AUTOMATED VERIFICATION OF DIGITAL IP

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

In VLSI Design

Submitted By

Ankit Jain

601762003

Under Supervision of

Dr. Manu Bansal

Assistant Professor

THAPAR INSTITUTE

OF ENGINEERING & TECHNOLOGY

(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

JULY, 2019

STMicroelectronics PVT. LTD.

Greater Noida, Uttar Pradesh 201308, India

CERTIFICATE

This is to certify that **Ankit Jain** (Registration No. 601762003), a student of M.Tech. (VLSI Design), **Thapar Institute of Engineering and Technology, Patiala** has completed one year (June 2018 – May 2019) internship program in **STMicroelectronics Pvt. Ltd., Greater Noida**. His title of dissertation is "**Automated Verification of Digital IP**".

MR. G.N. CHOUDHARY

Principal Engineer

STMicroelectronics Pvt. Ltd.

DECLARATION

I, **Ankit Jain** hereby declare that the work presented in this thesis entitled **"Automated Verification of Digital IP"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at **Electronics and Communication Engineering Department**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of **Dr. Manu Bansal (Assistant Professor, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology)** from **June 2018** to **May 2019**. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date: 5/8/2019

ANKIT JAIN

601762003

It is certified that the above statement made by student is correct to the best of my knowledge and belief.

Date: 5/8/2019

Dr. MANU BANSAL

Assistant Professor

Department of Electronics and Communication Engineering

Thapar Institute of Engineering & Technology

(A deemed to be University), Patiala, Punjab

ACKNOWLEDGEMENTS

I express my heart full indebtedness and owe a deep sense of gratitude to Dr. Manu Bansal, Assistant Professor, Electronics and Communication Department, Thapar Institute of Engineering and Technology, Patiala for her guidance and support with encouragement to go ahead. I would like to express my gratitude and sincere thanks to my advisor Mr. GN CHOUDHARY, Principal Engineer, STMicroelectronics Pvt. Ltd. for his guidance and review.

I am also thankful to Dr. Alpana Agarwal, Head of Department, Electronics and Communication Engineering Department (ECED), Dr. Hem Dutt Joshi, PG Coordinator and Dr. Anil Arora, Program Coordinator, ECED and the entire faculty and staff of Electronics and Communication Engineering Department. Last but not the least, I would like to thank my parents for their years of unyielding love and encourage. They have always wanted the best for me and I admire their determination and sacrifice.

Ankit Jain

ABSTRACT

IP reuse in SoC has led to faster time to market. However, to cop up with this pace, new practices need to be followed to verify the functionality of IP derivative. These blocks or IPs must be verified independently before shipping to ensure proper working and conformance to protocols that they are implementing. But, since the application of all IPs will vary from SoC to SoC, the verification environment must consider the important features and functions that are critical for that application. Universal verification methodology is the latest verification methodology being followed. To reduce communication overhead between designer and verification engineer, the setup of verification environment can be automated, thus eliminating human intervention which can introduce some errors and providing designer an easy interface to test the IP initially.

For this, bus functional model of a master bus interface need to be created to initiate the transactions on IP. These models drive slave interface of the IP in order to confirm the protocol is being followed

Every IP being created is packaged using IP-XACT standard (xml format). Along with xml files, generators are created which extract the information from these xml files and generate the various files like C Hardware Abstraction layer, document of memory map, RTL of memory map, etc. In order to increase the productivity of verification at designer end, IP-XACT can be processed and used to verify the functionality of registers. Using IPXACT xml file, all the data related to registers is extracted and a structured database is created in system verilog language. This data can be used for the verification of registers to indicate functional errors on simulation console screen.

TABLE OF CONTENTS

CERTIFICATE……………………………………………………………………………………….ii
DECLARATION.. iii
ACKNOWLEDGEMENTS iv
ABSTRACT v
TABLE OF CONTENTS vi
LIST OF TABLES viii
LIST OF FIGURES.. ix
LIST OF ABBREVIATIONS x
1. INTRODUCTION..1
1.1 BACKGROUND..1
1.2 FUNCTIONAL VERIFICATION...2
1.2.2 SYSTEM VERILOG...4
1.3 UNIVERSAL VERIFICATION METHODOLOGY (UVM)...5
1.3.1 UVM TESTBENCH ARCHITECTURE 6
1.3.2 UVM REGISTER ABSTRACTION LAYER 9
1.3.3 FUNCTIONAL COVERAGE AND ASSERTIONS 9
1.4 IP-XACT STANDARD..10
1.5 LITERATURE REVIEW 11
1.6 PROPOSED METHODOLOGY..13
2. BUS INTERFACES 14
2.1 BUS FUNCTIONAL MODEL...14
2.2 AMBA BUS PROTOCOLS 14
2.2.1 AMBA APB PROTOCOL 15
2.2.2 AMBA AHB PROTOCOL..17
2.2.3 AMBA ATB PROTOCOL 22

2.3 IP INTERFACE 24
3. SCRIPTING 27
3.1 TESTBENCH SETUP 27
3.2 IPXACT MEMORY MAP 29
4. SIMULATION RESULTS 32
4.1 BUS TRANSACTIONS 32
4.1.1 IPS INTERFACE 32
4.1.2 APB INTERFACE 33
4.1.3 ATB INTERFACE 34
4.2 TESTBENCH SETUP 35
5. CONCLUSION AND FUTURE SCOPE 36
5.1 CONCLUSION 36
5.2 FUTURE SCOPE 36
6. REFERENCES 37
7. APPENDIX 39
7.1 IPS Driver BFM 39
7.2 APB Driver BFM 41
7.3 AHB Driver BFM 48
7.4 ATB SLAVE MODEL 55

LIST OF TABLES

Table 2.1 List of APB signals.....16
Table 2.2 List of AHB signals.....19
Table 2.3 List of ATB signals.....23
Table 2.4 List of IP Interfaces.....24
Table 2.5 IPS Address Bus Range.....25

LIST OF FIGURES

Figure1.1 DUV integration with testbench....3
Figure1.2 Self-checking testbench integration with DUV 3
Figure1.3 Verification methodology timeline 5
Figure1.4 UVM class hierarchy 6
Figure1.5 A typical UVM platform structure....7
Figure1.6 Field Access polcies....12

Figure 2.1 ARM AMBA versions 15
Figure 2.2 Basic APB transfer....15
Figure 2.3 APB operating states....17
Figure 2.4 BLOCK DIAGRAM FOR AHB INTERCONNECT 18
Figure 2.5 AHB slave interface ports....19
Figure 2.6 Single AHB transfer....20
Figure 2.7 Multiple AHB transfers....20
Figure 2.8 ATB components 22
Figure 2.9 Basic ATB transaction 23
Figure 2.10 Clocking strategies of IPS interface....25

Figure 4.1 IPS read transactions....32
Figure 4.2 IPS write transactions....32
Figure 4.3 IPS write-read transactions 33
Figure 4.4 APB back to back read transactions....33
Figure 4.5 APB back to back write transactions 33
Figure 4.6 APB back to back write–read Transactions 34
Figure 4.7 Constrained random sequence generations 34
Figure 4.8 ATB data transaction 34
Figure 4.9 Input console for testbench setup....35
Figure 4.10 Coverage Report 35

LIST OF ABBREVIATIONS

AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ANSI C	American National Standards Institute for the C programming language
APB	Advance Peripheral Bus
API	Application programming interface
ATB	Advanced Trace Bus
BFM	Bus Functional Model
CDC	Clock Domain Crossing
DUT	Design under Test
DUV	Design under Verification
I/O	Input/Output
IP	Intellectual Property
IPI	Intellectual Property Interface
RAL	Register Abstraction Layer
RAT	Register Access Technology
RTL	Register-Transfer Level
RW1	Read Write Once
SoC	System on Chip
TLM	Transaction level modeling
TTM	Time to Market
UVM	Universal Verification Methodology
VC	Virtual Component
VIP	Verification Intellectual Property
W1	Write Once
XML	eXtensible Markup Language

CHAPTER 1

1. INTRODUCTION

1.1 BACKGROUND

The repetitive reuse of IP in SoC designs has enabled fast implementation of designs with different parameters. To design a SoC in time without affecting the quality, large number of IP components are to be integrated. In order to achieve this level of integration, IP used must not contain any defects or bugs. This require all the verification of all the features claimed by all the IPs used in a SoC.

Due to variation in SoC implementation, each IP has to be verified separately for the claimed features and protocol used with the specified parameters. This means each time IP is used, for verification testbench has to be developed again. This activity takes large time forming a major part in design flow, bringing down the productivity and thus impacting the TTM.

Since an SoC uses components having standard bus interface for interconnection like AMBA, integrating processors and other IPs becomes easy. So if IP used is verified, while assembling them designer is assured the IPs instantiated will follow the protocols.

However, with the rising complexity of design, coverage of functional verification is a major challenge faced by both design and verification teams. Design verification (DV) is a large and complex domain that contains many technologies, languages, and methodologies. Decreasing the time consumed for verification without impacting the quality of the verification is a big challenge. To achieve this, verification environment is to be set up as fast as possible.

While designing a module, designer has to perform verification to check that implementation is going on a right track. The task to set up a testbench through which a designer may easily perform verification of certain functionalities can be automated with the scripting.

1.2 FUNCTIONAL VERIFICATION

The objective of verification is to ensure that the design is implemented correctly as claimed in specification. Verification consumes 70% to 80% of the effort in design cycle and is an important step in the design development, so verification becomes a step in the design process which can bring problems. The functional verification becoming an obstruction is an effect of ever rising design complexity. Majority of ASICs implemented go through a minimum of one additional tape out with major number of tape outs are due to presence of bugs related to functionality.

It is very important to arrange and plan the verification. If not done, it becomes impossible to carry out and demonstrate that verification is done correctly. It is essential that all the features are declared and verification completion is monitored with the errors found in the design. If either of design or verification is incomplete, the design cannot be send for tape out.

Verification is carried out by driving the design under verification (DUV) with directed or random stimuli [1]. With the help of a RTL simulator the modeled design can be exercised with the applied patterns and the outcomes of applied pattern are obtained and later on can be inspected for debugging purpose if required.

Directed test pattern use fix stimuli. This yields same output every time design is simulated. Directed testing is needed in case of verifying a specific functionality of IP. The stimuli values are stated by the design engineer who made the test case. The problem with static patterns is that it can’t be ensured the functionality is verified for every scenario. It might be possible the design may break under certain operating scenario.

Random test generates randomized patterns for verification. The important thing in random pattern generation is that patterns generated should be meaningful, so stimuli generation needs to be constrained. To do so constraints are developed, decreasing the possible values of patterns. Then with the help of constraint solver the new sets of test patterns are generated each time which are somewhat randomized.

In both cases, randomized and directed tests, the design under verification is integrated with a testbench. The testbench generates the required global signals like clock, reset and other signals and drive them to the design under verification (DUV). A diagram of the DUV and testbench is shown in Figure 1.1.

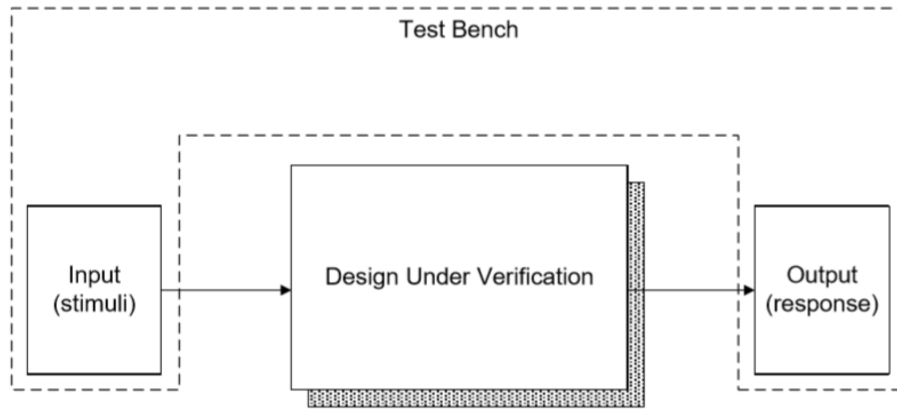

Figure 1.1 DUV integration with testbench

It is important to design a testbench which drive DUV with test patterns, sample output response and compare the result with a reference model which generates a correct response for the applied stimuli to DUV (as in Figure 1.2).

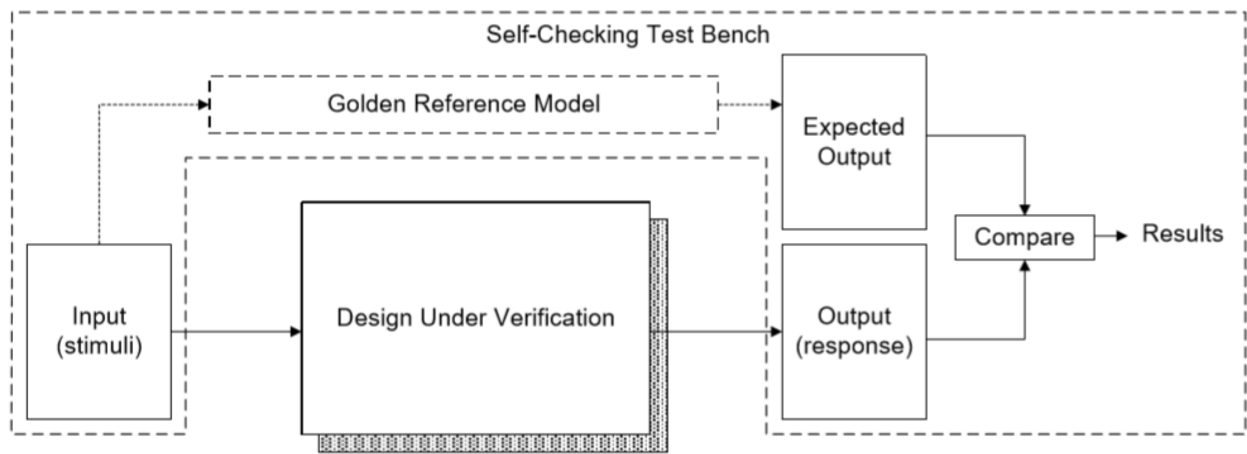

Figure 1.2 Self-checking testbench integration with DUV

In case of directed test pattern, the stimuli are always the same and the response of reference model is also repeatedly the same. In randomized tests, different test patterns are obtained each time. Thus using random stimuli to drive DUV, output values obtained will be also different which cannot be known in advance. For this a reference model is needed which generated correct outputs for comparison. The reference model is regarded as ideal, thus also known as Golden Reference Model.

