Design and implementation of AXI UPSIZER DOWNSIZER for GPU based subsystem

Major Project Report

Submitted in fulfillment of the requirements for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Kush Rami

(17MECE15)

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May, 2019

Design and implementation of AXI UPSIZER DOWNSIZER for GPU based subsystem

Major Project Report

Submitted in fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

By

Kush Rami
(17MECE15)

Under the guidance of

External Project Guide:

Mr. Prodip Kumar Kundu

Staff Engineer,

ARM embeddded technology PVT. LTD.,

Banglore.

Internal Project Guide:

Dr. Tanish Zaveri

Professor

E&C Engineering Department,

School of Technology, Nirma University

Electronics & Communication Engineering Department

School of Technology-Nirma University

Ahmedabad-382 481

May 2019

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Kush Rami
17MECE15

Certificate

This is to certify that the Major Project entitled **"Design and implementation of AXI UPSIZER DOWNSIZER for GPU based subsystem"** submitted by **Kush Rami (17MECE15)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr Tanish Zaveri
Internal Guide

Dr N. P. Gajjar
PG Coordinator (Embedded System)

Dr D. K. Kothari
Head, EC Dept.

Dr Alka Mahajan
Director, IT - NU

Date :

Place : Ahmedabad

Statement of Originality

I, **Kush Rami**, Roll. No. **17MECE15**, give undertaking that the Project Report on "**Design and implementation of AXI UPSIZER DOWNSIZER for GPU based subsystem**" submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in **Electronics and communication (Embedded System)** of Institute of Technology, Nirma University, Ahmedabad, contains no material that has been awarded for any degree or diploma in any university or school in any territory to the best of my knowledge. It is the original work carried out by me as part of on-going research work in ARM embedded technology Pvt. Ltd. and I give assurance that no attempt of plagiarism has been made. It contains no material that is previously published or written, except where reference has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

____

Date:

Place:

Endorsed by

Dr. Tanish Zaveri

Acknowledgement

Let me take the opportunity to express my deep regards to Mr. Prodip Kumar Kundu (Project Manager) for assigning me such project and providing his guidance and constant encouragement during the project. I would also like to thank Rakshita Agarwal (Mentor) for her guidance, help and inspiring me to put my best efforts.

I would like to express my gratitude & sincere thanks for generous assistance to my guide Dr. N. P. Gajjar, Professor,PG Coordinator, Embedded System, Institute of Technology, Nirma University, Ahmedabad for his guidance and constant encouragement during my course of project. Special thanks to Dr. Tanish Zaveri (Professor, ITNU) who has always been an inspiration and guided us with his experience.

I would also like to thank all faculty members of Nirma University for providing encouragement and exchanging knowledge during my post-graduate program.

I would like to express my gratitude towards my parents and my sister for constant support and encouragement in life. I also wish to express my heartfelt appreciation to my friends and colleagues at ARM who have rendered their support throughout my project, both explicitly and implicitly.

- Kush Rami
17MECE15

Contents

Declaration iii

Certificate iv

Statement of Originality v

Acknowledgements vi

Abstract xiii

Abbreviation Notation and Nomenclature xiv

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement 1

1.3 Approach 2

1.4 Scope of Work 2

1.5 Outline of Thesis 3

2 Literature Survey 4

2.1 Overview 4

2.2 Overview of AXI Protocol 5

2.3 AXI UPSIZER DOWNSIZER design overview 6

2.4 AXI Signal Description 7

3 Design of AXI UPSIZER DOWNSIZER Module **11**

3.1 Block Diagram 11

3.2 Introduction to AXI UPSIZER DOWNSIZER module 12

3.3 Schematic Design 13

3.4 AXI UPSIZER DOWNSIZER -TOP Block Diagram 14

3.4.1 Request Controller 14

3.4.2 Read Buffer Controller 16

3.4.3 Writer Buffer Controller 17

3.4.4 Response Controller 18

4 Implementation of RTL design **20**

4.1 Implementation logic 20

4.1.1 Request controller 20

4.1.2 Response controller 21

4.1.3 Write/read Buffer Controller 21

4.2 Results 23

4.2.1 Compilation results 23

4.2.2 Write transaction result 24

4.2.3 read transaction result 24

5 Verification and Results **27**

5.1 Verification Plan 27

5.1.1 UVM Top level Testbench 27

5.1.2 Performance Counters 28

5.1.3 Scoreboard Implementation 30

5.1.4 UVM Testbench parameters 32

5.2 Results 35

6 Conclusion **39**

6.1 Conclusion 39

6.2 Future work 40

References **41**

List of Tables

2.1 Write Address channel signals[1] 8
2.2 read Address channel signals [1] 9
2.3 write data channel signals[1] 9
2.4 read data channel signals [1] 10
2.5 write response channel signals[1] 10

List of Figures

3.1 Block Diagram 11

3.2 Module Block Diagram 12

3.3 Behavioral Block Diagram 14

3.4 AXI-TOP Block Diagram 15

4.1 Logic diagram of Request controller 20

4.2 Logic diagram of Response controller 21

4.3 Logic diagram of Write/read Buffer controller 22

4.4 Result of compile log 23

4.5 DUT instantiating in simulator 24

4.6 Signals and data of write transaction 25

4.7 Timing Waveform of write transaction 25

4.8 Signals and data of write transaction 26

4.9 Timing Waveform of write transaction 26

5.1 Block diagram of testbench 28

5.2 Scoreboard design 31

5.3 Master Configuration parameters 32

5.4 Slave 1 Configuration parameters 33

5.5 Slave 2 Configuration parameters 34

5.6 Result of Master transaction 35

5.7 Result of Slave 1 transaction 36

5.8 Result of Slave 2 transaction 37
5.9 Result of Basic test 38

Abstract

To increase the performance of ARM System on chip(SoC) subsystem, need to increase the data flow from system components to memory. This report focuses on how advanced micro controller bus architecture(AMBA) protocol can be expanded from 128bit to 256bit data width on the bus. To make this possible, one internal module is developed which takes care of all protocol specification and rules. The generated module is verified by number of tests which includes read and write transactions. The UVM test bench is developed for this particular unit level verification, for which protocol checkers are implemented and performance counter will measure successful number of transaction. After that performance parameters of the system will be evaluated and the module will be integrated in the subsystem to increase the data flow. This project can be extended to for protocol like ACE and CHI.

Abbreviation Notation and Nomenclature

AMBA .... Advanced Micro controller Bus Interface

AXI .... Advanced Extensible Interface

ACE .... AXI coherency entension

GPU .... Graphics Processing Unit

CCI .... Cache coherent interconnect

DMC .... Dynamic memory controller

APB .... Advanced Peripheral Bus

DMA .... Direct Memory Access

Chapter 1

Introduction

1.1 Motivation

The existing subsystem does the data transaction of 128bit data width from Graphics processing Unit(GPU) to memory. The internal structure of this path contains GPU, Cache coherent interconnect(CCI), dynamic memory controller(DMC), and memory. At a time GPU can only initiate the transaction of 128bit for read, write or snoop transactions. From GPU to CCI, data transaction happens based on Advanced Microcontroller Bus Architecture(AMBA) Protocol. The protocol specification supports the data transaction from 64bits to 1024bits [1]. So we want to expand the data transaction width from 128bits to 256bits so that GPU can run more payload in existing subsystem. This enhancement has may challenges to in terms of configuration support, hardware support, performance parameters etc. which we want to overcome by developing this project.

1.2 Problem Statement

In The subsystem GPU has AXI cohenrancy extension (ACE) Master and CCI has slave pair for transaction. The master will initiate the 256bits transaction as it can be configured from 128 bit to 256 bits. In CCI the ACE slave can not be

configured form 128bit to 256 bits. But CCI has availability of two slaves[4]. So, we need to add a verilog module that makes that happen such a way that 256bit transaction which was initiated by master, gets completed in as 128 bits transaction for CCI slaves. There are ways of approaching this problem which is discussed in next section.

1.3 Approach

There are two approach to achieve this enhancement goal.

1. We can make a module between GPU and CCI where we can use one 256bits ACE master and two 128bits CCI slaves to make 256 bits read/write transaction happen. In this approach we need extra slave from CCI side.

2. We can make a module between GPU and CCI which takes 256bits transaction from GPU master and forwards it to CCI slave with two serial 128bits transaction. In this approach we need extra time to complete the transaction.

The first approach is the one being implemented in this report and the resulting waveform. The second approach will be implemented in the future scope of work so that we can compare the performance of implementing both the approaches.

1.4 Scope of Work

To design a verilog module that make this change happen in exsting subsystem[5]. The whole module will have number of sub modules which takes care of read and write transaction separately. The design requirement of the module must match with the existing system parameters. After designing the full top module we meed to test it in questa simulator. This unit level testing will verify the design and generate the waveform as per timing waveform requirement.

