TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Design in HDL of a DMA engine that complies with the AMBA 5 CHI communication protocol

---

Author:
Angelos KOURKOULOS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Sotirios IOANNIDIS
Dr. Aggelos IOANNOU
(FORTH/LBNL)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

June 27, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design in HDL of a DMA engine that complies with the AMBA 5 CHI communication protocol

by Angelos KOURKOULOS

A Direct Memory Access (DMA) is a system that allows a device to transfer data directly to and from main memory, bypassing the central processing unit (CPU). This can be useful for transferring quickly large amounts of data, as it allows the CPU to perform other tasks while the DMA controller handles the data transfer. The aim of this thesis is to design, optimize and verify in HDL an IP Core (Intellectual Property Core) DMA engine that complies with AMBA 5 CHI(Coherent Hub Interface) protocol and efficiently transfers data to and from the CHI hub which can be used in HPC (High-Performance Computing) to improve the performance. This DMA controller is designed to be able to handle a scalable amount of memory transfers, generically schedule them based on the user's demands and transfer the appropriate data at any address byte offset in memory. The proposed DMA is designed to work with systems that use AMBA 5 CHI architecture as it is a state-of-the-art technology designed by ARM that classifies different components in a system by node type and provides a means for communication between nodes. CHI is designed for High bandwidth, efficiency, scalability, and reliability, while it offers the capability for memory and cache coherency which are the basic reasons that CHI is widely used in HPC. By complying with this protocol, the presented IP Core can utilize the advantages that CHI provides as well as the features of the DMA architecture and be a useful tool to improve the performance of systems that would incorporate it.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design in HDL of a DMA engine that complies with the AMBA 5 CHI communication protocol

by Angelos KOURKOULOS

Η Άμεση Πρόσβαση Μνήμης (DMA) είναι ένα σύστημα το οποίο επιτρέπει σε μια άλλη συσκευή να μεταφέρει δεδομένα σε και από την κεντρική μνήμη προσπερνόντας την κεντρική μονάδα επεξεργασίας (CPU). Αυτό ειναι χρήσιμο για τη μεταφορά μεγάλου όγκου δεδομένων γρήγορα αφού επιτρέπει στη CPU να εκτελεί άλλες διεργασίες κατα την διάρκεια που η DMA διαχειρίζεται την μεταφορά των δεδομένων. Ο στόχος αυτής της διπλωματικής είναι η σχεδίαση σε HDL ενός IP Core DMA για διασύνδεση με συστήματα μέσω του πρωτοκόλου AMBA 5 CHI(Coherent Hub Interface) η οποία θα μεταφέρει αποτελεμσατικά δεδομένα μέσα στο κεντρικό σύστημα, η οποία μπορεί να χρησιμοποιηθεί για τη βελτίωση της απόδωσης κάποιου HPC. Αυτη η DMA ειναι σχεδιασμένη ώστε να μπορεί να διαχειριστεί ενα κλιμακόμενο αριθμό μεταφορών, να τις δρομολογεί με βάση τις απαιτήσεις του χρήστη και μεταφέρει τα απαραίτητα δεδομένα σε οποιαδίποτε διεύθηνση μνύμης. Η προτεινόμενη DMA είναι σχεδιασμένος να λειτουργεί με συστήματα που χρησιμοποιουν την AMBA 5 CHI αρχιτεκτονική αφού είναι τεχνολογία αιχμής σχεδιασμένη από την ARM η οποία κατατάσει διαφορετικά αντικείμενα μέσα σε ένα σύστημα με την μορφή κόμβων και παρέχει ένα μέσο επικοινωνίας μεταξύ τους. Το CHI είναι σχεδιασμένο να παρέχει υψηλό εύρος ζώνης, αποδοτικότητα, επεκτασιμότητα και αξιοπιστία ενώ προσφέρει δυνατότητα για κύρια μνήμη και μνήμη cache coherency που είναι οι βασικοί λόγοι που χρησιμοποιείται ευρέως σε υπολογιστές υψηλής απόδοσης. Με τη συμβατότητα σε αυτό το προτόκολο το παρουσιαζόμενο IP Core μπορεί να αξιοποιήσει τα πλεονεκτήματα που προσφέρει το CHI καθώς και τα χαρακτηριστικά της αρχιτεκτονικής της DMA ώστε να είναι ένα χρήσιμο εργαλείο για να βελτιώσει την απόδοση των συστημάτων που θα το ενσωματώσει.

Acknowledgements

I would like to express my deepest appreciation and gratitude to all those who have helped me throughout the completion of this thesis.

First of all, I would like to thank my Diploma Thesis Committee, Prof. A. Dollas, Assoc. Prof. S. Ioannidis, and Dr. A. Ioannou for their support of this thesis and their comments on the text.

I would especially like to acknowledge Dr.Ioannou, for their unwavering guidance, support, and patience throughout the entire process. Their valuable insights and feedback have been instrumental in shaping my research and strengthening the quality of this work.

I am also deeply indebted to Dr. Pantelis Xirouchakis, for their guidance and expertise in my thesis subject. Their feedback and support have been invaluable throughout the completion of this project.

I would also like to extend my heartfelt thanks to my family and friends for their unwavering encouragement, understanding, and support. Their belief in me has been a constant source of inspiration and motivation.

Finally, I would like to thank the Technical University of Crete, Electrical and Computer Engineering, and all the faculty and staff who have contributed to my academic journey. Their dedication to excellence has instilled in me a passion for learning and a drive for success.

Contents

List of Figures

List of Tables

List of Abbreviations

AHB	**A**dvanced **H**igh-performance **B**us
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
AI	**A**rtificial **I**ntelligence
APB	**A**dvanced **P**eripheral **B**us
AXI	**A**rtificial e**X**tensible **I**nterface
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
CHI	**C**oherence **H**ub **I**nterface
DMA	**D**irect **M**emory **A**ccess
DDR	**D**ouble **D**ata **R**ate memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
GPU	**G**raphic **P**rocessor **U**nit
HDL	**H**ardware **D**escription **L**anguage
HPC	**H**ight **P**erformance **C**omputing
I/O	**I**nput / **O**utput
LUT	**L**ook **U**p **T**able
MCU	**M**icro**C**ontroller **U**nit
MPU	**M**icro**P**rocessor **U**nit
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MUX	**M**ultipexer
RAM	**R**andom **A**ccess **M**emory
RTL	**R**egister **T**ransfer **L**evel
SSD	**S**olid **S**tate **D**rive
SoC	**S**ystem **o**n **C**hip

Chapter 1

Introduction

1.1 Motivation

High-Performance Computing (HPC) refers to the use of powerful computers or computing clusters that use parallel processing and advanced algorithms for solving complex computational problems and processing massive multi-dimensional datasets delivering significantly higher performance compared to traditional computers. HPC systems typically consist of clusters of computers or supercomputers that work together to provide results about difficult problems that require highly intensive calculations such as automated sequence DNA, simulations, artificial intelligence (AI), analyzing radar and GPS systems, stock trading, etc. However, a critical bottleneck in HPC performance is usually data movement. To encounter this problem many HPC systems implement devices called DMA engines which is a key component in reducing this bottleneck.

Direct Memory Access (DMA) is a specialized hardware technology that is widely used in modern digital systems and HPC. It is very useful because it works with processors and reduces their workload. As the term indicates DMA's purpose is to directly access data from memory i.e. without the involvement of a processor. This way many devices can access memory directly through DMA and leave processor with more free time to execute other tasks. Subsequently, the parallelism of the system is improved hence the overall performance is boosted[1]. Also, DMA typically has higher bandwidth and it can transfer data in larger chunks than the processor without the delays and interruptions that can occur when the CPU is involved in the process hence the data movement can be more efficient. When the DMA has finished the data movement, the core can be informed of the completion by polling the status of the DMA or by receiving an Interrupt from the engine. Since

DMA handles the data movement it allows the CPU to enter a low-power state which can reduce the overall power consumption of the system as the processor is one of the units that consume the most energy [2]. Some other hardware entities that usually use DMAs are disk drive controllers, graphics cards, network cards, sound cards, etc.

Another very important and common technique that many HPC and in general computer systems with multiple processors use nowadays to improve their performance is coherency. Coherency refers to the ability of devices in a system to access shared memory in a consistent and coherent manner which reduces the impact of data movement on system performance by ensuring that all processing units have access to the most up-to-date data. This means that all devices can see the same view of the memory and can read and write to it without interference. Also, there are some specialized types of coherency called cache coherency and IO coherency which can help to further improve the performance by giving specialized characteristics to the devices that implement them. Cache coherency refers to the use of a cache coherency protocol to ensure that all processors have a valid form of data within their caches which are not modified in another processor's cache by keeping track of the status of each cache line in the system. Cache coherency is often used in multi-processor systems to improve performance by reducing the number of memory accesses and minimizing cache coherence traffic. Although in such systems the conservation of cache coherency is a vital requirement for many processors, there is a need for some components to read and write the last snapshot of memory but do not require to be informed of changes in data by other processors from the cache coherent protocol as they probably do not implement a cache. These devices can coexist with other cache-coherent components but they are in a different category and referred to as IO-coherent. Coherence protocols are implemented by systems that allow the connected components to successfully obtain memory/cache coherency by obeying a set of rules defined by a communication protocol. There are a variety of such protocols that promise sustainable memory coherency with a high communication rate. A very recent protocol that ensures the conservation of coherency and high-speed transmission that this thesis is focused on is the AMBA 5 Coherence Hub Interface (CHI).

AMBA (Advanced Microcontroller Bus Architecture) is an interconnect specification that ARM developed in the late 1990s and it is widely used for on-chip communications in systems-on-a-chip (SoCs), application-specific integrated circuits (ASIC), and embedded systems. It defines a set of interfaces for interconnecting a large number of devices such as processors, memory controllers, and peripheral devices, and provides a standard way for these components to communicate with each other. AMBA introduced many buses with the corresponding communication protocols starting from the Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB) in 1996. In the next few years, AMBA evolved its protocols and released new updated buses and interfaces. The latest protocol that AMBA supports is the Coherence Hub Interface (CHI) which is an evolution of the AXI Coherency Extensions (ACE) protocol. CHI is a top-of-the-line protocol for creating large and complex systems. It offers scalability, high bandwidth, and high availability which can help to improve the performance of many applications. Also, it provides capabilities for Coherency, Coherence's distributed caching, and IO Coherency that speed up the load-store Data operations and significantly decrease memory access time. AMBA 5 CHI is used in a variety of systems that require efficient communication as the components in the system increase and the volume of traffic grows. Some of the systems that employ CHI in practice are the Cache coherent network based on ARM CoreLink CCN-504 [3] [4] and the CoreLink CMN-600 [5] which interconnects Cortex-A76/A60 and other components with dynamic memory controllers. Due to its special characteristics, CHI has been also recently implemented for gem5 Ruby which performs a detailed simulation model for memory subsystems[6]. In general, this protocol applies very effectively in a variety of applications such as automotive, mobile, data centers, and networking.

In this thesis we thoroughly designed the architectural components of a CHI-compliant IP Core DMA engine that can potentially be utilized and improve HPC systems, implemented in system-verilog HDL (Hardware Description Language), and verified its operation by the use of behavioral simulations. The DMA controller is designed to interact with AMBA 5 CHI protocol and allow the CPU to perform other tasks since DMA transfers data to and from memory exploiting the advantages that the CHI interconnection offers. At the end of this work, the interconnection independent through and latency are measured with the use of behavioral simulation, and the utilization of the system is approximated from the synthesis tool with great results in all measurements.

1.2 Scientific Contributions

This thesis gave us the opportunity for developing a modern and effective DMA engine that can be utilized to improve the performance of a multiprocessor system while advancing our knowledge about the subject for further development in the future. It also provided us the chance for advancing our expertise in the architectural design field, improving our skills in hardware optimization and debugging, and enhancing our experience with HDL and the tool of Xilinx. The thesis presents in detail the architecture of a CHI-based DMA Controller, a subject for which, to the best of our knowledge, there is no similar work. The designed IP Core allows devices that cooperate with it to move data in cache-coherent systems compatible with the CHI protocol. The CHI compliance of the DMA engine allows it to operate more efficiently, enhancing the speed of data transfers and benefiting the other devices to drastically improve their performance. The cache coherence feature that CHI provides has a big influence on the performance of the system, as the DMA can read data from some processor's cache, if it has the desired data, instead of the main memory. In addition, the proposed DMA has been designed to be generic and parameterized for effortless future expansion and to be flexible in adjustments that could occur due to changes in technology or demands of the application. More specifically, the engine can receive a programmable amount of transfers using an architecture that leverages a BRAM as transfer Descriptor space. Also, it provides a generic method for scheduling the execution of memory transfers from different processes, which is useful as the optimal scheduling way can vary according to the implemented communication protocol or the application. Furthermore, as already stated, the proposed DMA controller utilizes the CHI as it is a state-of-the-art communication protocol and provides high-performance capabilities for coherence systems. However, as the technology evolves, a new better protocol could be released and outperform CHI. For this reason, the DMA engine has been designed to keep all the communication logic within one module. This can be easily modified and make the engine operate with any different state-of-the-art protocol without changing the core of the design. In this case, the generic scheduling is very helpful, as it adapts to the protocol demands. Overall, the use of an IP core DMA that uses the AMBA 5 CHI protocol enables system designers to easily integrate the DMA engine into their designs, making it a versatile and cost-effective solution for HPC applications. It can achieve the maximum possible throughput and minor

latency in most cases and improves the performance and efficiency of the system by reducing the data movement bottleneck and enabling faster and more efficient data transfers between processors and memory.

1.3 Thesis Outline

The rest of this thesis is organized as follows.

- **Chapter 2 - Theoretical Background and Related Work:** Briefly description of specifications about other projects based on DMAs and the CHI.
- **Chapter 3 - Architecture of DMA:** Presentation of the architecture of the IP Core and analyzing the implementation of its main components.
- **Chapter 4 - Simulation and Testing:** Description of the simulation and verification methods used for the validation of the system
- **Chapter 5 - Results:** Performance representation of the DMA. Throughput, Latency, and Utilization.
- **Chapter 6 - Conclusions and Future Work:** Conclusion of the thesis and report for optimizations and further development that can be integrated in the future.

Chapter 2

Theoretical Background and Related Work

The concept of using a DMA to improve the performance of the CPU is well known since last century. The first personal computer of IBM that used a DMA engine, is the IBM PC (model 5150) and released in 1981 [7], while older computers like IBM System/360 included channels for transferring data between a peripheral and main memory, that could be compared to DMA [8]. For this reason, there are many projects on the subject each of which targets a different field of application. Some recent publications about DMA controllers are presented in the following sections.

2.1 DMAs based on the AHB bus architecture

The AHB (Advanced High-performance Bus) protocol is a high-performance, synchronous, and pipelined bus protocol. It is developed by ARM and it is one of the protocols specified by AMBA. It is used to connect high-speed components in a system design. It is designed to provide a high-speed interconnect between numerous devices, such as processors, memories, and peripherals.

2.1.1 Design and Implementation of a Direct Memory Access Controller Based on Microcontroller Unit

This work was made by Chao Lu and published in 2022 [9]. In this publication, the author describes the design of a DMA controller based on AHB-Lite protocol aiming to exempt MCU (Microcontroller Unit) from data transfers to improve the overall system performance. This proposed design presents

a multichannel controller with a fixed priority level arbitration for the selection of each channel. It also supports a configurable source and destination data width, configurable burst type, and 4 methods to transfer data between memory and peripheral.

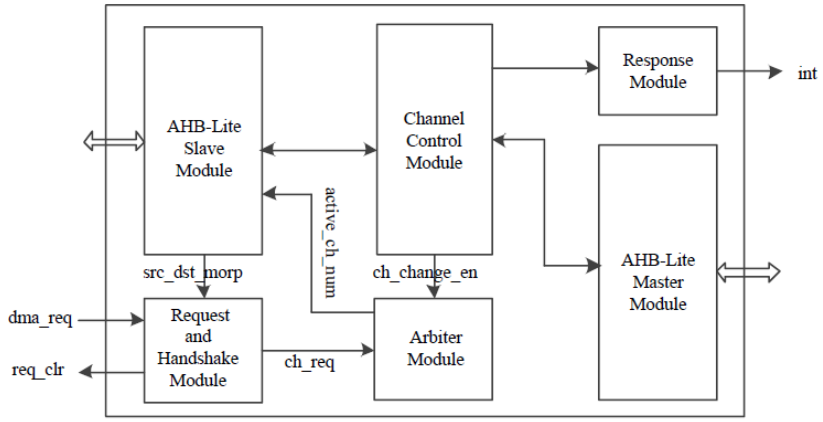

FIGURE 2.1: Overall architecture of DMA controller (Source: Chao LU)[9]

As shown in the diagram of the architecture Fig.2.1 there is the AHB-Lite slave module which receives the appropriate information from the CPU in AHB compliant manner about the configuration of channels. Then there are the Channel request and transfer control modules which produce the transfer request signal according to the targeted devices and generate the necessary control signals for the data transfer process respectively. The Arbiter module gives priority based on the request signals of each channel to the highest priority request. Moreover, when a burst transfer or all of a whole channel is finished the Handshake module generates a clear signal in response to the clear request signal of the peripheral unit. Also, there is a response module to generate an interrupt when a transfer is over. Lastly, the AHB-Lite master module is responsible for the execution of the transfer based on the control information for the appropriate channel and the AHB interface.

Finally, the proposed DMA controller was verified based on the ARM Cortex-M3 system as shown in the figure with transfers triggered from software and hardware, and by using transfer channel switching.

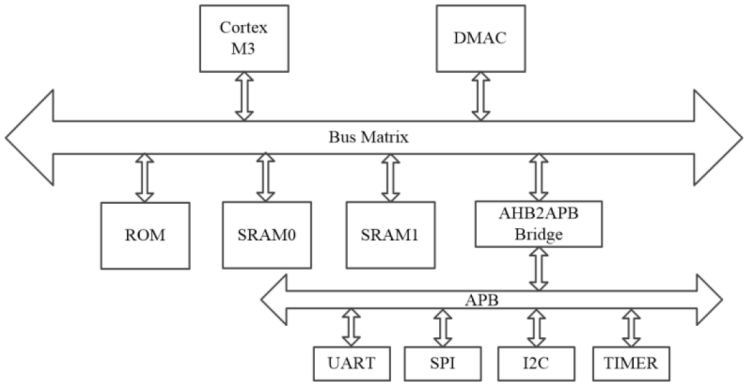

FIGURE 2.2: Environment diagram for DMA verification (Source: Chao LU)[9]

2.1.2 Design and Implementation of an Advanced DMA Controller on AMBA-Based SoC

In this publication of 2009 the authors Guoliang Ma and Hu He describe the Design and implementation of a DMA engine based on AMBA protocols[10]. More specifically the DMA works with both AHB and APB(Advanced Peripheral Bus) protocols as it lies between the two buses and it can interact with memory, MPU(microprocessor unit) and also functions as an APB bridge for peripheral communication Fig.2.3.

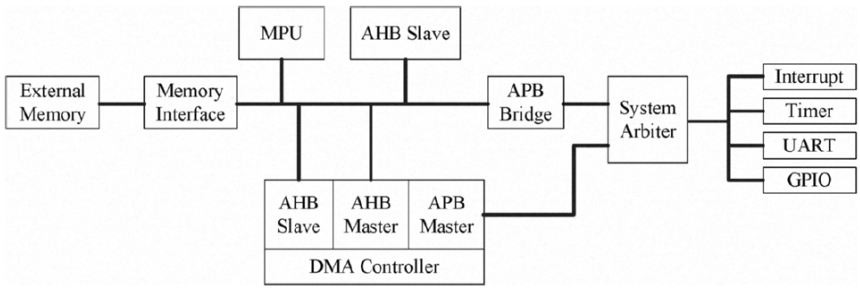

FIGURE 2.3: DMA Connectivity (Source: Guoliang Ma)[10]

The controller achieves the two buses to run in parallel by adopting buffer or non-buffer data transfer mode. This proposed DMA implements 8 channels with trigger support from hardware and software and it is able for linking operations, channel chaining transfers, and multi-dimensions transmission. The arbitration of the channels adopts hardware priority combined with a weighted priority rotational algorithm.

The procedure begins with the MPU programming the parameter set of the appropriate channel by the AHB slave interface. The requests from peripherals, software, or chaining transfers are handled by the Request and Respond

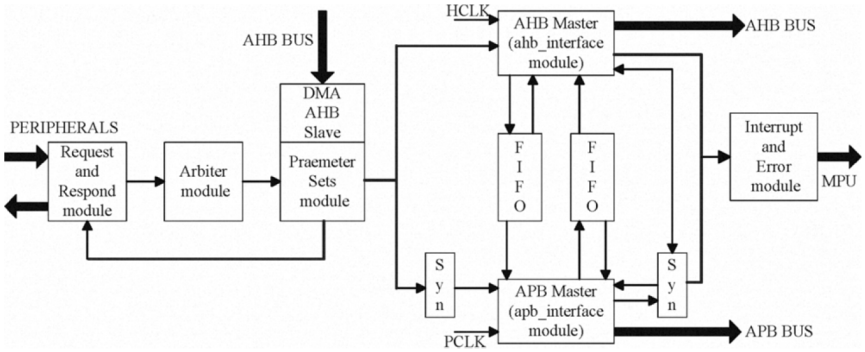

FIGURE 2.4: DMA Architecture (Source: Guoliang Ma)[10]

module shown in figure 2.4. These requests are controlled by the arbiter module and program the corresponding parameter set for the submission of the transfer. Subsequently, the AHB master and APB master modules assert the appropriate signals to control the corresponding bus and execute their part of the transfer between the bus and FIFOs. AHB and APB operations are independent and can process in parallel. When the transfer is over or there is an error an interrupt is generated for the MPU.

2.1.3 AMBA Based Advanced DMA Controller for SoC

Again in this work, the authors Abdullah Aljumah and Mohammed Altaf Ahmed propose a DMA controller design based on AHB-Lite and APB protocols to enhance the overall performance of SoC(System on Chip)[11]. The proposed DMA functions with both AHB and APB buses and allow them to work simultaneously by implementing buffering mechanism with an asynchronous FIFO for their synchronization. It is designed to perform a large volume of data transfer with very low timing characteristics while keeping a low gate count.

The designed DMA consists of a generic FIFO, a priority Arbiter, and AHB and APB modules for the interconnection with the buses and the transfer of data Fig.2.6. Also, in the figure can be seen the connection of an ARM processor and the system memory with the DMA. The procedure begins when the CPU configures the DMA. Subsequently, when the priority Arbiter grants the appropriate request it informs the AHB master to initiate the transfer. The AHB master then completes the transmission between the AHB and FIFO. In the same way, the APB master executes the transmission between FIFO and APB when it achieves control of the bus. The AHB and APB interfaces are independent and can operate concurrently. The DMA can achieve

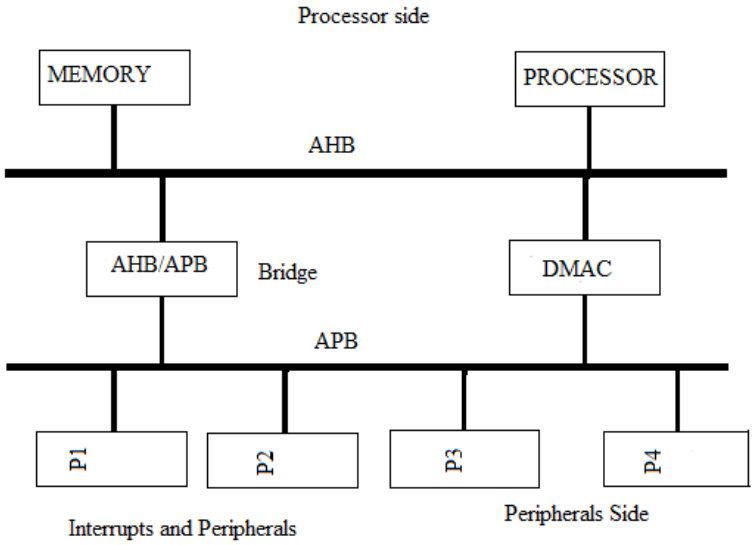

FIGURE 2.5: AMBA system (Source: Abdullah Aljumah)[11]

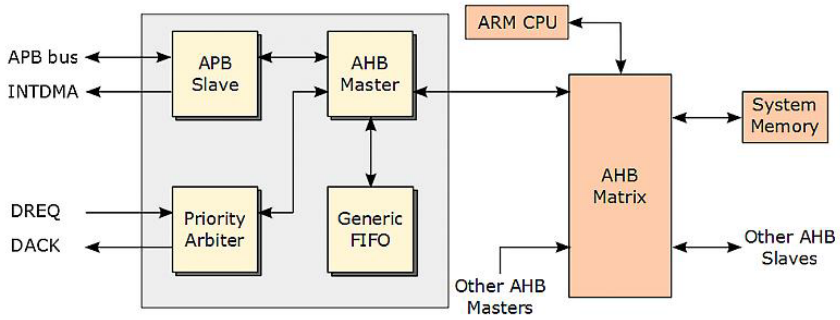

FIGURE 2.6: DMA Architecture (Source: Abdullah Aljumah)[11]

memory-to-memory, peripheral-to-memory, and memory-to-peripherals. In this implementation, the DMA can work only in one master and many slaves mode(AHB-Lite protocol) but in a later work of the project, the multiple masters and multiple slaves mode is achieved.[12]

2.2 DMA projects based on AXI bus protocol

The AXI (Advanced eXtensible Interface) is another widely used communication protocol of AMBA. This protocol is designed to achieve communication between components in an SoC with high bandwidth, low latency, and high throughput. Although AXI and AHB both target common goals they have many differences in their architecture and feature support like outstanding or out-of-order transactions with AXI being the more advanced protocol.