1.2.2 SYSTEM VERILOG

System Verilog is a standard language maintained by IEEE 1800 standard [2], for modelling hardware as well for verification of hardware designs. System verilog incorporated many C features bringing concepts like object oriented programming.

The IEEE formed a fresh language standard number, 1800-2005, and named it System Verilog, in order to explain the new features added in language. System Verilog isn't a different new language; it is just an add-on of Verilog language constructs.

In 2009, Verilog (2005) and System Verilog (2005) were merged and a single standard was formed. "The original 1364 Verilog standard was terminated, and the IEEE ratified the 1800-2009 SystemVerilog-2009 standard as a complete hardware design and verification language. In the IEEE nomenclature, there is no longer a current Verilog standard" [3]. There is only a System Verilog standard.

System Verilog provides support for modelling hardware at behavioral, register transfer level (RTL) and gate-level abstractions. System Verilog also provides support for writing testbenches using object-oriented programming.

There are four language categories that comes under the System Verilog:

1. System Verilog Object Oriented language for functional verification
2. System Verilog language for Design
3. System Verilog Assertions (SVA) language, and
4. System Verilog Functional Coverage (FC) language

With the use of system verilog, the 3 main parts of verification can be defined:

1. Stimuli generation using OOP concepts.
2. Response Checkers to check that design is giving right outputs.
3. Coverage collection to ensure that design is verified for the intended scenarios.

1.3 UNIVERSAL VERIFICATION METHODOLOGY (UVM)

Verification methodology is a set of coding practice adopted by a group of people or a group of company. Universal Verification Methodology (UVM) [4] evolved because of the need of unified verification framework as verification engineers across different industries and regions were using different verification environment based on the methodology developed by them. Moreover, not all simulation tool vendors are compatible with each and every methodology. This blocks the reusage of verification environment and development of verification IPs (VIP).

A standards organization for electronic design automation (EDA) and IC manufacturing, Accellera Systems Initiative, decided to establish the UVM in 2009. Open Verification Methodology from Cadence and Mentor was chosen as a base where the UVM is built on. UVM is created in co-operation between several companies in verification industry. Nowadays Mentor Graphics, Synopsys, Cadence and Aldec are co-operating in UVM development.

UVM is made on top of Open Verification Methodology (OVM) which is a merger of Advanced Verification Methodology (AVM) and Universal Reuse Methodology (URM). UVM also incorporates ideas of e Reuse Methodology (ERM) and some ideas and code of Verification Methodology Manual (VMM). Figure 1.3 displays the evolution timeline of various verification methodologies. It can be seen that UVM contains many ideas or features of various verification methodologies.

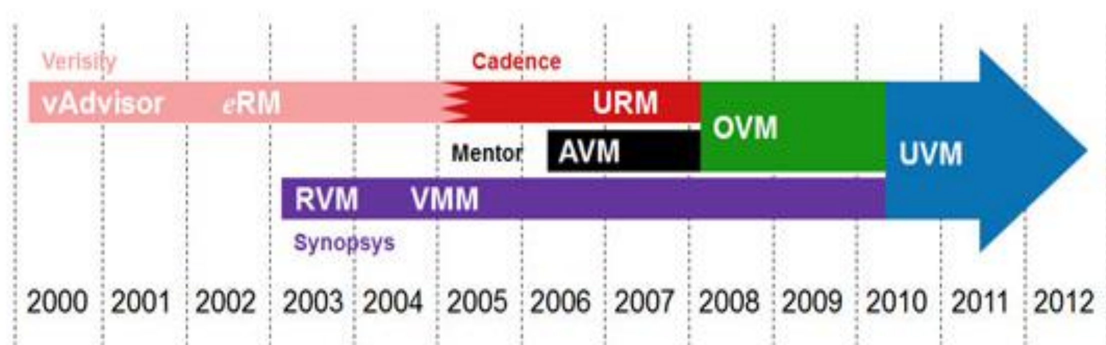

Figure 1.3 Verification methodology timeline

UVM is a library of classes developed in System Verilog. It provides common, reusable architecture for developing verification components and allows vertical & horizontal reuse (in IP/SOC & in different projects).

There are a total of 357 classes, 938 functions, 99 tasks and 374 macros [5] present in UVM library. Since UVM evolved from OVM and VMM, thus it encapsulates new and old methods in order to obtain same goal. Also in UVM, parent classes can be extended from child classes. Figure 1.4 displays some of the child classes with their relationship. A reusable verification environment can be swiftly setup by using these classes that are already available in library.

Figure 1.4 UVM class hierarchy

1.3.1 UVM TESTBENCH ARCHITECTURE

Using merely 3 percent of the library classes, UVM based testbench architecture [1] can be setup as shown in figure 1.5. As a UVM testbench can be made from reusable VIPs [6], so it is possible to use VIP's from another vendor in the testbench to speed up the verification testbench setup. The basic blocks present in a UVM testbench are described on the following page.

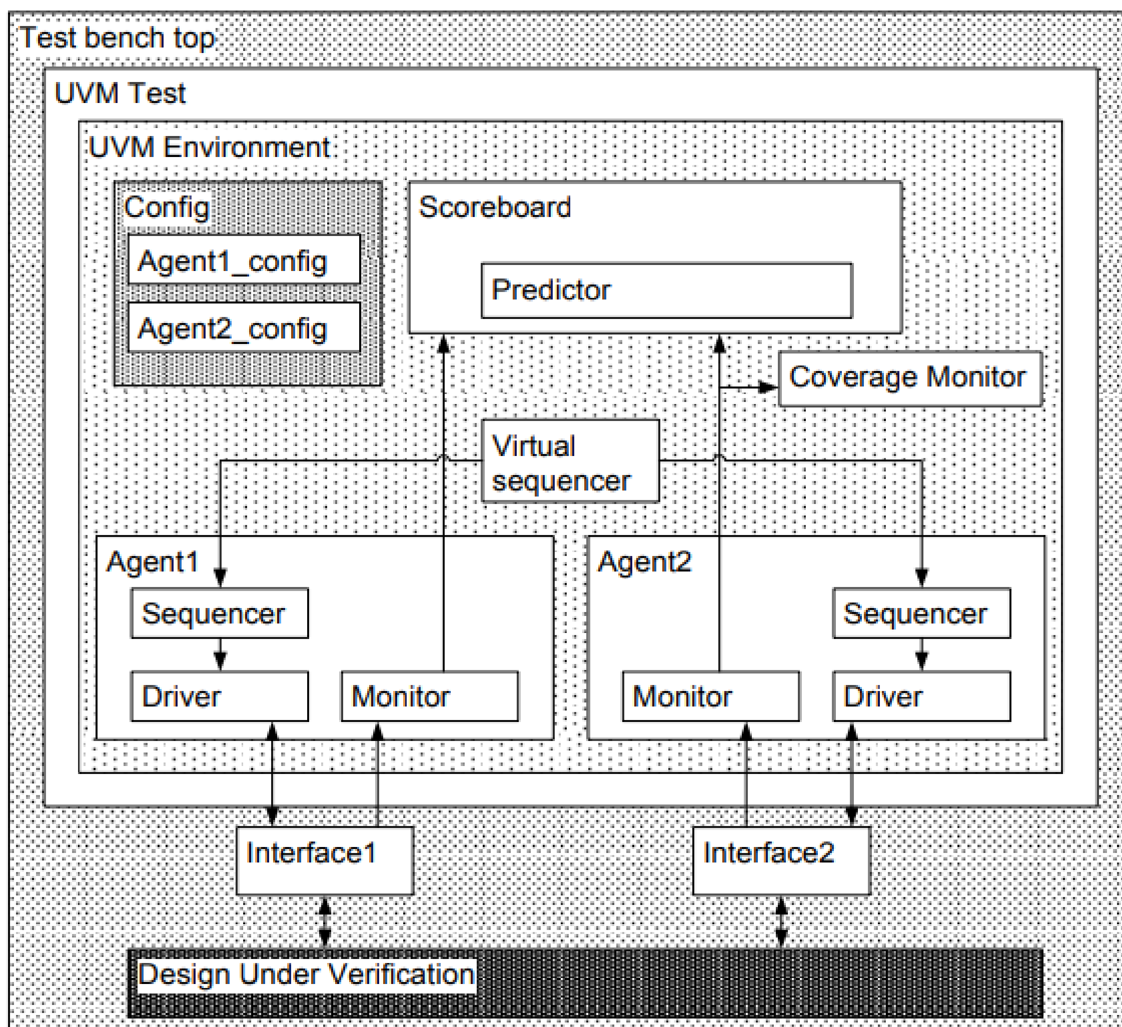

Figure 1.5 A typical UVM platform structure

a) Testbench Top

This module instantiate the DUV, and the corresponding interfaces. The interfaces are assigned to suitable respective ports of DUV and they are stored in UVM configuration database. With the help of this database, components of UVM produced in run time can be configured.

b) DUV

The DUV refers to the IP whose functionality is to be verified.

c) UVM Test

UVM Test configures and creates the UVM Environment. UVM Environment configures the encapsulated blocks with the configuration driven by UVM Test.

d) UVM Environment

The environment includes agents, scoreboard, coverage collector, virtual sequencers, configuration databases, etc.

e) Agent

Agent is categorized as active or passive. Active agent contains all the three components driver, sequencer and monitor. A passive agent only contains the monitor, i.e. it only probes DUT port for sampling, but doesn't provide pattern to the ports.

f) Sequencer, Driver and Monitor

Sequencer is the component that handles generation of sequence and driving the interfaces with that sequence. Since an DUT can be required to be driven by multiple sequences, virtual sequencer is used to schedule the sequence with agents. The transaction may consist a case where a write and read is to be performed. In this case a virtual sequencer may generate sequence that is read followed by write. The driver drives the DUT ports and monitor samples the driven port as well port where data is expected to perform a comparison.

g) Coverage and Scoreboard

Scoreboard is a comparator that compares output values received through the monitor with a value generated by testbench to verify if the output of DUT is correct. Coverage collector provides coverpoints to track the coverage report, if the DUT is verified in regression with constrained random inputs application.

h) Sequence item

This component is made up of data fields required for input vector production. Input patterns are produced by randomization. Constraints are also added to limit the set of stimuli produced.

i) Sequence

Sequence generate series of sequence items that is passed to driver through sequencer.

To promote re-usability, the lower level environments provide easy methods to reduce the effort for configuring the environment of upper level.

1.3.2 UVM REGISTER ABSTRACTION LAYER

The UVM register model gives a way to monitor the data of register in a DUV and a convenience layer for accessing register and memory locations within the DUT.

The register abstraction is the structured database of a hardware-software register specification, since that is the basic starting requirement for starting a hardware design and for starting verification. Moreover, this data is also used by software engineers to make software. It is very essential that all the three teams always have a reference to the common data specified and it is also important that the design is verified with a correct model.

The UVM register model is designed to facilitate productive verification of programmable hardware. The information captured in register model is divided in 5 categories: field, register, memory, register block, and register map. UVM Register Abstraction Layer (RAL) can be generated with IP-XACT model capturing the data related to the registers of the design under verification (DUV).

The UVM RAL allow usage of UVM built in register tests such as bit bash test and reset test. The reset test checks the default values of the registers. Bit bash test performs toggled write value on all bits one by one and then reads the register.

1.3.3 FUNCTIONAL COVERAGE AND ASSERTIONS

The goal of functional coverage is to determine if the implemented design requirements defined in the specification are working as intended. In addition, functional coverage is a good indicator to measure verification progress, better than code coverage. From a high-level perspective, there are two main steps present when creating a functional coverage model: identifying the functionality or design to be verified, and implementing the coverage model [7] using language features to measure the intended functionality or design.

The disadvantage of functional coverage is that it can’t be extracted automatically, a lot of manual work is required [8]. Through system verilog, the implementation of coverage models can be performed with covergroups, or with assertions.

A coverpoint covers a variable or an expression, and it includes a set of bins associated with its sampled values or its value transitions. The bins can be automatically generated by the tool or manually declared. Automatic generation is useful when creating separate bins for a range of values. A bin will be marked as covered if the value associated to the bin is present in the coverpoint variable at the covergroup sampling

event. Cross coverage can also be created between all the bins for the chosen coverpoints. A covergroup can contain one or more coverpoints.

An assertion specifies excepted behavior in a design and it is mainly used to validate that the behavior is correct. In addition, the cover directive found in assertions can be used to provide functional coverage. Assertions are not as practical as covergroups at checking data values, delays and multiple data points, but they are useful at detecting the occurrence of some specific series of Boolean values.

There are two types of assertions: immediate and concurrent assertions. Immediate assertions are simple, non-temporal domain assertions that are executed like statements in a procedural block. Immediate assertions can only be specified with a procedural statement and the evaluation happens immediately with the values updated at that moment. On the other hand, concurrent assertions describe behavior that spans over time and are great at verifying specific sequences. The evaluation model is based on a clock and the assertions are evaluated only at the occurrence of a clock tick.

1.4 IP-XACT STANDARD

IP-XACT is a standard created by spirit consortium, now a part of IEEE 1685 [9]. IP-XACT follows schema as defined in XML. The main purpose of this standard is to allow sharing of data related to IP easily across different groups or companies. IP-XACT also provides standardized API for manipulation of data, called as tight generator interface.

It is essential to observe that xml don’t perform any operation, it just captures data systematically, making exchange of data easy. With the help of TGI which is a script, required data from the design can be extracted and also design can be configured.

A component has many attributes that is captured separately in xml. It consists of:

- Memory maps
- Registers
- Bus interfaces
- Ports
- Views (additional data files)
- Parameters
- Generators

When multiple components are connected, it becomes an IP-XACT design file. This includes the attributes above, as well as the interconnect information of all the components in the design.

1.5 LITERATURE REVIEW

Kwanghyun Cho et al. [10] have shown how the productivity in implementing an SoC is increased if IPs are reused. They have developed flow for design and verification based on the fact that using standardized bus protocols reduces integration time for hundreds of IP. They have reported a reduction of 30% time for 1st implementation and 50% time reduction for the derived platforms.

Hung Yi Yang [11] discusses the challenges and difficulties faced in detecting bugs in design. As the complexity of design is increasing, verification cycle takes more time to complete. This caused the evolution of UVM which provides a defined way to set up testbench, but this requires handling of simulations effectively. Even employing UVM for verification, it is possible that chip has to go for re spin due to bug related to functionality, which causes loss in terms of cost as well as in TTM.

Amr Hany et al. [12] has proposed to use assertions and coverage for verification together in the flow. In order to accelerate the time needed to attain the goal of coverage closure, simulation is to be carried out with a planned verification plan. Assertions help in catching the bugs while coverage report gives the quality of verification carried out. They have also combined the use of CDC verification along with power domain verification in single flow to create the test setup.