1.5 Outline of Thesis

The whole thesis focuses on the design, implementation and testing of the module which needs to be implemented in the approach 1. First chapter talks about the introduction of the whole project. Second chapter gives highlights on literature survey and design requirement gatherings. The third chapter talks about the implementation of the module.The fourth chapter gives the resulting waveform of tested design. The fifth chapter talks about future work and conclusions.

Chapter 2

Literature Survey

2.1 Overview

Designing a custom system-on-chip doesn't have to be risky difficult or expensive. Let's walk through the five steps to get from concept to silicon the fastest easiest lowest risk way with arm. The first step is defining the needs of the system the functionality and the requirements then select the IP blocks that satisfy your requirements arm off is industry-leading arm cortex-m0 and cortex m3 CPUs through arm design start quickly easily and for no upfront fee then you need to connect the IP and peripherals together arm system design kits include a preverified subsystem to connect the CPU with the other system components for faster more confident designs next ensure the solution you've built meets all functional requirements using arm verified IP means you only need to focus on verifying the additional IP and connections finally you're ready to perform the implementation and go to production there is a wide choice of affordable silicon nodes available from a range of manufacturers and arm offers free access to thousands of physical IP libraries to help and if you don't have the design experienced in-house arm proof design partners can help develop all or part of your custom SOC.

The subsystem of ARM consist many things on a system on chip. the step 4

of verification and testing of design is done by my team. For my project the main focus is enhance the system performance by increasing data transaction width. For that we need to make sure how our design will get modified in process and the existing system will mot get affected because of the new change. There are two version of Advanced micro controller bus architecture (AMBA). First version is Advanced extensible interface (AXI) and the second is AXI coherency extension (ACE) [2]. This two protocol used as we need any data transaction between the two components like GPU and Interconnect. Next sections in this chapter gives the overview of the AXI and ACE protocol signals. Then how the module design needs to be implemented in our subsystem.

2.2 Overview of AXI Protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between master and slave components.

The AXI protocol:

- The axi protocol is made for high bandwidth and timing low-latency designs.
- It is also made for high-frequency operation which do not use complex bridges.
- For most of components the AXI protocol supports all the interfaces
- The axi protocol is suitable for memeory controllers.
- For interconnect architecture it provides good flexibility.
- For AHB APB interface it is backward compatible.

The key features of the AXI protocol are:

- AXI protocol has different address, and data space.
- It supports unaligned transfers.
- When it uses only one start address and does the burst mode transfers.
- For DMA it supports different address and data channels.
- Different multiple transfers can be supported in axi protocol.
- Out of order transactions are supported.

2.3 AXI UPSIZER DOWNSIZER design overview

From the above channel description and definition we know that AXI protocol supports the 256 data width transaction for both read and write.So in our subsystem we have Graphics processing unit (GPU) and Cache coherent interconnect (CCI) that does the transaction based on ACE protocol which is advance version of AXI protocol. The GPU ACE master and CCI ACE slave combination make the read and write transaction happen only taking 128 bits data at a time. Now GPU master can be configured with the parameter of AXI data width. So by changing that we can do the transaction of 256bits read and write transaction. But from interconnect that AXI data width is not configurable. But we have two slaves available in interconnect of 128bits data width. So using both of them we can make a 256bits transaction possible[4]. The AXI UPSIZER DOWNSIZER module will handle the dividing upper bits of data to slave 1 and lower bits of data to slave 2. The more implementation detailes are discussed in chapter 3.

2.4 AXI Signal Description

Global signals

Signal: ACLK

Source: Clock source

Description: Global clock signal

Signal: ARESETn

Source: Reset source

Description: Global reset signal, active Low

a. **Write address channel**

b. **Read address channel**

c. **Write data channel**

d. **Read data channel**

e. **Write response channel**

Signal	Source	Description
AWID	Master	The identification tag for the write address group of signals.
AWADDR	Master	The write address gives the address of the first transfer in a write burst transaction.
AWLEN	Master	The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
AWSIZE	Master	Indicates the size of each transfer in the burst.
AWBURST	Master	The burst type and the size information, determine how the address for each transfer within the burst is calculated.
AWLOCK	Master	Provides additional information about the atomic characteristics of the transfer.
AWCACHE	Master	Indicates how transactions are required to progress through a system.
AWPROT	Master	Indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
AWQOS	Master	QoS identifier sent for each write transaction.
AWREGION	Master	Permits a single physical interface on a slave to be used for multiple logical interfaces.
AWUSER	Master	Optional User-defined signal in the write address channel.
AWVALID	Master	Indicates that the channel is signaling valid write address and control information.
AWREADY	Slave	Indicates that the slave is ready to accept an address and associated control signals.

Table 2.1: Write Address channel signals[1]

Signal	Source	Description
ARID	Master	The identification tag for the read address group of signals.
ARADDR	Master	The write address gives the address of the first transfer in a read burst transaction.
ARLEN	Master	The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
ARSIZE	Master	Indicates the size of each transfer in the burst.
ARBURST	Master	The burst type and the size information, determine how the address for each transfer within the burst is calculated.
ARLOCK	Master	Provides additional information about the atomic characteristics of the transfer.
ARCACHE	Master	Indicates how transactions are required to progress through a system.
ARPROT	Master	Indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
ARQOS	Master	QoS identifier sent for each read transaction.
ARREGION	Master	Permits a single physical interface on a slave to be used for multiple logical interfaces.
ARUSER	Master	Optional User-defined signal in the read address channel.
ARVALID	Master	Indicates that the channel is signaling valid read address and control information.
ARREADY	Slave	Indicates that the slave is ready to accept an address and associated control signals.

Table 2.2: read Address channel signals [1]

Signal	Source	Description
WID	Master	The ID tag of the write data transfer.
WDATA	Master	Write data
WSTRB	Master	Indicates that the byte lanes that hold valid data. There is one write strobe bit for each 8 bits of the write data bus.
WLAST	Master	Indicates the last transfer in a write burst.
WUSER	Master	Optional User-defined signal in the write data channel.
WVALID	Master	This signal indicates that valid write data and strobes are available.
WREADY	Slave	This signal indicates that the slave can accept the write data.

Table 2.3: write data channel signals[1]

Signal	Source	Description
RID	Slave	The ID tag of the read data transfer.
RDATA	Slave	read data
RRESP	Slave	Indicates the status of the read transfer.
RLAST	Slave	Indicates the last transfer in a read burst
RUSER	Slave	Optional User-defined signal in the read data channel.
RVALID	Slave	The channel is signaling the required read data.
RREADY	Master	Indicates that the master can accept the read data and response information.

Table 2.4: read data channel signals [1]

Signal	Source	Description
BID	Slave	The ID tag of the write response.
BRESP	Slave	Indicates the status of the write transaction.
BUSER	Slave	Optional User-defined signal in the write response channel.
BVALID	Slave	Indicates that the channel is signaling a valid write response..
BREADY	Master	Indicates that the master can accept a write response.

Table 2.5: write response channel signals[1]

Chapter 3

Design of AXI UPSIZER DOWNSIZER Module

3.1 Block Diagram

The block diagram of Subsystem:

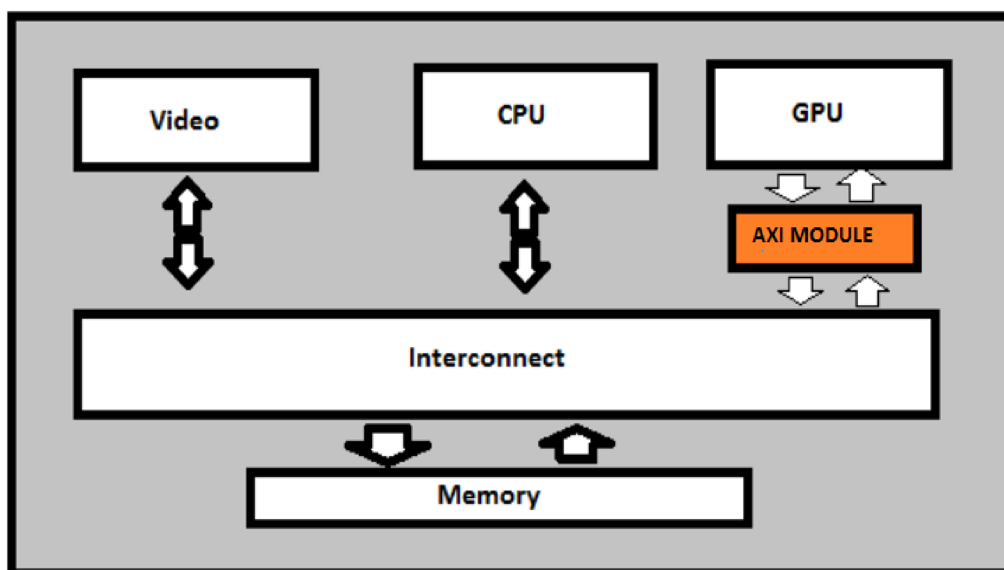

Figure 3.1: Block Diagram

On the first phase we are implementing Module for AXI protocol. So implementation of module is known as AXI UPSIZER DOWNSIZER.
The block diagram of AXI UPSIZER DOWNSIZER:

