FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

BACHELOR'S THESIS

MULTI-WRITE DISTRIBUTOR BUS

Author Joonas Ojanen

Supervisor Jukka Lahti

December 2022

Ojanen J. (2022) Multi-Write Distributor Bus. University of Oulu, Degree Programme in Electronics and Communications Engineering. Bachelor's Thesis, 28 s.

ABSTRACT

This thesis studies the design and verification for Multi-Write distribution with AMBA AXI protocol to reduce the SW execution time. First, Bus protocol and microarchitecture is looked at and Verilog/VHDL coding processes are studied. And appropriate verification method is examined. Last challenges/open items and possibilities for the future improvements are discussed. The development of standalone design/verification environment build is covered in the work.

Key words: AMBA, Telecommunications, VHDL, Verification, System-on-chip, Verilog.

Ojanen J. (2022) Moni-Kirjoitus Jakeluväylä. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 28 s.

TIIVISTELMÄ

Tässä opinnäytetyössä tutkitaan monikirjoitusjakelijan suunnittelua ja verifiointia AMBA AXI protokollalla ohjelmiston suoritusajan lyhentämiseksi. Ensin tarkastellaan väyläprotokollaa ja mikroarkkitehtuuria ja tutkitaan Verilog/VHDL koodausprosesseja sekä asianmukaisia verifikaatio menetelmiä. Viimeiseksi keskustellaan haasteista/avoimista kohteista ja mahdollisista tulevaisuuden parannuksista. Työssä käsitellään erillisen suunnittelu-/todentamisympäristön rakentamisen kehittämistä.

Avainsanat: AMBA, Tietoliikennetekniikka, VHDL, Verifikaatio, System-on-Chip, Verilog.

TABLE OF CONTENTS

FOREWORD

The purpose of this bachelor's thesis was to make Multi-Write distributor bus system and it was made at Nokia.

I want to thank Jukka Lahti for supervising this bachelor's thesis. I also want to thank Makoto Takami and my whole team for helping me throughout the work. And I also want to thank Harri Nissi and Sami Leskelä for offering me this opportunity.

Oulu 8.12.2022

Joonas Ojanen

EXPLANATIONS OF ABBREVIATIONS AND SYMBOLS

AXI	Advanced eXtensible Interface
HW	Hardware
AMBA	Advanced Microcontroller Bus Architecture
SoC	System on chip
IP	Intellectual Property
BTS	Base Transceiver Station
WAN	Wide Area Network
MIMO	Multiple-Input-Multiple-Output
SW	Software
AI	Artificial Intelligence
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
UART	Universal Asynchronous Receiver-Transmitter
SPI	Serial Peripheral Interface
AR	Read Address Channel
R	Read Data Channel
AW	Write Address Channel
W	Write Data Channel
B	Write response Channel
SI	Subordinate Parameters
MI	Manager Parameters
RTL	Register-Transfer level
VHDL	Hardware Description Language
DUT	Design Under Test
UVM	Universal Verification Methodology
CISC	Complex Instruction Set Computer

1 INTRODUCTION

Nowadays the system-on-chip (SoC) development has become more complex for the multifaceted user requirement on hardware (HW) and software (SW) development has become much more complex than HW development for the longer lifecycle of SW products on SoC, hence the HW design needs to be more SW friendly. Therefore, the HW needs to be designed to support the SW requirements in advance. This is also affecting the design costs increasingly, as shown in the Figure 1 the SW cost is almost twice the amount compared to other costs [5].

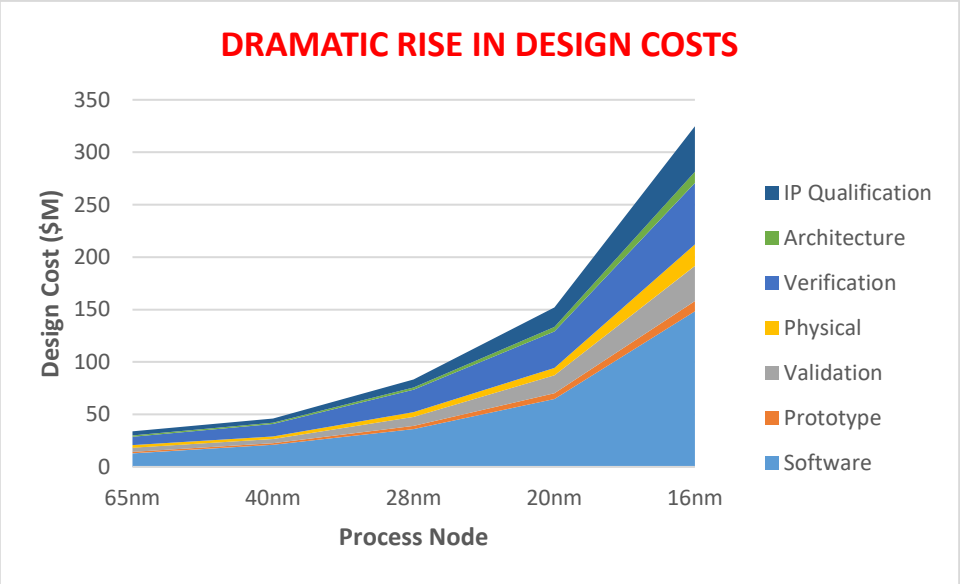

Figure 1. Design cost diagram [5].

In this thesis made for Nokia the purpose was to create Multi-Write Distributor Bus that duplicates the same write transaction through Advanced eXtensible Interface (AXI) system bus to different HW components using the ARM Advanced Microcontroller Bus Architecture (AMBA) bus to satisfy the SW requirements. Multicasting has been studied before using algorithms, but these solutions are often very complex and hard to implement, and they cause latencies due to the complexity of the code and architecture.

This introduction chapter goes through first market status of the ARM and AMBA, after that the base transceiver station (BTS), third some background of the project, and the last section introduces prior research of the subject.

Master and slave are referred as manager and subordinate in this thesis.

1.1 Market status

In the SoC industry ARM is the leading technology provider of processor intellectual property (IP). ARM is widely used in all aspect of electronics. They provide wide range of processor IPs to meet the power, performance, and cost requirements of different devices. [1]

AMBA protocol is an open standard, on-chip interconnect specification, which connects and manages the functional blocks in a SoC [2]. AMBA protocol can be used in any processor architecture, and it is freely available. Like ARM processor IPs, the AMBA is widely used in semiconductor industry, due to a fact that it is well supported, it allows compatibility between IP components from different vendors, it is flexible supporting wide range of SoCs, and it is cost efficient being free, so it lowers SoC development costs.

1.2 Base Transceiver Station

A BTS is the facility to make the wireless connection among the radio communication device and the network like wide area network (WAN). There are multiple transceiver channels to support Multiple-Input-Multiple-Output (MIMO) in one BTS to cover multiple careers. BTS system is rather homogeneous architecture to control the plural FrontEnd devices and antennas from single processing unit. Hence the configuration among the FrontEnd devices and antennas has the similarity on each configuration. Figure 2 shows the example BTS system which makes the parallel TX/RX paths with the same components.