Gaurav Sharma, Lava Bhargava and Vinod Kumar [13] have reported that traditional verification methods fails to give the result in terms of computing resources used due to increase in complexity of design. This causes more and more registers to be incorporated in a SoC or IP. They have created a script to automate the verification of registers implemented in a design, including all the attributes associated with the registers. They are creating a UVM based testbench while including coverage and RAT, and used questasim for simulation. They have reported a 100% functional coverage and improved the disk usage in comparison to verification carried out using system verilog and UVM.

Namdo Kim et al. [14] have automated the setup of testbench for verification of registers at chip top level consisting of 100M gate counts. With this they have successfully created a testbench consisting of 2 million lines of code, utilizing data in excel. However the test cases are to be added manually. Moreover the testbench created is configurable which helps in saving simulation time, as result they managed to increase performance by 50%. Also they have used appropriate naming, decreasing learning curve of designers. It becomes easy to manage and debug the errors, thus allowing them to close the project quickly and start a new one.

Ashutosh Kumar Singh, G.S. Tomar and Anurag Shrivastav [15] have discussed the various protocols offered by AMBA and compared them for their performance. They have highlighted the fact that interconnection of IPs for exchanging information on a platform poses various challenges and can

also impact the performance of chip. Using an on chip interconnect solutions such as those provided by AMBA, integration can be carried out fast with correct implementation in first time. They have also described various versions of AMBA protocols.

D. Shin, S. Abdi and D.D. Gajski [16] have discussed the creation of BFM from TLM, created at higher level of abstraction. Since for exploring design space simulation has to be fast, thus TLM models are used initially to find optimal bus interface. But TLM don't contain any timing related information, thus buses are to be modeled as BFM. For this, algorithms and methodologies are created by them to automate the transformation of buses used from TLM to BFM, thus also eliminating the chances of error and saving the time.

V. Berman [17] has described the standardized method of storing the information related to IPs using an xml format, developed by the consortium of SPIRIT, called as IP-XACT. Now this standard is developed and maintained by IEEE. IP-XACT uses schema to structure the stored database so that data can be easily exchanged. Also the tools implementing this standard are required to provide APIs for the integration, configuration of IPs. Moreover, required data can also be extracted using these APIs.

Mark Litterick, Marcus Harnisch [18] has described working of register model in UVM environment. They have shared best practices to test registers, fields, their effects in UVM environment. They have summarized the access policies for fields as shown in figure 1.6

	NO WRITE	WRITE VALUE	WRITE TO CLEAR	WRITE TO SET	WRITE TO TOGGLE	WRITE ONCE
NO READ	-	WO	WOC	WOS	-	WO1
READ VALUE	RO	RW	WC W1C W0C	WS W1S W0S	W1T W0T	W1
READ TO CLEAR	RC	WRC	-	WSRC W1SRC W0SRC	-	-
READ TO SET	RS	WRS	WCRS W1CRS W0CRS	-	-	-

Figure 1.6 Field Access polcies

Haytham Saafan, M. Watheq El-Kharashi and Ashraf Salem [19] have addressed the challenges of integrating IPs I/O or bus interfaces. They have suggested to use formal verification to verify the interconnects, but this requires storage of bus details in some standard format. They have suggested to use IP-XACT for capturing data and in case no formal data is captured then netlist of previous version can be used for comparison of interconnections.

1.6 PROPOSED METHODOLOGY

As the complexity of a chip design is increasing, it is no longer feasible to setup a testbench manually. In order to reduce the communication between designer and verification, a method is defined to automate the setup of testbench. Since IPs design used one of the many on chip bus interconnection protocol. The bus functional model of some standard protocols used are designed and incorporated in testbench. But if required models for other bus interfaces can also be designed and easily incorporated in the automated method. For the purpose of automation python is used as it provides many built in libraries and have a short learning curve.

The initial verification starts with the checking of registers implementation. The designer also captures all the information related to registers in a structured format called IP-XACT (an industry standard). A script is created to extract the information from this higher level of abstraction data to system verilog data structure. Using this structure directly in testbench, the verification of registers can be automated. Moreover, the testbench created by script is configurable for register testing, i.e. if required designer can run read only test, write and read test, or byte access test.

After verifying the registers, designer can perform some basic functionality test to ensure correctness of design before passing it to verification team. Also for all the test cases run, the coverage reports can be merged and analyzed.

CHAPTER 2

2. BUS INTERFACES

In order to facilitate the communication between different digital intellectual property blocks on chip, a protocol is followed ensuring the exchange of data takes place without any error. Many different protocols targets different challenges like pipelining, low power communication, burst transactions, etc. The bus interface can be said to be in either slave mode or master mode. An IP can have multiple bus interfaces to establish connection with the external components in a system on chip. The majority of IPs uses open bus interface protocols to increase the reusability. Since verification of IP has to be carried out as a standalone component, bus protocols used in an IP are needed to be modeled.

2.1 BUS FUNCTIONAL MODEL

Since for verification purpose, external components are not required to be synthesizable, but are rather required to initiate a transaction or respond to the initiated transaction. For this purpose, Bus Functional Model (BFM) is used which are the tasks written in hardware description language such as verilog, system verilog, system C, etc. Thus the significance of these models is to simulate the bus transactions before synthesizing and testing software on the actual silicon hardware. At receiving side of IP, correspond to the protocol BFM samples signal and at sender side of BFM, tasks creates transactions according to the protocol.

Once these models are created, they can be reused again for verification by connecting the design under verification interface ports to the corresponding model. Simulating the IP using these models gives designer confidence that the RTL being made is implemented correctly.

2.2 AMBA BUS PROTOCOLS

ARM provides specification of different types of bus protocols for interconnection of blocks in a SoC under name of Advanced Microcontroller Bus Architecture (AMBA) .Many of the protocol specification provided are open, which helps in utilization of IP implemented using these protocols by different vendors easy. Thus a large and much complex functionality can be implemented reusing these IP along with ensuring success at first time. The various protocols specification released under AMBA family can be seen in the figure 2.1.

- **AMBA 1: 1996**
  - Advanced System Bus (ASB)
  - Advanced Peripheral Bus (APB)
- **AMBA 2: 1999**
  - AMBA High-performance Bus (AHB)
  - Advanced System Bus (ASB)
  - Advanced Peripheral Bus (APB2 or APB)
- **AMBA 3: 2003**
  - Advanced Extensible Interface (AXI3)
  - Advanced High-performance Bus Lite (AHB-Lite v1.0)
  - Advanced Peripheral Bus (APB3)
  - Advanced Trace Bus (ATB v1.0)
- **AMBA 4: 2010**
  - AXI Coherency Extensions (ACE)
  - AXI Coherency Extensions Lite (ACE-Lite)
  - Advanced Extensible Interface (AXI4)
  - Advanced Extensible Interface 4 Lite (AXI4-Lite)
  - Advanced Extensible Interface 4 Stream (AXI4-Stream v1.0)
  - Advanced Trace Bus (ATB v1.1)
  - Advanced Peripheral Bus (APB4 v2.0)
- **AMBA 5: 2013**
  - Advanced High-performance Bus (AHB 5, AHB-Lite)
  - Coherent Hub Interface (CHI)

Figure 2.1 ARM AMBA versions

2.2.1 AMBA APB PROTOCOL

The Advanced Peripheral Bus (APB) [20] protocol bus is used in slave IPs which has very less bandwidth requirement and where power used is of prime importance. The APB provides a very simplistic interface to the slave IP. The APB interface has separate data buses, thus allowing either write or read to take place in a single transaction. The APB protocol requires minimum two clock cycles to complete a transaction as shown in figure 2.2.

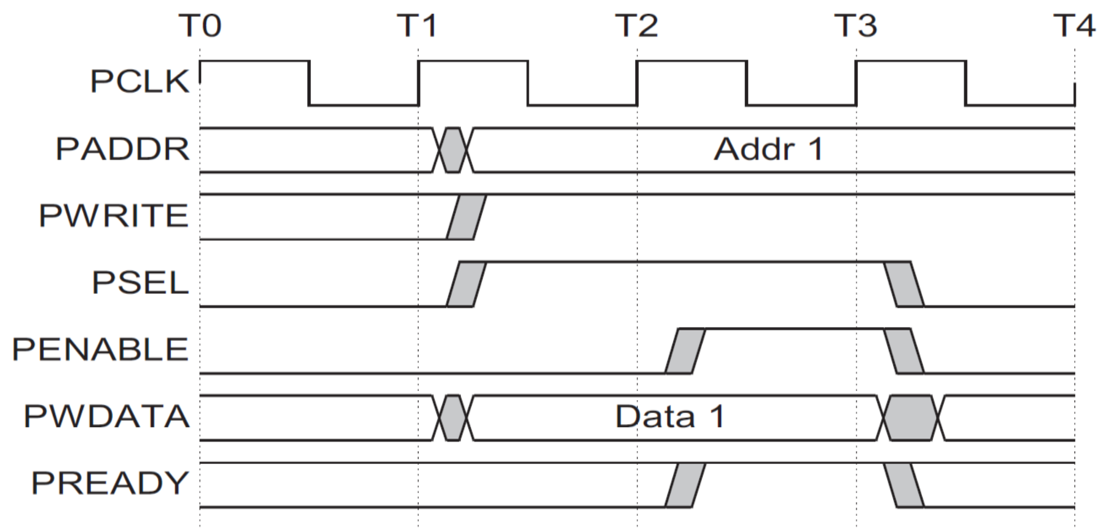

Figure 2.2 Basic APB transfer

At T1, master drives PSEL, PADDR and PWRITE. These signals are needed to be kept stable until transfer is completed. During T1-T2, the address decoding takes place, which should be completed in same cycle. On following next clock cycle at T2, master must drive PENABLE high. In this cycle, if slave drives PREADY high then transfer is done within the same cycle but if PREADY is driven low, the transfer is extended until PREADY is driven high. The signals used in APB are listed in the table 2.1, along with their description.

Table 2.1 List of APB signals

Signal Name	Description
PCLK	Defines system clock frequency of IP, all transfers are synchronous to positive edge of the clock.
PRESETn	Defines reset for signal, it is an active low signal.
PSEL	Selects IP for transaction.
PENABLE	Indicates next cycle of transfer after address decoding.
PADDR	Provides address of register for transaction.
PWRITE	Selects register for read when low and for write when high.
PWDATA	Provides data to be written in register.
PRDATA	Provides data read from register.
PREADY	Indicates if IP status, high when not busy. Can be used to stretch the transaction for some cycles of clock.
PSLVERR	Indicates if transfer gets failed.
PPROT	Provides the transaction security level.
PSTROBE	Provides which bytes used for write data transaction.

The functioning of APB protocol can be broken down in a FSM as shown in figure 8. There are three states in this FSM, listed below:

1) IDLE: When IP is out of the reset and waiting for a transfer to start, then APB bus is said to be in IDLE state.
2) SETUP: When IP is selected for the transfer, i.e. the PSEL of this IP's APB bus interface is driven high, thus the APB bus comes under SETUP state. In this state, address decoding is to be completed within the same clock cycle. This bus remains in this state for single cycle only.
3) ACCESS: After SETUP state the FSM moves to ACCESS state when PENABLE goes high. During the transition from SETUP to ACCESS state, select, write, write data and address must remain constant.

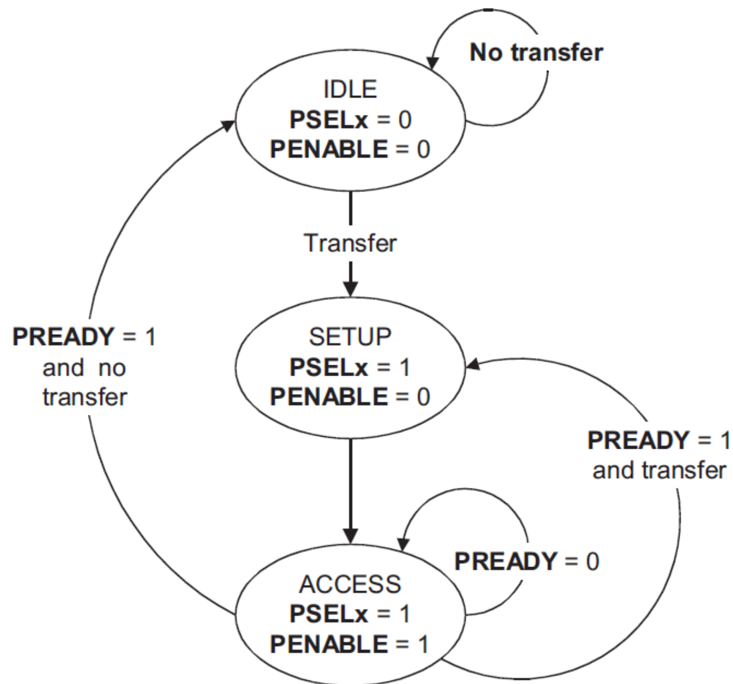

Figure 2.3 APB operating states

To move from ACCESS state, the dependency remains on the PREADY signal generated by the slave IP. If PREADY is kept low by the IP, then transfer is extended and FSM is retained in same state. But, if PREADY is driven high, FSM exits from this state either to IDLE or SETUP depending on if back to back transfer is to be carried out.

2.2.2 AMBA AHB PROTOCOL

AHB [21] protocol is used when high performance is required between masters and slave. On a system level, some IPs can become a bottleneck, to avoid such situations this bus protocol is used as it can do burst transfers and pipeline the transfers increasing the overall speed of transfers. Moreover AHB supports data buses of more than 32 bits, increasing the bandwidth. AHB interconnect doesn't contain any tri-state logic. Also a transaction can be split in case a slave is taking multiple cycles to complete. Generally it is recommended to split the transaction if more than 16 cycles are going to be required in order to prevent stalling of system.

Figure 2.4 shows the components used in a AHB interconnect.

