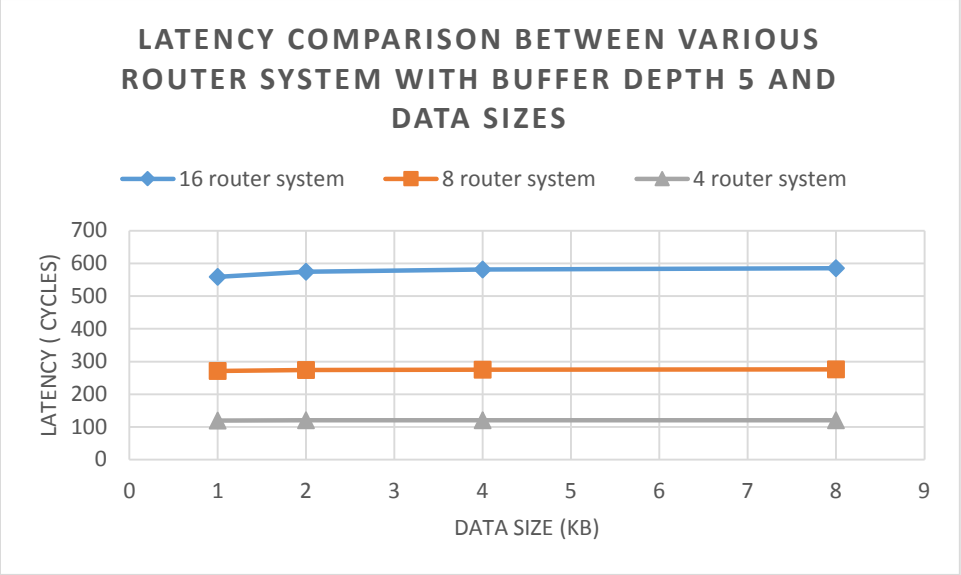


Figure 5-19: Latency comparison between various router system with buffer depth 5 and data sizes

The throughput comparison for 10 depth buffer for the 4, eight and sixteen router system is shown in Figure 5-20. From this result, the maximum throughput for these systems are 4.3 bit per cycle.

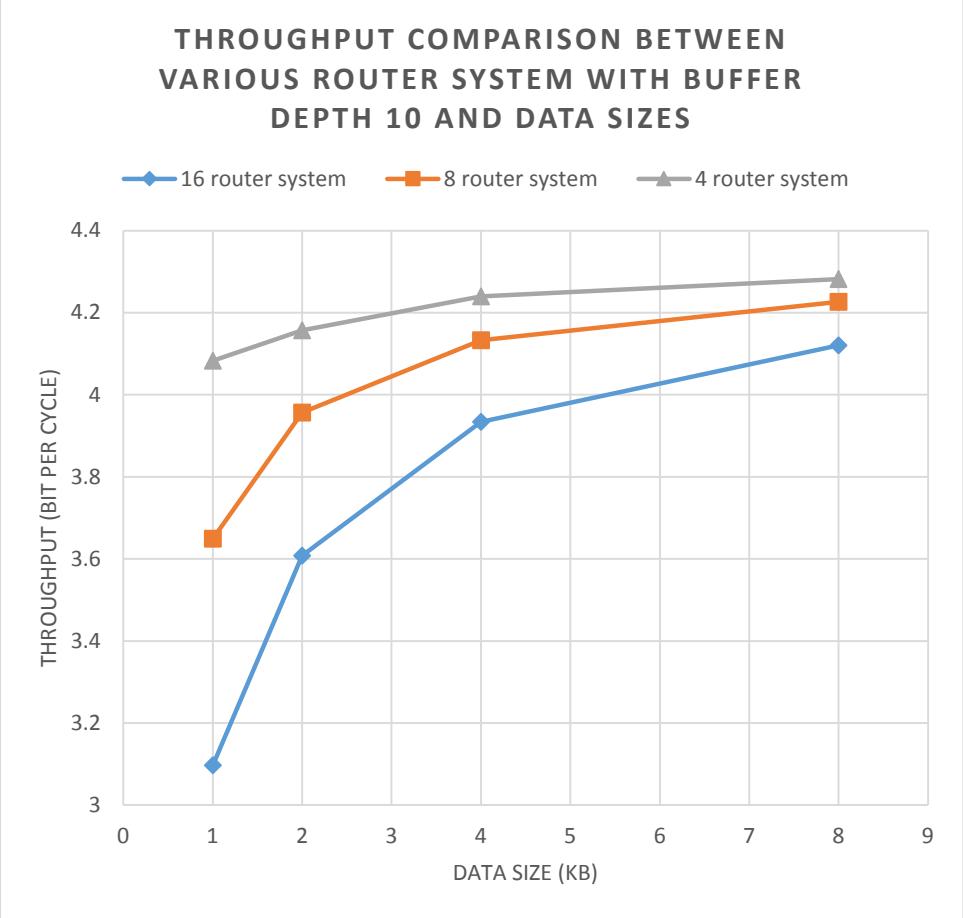


Figure 5-20: Throughput comparison between various router system with buffer depth 10 and data sizes

The sixteen router system with buffer depth 10 has the largest latency compared to 4 and eight router system because the larger the router system, the more buffers the system has for the corresponding normal cores.

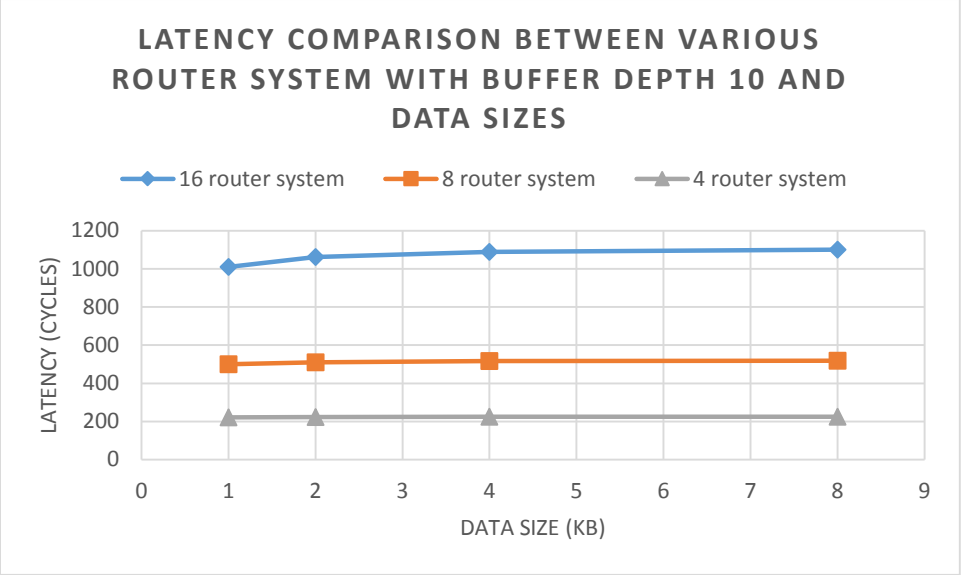


Figure 5-21: Latency comparison between various router system with buffer depth 10 and data sizes

The next subsection will discuss the methods to scale and reconfigure the platform.