2.2.1 AXI Central Direct Memory Access by Xilinx

Xilinx introduces an IP Core of a Central Direct Memory Access based on the AXI4 communication protocol that provides high bandwidth transfers between a memory-mapped source address and a memory-mapped destination address for use in an embedded system [13]. The proposed IP core supports three AXI interfaces, one AXI4 for the transfer of data, one AXI4-Lite slave for accessing the registers of the DMA, and one AXI4 master for the optional Scatter/Gather function. It supports optional Scatter/Gather for sequencing tasks from the CPU and also supports optional Data Realignment Engine.

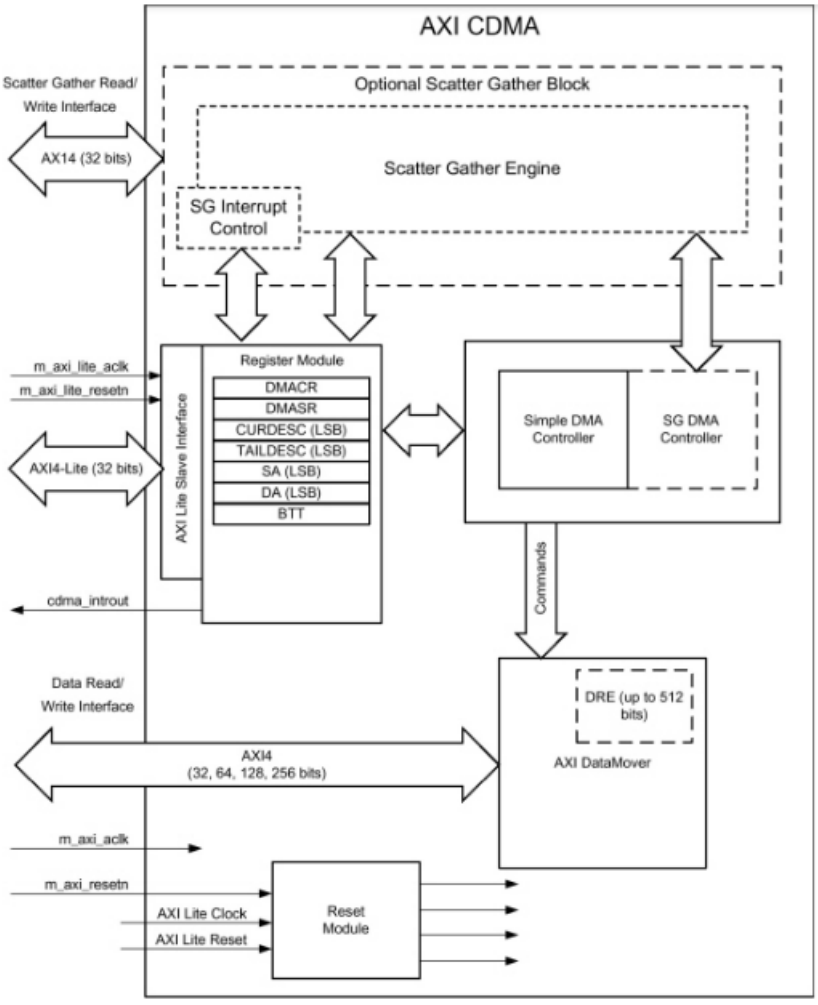

FIGURE 2.7: CDMA Architecture (Source: Xilinx)[13]

The architecture of the engine is shown in figure 2.7 and it is composed of many sub-modules. The Register module is programmed from the CPU through the AXI4-Lite slave interface to configure the transfer's variables as it contains Control, Status, SrcAddr, DstAddr, etc. registers. In addition,

there is the AXI DataMover block which is responsible for the transfer of data with high throughput. DataMover offers 4 Kb address boundary protection with automatic burst partitioning, queuing of multiple transfer requests, and byte-level data realignment as it contains the Data Realignment Engine(DRE). DRE is an optional mechanism that can provide the capability for read and write transfers from any address byte offset as data are realigned to the byte level. The coordination of DataMover for the command loading, the update of status, and in general for the control of the system manages the DMA Controller module. Finally, there is the Scatter Gather Engine which is an optionally included module and it uses a dedicated AXI master interface so it can fetch, process, and update a transfer descriptor chain created by the software for off-loading CPU management tasks to hardware automation. The proposed IP Core is designed to improve the performance of embedded applications.

2.2.2 Design and Verification of Configurable Multichannel DMA controller

In this work, the authors Meet Dave, Santosh Jagtap describe the design of their multi-channel AMBA-based DMA controller for efficient and effective data transfers between processor and memory[14]. The project supports a configurable number of channels(although the design is presented with 4), different arbitration schemes, and uses asynchronous FIFOs which make the controller a flexible and versatile solution.

The structure of the system is composed of 4 channels, one priority arbiter, a descriptor, some registers for each channel, asynchronous FIFOs, address multiplexers, three AXI masters, and one AXI slave Fig.2.8. The first AXI master is associated with the AXI interface of its bus while the rest of the masters can be connected to external memory or DDR. The usual operation of the DMA is to transfer a big load of data from the channel with the highest priority and then it switches and serves the next priority channel. However, there is a second operation mode called the Descriptor transfer mode. In this case, the processor sends transfer's information to the descriptor buffer ahead of schedule. Subsequently, the Arbiter will select the channel with the highest priority, Descriptor's information will load in the work register and finally, the task will be executed. During the execution, the next transfer is loaded in the prefetch register and in this way the DMA completes continuously all the transmissions.

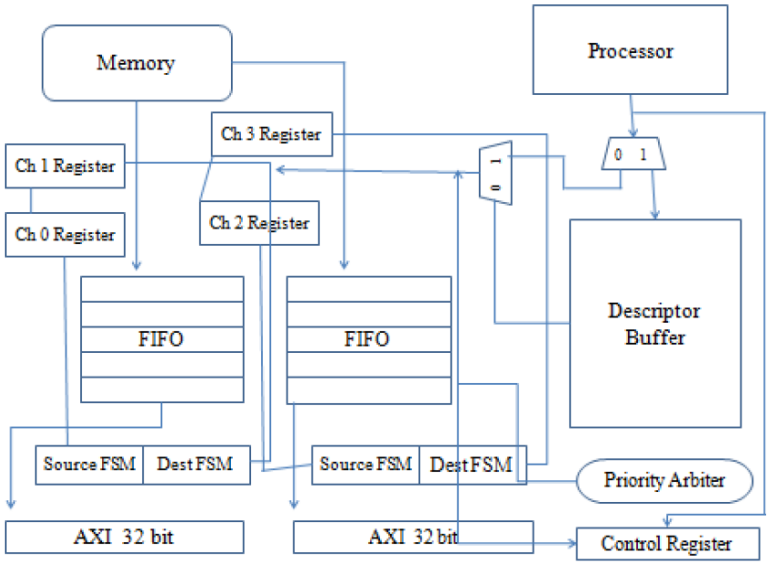

FIGURE 2.8: DMA Architecture (Source: Meet Dave and Santosh Jagtap)[14]

2.3 Power efficient DMA controllers

2.3.1 High Performance Low Power AHB DMA Controller with FSM Decomposition Technique

In this publication by Chetan Sharma and Dr. D. K. Chauhan the authors present the Design of a DMA controller and they try to optimize this design in terms of power consumption[15]. The technique used for the reduction of the total power consumption is called FSM decomposition and it splits the FSM into more and smaller FSMs to save the energy of the unused ones.

The architecture of this DMA engine is depicted in the figure 2.9 and it is designed to transfer data from memory to peripherals and vice versa. The DMA is constructed to comply with the AMBA AHB bus and for this reason, it implements the AHB communication protocol. Hence there is one AHB master and one AHB slave module. The AHB slave receives transfer requests from the processor while the AHB master transfers the data to/from memory. There are also 2 FIFOs, one for the data transmission from peripheral to AHB master and hence to memory and the second for transmission from memory to peripheral. In this way, the system succeeds to execute the data transfers and allows the processor to execute other tasks and enhance its performance.

The AHB DMA controller uses FSM components to schedule its operations. To reduce the power consumption of the controller the paper describes the decomposition of the FSM into 2 parts which are coupled with each other

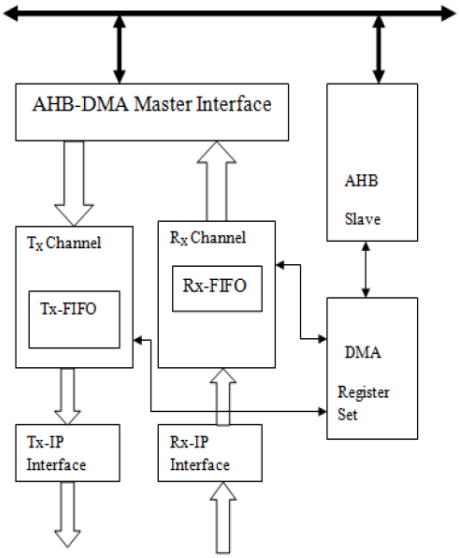

FIGURE 2.9: DMA Architecture (Source: Chetan Sharma)[15]

Fig.2.10. In this way, one of the two sub-machines will work most of the time while the other will be idle saving energy. The switching activity between sub-machine has a crucial role in power saving which will be minimized by the technique.

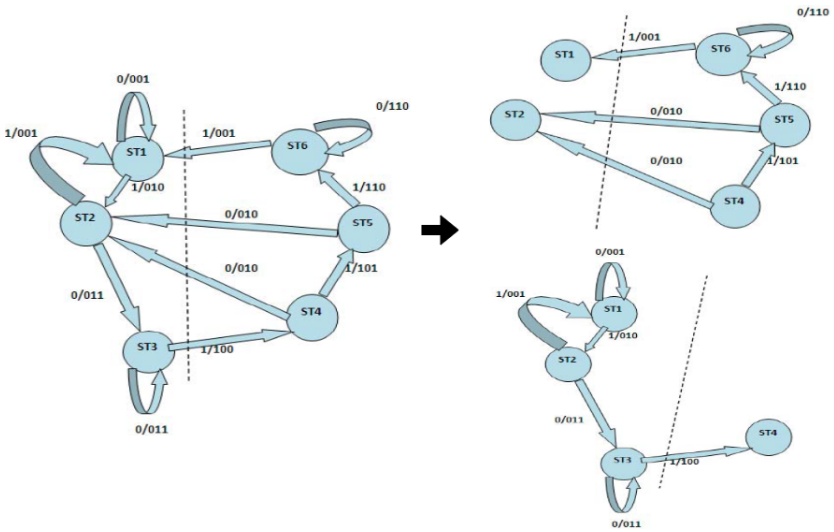

FIGURE 2.10: FSM decomposition (Source: Chetan Sharma)[15]

The simulation results show the new DMA controller saves 30% of the total power. The drawback of the controller is when both sub-FSM are working where the is no power benefit.

2.3.2 Design and implementation of Efficient Direct Memory Access (DMA) Controller in Multiprocessor SoC

This paper by authors Yasha Jyothi M Shirur, Kritika M Sharma, and Aishwarya published in 2018 and describes the design and implementation of a DMA that suits the MPSoC (Multiprocessor System on Chip) environment[1]. The authors describe that the area and power consumption in modern MPSoc is a major concern. For this reason, this approach performs area and power optimizations compare to the existing method which is also implemented.

The proposed DMA can execute data transfers related to both memory and peripherals. It also supports word or byte transfers, up to eight transfer channels with configurable priority, burst, cycle stealing, or transparent mode, and interrupt or polling transfers.

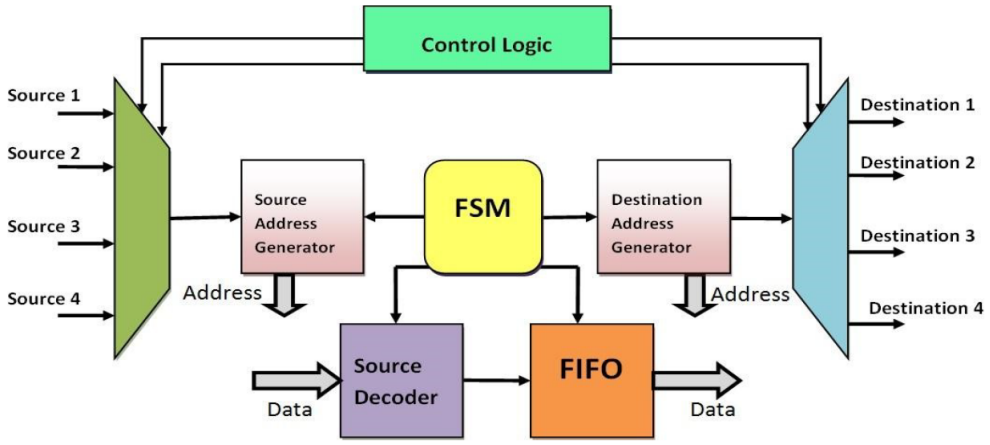

FIGURE 2.11: Block diagram of DMA (Source: Yasha Jyothi M Shirur)[1]

The DMA controller consists of 2 multiplexers, Source and Destination address generators, an FSM, a Source Decoder, a FIFO, and the control logic Fig.2.11. The muxes are used to select the source and destination. The control logic provides the starting source and destination addresses to the 2 address generators respectively and they produce the next addresses for the transfer by increasing the given addresses by a step they receive as input. Source Decoder configures the inserted data based on control signals received from FSM to give them the appropriate format. Some of the effects that the decoder can make on data are the type of endian and the choice of the number of bytes that will be used for the transfer. Finally, there is the FSM which is the heart of the system as it provides the necessary control signals for every stage of the transfer.

The paper, also describes the optimizations that were applied in the microarchitecture for some of the modules which impact the overall area and power consumption compared to the conventional method. The main improvements were the input bits reduction of the Source and Destination address generators' signals and the reduction of the number of multiplexers used by the Source decoder. According to the publication, the consumption of power reduced by 22% and are by 20%.

2.4 CHI projects

Although many projects and implementations of efficient DMA controllers have been introduced, as shown earlier, a publication about a DMA that uses the AMBA 5 CHI protocol could not be found. However, there are some publications about other devices that comply with the CHI protocol. Some of them are presented in this section.

2.4.1 Extending a modern RISC-V vector accelerator with direct access to the memory hierarchy through AMBA 5 CHI

This thesis produced by Roset Julia in 2022 participates in the improvement of the eProcessor of the EuroHPC project [16]. More specifically, this thesis aims to improve the RISC-V vector accelerator which is the co-design of eProcessor by changing the way it accesses the level 2 cache. In the previous version of the project, the accelerator could access the L2 cache with an OVI protocol through a scalar processor core and an NoC(Network on Chip) which introduces significant latency. To attack this problem this project achieved direct access to the NoC by the use of the AMBA 5 CHI protocol.

As shown in figure 2.12 the project implements an RN-I(IO coherent Request Node) which is responsible for generating and executing read and write CHI transactions with the L2 cache through the NoC. The Load/Store Unit of the system sends the appropriate information to the RN-I with a custom interface so it can read or write the desired data from the L2 cache. In this way, the memory state is preserved according to the RISC-V memory coherence model. Final results showed that the accelerator could successfully access the L2 cache replacing the old interface and improving the performance of the system.

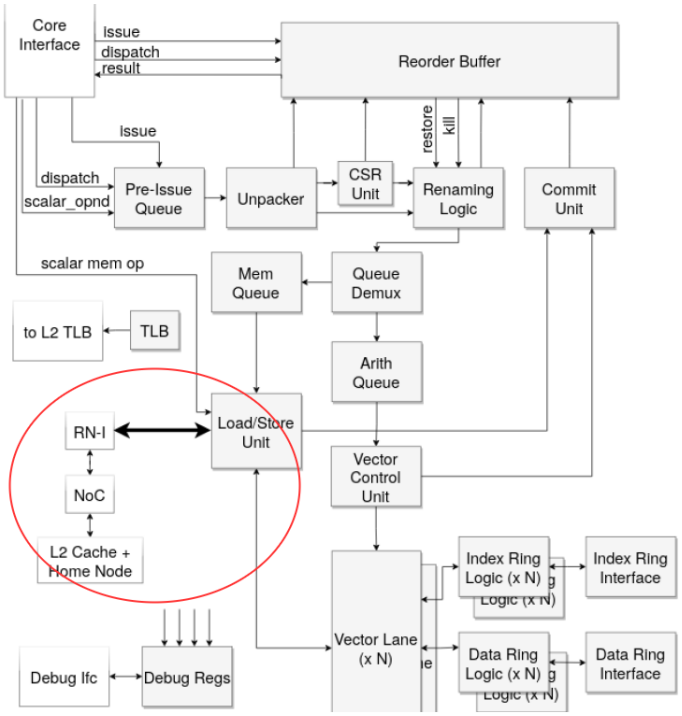

FIGURE 2.12: system block diagram (Source: Roset Julia)[16]

2.4.2 Design of an Open-Source Bridge Between Non-Coherent Burst-Based and Coherent Cache-Line-Based Memory Systems

This paper was published by authors Matheus Cavalcante, Andreas Kurth, Fabian Schuiki, and Luca Benini in 2020[17]. They describe that in heterogeneous computers CPU executes the serial part of the program while Programmable Manycore Accelerators (PMCAs) operate on the parallel sections which creates the need for efficient data sharing between these components. However, PMCAs are based on non-coherent memories where data is usually transferred by DMA controllers in burst mode. For this reason, they designed a bridge that connects burst-based non-coherent memory hierarchy with a coherent cache-line-based one. More specifically, this paper describes the design of the bridge that converts the AXI to CHI protocol Fig.2.13 as it connects a multi-core accelerator that uses the AXI protocol for its internal communication with a host CPU that uses CHI for coherent cache line accesses.

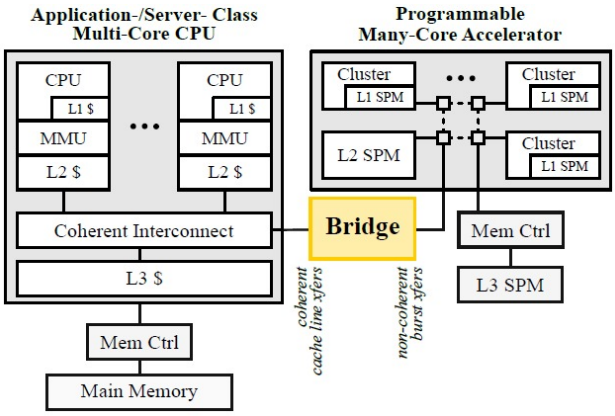

FIGURE 2.13: heterogeneous computer with the bridge highlighted (Source: Matheus Cavalcante)[17]

As shown in figure 2.14 the architecture contains several protocol translators which are simple FSMs that send requests and receive responses. The translators work in parallel and each one converts one AXI transaction to CHI. The generated CHI FLITs from all translators are multiplexed to satisfy the CHI protocol. The AXI transactions with the same AXID are handled by the same translator to preserve the order of the requests. In this case, the order of the responses must be the same and for this reason, Reorder Buffers(ROB) are employed to adjust the order of the responses.

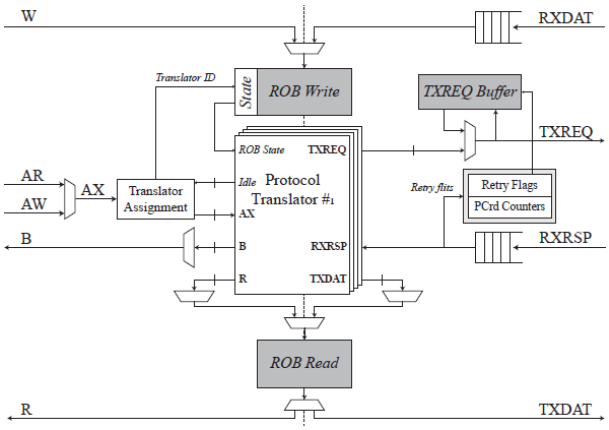

FIGURE 2.14: bridge architecture (Source: Matheus Cavalcante)[17]

The design was implemented and simulated successfully and the results showed that the bridge can achieve up to 97% of peak throughput over a wide range of realistic linear algebra kernels

2.4.3 Thesis Approach

Although there are many implementations of Direct Memory Accesses with a variety of communication protocol utilization and architectures as previously shown, a publication for an analytic description of a DMA engine that complies with the AMBA 5 CHI protocol could not be found. In addition, coherence has become a critical aspect of high-performance computing as without it HPC systems would not be able to deliver the computational power required for many complex applications. Also, Coherence Hub Interface is the most modern of the protocols introduced by AMBA and it offers high-performance coherence interconnection. For these reasons, and as to the best of our knowledge there are not any CHI-based DMAs, this work provides a detailed description of the design of an IP Core scalable DMA controller with generic scheduling, capability for reading and writing to memory at any address byte offset, and a CHI-Converter that makes the engine compliant with the AMBA 5 CHI interface. In comparison with the other DMA controllers except for the fact that our implementation uses the most modern and efficient communication protocol, it is more flexible than most of the other models as it can handle reliably and execute a configurable from the user number of independent memory transfers which most of the other engines can not. These features make the controller an appropriate component for many systems, as it can be easily adjusted for satisfying the demands of the application. The proposed DMA can be smoothly integrated into a bigger Coherent system as an IO Coherent node and efficiently execute the data transfers minimizing the workload on the processor and improving the overall system's performance.

Chapter 3

Proposed Architecture and Design

3.1 Architecture of DMA

This section presents the high-level architecture of the DMA system that was designed in this thesis. Initially, an overview of the necessary interface for interacting with the DMA Controller from both the processor and CHI perspective is provided. Subsequently, a comprehensive analytical description is presented, outlining the architecture and module arrangement within the system.

3.1.1 Interface of DMA

This DMA, as already described, is designed to transfer Data from one memory location to another by using the CHI communication protocol. The necessary procedure for the transferring of data begins when the appropriate instructions are communicated to the DMA. The DMA is able to handle many transfers at a time by accepting the instructions for the next transfer before the previous ones are finished. This is achieved by implementing a BRAM which holds the information for a memory transfer in each one of its addresses.

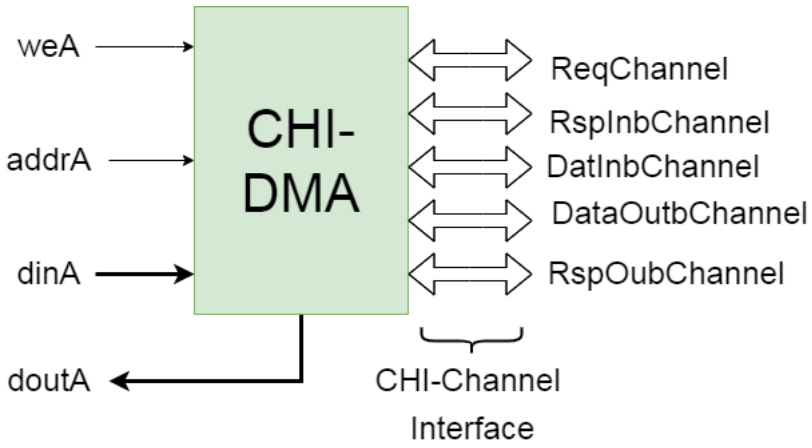

FIGURE 3.1: DMA Interface

The only signals which are needed for the unit which will assign the memory transfer to the DMA are the signals of a BRAM port. These signals are:

1. WE(input): Write Enable is an 8-bit signal. Each enabled bit of this signal indicates the corresponding field of the Data that will be written in DMA
2. ADDR(input): The Address signal indicates the position in BRAM where the instruction for the transfer will be stored
3. DataIn(input): This signal contains the information for the transfer, is composed of 8 fields and each field is being written in DMA when the corresponding WE bit is active
4. DataOut(output): This signal provides the information of a transfer that has been assigned to DMA at the given Address.