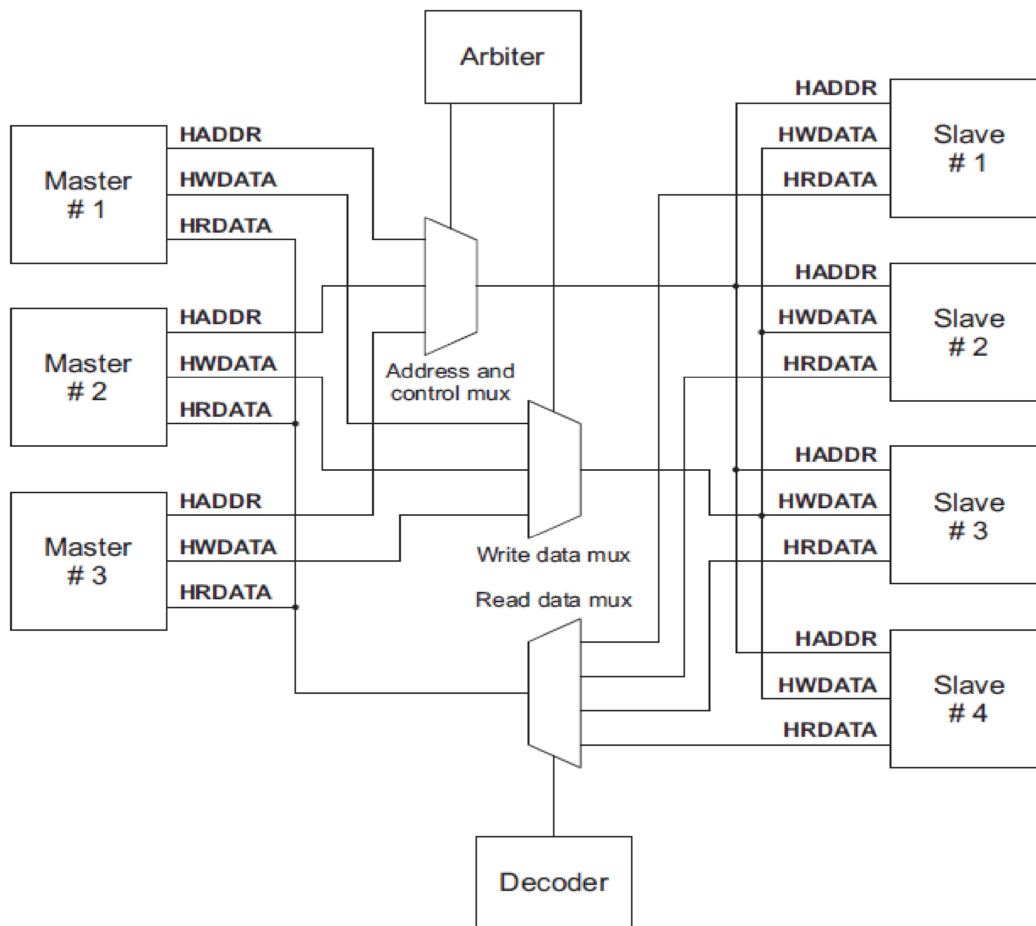

Figure 2.4 BLOCK DIAGRAM FOR AHB INTERCONNECT

The components present on an AHB interconnect are:

1) Masters: AHB supports multiple masters, but at a time only one master can access a component.
2) Slaves: Multiple slaves can be connected to AHB bus system. However only a single slave responds to read/write transaction.
3) Arbiter: The arbiter selects which master should be granted access in case multiple masters assert request for transaction.
4) Decoder: MSB bits of address are used to decode which slave is being accessed. So the address and control signals reaching to all slaves can be identified for which slave those signals are to be used.

The AHB interface can be used for single master, for this a centralized multiplexer and decoder is required on interconnect. Whereas for multiple masters support, an arbiter and multiplexers are required on interconnect. For IPs designed to be used as slave, figure2.5 illustrates signals used by a slave AHB interface.

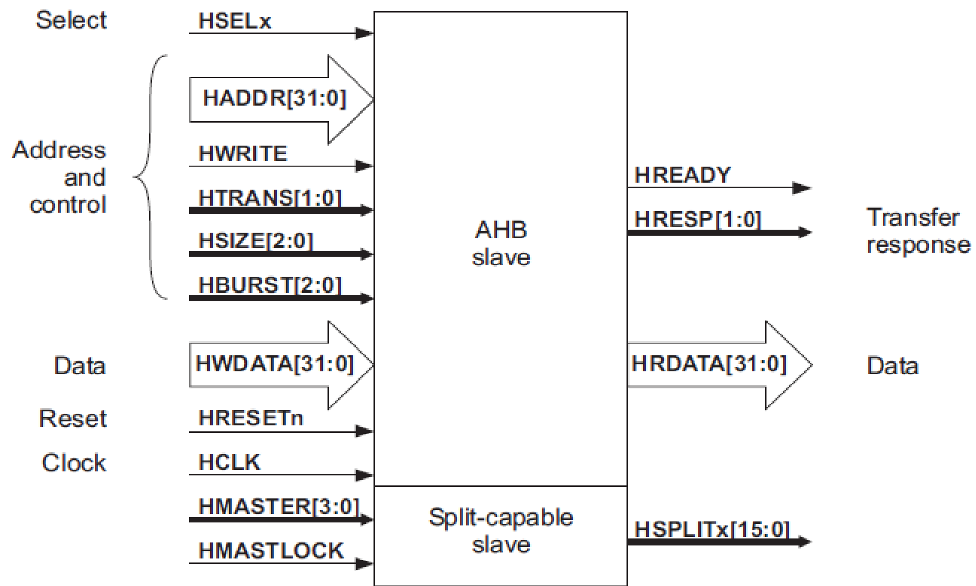

Figure 2.5 AHB slave interface ports

Description of signals used in slave AHB interface is given in Table 2.2.

Table 2.2 List of AHB signals

Signal Name	Description
HCLK	Defines system clock frequency of IP, all transactions are synchronous to positive edge of clock.
HRESETn	Defines reset for signal, it is an active low signal.
HADDR	Provides address for transactions.
HBURST[2:0]	Defines if transaction is single, increment or wrap type.
HPROT	Used to provide information about type of data, access level, buffering, and caching.
HSIZE	Defines size of a single transfer, supports maximum 1024 bits.
HTRANS[1:0]	Defines the present transaction state(idle, busy, non sequential, or sequential)
HWDATA	Provides data to be written
HWRITE	Indicates write operation if high else indicates read operation.
HRDATA	Provides read data from registers.
HREADY	Indicates completion of transfer if high.
HRESP[1:0]	Defines status of transfer(okay, error, retry, or split)
HSELx	Indicates transfer for the selected slave from multiple slaves.
HMASTLOCK	Defines the transaction is a locked sequence by the master
HMASTER[3:0]	Indicates which master (max. 16) is accessing the bus, generated by arbiter.
HSPLITx[15:0]	Indicates which master can retry a split transaction.

The basic AHB transaction consists of 2 phases as shown in figure 2.6, an address phase and a data phase. Both these phases are pipelined to increase the throughput of the system.

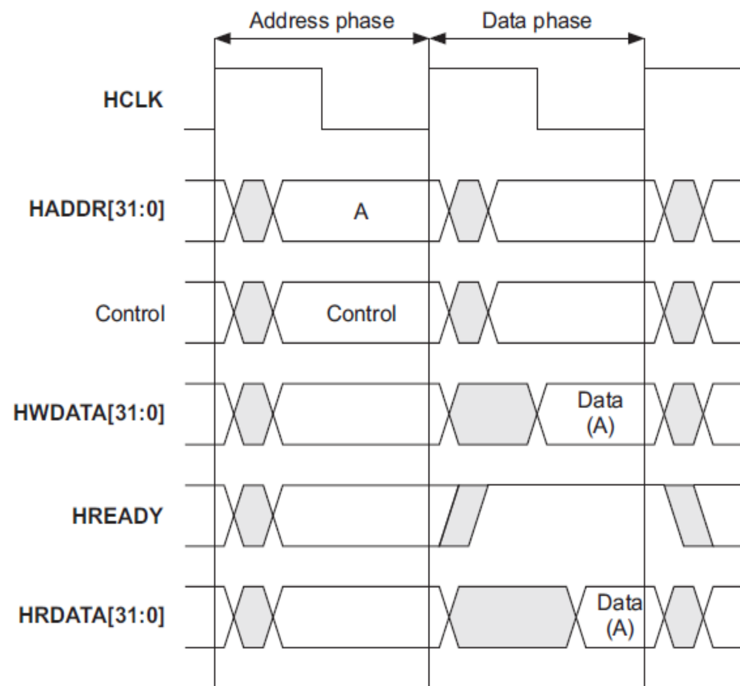

Figure 2.6 Single AHB transfer

However if a single transfer is extended due to additional wait states using HREADY signal, the next transfer's address phase is also stretched as shown in figure 2.7.

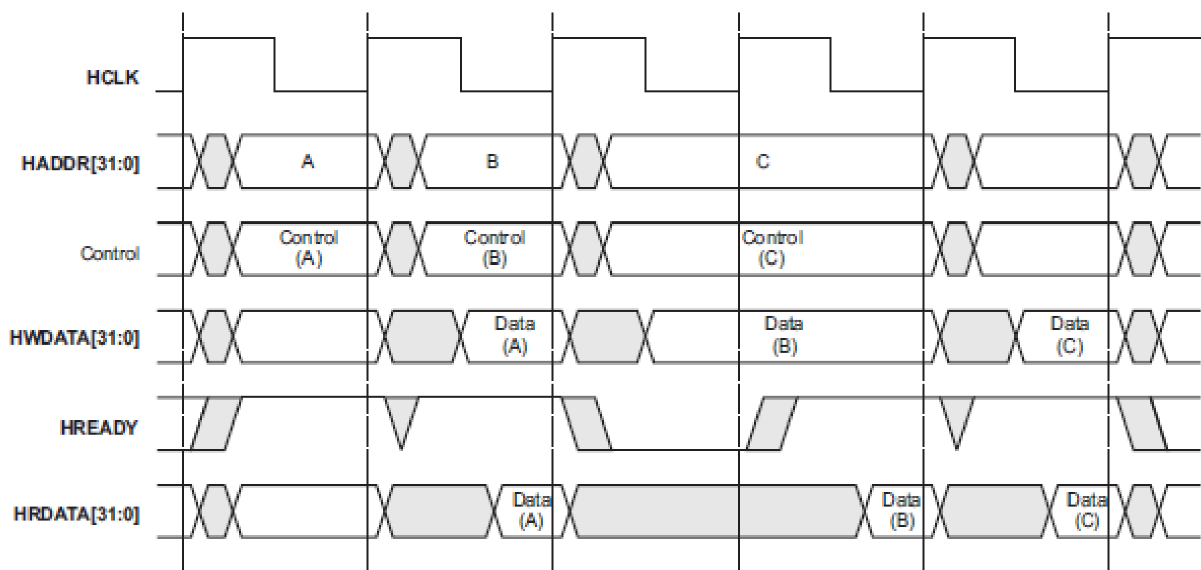

Figure 2.7 Multiple AHB transfers

If the transaction initiated is a first transfer or a single transfer, then HTRANS indicates non sequential access. If the transfer is of proceeding continuous address locations, HTRANS indicates sequential access. In a transaction, the attributes of a single transfer data are indicated by HBURST and HSIZE. In a burst transaction, there are two types of transfers, increment or wrap. If an increment burst is used, bursts shall not go beyond 1KB address boundary. For wrap transfers, if the starting address of the transfer is not aligned to the total number of bytes in the burst (size x beats) then the address of the transfers in the burst will wrap when the boundary is reached. The slave can give one of the four responses. Okay response in combination with HREADY is used to indicate if transfer is completed or slave can give either error, retry or split response in next cycle, thus taking minimum 2 cycles to generate these responses.

The basic difference between split and retry response is the way the arbiter allocates the bus after a split or a retry has occurred:

- For retry response, the arbiter will continue to use the normal priority scheme and therefore only masters having a higher priority will gain access to the bus.
- For a split response, the arbiter will adjust the priority scheme so that any other master requesting the bus will get access, even if it is a lower priority.

For a split transaction, the way it is handled is described below:

1. The master starts the transfer and issues address and control information.
2. If the slave is able to provide data immediately it may do so. If the slave decides that it may take a number of cycles to obtain the data it gives a SPLIT transfer response.
During every transfer the arbiter broadcasts a number, or tag, showing which master is using the bus. The slave must record this number, to use it to restart the transfer at a later time.
3. An arbiter grants other masters use of the bus and the action of the SPLIT response allows bus master handover to occur. If all other masters have also received a SPLIT response then the default master is granted.
4. When the slave is ready to complete the transfer it asserts the appropriate bit of the HSPLITx bus to the arbiter to indicate which master should be re granted access to the bus.
5. The arbiter observes the HSPLITx signals on every cycle, and when any bit of HSPLITx is asserted the arbiter restores the priority of the appropriate master.
6. Eventually the arbiter will grant access of bus to the master so it can re-attempt the transfer. This may not occur immediately if a higher priority master is using the bus.
7. When the transfer eventually takes place the slave finishes with an OKAY transfer response.

2.2.3 AMBA ATB PROTOCOL

Advanced Trace Bus [22] is used to stream data related to the instruction and data in real time. This gives support for software debugging without halting a system. It is a general bus protocol used to stream non formatted data. A component having support for tracing such as coresight system has this interface. The master streams the data on the ATB interface, while slave reads this data from this interface. The basic components present in a system having trace support is shown in figure 2.8.

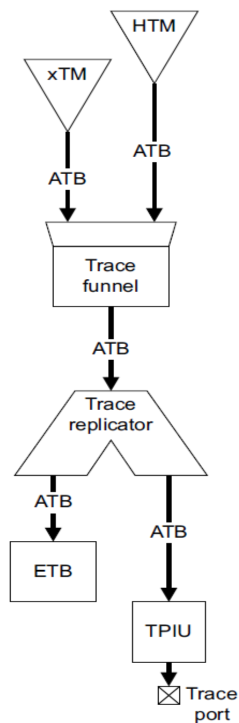

Figure 2.8 ATB components

1) ETB: Embedded Trace Buffer refers to a memory storage component.
2) xTM: Trace Macrocell is a component that produces trace data from a processor. It consists of either two components:
    a. ETM, Embedded Trace Macrocell produces traces for both instruction as well as data.
    b. PTM, Program Flow Trace Macrocell only generates instruction trace.
3) HTM: AHB Trace Macrocell produces trace information regarding the AHB interconnect.
4) TPIU: Trace port interface unit provides connection to extraneous storage device.
5) Trace replicator: It produces copies of trace data.
6) Trace funnel: It merges trace data and streams that data on a single

The trace sources generate ATB data that can be re transmitted by trace links to the end component receiving trace data.

Description of ATB interface is given in Table 2.3

Table 2.3 List of ATB signals

Signal Name	Description
ATCLK	Gives clock frequency of tracing components.
ATRESETn	Gives reset for tracing components, asynchronous active low .
ATBYTES [m:0]	Defines valid bytes of ATDATA
ATDATA [n:0]	Gives trace data
ATID [6:0]	Identifies generator of trace data
ATREADY	Controls completion of ATB transfer.
ATVALID	Controls start of ATB transfer, if low all other signals related to data are invalid.
AFVALID	Gives command to the buffers to flush data.
AFREADY	Gives feedback that flush has been completed.