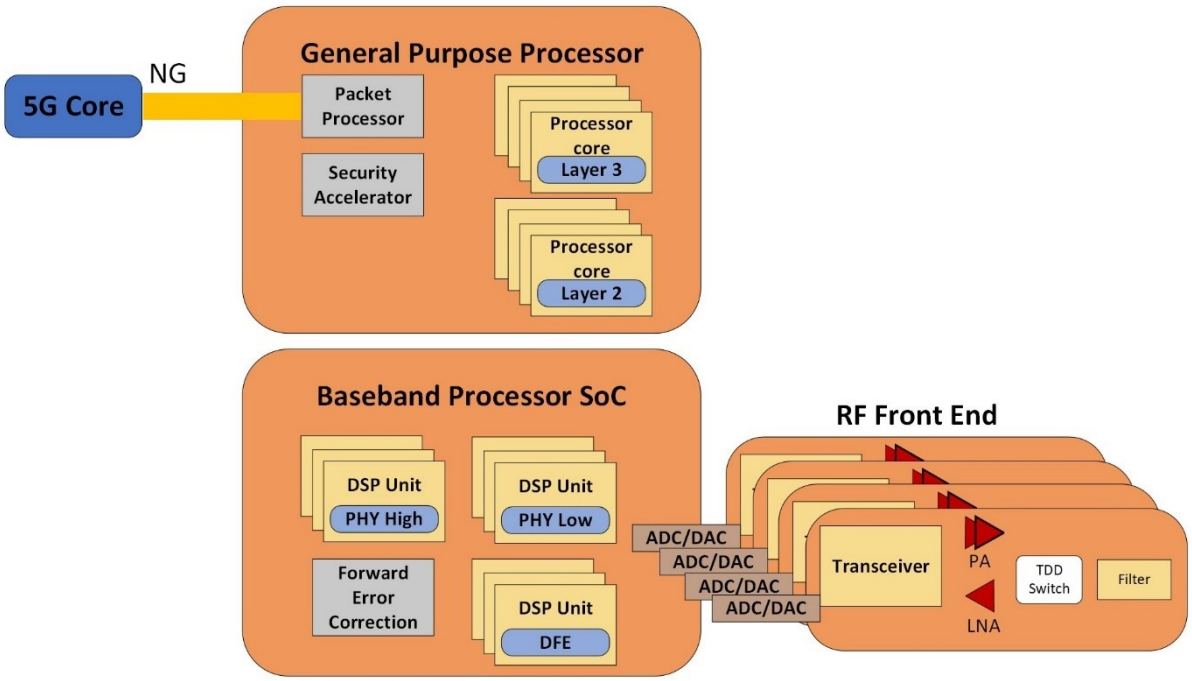

Figure 2. 5G New Radio BTS system example [12].

1.3 Background

The purpose of this thesis is to solve that the SW faces long durations to initialize or setup the same HW components in the different locations [10]. Currently same configuration is done in all 4 blocks for Uplink and Downlink and this needs to be avoided. To avoid this problem multicast is created.

1.4 Prior research

Multicast function has not been studied that much before in AMBA system because the principle of AMBA bus is one-to-one connection between one bus manager as initiator and one bus subordinate as target. There are couple of prior works that can be used as comparison or reference for the thesis. In this chapter Arteris artificial intelligence (AI) package as consumer level and multicast with 2d-mesh system are introduced shortly.

Arteris AI package uses intelligent multicast that provides optimized usage of on-chip and off-chip bandwidths. This way more efficient updates of the multicast data can be done. [3]

2d-mesh system is algorithm-based multicast. This multicasting option divides the network into a set of subnetworks. Each subnetwork is controlled individually with router. The data packet is routed using the algorithm through the routers from the input port to the output port in one clock cycle. [4]

2 MICROARCHITECTURE

This chapter goes through generally the microarchitecture and AXI protocol. First part is about the microarchitectural concept of bus design, second part is about the AXI protocol and more specifically about AXI4 and AXI4-Lite protocols.

2.1 Microarchitectural concepts

Microarchitecture is a set of digital logic, which combines implementation of memory, registers, arithmetic logic units, multiplexers, or any other digital logic blocks what there are. Earlier mentioned ARM has for an example many different microarchitectures, each with different cost, performance, and complexity. One of these microarchitectures is bus which affects the system performance.

General bus system has interface, address decoder/router and arbiter. The transaction goes through interface from bus manager with address to the destination, the decoder switches the access path based on the routing information and gets to the subordinate interface after arbitration when competing with the other transaction. The subset of the signals varies depending on the bus protocol.

2.2 AXI Advanced microcontroller Bus architecture

AXI protocol is widely used among SoC designers today and it is a part of the AMBA specification. The AXI is point to point protocol between manager and subordinate interface [8]. The Figure 3 shows five main channels of AXI interface. In this chapter two sub chapters about AXI4 and AXI4-Lite are looked through.

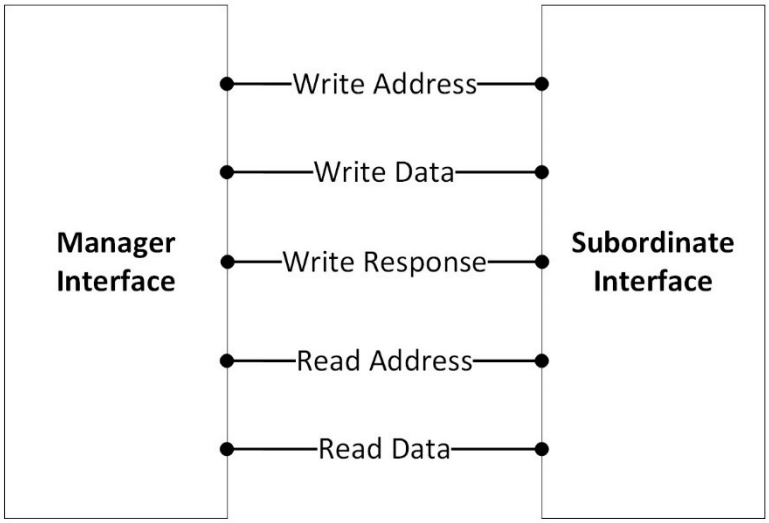

Figure 3. AXI Interface [8].

2.2.1 AXI4

AXI4 is the widely used protocol and it was made for low latency and high bandwidth applications. Purpose of the AXI4 protocol is to allow communication between the manager

typically CPU or direct memory access (DMA) and subordinate typically dynamic random-access memory (DRAM), universal asynchronous receiver-transmitter (UART) or serial peripheral interface (SPI). AXI4 interface signals are shown in Table 1. [9]

Table 1. AXI4 interface signals [9].

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	BID	ARPROT	RRESP
-	AWID	WUSER	BUSER	ARID	RID
-	AWLEN	WLAST	-	ARLEN	RLAST
-	AWSIZE	-	-	ARSIZE	RUSER
-	AWBURST	-	-	ARBURST	-
-	AWLOCK	-	-	ARLOCK	-
-	AWCACHE	-	-	ARCACHE	-
-	AWREGION	-	-	ARREGION	-
-	AWUSER	-	-	ARUSER	-
-	AWQOS	-	-	ARQOS	-

All transaction channels shown in the Figure 3 use the same xVALID/xREADY pair in the handshake process. This is a two-way mechanism so both subordinate and manager control the process, subordinate generates the xVALID signal to indicate when the information, data or address is available, and manager generates the xREADY signal to inform that it can accept the information [9]. Transfer occurs only when xVALID and xREADY signals are both HIGH [9]. Figure 4 shows example of handshake.

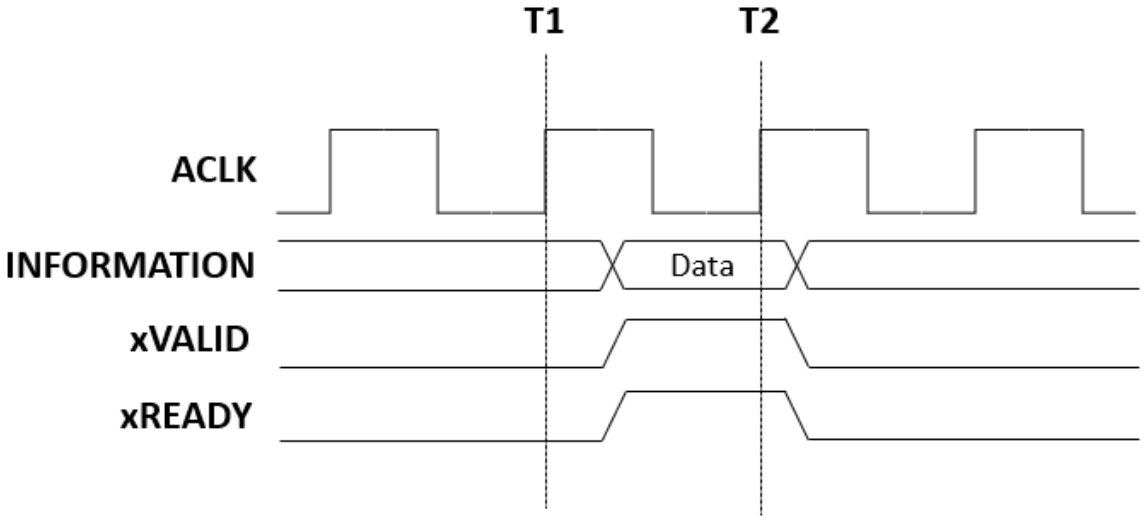

Figure 4. xVALID/xREADY handshake [9].