Address input is a 10-bit signal which allows up to $2^{10} = 1024$(as the size of the BRAM) different transfers to be handled at the same time by DMA but this number can be modified easily to fit the needs of the application as it is parameterized. In order to find an available address, the processor has to examine the Status of the address by reading BRAM. Data In and Out signals are 256-bit and contain all the information about a transfer. The fields of these Data are 32-bit wide and indicate (from the least significant bits): the address from where the data will be taken, the address where the data will be transmitted, the number of bytes for the transmission, the number of bytes that have been scheduled for transfer, the status of transfer and 3 unused fields 3.1.

	reserved	Status	Sent Bytes	Bytes To Send	DstAddr	SrcAddr
Description	unused	state of Descriptor	scheduled bytes	total number of bytes	address of destination	address of source

TABLE 3.1: fields of Descriptor

In order for the processor to assign a new transfer to the DMA it should first read the BRAM of the system by setting a valid value to the Address input. In the next cycle, the information about the transfer which is assigned at the given address will be received on the DataOut output. Subsequently, the processor has to check the value of the Status field of the received Data. In the case where the Status is in Idle state(value = 0) then the address that was given is free for writing a new transfer because if there was another transfer at this address it is finished. In the case where the Status is in Error state(value = 2) then the transfer that was written in the given address is finished but it didn't complete successfully and the processor can try to re-write it or ignore it and write a new transfer at this address. Finally, in the case where the Status is in Active State(value = 1) then there is a transfer in this address that is not finished yet and the processor must not modify the information of this transfer in order to be executed correctly, hence it has to search for another address. When the processor finds a free address to assign the next transfer it has to set the Status field active with the appropriate information of the transfer in order for DMA to start its execution. When DMA completes a FULL transfer, it changes the Status of this transfer from Active to Idle if data were moved normally or Error if there was a problem during the procedure. However, this system does not implement an interrupt line to inform the CPU about the completion of a transfer as the DMA processes multiple transfers at a time, and an interrupt request line is not scalable. For this reason, even if in some cases it is time-consuming, the only way for the processor to find out when a transfer has been finished is with the polling method by observing when the Status field of an assigned transfer switches from Active to Idle or Error.

For the transferring of Data, five channels are implemented as the CHI requires. These channels are connected with the CHI interconnect so the data can be read and written at the appropriate location of memory. The five channels are Request Channel, outbound Response and Data channels, and

inbound Response and Data channel Fig.3.2.

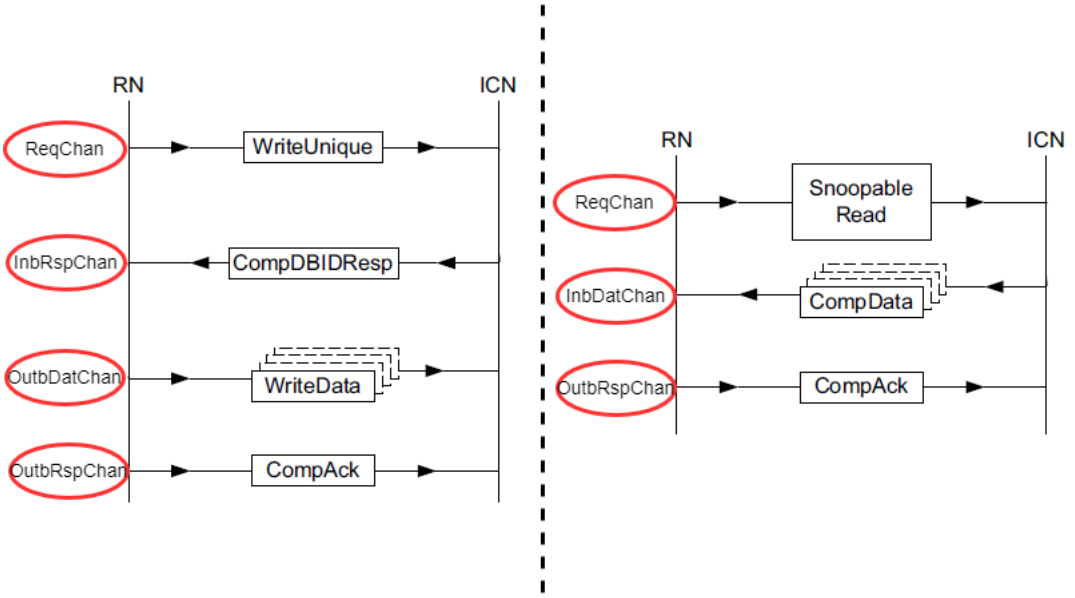

FIGURE 3.2: Channel Overview

DMA generates read and write requests for the CHI system with the Request channel, then the interconnect replies on the Data or Response channel with the necessary information, and finally DMA after processing the received data send them to the outbound Data channel for write. In this way, every transfer that was given by the processor is executed.

3.1.2 High-level design of DMA

The basic components of the DMA as seen in Figure 3.3 DescBRAM (BRAM), Scheduler, CHi-Converter(CHI-Conv) and a BarrelShifter. DescBRAM is composed of many Descriptors which is the main module that stores information for a transaction and every Descriptor is practically a different address of the BRAM. When the processor finds an empty Descriptor in BRAM, which can be found by reading it from one input/output port of BRAM that belongs exclusively to the processor, the procedure begins.

Firstly processor writes the Descriptor with the appropriate information for the transfer and at the same time the corresponding address pointer is written in the main FIFO of the system which contains all the addresses of non-scheduled Descriptors in order for them to be scheduled. Except for the processor, there is another module called Scheduler which needs access to the main FIFO. For this reason, an Arbiter is necessary for allowing only one

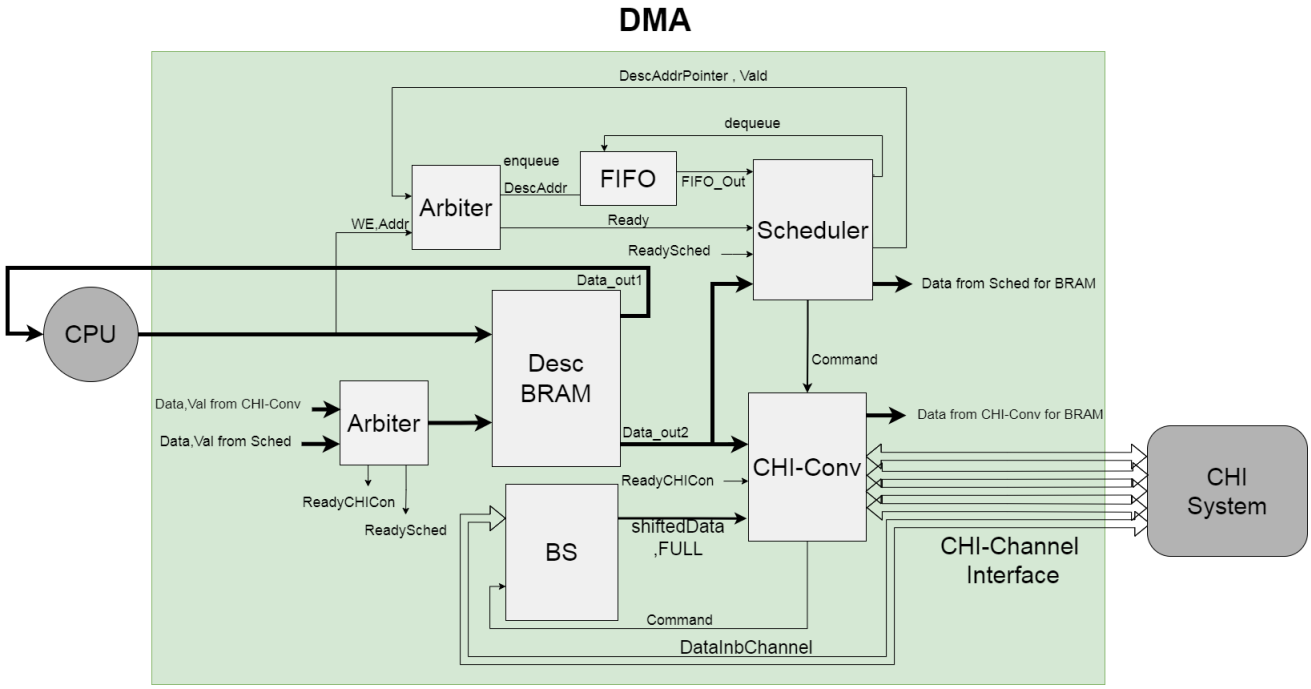

FIGURE 3.3: Architecture of DMA

component to access FIFO for writing at a time. This Arbiter gives always priority to the processor so the execution of its program won't slow down. With this design, the processor can write in DMA a new transfer at any time, even in consecutive cycles without being delayed. It is safe to assume that the hazard of the Arbiter never letting the Scheduler use FIFO because the processor is writing in every cycle will never happen because a real processor would not do that for a large period of time. Even if this bad scenario could occur, the engine would not stop working it would just slow down and it would finish the operations after BRAM would be full and the processor would not have an empty Descriptor to write in.

When FIFO is not empty Scheduler starts its operation by reading the Descriptor's data, through the second input/output port of DescBRAM, of the address pointer which is stored in FIFO. When the Descriptor's data have been read Scheduler passes the appropriate information about a part of the Descriptor's transfer to the CHI-Converter which is responsible to execute the proper CHI transactions. While Scheduler passes the information to CHI-Converter it also updates the Descriptor's stored information about how many bytes are scheduled. The amount of bytes for transfer that will be sent to CHI-Converter for execution is a generic value determined by the user who implements the DMA by a parameter called Chunk. When this Chunk

of transactions has been scheduled, Scheduler must schedule the next Descriptor and leave the previous one for later scheduling. This is achieved by dequeuing the first pointer from the FIFO, re-enqueuing it back in the queue, and obtaining the new pointer. This operation is possible only after the Arbiter responds to the request of the Scheduler to access the main FIFO. If the Arbiter can't give the Scheduler access to the main FIFO because the processor is writing it too then Scheduler has to wait until the processor is finished and Arbiter changes the occupation of FIFO. With this procedure, every active Descriptor is served equally, and a Descriptor with a very small transaction at the end of the queue won't need to wait a very long time in cases where are many addresses of Descriptors with huge transactions in front of it in FIFO. When the Scheduler schedules all transactions for a Descriptor then it removes its address pointer of the start of FIFO without waiting to obtain access from the Arbiter because there is no need to rewrite the pointer at the end of the queue and keep servicing the next Descriptor.

When a CHI-Read have been sent by CHI-Converter and the Read-Response arrives then the data enters as an input to the BarrelShifter module where they are getting shifted by a proper amount of bytes and then they are driven to the CHI-Converter where a CHI-Write transaction will begin. When all CHI-Writes of a Descriptor have been finished and there are no other transactions for this Descriptor to be made then the Status register of the Descriptor must be set to Idle so it can be recognized as a finished Descriptor and the processor can be informed for the end of the transfer and rewrite the Descriptor with the next task if needed. This status update is initiated by CHI-Converter and happens from the second input port of BRAM since the first input port belongs exclusively to the processor. Although the second input port is already used by Scheduler to update some of the Descriptor's fields while scheduling them, as previously mentioned in this section, so a second Arbiter has to be used to allow only one of the updates to happen at a time because the updates would probably be for different Descriptors and should be only two input ports in the BRAM.

3.2 Design of Components

3.2.1 Descriptor, BRAM

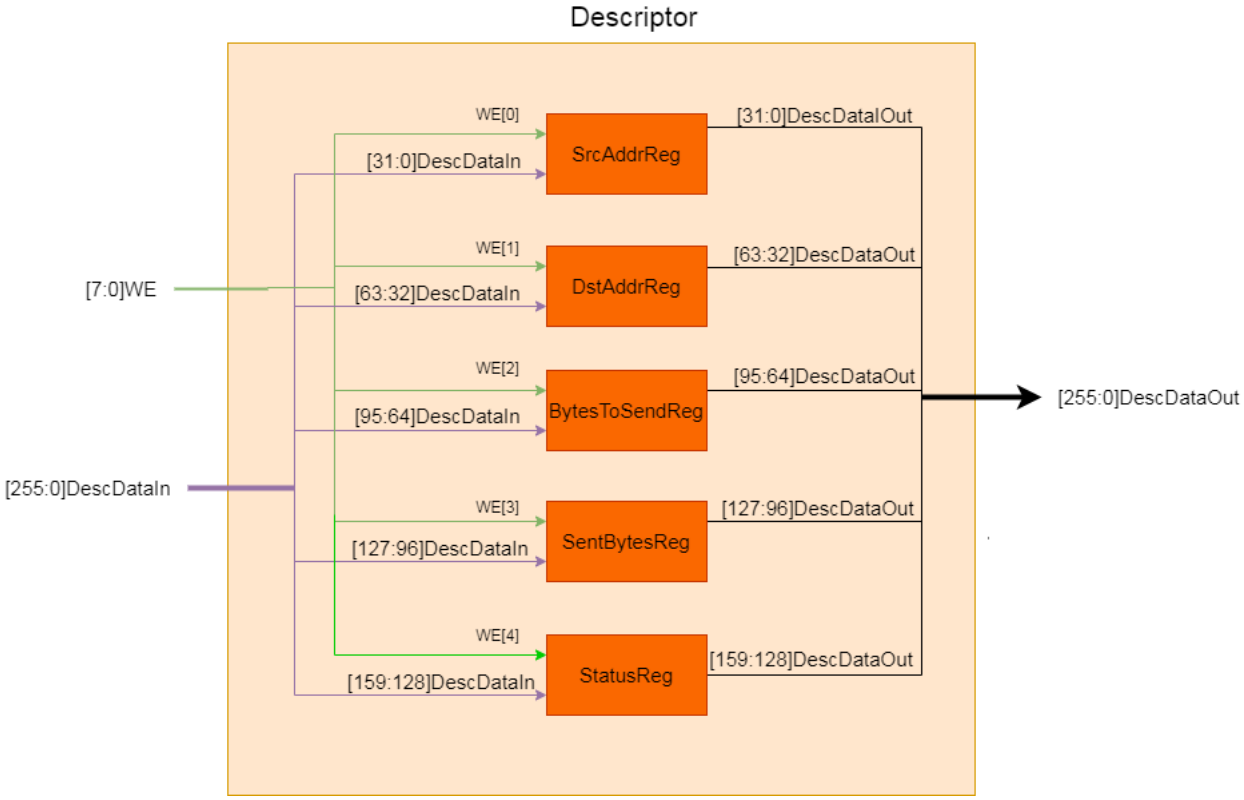

FIGURE 3.4: Descriptor

Descriptor is the main module that stores information for one memory transfer. There are two input signals inserted in Descriptor: WE which is 8-bit wide, DataIn which is 256-bit package, and one output: DataOut which is also 256-bit Fig.3.4. Descriptor is composed of 8 32-bit wide registers: Source Address (SrcAddr) that keeps the memory address from where data should be read, Destination Address (DstAddr) that keeps the memory address where read data should be written, Bytes To Send (BTS) that keeps the number of bytes that should be transferred from the Source to Destination address, SentBytes (SB) that keeps the number of bytes that have been already transferred(or at least they have been scheduled to be transferred), Status that keeps the state of the Descriptor which can be idle, active or error, etc. and 3 more registers which are reserved and have no use so they are not present in the image. The reserved registers are needed in Descriptor so its width will be a power of 2 and it can represent a memory line (of BRAM). Each register is written when the corresponding bit of input WE is set and

the value that takes is the corresponding 32-bit from DataIn package. For example, SrcAddr takes the first 32 bits of DataIn DstAddrReg takes the second 32 bits which are from bit 32 to 63 and so on. Finally, all output bits of every register are getting merged to a 256-bit package with the same logic as the input DataIn and create the output of the module DataOut. It is worth noting that the signals Data in and out are 256-bit wide but the useful information that one Descriptor stores is $32 \cdot 5 = 160$ bit as the last 3 registers which stores the most significant $256 - 160 = 96$ bits are not used. The Descriptor should be written from the processor only when the status register is in idle or error state and when the Descriptor is written and Status changes to active state, then it shouldn't be modified until all its transfer is over and status changes state again. SentBytes field will be updated from the Scheduler every time a number of bytes will be sent (or scheduled) and when all bytes are scheduled and the full transfer of the Descriptor has been truly finished then the idle value will be written to StatusReg to make the Descriptor available for use from processor again. If something goes wrong with the procedure, then the error value should be written in the status and the problem should be handled appropriately.

DescBRAM:

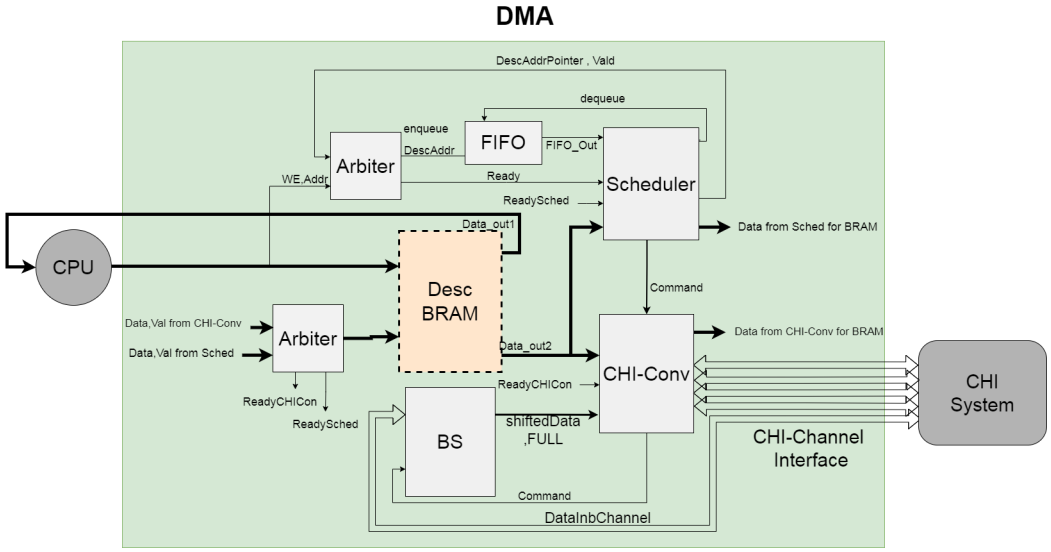

FIGURE 3.5: Location of BRAM in the Design

To improve the system and allow the DMA to handle more than one transfer at a time it is necessary to implement a structure that contains more than one Descriptor for holding the information for many transfers. As the number of Descriptors increases this register space structure which must classify

and address all of the Descriptors tends to become a BRAM Fig.3.6. For this reason, in this DMA instead of having a custom module with a big number of different Descriptors, the system implements a DescBRAM which has the same use. Each row(Address) of DescBRAM represents 1 Descriptor. Each row of BRAM consists of many columns that represent the corresponding registers of Descriptor. More specifically each row of BRAM has 8 columns of 32 bits each, which makes the row width 256 bits. BRAM has 2 ports for read-write, one that belongs exclusively to the processor so it can read the Descriptors and write the appropriate transactions in them, and one for the components of the DMA. Each port of BRAM is composed of 3 inputs: WE which is 8 bits and decides which columns of the Descriptor will be written, the Address which determines the chosen Descriptor, and 256 bits Data packet with the appropriate fields for the Descriptor. In the same way, as Descriptor was, Data packet's fields should be :

1. SrcAddr: Source Address of transaction
2. DstAddr: Destination Address of transaction
3. BytesToSend: transaction's number of bytes should be transferred
4. SentBytes: Number of already sent bytes
5. Status: State of Descriptor (Idle, Active, Error, etc)
6. Reserved: 3 more fields that are reserved and are not used

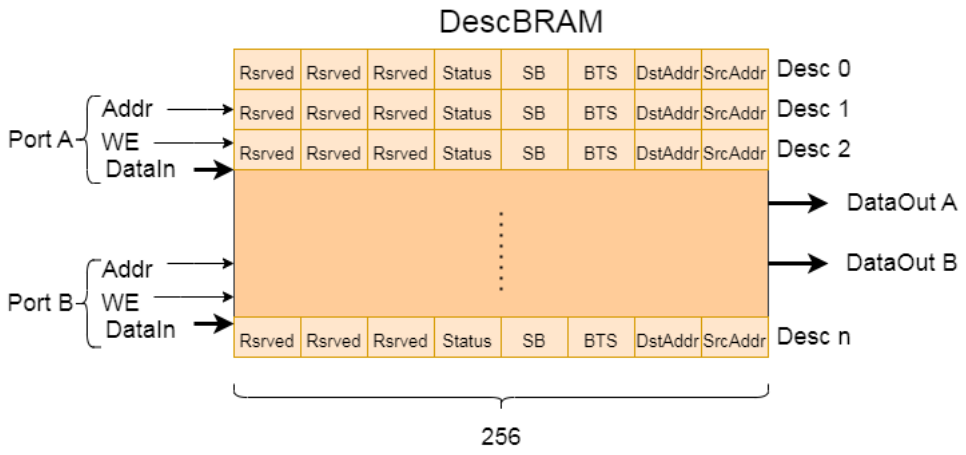

FIGURE 3.6: DescBRAM

Every time Processor needs to make a memory transfer it should read DescBRAM from DMA and when it finds a Descriptor with the Status field to be Idle or Error it can write the desired transfer in this Descriptor Fig.3.5.

When a Descriptor of DescBRAM is written, the appropriate input signals are driven to the top Arbiter which gives them priority, and hence the address pointer is written in the main FIFO which will initiate the scheduling for the written Descriptor and its transfer will be executed. The Hazard of both ports writing the same Descriptor is ignored because it will never happen as long as the processor won't write a Descriptor with an active status. DMA's component will not modify any Descriptor with idle or error status so it is the processor's responsibility not to update an active Descriptor and cause such a hazard in order for the transfer of this Descriptor to be executed normally.

3.2.2 Scheduler

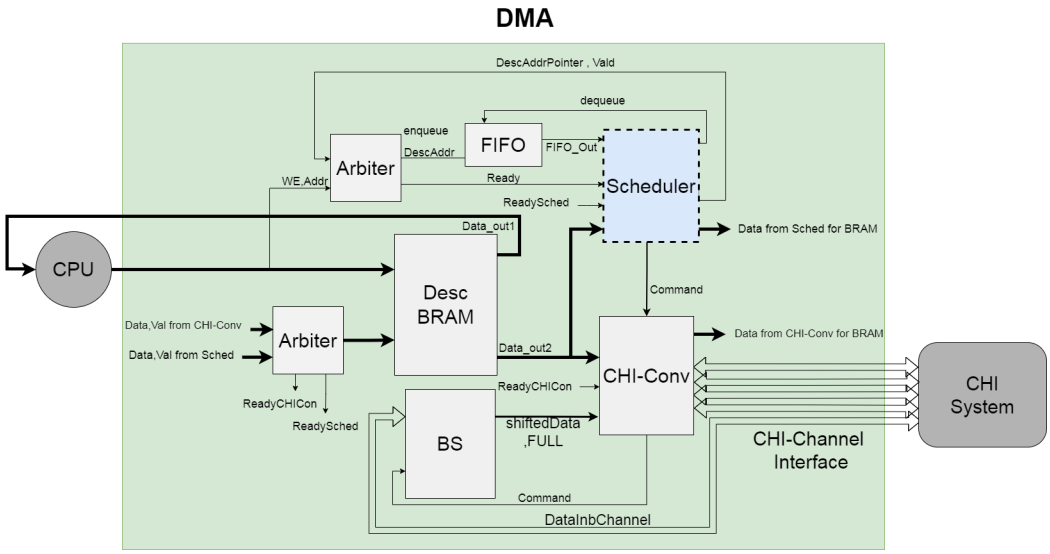

FIGURE 3.7: Location of Scheduler in the Design

Scheduler is responsible to manage the service order and the number of transactions that will be scheduled for each ready Descriptor. The process of Scheduler is to read the Descriptors from BRAM and pass the appropriate information to CHI-Converter as shown in Fig. 3.7. Its operation is important for making the transactions of active descriptors be executed equally by scheduling a part of each Descriptor's transfer at a time. To accomplish this operation the scheduler is composed of an FSM (Finite State Machine) which is the component that produces the necessary control and output signals, one register to keep the address of the Descriptor read from BRAM, and some combinational logic Fig.3.8.