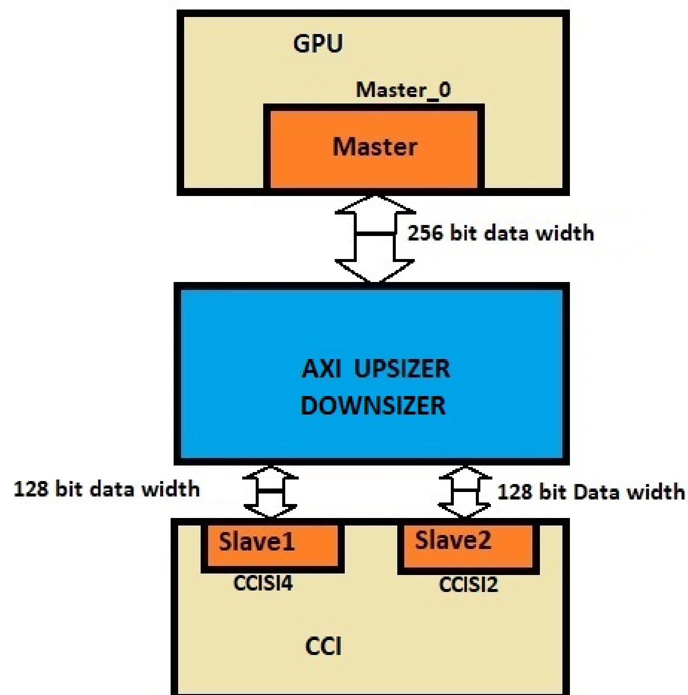

Figure 3.2: Module Block Diagram

3.2 Introduction to AXI UPSIZER DOWNSIZER module

Existing Subsystem has bus width of 128 bit from GPU to CCI to DMC. We want to make that 256bit. From GPU Master, the data width is configurable from 128bits to 256bits. But CCI Slaves data width is not configurable. So, we can have two 128bit slaves of CCI connected to one 256bits master of GPU, in between one AXI

UPSIZER DOWNSIZER module present.The main functionality of AXI UPSIZER DOWNSIZER block is to handle a particular data transaction from a master into two slaves and combine the responses from these two slaves and send it to the master as a single response.

- AXI UPSIZER DOWNSIZER block has one master interface and two slave interfaces. Master has 256bit data width and slaves are 128bit data width which supports AXI.
- Master sends the same AXI request to two slaves.
- For a particular transaction master will provide 256bit data and it will be distributed into two 128bit slave data and combine the response of two slaves and give it back to master.
- The block takes care of all axi interfaces requests and response signals.

3.3 Schematic Design

Figure below shows the high-level block diagram of AXI UPSIZER DOWNSIZER.

The block mainly consists of

- Request Controller
- Read Buffer/memory Controller
- Write Buffer/memory Controller
- Response Controller
- Snoop Signals controller

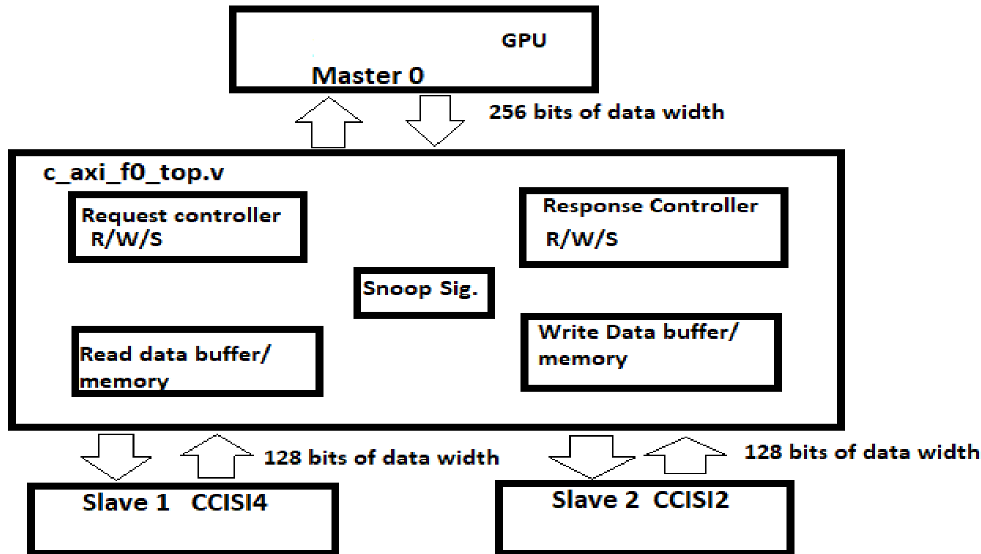

Figure 3.3: Behavioral Block Diagram

3.4 AXI UPSIZER DOWNSIZER -TOP Block Diagram

3.4.1 Request Controller

This block will provide the control logic for request channels Read Address Channel, Write Address Channel, Write Data Channel, Snoop address channel and snoop data channel.

a. Read address channel

b. Write address channel

c. Snoop Address channel

d. Write data channel

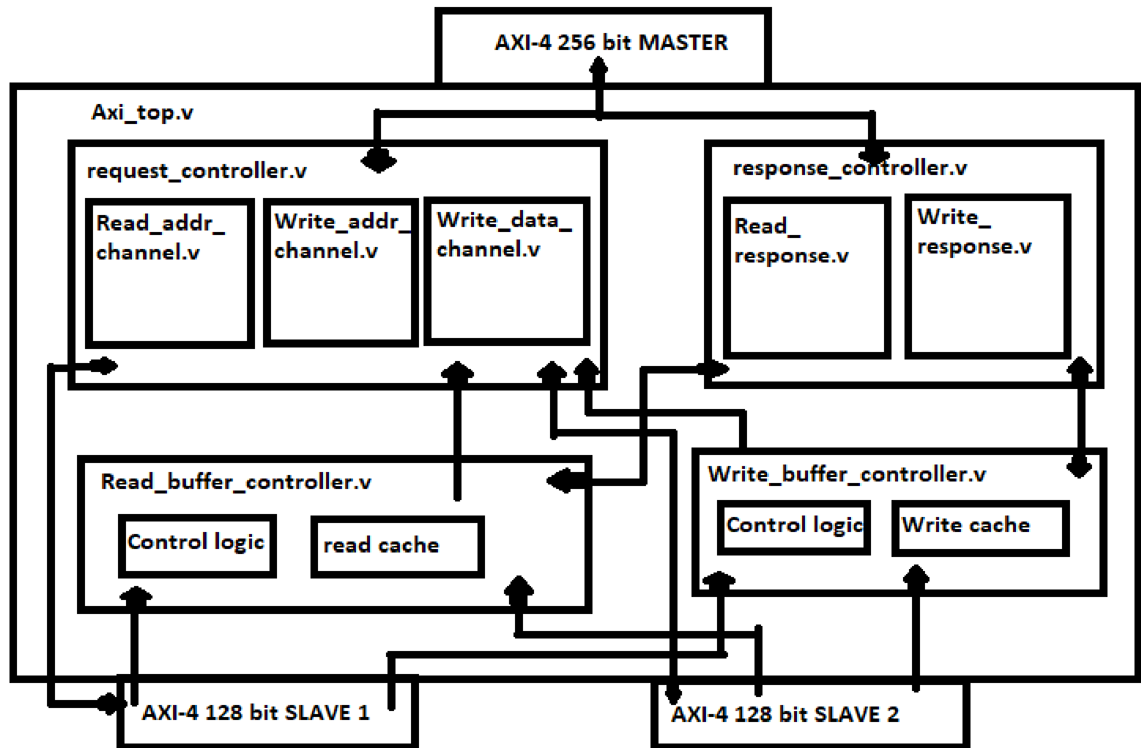

Figure 3.4: AXI-TOP Block Diagram

e. Snoop data channel

Process flow:

- The requests from all five channels are broadcasted to both the slaves.
- The main functionality of this block is to generate slave ready to master only when both the slaves have responded for a particular transaction.
- For each master request, Controller will be generating two internal valid signals one each to two slaves. By this way valid signal of fastest slave will be pulled low after it gives the ready avoiding the duplication of same requests.

- The request controller makes the ADDRESS field such that it will give first slave half ADDRESS of the memory and second slave second half ADDRESS of memory to read or write or snoop transaction.

3.4.2 Read Buffer Controller

Read buffer controller implements one memory buffer for 256 bits data. Loading data coming from slave1 128 lower bits and 128 upper bits from slave2. Combining them to 256 bits and give it back to GPU master.

Read response memory:

The depth of the memory array is 64 and width is 2122.Read buffer controller takes input from response controller to update the send field which indicates how many resolved beats are send to master. Also it flushes the array and rearrange the priority scheme once it gets the flag from response controller stating that all the response for a particular array is send.

Slave 1 read response:

- Once RVALID from Slave1 comes, the controller decodes the corresponding memory array with RID ARID matching. After decoding, update the rlast and increment the beat field.