5.4 *Verification Environment Scalability and Re-configurability*

The platform consists of parameterized modular verification environments that can be instantiated according to the targeted NoC architectures and the processing node architectures.

We used the following concepts to construct a scalable and reconfigurable NoC verification platform: *Virtual Sequencer*, *Configuration Objects*, and *Sequence Library*.

5.4.1 Virtual Sequencer

Each environment in the platform has one or more *Sequencers* on which the *Sequences* can be injected into the DUT. In UVM, there is a *start* method to start a *Sequence* on a particular *Sequencer*. To be able do so, we must obtain a handler to that particular *Sequencer*. The *Virtual Sequencer* refers to a *Sequencer* that has all the handlers to the *Sequencer* in the platform. This essentially allows us to control and coordinate all the stimulus generators.

The *Virtual Sequencer* setup is illustrated in Figure 5-22. In the figure, the AHB Seqr 0 acts as the handler corresponds to the AHB ENV 0. Similarly, the NoC Seqr 0 provides a handler to the NoC ENV 0 environment. After the platform is constructed in the *build phase*, we connect the handlers in the *Virtual Sequencer* to the respective *Sequencers*. In UVM, these are simply assignments of handler variables.

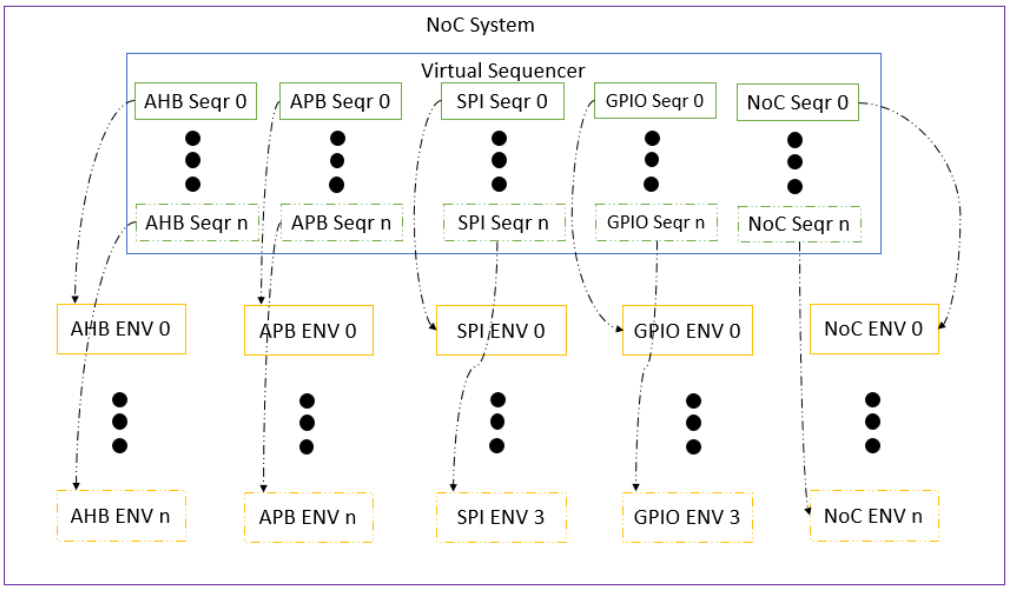


Figure 5-22: *Virtual Sequencer* with lower level *Sequencer* handler

Since our platform is parameterized, such assignments can be done automatically because the hierarchical paths of the *Sequencer* handlers are well defined by the parameters. Part of the *Virtual Sequencer* code with the AHB and SPI handler is shown in Figure 5-23.

```
476 for(int i = 0; i != noc_cfg.num_of_special_core+noc_cfg.num_of_normal_core; i++) begin
477         virtual_sequencer.ahb_seqr[i] =  ahb_envs[i].master_agent.sequencer;
478         virtual_sequencer.spi_seqr[i] =  spi_envs[i].agents[0].sequencer;
```

Figure 5-23: Part of the *Virtual Sequencer* code

5.4.2 Configuration Objects

Most of the parameters of our platform are controlled through the *Configuration Objects*. As discussed previously, UVM provides the configuration database class to facilitate such use case. This database can be used to store virtually any objects.

In our platform, the appropriate system component can be reconfigured which includes the number of masters and slaves for each environment. This also includes the slave address mapping for systems such as AHB and APB. The discussion of the *Configuration Objects* in this section is based on the NoC *Configuration Object* as shown in Figure 5-24. We shall describe each of its parameters and show how to use these parameters to configure the *Environments* and its sub-components.