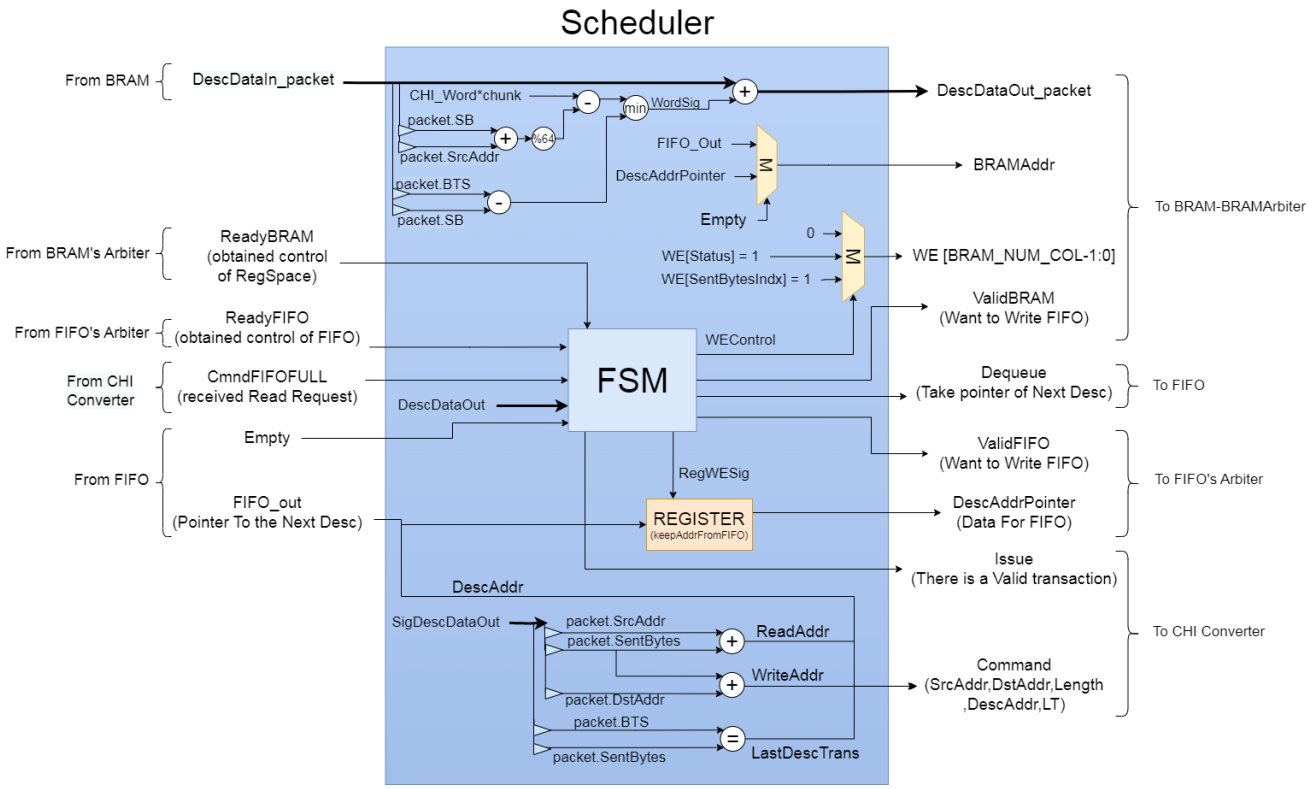

FIGURE 3.8: Scheduler

Scheduler has six inputs which are: one Data packet that is 256 bit, which comes from BRAM and contains all the information for one Descriptor, two Ready signals that come from the two Arbiters (BRAM's Arbiter and FIFO's Arbiter), and inform if there is access through Arbiter to the corresponding module when access is requested, a CmndFIFOFULL signal that comes from a FIFO placed in CHI-converter module and when is set indicates that CHI-converter can not receive more instructions from scheduler hence there is no more space to schedule other transaction and two signals from the main FIFO that keeps the address pointer of Descriptors which are an Empty signal and the address pointer.

For the first output, the number of bytes that will be scheduled is added to the SentBytes field of the Data packet that came from BRAM and it constructs a new packet that goes back to BRAM and will update the corresponding Descriptor. The number of bytes which are added to the input Data packet and scheduled is depending on the parameter Chunk which is the number of desired CHI transactions. If Descriptor has enough bytes then the number of bytes that will be scheduled and be added to the SentBytes field of DataIn packet is $Chunk \cdot ChiDataWidth$(which is 64 bytes in this case) else it is all of the remaining bytes of Descriptor $BytesToSend - SentBytes$. In the case

where it is the first chunk that will be scheduled from a Descriptor so its SentBytes field is zero then instead of scheduling $Chunk \cdot ChiDataWidth$ bytes, Scheduler sends $Chunk \cdot ChiDataWidth - (SrcAddr \mod ChiDataWidth)$ bytes. This is an optimization so the module that will make the transactions(CHI-Converter) will never need to read the same memory line twice for the same Descriptor. This is happening because the source address is probably misaligned compared to the data pieces that will be read from the memory and by subtracting the misalignment of SrcAddr the last read transaction of the chunk will be finished on an aligned address in memory which prevents the reading of the same memory line again from the next chunk of the same Descriptor. The representation of the optimization can be seen in Fig.3.9

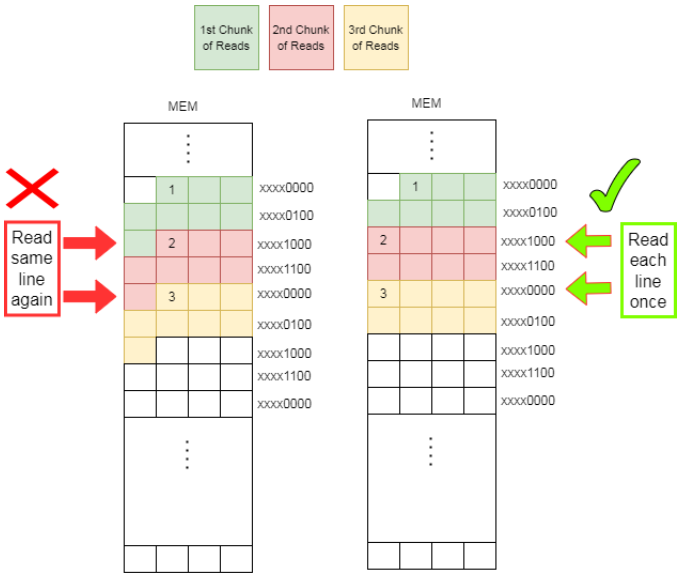

FIGURE 3.9: optimized scheduling

This way Scheduler provides a generic way of scheduling based on the parameter Chunk that can adapt to the needs of the application. To ensure optimal system performance, it is essential to set a sufficiently large value for the Chunk parameter. This ensures that the CHI-Converter always has the opportunity to initiate a new CHI transaction until the Scheduler proceeds to schedule the next command(a chunk of transactions).

Furthermore, there are two more outputs signals that go to BRAM which are Write Enable (WE) which is controlled by the FSM and will enable the necessary bytes for the update, and the BRAMAddress which is the address obtained from FIFO, and which is the position of Descriptor in BRAM. In order to connect the above signals to BRAM it is necessary to get access from BRAM's Arbiter which is requested by a Valid output signal controlled by

FSM. When this Valid signal and the corresponding Ready signal from Arbiter are active then all the appropriate signals are connected to BRAM and they will read and write it on the next cycle. FSM is also controlling the output signals: Dequeue that ejects the first Descriptor address pointer from the FIFO when a chunk for a Descriptor has been scheduled, a Valid signal which requests control from Arbiter to this FIFO when is set in order to enqueue back the address pointer if needed, and an Issue signal which indicates that there is a valid Chunk of transactions ready for CHI-Converter. The operation of the FSM and how it produces the necessary signals is described below Fig.3.10.

Another necessary action is to store the address pointer obtained by the input signal that comes from FIFO inside a register. This is happening every time the FSM set a Write Enable signal for this register. This action is important because there should be a way to remember the address pointer, so it can be written back to FIFO when a dequeue operation happens and this address pointer is ejected from the FIFO. Also in the case where there is only one pointer in FIFO then the next address that will be used for reading BRAM is the same address pointer which is dequeued, stored in the register, and will be written back to FIFO so Scheduler uses it directly from the register for read BRAM to avoid one cycle delay instead of writing it to FIFO, reading it again and then read from BRAM.

Finally, the last thing Scheduler does is to pass the right information to CHI-Converter so every transaction can be executed. This is accomplished by using the input DataIn that comes from BRAM and creating the fields: $ReadAddress = SrcAddr + SentBytes$, $WriteAddress = DstAddr + SentBytes$, $Length = Chunk \cdot ChiDataWord$ (64) or $Chunk \cdot ChiDataWidth - SrcAddr \mod ChiDataWidth$(if possible), $DescAddr$ and $LastDescTrans$ that composes the command packet.

- ReadAddress: calculated by $SrcAddr + SentBytes$
- WriteAddress: calculated by $DstAddr + SentBytes$
- Length: $calculatedbyChunk \cdot ChiDataWord$ (64) or $Chunk \cdot ChiDataWidth - SrcAddr \mod ChiDataWidth$(if possible)
- DescAddr: is the address of Descriptor in BRAM
- LastDescTrans: indicates the last chunk of a Descriptor

This is the required information for a chunk of CHI transactions to be executed which are valid when the Issue output is set. The 2 last fields of

the command(LastDescTrans and DescAddr) are necessary to be configured when the upcoming transaction to be scheduled represents the final transaction of a Descriptor. In this way, CHI-Converter can update the status field of the appropriate Descriptor.

FSM of Scheduler:

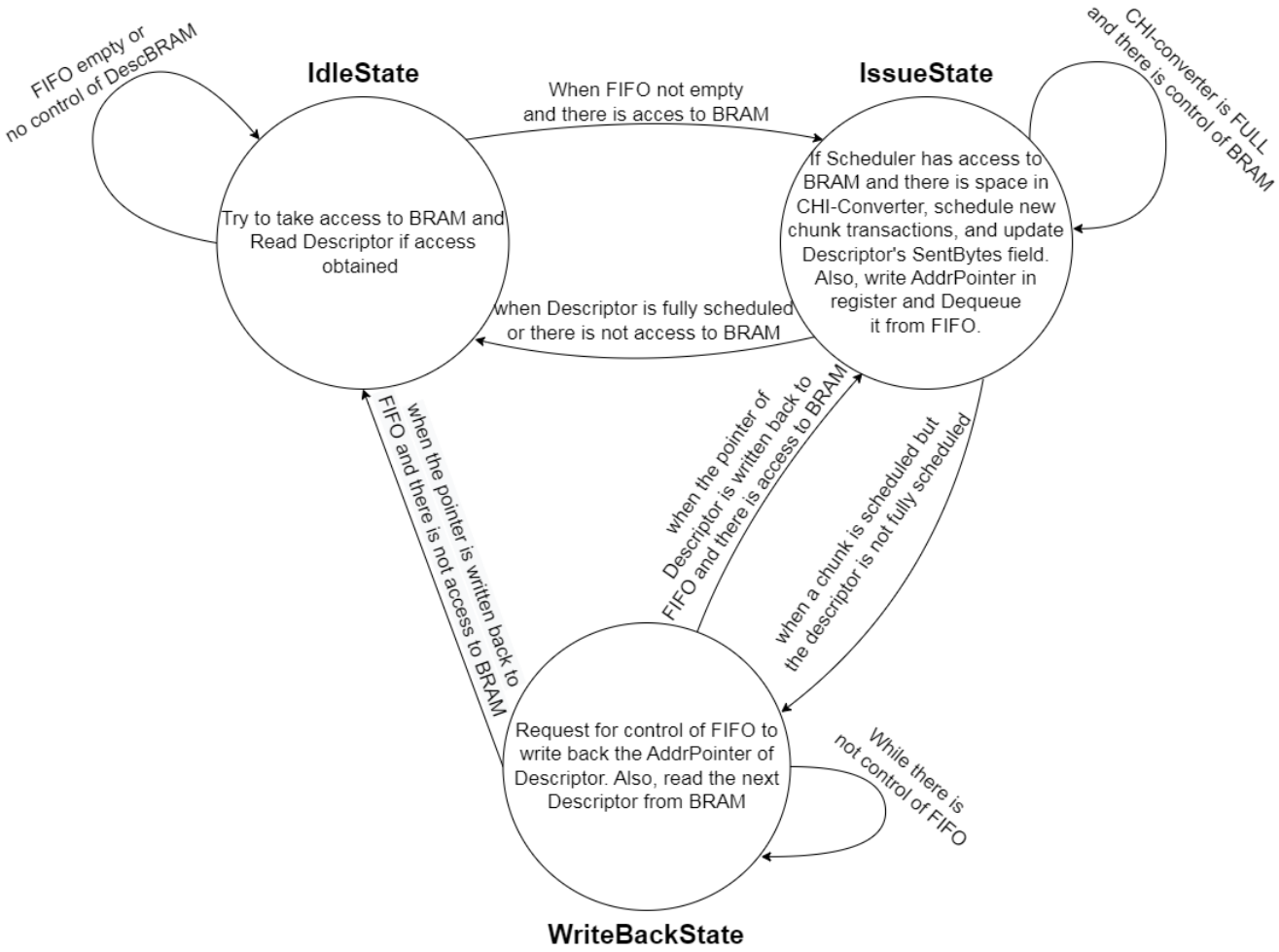

FIGURE 3.10: FSM of Scheduler

Scheduler's FSM is a mealy state machine, as it provides outputs that depend on the state and the inputs. There are three states in this FSM: IdleState, IssueState, and WriteBackState Fig.3.10. At the beginning of the procedure, the state is set to IdleState. In IdleState basically, FSM is doing nothing by keeping all its output to zero as long as the input signal Empty is on, so there are no ready Descriptors to be served. When the Empty input gets low, which indicates that there is an address pointer in FIFO, the FSM requests control of BRAM from Arbiter by setting the output signal ValidBRAM, so

the appropriate data can be read. As long as the input signal ReadyBRAM is low, there is no access to BRAM yet, so the FSM stays in IdleState and keeps the ValidBRAM on. When the ready input from BRAM is activated then the access has been obtained and on the positive edge of the next clock cycle, the Data are read and the state transitions to IssueState.

In IssueState FSM sends a new command that is a number Chunk of CHI-transactions when possible to be scheduled and updates the appropriate Descriptor in BRAM when the appropriate conditions are met. In this state, firstly the signal WE for the register is activated, and the address pointer is written to the register. When the input signal CmndFIFOFULL from CHI-converter is set then there is no space to schedule extra transactions so the FSM is just waiting. When the input signal FULL is off and there is the control of BRAM(ReadyBRAM set) then a command which is Chunk transactions can be scheduled so the FSM enables the output signal Issue and controls the WE output signal to take the right value for updating BRAM. If the control of BRAM is lost, the state changes to IdleState until control is regained. In case when there is control of BRAM, input FULL is off and the remaining bytes are the last command for the Descriptor then the Dequeue signal is enabled to remove the first element of FIFO, and as there is no reason for the address pointer to be written back in FIFO the state changes to IdleState.

When the state is WriteBackState Scheduler tries to re-enqueue the address pointer of Descriptor to FIFO because there are more transactions to be scheduled later. FSM firstly deactivates all signals except ValidBRAM which request access to BRAM and the ValidFIFO output signal which requests access to FIFO to re-enqueue the address pointer. As long as access to FIFO cannot be obtained, the FSM is waiting in WriteBackState. Subsequently, when the ReadyFIFO input signal is enabled which means that the control of FIFO has been obtained then the address pointer is written back to FIFO and if there is still access to BRAM then the state changes to IssueState and FSM start scheduling new transaction otherwise the state changes to IdleState where it will stay until control of BRAM is re-obtained.

The optimal Chunk (the number of scheduled transactions each time) in order for the DMA to operate in the faster possible way depends on the nature of this FSM. The Chunk should be big enough so the CHI-Converter will always have enough work to do until the next command gets scheduled. In this case, the FSM needs at least 2 cycles to return from the IssuState back to IssuState to schedule the next command and to keep CHI-Converter busy

needs a Chunk bigger than 1. Also, there is the case where the control of BRAM or FIFO is lost for a few cycles because the corresponding Arbiter gives priority to some other module and FSM is late to return back to the IssuState which creates the need for a bigger Chunk to accomplish the optimal scheduling. If the value of Chunk, with this system, is greater than 4 – 5 then FSM will always have more than enough time to schedule the next command before CHI-Converter finishes all the requests for the transactions. If in an unfortunate case, many Descriptors have very small transactions then the scheduler will not be able to schedule Chunk transactions, and it will be forced to schedule as many transactions as there are in each Descriptor at a time that can result in CHI-Converter waiting for Scheduler for the next command which can make the system slower. This problem can be solved by changing the design of the DMA and instead of having arbiters, use a different structure that would store the Status of every Descriptor in a Different component and wouldn't force the modules of the system to conflict on BRAM's Arbiter as they both need to use the same ports of BRAM which can be an optimization for future work. For the current design, the throughput of transferring data from Scheduler to CHI-Converter and hence of the overall system is determined from the size of transactions in Descriptors if the value of the Chunk is big enough for optimal scheduling.

3.2.3 CHI-Protocol

In this section, we provide an overview of the CHI architecture specifications. This overview aims to facilitate a better understanding of the subsequent module designs described in the following sections.

CHI protocol[18] introduces 3 types of nodes :

1. Request node (RN): Generates protocol transactions, including reads and writes to the interconnect (CHI system)
2. Home node (HN): Node located within the interconnect that receives protocol transactions from RNs, schedules them and drives them to the correct Slave node
3. Slave node (SN): An SN receives a request from an HN, completes the required action and returns a response.

This DMA is a Request node, since it has to generate new requests to transfer data from one memory location to another by reading and writing them to

memory. Every type of Node can be further divided into some categories. A Request node can be :

- RN-F Fully coherent Request Node: Must communicate via snoop channel with other request nodes to update data when a change happens at shared address location
  - Includes a hardware-coherent cache
  - Permitted to generate all transactions defined by the protocol
  - Supports all Snoop transactions.
- RN-D IO coherent Request Node with DVM (Distributed Virtual Memory) support:
  - Does not include a hardware-coherent cache.
  - Receives DVM transactions.
  - Generates a subset of transactions defined by the protocol.
- RN-I IO coherent Request Node: Can read and write the last snapshot of memory and force the other Request nodes to update their Data, but it won't answer on snoops or be informed of changes to Data by other Requesters.
  - Does not include a hardware-coherent cache.
  - Does not receive DVM transactions.
  - Generates a subset of transactions defined by the protocol.
  - Does not require snoop functionality.

The Designed DMA is an IO coherent Request Node (RN-I) which means that it won't communicate directly with other Request nodes. It will communicate with the interconnect to read the last snapshot of memory, and the interconnect is responsible to find which node has valid data of the requested address and send them to DMA. The CHI channels that DMA needs to implement are:

1. Request Channel: Begin a new read or write Request
2. Outbound Response Channel: Response to received Data or another Response
3. Outbound Data Channel: Send Data for write

4. Inbound Response Channel: Receive Response for a corresponding Request
5. Inbound Data Channel: Receive Data for a Read Request

It is worth noting that in general there is one more inbound channel called Snoop channel, and it is used for Request nodes to update their cashes when a change happens on its Data from other Request nodes. However, the Snoop channel is not implemented in this case because this is an IO-coherent and not a fully coherent DMA which means that it does not integrate a cash or needs feedback for data changes so Snoop transactions are not supported.

Every CHI channel consists of 4 signals :

1. FLITPEND: One-bit signal, indicates that a FLIT will be sent on the next cycle.
2. FLITV: One-bit signal, indicates that data are valid and should be received.
3. FLIT: Many bits vector, consists of many fields that contain the information of the channel(Request, Response, Data)
4. LCRDV: One-bit inverse signal, can be asserted by the entity of the opposite side of the channel's direction and indicates that the receiver is ready to accept one more FLIT

An example of a CHI channel is shown in figure Fig.3.11 that represents the Request Channel :

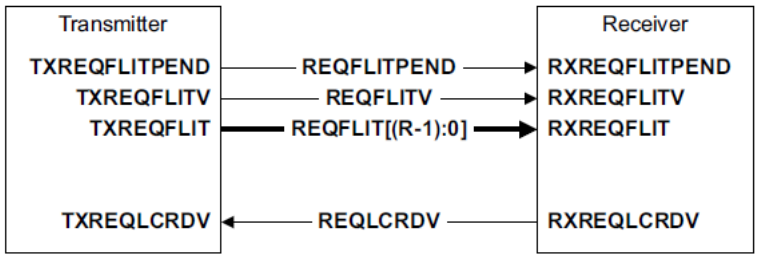

FIGURE 3.11: Request Channel

The signal FLITPEND is used to prepare the receiver to wait for the next FLIT on the next cycle, but it is not obligatory for the system to operate normally

since it can receive the corresponding FLIT just by observing the FLITV signal. In this DMA there is not any performance improvement that could possibly happen with the use of the FLITPEND signal and for this reason, this signal is always 0 (inactive).

Every channel has a vector of signals called FLIT. FLIT is the information of the transaction. There are 3 types of FLITs :

- REQFLIT: Contains all the information about a Request such as if it's a Read or Write request(opcode), Addr, Size, etc.
- RSPFLIT: Contains information about the Response to a Request such as type (opcode), if there is an error or the message has been received successfully etc.
- DATFLIT: It is the largest FLIT. Contains the requested Data, what type of Data is it(opcode), if there is an error etc.

All FLITs have a field that is called TxnID (Transaction ID). CHI requires that every transaction has a unique TxnID. TxnID is an 8-bit field that allows every Request node to do up to 256 outstanding transactions. If a Request node generated all 256 transactions without receiving any Response from the interconnect, then it has to wait for a response in order to reuse the finished TxnID. The Responder has the responsibility to use the same TxnID for the response with the request to allow the Request node to know where this response is referring to and which TxnID it can reuse.

CHI is a credit-based protocol. In credit-based protocols, every node can generate as many transactions as the number of credits they have received. In CHI, for example, every channel has an LCRDV signal which indicates when credit is given for the corresponding channel. Every cycle that the LCRDV signal is on, the owner of the channel gains one more credit, and it has the responsibility to count the number of received credits on each channel and decrease this number when using one credit by sending a FLIT with the activation of the signal FLITV. When a channel has 0 credits then it is not possible for the node to send a FLIT on this channel, and it is not possible to use a credit from a different channel, so it has to wait until it receives a new credit on this specific channel. In general, there is not a specific number of Credits that can be sent in a credit-based protocol, but CHI doesn't allow more than 15 Credits to be sent on each channel.

Every Flit of CHI has a field called opcode which determines the type of transaction. For example, the CHI supports 38 different opcodes for Requests channel, 11 for Response channel, and 10 for Data channel. This DMA needs to do 2 types of Requests: Read and Write. More specifically, it has to do a read of the last snapshot of memory and a write that will inform every node which keeps data from the written address that data have been modified. This is succeeded by the use of the opcodes: ReadOnce(0×03) for read requests and WriteUniquePtl(0×18) for write requests. For those requests, there are a few possible sequences that could happen. Some of them are described below :

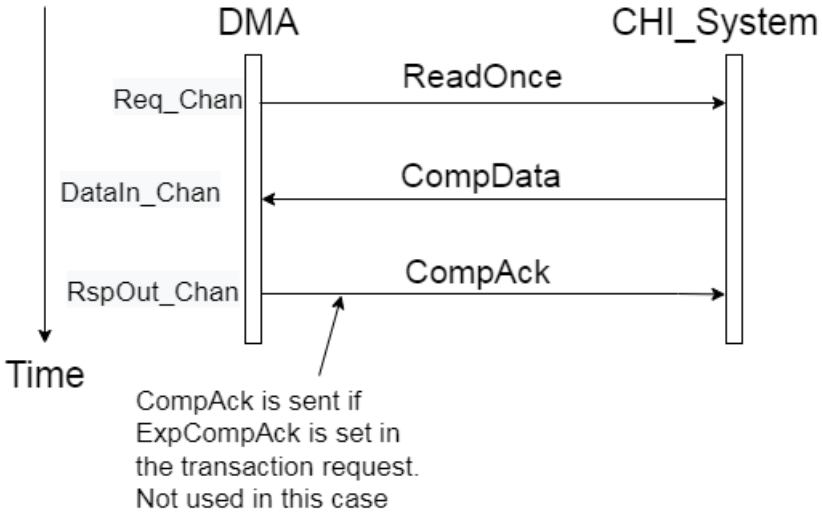

FIGURE 3.12: Read Transaction

Read Transaction Fig.3.12:

1. Firstly when there is at least 1 credit on the Request channel then the Request node generates a read request with: opcode ReadOnce, a unique TxnID, the required address, ExpCompAck field (which indicates if there will be an extra acknowledgment response from requester) etc. The ReadOnce opcode is used because DMA is an IO-CoherentNode and ReadOnce transaction is used to read the last snapshot of a Memory Address

2. After some time when the requested data are ready and at least 1 credit has been sent from the requester on the Data channel, the interconnect sends the Data Response FLIT with opcode CompData, TxnID the same

with the request, the corresponding data, the DBID field which the Requester must use as a TxnID field if it will send a CompAck Response, and other fields.