AXI4 read request happens on channels read address (AR) and read data (R). The handshake is performed in both channels. AR channel is used by the manager to send read request to

subordinate. First manager sets AR channel signals like ARLEN, ARSIZE and ARADDR, signals are shown in the Table 1 then ARVALID is set to high signaling that new transaction started. After that subordinate asserts ARREADY signaling to manager that the new transaction has been accepted and after cycle R channel signals like RRESP, RDATA and RID are set, signals are shown in Table 1 then RVALID and RREADY are set to high indication completion status of the read transaction. [9]

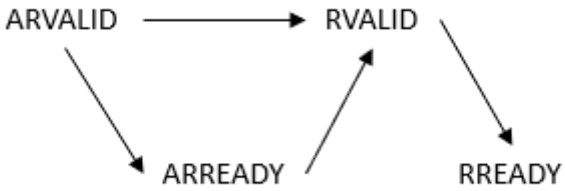

Figure 5. Read request [9].

AXI4 write request happens on channels write address (AW), write data (W) and write response (B). The handshake is performed in all three channels. AW channels is used by manager to send write request to subordinate. First manager sets AW channel signals like AWLEN, AWSIZE and AWADDR, signals are shown in the Table 1 then AWVALID is set to high. At the same time manager also sets W channel signals like WLAST, WDATA and WSTRB, signals are shown in the Table 1 and sets WVALID high. Subordinate raises AWREADY when it is ready with data on AW channel and WREADY is asserted signaling manager that write data is accepted. And last subordinate sets B channel signals like BRESP and BID, signals are shown in Table 1 and then BVALID is asserted signaling manager that response to write is done, when manager is ready it asserts BREADY signaling that transaction is completed. [9]

AXI4 also supports outstanding write, it is same as the single write, but the manager doesn't wait response from B channel.

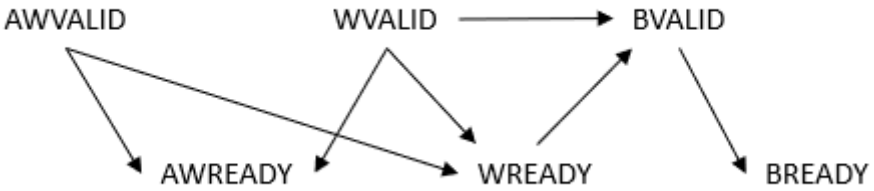

Figure 6. Write request [9].

AXI4 also supports burst write and read request, it is same as the single write or read but multiple writes or reads can be conducted with one AW or AR handshake. Burst control information sent at the beginning of a transaction is controlled with AXLEN, AXSIZE, AXBURST signals. There are three different burst types FIXED, INCR and WRAP [9].

In Figure 7 FIXED burst access is shown, first there is AW handshake, after that there is four W handshakes and when the fourth W handshake occurs WLAST is also asserted to signal that it is the last write transfer in the burst. And last there is B handshake to signal that the transaction is completed.

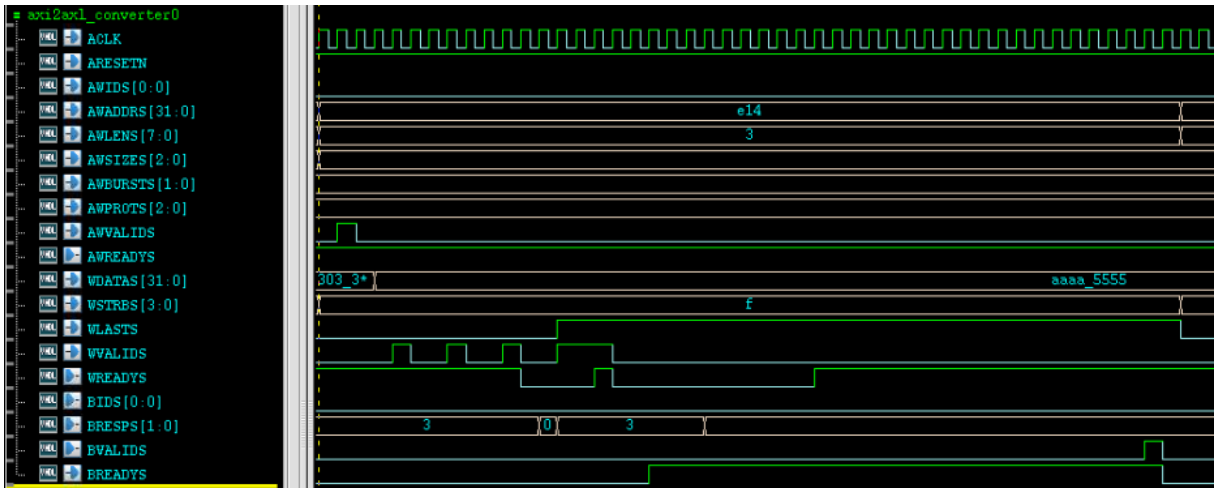

Figure 7. FIXED Burst access waveform.

2.2.2 AXI4-Lite

AXI4-Lite is subset of the AXI4 protocol with simpler and smaller interface. The AXI4-Lite interface signals are shown it the Table 2 AXI4-Lite subordinate cannot be connected to the AXI4 manager because it doesn't support burst access and transaction ID, specific protocol converter which supports AXI4 to AXI4-Lite conversion is needed for this task. However, AXI4-Lite manager can be connected to the AXI4 subordinate connecting the non-used signals with their default values. [9]

Table 2. AXI4-Lite interface signals [9].

Global	**Write address channel**	**Write data channel**	**Write response channel**	**Read address channel**	**Read data channel**
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

3 MULTI-WRITE DISTRIBUTOR BUS

This chapter goes first through the thesis project implementation using the previous chapters theory and research and last development summary.

This chapter can also be used as documentation for the environment.