The ***num_of_routers*** specifies the number of NoC master while the ***num_of_slave_routers*** is used to specify the number of NoC slaves. These are used in to configure the components in the NoC *Environment*. The ***num_of_normal_core*** and ***num_of_special_core*** define the number of cores in the platform. As an example of their usage, these parameters are needed to determine the connections between the *Virtual Sequencer* and the *Sequencer* for the AHB and SPI *Environment* as discussed in Figure 5-23 in the previous section.

```
13 //------------------------------------------------------------
14 // NOC Configuration Information
15 //------------------------------------------------------------
16 class noc_config extends uvm_object;
17
18         noc_config master_configs[$];
19         noc_config slave_configs[$];
20
21         int num_of_routers;
22         int num_of_slave_routers;
23         int num_of_normal_core;
24         int num_of_special_core;
25
26         `uvm_object_utils_begin(noc_config)
27         `uvm_field_string(name, UVM_DEFAULT)
28         `uvm_field_int(num_of_routers, UVM_DEFAULT)
29         `uvm_field_int(num_of_normal_core, UVM_DEFAULT)
30         `uvm_field_int(num_of_special_core, UVM_DEFAULT)
31         `uvm_field_int(num_of_slave_routers, UVM_DEFAULT)
32         `uvm_field_queue_object(slave_configs, UVM_DEFAULT)
33         `uvm_field_queue_object(master_configs, UVM_DEFAULT)
34         `uvm_object_utils_end
35         //------------------------------------------------------------
36         // new
37         //------------------------------------------------------------
38         function new (string name = "unnamed-noc_config");
39                 super.new(name);
40         endfunction
41         //------------------------------------------------------------
42         // Additional class methods
43         //------------------------------------------------------------
44         extern function void add_master(string name, uvm_active_passive_enum is_active =
   UVM_ACTIVE);
45         extern function void add_slave(string name, uvm_active_passive_enum is_active =
   UVM_ACTIVE);
46 endclass  : noc_config
```

Figure 5-24: Part of NoC *Configuration Object*

In addition to the parameters described, there are two functions in the NoC *Configuration Objects*. These can be used to add slaves in the NoC *Environment* using ***add_slave()*** function or to add masters through the ***add_master()*** function. The masters and slaves have previously been discussed in Chapter 4 Section 4.1.

The detailed implementation of the ***add_master()*** function for the NoC *Configuration Object* is shown in Figure 5-25. Each time when this function is called, a new master configuration object is created and stored in the **master_configs** queue.

```
//-------------------------------------------------------
// noc_config - Creates and configures a master agent config and adds to a queue
//-------------------------------------------------------
function void noc_config::add_master(string name, uvm_active_passive_enum is_active = UVM_ACTIVE);
        noc_config tmp_master_cfg;
        tmp_master_cfg = noc_config::type_id::create("master_configs");
        tmp_master_cfg.name = name;
        tmp_master_cfg.is_active = is_active;
        master_configs.push_back(tmp_master_cfg);
        num_masters++;
endfunction : add_master
```

Figure 5-25: Add Master Function

Similarly, Figure 5-26 shows the implementation of the slave parameter. Each time when this function is called, a new slave configuration object is created and stored in the **slave_configs** queue.

```
//-------------------------------------------------------
// noc_config - Creates and configures a slave agent config and adds to a queue
//-------------------------------------------------------
function void noc_config::add_slave(string name, uvm_active_passive_enum is_active = UVM_ACTIVE);
        noc_config tmp_slave_cfg;
        tmp_slave_cfg = noc_config::type_id::create("slave_configs");
        tmp_slave_cfg.name = name;
        tmp_slave_cfg.is_active = is_active;
        slave_configs.push_back(tmp_slave_cfg);
        num_slaves++;
endfunction : add_slave
```

Figure 5-26: Add Slave Function

To use this NoC *Configuration Object*, it is registered to the *Factory* in the platform. The usage of the *Factory* has been described previously in Chapter 3 Section 3.1.5. This Configuration Object is passed into the NoC Environment and its subcomponents. Figure 5-27 Figure 5-27 shows part of the code that perform this task.

```
140                 // noc
141                 noc_cfg = noc_system1_1_m2x2_34_config::type_id::create("cfg");
142                 uvm_config_object::set(this, "noc_env0*", "cfg", noc_cfg);
143                 `uvm_info(get_type_name(),$psprintf("noc_cfg = %s", noc_cfg.sprint
    ()),UVM_LOW)
144
```

Figure 5-27: Setting NoC *Configuration Object*

In each of the components, Figure 5-28 shows the code used in the build phase to retrieve the *Configuration Objects*. This Configuration Object can then be used to configure its sub components.

```
29 if(cfg == null) begin   // check whether tb set the config
30   if (!uvm_config_db#(noc_config)::get(this, "", "cfg", cfg)) begin     // try to get config from
   database
31    `uvm_error("NOCFG",{"config must be set for: ",get_full_name(),".cfg"});      // no config found
32  end
33 end
```

Figure 5-28: Code to retrieve the Configuration Objects

Figure 5-29 shows part of the NoC *Environment* code. The ***num_of_routers*** determines the number of NoC master *Agents* that to create. The information stored in the each of the ***master_configs*** is retrieved to configure the master *Agents*. These *Agents* will create their *Driver*, *Sequencer* and *Monitor* accordingly.

```
38 for(int i = 0; i < cfg.num_of_routers; i++) begin
39
40         string sname,sname_with_star ;
41         sname = $psprintf("agents[%0d]", i);
42         sname_with_star = $psprintf("agents[%0d]*", i);
43
44         uvm_config_int::set(this, sname_with_star, "id", i); // distribute master_agent
   index
45         uvm_config_object::set(this, sname_with_star, "cfg", cfg.master_configs[i]); // distribute
   master_config
46         agents[i] = noc_agent::type_id::create(sname, this);
47
48         if (agents[i] == null)
49                 `uvm_info(get_type_name(), $psprintf("agents[%0d] is null", i), UVM_LOW)
50         else
51                 `uvm_info(get_type_name(), $psprintf("agents[%0d] is not null", i), UVM_LOW)
52 end
```

Figure 5-29: Part of NoC *Environment* Code

Reusing the same environment, multiple instances of the *Agents* can be invoked through the ***add_slave()*** and ***add_master()*** function. This allows the verification environment to be scaled horizontally.