3. Finally, after the Data have been received, the requester must send a Response with opcode CompAck on the Response channel to indicate that Data was received normally if the request that happened had the ExpCompAck field on. If this field is 0 when the request is sent, then the responder does not wait for CompAck Response and the transaction is over when Data is received by the Request node. If the Request node has active ExpCompAck on the request and will send CompAck Response, then the TxnID of the response must be the same as the DBID field that was received previously in order for the interconnect to understand which transaction this CompAck is referring to.

In this DMA the ExpCompAck is always off in Read request, hence the CompAck Response is not sent because there is not any performance difference.

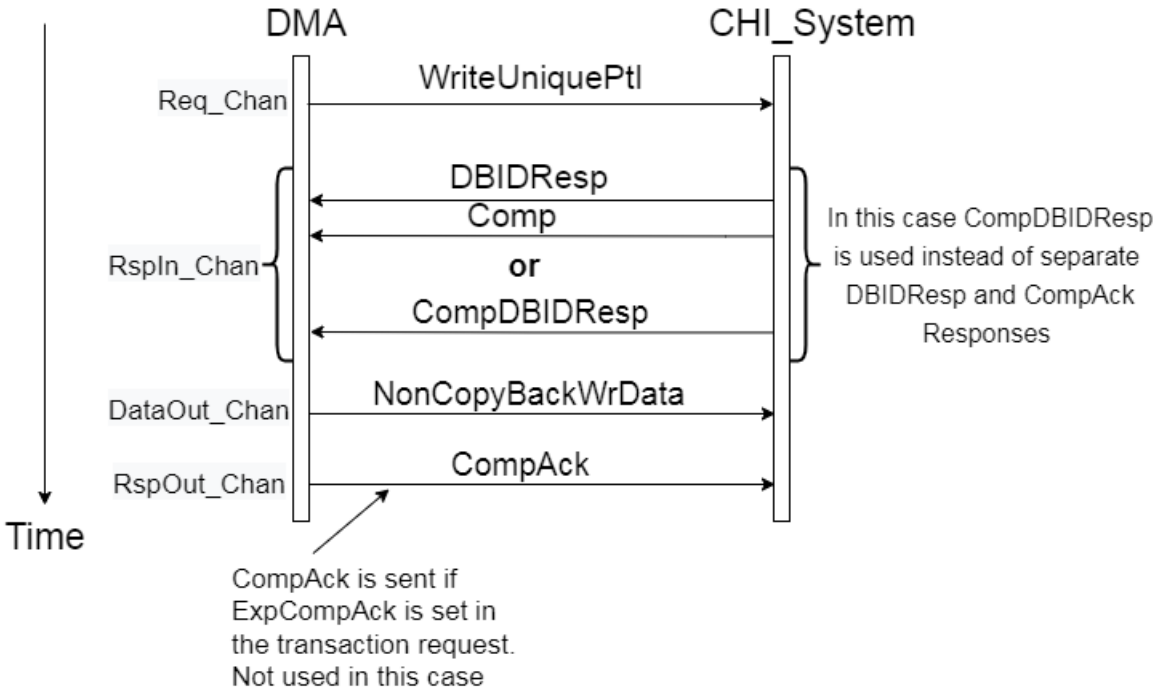

FIGURE 3.13: Write Transaction

Write Transaction Fig.3.13:

1. Firstly when there is at least 1 credit on the Request channel then the Request node generates a write request with opcode WriteUniqueuePtl,

a unique TxnID, the required address, ExpCompAck field, etc. WriteUnique transactions are used from IO-Coherent nodes to perform a store as it inform the interconnect to notify the RN-F nodes that had a copy of the data from the same address that data changed.

2. In this case, there are 2 options for transactions that could happen. The first is to Receive 2 separate responses the DBIDResp and Comp on the Response channel with random order and delay, and the second case is to receive a CompDBIDresp which is the other 2 responses combined.

   (a) In the first case when there are enough credits, the Response comes with a DBIDResp opcode, an Error field a DBID field, etc. DBIDResp response provides a data buffer identifier to indicate that it can accept the written data for the transaction. Also, a Response that will come or Response channel is the Comp (opcode) Response. This response can come before or after of DBIDResp. Comp response indicates that the transaction is observable by other Requesters.

   (b) In the second case, one response can come with opcode CompDBIDResp, a DBID field, etc, and indicates that the Responder can accept the data for write and the transaction is observable by other Requesters.

3. Subsequently, when the DBIDResp Response or CompDBIDResp is received the Request node sends the Data for write with the Data Response NonCopyBackWrData (opcode) if DBIDResp and CompAck were sent separately or NCBWrDataCompAck (opcode) if one CompDBIDResp was sent. The requirement for the Data Response is to set the TxnID equal to the DBID that is received with the Response, so the receiver can understand which transaction the sent data are referring to. At this point, data have been written, and the transaction is over if the ExpCompAck field isn't set on the corresponding request.

4. Finally, in the case where the ExpCompAck was active in the write request, then the Requester has to send a CompAck Response on the Outbound Response channel to interconnect and indicate that the data was received successfully. In this case, the TxnID of the Response must be equal to the DBID that was received in the previous DBIDResp response. If the ExpCompAck field is 0 when the request is sent, then the Responder does not wait for CompAck Response and the transaction

is over when Data was sent with the NonCopyBackWrData or NCBWrDataCompAck Data transaction by the Request node.

In this DMA the ExpCompAck is always off in Read request, hence the CompAck Response is not sent because there is not any performance difference. However, a programmable way for including a CompAck response from DMA could be implemented in a future work in order to make the DMA compatible with every kind of slave (if the slave requires a CompAck to operate correctly). Also, the NonCopyBackWrData is used as a Data response in any case (separate DBIDResp and Comp or CompDBIDResp). All the above information can be found in the C version of AMBA 5 CHI manual of ARM[18].

3.2.4 CHI-Converter

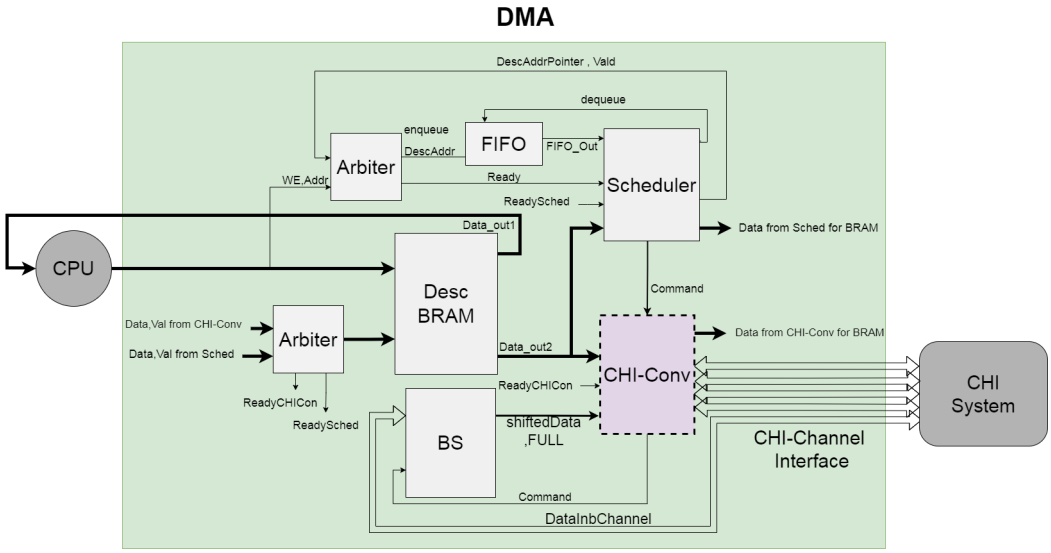

FIGURE 3.14: Location of CHI-Converter in the Design

In general, CHI allows data width of 128, 256, or 512 bits with a number of 4, 2, or 1 packets respectively. In this case, the data width is 512 bits (64 Bytes) which makes the number of transferred packets 1 for each request. Source, Destination addresses can be misaligned, which means that data for write can not be sent unchanged but should be shifted left or right according to these addresses. Also, there are cases where one read should become two writes transaction or in reverse. For this reason, CHI-Converter does not receive data directly from the Inbound Data Channel but requests them from Barrel Shifter which is responsible for shifting, merging read data from different

transactions and delivering the data to CHI-Converter every time they are ready.

CHI-Converter is the module that receives commands from the Scheduler of the system and based on the command's information it transfers data from one memory location to another by using the Coherence Hub Interface protocol Fig.3.14. To accomplish this operation, CHI-Converter has to generate a few Read and Write transactions for each command.

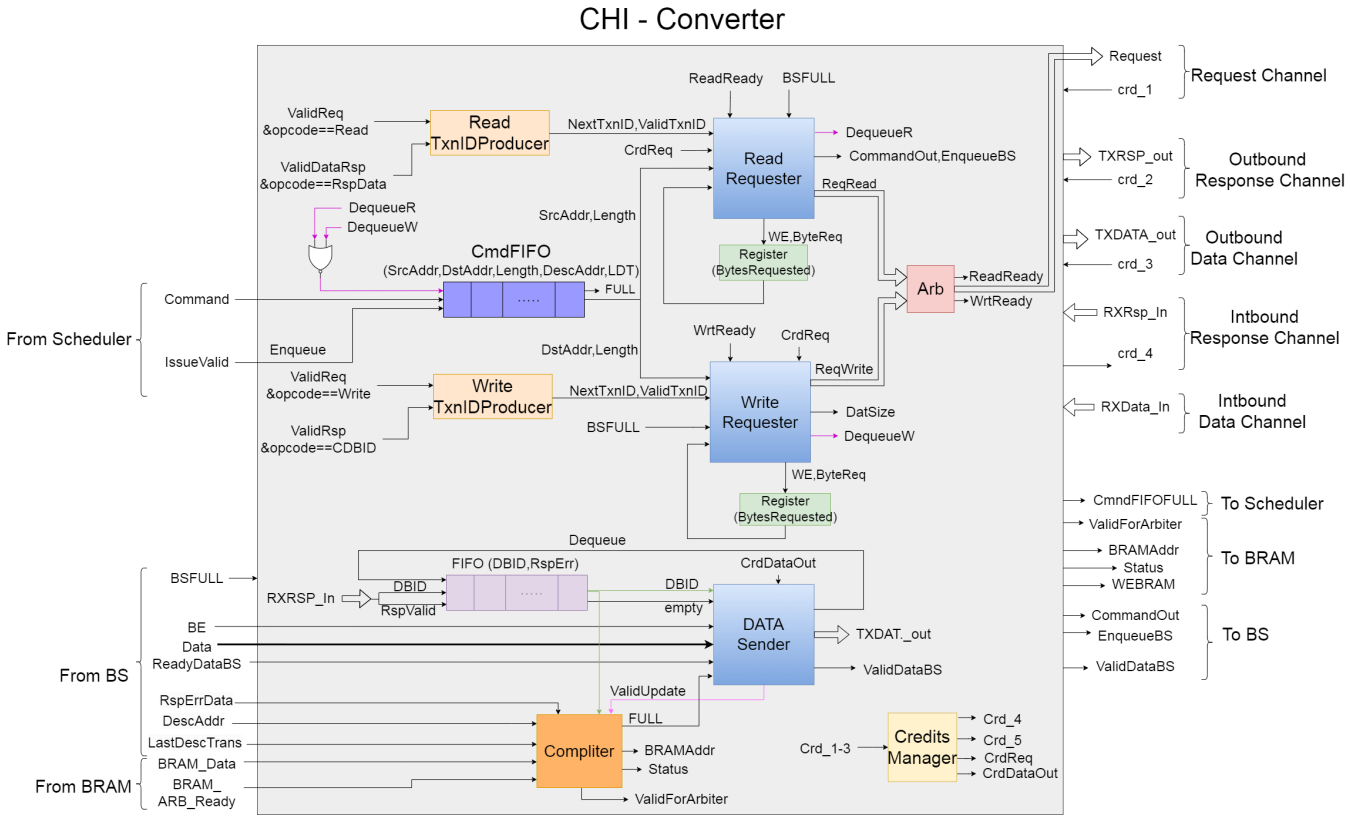

FIGURE 3.15: CHI-Converter

Read and Write Requests are generated by the Read and Write Requester blocks, respectively, Fig.3.15. These blocks create the correct request FLITs by completing their appropriate fields. Some of these fields are SrcID, TgtId, QoS, etc. which are parameters of the module. One of these fields is the opcode, which indicates the type of request. Read Requester always uses the appropriate opcode for ReadOnce request, which indicates that the transaction is a read and requests the last snapshot of memory at the address field of FLIT. In the same way, the write Requester uses the opcode for the WriteUniquePtl request, which means that it is a write transaction and data must be written at the Address of FLIT. When there is at least one credit in the

Request channel, Command FIFO is not empty and the Requesters have created the appropriate FLIT then if they want to transmit it they must enable the FLITV signal of the outbound request channel. However, there is only one Request channel for both Read and Write requests, which creates the need for an Arbiter. Each of the Requester blocks asks for control of the Request channel from the Arbiter when they want to send a new request. When only one of the Requesters asks for permission of the Request channel, then the Arbiter gives this module immediately permission. When both Requesters want the permission, then the Arbiter gives alternately permission, called round-robin. The reason for the selection of the round-robin method on arbiter is because a full memory transfer needs the response from both a read and a write request. For this reason, it would be preferable if the CHI-Converter could receive read and write responses alternately for starting the data transmission for write as fast as possible, instead of receiving all read or all write responses continuously and waiting for the corresponding response to arrive.

Every time the Read Requester generates the first transaction from each command, it passes the corresponding command to the Barrel Shifter, so it can modify the incoming data correctly. Each command in CHI-Converter has come from Scheduler, and it can contain a "Chunk" number of CHI-transactions. This means that CHI-Converter has to execute at least "Chunk" number of Reads and "Chunk" number of Writes for each command, as the engine has to copy the data from one memory location and transfers them to a different one. This is achieved by storing in 2 registers the number of requested Read and Write bytes for the current command. This number is calculated by adding to the previous values of these registers the outcome of 64 (which is CHI data width) -(the corresponding command address + the previous value of the register) modulo 64. This way each Requester knows the number of requested bytes for the top command and by adding this number to the corresponding Address of the command they generate the appropriate Address field of FLIT which is necessary for the next Read and Write transactions. If the value of one register becomes greater than the length of the command then the corresponding Requester has requested all of the transactions of the command, it stops asking permission for the request channel and activates its dequeue signal which means that this block wants to operate with the next command. When both values of registers are greater than the command's length, then all transactions from both Requesters have been sent. Subsequently, both Read and Write Requesters have the corresponding

dequeue signal enabled which means that they agree to Dequeue the finished command and the 2 registers that store the number of requested bytes are set back to 0.

CHI allows up to 256 different TxnID since TxnID is an 8-bit field. Each Read or Write Request must have a unique TxnID in order for every response to be distinguished. For this reason, there are 2 blocks called TxnIDProducer that generate a unique TxnID for Read and Write Requesters respectively. The first TxnIDProducer generates TxnIDs from 0 to 127 and the second from 128 to 255. TxnIDProducers give every ID of their range to the corresponding Requester with circular order (after 127 generates 0 and after 255 generates 128 respectively) by increasing a counter every time a request is made and containing one more register that counts the number of free TxnID. These registers are set to 128 when the system resets, and this number is decreased when the corresponding Read or Write Request begins. When a response comes, then the corresponding register increases. When one of the registers is 0 then the corresponding ValidTxnID signal deactivates and prevents the Requester from generating a new request. When a response comes and the register's value increase, then the corresponding ValidTxnID will activate again, and the Requester will be able to generate its next request. The given TxnIDs are always unique because in order to repeat a TxnID, TxnIDProducer must give the rest 127 TxnIDS first. If all TxnIDs are given, then the counter for free TxnIDs is 0 and TxnIDProducer can't give more TxnIDs. If a response is received and the counter increases then the finished transaction must be the first TxnID that has been sent because all responses are coming in order and then the first TxnID can be reused.

In CHI-Converter there is one more FIFO Fig. 3.15 which stores the appropriate information that the blocks Data Sender and the Completer need to construct each Data FLIT for write transactions, and update DescBRAM respectively. The FIFO is needed for keeping the received DBID field that comes from the Inbound Response Channel in response to every write request which will be used for the next Data write transaction and the corresponding error, if there is any, in order Completer to update the Status of Descriptor if needed by the use of a small FSM described below Fig.3.17. DBID indicates the Data Buffer ID that the CHI interconnect has reserved for the Data that will be sent on the next Data Write, and it must be used in the FLIT of Data Write as the TxnID field. This FIFO enqueues the DBID and error fields from Response FLIT every time there is a valid DBIDResp

or CompDBIDResp response on the inbound Data channel. Every time this FIFO is non-Empty and there is at least 1 credit on the outbound Data Channel, then the block Data sender requests the shifted Data and the Byte Enable field from Barrel Shifter by enabling the ValidDataBS signal. When Barrel Shifter has received the appropriate inbound read Data, it enables the input of CHI-Converter ReadyDataBs and places the correct values to input signals Data and BE. This way Data sender receives the configured Data and constructs the corresponding Data FLIT with opcode NonCopyBackWrData which is sent on the outbound Data channel. Every time a valid FLIT is sent on the outbound Data channel and a FULL memory transfer is completed the Completer updates Descriptor's Status if needed and the FIFO with the appropriate DBID and RespError fields are dequeued, so the Data Sender will gain the next fields for the next Data write.

Completer receives DescAddr, RespErr, and LastDesctrans(Final Transaction of Descriptor) from DBID FIFO and from Barrel Shifter when it responds to DataSender's request and if there is an error or all transactions of a Descriptor are over it updates the status of the corresponding Descriptor.

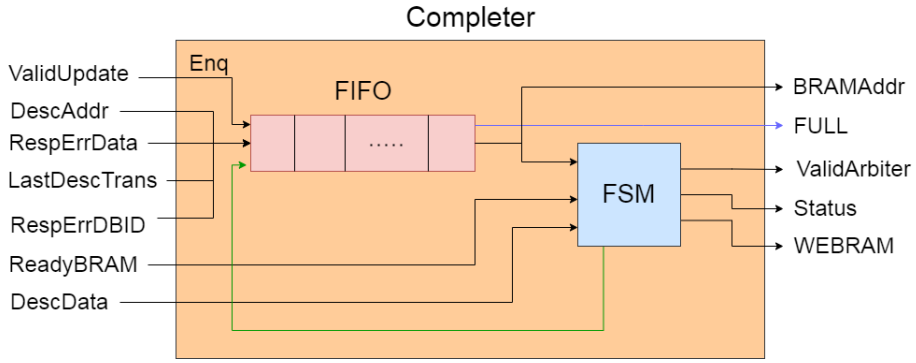

FIGURE 3.16: Completer

Completer is composed of one FIFO and one FSM Fig.3.16. Every time a data write is sent on the outbound data channel if the error field from Data or DBID is not 0 or LastDescTrans is active then the ValidUpdate signal is enabled, and these 3 fields are written inside the FIFO. If FIFO is not empty then FSM is executing the following operation Fig.3.17:

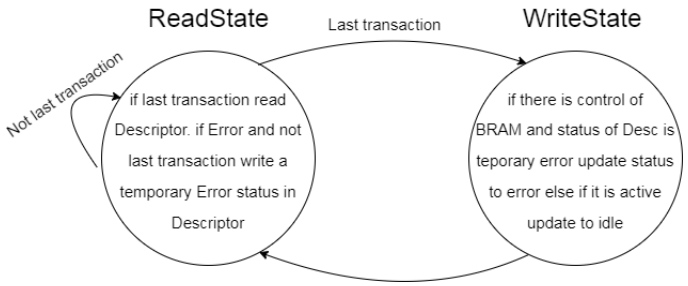

FIGURE 3.17: FSM of Completer

When the top element of FIFO has a non-zero error field, and it is not the last transaction of the Descriptor, then FSM requests to take access to BRAM by activating the ValidArbiter signal. If the Arbiter responds with the corresponding ready signal, then FSM enables the correct bits of the Write Enable signal, writes the Descriptor at the given address with a temporary error status, and dequeues the first element of FIFO. In the case where the first element of FIFO indicates the last transaction of the descriptor then FSM reads this Descriptor on the first cycle and on the second, it transitions to Write State where it will write the Status of the Descriptor with Idle if Descriptor has an Active Status or with Error if there is a non-zero error in FIFO or the status of the Descriptor was temporary Error. After this operation, FSM extracts the first element of FIFO and returns back to the starting State. In this way, Descriptors stay up-to-date every time a memory transfer has been fully completed and the processor is able to know when to re-write the finished Descriptor.

Finally, the last component of the CHI-Converter is the Credits Manager. Each channel has a reverse signal on its direction which is called LCRDV and indicates when a channel receives a Credit and thus it is able to execute one more transmission. This block is responsible for counting the number of Credits on the outbound channels and sending the appropriate number of credits on the inbound channel. Credit Manager contains three counters, one for each inbound channel which are increasing when the corresponding LCRDV signal is set and decreases when a transmission is executed on the same channel. This way, this module can inform every other component of CHI-Converter when there are enough inbound credits for them to use one of the channels. Another operation of this component is to send credits on the inbound response channel. This task is a little trickier as there should not be given more credits than the free space in DBID FIFO because there would be the danger of overflowing. For this reason, there are two counters for

the inbound response channel, one for counting the number of given credits which increases every time the LCRD signal is set and decreases when a response is received and one for counting the difference of the free spaces in FIFO with the given credits on this channel. When this difference is greater than zero and the number of given credits on this channel is less or equal to the max number of credits that CHI allows which is 15 then an extra credit is transmitted on this channel. This way the inbound response channel will always have the maximum number of credits without being able to overflow the DBID FIFO. It is worth noting that there is not any logic for managing the credits on the inbound Data channel as the master of this channel is Barrel Shifter and it has the responsibility for the management of its credits.

CHI-Converter by implementing all these components described before is able to interact with a CHI interconnect and be the master which will initiate and execute a full memory transfer based on the commands that will be inserted within its FIFO. It always requests Data from memory by the ReadOnce transaction which will give the last snapshot of the memory and writes data with the WriteUnqieuPtl transaction which will make the other masters to be informed for the written data if they want to use them which makes CHI-Converter a Requester I/O Coherent node of CHI system. CHI-Converter is able to operate with the type of responses that already have been described before, but it can be further developed in the future to support more sophisticated responses from the CHI interconnect. The Disadvantage of the CHI-Converter is that it supports only in-order execution so it requires receiving all responses in the same order as requested to operate normally. This issue can be solved in the future by implementing one extra BRAM where all responses will be stored based on the TxnID and they can be accessed in any order, which will allow CHI-Converter to handle out-of-order responses.

3.2.5 Barrel Shifter

Barrel Shifter is designed with the hypothesis that all the data responses are received in the same order in which they have been requested.