- Beat field initially will be 0. Based on the occurrence of each RVALID, beat will be incremented.

- Once we update the array w.r.t RVALID, check the corresponding data field in the memory array. If these two fields match, then update the resolved bit of the array. Otherwise continue checking until the next RVALID comes.

Slave 2 read response:

- Once RVALID from Slave2 comes, the controller decodes the corresponding memory array with RID ARID matching. After decoding, update the rlast and increment the beat field.
- Beat field initially will be 0. Based on the occurrence of each RVALID, beat will be incremented.
- Once we update the array w.r.t RVALID, check the corresponding data field in the memory array. If these two fields match, then update the resolved bit of the array. Otherwise continue checking until the next RVALID comes.

3.4.3 Writer Buffer Controller

Write buffer controller implements one memory buffer for 256 bits data and distribute that in 128 lower bits to slave1 and 128 upper bits to slave2.
Write response memory:
The depth of the memory array is 64 and width is 21.Write buffer controller takes input from response controller to update the send field which indicates how many resolved beats are send to master. Also it flushes the array and rearrange the priority scheme once it gets the flag from response controller stating that all the response for a particular array is send.

Slave 1 write response:

- Once BVALID from Slave1 comes, the controller decodes the corresponding memory array with BID AWID matching. After decoding, update the wlast and increment the beat field.

- Beat field initially will be 0. Based on the occurrence of each BVALID, beat will be incremented.
- Once we update the array w.r.t BVALID, check the corresponding data field in the memory array. If these two fields match, then update the resolved bit of the array. Otherwise continue checking until the next BVALID comes.

Slave 2 write response:

- Once BVALID from Slave2 comes, the controller decodes the corresponding memory array with BID AWID matching. After decoding, update the wlast and increment the beat field.
- Beat field initially will be 0. Based on the occurrence of each BVALID, beat will be incremented.
- Once we update the array w.r.t BVALID, check the corresponding data field in the memory array. If these two fields match, then update the resolved bit of the array. Otherwise continue checking until the next BVALID comes.

3.4.4 Response Controller

This block will have the control logic for channels Read Response (rdata) Channel, Write Response Channel and snoop response channel.

- The basic functionality of this logic is to sent the final response back to master based on the resolved response field and send(only in read case) field in memory array.

- Logic starts checking from the first array and based on resolved state, controller send the response to master w.r.t master ready. Controller makes sure that response will be send every cycle if any resolved beat is available.

- Once all the responses in an array is send to master, then this information will be send to buffer controllers to flush the array and rearrange the priority setting.

Chapter 4

Implementation of RTL design

4.1 Implementation logic

4.1.1 Request controller

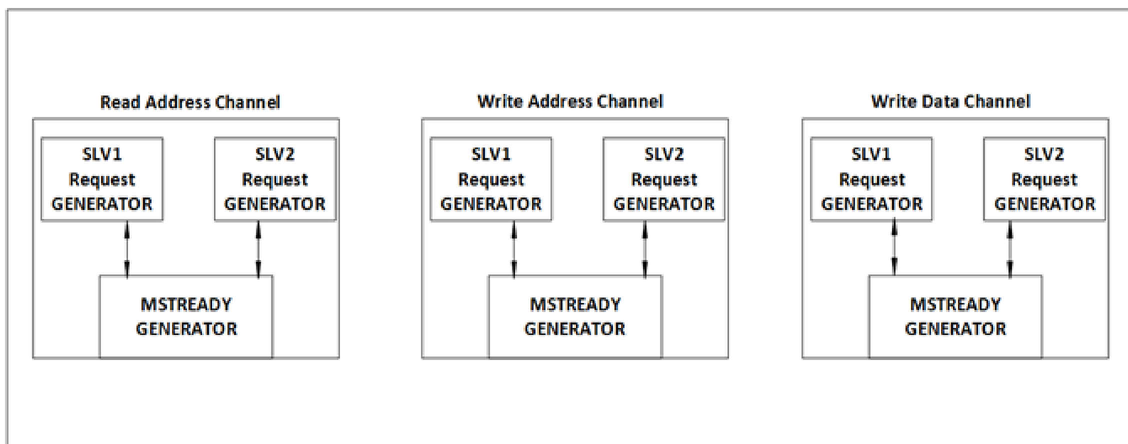

Figure 4.1: Logic diagram of Request controller

Request Controller block provides logic for the following things

- Generation of Slave 1 and Slave 2 request signals for Read Address channel, Write Address channel and Write Data Channel based on slave ready signals.

- Generation of Ready signals to Master for the requests coming Read Address channel, Write Address channel and Write Data Channel.
- Request Controller also takes input from read/write buffer controllers or the generation of Master Ready for Read/Write address channels.

4.1.2 Response controller

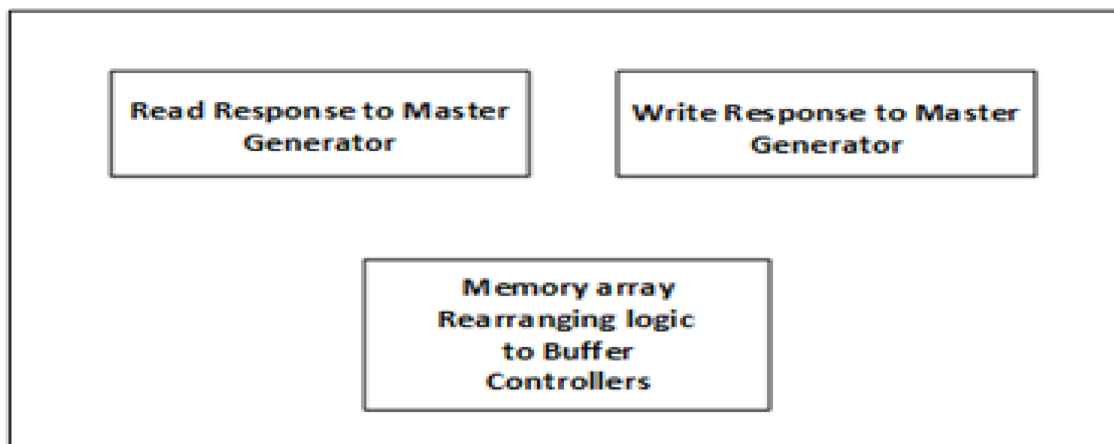

Figure 4.2: Logic diagram of Response controller

Response Controller block provides logic for the following things

- Send the Read/Write response to master based on the resolved bit information from Read/Write Buffer Controllers.
- Generation of memory array rearranging input to read/write controller based on the response send to master.

4.1.3 Write/read Buffer Controller

Read/Write Buffer Controller block provides logic for the following things

- Response memory for storing the response from Slave 1 and slave 2 for read transactions.

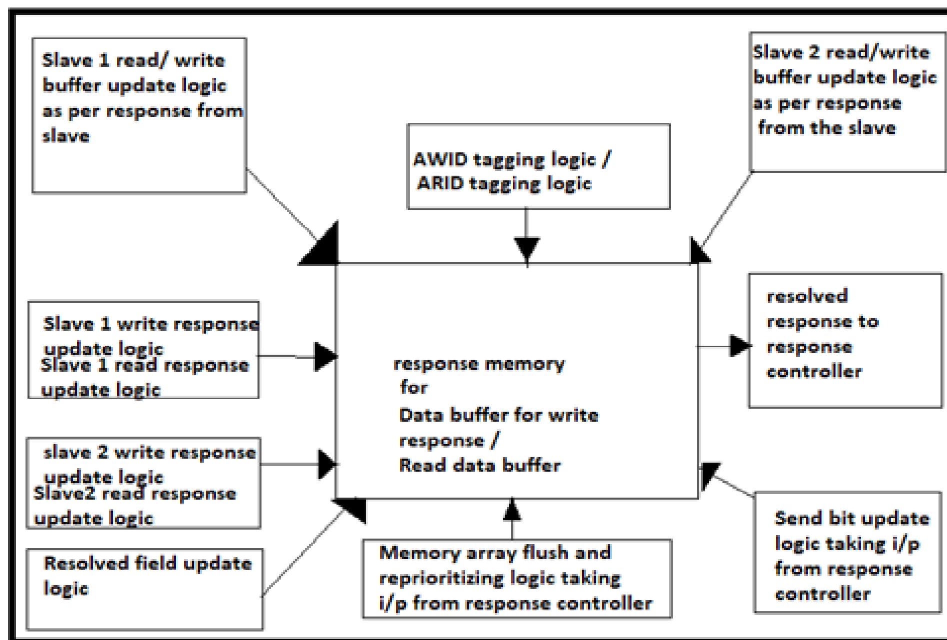

Figure 4.3: Logic diagram of Write/read Buffer controller

- Response memory for storing the response from Slave 1 and slave 2 for write response fields in write transactions.
- ARID/AWID tagging control to Response Memory.
- Two separate buffer update logic control for Response Memory based on response from Slave 1 and Slave 2 Interface.
- Slave 1 response control for updating Slave 1 fields in Response memory.
- Slave 2 response control for updating Slave 2 fields in Response memory.
- Control for the resolved bit updating of Response memory.
- Memory array flushing and re prioritizing control - taking input from Response Controller once all the information of a particular transaction is send to Master.