3.1 Implementation

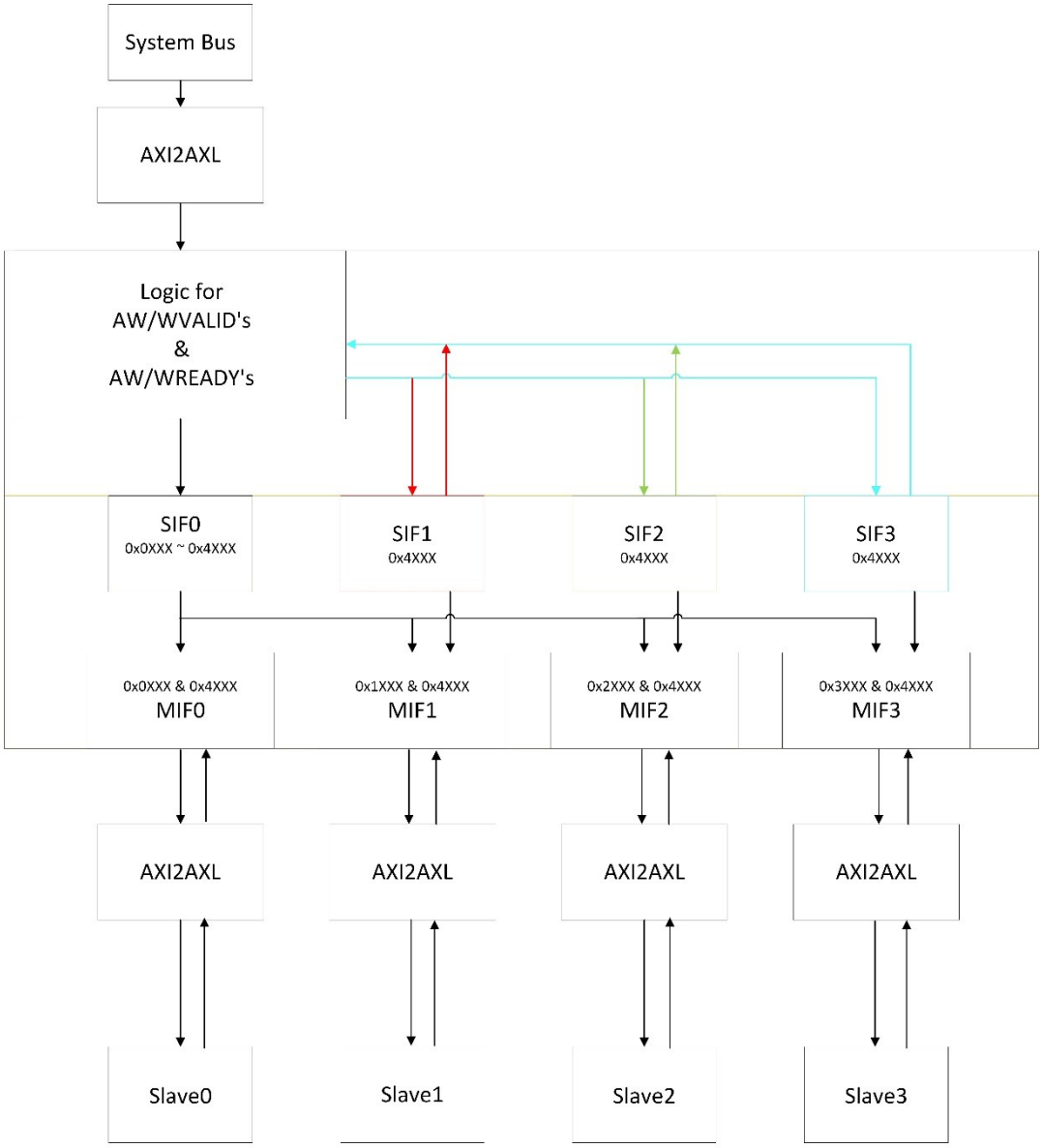

Figure 8. Multi-Write Distributor Bus.

Design consists of different components like ARM NIC-400 bus, AXI2AXL converter and memory model, there is also AW/WVALID and AW/WREADY control. These components and logic part form the Multi-Write distributor bus which is shown in Figure 8.

This chapter has four sub chapters that introduces ARM NIC-400 bus, implementation of the simulation environment, verification, and test sequences.

3.1.1 ARM NIC-400 bus

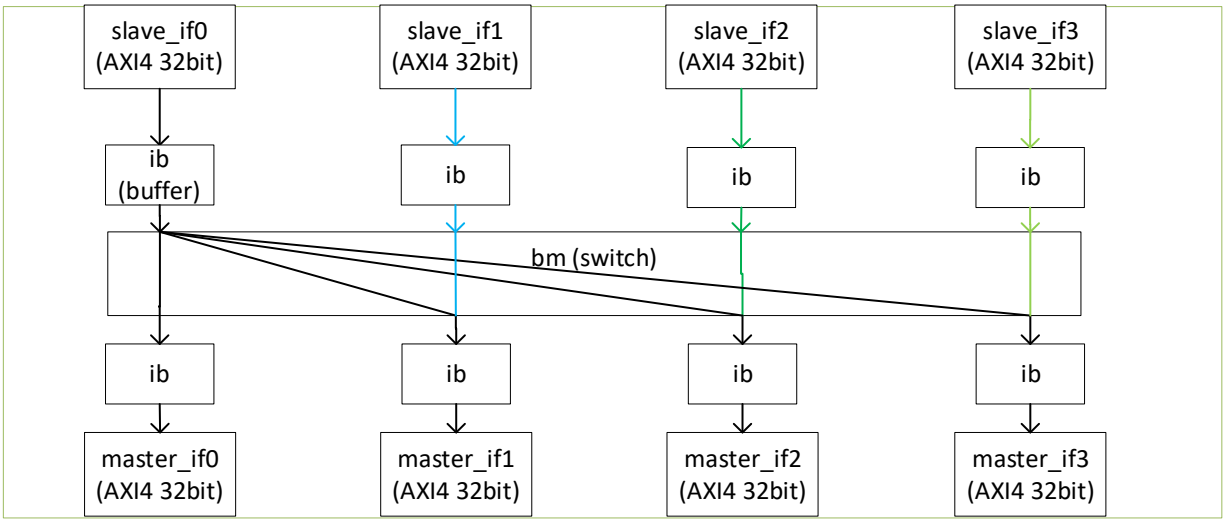

Figure 9. ARM NIC-400 bus.

ARM NIC-400 [12] is used for the routing and address decode in the Multi-Write distributor bus, ARM NIC-400 bus is shown in Figure 9. NIC-400 in this research is simple to have one switch function to include address decode and routing to cover four subordinate interfaces and four manager interfaces. Table 3 and Table 4 shows the routing and addressing on NIC-400 bus.

Table 3. Manager and Subordinate visibility on NIC-400 bus.

master_if / slave if	master_if0	master_if1	master_if2	master_if3
slave_if0	X	X	X	X
slave_if1	-	X	-	-
slave_if2	-	-	X	-
slave_if3	-	-	-	X

Table 4. Address map on NIC-400 bus.

Region	**Start Address**	**End Address**	**Size**	
master_if0	0x4000	0x4FFF	4	Kbytes
master_if1	0x3000	0x3FFF	4	Kbytes
master_if2	0x2000	0x2FFF	4	Kbytes
master_if3	0x1000	0x1FFF	4	Kbytes
master_if0	0x0000	0x0FFF	4	Kbytes

NIC-400 is generated by ARM Socrates [11] using Verilog hardware description language for the routing function of Multi-Write distributer bus. The detail parameters are shown in Appendix 1.

This kind of routing and addressing is needed because AXI is point-to-point protocol, so it connects one manager interface to one subordinate interface. And in this case one AXI manager interface is divided to four AXI subordinate interfaces. In other words, the AXI manager interface can only receive one set of signals back, therefore the interconnect is needed.

AW/W channel signals that are from Verilog Task are connected to slave_if0/if1/if2/if3, not including WALID/READY signals which are part of the logic part. B/AR/R channel signals are only connected from Verilog Task to slave_if0. And every master_if0/if1/if2/if3 interface has own port connections. The port connections are shown in Appendix 2.

3.1.2 Simulation environment

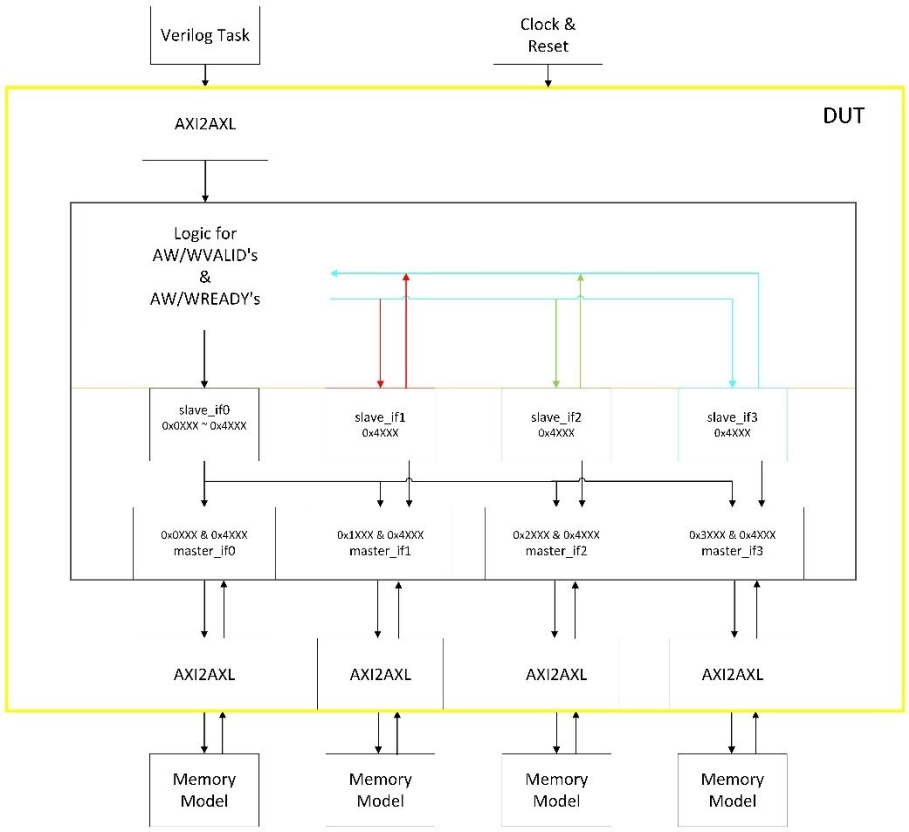

Figure 10. Simulation environment.