Figure 5-30 shows how we configure the platform to model a four routers NoC system. There are four cores in the system, a special core and three normal cores, which are specified through the ***num_of_normal_core*** and ***num_of_special_core*** parameters.

```
140             // noc
141             noc_cfg = noc_config::type_id::create("cfg");
142             noc_cfg.num_of_routers = 4;
143             noc_cfg.num_of_normal_core = 3;
144             noc_cfg.num_of_special_core = 1;
145             noc_cfg.add_master("master0", UVM_PASSIVE);
146             noc_cfg.add_master("master1", UVM_PASSIVE);
147             noc_cfg.add_master("master2", UVM_PASSIVE);
148             noc_cfg.add_master("master3", UVM_PASSIVE);
149             uvm_config_object::set(this, "noc_env0*", "cfg", noc_cfg);
150             `uvm_info(get_type_name(),$psprintf("noc_cfg = %s", noc_cfg.sprint()),UVM_LOW)
```

Figure 5-30: NoC configuration for four router system

Figure 5-31 shows the NoC *Configuration Objects* to configure the existing platform for an eight routers system. The parameters inside the NoC *Configuration Object* are changed accordingly. For this system, it consists of a special core and seven normal cores.

```
140             // noc
141             noc_cfg = noc_config::type_id::create("cfg");
142             noc_cfg.num_of_routers = 8;
143             noc_cfg.num_of_normal_core = 7;
144             noc_cfg.num_of_special_core = 1;
145             noc_cfg.add_master("master0", UVM_PASSIVE);
146             noc_cfg.add_master("master1", UVM_PASSIVE);
147             noc_cfg.add_master("master2", UVM_PASSIVE);
148             noc_cfg.add_master("master3", UVM_PASSIVE);
149             noc_cfg.add_master("master4", UVM_PASSIVE);
150             noc_cfg.add_master("master5", UVM_PASSIVE);
151             noc_cfg.add_master("master6", UVM_PASSIVE);
152             noc_cfg.add_master("master7", UVM_PASSIVE);
153             uvm_config_object::set(this, "noc_env0*", "cfg", noc_cfg);
154             `uvm info(get type name(),$psprintf("noc cfg = %s", noc cfg.sprint()),UVM LOW)
```

Figure 5-31: NoC Configuration for eight routers system

Similarly, the configuration of the sixteen router system is as shown in Figure 5-32 which consist of a special core and fifteen normal cores.

```
140             // noc
141             noc_cfg = noc_config::type_id::create("cfg");
142             noc_cfg.num_of_routers = 16;
143             noc_cfg.num_of_normal_core = 15;
144             noc_cfg.num_of_special_core = 1;
145             noc_cfg.add_master("master0", UVM_PASSIVE);
146             noc_cfg.add_master("master1", UVM_PASSIVE);
147             noc_cfg.add_master("master2", UVM_PASSIVE);
148             noc_cfg.add_master("master3", UVM_PASSIVE);
149             noc_cfg.add_master("master4", UVM_PASSIVE);
150             noc_cfg.add_master("master5", UVM_PASSIVE);
151             noc_cfg.add_master("master6", UVM_PASSIVE);
152             noc_cfg.add_master("master7", UVM_PASSIVE);
153             noc_cfg.add_master("master8", UVM_PASSIVE);
154             noc_cfg.add_master("master9", UVM_PASSIVE);
155             noc_cfg.add_master("master10", UVM_PASSIVE);
156             noc_cfg.add_master("master11", UVM_PASSIVE);
157             noc_cfg.add_master("master12", UVM_PASSIVE);
158             noc_cfg.add_master("master13", UVM_PASSIVE);
159             noc_cfg.add_master("master14", UVM_PASSIVE);
160             noc_cfg.add_master("master15", UVM_PASSIVE);
161             uvm_config_object::set(this, "noc_env0*", "cfg", noc_cfg);
162             `uvm_info(get_type_name(),$psprintf("noc_cfg = %s", noc_cfg.sprint()),UVM_LOW)
```

Figure 5-32: NoC Configuration for 16 routers system

Figure 5-33 illustrates graphically the constructed platform based on the parameters in the NoC *Configuration Object*. Through the setting of these parameters, the corresponding checkers are also instantiated. Each AHB2NoC bridge link will have a pair of AHB2NoC Send and Receive Scoreboard to monitor the transfer. Another NoC Scoreboard to verify the transfers within the router is setup. The

scoreboards are identified with two indices. The first index would denote the node number of the router. The second index would indicate the link to each of the nodes. For example, NoC Sbrd [1][0] would monitor the transfer from router 1 to router 0 and NoC Sbrd [0][1] would monitor the transfer from router 0 to router 1.

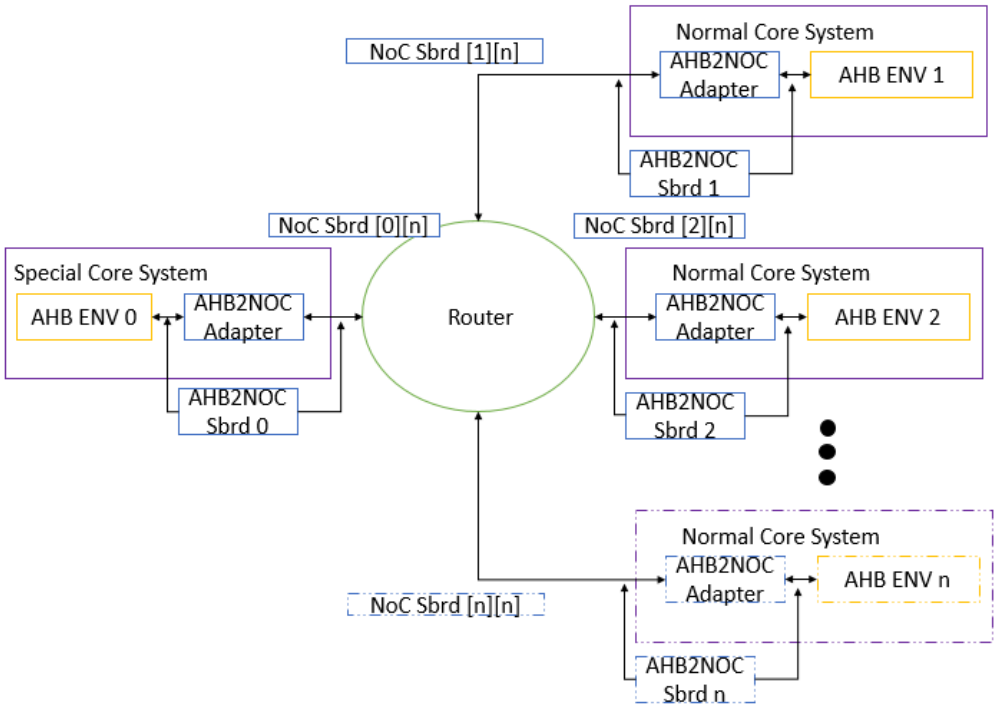


Figure 5-33: NoC Platform

For different network router architecture with the same number of routers, the new router can be used to replace the old router. In the case where the number of routers is different, the top design module definition has to be changed in accordance to the router's input output ports.