- Control for sending the Resolved response to Response controller.
- Control for updating the send field (in case of read) based on the resolved beats which are already send to master.

4.2 Results

4.2.1 Compilation results

Below Figure 4.4 and 4.5 shows that there are no syntax and no compilation error of generated RTL module.

```
-- Compiling package vip_axi4_slave_pkg
-- Compiling interface vip_axi4_signal_if
-- Compiling package axi_splitter_test_pkg
-- Importing package vip_apb4_uvc_pkg
-- Importing package vip_axi4_slave_pkg
-- Importing package vip_axi4_master_pkg
-- Compiling module Axi4PC
-- Compiling interface reset_if
-- Compiling module tbench
-- Importing package axi_splitter_test_pkg

Top level modules:
        tbench
End time: 10:17:01 on Dec 03,2018, Elapsed time: 0:00:02
Errors: 0, Warnings: 0
QuestaSim-64 vopt 10.4d_2 Compiler 2016.03 Mar  3 2016
```

Figure 4.4: Result of compile log

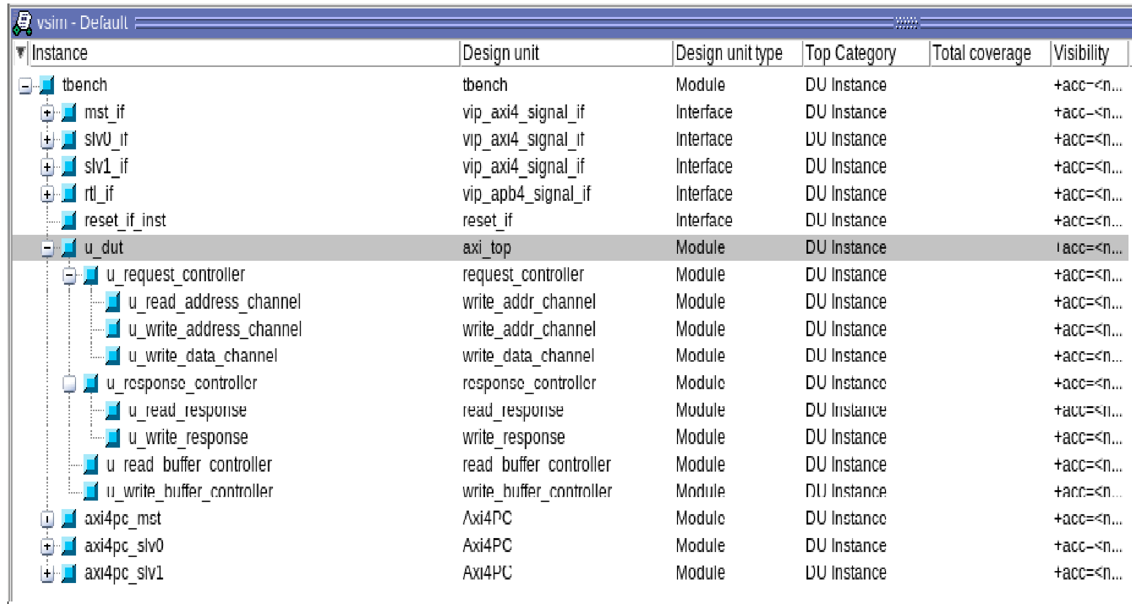

Figure 4.5: DUT instantiating in simulator

4.2.2 Write transaction result

Below Figures 4.6 and 4.7 shows that the write transaction happens for GPU as a 256 bits data send to the slaves. Module Divides that data into two parts of upper and lower bits to two different slaves.

4.2.3 read transaction result

Below Figures 4.8 and 4.9 shows that read data for the GPU master has come successfully as 256bits wider. where slave 1 and slave 2 has responded out of order in time.