Barrel Shifter is the component that receives Data responses from the external CHI system, and it is responsible for shifting and merging Data in order to be written at the correct memory address Fig.3.18. More specifically, the number of Data that will be read from Memory is 64 bytes as the number of Data that

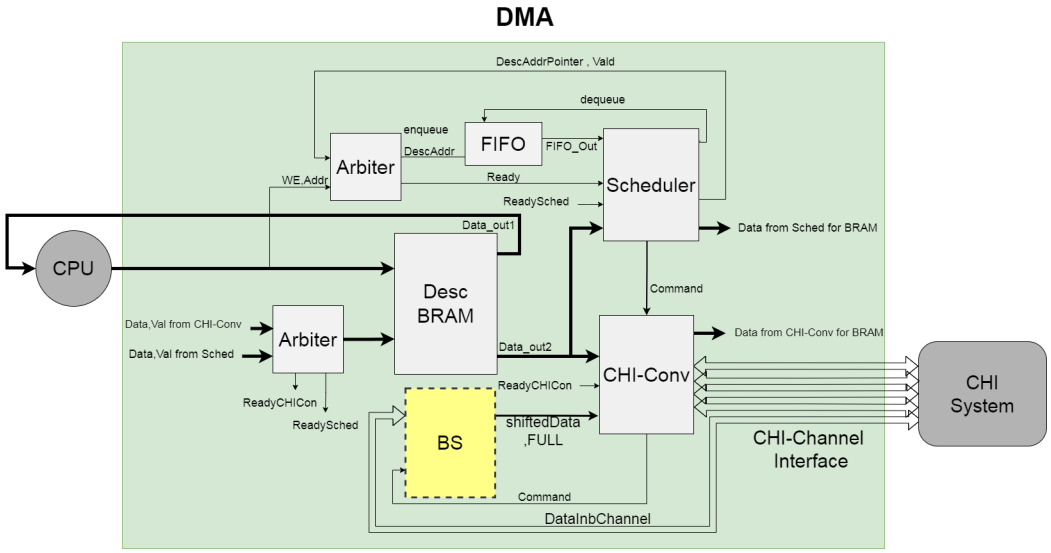

FIGURE 3.18: Location of Barrel Shifter in the Design

will be sent for write. However, the Source and Destination Addresses of transfers may be aligned with the piece of Data that the CHI system uses for every Data transaction (64 bytes). In addition, Source and Destination address can be misaligned within this piece of Data (ie. SrcAddr modulo CHI-DATA-WIDTH != DstAddr modulo CHI-DATA-WIDTH) which makes the use of Barrel Shifter necessary for adjusting the read Data so they can be ready for write transactions.

Barrel Shifter contains 2 FIFOs: one for commands and one for Data Fig. 3.19. Commands come from CHI-Converter every time it begins a new transfer of a chunk and are stored in FIFO when the Enqueue signal is set. Commands as in CHI-Converter are composed of the fields: SrcAddr, DstAddr, Length, DescAddr(Descriptor's Address), and lastDescTrans (Last transaction of Descriptor) which are needed to generate the correct shift and some of the outputs. When the CHI interconnect responds to a Read request generated by the CHI-Converter, it promotes the requested data within the DataFLIT on the Inbound Data Channel. Every time the FLITV signal of this channel is enabled, the read data and the corresponding error field are being stored in the Data FIFO of the Barrel Shifter. This FIFO is used to keep all the obtained Data in order, so they can be modified and passed to the CHI-Converter when it requests them by activating the ValidDataBS signal.

The shift that must apply on Data is identical for every other Data word from the same Chunk(bytes that have been requested from one Command) because the misalignment for all transactions within a command is the same as

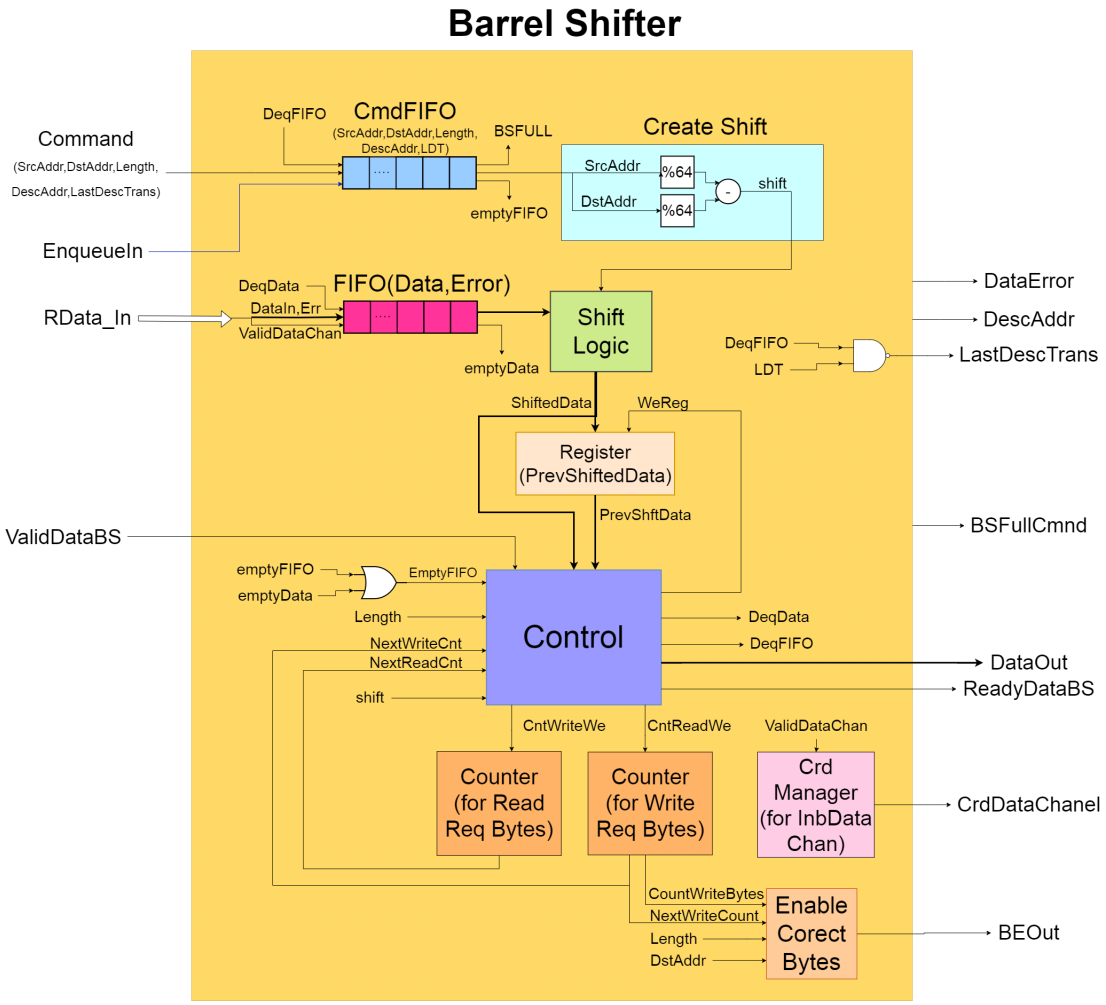

FIGURE 3.19: Barrel Shifter

all the data are being stored at contiguous memory addresses. The proper shift(in Bytes) that must apply to each Data piece from the same Chunk is the difference of Source and Destination address modulo the Width of CHI Data (64 bytes). The modulo operation is very easy to apply on the addresses as the CHI-Data-Width is a power of 2, so it just needs to ignore the least square root of CHI-Data-Width bits which is $\sqrt{64} = 6$ bits. The shift of Data is always right and circular, which means that the least significant bytes become the most significant and the rest are moved a few positions right. For this reason, the subtraction of Source mod 64 and Destination mod 64 Addresses always gives the correct shift for both cases that Data should be moved right or left. In the first case, where $ScrAddr \mod 64 > DstAddr \mod 64$ so read data must be shifted right, it is obvious that the difference of these numbers is the appropriate shift needed for the read data to be aligned with the data that will be written. In the second case where $ScrAddr \mod 64 < DstAddr \mod 64$ the subtraction of the two values produces a

negative number which is the two's complement of a result x that would be produced if the subtraction had been executed with the two values in reverse. Nevertheless, by representing this two's complement result $-x$ with $\sqrt{ChiDataWidth} = \sqrt{64} = 6$ bits which is the width of shift signal we end up with the number $ChiDataWidth - x = 64 - x$ which is the amount of right circular shift needed to align the data that is equivalent with the left shift of x bytes.

Another very important component that Barrel Shifter contains is the block which is responsible for shifting Fig.3.20. This block receives two inputs: the Data and the shift signal which is created in the way described before. The function of this module is to execute a right circular shift on the inserted Data by the amount of its input signal "shift" and provide the processed Data on its output. This block is pure combinational logic. It is composed of layers of

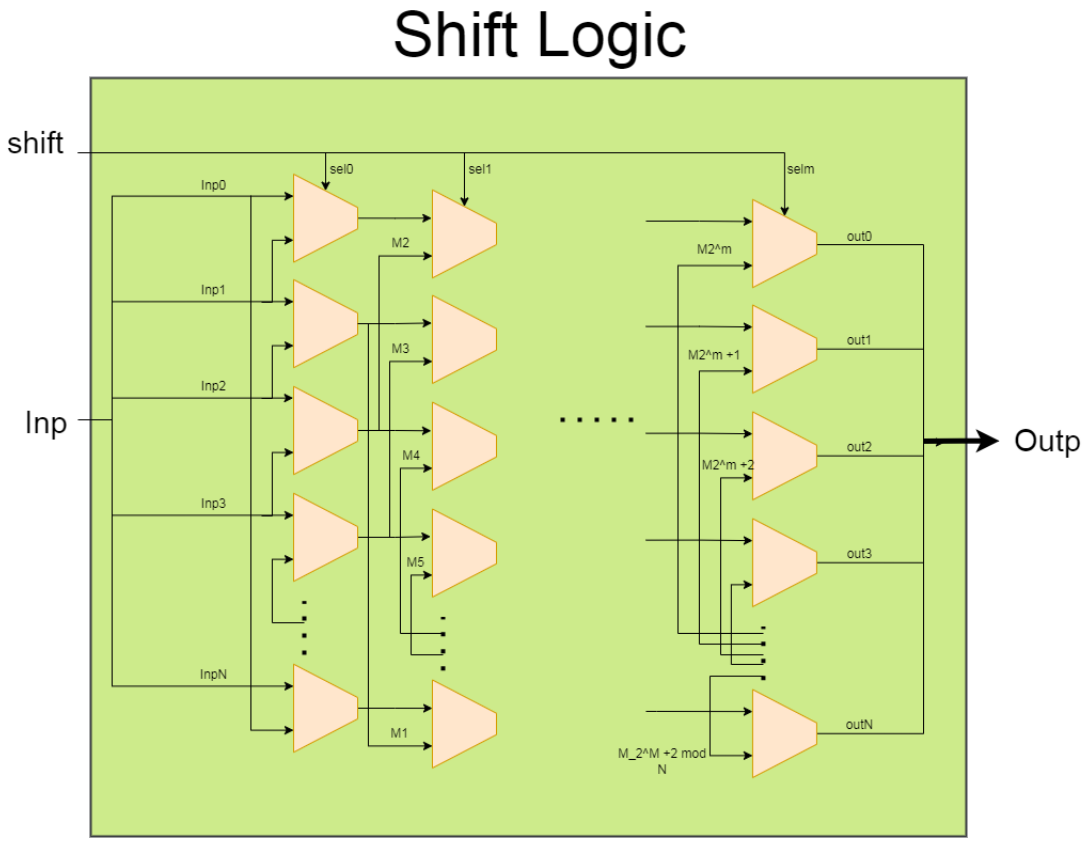

FIGURE 3.20: Shifting block

multiplexers. Its layer contains 64 multiplexers as the Data width (in bytes) of CHI. Each multiplexer of the first layer receives on its first input the corresponding data byte unchanged and on its second input the next byte which is the first input of the next multiplexer of the same layer with the last multiplexer's second input to be the first data byte. Similarly, each multiplexer of

the second layer receives on its first input, the output of the corresponding multiplexer of the last layer, and on its second input, the output of the multiplexer that corresponds to its position + 2 (for example the second input of the first multiplexer of the second layer is the output of the third multiplexer of the first layer, etc.). The last multiplexers which do not have corresponding multiplexers from the last layer to its position + 2 receive the output of the multiplexer at position + 2 mod 64(CHI-Data-Width) as the shift that must be done is circular(in this example the last 2 muxes take the output of the first 2 muxes from the last layer). Subsequently, the multiplexers of m_th layer will receive on their first input the output of the multiplexers of the $m-1$ layer and on their second input the output of the $(2^{m-1}+n) \mod N$ of the $m-1$ layer where n the position of the multiplexer and N the number of multiplexers on its layer (64). With this structure, the inserted Data can be shifted by any amount from zero to 2^m-1 (in bytes) with m being the number of layers. Thereby, the number of layers that are needed for data to be shifted for any amount between the range 0 - 63 are 6 layers and the total number of multiplexers required for this structure are $m \cdot N$ but $m = \log_2 N$ so $m \cdot N = N \log_2 N = 64 \log_2 64 = 64 \cdot 6 = 384$ multiplexers. The second input of this component is the control signal shift, which determines the amount that data must be moved. This signal has width as the number of layers of multiplexers, which is 6 in this case. Each bit of this signal is connected with the corresponding layer and controls every multiplexer within this layer. This way each active bit of shift signal forces data to be shifted by an amount of 2^n bytes where n is the position of the active bit and the total shift that is produced is the sum of the effect from all layers that the control signal has activated. The output of this block is the Data word that has been generated at the last layer of multiplexers and it is the read Data aligned with the Destination Address.

After Data is shifted the Control block of Barrel Shifter decides based on the request of data from CHI-Converter and the command if it is required for the shifted Data to be written in a register and the Data FIFO to be dequeued by enabling the appropriate control signals. Storing shifted data in a register is necessary in many cases as on the next cycles after the data FIFO is dequeued the previous shifted data must be available, in order to be merged with the next shifted Data to create the next CHI data word for the next transaction.

As Barrel Shifter needs to receive the inbound Data from the external CHI

system, it requires the occupation of the inbound Data channel. For this reason, a component that is responsible for the management of the credits on this channel is necessary in Barrel Shifter's implementation. Credit Manager is composed of two counters: one that counts the number of given credits and one that keeps the difference between the free space of Data FIFO and the given credits. The first counter increases every time the LCRD signal of the inbound Data channel is activated and decreases when a data response is received. The second counter increases when an element is extracted from Data FIFO and decreases when a valid response is received on the data channel. Finally, this module transmits an extra credit every time the number of credits that the external system possesses is less than the maximum number that CHI allows which is 15 and the number of sent credits is less than the free spaces in FIFO. In this way, the external system will always have enough credits as long as there is free space in FIFO to accept the inbound data responses.

Another important operation that the Barrel Shifter should do is to generate the appropriate Byte Enable signal. This signal is a necessary field that CHI-Converter requires to transmit the Write Data response and it indicates the position of the bytes in Data that is valid for write. BE field is produced by combinational logic. If the Destination Address modulo CHI-Data-Width + command's length is less than CHI-Data-Width (64) then the appropriate bytes that should be enabled are intermediate of the CHI-Data-Word and should be the bytes from the position DstAddr mod 64 to DestAddr mod 64 + command's length. In the case where data are not intermediate of the CHI-Data-Word which means that Destination Address modulo CHI-Data-Width + command's length is greater than CHI-Data-Width (64) then if it is the first transaction of this command the Bytes that should be enabled are the most significant bytes from the position DestAddr mod 64. In the same case but when the next transaction is neither the first nor the last from the corresponding command then all bytes must be enabled as the data that will be transmitted are an intermediate piece of the full chunk and all the bytes must be written. Finally, when the next transaction is the last but not the first of a command then the least remaining bytes should be enabled for the completion of a full chunk transmission. In order for the combinational logic which manages BE to know if the following transaction is the first, an intermediate, or the last, there is a counter that counts the transmitted Data for write.

The counter of transmitted Data for write is reset to zero every time a command is ejected from command FIFO. Otherwise, when CHI-Converter requests the next shifted data by activating the ValidDataBS signal and Barrel Shifter responses with the Data and the ReadyDataBS signal then the control of the system will activate the counter which will be increased by the number of bytes needed from the Destination address modulo 64 + the current value of counter until the next aligned address. In this way every time a Data word is transmitted this counter computes the number of bytes that have been sent and hence the amount of remaining bytes for each command can be calculated. Likewise, there is a counter that counts the received read Data. Every time new Data are extracted from FIFO and delivered to CHI-Converter, the appropriate signal from the control is enabled, and the read counter computes the total amount of bytes that have been read for the first command in FIFO in the same way as the write counter.

The two counters are necessary not only for the activation of the correct bits of BE field, but also for helping the Control block of the system to know whether it is the first or last read and write transaction for each command, so it can manage the appropriate control signals of the module. Control is responsible for the activation of the two counters, the Write Enable signal for the register that keeps the previous sifted data, the dequeue operation of FIFOs, the activation of the ReadyDataBS signal to indicate when data is ready for CHI-Converter and the creation of the correct data by merging the previous and current shifted data. The value that each signal will take depends on the length of the command, the misalignment of the Source and Destination Address, and the position of the transaction within the command (if the executed transaction is the first, an intermediate, or the last one). Control operates only when both of the FIFOs are not empty.

The Control block operates appropriately and produces the necessary signals based on the case that source and destination address of the command define. There are 3 cases that the control should behave differently. In the first case, Source and Destination addresses are aligned, as the right memory representation in the figure 3.21, or the length of the command is very small and all bytes of both read and write transactions are placed within only one line as seen in the left memory representation Fig.3.21. The colors represent the data movement for every transaction of a chunk. In this scenario control passes shifted data directly from the combinational shift logic to the output as there is no need for merging the data. It also activates the data ready signal,

FIGURE 3.21: Aligned Data Transfer

the counters and dequeues the data FIFO every cycle data are sent to CHI-Converter. When the data for the last write transaction have been sent, then control dequeues command FIFO too and the two counters are refreshed.

In the second case, shown in figure 3.22 where addresses are misaligned and data must be shifted left, the left memory representation shows the data transfer for every transaction of a chunk where the number of read and write transactions are the same in contrast with the right representation where the number of writes are one more than the number of reads. In this particular scenario, control merges the most significant, 64 – shift, bytes of the next shifted read data with the least, shift, bytes of the previous shifted read data that have been stored in the register. This way, control always produces the correct data on output as it keeps on the write enable signal for shifted Data register every time a new data piece is sent to CHI-Converter. Similarly to the previous case, control also activates the counters, the ready signal, dequeues data FIFO when data are requested and dequeues command FIFO when it is the last write transaction. However, in this case there is the scenario where the top data in FIFO are from the last read transaction of the current command but the next data that will be sent are not for the last write transaction. This scenario happens for example when one read must become two writes. In this case, the read counter won't be activated, and Data FIFO won't be dequeued on the first cycle, but they will do on the next one.

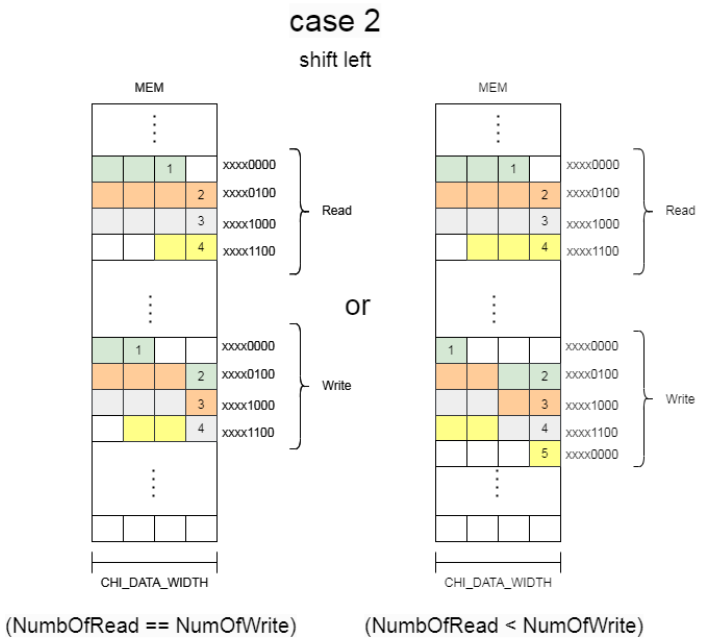

FIGURE 3.22: Misaligned Data Transfer with left shift

In the last case Fig.3.23 where also addresses are misaligned, data must be shifted right. Similarly, with the previous case, the left representation of the figure shows the movement of data from one memory location to another with the number of writes equal to the number of reads and correspondingly in the right representation the number of reads are 1 more than the number of writes. Data out in this case is again the concatenation of the most significant, 64 – shift, bytes of the next shifted read data with the least, shift, bytes of the previous shifted read data that have been stored in the register. The difference, in this case, is that the first read data are not enough for a Data write to be sent, so control writes the shifted data in the register, dequeues the data FIFO and updates the read counter without sending Data to CHI-Converter (ready deactivated). After this important detail on the first cycle that differs from the left shift case, the continuation of the process is the same, including the scenario of the last read becoming two writes. This way the control of the system handles each one of the cases and the Barrel Shifter is able to shift Data from every direction and pass the data to CHI-Converter in the appropriate way for all CHI transfers in commands to be completed.

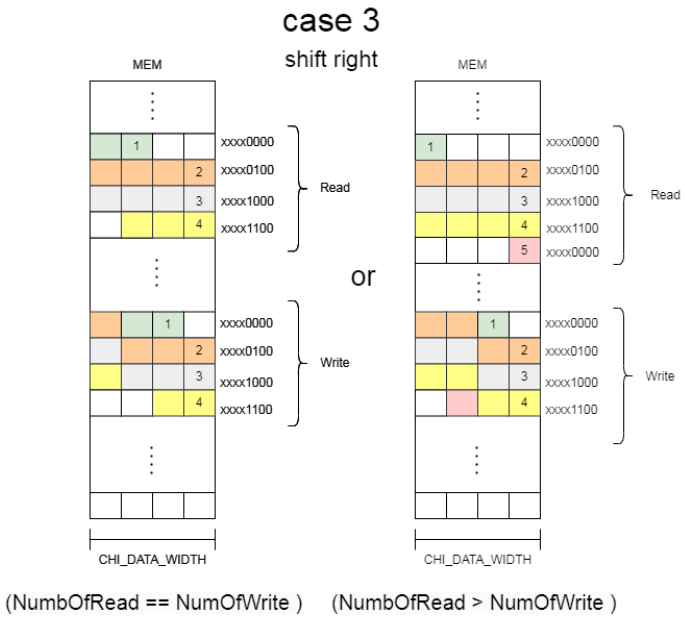

FIGURE 3.23: Misaligned Data Transfer with right shift

Chapter 4

Simulation and Testing

To verify that the system operates expectedly, a behavioral simulation is executed for each module individually as well as for their combination, which constructs the whole DMA by using the vivado simulator (XSIM). The Test-Benches that were created to verify the individual components are simple, as they just test the behavior of the module by testing a large number of different inputs and trying to cover every corner cases. On the other hand, the testing of the full system is a more difficult task, hence its Testbench is more complex.

4.1 Verification of sub-modules

4.1.1 Simulation of Scheduler

For the verification of the Scheduler, an exhaustive examination of all possible corner cases is executed by creating a Testbench which assigns the appropriate input at the correct time to make the conditions for these cases to be observed. This method of verification can apply in this module because the number of different important cases is small, so the test can be deterministic. For this reason, the correctness of the behavior of the Scheduler is verified by looking at the waveforms that the simulation tool provides and there was no need for implementing a self-checking test.

Note: ReadyBRAM and ReadyFIFO indicate that there is access through Arbiters to the BRAM and the main FIFO respectively, EmptyFIFO indicates there is no address pointer in the main FIFO, and CmndFIFOFUll indicates that CHI-Converter can't accept more commands because its FIFO is FULL.

Corner Cases			
Input Signals	IdleState	IssueState	WriteBackState
!ReadyBRAM	check	check	check
!ReadyFIFO	don't care	don't care	check
EmptyFIFO	don't care	impossible	check
CmndFIFOFULL	don't care	check	don't care

TABLE 4.1: corner cases of scheduler

The test which is created for verifying this module has successfully shown that the FMS of the component always returns to IdleState whenever the control of BRAM is lost independently of its current state. Moreover, it checks that the FSM stays in the IssueState every time the CHI-Converter can't accept more transactions and in the WriteBackState when the control of FIFO has not been obtained. Also, the module's behavior when there is only one transfer in the system is important to be checked. This case causes the FSM to transition in WriteBackState when the main FIFO of the system is empty. The behavior, in this case, has also been verified as the module uses the address pointer that was stored in the register to read the BRAM and write it back to FIFO simultaneously. Finally, the update of BRAM has been checked and operates correctly even when the processor has assigned absurd transfer instructions as the SentBytes field is greater or equal to the BytesToSend field where Scheduler updates the Status field to Idle. In this way, the verification of the Scheduler is completed and the module is ready to be integrated into the whole system.

4.1.2 Simulation of Barrel Shifter

The verification of Barrel Shifter accomplished by the usage of 2 different methods. The Barrel Shifter receives all the inbound data responses for all read requests of the same command in order, and it shifts and merges the inserted data to create the corresponding data for write transactions of the same command before it serves the next one. For this reason, Barrel Shifter uses specific patterns to create the appropriate data, which depends on the direction of the shift and the number of reads/writes for each command. Hence, the first method verifies the operation of the module by checking that every possible pattern is executed correctly, Fig.4.1.