In this study, various options for implementing the simulation environment using the AXI protocol were initially considered. Simulation environment architecture shown in Figure 10 was chosen because the design is most forward and simplest.

The simulation environment contains design under test (DUT), Verilog Task block, Clock & Reset block and four Memory Models, these are shown in Figure 10. The DUT is top level module where the design which is being tested is instantiated. The DUT is controlled with Verilog Task block which contains different test cases that test different features of the design, the test cases are explained in the chapter 3.1.4, Clock & Reset block is also defined in the DUT. And last there are Memory Models which have only few registers to inspect the outputs of the DUT.

3.1.3 Verification

There are many ways to verify the design such as functional verification, formal verification, and universal verification methodology (UVM) [14]. Verification is done to ensure the correct functionality of the design. Therefore, it is used to check RTL code through various tests to ensure that it complies with the design specification. This is an important process because before the design is put into production it must work correctly and it also consumes lot of the time, so it is important part of the design flow.

In this work verification method was functional verification. Functional verification is commonly used verification method in which a design is verified using test cases like tasks and all the test cases are created for a specific purpose. In this work individual writes and reads, multi writes and reads, FIXED and INCR burst writes and reads, and outstanding writes were tested. In functional verification, only the data of inputs are fed into the DUT trough test cases and outputs of the DUT are examined to ensure correct functionality. The output data was examined inspecting waveforms and different printouts. For further development this verification method is not a comprehensive enough, as the test cases only test the specific tests, and the inputs are defined on a test-by-test basis. Better solution would be that the inputs are defined completely random which would be more comprehensive.

3.1.4 Test sequences

This chapter goes through generally what test cases were used in this work for testing different parts of the DUT. In this work there are five different test cases, axi_write, axi_read, axi_burst_write, axi_burst_read and axi_posted_write (posted is referred as outstanding in the text) that are implemented using tasks. Tasks (Appendix 3) are sections of a code that can be reused, and can have any number of inputs and outputs, so it is good for functional verification unlike normal functions that can only have one output. Automatic keyword can also be used with tasks, automatic means that simulation tool uses dynamic memory allocation. The functionally of the test cases are explained in the chapter 2.2.1.

3.2 Development summary

Time spent for studying the protocol, developing the Simulation environment and verification was about 300 hours. Especially new tools and protocols took time to learn, but most time-consuming part was the verification.

4 DISCUSSION

The purpose of this thesis was to create Multi-Write distributor bus that duplicates the same write transaction to AXI system bus to different HW components using the ARM AMBA bus. Multi-Write distributor bus was successfully integrated into a functional simulation environment and the environment was functionally verified. And future development variations and open items were discovered.

Most of the SoC support Cache system. It is very fast and expensive memory that stores frequently used data temporally. Data in cache can be accessed faster than the data stored in main memory, this saves time considerable whereas if the data needs to be fetched from the main memory the central processing unit (CPU) will be stalled. When accessing to repeatedly used data the use of cache provides huge speed boost as advantage [6]. This Multi-Write distributor bus cannot be applied in Cache operation since AXI4-Lite doesn't support AxCACHE signal to show the memory type. However, it is also difficult to apply if this Multi-Write distribute bus is designed with AXI4 because the Multi-Write function violates the cache coherency.

Pipelining, it is most used microarchitectural concept today. Main concept of the pipelining is that it can accept multiple instructions to overlap in the CPU making it retire the instructions faster [7]. Multi-Write distributer bus cannot be applied since CPU internal signal processing doesn't use AMBA protocol. However, multi-write distribute idea may help for the optimization of prefetch instruction, for example to store the same data on multiple CPU registers to detect the same register write instruction in advance. This may link to Complex Instruction Set Computer (CISC).

There is also limitation for Multi-Write because Multi-Read is not available for this bus system. In this bus system read is always done through SIF0 (Figure 8) and it can only get one read response back at once. This means that the SW needs to confirm that the data is correctly written after every end point. To allow Multi-Read one solution would be logic that could merge four different responses to one.

The functional verification was done using different test cases. The test cases tested the basic operations of the Multi-Write distributor bus, and they were run in different variations several times. All the test cases which were being tested worked correctly. So, it can be concluded that the environment works like it should. As a future open item more test cases are needed for better coverage, or the simulation environment should be verified using advanced verification methods like formal UVM [14] or formal verification [15].

In this thesis only one System Bus interface was assumed, so for future development plural System Bus interfaces can be considered. When plural System Bus interface is used bus configuration inside SoC is rather complicated to have plural managers, which needs to access to the same address space passing through the same interconnect. When plural manager interfaces come to this interconnect, the current design doesn't work since AXI protocol cannot be merged on SIF0~3. One solution for this issue is to provide loop back path after decoding with same design concept. This enables only multi distribute address space (0x4000~4FFF) to pick up on MIF0 after NIC-400 decode. This type of variation is shown in Figure 11. This is just one way to implement the plural manager interface, there are also other ways although not as simple and they may be more difficult to implement.

The Multi-Write distributor bus can also be further developed by making better subordinate memory because now it is only memory model with few registers, so more extensive verification would be possible. Or chancing simulation environment structure can be considered if it is noticed to be better.

In conclusion, the goals of this thesis were achieved and various possible ways to create the simulation environment were found and as well as what future developments could be.

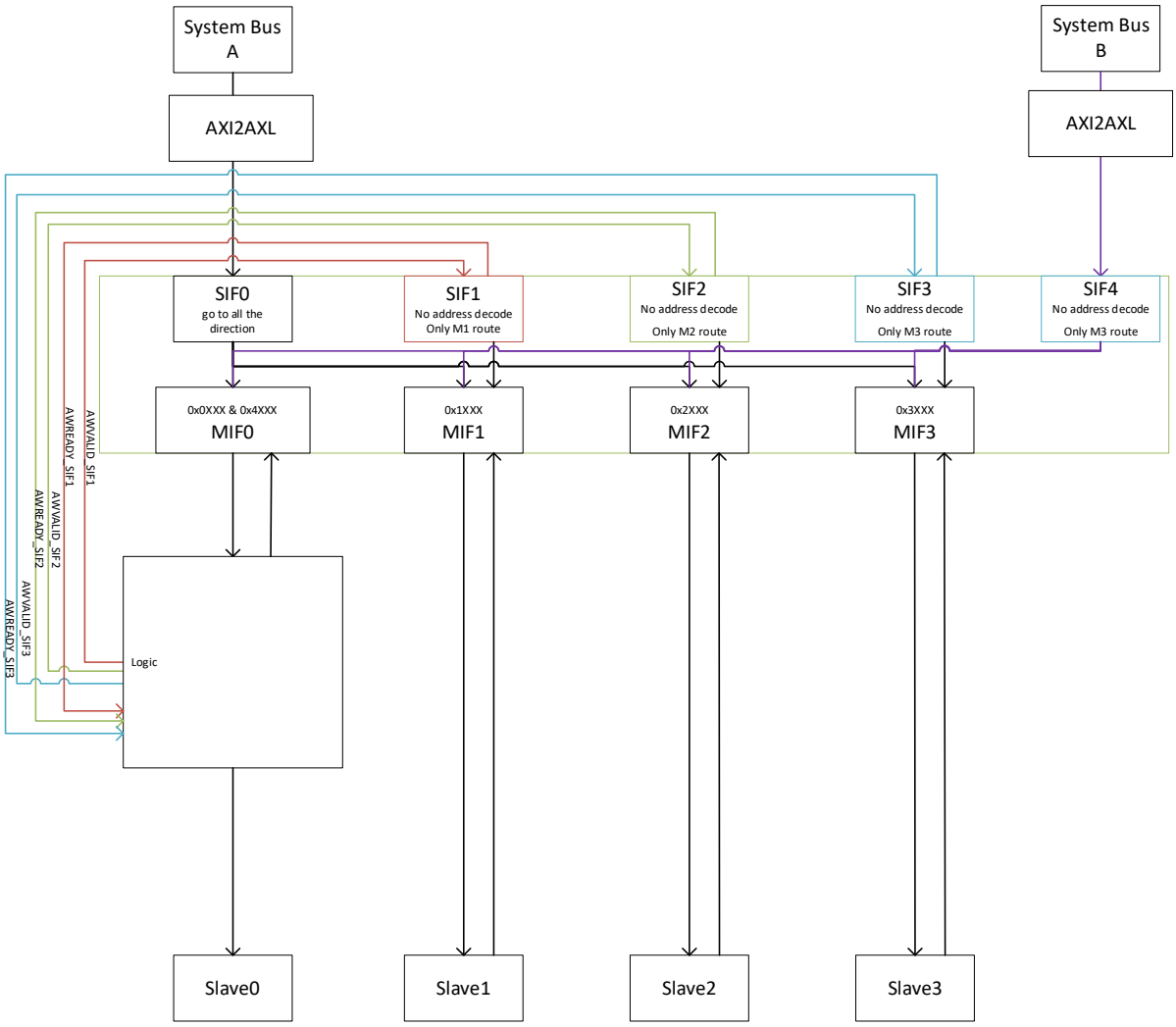

Figure 11. Alternative variation with plural System Bus interfaces.