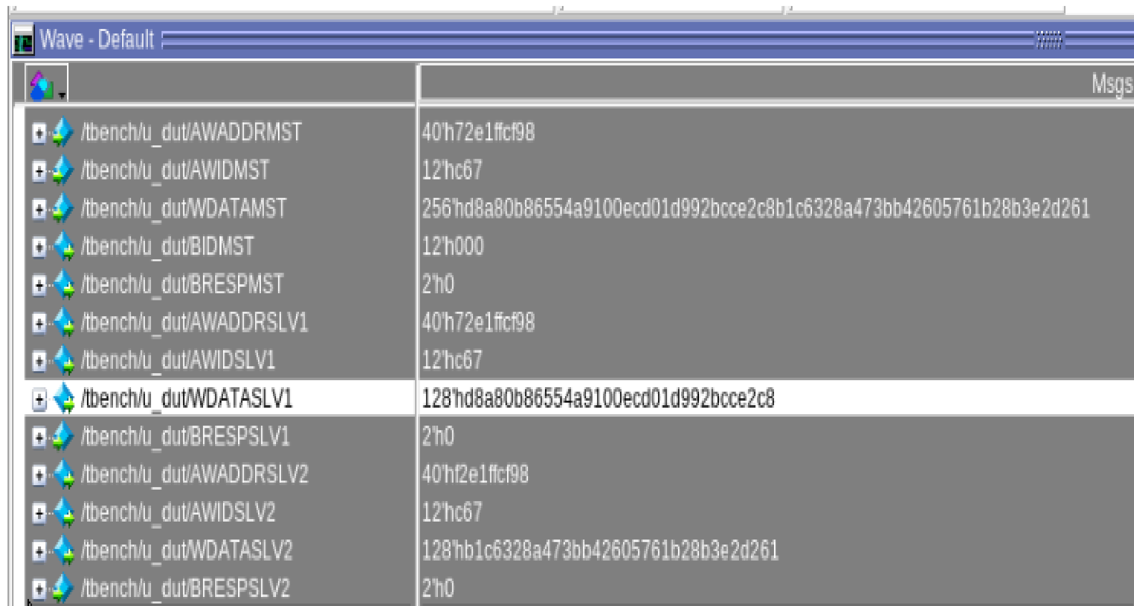

Figure 4.6: Signals and data of write transaction

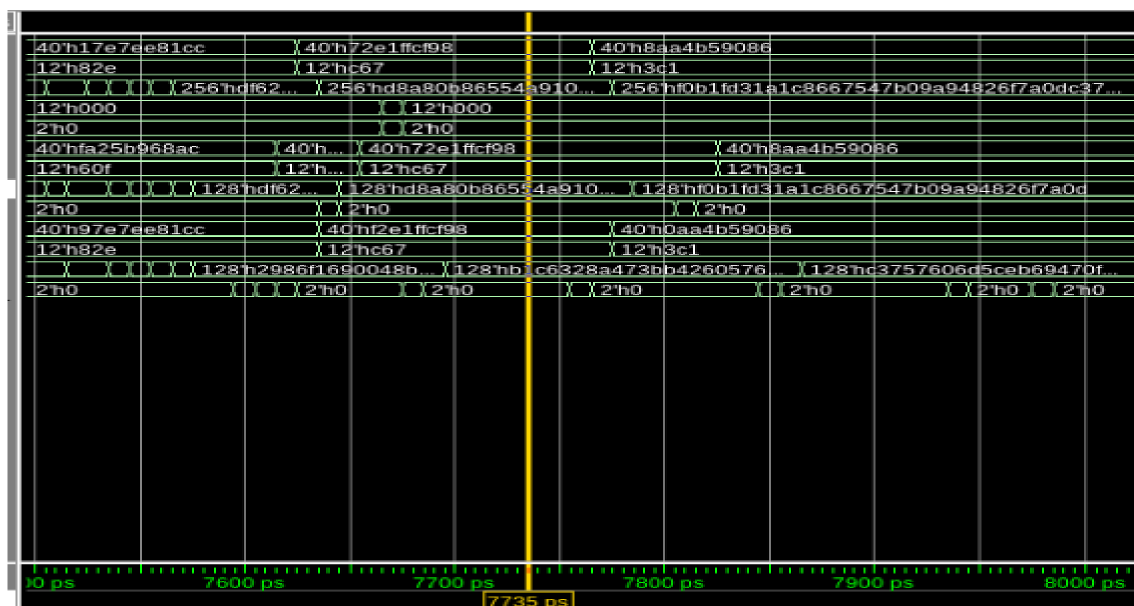

Figure 4.7: Timing Waveform of write transaction

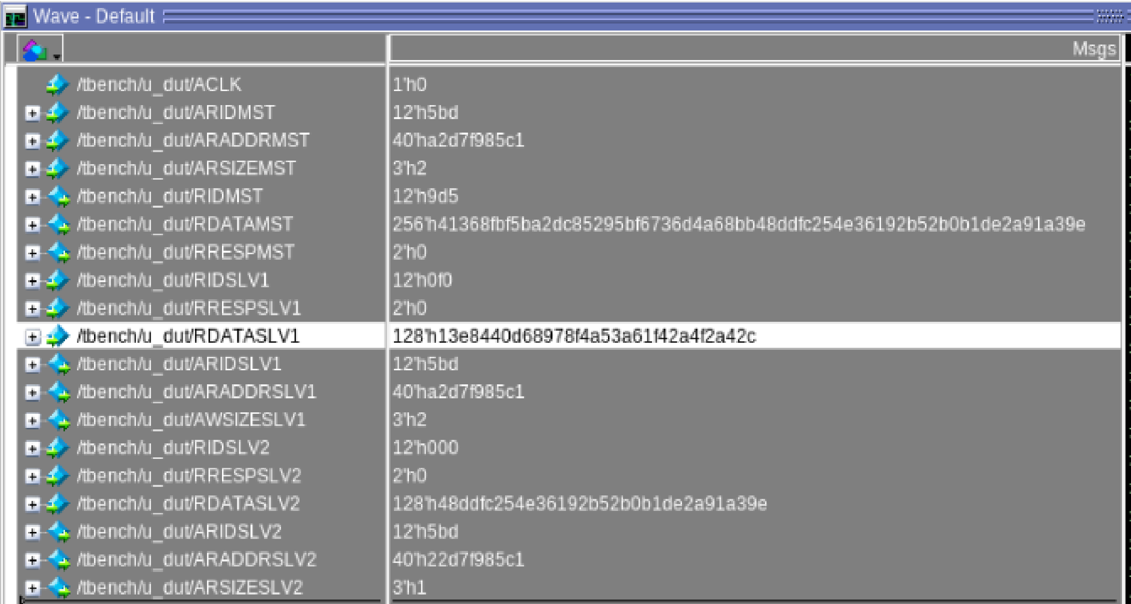

Figure 4.8: Signals and data of write transaction

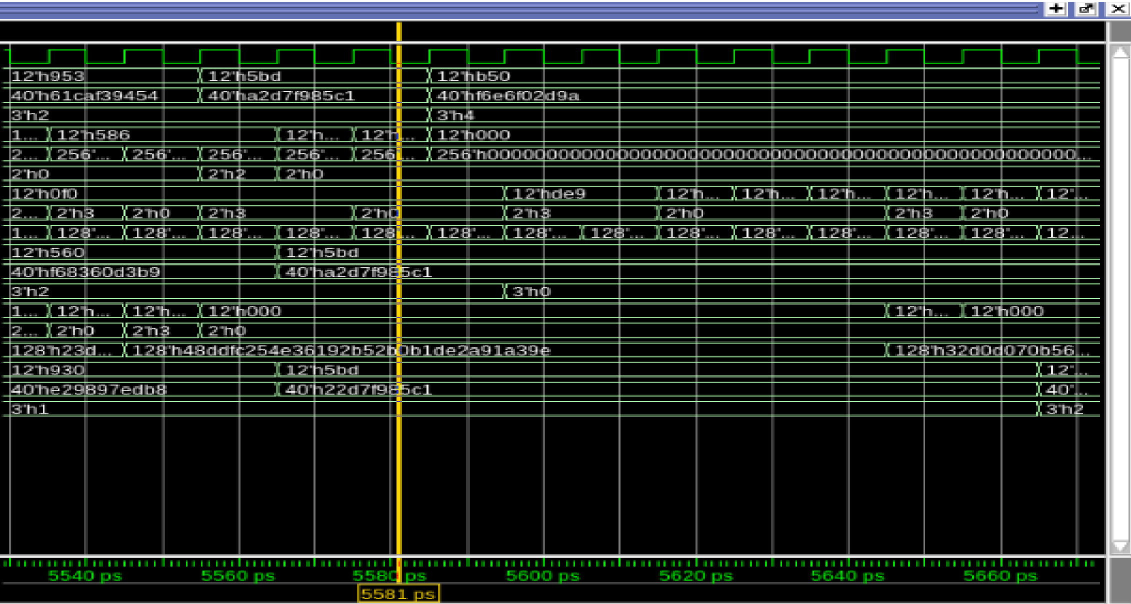

Figure 4.9: Timing Waveform of write transaction

Chapter 5

Verification and Results

5.1 Verification Plan

5.1.1 UVM Top level Testbench

This chapter describes the verification methodology for AXI UPSIZER DOWNSIZER. It lists the verification requirements and strategy to achieve those. It also explains the components/VIP used i.e. Scoreboard checking mechanism.

The main functionality of AXI UPSIZER DOWNSIZER block is to split a particular transaction from a master in to two slaves and combine the responses from these two slaves and send it to the master as a single response.

- AXI UPSIZER DOWNSIZER block has one master interface and two slave interfaces. Both master and slave are 128bit AXI4 interfaces.
- Master sends the same AXI request to two slaves.
- One slave gives the higher data width response and the other gives the lower data width response for a given transaction.
- The block takes care of broadcasting the AXI requests to both the slaves and from the response path, it properly buffers all the responses and sends it back

to master only when both SlaveData response and SlaveTiming response is available for a particular transaction.

- In addition to this, performance counters have been implemented into the design, accessible by an APB interface.

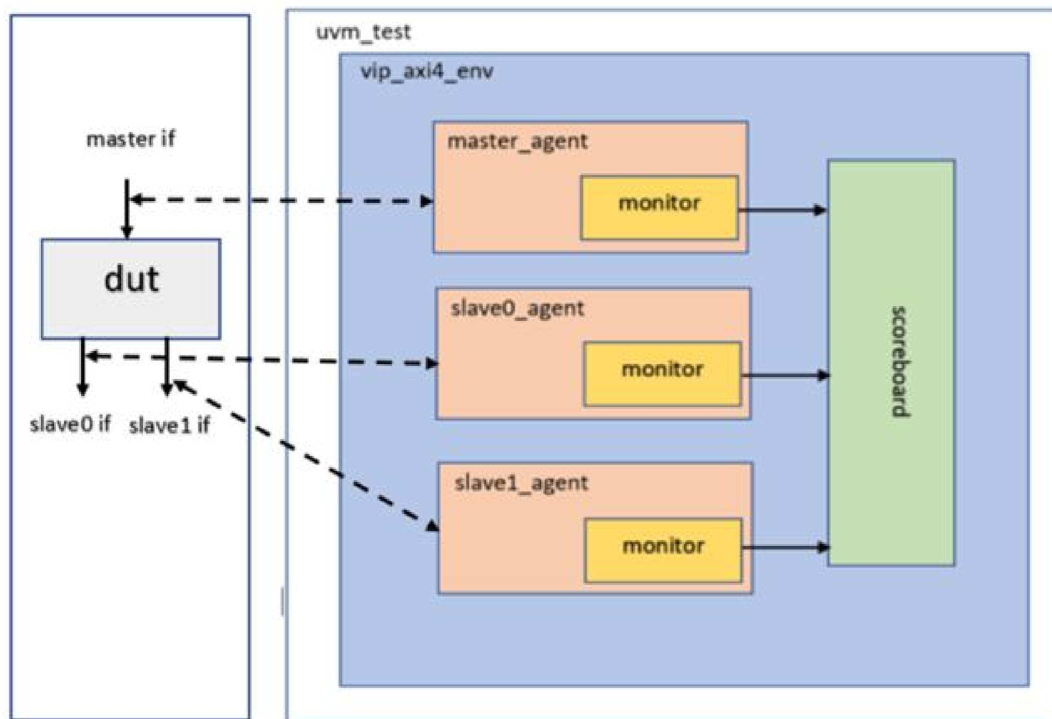

Figure 5.1: Block diagram of testbench

5.1.2 Performance Counters

AXI UPSIZER DOWNSIZER implements performance counters which can be accessed via APB interface. There are totally eight 64 bits registers which constitutes to sixteen 32-bit APB registers.

- Read Request Stall Counter Lower : Shows the number of cycles where Slave 1 accepted the read request but Slave 2 didnt. Lower 32 bits of the counter

- Read Request Stall Counter Upper : Shows the number of cycles where Slave 1 accepted the read request but Slave 2 didnt. Upper 32 bits of the counter.
- Write Request Stall Counter Lower : Shows the number of cycles where Slave 1 accepted the write request but Slave 2 didnt. Lower 32 bits of the counter.
- Write Request Stall Counter Upper : Shows the number of cycles where Slave 1 accepted the write request but Slave 2 didnt. Upper 32 bits of the counter.
- Read Response Stall Counter Lower : Shows the number of cycles for which we have RLAST on Slave 1 but not for Slave 2 for every transaction. Lower 32 bits of the counter.
- Read Response Stall Counter Upper : Shows the number of cycles for which we have RLAST on Slave 1 but not for Slave 2 for every transaction. Upper 32 bits of the counter.
- Write Response Stall Counter Lower : Shows the number of cycles for which we have BRESP on Slave 1 but not for Slave 2 for every transaction. Lower 32 bits of the counter.
- Write Response Stall Counter Upper : Shows the number of cycles for which we have BRESP on Slave 1 but not for Slave 2 for every transaction. Upper 32 bits of the counter.
- Read Request Number Counter Lower : shows the total number of Read Requests passed via AXI-UPSIZER DOWNSIZER. Lower 32 bits of the counter.
- Read Request Number Counter Upper : Shows the total number of Read Requests passed via AXI-UPSIZER DOWNSIZER. Upper 32 bits of the counter.
- Write Request Number Counter Lower : Shows the total number of Write Requests passed via AXI-UPSIZER DOWNSIZER. Lower 32 bits of the counter.