Possible Shift scenarios for a Command			
Number of Reads/Writes	Shift Left	Shift Right	No Shift
Reads > Writes	impossible	check	impossible
Reads < Writes	check	impossible	impossible
Reads==Writes	check	check	check
1-Read 1-Write	check	check	check

TABLE 4.2: Shift scenarios of Barrel Shifter

FIGURE 4.1: Shift Cases of a Command

For the first method, the test starts by inserting commands that force Barrel Shifter to execute each one of the possible patterns. In order for the module to complete its operation for each of the commands, it is necessary to receive the corresponding data responses on the inbound Data channel. For this reason, the TestBench implements a mechanism for counting the credits that Barrel Shifter sends, and if there is a non-consumed credit on the channel it sends after a small random delay an inbound Data response with a random Data field. All the possible cases which are checked by the execution of the inserted commands which are visualized in the figure above are:

a small shift where all data for both read and write are placed within a data line, a multi-read/write aligned transfer, a left shift with the number of reads to be less and equal with the number of writes, and with the same way a right shift with the number of reads to be greater and equal to the number of writes. The verification for each one of the patterns, except by looking at the waveforms, which is more difficult as both the inserted and shifted data are 512-bit vectors and their comparison is time-consuming, it is also achieved by some software code that is integrated into the TestBench. This code, stores every inserted data of the same command in a wide vector, each one next to the others as well as the given data with the corresponding BE output active in a different vector. Then every time a command is over it checks if every bit of the first vector in the range [SrcAddr mod 64(CHI-DATA-WIDTH in bytes), SrcAddr mod 64 + Length] is equal to the corresponding bit of the range [DstAddr mod 64, DstAddr mod 64 + Length] of the second vector. If all the bits of the 2 vectors are matching then the Barrel Shifter operated correctly for this command and a success message is displayed, else the program stops, and an error message is printed.

As the TestBench implements this software for automatic verification of each command it is obvious that the second method that is used to verify this component is by inserting a large number of commands with random source, Destination addresses, and Length. In this way, each case of those described above is tested many times with different shifts and Lengths. Also, this method covered many of the corner cases, as when the FIFOs are FULL or when all of the credits have been given. Finally, after a huge amount of random commands (around 50000) were executed successfully, the verification of Barrel Shifter is over as every possible command passes the test.

4.1.3 Simulation of CHI-Converter

The TestBench of the CHI-Converter is designed to receive CHI-Requests and to respond with the appropriate CHI-Responses after a random delay. First of all, this TestBench implements a mechanism for counting the credits that have been sent on the inbound channel and provides its own credits on the outbound channels after a random delay. Moreover, it implements 2 FIFOs, one for storing all the uncompleted read and one for the uncompleted write requests. The corresponding FLITS are enqueued in FIFOs every time the FLITV signal is active and there is at least one unused credit on the request channel. Subsequently, when read FIFO is not empty, the TestBench

transmits on the inbound Data channel a CompData response with random data and the appropriate fields after a random time, and in the same way, when write FIFO is not empty, it transmits a CompDBIDResp response on the inbound response channel. However, CHI-Converter can not handle the inbound Data responses on its own so the Barrel Shifter module which is already verified has been implemented in the TestBench and it receives the inbound data and then passes it to CHI-Converter. In this way, CHI-Converter is able to interact with the TestBench and execute all of the transfers for each inserted command.

For the verification of the module, TestBench starts by inserting a large number of commands in CHI-Converter with random Source, Destination addresses, and Length which are also stored in an array. When the module receives the first command as well as the necessary credits, it starts generating read and write requests and the CHI interaction of the module with the TestBench begins. Every time a valid FLIT is transmitted on any channel, an automatic mechanism checks if there is at least one credit on this channel. If there are no credits on this channel then the operation stops and an error message is printed, differently, a copy of the FLIT is stored in an array according to the transmission channel. Also, another condition that the test is checking during the execution of transactions is the correctness of the TxnID field. Every time a CHI request is transmitted, an automatic procedure checks if its TxnID is unique or if another uncompleted transaction has already used it. This is succeeded by comparing it with all the TxnIDs of the Request FLITs that have been previously stored in the array which haven't receive a response, so the FLITs in the corresponding position of the other arrays are 0.

When all transactions are over and hence all of the commands have been executed, a Task is triggered which checks the operation of the module by looking at the arrays that have been written with the information about the commands and every transaction on each channel. The conditions that this task examines in order to verify that all the transactions were executed correctly are :

1. All the read and write requests must use the appropriate address field based on the Source and Destination addresses of the corresponding command

2. All read requests must have the correct opcode ReadOnce

3. All the inbound Data responses must have opcode CompData
4. All the write requests must have the correct opcode WriteUniquePtl
5. All inbound responses must have opcode CompDBIDResp or DBIDResp
6. the TxnID of the inbound responses must have the same TxnID as the requests
7. All outbound Data transactions must have NonCopyBackWrData
8. All outbound Data transactions must have the same TxnID as the DBID received in the DBIDResp response

If the above conditions apply to every transaction that has been transmitted and stored in the arrays then a successful message is displayed else an error message is printed with the information about the transaction that failed the test.

This TestBench is executed many times with different delay times to verify the module's behavior under different conditions. More specifically, in one of the executions the TestBench was giving extra credits for the outbound channels after a long time which resulted in the Command and Data FIFOs being FULL most of the time. In a different execution, TestBench was giving Data Responses very rarely and DBID responses very fast which resulted in the DBID FIFO being FULL very often, etc. After many possible executions passed all of the tests, the behavior of the CHI-Converter to interact with a CHI interconnect with the CHI protocol is fully verified.

4.1.4 Simulation of Completer

The operation of the module to update the BRAM when a full transfer of a Descriptor is over has been individually tested by a different TestBench which verifies the behavior of the sub-module Completer. In the same way as the previous testing, it inserts in completer many different instructions about Descriptors that their transfer has been fully finished, or an error occurred during the process. Then Completer requests the control of BRAM and tries to read Descriptor's Data by setting its appropriate output signals. When these signals are active, TestBench enables the input that allows permission in BRAM and gives random Descriptor's Data to the module with the Status field to be Active or TempError. Finally, Completer updates the Status of Descriptor by managing its output signals WE, Address, and StatusData.

For the verification of this sub-module the TestBench as for CHI-Converter, stores the inserted instructions, the read Descriptor's data, and the output signals of the component in arrays every time the input ValidUpdate or the output WE signals are not zero respectively. When an update for each inserted instruction is executed and the FIFO of the Completer is empty, the system verifies that the process operated correctly by checking the appropriate information that was stored in the arrays. The conditions which are checked are :

1. if there is an error in the transfer, and it is not the last transaction of Descriptor status should be updated to TempError
2. if there is not an error, it is the last transaction, and Descriptor's Status is Active the updated status must be Idle
3. if it is the last transaction, and Descriptor's Status is tempError the updated status must be Error
4. if there is not an error, it is the last transaction, and Descriptor's Status is Active the updated status must be Error

Note: tempError Status is a value that Completer writes in Descriptor when an error occurred in the process, but Descriptor has not fully transferred, so the processor should not rewrite this Descriptor yet. TempError Satus will always end up being updated to Error when Descriptor's transfer has been fully completed.

After it is checked that Completer is operating successfully in all cases and there are no problems with the FIFO being FULL, the verification of the submodule Completer and hence for the module CHI-Converter is completed.

4.2 Verification of FULL system

4.2.1 Functionality Check

In order to verify that the full DMA operates as expected, a TestBench was created which examines the functionality of the system. The verification of the functionality is divided into two parts.

First part of verification:

In this part, Testbench checks if the operation is executed correctly until the scheduling of the commands. This is succeeded by using a TestVec-

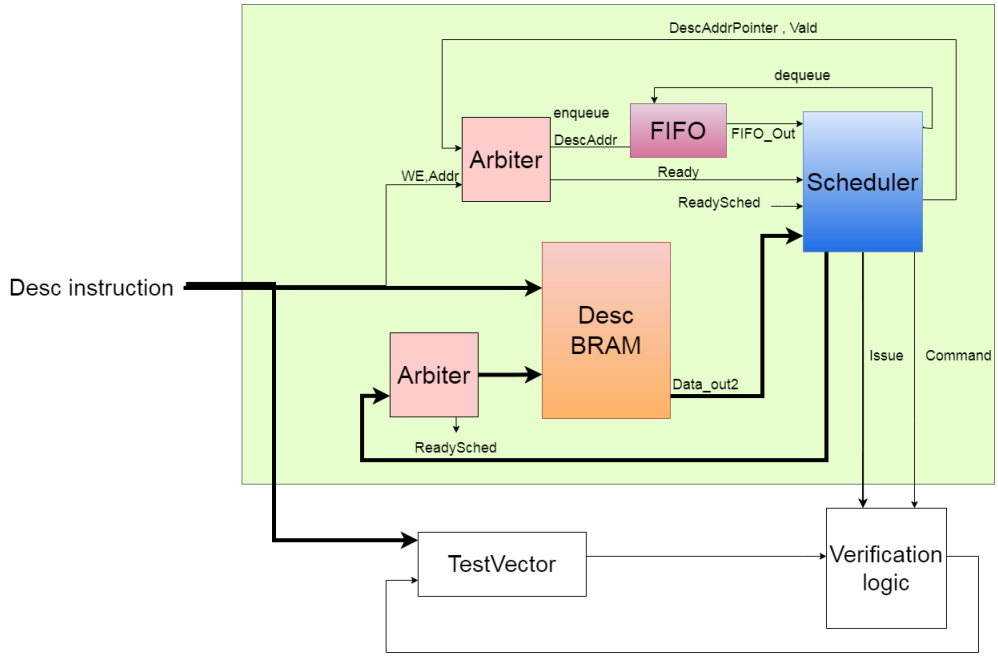

FIGURE 4.2: scheduling verification

tor for storing the information that the processor writes in every Descriptor and some logic that checks if the commands that Scheduler sends to CHI-Converter are correct based on the corresponding Descriptor. More specifically TestVector, just like BRAM, stores the information for every Descriptor that is assigned in DMA which are the Source, Destination addresses, the number of bytes to send, the number of scheduled bytes, and the last field is the LastDescTrans which indicates if all of the bytes of the Descriptor have been scheduled. Every time the WE input is active, the first four fields are written in TestVector at the corresponding input address. Subsequently, every time the Scheduler enables the issue output signal which indicates that there is a valid command and the CmndFull of CHI-Converter is disabled, then the verification logic checks if the scheduled command can be valid based on the corresponding inserted transfer. The examination operation begins by checking if the Source and Destination addresses of the command are the same as the Source and Destination addresses of the transfer in TestVector at the command's address + the number of its scheduled bytes. If that's true, then the command contains the correct addresses and the SendBytes field of TestVector is updated by adding the Length of the command. The next field which is checked is the LastDescTrans of the command. If this field is on then this command should be the last one for the corresponding transfer. For this

reason, if LastDescTrans is on then the system checks if the SentBytes field of the transfer + the length of the command is the same as the BytesToSend field of the same transfer. Otherwise, if the LastDescTrans of command is off, then the SentBytes field of the transfer + Length of command must be less than the BytesToSend field. If that's correct, the LastDescTrans field of the corresponding transfer in TestVector is also updated. In case, where one of the above conditions is not satisfied, the TestBench stops and an Error message is displayed. Lastly, when all transactions of the test are over, the system checks if all transfers which are written in TestVector are completely scheduled by examining if all of the SentBytes fields are equal with the corresponding BytesToSend fields and if the LastDescTrans is enabled for all transfers. In this way, the system can ensure that all transfers have been fully scheduled and none is lost somehow in the process. A representation of the verification structure is shown in figure 4.2.

Second part of verification:

In this part of verification, Testbench checks if the CHI interaction between the DMA and the interconnect is executed correctly based on the commands that CHI-Converter receives and if the data are transferred properly.

After the first part, it is verified that the commands which are inserted in CHI-Converter are correct, and the verification of the rest of the system is based on these commands. So a new structure which is shown in Fig.4.3 is built to ensure that the rest of the system operates correctly based on the produced commands. First of all, six FIFOs are implemented for storing information about the received commands and the transactions which are transmitted on every CHI channel. Two of those FIFOs keep the read and write request FLITs respectively, one the inbound Data FLIT, one the inbound Response FLIT, one the outbound Data FLIT, and one the inserted commands. When all of the FIFOs are not empty, then a transfer of a CHI-Word has been finished and the verification logic checks if all the CHI transactions have been executed correctly based on the corresponding command. The conditions which are examined are the same as those described in the verification of CHI-Converter. If there is an error in the process, then the operation stops and a message is displayed. Otherwise, if the transfer is executed correctly, the verification logic extracts the first element of every FLIT FIFO and examines the next transfer when it is finished. If all the transfers of a command are completed which can be understood by counting the number of transferred

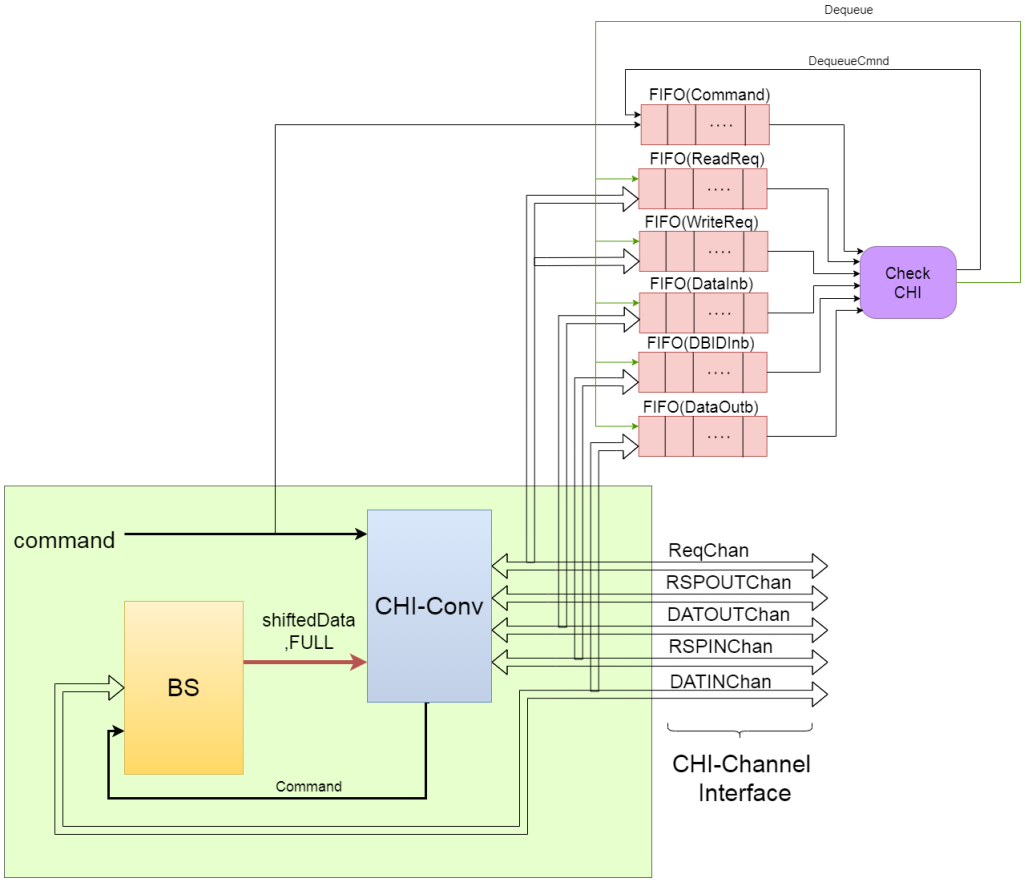

FIGURE 4.3: CHI verification

bytes, then the FIFO that stores the commands extracts its first element at the same time with the other FIFOs.

There is a large vector on the other side of the CHI channel interface which represents the DDR of the CHI system, which is initialized randomly when the reset signal is active. All the transmissions which are sent to DMA in response to read requests contain data from the requested address of this vector, and all the data that are sent from the DMA for write with the corresponding BE bit active are stored in the same way at the requested address of the same vector. This pseudo-DDR is very important, as it provides a way to truly verify that the DMA has transferred the data correctly. Every time, a finished command has the LastDescTrans field active which indicates that the whole transfer of a Descriptor is completed, the verification logic reads the instructions SrcAddr, DstAddr, BytesToSend that were written in TestVector of the first part of verification at the command's address, and it compares every bit in the range [SrcAddr, SrcAddr + BytesToSend] with the corresponding bits in [DstAddr, DstAddr + BytesToSend] of DDR. If all bits of the 2 ranges are the same then the transfer is executed successfully and a

message is printed with the address of the completed Descriptor.

4.2.2 Interaction With DMA

As we can not have access to an implemented CHI system, there is no need to download the designed DMA in an FPGA because there is no way to check its functionality by observing its interaction with a real system. Therefor, the evaluation of the DMA is accomplished via synthesis which provided a good approximation about the maximum frequency that the system can handle and its utilization. For this reason, the verification of the system is accomplished exclusively by simulation and the interaction with the DMA is achieved by the use of two independent virtual units, Fig.4.4. One of these two units is necessary for assigning to DMA the appropriate transfers for execution, and the other one is for responding in CHI compliant way to DMA on the CHI requests that DMA generates.

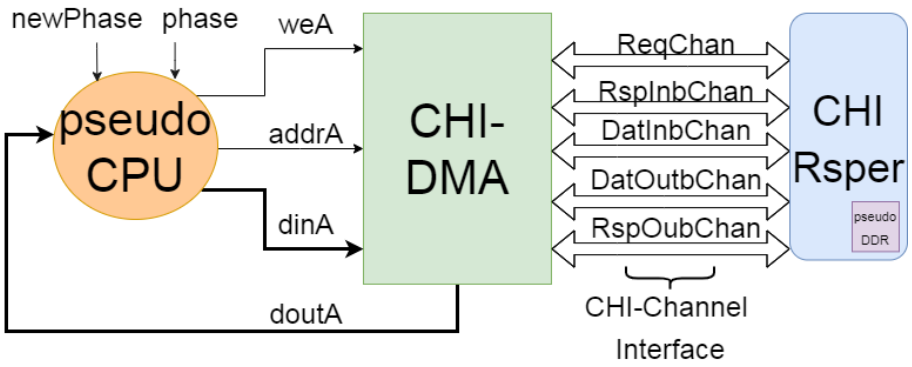

FIGURE 4.4: interaction with DMA

Pseudo CPU:

Pseudo-CPU is the unit that writes the Descriptors of the DMA for the appropriate transfers to be initiated. This component behaves differently based on its current phase. In order for the Pseudo-CPU to transition in any of its valid phases, it has to receive the appropriate inputs. More specifically, the unit is controlled by 2 inputs: newPhase, and phase. At the positive edge of the clock, if the input newPhase is active, then the second input phase is written in a register that the module contains and the behavior of the Pseudo-CPU is determined based on this register. Pseudo-CPU operates in 9 different phases and its behavior is:

1. phase 1: Inserts in DMA 1 small transfer which will need one read and one write transaction to be completed as all its read and write data are placed within one line respectively.
2. phase 2: Inserts in DMA 1 small misaligned transfer which will need two read and one write transactions to be completed
3. phase 3: Inserts in DMA 1 small misaligned transfer which will need one read and two write transactions to be completed
4. phase 4: Inserts in DMA 1 large misaligned transfer which will need many read and many write transactions to be completed
5. phase 5: Inserts in DMA continuously many(250) small transfers which will need one read and one write transaction to be completed
6. phase 6: Inserts in DMA continuously many(250) small misaligned transfers which will need one or two read and one or two write transactions to be completed
7. phase 7: Inserts in DMA continuously many(15) large misaligned transfers which will need many read and many write transactions to be completed
8. phase 8: Inserts in DMA many (45) large and small misaligned transfers with a random delay between the insertions and will need single or multi read/write transactions to be completed
9. phase 9: Inserts in DMA many (450) misaligned transfers with random size after a random delay between the insertions and in random Descriptors

Note: In all phases, the module inserts the transfers at continuous Descriptor addresses except at the last one which the Descriptors are random, so the module needs to read them first, and then if their status is Idle or Error it writes the instructions for the transfer.

Pseudo CPU assigns transfers in DMA by managing the input signals of the first port of BRAM. When the module inserts a new transfer it always sets the SentByte field to zero and the Status field to one which indicates the Active Status. In addition, the pseudo-CPU must ensure when it assigns a new transfer that the ranges [Source Address, Source Address + Bytes To Send] and [Destination Address, Destination Address + Bytes To Send] do not overlap. This is important because two such overlapping ranges may cause an

incorrect transfer by the DMA as the number of data which are moved at a time is 512 bits (one CHI-Word) and in this way, some data may be overwritten before they are read from the engine resulting in the subsequent data that gets moved being the written data instead of the expected data. This scenario with the overlapping ranges is not expected to happen in real applications except if there is a software bug and for this reason, this DMA does not implement any mechanism to handle this. Noteworthy is that two such overlapping ranges with safe distance could be a valid behavior, nevertheless, it was not considered of significant usability, and would impair correctness checking. Another condition that pseudo-CPU should control, in phases where many transfers will be assigned in DMA, is to prevent 2 independent transfers to write in the same region of memory. This case except it is an absurd behavior from the software also does not allow the TestBench to verify the correctness of the overlapping transfers, as when it checks if the data have been moved in memory it won't find the right result on the destination region and the test will fail. For these reasons, this module is designed to use correctly distributed areas for Source and Destination addresses to allow the TestBench to successfully check the result of the movements and hence the operation of the system.

CHI-Responder:

The second unit which is necessary to interact with the DMA so the TestBench will be able to verify the function of the system is called CHI-Responder. This module is responsible for receiving CHI-Requests from the DMA, and it produces in order the appropriate CHI-Responses. More specifically, CHI-Responder counts the number of credits that are sent on each channel and when a ReadOnce or a WriteUniqueuePtl request is sent on the request channel, it enqueues the requested FLIT in the corresponding (read or write) FIFO. Subsequently, when one of these FIFOs is not empty then the module waits for an amount of time to simulate the delay until the response of the CHI-system, and then it transmits a CompData response on the inbound Data channel or a CompDBIDResp response on the inbound Response channel respectively according to the non-empty FIFO. The delay times before each response and credit transmission can be random or constant and are parameterized, so they can be adjusted to the demands of the simulation.

CHI-Responder also implements a Large array that is initialized randomly at the start of the process when the reset signal is enabled, representing the

CHI-system's DDR as mentioned in the previous subsection. Every time a CompData response is transmitted by the Responder the data field attached to the response takes the value of this pseudo-DDR at the position of the corresponding requested address. In the same way, the Data which are transmitted with the NonCopyBackWrData response on the outbound data channel are written in the same array at the requested address. However, the address of the write request and the data are transmitted in different FLITs by the DMA as the write CHI transaction is a three-way handshake. For this reason, an extra FIFO is implemented in this module to keep the requested addresses for the write transactions until the corresponding data are received, so they can be written at the correct position of pseudo-DDR. In this way, the test can successfully check the operation of the DMA by observing if the data are properly moved in this array.

4.3 Simulation

The system was simulated many times under different conditions as the behaviors of the CHI-Responder and pseudo-CPU were adjusted to cover most of the regular and extreme cases that could happen. Although some of these cases are described in this section, the presented waveforms are produced by the simulation where CHI-Responder was giving unlimited credits to the DMA and also was responding after a constant number of cycles (11) from the moment it was receiving each request, as this is the simulation which will be used to measure the performance of the system.

4.3.1 Unlimited Credits Simulation

The first phase of the simulation with the described CHI-Responder is presented analytically to clarify which is the structure of the CHI-transactions that are generated by the DMA and how they are represented in the waveforms.

Phase 1:

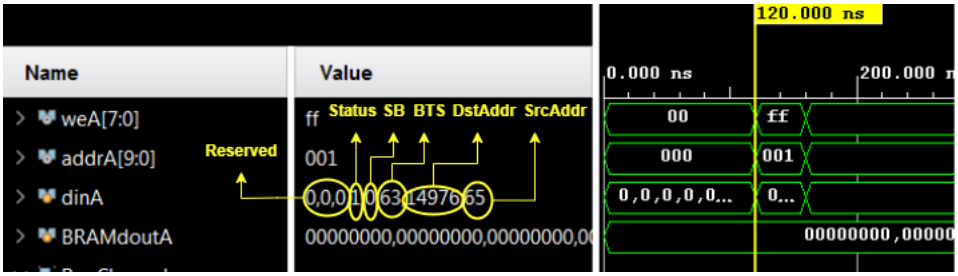

FIGURE 4.5: phase 1: transfer insertion

At the start of the phase, the pseudo-CPU assigns a transfer to DMA at the descriptor address 1 by enabling all the bits of WE input, Fig.4.5. At the same time, the information about the transfer is delivered in DMA from the DataIn input. The values of the fields that compose the Data are SrcAddr: 65, DstAddr: 14976, BytesToSend: 63, Status: 1(Active), and the rest fields, which are SentBytes and Reserved, are 0 as expected. The $SrcAddr \mod 64 (CHI - Data - WIDTH) = 65 \mod 64 = 1$ and $DstAddr \mod 64 = 14976 \mod 64 = 0$ and as the number of bytes that will be sent are 63 the transactions which are expected to be generated are one read and one write because all the bytes that will be read and be written are within only one line respectively.