5.4.3 Sequence Library

The scalability of the environment can also be scaled vertically. Using the basic *Sequences* that have been discussed in the previous sections, other *Sequences* can be extended from these *Sequences*. Figure 5-34 shows part of the ***ahb2noc_special_base_seq***. The *Sequence* is extended from one of the AHB Master Basic *Sequence*. This is used to generate and read transfers for the AHB2NoC Bridge.

```
65 class ahb2noc_special_base_seq extends basic_multiple_seq;
66
67         function new(string name="ahb2noc_special_base_seq",
68                 uvm_sequencer_base sequencer=null,
69                 uvm_sequence parent_seq=null);
70                 super.new(name);
71         endfunction : new
72
73         // Register sequence with a sequencer
74         `uvm_object_utils(ahb2noc_special_base_seq)
75         `uvm_declare_p_sequencer(ahb_master_sequencer)
76
```

Figure 5-34: part of the *ahb2noc_special_base_seq Sequence*

An ***ahb_aes_base_seq*** as shown in Figure 5-35 is also extended from the same ***basic_multiple_seq***. This *Sequence* generates the plaintext and key for the AES encryption. There is also a task to read the cipher text from the AES after the encryption is completed. This shows that the basic *Sequences* can be reused to develop the other *Sequences*.

```
class ahb_aes_base_seq extends basic_multiple_seq;

        function new(string name="ahb_aes_base_seq",
                uvm_sequencer_base sequencer=null,
                uvm_sequence parent_seq=null);
                super.new(name);
        endfunction : new

        // Register sequence with a sequencer
        `uvm_object_utils(ahb_aes_base_seq)
        `uvm_declare_p_sequencer(ahb_master_sequencer)
```

Figure 5-35: part of *ahb_aes_base_seq Sequence*

Our platform can be easily extended to include a new system component. We briefly describe the required steps in the following paragraphs.

Firstly, the corresponding environment of that component has to be developed. This environment will be instantiated in the platform using a parameterized variable that is configurable through the *Configuration Objects*.

Next, a *Sequencer* handler for this new environment has to be added to the existing *Virtual Sequencer*. This is because a *Sequence* can only be started on the *Sequencer* with the same type.

The new test *Sequences* corresponding to the new architecture have to be created as well because of the new test scenarios for the new NoC architecture. These test cases can be derived from the *Sequences* library.

In addition to the above, a new virtual interface has to be defined when developing the new component. This virtual interface is created at the top level module of the platform to bind the new system component and the platform.

5.5 *Summary*

This Chapter starts off by giving an overview of our platform and verification plans for our UTAR NoC based on the design specifications. Only a few main modules are used as illustrations in order to keep the clarity and conciseness of this dissertation. The discussion is followed by the various *Sequences* that we have developed for various modular verification environments. The design exploration that we have performed on a four-, eight-, and sixteen-router system illustrates the re-configurability and scalability of the verification environment also has been discussed. We have also demonstrated by our experimentation with various buffer sizes, data sizes and router numbers that this platform can be readily used for architectural exploration. The final section of this chapter discussed the scalability and re-configurability approach for the module and system level verification environment. The next Chapter shall conclude this dissertation.

CHAPTER 6

CONCLUSION

This chapter summarizes the dissertation. The development of the various verification components that is used to assist the verification of our UTAR NoC will be highlighted. This summary will also include the summary of the methods that is involved to make the verification environment scalable and reconfigurable. The discussion ends with the stress on the importance of design verification and the reasons reusability, scalability and configurability are very desirable attributes.

6.1 Conclusion

The Universal Verification Components (UVC) are component models which are developed based on the Universal Verification Methodology (UVM). The various UVCs that we have developed models the protocol layers such as AMBA High-Speed Bus (AHB), AMBA Advanced Peripheral Bus (APB), GPIO, parallel port, CONNECT network, and SPI.

These verification components are used to generate the stimulus and monitor the bus connected to the Design Under Test (DUT). From these component level environments, we have reused them to design our sub-system level verification environment. These component and sub-

system level environments are used to form our UTAR NoC system level verification environment. With minimal effort, these similar environments can be setup quickly for verification of different architectural requirements and scale according to the network size.

The detailed architecture of the verification environment from a component to system level has been discussed in Chapter 4 Verification Platform Architecture. The component verification platform has been used to verify the modules that we have designed, namely, the AHB2NoC Bridge, AES encryption model, GPIO, parallel port, AHB master and slave models and SPI. These components have been used to assist in the verification process of the UTAR NoC. The architecture of our UTAR NoC system level verification environment is derived from the modular and subsystem level UVCs environments are also explained in that chapter.

These environments can easily be scaled horizontally. This can be done by using the *Configuration Objects* to configure the verification environment to expand according to the system needs. Multiple components or *Agents* can be instantiated within a verification environment based on these parameters. The parameters have also been used in our system level verification environment to instantiate multiple modular environments and checkers based on the number of processors in the system. These will allow each of the environments to monitor the responses from the processors and its peripherals.

Another aspect of scalability has been demonstrated through the use of the basic *Sequences* that basically models the protocol layer. These *Sequences* is used in higher level *Sequences*. By using this approach, the environment can be said to be able to scale vertically as well. The lowest level is the basic *Sequences* that act as the interface to access the *Driver* and *Monitor*. In the higher level *Sequences*, more focus can be put into designing various test cases or scenarios to exercise the system under test by utilizing the lower level *Sequences*.

The result of our verification environment is discussed in Chapter 5. In the results and discussion section, the discussion starts off with the various basic *Sequences* that we have designed as stimulus for our system. This includes a *Sequence* that can be used to generate the traffic and measures the performance of a four-, eight- and sixteen-router system for architectural performance exploration purposes.

The first exploration is done to investigate the effect of various buffer sizes of the AHB2NoC Bridge on the systems. A protocol where the special core sends a block of data with data block size the same as the AHB2NoC bridge buffer depth is used. From the results, the system would be the least efficient when the amount of data is lesser than the total buffer size of the AHB2NoC Bridge. By increasing the buffer size, the throughput of the system can be increased. The Ahb2NoC Bridge with buffer depth 5 gives the most efficient throughput. Increasing the

buffer depth further will only increase the throughput by 0.1 bits per cycle. The maximum throughput of a 2 buffer depth is 3.55 bits per cycle compared to a 5 buffer depth which is 4.1 bits per cycle. For the 10 deep buffer system, the maximum throughput is 4.3 bits per cycle.