- Write Request Number Counter Upper : Shows the total number of Write Requests passed via AXI-UPSIZER DOWNSIZER. Upper 32 bits of the counter.
- Read Beat Number Counter Lower : Shows the total number of Read beats passed via AXI-UPSIZER DOWNSIZER. Lower 32 bits of the counter.
- Read Beat Number Counter Upper : Shows the total number of Read beats passed via AXI-UPSIZER DOWNSIZER. Upper 32 bits of the counter.
- Write Beat Number Counter Lower : Shows the total number of Write data beats passed via AXI-UPSIZER DOWNSIZER. Lower 32 bits of the counter.
- Write Beat Number Counter Lower : Shows the total number of Write data beats passed via UPSIZER DOWNSIZER. Upper 32 bits of the counter.

5.1.3 Scoreboard Implementation

The testbench utilizes a UVM scoreboard to predict the DUT functionality and match with the DUT behavior.

Before fifo read This fifo captures each beat of a read transactions at the master interface side as an entry. Whenever the master agent receives a beat it pushes the packet to the fifo i.e. request along with the beat response. Before fifo write This fifo captures a complete write transaction at thte master interface side as an entry. Whenever the master agent receives a complete write transaction i.e. request along with the response, it creates a packet with all of the signal values in it and pushes to the fifo. After fifo read slave 0 - Same as Before fifo read, but used to capture the packet on the Slave data interface. After fifo write slave 0 - Same as Before fifo write, but used to capture the packet on the Slave data interface. After fifo read slave 1 - Same as Before fifo read, but used to capture the packet on the Slave Timing interface. After fifo write slave 1 - Same as Before fifo write, but used to capture the packet on the Slave Timing interface.

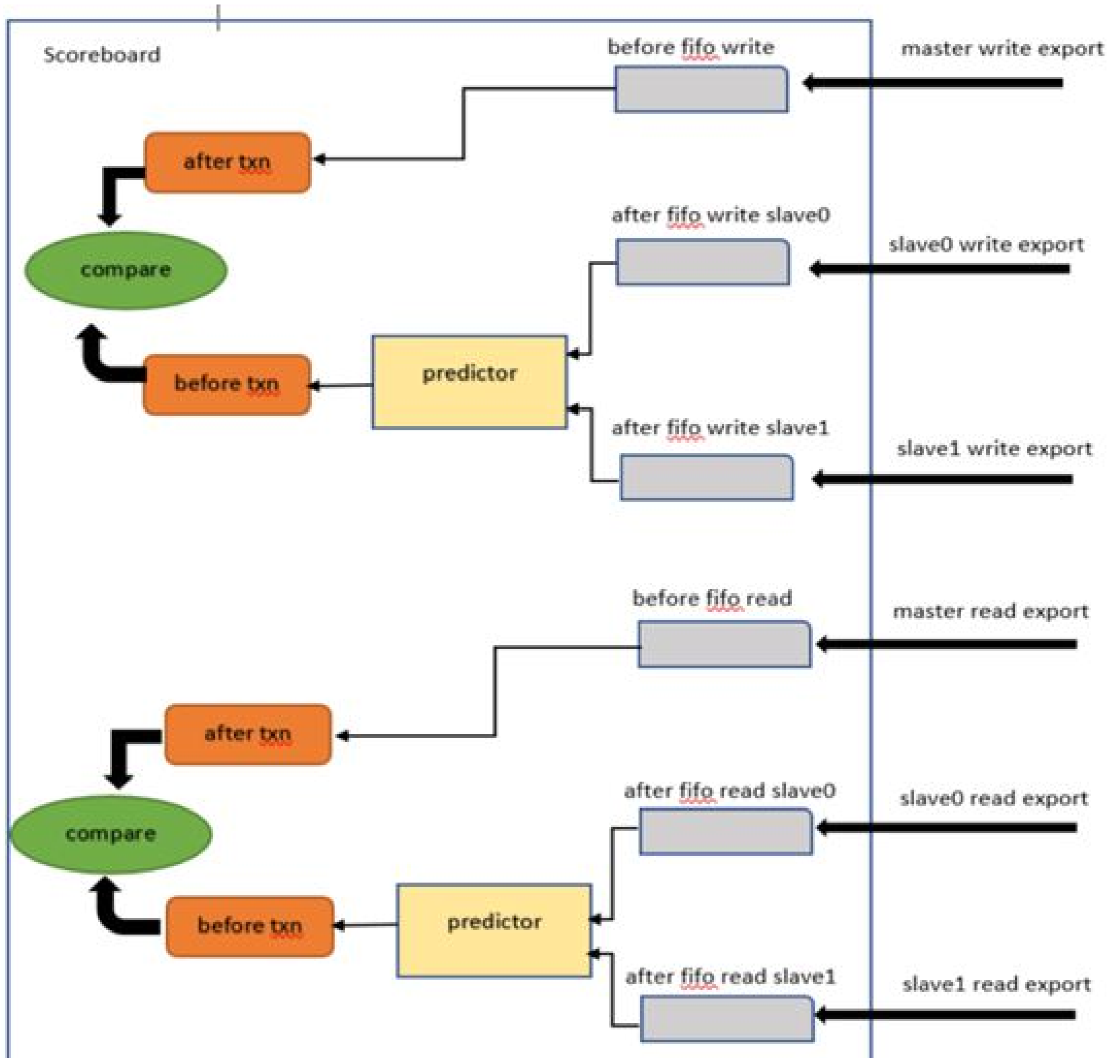

Figure 5.2: Scoreboard design

- Scoreboard checking We take the latest packet in the before read/write fifoi.e after txn and compare it with the before txn which comes after the predictor logic. Here predictor does nothing but the forwarding of transaction/beat packet of slave 0 fifo. So for example, if slave 0 have responded, it will have a transaction/beat packet in the slave0 fifo. Now, if the splitter forwards this reponse to master port without looking for slave1 response, checking will fail with an error stating slave 1 fifo doesnt have the response yet and therefore master shouldnt have forwarded the response.

- AXI4 protocol checking AXI4 protocol checkers have been instantiated at the all three AXI interfaces of DUT.

5.1.4 UVM Testbench parameters

```
=====================================================================================
                 tbench.mst_if AXI4 Interface configuration :
=====================================================================================
The interface has been configured with the following parameters:

addr_width = 40 bits, data_bus_width = 32 bytes, wid_width = 12 bits, rid_width = 12 bits
awuser_width = 0 bits, aruser_width = 0 bits
wuser_width = 0 bits, ruser_width = 0 bits, buser_width = 0 bits
protocol_level = AMBA_4
atomic_transactions = 0
dvm_v8_1 = 0
cache_stash_transactions = 0
deallocation_transactions = 0
trace_signals = 0
poison = 0
data_check = 0
untranslated_transactions = 0
port_type_accelerator= 0
port_type_peripheral= 0
clean_shared_persist = 0
loopback_signals = 0
low_power_signals = 0
nsaccess_identifiers = 0
zero_invalid_byte_lanes = 0
```

Figure 5.3: Master Configuration parameters

```
# ========================================================================================
#                   tbench.slv0_if AXI4 Interface configuration :
# ========================================================================================
# The interface has been configured with the following parameters:
#
# addr_width = 40 bits, data_bus_width = 16 bytes, wid_width = 12 bits, rid_width = 12 bits
# awuser_width = 0 bits, aruser_width = 0 bits
# wuser_width = 0 bits, ruser_width = 0 bits, buser_width = 0 bits
# protocol_level = AMBA_4
# atomic_transactions = 0
# dvm_v8_1 = 0
# cache_stash_transactions = 0
# deallocation_transactions = 0
# trace_signals = 0
# poison = 0
# data_check = 0
# untranslated_transactions = 0
# port_type_accelerator= 0
# port_type_peripheral= 0
# clean_shared_persist = 0
# loopback_signals = 0
# low_power_signals = 0
# nsaccess_identifiers = 0
# zero_invalid_byte_lanes = 0
# ========================================================================================
```