For 7 bits data, ATBYTES signal is not required. ATID can be any value between 0x01-0x6F, for generating a trace trigger, ATID 0x7D is used. ATID values 0x00, 0x70-0x7C, 0x7E, and 0x7F are refrained from using as they are kept for use in coresight systems.

This protocol uses handshaking mechanism with ATVALID and ATREADY signals. It is recommended that ATB slave responds within a single clock cycle. If ATB slave cannot meet this requirement, then it recommends using appropriate internal buffering, which must buffer data for one or more cycles. A basic transaction is shown in figure 2.9

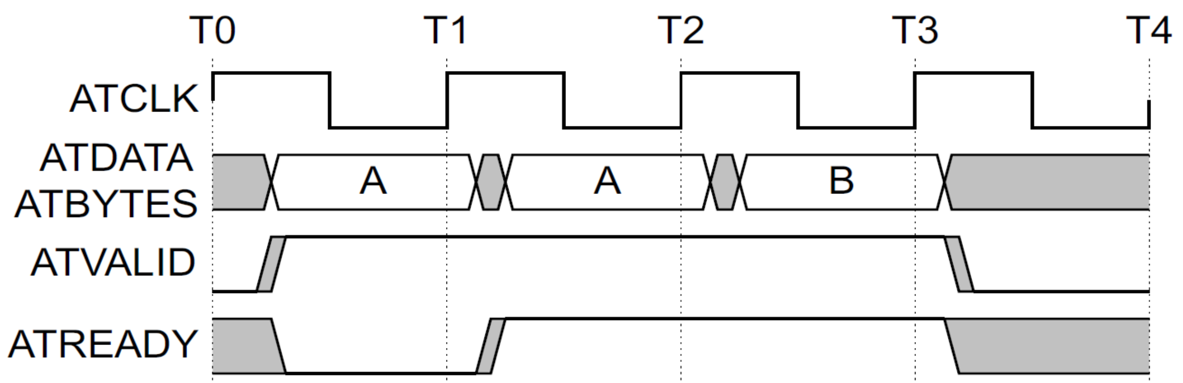

Figure 2.9 Basic ATB transaction

At T0, an ATB transaction is initiated by driving ATVALID high. But since ATREAY is driven low, the transaction is stalled, and transaction is stretched to next clock cycle. At T2, first ATDATA transfer is

completed followed by next ATDATA transfer completion at T3. At T4, transfer is ignored as ATVALID is driven low.

2.3 IP INTERFACE

The Intellectual Property interface (IPI) [23] is an open standard protocol released in 1999. It was released with an aim to provide a uniform I/O interface across multiple components, so that when these components are used in SoCs having different architecture then no RTL changes are required to be done. This allows a SoC released having less TTM The IP interface targets all types of I/Os in a component. The I/O ports are divided into different interfaces according to their functionality. The IP interface provides 8 interfaces created according to their functionality and uses color naming convention as listed in table 2.4.

Table 2.4 List of IP Interfaces

Bus Name	Associated block
DarkBlue line	Contains signals related to DMA interface
ForestGreen line	Contains signals related to FIFO interface
Green line	Contains signals that are global i.e. routed to all the blocks in a system.
Indigo line	Contains signals related to interrupts.
Magenta line	Contains signals related to bus masters.
Purple line	Contains signals related to pad interface
SkyBlue line	Contains signals related to slave bus of an IP
Tan line	Contains signals related to test interface.

The skyblue line provides a standard interface for accessing registers on an IP module. Apart from skyblue line, green line signals group I/Os that are routed to every VC or IP i.e. signals like clocks, resets, etc. An IP can use any combination of above colored line bus interfaces according to the functionality. For accessing IP registers, according to functionality signals are grouped as listed below:

a) Clock signals
b) Data signals
c) Protocol signals

a) Clockiing Signals:

A few IPs can use a different clock signal (ipg_clk_s), different from the system clock signal (ipg_clk), in order for operation of the registers. If this is the case, then ipg_clk_s should be in synchronization of ipg_clk, having target of zero percent skew between the two clocks. If separate ipg_clk_s signal is not

present in the system, then ipg_clk is used to drive it. As shown in figure 2.10, in case module is not enabled, then ipg_clk_s can be either to ipg_clk or remain unchanged.

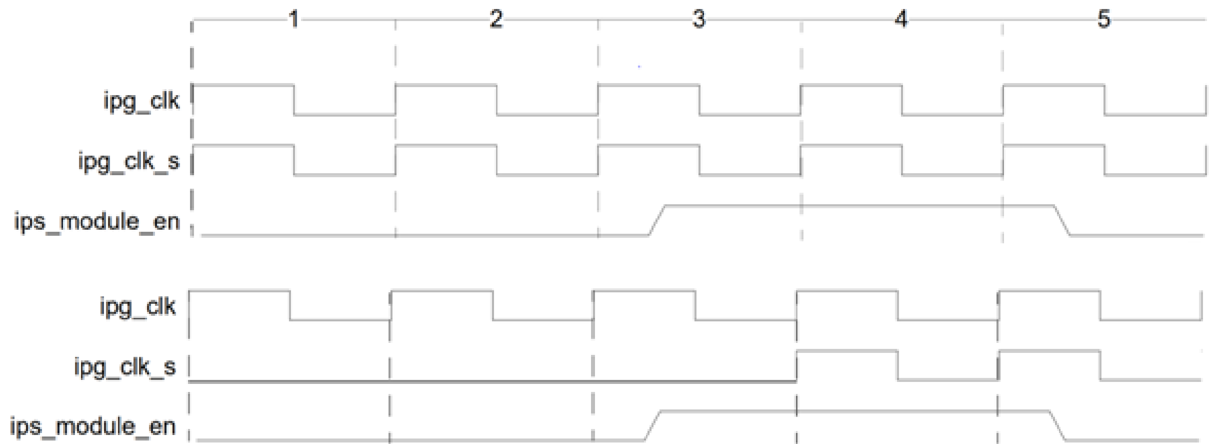

Figure 2.10 Clocking strategies of IPS interface

b) Data signals:

Write/Read bus, ips_wdata/ips_rdata bus is used to transfer data into/from the registers of the IP, having width in the power of 2 and multiple of 8 (8,16,32,64). The bus signals are considered to be active if operation being performed on registers is write/read while module enable is asserted, including all the wait states. Also the transfer shouldn't be indicated as erroneous.

c) Protocol signals:

Address bus of maximum 64 bit is supported by sky blue line interface. However, for decoding of registers in an IP, certain lower bits of address bus should not be used inside the component based on the width of data bus as shown in table 2.5.

Table 2.5 IPS Address Bus Range

Data bus width	Address Bus used
8	Ips_addr[63:0]
16	Ips_addr[63:1]
32	Ips_addr[63:2]
64	Ips_addr[63:3]

The IPS interface uses module enable along with wait signal for completion of a trasaction. If module enable is asserted and wait signal is de-asserted, then the traction is completed in same single cycle.

However, if wait signal is asserted, then the transaction is stretched until wait signal is de-asserted. Address data, read/write signal, byte enable, write data should be held unchanged till the transaction is completed. The read data and transfer error signal are required to be valid during the last cycle of transaction.

Figure 2.11 shows read/write transactions using IPI sky blue line, where all signals changes with respect to rising edge of clock.

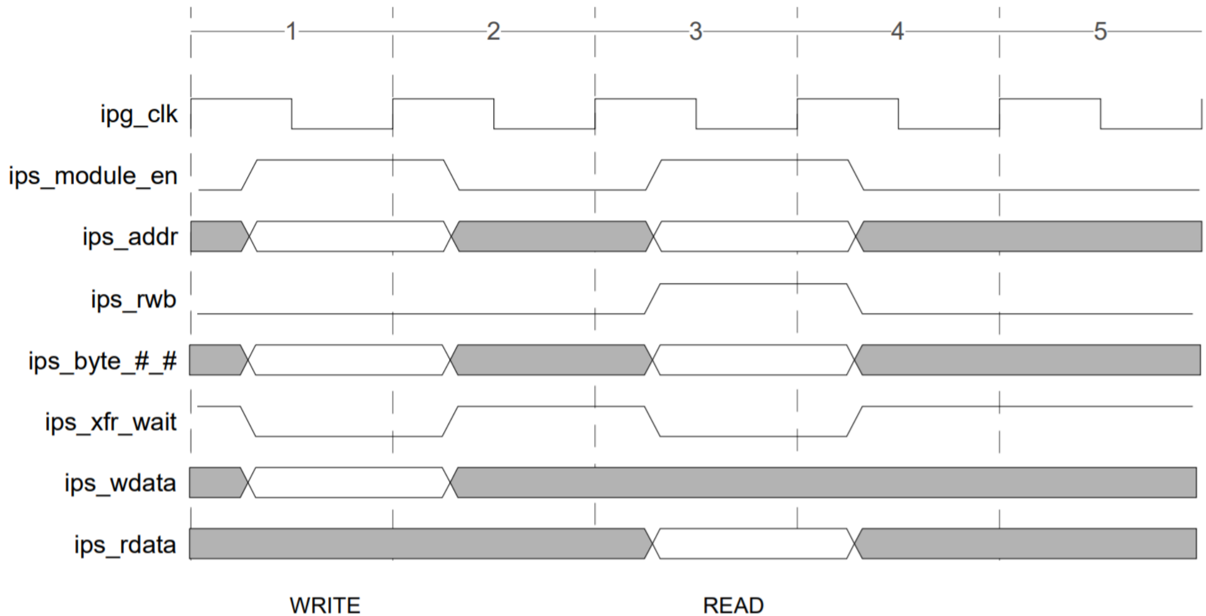

Figure 2.11 IPI skyblue line transaction

The single clock cycle between the two transactions is not required. This interface also supports back to back transactions also.

CHAPTER 3

3. SCRIPTING

This chapter describes how testbench setup can be automated for an IP in order to verify the functionality. Moreover using IP-XACT, details related to registers can be retrieved using a script as described in the subsection below.

3.1 TESTBENCH SETUP

For the automated creation of a basic testbench, the module which is to be verified has to be written using system verilog wherein the ports are to be declared in ASNSI-C style. In ANSI-C syntax, direction of port, data type of port, width of port, all is declared one time during module port declaration. This file having ports mentioned in ANSI-C style is used as DUT top.

With the help of this created testbench, a design engineer can easily verify the basic functionality of its design in very less time as this script generates a testbench which be easily modified to create certain test cases. The testbench generated verifies the memory map of the DUT.

Python is used as language for scripting to parse the DUT top file. The script parses the system verilog file, and first of all removes all the comments in it. Then this file is parsed for the extraction all the top level parameters defined during module declaration, also all the local parameters are extracted. Due the presence of these parameters in the testbench, the testbench becomes generic and can be used for different combinations of parameters.

In the next step, input and output ports are extracted along with their port widths. The count of input and output ports is provided on the console. If port are parameterized, the port parameters are retained within the testbench also. These ports are then declared in testbench for the connection with DUT. After declaring the ports, all the input ports are initialized to a zero value.

The testbench now instantiates the DUT top module. The path of this top module is to be provided by the user at the time of script invoke. The instantiated DUT component is passed with all the parameters along with the ports connection from the testbench to DUT. This allows overriding of parameters in DUT with testbench. However, the connection of DUT register's bus interface with testbench has to be done manually.

Since all the digital IP modules use a bus interface to do transactions on the registers, models of these buses are created first using system verilog language. The models are created for the following bus interfaces:

a) IPI bus interface, skyblue line (master)
b) APB interface (master)
c) AHB interface (master)
d) ATB interface (slave)

The testbench generates a clock signal whose period can be controlled using a `define, thus modulation of DUT operating frequency can be done easily. After this, reset is de-asserted. The duration during which DUT module remains in reset is also controlled using a `define.

After lifting of reset, the testbench does basic transactions on registers. The testbench is designed to do transactions on 16KB space. Since the testbench uses `define to declare location of last address, user can easily modify the value of last address space.

For verification, the cadence irun tool is used, which includes incisive simulator.

The script generates following files:

a) tb.sv: This is the testbench for DUT
b) file_list.txt: This file contains the path of all the modules used inside the DUT.
c) input.tcl: This file contains commands that can be passed to simulator after compilation.
d) run_cmd: This is the single makefile which user can use for simulation, contains switches for include directory, setting top module, coverage collection, invoking tool in GUI, etc.

After simulation is completed, GUI interface is invoked to view the waveforms and design. Also tool dumps database for simvision and Incisive Comprehensive Coverage (ICC) tool so later on the reports can be reviewed.

Using Integrated Metrics Center (IMC) tool, reports can be analyzed in batch mode and a consolidated report of functionality coverage can be obtained.

1) Merge all the testcases coverage report by providing a file containing path of all coverage reports:

   imc> merge –runfile cov_merge_run_file –out merged_cov_out –metrics all

2) Load the merged coverage report:

   imc> load –run cov_work/scope/merged_cov_out

3) Exporting report

   imc> report –summary –out detail_rpt_name.rpt –inst –metrics overall

3.2 IPXACT MEMORY MAP

For the individual slave IP, a single memory map is defined. This memory map is referenced with the help of bus interface. In IPXACT xml file, memory maps tag encapsulates multiple memory map tagged elements. The memory map contains following elements, which are described below:

a) Name group elements: This includes name, display name and description of the memory map.
b) Memory map group: This includes any of the address block, bank, subspace map.
c) Memory remap: In a certain remapped state, it describes additional address block, bank.
d) Address unit bits: This element describes number of bits in a single address location.

Address block is used to describe a block of memory. Further inside the address block element, following elements are present:

a) Name group elements: Name of the different address blocks should be unique inside a single memory map.
b) Base address: It is the starting address of the block inside the memory map. It is a configurable element, i.e. the value of this element can be parameterized.
c) Address block definition group: Stores the data related to the block.

Address block definition group contains following elements:

a) Type identifier: This indicates the same information is captured by multiple address block's address definition group.
b) Block size group elements: It contains two elements, both configurable. Range shows the no of addressable locations, expressed in no of address unit bits. Size defines the maximum width of the register which can be present in this address block.
c) Memory block data group: It contains four elements which specify the functionality of address block.
d) Register data group: It contains data related to registers and register files. The name of register and register file should be unique in the containing address block.