These performance measurements are further extended to an eight and sixteen router system to demonstrate the re-configurability and scalability of the architecture and verification environment that we have proposed. It also serves as a performance comparison among the four, eight and sixteen router system. It is observed that as the number of nodes in the NoC increases, the latency of the system also increases. This factor has to be taken into account when designing the architecture so that the overall system performance is not significantly affected.

In conclusion, this dissertation has demonstrate the concepts involved in the development of a reusable, scalable and reconfigurable multi-processor Network-On-Chip virtual prototyping platform based on the Universal Verification Methodology (UVM). Object Oriented Programming (OOP) and UVM concepts along with System Verilog has provided an essential verification methodology and language for the current ASIC design challenges.

Any manufacturing defects and crystalline imperfections in the silicon wafers could cause faults at random locations, and therefore has to be discovered. Tests are developed to discover these faults. Achieving full functional, line, code and toggle coverage could provide a high degree of these fault coverage. Verification itself could therefore easily take up to 80% of the all the available resources in the ASIC design. Thus, a reusable, scalable and reconfigurable verification component and platform would certainly lessen the effort of the verification process and to ease design exploration.

REFERENCES

Anderson, T. L., 2010. *UVM: Extending Standardization From Language To Methodology,* s.l.: Chip Design Magazine.

ARM , 2003. *AMBA™ 3 APB Protocol v1.0 Specification,* s.l.: s.n.

ARM, 2006. *AMBA® 3 AHB-Lite Protocol v1.0 Specification,* s.l.: ARM Limited.

Ashwin P. Patel,Vyom M. Bhankhariya, and Jignesh S. Prajapati, 2013. An Overview Of Transaction-Level Modelling (TLM) In Universal Verification Methodology (UVM). *Journal Of Information, Knowledge And Research In Electronics And Communication Engineering,* 2(2), pp. 542-546.

Aynsley, J., 2012. *Easier SystemVerilog with UVM: Taming the Beast.* San Jose, DVCon 2012.

Ballance, M., 2009. *Evolving the Coverage-Driven Verification Flow,* s.l.: Mentor Graphics.

Bhaumik Vaidya, and Jaydeep Bhatt, 2013. Design of a robust verification environment for AMBA AHB System in Universal Verification Methodology. *International Journal of Research in Computer Engineering and Electronics,* pp. 1-6.

Black, D. C., 2013. *A Tale of Two Languages: SystemVerilog & SystemC.* San Jose, DVCon 2013.

Bromley, J., 2013. *If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language.* Paris, Specification & Design Languages (FDL), 2013 Forum on.

Brown, S., 2009. *TLM-Driven Design And Verification – Time For A Methodology Shift,* s.l.: Cadence Design Systems INC.

Chris Spear, and Greg Tumbush, 2012. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features.* 3rd edition ed. s.l.:Springer.

Cummings, C. E., 2012. *The OVM/UVM Factory & Factory Overrides How They Work - Why They Are Important.* s.l., Sunburst Design, Inc., pp. 1-48.

David C. Black, Jack Donovan, Bill Bunton, and Anna Keist, 2009. *Systemc from the Ground Up.* 2nd ed. s.l.:Springer.

Donatella Sciuto, Grant Martin, Wolfgang Rosenstiel, Stuart Swan, Frank Ghenassia, Peter Flake, and Johny Srouji, 2004. *SystemC and SystemVerilog: Where do they fit? Where are they going?*. Washington, IEEE Computer Society.

Foster, H., 2013. *Wilson Research Group 2012 Functional Verification Study,* s.l.: Mentor Graphics Corp.

Ghenassia, F., 2005. *Transaction-Level Modeling With SystemC: TLM Concepts and Applications for Embedded Systems.* 1st ed. s.l.:Springer.

Janick Bergeron, Eduard Cerny, Alan Hunter & Andy Nightingale, 2005. *Verification Methodology Manual for SystemVerilog.* 1st ed. s.l.:Springer.

Janick Bergeron, Fabian Delguste, Steve Knoeck, Steve McMaster, Aron Pratt, and Amit Sharma, 2013. *Beyond UVM: Creating Truly Reusable Protocol Layering.* s.l., DVCON 2013 .

Jie Chen,Cheng Li, and Paul Gillard, 2011. *Network-on-Chip (NoC) Topologies and Performance: A Review.* s.l., ocs2011.

Krolikoski, S., 2011. Evolution of EDA standards worldwide. January/Febuary, pp. 72-75.

L.Swarna Jyothi,Harish R, and Dr.A.S.Manjunath, 2008. Reusable Verification Environment for verification of Ethernet packet. *IJCSNS International Journal of Computer Science and Network Security,* 8(11), pp. 226-235.

Lu Kong, Wu-Chen Wu, Yong He, Ming He, and Zhong-Hua Zhou, 2009. *Design of SoC verification platform based on VMM methodology.* Beijing, IEEE.

Martin. G, and Smith. G, 2009. High-Level Synthesis: Past, Present, and Future. *Design & Test of Computers, IEEE*, July-August, pp. 18-25.

Mentor Graphics Corporation and Cadence Design Systems, Inc, 2007. *Open Verification Methodology (OVM),* s.l.: Mentor Graphics Corporation and Cadence Design Systems. Inc.

Pandya, K., 2013. Network Structure or Topology. *International Journal of Advance Research in Computer Science and Management Studies,* 1(2), pp. 22-27.

Rich Edelman, and Raghu Ardeishar, 2013. *Sequence, Sequence on the Wall – Who's the Fairest of Them All?.* San Jose, DVCon 2013.

Ruggiero, C., 2009. *Verification Methodology Matters.* s.l.:Chip Design Magazine.

Salemi, R., 2013. *Uvm primer.* 1st edition ed. s.l.:Boston Light Press.

Shvartz, A., 2003. *Maximizing Verification Productivity: eVC Reuse Methodology (eRM).* s.l.:Design & Reuse.

Sridevi Chitti, and Dr. G.Krishnamurthy, 2013. Reusable Verification Environment for Verification of Ethernet. *International Journal of Emerging Trends in Electrical and Electronics*, 5(2), pp. 22-25.

Verisity, 2004. *e Reuse Methodology (eRM) Developer Manual,* s.l.: Verisity.