Figure 5.4: Slave 1 Configuration parameters

```
# =========================================================================================
#                    tbench.slv1_if AXI4 Interface configuration :
# =========================================================================================
# The interface has been configured with the following parameters:
#
# addr_width = 40 bits, data_bus_width = 16 bytes, wid_width = 12 bits, rid_width = 12 bits
# awuser_width = 0 bits, aruser_width = 0 bits
# wuser_width = 0 bits, ruser_width = 0 bits, buser_width = 0 bits
# protocol_level = AMBA_4
# atomic_transactions = 0
# dvm_v8_1 = 0
# cache_stash_transactions = 0
# deallocation_transactions = 0
# trace_signals = 0
# poison = 0
# data_check = 0
# untranslated_transactions = 0
# port_type_accelerator= 0
# port_type_peripheral= 0
# clean_shared_persist = 0
# loopback_signals = 0
# low_power_signals = 0
# nsaccess_identifiers = 0
# zero_invalid_byte_lanes = 0
```

Figure 5.5: Slave 2 Configuration parameters

5.2 Results

Figure 5.6 shows the read transaction received of master. Figure 5.7 and 5.8 shows generated the read transaction of slave 1 and slave 2 respectively. Figure 5.9 is the final result of basic test.

```
# ------------------------------------------------------------------------------------------------------------------------------
# Name                Type                              Size  Value
# ------------------------------------------------------------------------------------------------------------------------------
# read transaction    vip_axi4_monitor_read_transaction -     @8736
#   protocol_level    vip_amba4_types::protocol_level_t 32    AMBA_4
#   trans_type        vip_amba4_types::trans_t          32    VIP_AMBA4_RESPONSE_TRANSACTION_GROUP
#   operation_type    vip_amba4_types::trans_sub_t      32    VIP_AMBA4_READ
#   ARID              integral                          32    'hc2c
#   ARADDR            integral                          64    'hd1bbbb2ee4
#   ARLEN             integral                          8     'h7
#   ARSIZE            integral                          3     'h0
#   ARBURST           integral                          3     'h1
#   ARLOCK            integral                          2     'h0
#   ARCACHE           integral                          4     'h6
#   ARPROT            integral                          3     'h3
#   ARQOS             integral                          4     'hb
#   ARREGION          integral                          4     'h3
#   ARUSER            integral                          192   'h0
#   arvalid_time      integral                          64    'd1485
#   arready_time      integral                          64    'd1455
#   arvalid_clock     integral                          64    'd148
#   arready_clock     integral                          64    'd145
#   Read Data Beat[0] array                             8     -
#     RDATA           integral                          256   'hc98a4edc4d9e05689bbf9f229000e55ac10bf6c56f218ef836c67a83bfb8ec0c
#     RRESP           integral                          2     'h0
#     RUSER           integral                          192   'h0
#     rvalid_time     integral                          64    'd1765
#     rready_time     integral                          64    'd1715
#     rvalid_clock    integral                          64    'd176
#     rready_clock    integral                          64    'd171
```

Figure 5.6: Result of Master transaction

```
# -----------------------------------------------------------------------------------------------------
# Name                    Type                                Size  Value
# -----------------------------------------------------------------------------------------------------
# read transaction        vip_axi4_monitor_read_transaction   -     @8754
#   protocol_level        vip_amba4_types::protocol_level_t   32    AMBA_4
#   trans_type            vip_amba4_types::trans_t            32    VIP_AMBA4_RESPONSE_TRANSACTION_GROUP
#   operation_type        vip_amba4_types::trans_sub_t        32    VIP_AMBA4_READ
#   ARID                  integral                            32    'hc2c
#   ARADDR                integral                            64    'hd1bbbb2ee4
#   ARLEN                 integral                            8     'h7
#   ARSIZE                integral                            3     'h0
#   ARBURST               integral                            3     'h1
#   ARLOCK                integral                            2     'h0
#   ARCACHE               integral                            4     'h6
#   ARPROT                integral                            3     'h3
#   ARQOS                 integral                            4     'hb
#   ARREGION              integral                            4     'h3
#   ARUSER                integral                            192   'h0
#   arvalid_time          integral                            64    'd1495
#   arready_time          integral                            64    'd1465
#   arvalid_clock         integral                            64    'd149
#   arready_clock         integral                            64    'd146
#   Read Data Beat[0]     array                               8     -
#     RDATA               integral                            128   'hc98a4edc4d9e05689bbf9f229000e55a
#     RRESP               integral                            2     'h2
#     RUSER               integral                            192   'h0
#     rvalid_time         integral                            64    'd1735
#     rready_time         integral                            64    'd1685
#     rvalid_clock        integral                            64    'd173
#     rready_clock        integral                            64    'd168
```

Figure 5.7: Result of Slave 1 transaction

```
# ---------------------------------------------------------------------------------------------------
# Name                  Type                               Size  Value
# ---------------------------------------------------------------------------------------------------
# read transaction      vip_axi4_monitor_read_transaction  -     @8760
#   protocol_level      vip_amba4_types::protocol_level_t  32    AMBA_4
#   trans_type          vip_amba4_types::trans_t           32    VIP_AMBA4_RESPONSE_TRANSACTION_GROUP
#   operation_type      vip_amba4_types::trans_sub_t       32    VIP_AMBA4_READ
#   ARID                integral                           32    'hc2c
#   ARADDR              integral                           64    'hd1bbbb2ee4
#   ARLEN               integral                           8     'h7
#   ARSIZE              integral                           3     'h0
#   ARBURST             integral                           3     'h1
#   ARLOCK              integral                           2     'h0
#   ARCACHE             integral                           4     'h6
#   ARPROT              integral                           3     'h3
#   ARQOS               integral                           4     'hb
#   ARREGION            integral                           4     'h3
#   ARUSER              integral                           192   'h0
#   arvalid_time        integral                           64    'd1495
#   arready_time        integral                           64    'd1465
#   arvalid_clock       integral                           64    'd149
#   arready_clock       integral                           64    'd146
#   Read Data Beat[0]   array                              8     -
#     RDATA             integral                           128   'hc10bf6c56f218ef836c67a83bfb8ec0c
#     RRESP             integral                           2     'h2
#     RUSER             integral                           192   'h0
#     rvalid_time       integral                           64    'd1575
#     rready_time       integral                           64    'd1575
#     rvalid_clock      integral                           64    'd157
#     rready_clock      integral                           64    'd157
```

Figure 5.8: Result of Slave 2 transaction

```
# ** Report counts by id
# [BASIC_TEST]    38
# [MST_RD_SEQ]  1005
# [MST_WR_SEQ]  1005
# [Questa UVM]     2
# [REG_SEQ]   287
# [REG_WR_SEQ]    32
# [RNTST]     1
# [TEST_DONE]     9
# [TIMOUTSET]     1
# [reg_write_seq]     1
# [scrbrd]  7281
# [tbench.mst_if]     1
# [tbench.slv0_if]     1
# [tbench.slv1_if]     1
# [uvm_driver #(REQ,RSP)]     2
# [uvm_monitor]     2
# [uvm_test_top.env.VIP_AXI4_MASTER.DRIVER]     2
# [uvm_test_top.env.VIP_AXI4_MASTER.MONITOR]     3
# [uvm_test_top.env.VIP_AXI4_SLAVE0.DRIVER]     2
# [uvm_test_top.env.VIP_AXI4_SLAVE0.MONITOR]     3
# [uvm_test_top.env.VIP_AXI4_SLAVE1.DRIVER]     2
# [uvm_test_top.env.VIP_AXI4_SLAVE1.MONITOR]     3
# [vip_apb4_env]     1
# [vip_axi4_env]     1
# ** Note: $finish    : /arm/tools/mentor/questasim/10.4d_2/questasim/linux_x86_64/
#    Time: 1831005 ps  Iteration: 59  Instance: /tbench
# Executing Axi4 End Of Simulation checks
# Executing Axi4 End Of Simulation checks
# Executing Axi4 End Of Simulation checks
# End time: 08:54:51 on Mar 25,2019, Elapsed time: 0:01:51
# Errors: 0, Warnings: 457
```

Figure 5.9: Result of Basic test

Chapter 6

Conclusion

6.1 Conclusion

The goal was to expand data width of interconnect from 128bit to 256bit in the subsystem.For acheiving this, AXI UPSIZER DOWNSIZER were implemented using verilog and UVM. This design is tested using arm test bench environment. The test results shows the read and write transaction for 256bits data. The protocol checker verified all the successful read and write transactions. The AXI UPSIZER DOWNSIZER module can be integrated in the subsystem which has AXI protocol for communication between the components.

6.2 Future work

In future we will continue to implement this module for snoop transaction as per AXI coherency extension (ACE) protocol. Once we test that module on unit level environment, we can integrate it in our subsystem. Once it gets integrated, the subsystem will run some payload with more data on the data bus and see the performance parameters has improved or not. The whole project will get enhanced based on the performance improvement.

References

[1] “AMBA AXI Protocol Specification”, ARM Limited,2017

[2] “AMBA ACE Protocol Specification”, ARM Limited,2017

[3] “Computer Architecture: A Quantitative Approach”, by John L. Hennessy and David A. Patterson (5th edition)

[4] ARM internal documents

[5] “Verilog HDL A guide to Digital Design and Synthesis”, by Samir Palnitkar (1st edition)