Memory block data group contains following elements:

a) Usage: Its value can be memory, register or reserved. If usage is memory then access type defines whether it is used as RAM, ROM or write only memory.
b) Volatility: Can be either true or false. It indicates if hardware can change the stored value. If this element isn't present then, it is assumed to be false for a field and unspecified for register, address blocks, bank.

c) Access: It can be RW, RO, WO, RW1, and W1. If this element isn't present, then it's value is presumed to be read write.
d) Parameters: They describe the parameters for the block.

Register element contains data related to registers within the following elements:

a) Name group elements: Name of different registers should be unique.
b) Dimension: Used to capture length of multi dimension array.
c) Address offset: Used to capture information for starting address offset to the address block or register file.
d) Register definition group: Captures data related to the register.
e) Parameters: They describe the parameters for the register.

Register definition group contains following elements:

a) Type identifier: It indicates same data is captured for multiple register's register definition group.
b) Size: It shows the size of the register, this element is configurable.
c) Volatility: It indicates if register value can be changed by the design. If this element is absent for register, no assumption is made for its value.
d) Access: It can be either RW, RO, WO, RW1, W1. If not defined, it's value is taken from the address block.
e) Reset: Stores the information related to reset of the register. This element further encapsulates two elements, value and mask.
f) Field: Stores information of bit fields present in the register.

All the field elements inside register contains following elements:

a) Name group elements: Contains name, display name and description of the field. The name of all fields must be unique in a register.
b) Bit offset: Captures information related to offset of field in a register.
c) Field definition group: Contains elements like type identifier, bit width (configurable) and field data group elements.
d) Parameters: Describe parameters for the field.

Field data group element captures additional properties of the field.

a) Volatility: It indicates if value of field can be altered by the design in run time. If this element is absent, then its value is assumed to be false.

b) Access: It can be RW, RO, WO, RW1, or W1. If this element is not present, then its value is taken from the register.

c) Enumerated values: It captures names for values in the field.

d) Modified write value: It describes how field value is affected on performing a write. It can be W1C, W1S, W1T, W0C, W0S, W0T, WS, WC or modify. If this element isn't present, then value written is stored as it is in the field.

e) Write value constraint: Captures the possible values that the field can have. If this element isn't present, then field can take any value in run time.

f) Read action: It describes how field value is affected on performing a read. It can be clear, set or modify. If this element isn't present, then no action is taken after read operation.

g) Testable: Describes if a field can be tested using simple registers test.

Below is the structure of the reference model for registers generated by parsing register xml using python scripting (using elementary tree library) :

1) In a file named RegisterMap, address blocks present in memory map are listed:

//////////////////////////// RegisterMap.txt /////////////////////////

```
register_prop [2-1:0] ControlSpace_register;
register_prop [2-1:0] MyBLOCK_register;
```

2) For each address block, a corresponding file is generated defining each register attributes in system verilog as shown below, only one register is shown here:

///////////////////////////// ControlSpace.txt ///////////////////////////

```
ControlSpace_register[0]='{access:read-
write,HARD_reset_value:32'b00000000000000000000000000000000,HARD_reset_mask:32'b00
000000000000000000000000001111,w_mask:32'b00000000000000000000000000000000,r_mas
k:32'b11111111111111111111111111111111,w1c_mask:32'b0000000000000000000000000000
0010,w1s_mask:32'b00000000000000000000000000000000,w1t_mask:32'b00000000000000000
000000000000000,w0c_mask:32'b00000000000000000000000000000000,w0s_mask:32'b00000
000000000000000000000000000,w0t_mask:32'b00000000000000000000000000000000,wc_ma
sk:32'b00000000000000000000000000000000,ws_mask:32'b0000000000000000000000000000
0000,wm_mask:32'b00000000000000000000000000000000,rc_mask:32'b0000000000000000000
0000000000000,rs_mask:32'b00000000000000000000000000000000,rm_mask:32'b000000000
00000000000000000000000};
```

Using above database created in system verilog, verification of registers can be automated.

CHAPTER 4

4. SIMULATION RESULTS

4.1 BUS TRANSACTIONS

The IP were simulated using the BFM designed and the waveforms for various protocols can be viewed in the following sections.

4.1.1 IPS INTERFACE

The contents of every address locations are read to verify value stored is correct. Figure 4.1 shows continuous read all registers in an address block. Herein, it is also checked if IPS transfer error signal is asserted on reading data from registers not present at that address.

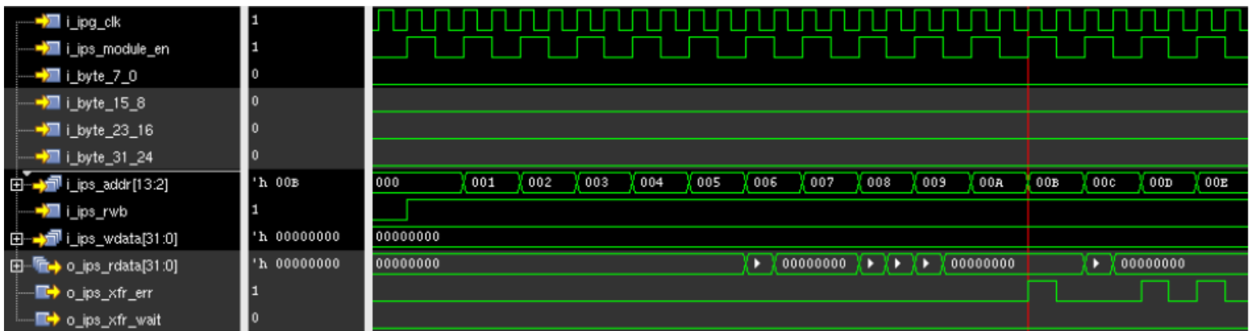

Figure 4.1 IPS read transactions

For each address location, data is written to check that data is not written at masked bits. Also the registers defined with software write access are only writable. Thus writing to these registers must give transfer error.

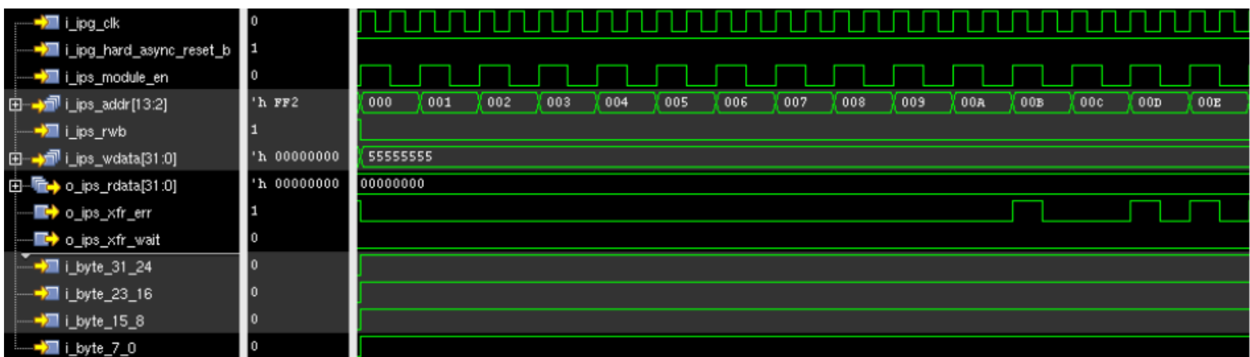

Figure 4.2 IPS write transactions

It is possible that doing some transaction on one register may affect the content of another register. To avoid such scenario, a read transaction should be followed by a write transaction on same register as shown in figure 4.3

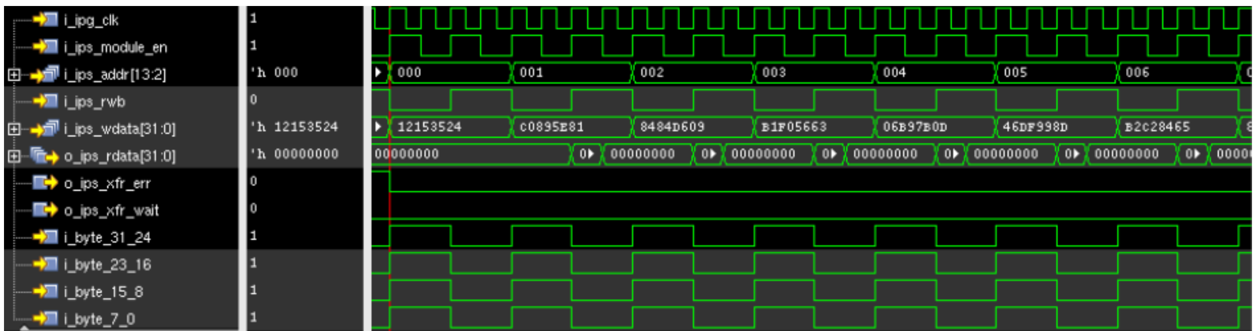

Figure 4.3 IPS write-read transactions

4.1.2 APB INTERFACE

Figure 4.4 shows APB back to back read transactions performed continuously on the registers. The reserved registers return slave error signal.

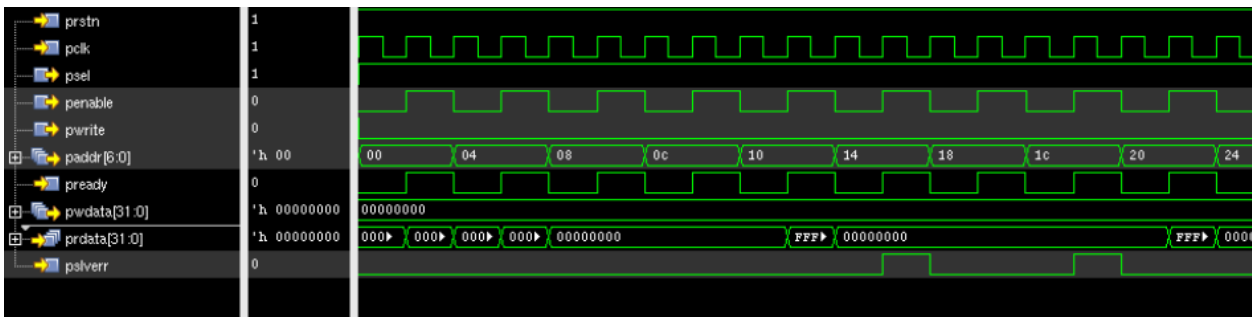

Figure 4.4 APB back to back read transactions

Figure 4.5 shows APB back to back write transactions performed continuously on the registers. The reserved registers return slave error signal. Also the register having read only access also returns slave error signal.

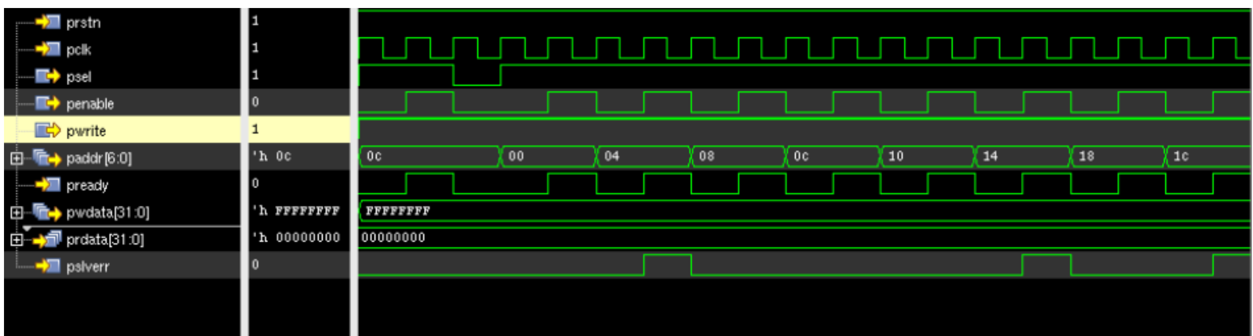

Figure 4.5 APB back to back write transactions

Figure 4.6 shows APB back to back write and read transactions performed continuously on the registers.

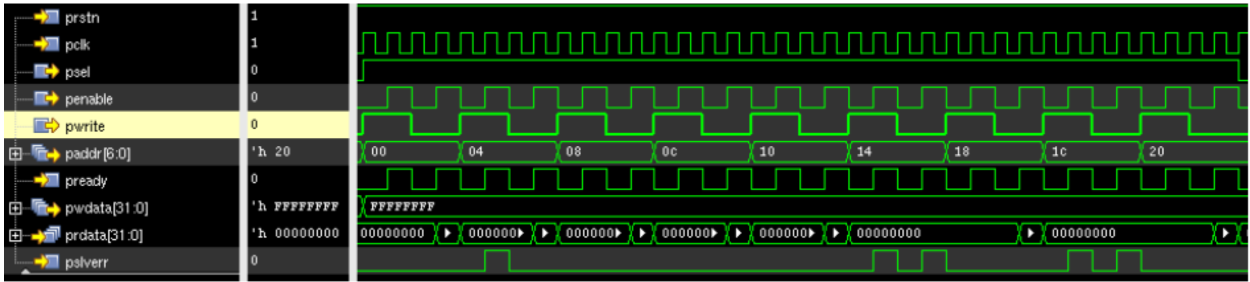

Figure 4.6 APB back to back write–read Transactions

4.1.3 ATB INTERFACE

To generate transactions for trace on ATB interface, data has to be provided by processor. In testbench, packets are generated using classes and the constrained random sequence is created on the DUV as shown in figure 4.7.

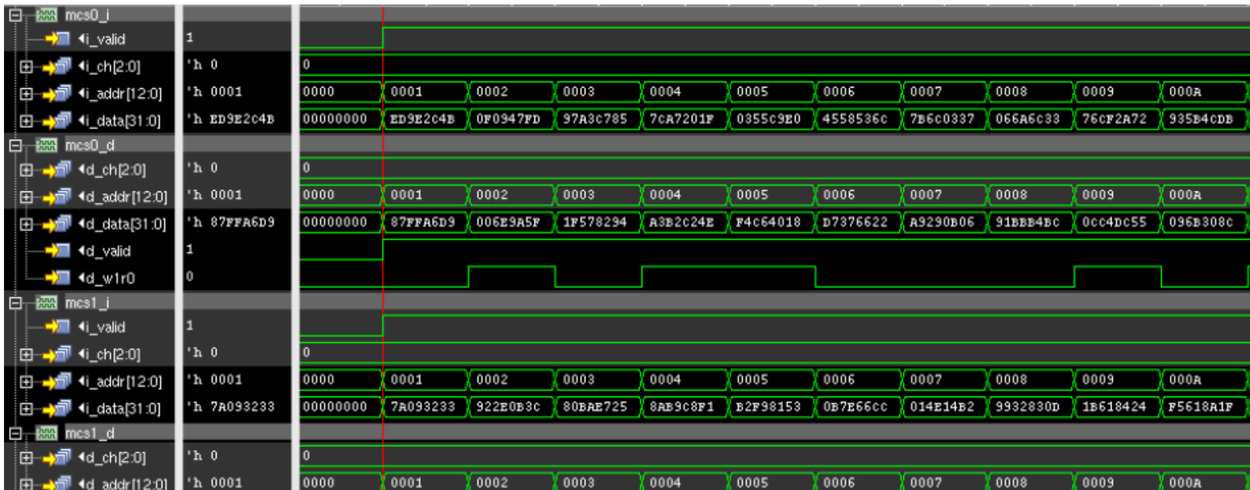

Figure 4.7 Constrained random sequence generations

The ATB trace data transactions are shown in figure 4.8.