Viney Malik,Rajesh Mehra, and Surender Ahlawat, 2013. Advanced Testbench Design using Reusable Verification Component and OVM. *International Journal of Computer Applications*, 73(15), pp. 36-40.

Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, and Byeong Min, 2011. *Beyond UVM for Practical SoC Verification.* s.l., IEEE, pp. 158-162.

APPENDIX A

Additional Verification Plan

A.0 *AHB Memory-Built-In-Self-Test (MBIST)*

In our system, we have implemented the MBIST module. The module injects various test patterns into the SRAM upon entering MBIST test mode. The purpose is to ensure that the SRAM is in good condition to operate. A test case to test the MBIST module to ensure that the MBIST module test the SRAM module.

Table 2-1: AHB MBIST Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_mbist_test	Set the chip into test mode. Replace the SRAM with SRAM model that can generate normal and faulty SRAM behaviour. Send start signal to mbist to send checkerboard, march-c pattern to SRAM including various faults test pattern, stuck-at-fault(SAF), transition fault(TF), Inversion Coupling Fault(CFin), Idempotent Coupling Fault(Cfid), Dynamic Coupling Fault(Cfdyn), AND and OR Bridging Fault(BF), State Coupling Fault(SCF), Address Decoding Fault A(AF_A), Address Decoding Fault b (AF_B), Address Decoding Fault C (AF_C), Address Decoding Fault D (AF_D). If the SRAM model in faulty mode, the test should detect fail flag set.	Ensure that the mbist can send various test patterns to verify the SRAM module and compare the results.

A.1 *AHB SRAM*

The ***ahb_sram_test*** generates the various types of the AHB transfer to write to and read from the SRAM. This ensures the SRAM can handle the AHB various AHB transfers correctly. In the event that there are invalid transfers, the SRAM should also be able to handle that situation. ***ahb_sram_invalid_test*** has been setup for this purpose. Table 2-2 shows the list of features and the criteria to verify the SRAM.

Table 2-2: AHB SRAM Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_sram_test	Generate multiple types of AHB transactions to the SRAM controller. Targeted address are randomized within the SRAM address map range. The order of transaction created:	Expects the correct transfer is written to SRAM and the correct content is read from the SRAM
	AHB SINGLE WRITE transaction(s), the number of SINGLE transaction created depends on the num_single variable value	
	AHB SINGLE READ transaction(s)	
	AHB SINGLE WRITE transaction(s) with an IDLE transaction inserted after every WRITE transaction(s)	
	AHB SINGLE READ transaction(s) with an IDLE transaction inserted after every READ transaction(s)	
	AHB SINGLE WRITE transaction(s) with a SINGLE READ transaction inserted after every WRITE transaction(s)	
	AHB INCR WRITE transaction(s), the number of transaction created depends on the num_incr variable value	
	AHB INCR READ transaction(s)	
	AHB INCR WRITE transaction(s) with BUSY transactions generated randomly	
	AHB INCR READ transaction(s) with BUSY transactions generated randomly	

Table 2-3: AHB SRAM Verification Plan Continued

Tests	Test Descriptions	Verification Criteria
	AHB INCRn WRITE transactions, the type of INCRn depends on the incr_n_burst variable value	
	AHB INCRn READ transactions	
	AHB INCRn WRITE transactions with BUSY transactions generated randomly	
	AHB INCRn READ transactions with BUSY transactions generated randomly	
	AHB WRAPn WRITE transactions, the type of WRAPn depends on the wrap_n_burst variable value	
	AHB WRAPn READ transactions	
	AHB WRAPn WRITE transactions with BUSY transactions generated randomly	
	AHB WRAPn READ transactions with BUSY transactions generated randomly	
ahb_sram_random_test	Generate random AHB transactions to the SRAM controller. Targeted address are randomized within the SRAM address map range.	Expects the SRAM is able to be accessed with various random transactions
ahb_sram_alternate_test	Alternately drive transactions to SRAM and APB interface to toggle HREADY pin of SRAM controller.	Expects the SRAM to be able to handle when the SRAM ready signal is toggled
ahb_sram_invalid_test	Drive double word and unalined transaction to SRAM.	Expects the SRAM to be able to handle invalid transfer
ahb_sram_stall_write_to_idle_test	Drive write transaction to SRAM and immediately after that continuously drive read transaction to put the SRAM into READ_STALL_WRITE state. After a few clock cycles, drive nReset low for 2 cycles. It will force the SRAM state from READ_STALL_WRITE to IDLE state.	

A.2 *AHB GPIO*

The GPIO is another mean of input for our system. The tests are generated to configure the GPIO as input. The correctness of the transfer is checked with the transfer received at the AHB. The GPIO is also set to output and the intended AHB transfer is sent to the GPIO. The GPIO is checked against the AHB transfer. Once the GPIO is verified to be able to send and receive transfer correctly, randomise test cases are done to randomly set each pin as input or output. The summary of test cases is shown in Table 2-4.

Table 2-4: AHB GPIO Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_gpio_in_test	Configure GPIO as input and compare the result of the transfer	Expects the GPIO to receive correct transfer
Ahb_gpio_out_test	Configure GPIO as output and compare the result of the transfer	Expects the GPIO to output the correct transfer
ahb_gpio_random_test	Randomly configure GPIO as input or output	Expects correct random read write transfer from the GPIO

A.3 *AHB2APB Bridge*

The AHB2APB Bridge allows the AHB transfer to be cast to APB transfer. During the 1st cycle of the transfer conversion, a minimum of 3 cycles are required for the complete APB transfer to be generated. The following APB transfer would require minimum 2 cycles.

The ***ahb_apb_bridge_test*** is used to check correctness of the converted transfer from AHB to APB and APB to AHB. It is also necessary to verify the correct slave select signals are generated and the transfer from the corresponding slaves are passed back to the AHB. This is done using ahb_apb_bridge_address_map_test. The ***ahb_apb_bridge_out_of_slave_test*** determines the response from the bridge when a non-existing slave is selected. This is to check the design so that it should produce a deterministic response for the system to be debugged. Table 2-5 shows the test cases for the AHB APB Bridge.