5 SUMMARY

The purpose of this study was to create Multi-Write Distributor Bus that duplicates the same write transaction through AXI system bus to different HW components using the ARM AMBA bus to satisfy the SW requirements. First the background of the study and prior research were presented to the reader. Next microarchitecture and the protocols used in this study such as AXI4 and AXI4-Lite were introduced, so the reader could understand the theory behind the work. Simulation environment integration was introduced focusing on the ARM NIC-400 bus block how it was built and why it is needed, what the environment contains, how it has been verified and what test cases were used to help the verification. Finally future development variations and open items were discussed.

The study shows that the Multi-Write distributor bus should be used as it meets the SW requirements and leads to reduced durations to initialize or setup HW components.

6 REFERENCES

[1] ARM web page URL:
Defining the Future of Computing – Arm®.

[2] AMBA protocol URL:
AMBA AXI and ACE Protocol Specification Version E (arm.com)

[3] Arteris AI package URL:
Arteris IP FlexNoC AI Package Datasheet.pdf

[4] Multicast with 2d-mesh system URL:
(PDF) A New Network on Chip Design Dedicated to Multicast Service (researchgate.net)

[5] Exponential rising costs will bring changes URL:
Exponentially Rising Costs Will Bring Changes | Pete's Posts Blog (electroiq.com)

[6] Cache URL:
What is a cache? Easily explained! - IONOS

[7] Pipelining URL
(PDF) Correct-by-construction microarchitectural pipelining (researchgate.net)

[8] AMBA AXI protocol overview URL:
Learn the architecture - An introduction to AMBA AXI (arm.com)

[9] AMBA AXI4 and AXI4-Lite protocol specification URL:
AMBA AXI and ACE Protocol Specification Version H.c (arm.com)

[10] Reducing the boot time of embedded Linux systems URL:
nbnfioulu-201810062898.pdf

[11] ARM Socrates User Guide ver 1.5 URL:
https://documentation-service.arm.com/static/5f7dddf7d3be967f7be46dd3?token=

[12] ARM CoreLink NIC-400 Technical Reference Manual URL:
https://developer.arm.com/documentation/ddi0475/g

[13] The challenges of building a 5G base station URL:
https://www.analogictips.com/the-challenges-of-building-a-5g-base-station/

[14] UVM URL:
UVM Tutorial for Beginners (chipverify.com)

[15] Formal verification URL:
Understanding Formal Verification - AnySilicon

7 APPENDICES

Appendix 1 NIC-400 Configuration Options
Appendix 2 NIC-400 subordinate and manager port connections
Appendix 3 AXI write and read task examples

Appendix 1 NIC-400 Configuration Options

NIC-400 AMBA Designer Subordinate Parameters (SI) Configuration Options.

Parameters	**Range**	**slave_if0**	**slave_if1**	**slave_if2**	**slave_if3**
Address Width	32 to 64	32	32	32	32
Data Width	32, 64, 128, or 256	32bits	32bits	32bits	32bits
Interface Protocol	AXI3, AXI4, AHB-Lite Subordinate, AHB-Lite Target, AHB-Lite Initiator	AXI4	AXI4	AXI4	AXI4
Frequency domain crossing	ASYNC SYNC 1:1 SYNC 1:n SYNC n:1 SYNC n:m	SYNC1:1	SYNC1:1	SYNC1:1	SYNC1:1
Read Issuing	1 to 32	4	4	4	4
Write Issuing	1 to 32	4	4	4	4
Trust Zone	secure, non-secure, per-access security	non-secure	non-secure	non-secure	non-secure
Id-Width	Configurable	0	0	0	0
QoS Type	None, Fixed, Programmable, From Manager	None	None	None	None
QOS Value	0 to 15	0	0	0	0
User Sideband signal width	Configurable	0	0	0	0
Lock Support	Select if you require this manager to support locked transfers	NA	NA	NA	NA
CDAS	Single subordinate subordinate per ID, subordinate per ID_reg	subordinate per ID	subordinate per ID	subordinate per ID	subordinate per ID
Buffering (AW/AR/R/W/B)	2-32	aw/ar/r/w/b: 2/2/2/2/2	aw/ar/r/w/b: 2/2/2/2/2	aw/ar/r/w/b: 2/2/2/2/2	aw/ar/r/w/b: 2/2/2/2/2
Write Tidemark default	0-16	0	0	0	0
Timing Closure	Absent, Present, (details in later tables)	Subordinate port: Absent Manager port: Absent	Subordinate port: Absent Manager port: Absent	Subordinate port: Absent Manager port: Absent	Subordinate port: Absent Manager port: Absent

NIC400 AMBA Designer Manager Parameters (MI) Configuration Options.

Parameter	Range	master_if0	master_if1	master_if2	master_if3
Address Width	32-64	32	32	32	32
Data Width	32, 64, 128, or 256	32	32	32	32
Interface Protocol	AXI3/4, Subordinate, AHB-Lite Target, AHB-Lite Initiator APB2/3	AXI4	AXI4	AXI4	AXI4
Frequency Domain crossing	ASYNC SYNC 1:1 SYNC 1:n SYNC n:1 SYNC n:m.	SYNC1:1	SYNC1:1	SYNC1:1	SYNC1:1
Read acceptance	1 to 32	4	4	4	4
Write acceptance	1 to 32	4	4	4	4
Total acceptance	1 to 64	4	4	4	4
Trust Zone	secure, non-secure, Boot-secure	non-secure	non-secure	non-secure	non-secure
Lock Support	Select if you require this manager to support locked transfers	No	No	No	No
Buffering (AW/AR/R/W/B)	2-32	0	0	0	0
Write Tidemark default	0-16	0	0	0	0
Timing Closure (Manager Port)	Absent, Present, (details in later tables)	Absent	Absent	Absent	Absent
Timing Closure (Subordinate Port)	Absent, Present, (details in later tables)	Absent	Absent	Absent	Absent
User Sideband signal width	configurable	0	0	0	0

Appendix 2 NIC-400 subordinate and manager port connections

NIC-400 slave_if0

```
nic400_test nic400_multibus (
  .clk0resetn(clk0resetn),
  .clk0clk(clk0clk),

  .AWADDR_slave_if0(AWADDR_slave_if0),
  .AWLEN_slave_if0(AWLEN_slave_if0),
  .AWSIZE_slave_if0(AWSIZE_slave_if0),
  .AWBURST_slave_if0(AWBURST_slave_if0),
  .AWLOCK_slave_if0(AWLOCK_slave_if0),
  .AWCACHE_slave_if0(AWCACHE_slave_if0),
  .AWPROT_slave_if0(AWPROT_slave_if0),
  .AWVALID_slave_if0(AWVALID_slave_if0_masked),
  .AWREADY_slave_if0(AWREADY_slave_SIF0),
  .WDATA_slave_if0(WDATA_slave_if0),
  .WSTRB_slave_if0(WSTRB_slave_if0),
  .WLAST_slave_if0(WLAST_slave_if0),
  .WVALID_slave_if0(WVALID_slave_if0_masked),
  .WREADY_slave_if0(WREADY_slave_SIF0),
  .BRESP_slave_if0(BRESP_slave_if0),
  .BVALID_slave_if0(BVALID_slave_if0),
  .BREADY_slave_if0(BREADY_slave_if0),
  .ARADDR_slave_if0(ARADDR_slave_if0),
  .ARLEN_slave_if0(ARLEN_slave_if0),
  .ARSIZE_slave_if0(ARSIZE_slave_if0),
  .ARBURST_slave_if0(ARBURST_slave_if0),
  .ARLOCK_slave_if0(ARLOCK_slave_if0),
  .ARCACHE_slave_if0(ARCACHE_slave_if0),
  .ARPROT_slave_if0(ARPROT_slave_if0),
  .ARVALID_slave_if0(ARVALID_slave_if0),
  .ARREADY_slave_if0(ARREADY_slave_if0),
  .RDATA_slave_if0(RDATA_slave_if0),
  .RRESP_slave_if0(RRESP_slave_if0),
  .RLAST_slave_if0(RLAST_slave_if0),
  .RVALID_slave_if0(RVALID_slave_if0),
  .RREADY_slave_if0(RREADY_slave_if0),
```