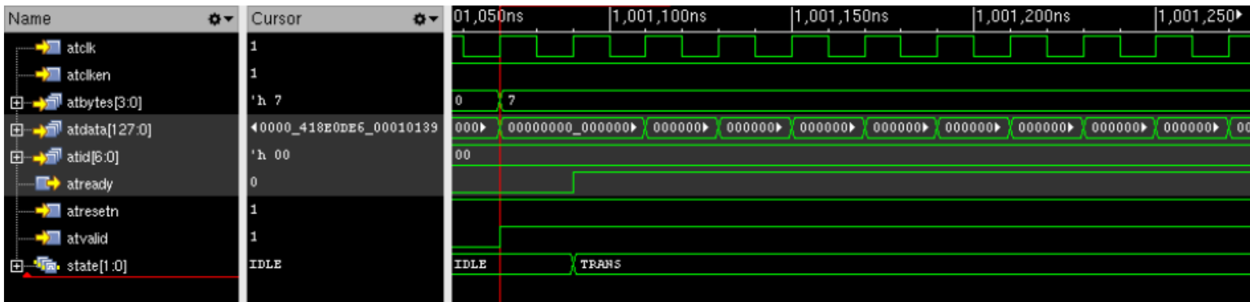

Figure 4.8 ATB data transaction

4.2 TESTBENCH SETUP

Figure 4.9 shows the console invoked for creation of testbench to verify the registers implemented in DUV. Only bus interface needs to be selected in order to set up environment as shown in figure 4.9.

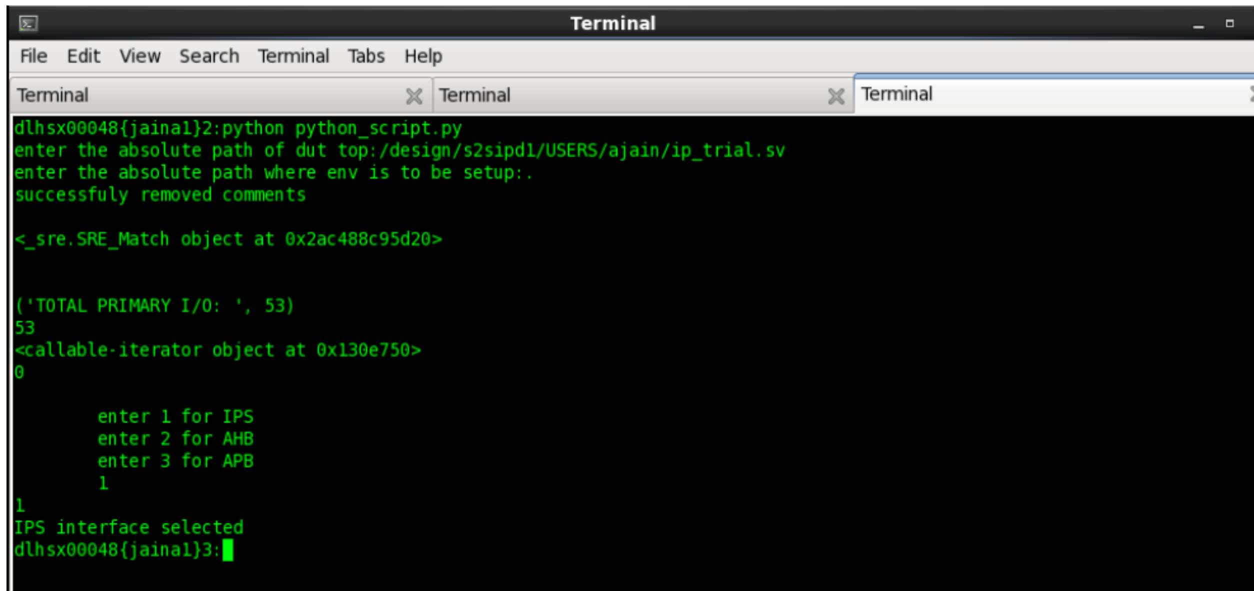

Figure 4.9 Input console for testbench setup

After the testbench is setup, registers implementation is verified. Additional test cases can be written to verify the functionality of DUV. At last, the coverage of all tests can be merged as shown in figure 4.10.

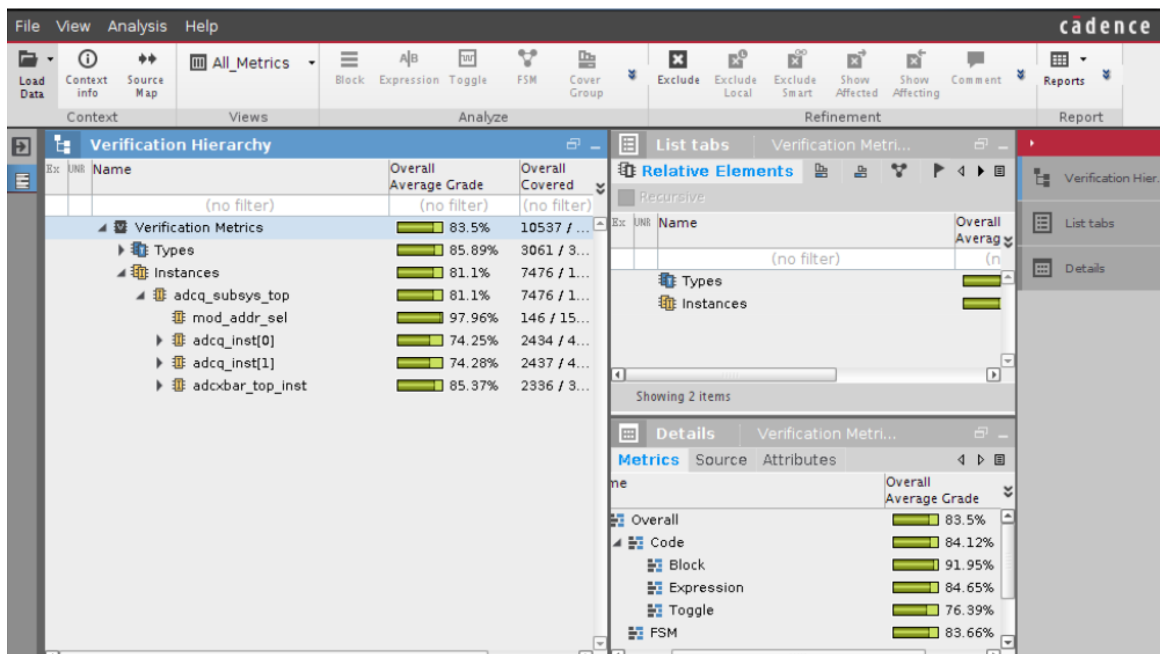

Figure 4.10 Coverage Report

CHAPTER 5

5. CONCLUSION AND FUTURE SCOPE

5.1 CONCLUSION

To verify an IP before using it inside an SoC requires verification. A script is made to setup the testbench, which performs basic transactions on IP registers utilizing the standard bus interfaces. An additional script parses the IPXACT register data. Bus functional model for various on chip bus interface has been studied and implemented as discussed. The testbench created gives designer easy way to perform basic tests on IP easily without learning UVM as testbench gives designer more of software like testing option, which also helps in making changes to the design as designer may encounter some problems which will be faced by the application developer. Also for all the test cases created, a merged coverage report is also generated to track the verification goal.

5.2 FUTURE SCOPE

As of now, ports are extracted from system verilog design file, but as IP-XACT standard adoption is increasing, the connection of testbench with the DUV can be automated. Since IP-XACT also captures the information related to bus interfaces used in an IP, the testbench can be setup for DUV modeled in any language. Moreover using the IP-XACT latest standard, testbench created can be parametrized as it captures the dependencies using system verilog syntax, not XPATH expressions.

CHAPTER 6

6. REFERENCES

[1] Ihanajärvi Miika. *Universal Verification Method And Environment For Risc Processor*. Tampere University Of Technology, Tampere, Finland, May 2016.

[2] IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) : IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language. IEEE, 2018.

[3] Stuart Sutherland and Don Mills. Synthesizing SystemVerilog Busting the Myth that SystemVerilog is only for Verification. *Synopsys User Guide (SNUG).* Silicon Valley, 2013, pp. 1–45.

[4] IEEE Std 1800.2-2017 : IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE, 2017.

[5] Stuart Sutherland and Tom Fitzpatrick. UVM-Light: A Subset of UVM for Rapid Adoption. Design and Verification Conference and exhibition. Europe, 2015.

[6] Tejas Pravin Warke. Verification of SD / MMC Controller IP Using UVM. Rochester Institute of Technology, Rochester, New York, May 2018.

[7] Martti Tiikkainen. *Automated Functional Coverage Driven Verification With Universal Verification Methodology*. University of Oulu, Oulu, Finland March. 2017.

[8] A. B. Mehta, *SystemVerilog Assertions and Functional Coverage*. New York: Springer 2014.

[9] IEEE Std 1685-2009 : IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. IEEE, 2010.

[10] Kwanghyun Cho *et al.* Reusable Platform Design Methodology For SoC Integration And Verification. *International SoC Design Conference*. South Korea: Busan, 2008, pp.78-81.

[11] Hung Yi Yang. Highly automated and efficient simulation environment with UVM. *VLSI Design Automation and Test.* Taiwan: Hsinchu, 2014.

[12] Amr Hany *et al.* Approach for a unified functional verification flow. *Saudi International Electronics Communication and Photonics Conference*. Greece: Fira, 2013, pp. 1-6.

[13] Gaurav Sharma, Lava Bhargava and Vinod Kumar. Automated Coverage Register Access Technology on UVM Framework for Advanced Verification. *International Symposium on Circuits and Systems*. Italy: Florence, 2018, pp. 1-4.

[14] Namdo Kim *et al.* How to automate millions lines of top-level UVM testbench and handle huge register classes. *International SoC Design Conference*. South Korea: Jeju Island, 2012, pp. 405-407.

[15] Anurag Shrivastava, G.S. Tomar and Ashutosh Singh. Performance Comparison of AMBA Bus-Based System On-chip Communication Protocol. *International Conference on Communication Systems and Network Technologies.* India: Jammu, 2011.

[16] Dongwan Shin, Samar Abdi and Daniel D. Gajski. Automatic Generation of Bus Functional Models from Transaction level Models. *Asia and South Pacific Design Automation Conference*. Japan: Yohohama, 2004.

[17] V. Berman (2006). Standards: The p1685 ip-xact ip metadata standard, *IEEE Design Test of Computers*, 23(4), 316 –317.

[18] Mark Litterick and Marcus Harnisch. Advanced UVM Register Modeling - There's More Than One Way to Skin A Reg. *Design and Verification Conference and exhibition.* CA: San Jose, 2014. pp. 1-12.

[19] H. Saafan, M. Watheq and A. Salem. SoC Connectivity Specification Extraction using Incomplete RTL Design: An Approach for Formal Connectivity Verification. *International Design & Test Symposium.* Tunisia: Hammamet, 2016, pp. 110-114.

[20] ARM. AMBA APB Protocol Specification. Available at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024c/index.html/ (Accessed on 11th September 2018).

[21] ARM. AMBA Specification Rev 2.0 https://developer.arm.com/docs/ihi0011/latest/amba-specification-rev-20 Accessed on 5th November 2018)

[22] ARM. AMBA™ 4 ATB Protocol Specification. Available at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024c/index.html/ (Accessed on 19th February 2019).

[23] Freescale. IP Interface. Available at http://read.pudn.com/downloads223/doc/project/1047813/IP_interface.pdf (Accessed on 9th July 2018).

CHAPTER 7

7. APPENDIX

7.1 IPS Driver BFM

```
`define ADDR_WD 8

`define WDATA_WD 32

`define RDATA_WD 32

`define BYTE_NUM 4  //number of bytes in wdata

module ips_driver(

        input ipg_clk,

        input ipg_clk_s,

        input ipg_hard_async_reset_b,

        input ipg_hard_sync_reset_b,

        input [`RDATA_WD-1:0]ips_rdata,

        output logic [`WDATA_WD-1:0] ips_wdata,

        output logic [`ADDR_WD-1:0]ips_addr,

        output logic ips_module_en,

        output logic [`BYTE_NUM-1:0]ips_byte_en,

        output logic ips_rwb,

        input ips_xfr_wait,

        input ips_xfr_err

);

logic access_ongoing=1'b0;


task wr_ips (

        logic [`ADDR_WD-1:0] addr,          //address of register

        logic [`WDATA_WD-1:0] data,         //data to be written

        logic [`BYTE_NUM-1:0] strb = 'hF    //byte select
```

```
);
            wait(access_ongoing == 'b0);
            @(posedge ipg_clk);
            access_ongoing = 1'b1;
            #10ps;
            ips_module_en = 1'b1;
            ips_rwb = 1'b0;//0;
            ips_addr = addr;
            ips_byte_en = strb;
            ips_wdata = data;

            @(posedge ipg_clk);
            wait(ips_xfr_wait == 'b0);
            #10ps;
            ips_module_en = 1'b0;
            access_ongoing = 1'b0;
            ips_rwb = 1'b0;
  endtask

  task rd_ips (
            logic [`ADDR_WD-1:0] addr,
            output logic [`RDATA_WD-1:0] data
);
            wait(access_ongoing == 'b0);
            @(posedge ipg_clk);
            access_ongoing = 1'b1;
            #10ps;
            ips_module_en = 1'b1;
```

```
            ips_addr = addr;
            ips_byte_en = 'h0;
            ips_rwb = 1'b1;//1;

            @(posedge ipg_clk);
            wait(ips_xfr_wait == 'b0);
            data = ips_rdata;
            #10ps;
            ips_module_en = 1'b0;
            access_ongoing = 1'b0;
  endtask
endmodule
```