Table 2-5: AHB APB Bridge Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_apb_bridge_test	Read and write to APB slave 1 and 2 and compare transfer results	Expects the correct transfer from AHB to APB and APB to AHB
ahb_apb_bridge_address_map_test	Check slaves address mapping	Expects the corrct APB slave is selected and the correct content from the selected slave
ahb_apb_bridge_out_of_slave_test	Set ahb master to access beyond apb slaves addresses	Expects no transfer from the bridge

A.4 *AHB Parallel Port*

The AHB parallel port module is used as the primary input for the NoC System AES encryption application. There are transmit and receive buffers for transmitting and receiving. The enable transmit interrupt can be set. When the parallel port has finished transmitting the transfer, the transmit interrupt is triggered. Similarly, when the parallel port finished receiving a transfer, the receive interrupt is triggered.

The ***ahb_pp_32_test*** is used to verify the parallel port module to be able to transmit and receive parallel port transfer correctly. The test also includes cases where the Cortex-M0 writes to the parallel port module until the transmit buffer is full. The test monitors whether the trasmit buffer full flag is set when it is full. To further ensure that the transfers in the buffers are not overwritten, the test sends transfers after the buffers are full. The overall AHB Parallel Port tests are shown in Table 2-6.

Table 2-6: AHB Parallel Port Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_pp_32_test	Transmit 32 bit ahb transfer to parallel port and compare the data.	Expects the parallel port to receive and send transfer correctly.
	Receive 32 bit ahb transfer from parallel port and compare the data.	
	Transmit 32 bit ahb transfer to parallel port and receive 32 bit transfer from parallel port and compare the data.	
	Transmit 32 bit ahb transfer to parallel port until buffer size and check transmit bf flag and compare the data.	
	Receive 32 bit ahb transfer until buffer size and check receive bf flag and compare the data.	
	Transmit 32 bit ahb transfer to parallel port and set transmit interrupt and compare the data.	
	Receive 32 bit ahb transfer and set receive interrupt and compare the data.	
	Receive 32 bit ahb transfer until buffer size and set receive interrupt and compare the data.	
	Receive 32 bit ahb transfer until receive buffer overrun	
	Transmit 32 bit ahb transfer until trasmit buffer overrun	
ahb_pp_32_random_test	Randomly transmit and receive 32 bit ahb transfer	
ahb_pp_32_send_recv_test	Transmit and receive 32 bit transfer at the same time	
ahb_pp_32_turn_around_test	Transmit and receive at the fastest turn around from input to output and from output to input	

A.5 *AHB Advanced Encryption Standard (AES)*

The main application for the UTAR NoC is to perform AES encryption. The AES encryption core is attached to each of the normal core unit. This AES core can perform 128 bit encryption. The encryption starts when an AHB transfer is done to the last plaintext, ***AES_PLAIN3***. When the encryption is done, the ***AES_CF*** flag is set. Interrupt can also be enabled. Upon receiving the cipher text, the interrupt can be triggered. A buffer is implemented to store the new plain text during continuous transfer.

During the test, a set of 4 32bit plain text is sent to the AES module to be encrypted. The output cipher text is gathered using the interrupt and polling method. The result is compared using a reference model of the AES which is implemented in the AES scoreboard. Using the ***ahb_aes1_change_text_key_encryption_test***, the plain text and key is changed when the AES is performing the encryption. This is to ensure that the the new plain text and key does not corrupt the encryption data. Since writing to 3rd set of plain text, ***AES_PLAIN3*** can start the encryption, a test is setup to verify that the ***AES_PLAIN1*** and ***AES_PLAIN2*** is set to 0 or the previous set value. The ahb_aes1_overlap_test is used to verify that the newest plain text is stored in the buffer for encryption. Table 2-7 sumarizes the tests for the AES module.

Table 2-7: AHB AES Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_aes1_interrupt_test	4 set of plain text and key is sent to the AES module to be encrypted and result is compared and check for 12 cycles for the aes encryption to finished	Produce the right cipher text based on the plain text and key given
ahb_aes1_polling_test	Enable interrupt. 4 set of plain text and key is sent to the AES module to be encrypted and result is compared	Produce the right cipher text based on the plain text and key given
ahb_aes1_random_interrupt_test	Randomize plain text and key is sent to the AES module to be encrypted and once plain text is encrypted, interrupt is set and result is compared	Produce the right cipher text based on the plain text and key given
ahb_aes1_random_polling_test	Randomize plain text and key is sent to the AES module to be encrypted and wait until aes encrypted flag is set and result is compared	Produce the right cipher text based on the plain text and key given
ahb_aes1_overlap_test	Overlap 2nd key and plain.	The encryption should be done using the new key and plain text
ahb_aes1_change_text_key_encryption_test	Change the text and key during encryption.	The encrypted cipher is expected to use the previous key instead of the new key
ahb_aes1_less_plain_key_test	Provide plain3 and key3 to start encryption without text 0,1,2 and key 0,1,2.	Plain and key 0,1,2 is expected to set to default 0 or previous set value.

A.6 *AHB-Lite Bus*

AHB-Lite Bus is used to connect the AHB components to the Cortex-M0 core to form an AHB system. Test cases has been develop to ensure that the AHB-Lite Bus decodes the correct slave select signals and the corresponding slave transfer is passed back correctly. Various types of AHB transfers are also used to verify the decoding process. The various AHB Lite Bus tests are shown in Table 2-8.

Table 2-8: AHB-Lite Bus Verification Plan

Tests	Test Descriptions	Verification Criteria
ahb_lite_bus_sw_address_map_and_transfer_test	AHB Master access all 7 slaves connected to the ahb lite bus including default slave and check for the hsel signals	Expects the AHB sends the right slave select signals and reads the slave data correctly.
	AHB Master reads hrdata, and hready from the slaves and compare with the master	
	Enable switch mode to perform address remap to switch between boot-load and normal mode	
ahb_lite_bus_sw_multi_transfer_test	AHB Master sends multiple type transfer to all the 7 slaves: single write, single read, single unpipelined write , single unpipelined read, incr read, incr write, incr write with busy, incr read with busy, incr_n write burst, incr_n read burst, incr_n write burst with busy, incr_n read burst with busy, burst_n read burst, burst_n write burst, burst_n write burst with busy, and burst_n read burst with busy	Expects the AHB-Lite bus to be able to handle various read and write transfers
ahb_lite_bus_sw_random_multi_transfer_test	AHB Master randomly send multiple type transfer to all the 7 slaves: single nonseq unpipelined, single nonseq, single idle, incr, incr_n, and wrap_n transfer	Expects the AHB-Lite bus to handle random read and write transfers correctly