NIC-400 slave_if1

```
.AWADDR_slave_if1(AWADDR_slave_if0),
.AWLEN_slave_if1(AWLEN_slave_if0),
.AWSIZE_slave_if1(AWSIZE_slave_if0),
.AWBURST_slave_if1(AWBURST_slave_if0),
.AWLOCK_slave_if1(AWLOCK_slave_if0),
.AWCACHE_slave_if1(AWCACHE_slave_if0),
.AWPROT_slave_if1(AWPROT_slave_if0),
.AWVALID_slave_if1(AWVALID_slave_if1_masked),
.AWREADY_slave_if1(AWREADY_slave_SIF1),
.WDATA_slave_if1(WDATA_slave_if0),
.WSTRB_slave_if1(WSTRB_slave_if0),
.WLAST_slave_if1(WLAST_slave_if0),
.WVALID_slave_if1(WVALID_slave_if1_masked),
.WREADY_slave_if1(WREADY_slave_SIF1),
.BRESP_slave_if1(),
.BVALID_slave_if1(),
.BREADY_slave_if1(1'b1),
.ARADDR_slave_if1(1'b0),
.ARLEN_slave_if1(1'b0),
.ARSIZE_slave_if1(1'b0),
.ARBURST_slave_if1(1'b0),
.ARLOCK_slave_if1(1'b0),
.ARCACHE_slave_if1(1'b0),
.ARPROT_slave_if1(1'b0),
.ARVALID_slave_if1(1'b0),
.ARREADY_slave_if1(),
.RDATA_slave_if1(),
.RRESP_slave_if1(),
.RLAST_slave_if1(),
.RVALID_slave_if1(),
.RREADY_slave_if1(1'b0),
```

NIC-400 slave_if2

```
.AWADDR_slave_if2(AWADDR_slave_if0),
.AWLEN_slave_if2(AWLEN_slave_if0),
.AWSIZE_slave_if2(AWSIZE_slave_if0),
.AWBURST_slave_if2(AWBURST_slave_if0),
.AWLOCK_slave_if2(AWLOCK_slave_if0),
.AWCACHE_slave_if2(AWCACHE_slave_if0),
.AWPROT_slave_if2(AWPROT_slave_if0),
.AWVALID_slave_if2(AWVALID_slave_if2_masked),
.AWREADY_slave_if2(AWREADY_slave_SIF2),
.WDATA_slave_if2(WDATA_slave_if0),
.WSTRB_slave_if2(WSTRB_slave_if0),
.WLAST_slave_if2(WLAST_slave_if0),
.WVALID_slave_if2(WVALID_slave_if2_masked),
.WREADY_slave_if2(WREADY_slave_SIF2),
.BRESP_slave_if2(),
.BVALID_slave_if2(),
.BREADY_slave_if2(1'b1),
.ARADDR_slave_if2(1'b0),
.ARLEN_slave_if2(1'b0),
.ARSIZE_slave_if2(1'b0),
.ARBURST_slave_if2(1'b0),
.ARLOCK_slave_if2(1'b0),
.ARCACHE_slave_if2(1'b0),
.ARPROT_slave_if2(1'b0),
.ARVALID_slave_if2(1'b0),
.ARREADY_slave_if2(),
.RDATA_slave_if2(),
.RRESP_slave_if2(),
.RLAST_slave_if2(),
.RVALID_slave_if2(),
.RREADY_slave_if2(1'b0),
```

NIC-400 slave_if3

```
.AWADDR_slave_if3(AWADDR_slave_if0),
.AWLEN_slave_if3(AWLEN_slave_if0),
.AWSIZE_slave_if3(AWSIZE_slave_if0),
.AWBURST_slave_if3(AWBURST_slave_if0),
.AWLOCK_slave_if3(AWLOCK_slave_if0),
.AWCACHE_slave_if3(AWCACHE_slave_if0),
.AWPROT_slave_if3(AWPROT_slave_if0),
.AWVALID_slave_if3(AWVALID_slave_if3_masked),
.AWREADY_slave_if3(AWREADY_slave_SIF3),
.WDATA_slave_if3(WDATA_slave_if0),
.WSTRB_slave_if3(WSTRB_slave_if0),
.WLAST_slave_if3(WLAST_slave_if0),
.WVALID_slave_if3(WVALID_slave_if3_masked),
.WREADY_slave_if3(WREADY_slave_SIF3),
.BRESP_slave_if3(),
.BVALID_slave_if3(),
.BREADY_slave_if3(1'b1),
.ARADDR_slave_if3(1'b0),
.ARLEN_slave_if3(1'b0),
.ARSIZE_slave_if3(1'b0),
.ARBURST_slave_if3(1'b0),
.ARLOCK_slave_if3(1'b0),
.ARCACHE_slave_if3(1'b0),
.ARPROT_slave_if3(1'b0),
.ARVALID_slave_if3(1'b0),
.ARREADY_slave_if3(),
.RDATA_slave_if3(),
.RRESP_slave_if3(),
.RLAST_slave_if3(),
.RVALID_slave_if3(),
.RREADY_slave_if3(1'b0),
```

NIC-400 master_if0

```
.AWLEN_master_if0(AWLEN_master_if0),
.AWSIZE_master_if0(AWSIZE_master_if0),
.AWBURST_master_if0(AWBURST_master_if0),
.AWLOCK_master_if0(AWLOCK_master_if0),
.AWCACHE_master_if0(AWCACHE_master_if0),
.AWVALID_master_if0(AWVALID_master_if0),
.AWPROT_master_if0(AWPROT_master_if0),
.AWADDR_master_if0(AWADDR_master_if0),
.AWREADY_master_if0(AWREADY_master_if0),
.WLAST_master_if0(WLAST_master_if0),
.WVALID_master_if0(WVALID_master_if0),
.WSTRB_master_if0(WSTRB_master_if0),
.WDATA_master_if0(WDATA_master_if0),
.WREADY_master_if0(WREADY_master_if0),
.BREADY_master_if0(BREADY_master_if0),
.BVALID_master_if0(BVALID_master_if0),
.BRESP_master_if0(BRESP_master_if0),
.ARADDR_master_if0(ARADDR_master_if0),
.ARLEN_master_if0(ARLEN_master_if0),
.ARSIZE_master_if0(ARSIZE_master_if0),
.ARBURST_master_if0(ARBURST_master_if0),
.ARLOCK_master_if0(ARLOCK_master_if0),
.ARCACHE_master_if0(ARCACHE_master_if0),
.ARPROT_master_if0(ARPROT_master_if0),
.ARVALID_master_if0(ARVALID_master_if0),
.ARREADY_master_if0(ARREADY_master_if0),
.RDATA_master_if0(RDATA_master_if0),
.RRESP_master_if0(RRESP_master_if0),
.RLAST_master_if0(RLAST_master_if0),
.RVALID_master_if0(RVALID_master_if0),
.RREADY_master_if0(RREADY_master_if0),
```