7.2 APB Driver BFM

```
`define ADDR_WD 7
`define WDATA_WD 32
`define RDATA_WD 32
module apb4_driver (
            input pclk,
            input prstn,
            input pready,
            input pslverr,
            input [`RDATA_WD-1:0]prdata,
            output logic psel,
            output logic [`ADDR_WD-1:0]paddr,
            output logic [`WDATA_WD-1:0]pwdata,
            output logic [3:0]pstrb,
```

```
        output logic [2:0]pprot,
        output logic pwrite,
        output logic penable
);
logic access_ongoing=1'b0;
//********************************* contiuous writes *************************************
task wr_apb4 (
        logic [`ADDR_WD-1:0] addr,
        logic [`WDATA_WD-1:0] data,
        logic [31:0] num_txn = 'b1,
        logic [3:0] strb = 4'hF,
        logic [2:0] prot = 3'b010
);
        wait(access_ongoing=='b0);
        @(posedge pclk);
        access_ongoing='b1;
        for(int i=0;i<num_txn;i++) begin
                #10ps;
                psel='b1;
                pwrite=1'b1;
                penable=1'b0;
                paddr=addr+i*4;
                pwdata=data;
                pstrb=strb;
                pprot=prot;

                @(posedge pclk);
                #10ps;
```

```
			penable=1'b1;
			wait(pready==1'b1);
			@(posedge pclk && pready);
		end
		#10ps;
		penable=1'b0;
		psel=1'b0;
		access_ongoing=1'b0;
endtask

//****************************** contiuous reads **********************************
task rd_apb4 (
		logic [`ADDR_WD-1:0] addr,
		output logic [`RDATA_WD-1:0] data,
		input logic [31:0] num_txn = 'b1,
		logic [2:0] prot=3'b010
);
		wait(access_ongoing=='b0);
		@(posedge pclk);
		access_ongoing='b1;
		for(int i=0;i<num_txn;i++) begin
			#10ps;
			psel=1'b1;
			pwrite=1'b0;
			penable=1'b0;
			paddr=addr+ i*4;
			pstrb=4'h0;
			pprot=prot;
```

```
                @(posedge pclk);
                #10ps;
                penable=1'b1;
                wait(pready==1'b1);
                @(posedge pclk && pready);
                data=prdata;
        end
        #10ps;
        penable=1'b0;
        psel=1'b0;
        access_ongoing=1'b0;
endtask
//************************************ write_b2b ***********************************

task wr_apb4_b2b (
        logic [`ADDR_WD-1:0] addr,
        logic [`WDATA_WD-1:0] data,
        logic [31:0] num_txn = 'b1,
        logic [3:0] strb = 4'hF,
        logic [2:0] prot = 3'b010
);
        wait(access_ongoing=='b0);
        @(posedge pclk);
        access_ongoing='b1;
        for(int i=0;i<num_txn;i++,data=data+`WDATA_WD'h5555_5555) begin
                #10ps;
                psel='b1;
```

```
                pwrite=1'b1;

                penable=1'b0;

                paddr=addr;

                pwdata=data;

                pstrb=strb;

                pprot=prot;


                @(posedge pclk);

                #10ps;

                penable=1'b1;

                wait(pready==1'b1);

                @(posedge pclk && pready);

        end


        #10ps;

        penable=1'b0;

        psel=1'b0;

        access_ongoing=1'b0;

endtask

//************************************ read_b2b ************************************

task rd_apb4_b2b (

        logic [`ADDR_WD-1:0] addr,

        output logic [`RDATA_WD-1:0] data,

        input logic [31:0] num_txn = 'b1,

        logic [2:0] prot=3'b010

);

        wait(access_ongoing=='b0);

        @(posedge pclk);
```

```
        access_ongoing='b1;
        for(int i=0;i<num_txn;i++) begin
                #10ps;
                psel=1'b1;
                pwrite=1'b0;
                penable=1'b0;
                paddr=addr;
                pstrb=4'h0;
                pprot=prot;

                @(posedge pclk);
                #10ps;
                penable=1'b1;
                wait(pready==1'b1);
                @(posedge pclk && pready);
                data=prdata;
        end

        #10ps;
        penable=1'b0;
        psel=1'b0;
        access_ongoing=1'b0;
endtask
//********************************** write_read_b2b **********************************
task wr_rd_apb4 (
        logic [`ADDR_WD-1:0] addr,
        logic [`WDATA_WD-1:0] data,
        output logic [`RDATA_WD-1:0] r_data,
```

```
        input logic [31:0] num_txn = 'b1,
        logic [3:0] strb = 4'hF,
        logic [2:0] prot = 3'b010
);
        wait(access_ongoing=='b0);
        @(posedge pclk);
        access_ongoing='b1;
        for(int i=0;i<num_txn;i++) begin
                #10ps;
                psel='b1;
                pwrite=1'b1;
                penable=1'b0;
                paddr=addr+i*4;
                pwdata=data;
                pstrb=strb;
                pprot=prot;

                @(posedge pclk);
                #10ps;
                penable=1'b1;
                wait(pready==1'b1);
                @(posedge pclk && pready);
                #10ps;
                psel=1'b1;
                pwrite=1'b0;
                penable=1'b0;
                paddr=addr+ i*4;
                pstrb=4'h0;
```

```
		pprot=prot;


		@(posedge pclk);
		#10ps;
		penable=1'b1;
		wait(pready==1'b1);
		@(posedge pclk && pready);
		r_data=prdata;
	end
	#10ps;
	penable=1'b0;
	psel=1'b0;
	access_ongoing=1'b0;
endtask
//*********************************************************************************
endmodule
```

7.3 AHB Driver BFM

```
`define ADDR_WD 32
`define WDATA_WD 32
`define RDATA_WD 32
`define SIZE_WD 3
module ahb_driver (
	input hclk,
	input hrstn,
	input hready,
```

```
            input [1:0] hresp,
            input [`RDATA_WD-1:0] hrdata,
            output logic hsel,
            output logic [`ADDR_WD-1:0]haddr,
            output logic hwrite,
            output logic [`SIZE_WD-1:0] hsize,
            output logic [2:0] hburst,
            output logic [3:0] hprot,
            output logic [1:0] htrans,
            output logic [`WDATA_WD-1:0] hwdata
            );
logic access_ongoing=1'b0;
logic [`RDATA_WD-1:0] rd_data;
logic [7:0] beat_size;
logic [11:0] txn_size;
logic [4:0] beats;
logic [31:0] x;
function integer clog2(input logic [11:0] value);
integer i;
begin
            for (i=0; value>0; i=i+1)
            value = value>>1;
end
```

```
return i;

endfunction



task wr_ahb (logic [`ADDR_WD-1:0] addr, //

            logic [`WDATA_WD-1:0] data,

                logic [16:0] incr_beats =4, //

                logic [7:0] num_txn = 'b1,

            logic [1:0] trans= 2'b10,   //default non-sequential txn

                logic [2:0]burst=3'b000,    //default single transfer

                logic [`SIZE_WD:0]size='b10,   //default 32 bits(1 word) transfer size

                logic [3:0] prot=4'b0011);  //default uncacheable,unbufferable,privilged,data access

            wait(access_ongoing=='b0);

            @(posedge hclk);

            access_ongoing='b1;

            for(int i=0;i<num_txn;i++) begin

                        #10ps;

                        hsel='b1;

                        hwrite=1'b1;

                        hprot=prot;

                        hburst=burst;

                        hsize=size;

                        htrans=2'b10;     //considering 1st txn can never be busy

                        beat_size=1<<size; //alternate beat_size=2**size;
```

```
case (burst)
        1:beats=incr_beats;
        2:beats=4;
        3:beats=4;
        4:beats=8;
        5:beats=8;
        6:beats=16;
        7:beats=16;
        default:beats=1;
endcase
txn_size=beats*beat_size;
haddr=addr+i*txn_size;
repeat (beats-1) begin
        wait (hready);
        @(posedge hclk && hready);
        #10ps;
        if (htrans != 'b01 )hwdata=data;
        htrans='b11;
        x=~((1<<(clog2(txn_size)+1))-1);
        if (hburst%2==1'b1)
                haddr=haddr+beat_size; //address for non wrap increment
        else if ((addr& x) +((beats-1)*beat_size))          //wrap around address condition
                        haddr=addr & x;
```

```
                    else  haddr=haddr+beat_size;
            end
            wait(hready);
            @(posedge hclk && hready);
        end
        #10ps;
        if (htrans!= 'b01) hwdata=data;
        htrans='b00;
        hburst='b00;
        hsel='b0;
        access_ongoing='b0;
endtask


task rd_ahb (logic [`ADDR_WD-1:0] addr, //
          output logic [`RDATA_WD-1:0] data,
          input logic [16:0] incr_beats =4, //
          logic [7:0] num_txn = 'b1,
        logic [1:0] trans= 2'b10,   //default non-sequential txn
          logic [2:0]burst=3'b000,    //default single transfer
          logic [`SIZE_WD:0]size='b10,   //default 32 bits(1 word) transfer size
          logic [3:0] prot=4'b0011);  //default uncacheable,unbufferable,privilged,data access
        wait(access_ongoing=='b0);
        @(posedge hclk);
```

```
access_ongoing='b1;

for(int i=0;i<num_txn;i++) begin

    #10ps;

    hsel='b1;

    hwrite=1'b0;

    hprot=prot;

    hburst=burst;

    hsize=size;

    htrans=2'b10;    //considering 1st txn can never be busy

    beat_size=1<<size;

    case (burst)

        1:beats=incr_beats;

        2:beats=4;

        3:beats=4;

        4:beats=8;

        5:beats=8;

        6:beats=16;

        7:beats=16;

        default:beats=1;

    endcase

    txn_size=beats*beat_size;

    haddr=addr+i*txn_size;

    repeat (beats-1) begin
```

```
                    wait (hready);

                    @(posedge hclk && hready);

                    #10ps;

                    if (htrans != 'b01)

                     begin

                      data=hrdata;

                     end

                    htrans='b11;

                    x=~((1<<(clog2(txn_size)+1))-1);

                    if (hburst%2==1'b1)

                      haddr = haddr+beat_size; //address for non wrap increment

                    else if ((addr & x) +((beats-1)*beat_size))

                                haddr = addr & x;

                        else

                                haddr=haddr+beat_size;

            end

            wait(hready);

            @(posedge hclk && hready);

end

#10ps;

if (htrans!= 'b01) data=hrdata;

htrans='b00;

hburst='b00;
```

```
        hsel='b0;
        access_ongoing='b0;
endtask
endmodule
```

7.4 ATB SLAVE MODEL

```
module atb_slave #(parameter int N = 128, parameter int WAIT_STATE = 0)(
  ////// Global Signals
  input atclk,
  input atclken,
  input atresetn,
  ////// Data signals
  input [$clog2(N)-4:0] atbytes,
  input [N-1:0] atdata,
  input [6:0] atid,
  input atvalid,
  output logic atready,
  ////// Flush control signals
  input afready,
  output logic afvalid
  ////// sync request signals
  //output logic syncreq
);
  logic [$clog2(WAIT_STATE+1)-1:0] ready;
```

```
logic clk;

enum logic [1:0] {RESET = 2'b00, IDLE=2'b01, TRANS=2'b10} state, next_state;

assign clk = atclken & atclk;

// state assignment

  always_ff @(posedge clk or negedge atresetn)

    if (!atresetn) begin

      state <= RESET;

    end

    else

      state <= next_state;

// Next State Decoder

  always_comb

    case (state)

      RESET: next_state = atresetn ? IDLE:RESET;

      IDLE:  next_state = atvalid ? TRANS:IDLE;

      TRANS: next_state = atvalid ? TRANS:IDLE;

    endcase

// Output Decoder

  always_comb

    case (state)

      RESET: atready = 1;

      IDLE : atready = 0;

      TRANS: begin atready = WAIT_STATE ? (ready == WAIT_STATE) ? 1:0 : 1; end
```

```
   endcase

  always_ff@(posedge clk, negedge atresetn)

   if (!atresetn) begin

       ready <= 0;

   end

   else

    if (state==TRANS)

        if (ready == WAIT_STATE)

             ready = 0;

        else

             ready = ready+1;

endmodule
```

601762003

ORIGINALITY REPORT

14%	11%	3%	7%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

	Source	
1	jultika.oulu.fi Internet Source	3%
2	dspace.cc.tut.fi Internet Source	2%
3	ddd.uab.cat Internet Source	2%
4	polimage.polito.it Internet Source	1%
5	Submitted to CSU, San Jose State University Student Paper	1%
6	Ashok B. Mehta. "ASIC/SoC Functional Design Verification", Springer Nature, 2018 Publication	1%
7	Submitted to Rochester Institute of Technology Student Paper	1%
8	www.synopsys.com Internet Source	<1%
9	ir.lib.uth.gr	

Internet Source

<1%

10	www.ijert.org Internet Source	<1%
11	Submitted to Indian Institute of Technology Jodhpur Student Paper	<1%
12	Submitted to Institute of Technology Carlow Student Paper	<1%
13	Submitted to SASTRA University Student Paper	<1%
14	SystemVerilog Assertions and Functional Coverage, 2016. Publication	<1%
15	Submitted to Amity University Student Paper	<1%
16	www.eda.org Internet Source	<1%
17	www.clivar.com Internet Source	<1%
18	sutherland-hdl.com Internet Source	<1%
19	Submitted to Universiti Teknologi Malaysia Student Paper	<1%

20	www.iject.org Internet Source	<1%
21	Submitted to Amrita Vishwa Vidyapeetham Student Paper	<1%
22	www.csauthors.net Internet Source	<1%
23	Submitted to CSU, San Jose State University Student Paper	<1%
24	Brian Bailey, Grant Martin. "ESL Models and their Application", Springer Nature, 2010 Publication	<1%
25	Shuo Wang, Fan Zhang, Jianwei Dai, Lei Wang, Zhijie Jerry Shi. "Making register file resistant to power analysis attacks", 2008 IEEE International Conference on Computer Design, 2008 Publication	<1%
26	www.diit.unict.it Internet Source	<1%
27	Harry Foster, Adam Krolnik, David Lacey. "Assertion-Based Design", Springer Nature, 2005 Publication	<1%
28	web.it.kth.se Internet Source	<1%

29	standards.ieee.org Internet Source	<1%
30	Braga, Diego, Franco Fummi, Graziano Pravadelli, and Sara Vinco. "The strange pair: IP-XACT and univerCM to integrate heterogeneous embedded systems", 2012 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2012. Publication	<1%
31	Submitted to Visvesvaraya Technological University Student Paper	<1%

Exclude quotes	On	Exclude matches	< 8 words
Exclude bibliography	On		