NIC-400 master_if1

```
.AWID_master_if1(AWID_master_if1),
.AWLEN_master_if1(AWLEN_master_if1),
.AWSIZE_master_if1(AWSIZE_master_if1),
.AWBURST_master_if1(AWBURST_master_if1),
.AWLOCK_master_if1(AWLOCK_master_if1),
.AWCACHE_master_if1(AWCACHE_master_if1),
.AWVALID_master_if1(AWVALID_master_if1),
.AWPROT_master_if1(AWPROT_master_if1),
.AWADDR_master_if1(AWADDR_master_if1),
.AWREADY_master_if1(AWREADY_master_if1),
.WLAST_master_if1(WLAST_master_if1),
.WVALID_master_if1(WVALID_master_if1),
.WSTRB_master_if1(WSTRB_master_if1),
.WDATA_master_if1(WDATA_master_if1),
.WREADY_master_if1(WREADY_master_if1),
.BID_master_if1(BID_master_if1),
.BREADY_master_if1(BREADY_master_if1),
.BVALID_master_if1(BVALID_master_if1),
.BRESP_master_if1(BRESP_master_if1),
.ARID_master_if1(ARID_master_if1),
.ARADDR_master_if1(ARADDR_master_if1),
.ARLEN_master_if1(ARLEN_master_if1),
.ARSIZE_master_if1(ARSIZE_master_if1),
.ARBURST_master_if1(ARBURST_master_if1),
.ARLOCK_master_if1(ARLOCK_master_if1),
.ARCACHE_master_if1(ARCACHE_master_if1),
.ARPROT_master_if1(ARPROT_master_if1),
.ARVALID_master_if1(ARVALID_master_if1),
.ARREADY_master_if1(ARREADY_master_if1),
.RID_master_if1(RID_master_if1),
.RDATA_master_if1(RDATA_master_if1),
.RRESP_master_if1(RRESP_master_if1),
.RLAST_master_if1(RLAST_master_if1),
.RVALID_master_if1(RVALID_master_if1),
.RREADY_master_if1(RREADY_master_if1),
```

NIC-400 master_if2

```
.AWID_master_if2(AWID_master_if2),
.AWLEN_master_if2(AWLEN_master_if2),
.AWSIZE_master_if2(AWSIZE_master_if2),
.AWBURST_master_if2(AWBURST_master_if2),
.AWLOCK_master_if2(AWLOCK_master_if2),
.AWCACHE_master_if2(AWCACHE_master_if2),
.AWVALID_master_if2(AWVALID_master_if2),
.AWPROT_master_if2(AWPROT_master_if2),
.AWADDR_master_if2(AWADDR_master_if2),
.AWREADY_master_if2(AWREADY_master_if2),
.WLAST_master_if2(WLAST_master_if2),
.WVALID_master_if2(WVALID_master_if2),
.WSTRB_master_if2(WSTRB_master_if2),
.WDATA_master_if2(WDATA_master_if2),
.WREADY_master_if2(WREADY_master_if2),
.BID_master_if2(BID_master_if2),
.BREADY_master_if2(BREADY_master_if2),
.BVALID_master_if2(BVALID_master_if2),
.BRESP_master_if2(BRESP_master_if2),
.ARID_master_if2(ARID_master_if2),
.ARADDR_master_if2(ARADDR_master_if2),
.ARLEN_master_if2(ARLEN_master_if2),
.ARSIZE_master_if2(ARSIZE_master_if2),
.ARBURST_master_if2(ARBURST_master_if2),
.ARLOCK_master_if2(ARLOCK_master_if2),
.ARCACHE_master_if2(ARCACHE_master_if2),
.ARPROT_master_if2(ARPROT_master_if2),
.ARVALID_master_if2(ARVALID_master_if2),
.ARREADY_master_if2(ARREADY_master_if2),
.RID_master_if2(RID_master_if2),
.RDATA_master_if2(RDATA_master_if2),
.RRESP_master_if2(RRESP_master_if2),
.RLAST_master_if2(RLAST_master_if2),
.RVALID_master_if2(RVALID_master_if2),
.RREADY_master_if2(RREADY_master_if2),
```

NIC-400 master_if3

```
.AWID_master_if3(AWID_master_if3),
.AWLEN_master_if3(AWLEN_master_if3),
.AWSIZE_master_if3(AWSIZE_master_if3),
.AWBURST_master_if3(AWBURST_master_if3),
.AWLOCK_master_if3(AWLOCK_master_if3),
.AWCACHE_master_if3(AWCACHE_master_if3),
.AWVALID_master_if3(AWVALID_master_if3),
.AWPROT_master_if3(AWPROT_master_if3),
.AWADDR_master_if3(AWADDR_master_if3),
.AWREADY_master_if3(AWREADY_master_if3),
.WLAST_master_if3(WLAST_master_if3),
.WVALID_master_if3(WVALID_master_if3),
.WSTRB_master_if3(WSTRB_master_if3),
.WDATA_master_if3(WDATA_master_if3),
.WREADY_master_if3(WREADY_master_if3),
.BID_master_if3(BID_master_if3),
.BREADY_master_if3(BREADY_master_if3),
.BVALID_master_if3(BVALID_master_if3),
.BRESP_master_if3(BRESP_master_if3),
.ARID_master_if3(ARID_master_if3),
.ARADDR_master_if3(ARADDR_master_if3),
.ARLEN_master_if3(ARLEN_master_if3),
.ARSIZE_master_if3(ARSIZE_master_if3),
.ARBURST_master_if3(ARBURST_master_if3),
.ARLOCK_master_if3(ARLOCK_master_if3),
.ARCACHE_master_if3(ARCACHE_master_if3),
.ARPROT_master_if3(ARPROT_master_if3),
.ARVALID_master_if3(ARVALID_master_if3),
.ARREADY_master_if3(ARREADY_master_if3),
.RID_master_if3(RID_master_if3),
.RDATA_master_if3(RDATA_master_if3),
.RRESP_master_if3(RRESP_master_if3),
.RLAST_master_if3(RLAST_master_if3),
.RVALID_master_if3(RVALID_master_if3),
.RREADY_master_if3(RREADY_master_if3)

);
```

Appendix 3 AXI write and read task examples

axi_write

```
task automatic axi_write;
  input [31:0] addr;
  input [31:0] data;
  begin
    `ifdef MULTIWRITE
      AWADDR_slave_if0 = addr;
    `elsif NODEBUS
      AWADDR_slave_if0 = {addr[29:0],2'b00};
    `endif

    WDATA_slave_if0 = data;
    WSTRB_slave_if0 = 4'b1111;
    WLAST_slave_if0 = 1'b1;

    fork
      begin
        fork
          begin
            wait(AWREADY_slave_if0);
            @(posedge clock);
            AWVALID_slave_if0 = 1'b1;
            @(posedge clock);
            AWVALID_slave_if0 = 1'b0;
          end
          begin
            wait(WREADY_slave_if0);
            @(posedge clock);
            WVALID_slave_if0 = 1'b1;
            @(posedge clock);
            WVALID_slave_if0 = 1'b0;
          end
          begin
            @(posedge clock);
            BREADY_slave_if0 = 1'b1;
            wait(BVALID_slave_if0);
            @(posedge clock);
            BREADY_slave_if0 = 1'b0;
          end
        join
      end
      begin
        #1000000;
        $display("Write timeout reached!");
      end
    join_any
    disable fork;

    if (BRESP_slave_if0 != 2'b00)
      $display("Error: Write response NOK! ");

    @(posedge clock);
    WLAST_slave_if0 = 1'b0;
    AWVALID_slave_if0 = 1'b0;
    WVALID_slave_if0 = 1'b0;
    WSTRB_slave_if0 = 4'b0000;
  end
endtask
```

axi_read

```
task automatic axi_read;
  input [31:0] addr;
  input [31:0] expected_data;
  begin
    `ifdef MULTIWRITE
      ARADDR_slave_if0 = addr;
    `elsif NODEBUS
      ARADDR_slave_if0 = {addr[29:0],2'b00};
    `endif

    ARLEN_slave_if0 = 0;
    ARSIZE_slave_if0 = 3'b010;
    fork
      begin
        wait(ARREADY_slave_if0);
        ARVALID_slave_if0 = 1'b1;
        @(posedge clock);
        ARVALID_slave_if0 = 1'b0;
        wait(RVALID_slave_if0);
        @(posedge clock);
        RREADY_slave_if0 = 1'b1;
      end
      begin
        #1000000;
        $display("Read timeout reached!");
      end
    join_any
    disable fork;

    if (RDATA_slave_if0 != expected_data) begin
      $display("Error: Mismatch in AXI4 read at %x: ", addr,
        "expected %x, received %x",
        expected_data, RDATA_slave_if0);
    end

    if (RRESP_slave_if0 != 2'b00)
      $display("Error: Read response NOK! ");


    @(posedge clock);
    ARVALID_slave_if0 = 1'b0;
    RREADY_slave_if0 = 1'b0;
  end
endtask
```