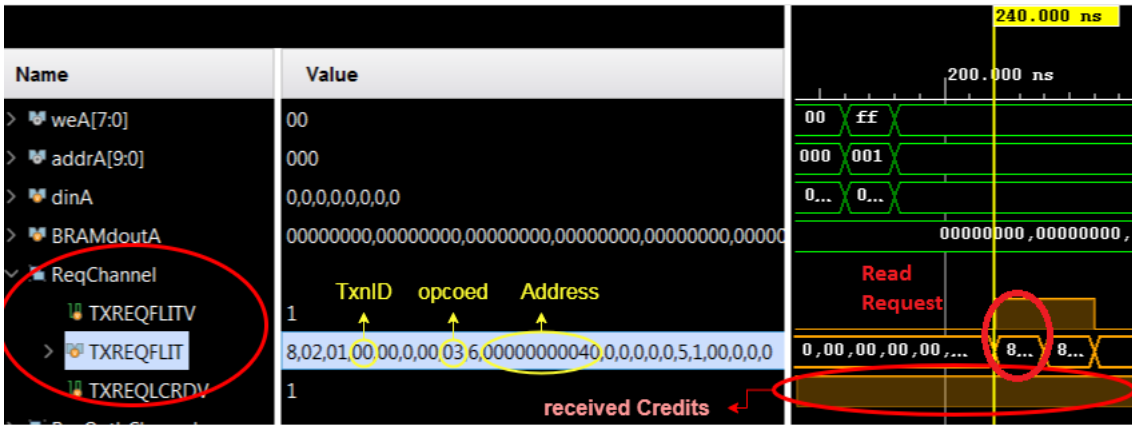

FIGURE 4.6: phase 1: Read Request

After a few cycles, the read request for the inserted transaction is generated by the activation of the TXREQFLITV of the request channel as shown in the figure 4.6. This operation is possible because the DMA has received several credits on the request channel, as the CHI-Responder makes sure to give the maximum number of credits to the DMA. Simultaneously with the assertion of the TXEREQFLITV signal, the appropriate values of the fields that compose the FLIT are set. The first 3 values of the TXREQFLIT vector are the QoS, TgtID, and SrcId which are parameters and can be adjusted according

to the system where the DMA will be used. Subsequently, there is the TxnID which is unique for every transaction, and it is 0 as it is the first request. Then the next 3 fields are always zero, as they are utilized in transactions that the DMA doesn't use. Afterward, there is the opcode field that indicates the type of request with the value of 3 as the transaction is a ReadOnce, and the size field with always the value of six as it indicates that the number of bytes that will be transferred is 64. The next field is the requested read address that has the value of 64 (40_{hex})as it is the previous aligned address of the assigned transfer's SrcAddr with the value of 65. Finally, the rest of the fields are zero or constants, and they are not useful for the transaction.

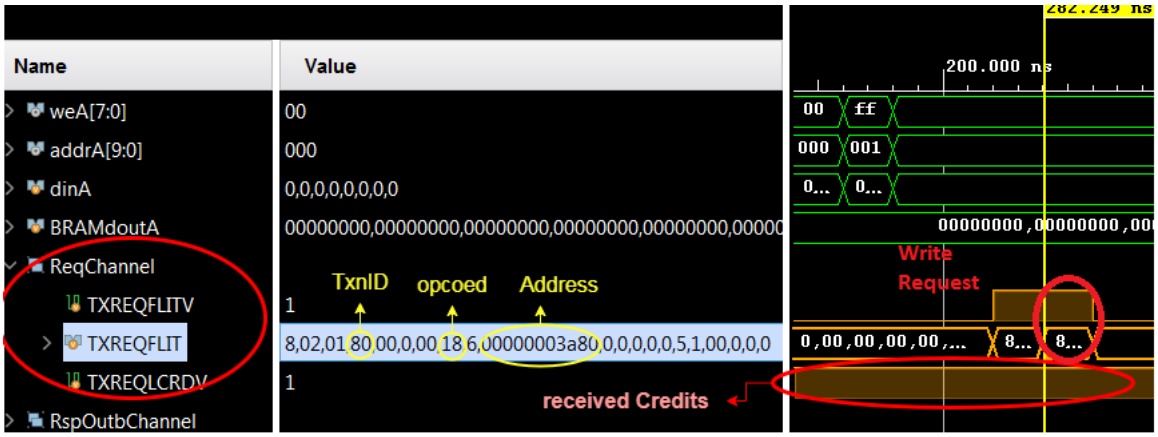

FIGURE 4.7: phase 1: Write Request

The next cycle, as seen in Fig.4.7, DMA generates the write request from the same transfer as the TXREQLCRDV signal was asserted for many cycles, so there are enough credits on the channel, although one credit is already used. The write transaction, in the same way as the read transaction, begins with the activation of the valid signal TXEREQFLITV and the assignment of the appropriate values in FLIT's fields . The only different values of fields in this request are the TxnID which is 128(80_hex) as it must be unique, the opcode which is 24(16_hex) as this is the value that indicates the WriteUniqueuePtl, and the address field which has the same value with the DstAddr 14976 as it is an aligned address (because it is a power of 64(CHI-DATA-WIDH))

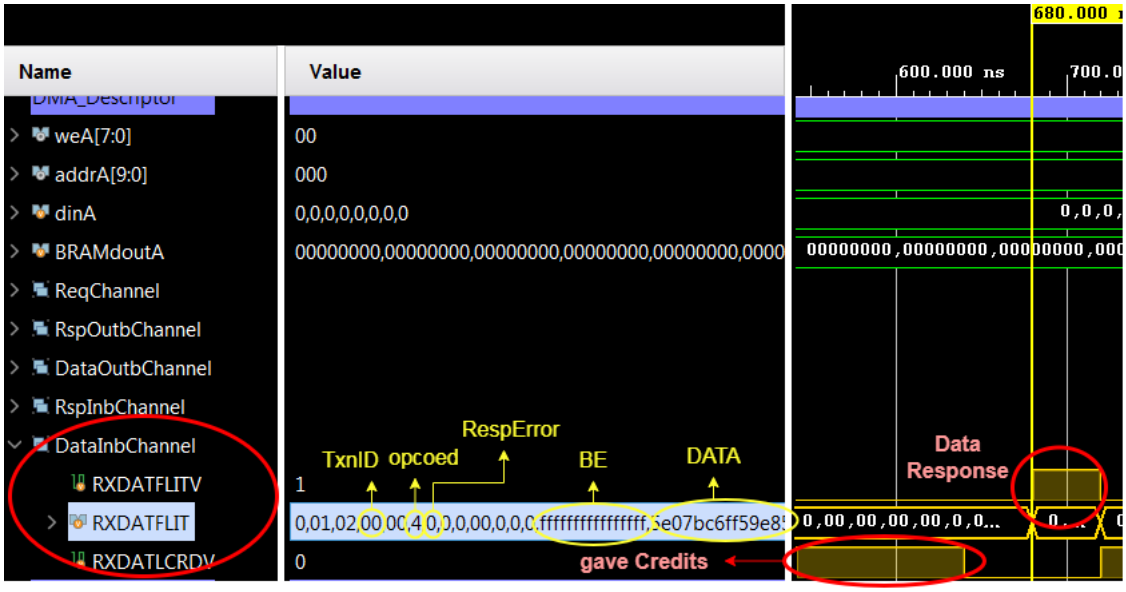

FIGURE 4.8: phase 1: Data Response

Eleven cycles after the read request has been sent and the DMA has sent the necessary credits, the data response arrives on the inbound data channel, Fig.4.8. The DMA notices the presence of the response by observing the valid signal of the channel. With the activation of this signal, the appropriate information is received with the FLIT on the inbound RXDATFLIT vector. As shown in the waveform, the TxnID field of the response that is placed after the Qos, TgtID, and SrcID fields has a value of 0 which is the same value of TxnID that was sent with the request as the CHI protocol requires. Two fields after it is the opcode which has the value of 4 as the response is a CompData. Next, it follows the RespError field, which indicates if there is an error in the response. The value of this field is 0 in this case, so there is no error, but it could be also 2 or 3 which indicates that is a data error or a non-data error respectively. Finally, there are the fields BE which is full of 1 because the requested bytes were 64 as the width of this field, and the Data which are 64 bytes from the address that was requested. The rest of the fields are not used for this transaction. After receiving this reply, the read transaction is over.

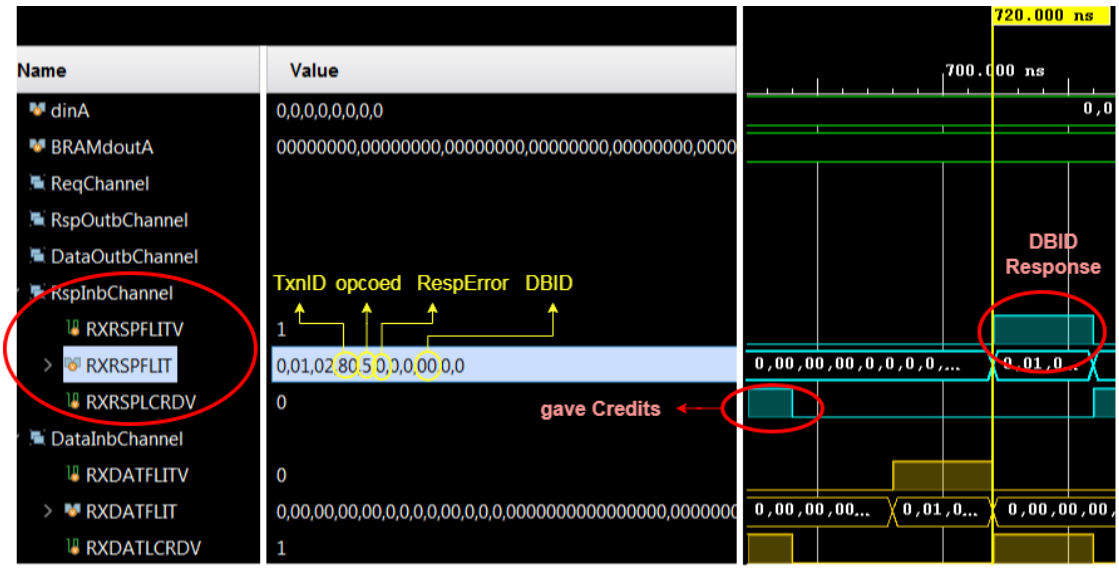

FIGURE 4.9: phase 1: DBID Response

On the next cycle, the DBID Response is received on the inbound Response channel as the write request was generated one cycle after the read and CHI-Responder responds after a constant number of cycles, Fig.4.9. The important fields of the received FLIT are the TxnID with value 128(80_{hex})which is the same as the sent TxnID with the request, the opcode with the value of 5 as the transaction is a CompDBIDResp response, the RespError that is zero as there is no error, but it could be 2 or 3 if there was a problem with the transaction and the DBID which indicates the ID of the data buffer that the CHI system has reserved for the data that will be sent from DMA, and it should be used as TxnID in the final write transmission. The transmission of this response is possible because the DMA sent at least one credit on the inbound response channel earlier.

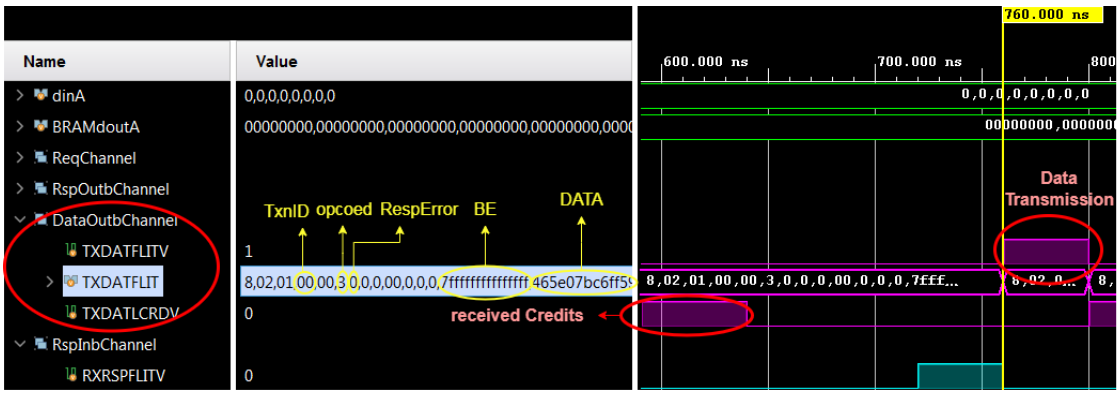

FIGURE 4.10: phase 1: Data Transmission

After both Data and DBID responses have arrived, the DMA generates the data transmission on the outbound data channel, Fig.4.10. This transmission is necessary to send the data that will be written at the requested address, and begins with the assertion of the TXDATFLITV signal. At the same time the TxnID field of FLIT takes the value of the previously received DBID with the response FLIT, the opcode the value of 3 which indicates the NonCopyBackWrData type of transmission, the RespError is zero as there was not any error in the received DBID response, the BE activates all of its bits except the first one as the bytes of data that must be written are the first 63 bytes because DstAddr mod 64 is 0 and the data have the same value with the data that received with read transaction but shifted one byte right. It is worth noting that if a response contains an error, the data transmission will still be executed with an error value in the error field, as CHI requires all transactions with errors to complete in a protocol-compliant manner. After the completion of this transmission, the write transaction has been finished and the transfer of the descriptor has been fully completed.

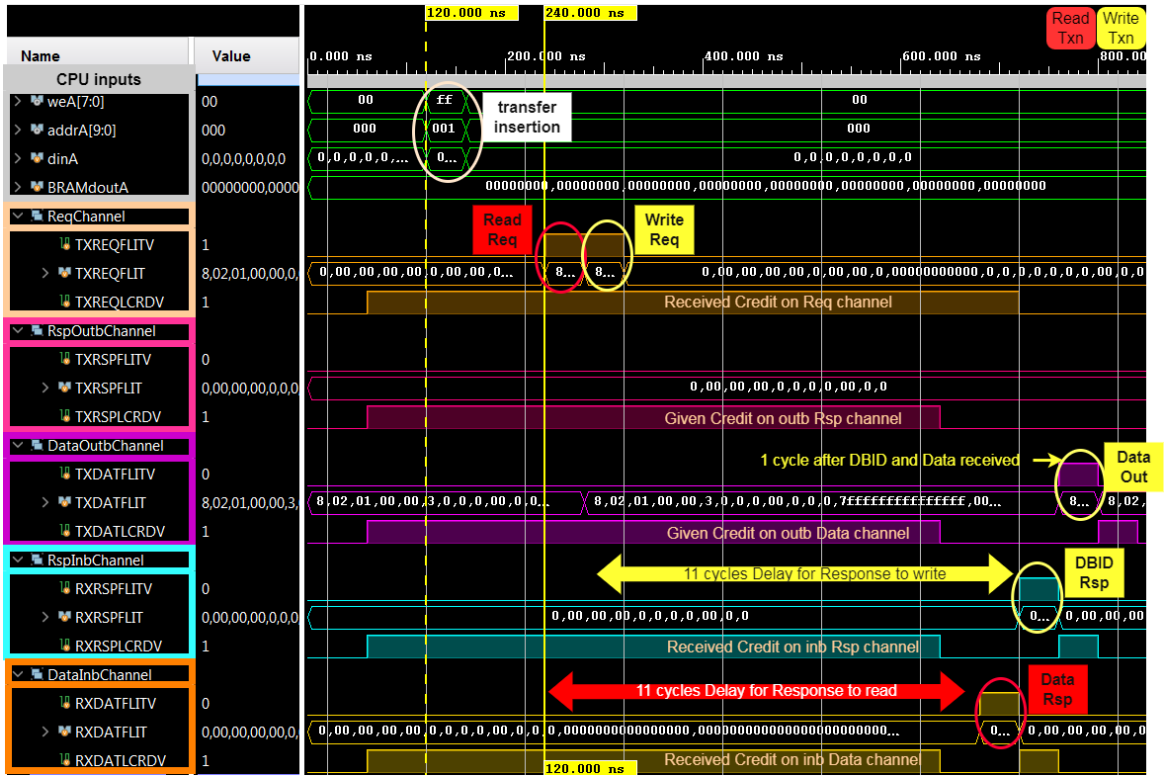

FIGURE 4.11: phase 1

The image Fig.4.11 presents the whole of phase 1 which is the combination of the previously described parts to clarify the timing between the events. When the phase is over, the automatic verification mechanism of TestBench

that is described at Functionality Check also displays the appropriate message to inform if the inserted transfers are executed successfully.

Phase 2:

A few cycles after the transfer of the last phase is finished, the phase changes to phase two. In this phase, the pseudo-CPU sends a transfer to DMA with $DstAddr \mod 64 < SrcAddr \mod 64$ which will need 2 read and one write transactions to be executed for the transfer to be completed. This phase is important to verify the ability of the DMA to generate more read than write transactions for one transfer and to shift, and merge the data properly to create the proper outbound data.

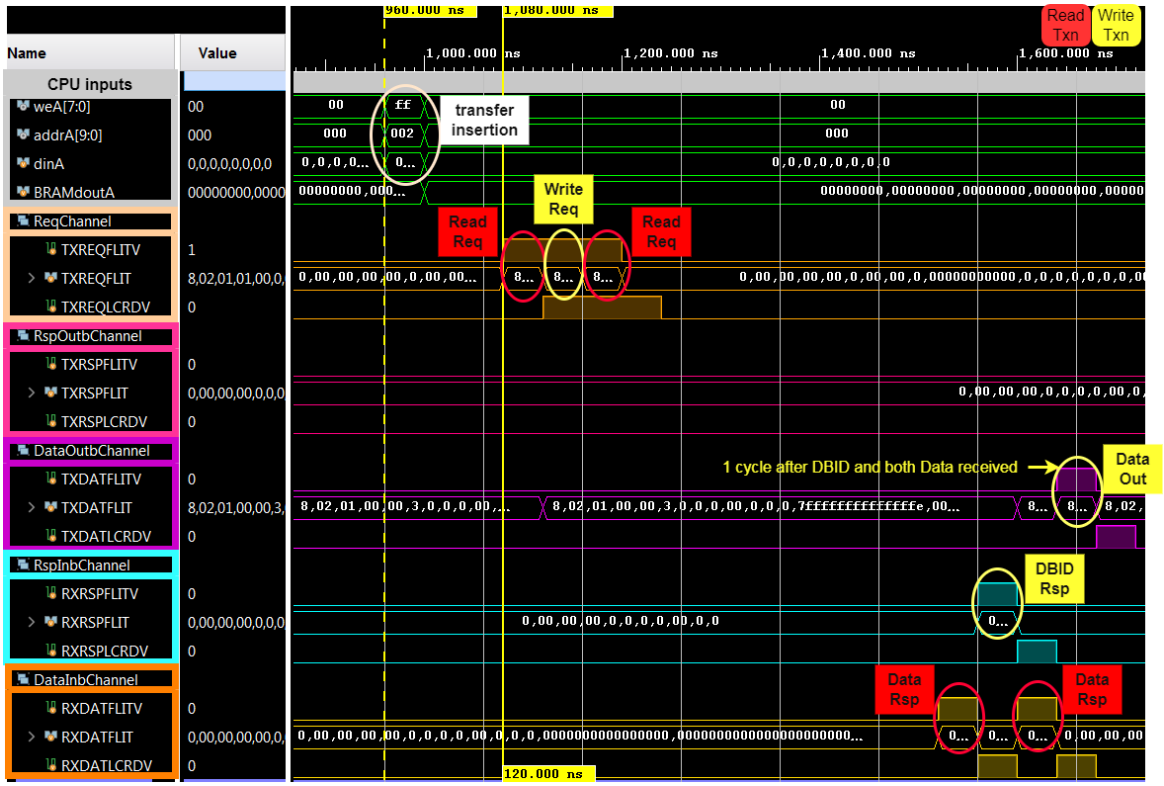

FIGURE 4.12: phase 2

As shown in the Figure 4.12 the phase begins with the insertion of the appropriate transaction in DMA. Three cycles later the requests' procedure starts and the DMA generates sequentially one read one write and extra read requests. Eleven cycles later, the first data response arrives on the inbound Data channel. Then the next cycle, the DBID response is received on the inbound Response channel. However, in this phase, these 2 responses are not sufficient for the data transmission to be generated as there are not enough data and the system waits for the next data response that was requested with

the extra read which arrives on the next cycle. Finally, DMA sends the data on the outbound Data channel, which is created with the shift and merge of the received data from the 2 data responses. Noteworthy is that there are no credit exchanges on any channel before the request and the responses, as both the DMA and the CHI-Responder have sent the maximum number of credits in the last phase, but they send extra credits every time they use some.

Phase 3:

After the completion of the last phase, the test moves to the third phase. In this phase, the pseudo-CPU assigns a transfer to DMA with $DstAddr \bmod 64 > SrcAddr \bmod 64$ and appropriate BTS field, so it will need 1 read and 2 write transactions to be executed for the completion of the transfer. This phase is important, in the same way as the last one, to verify the ability of the DMA to generate more write than read transactions for one transfer and to shift, and merge the data properly to create the proper outbound data.

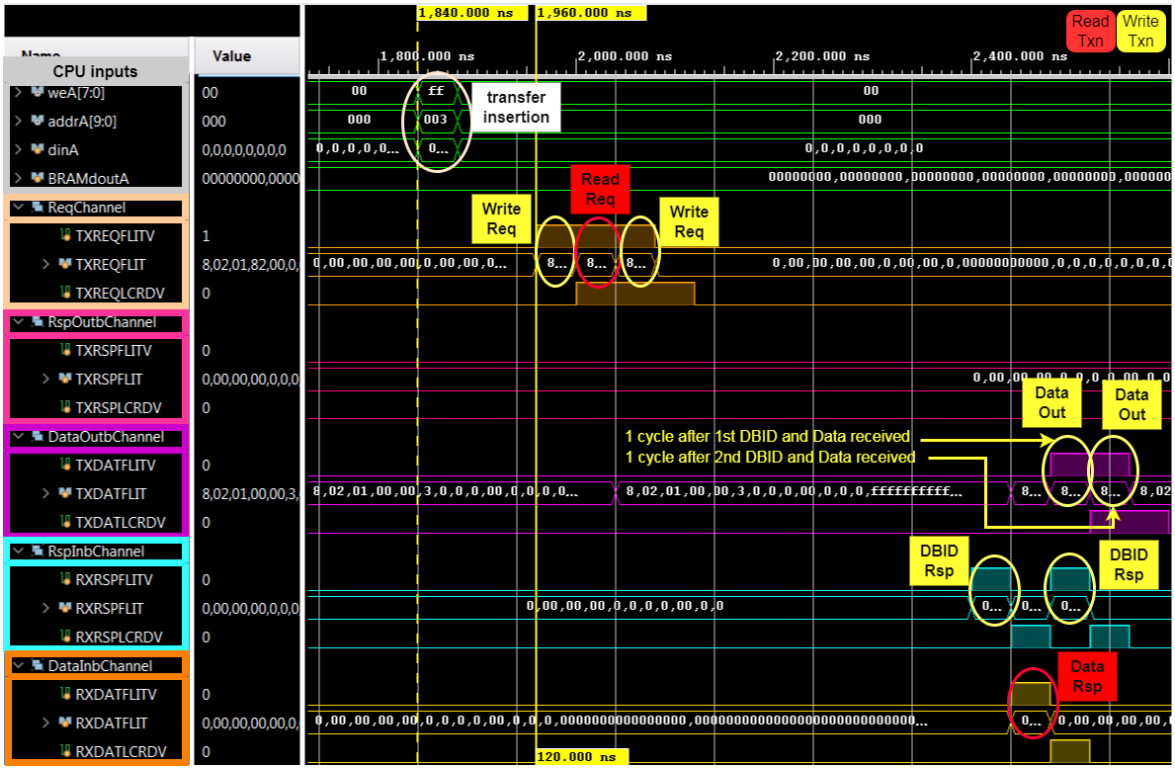

FIGURE 4.13: phase 3

As in every phase, the procedure starts with the pseudo-CPU to insert the transfer in DMA, Fig.4.13. Tree cycles later, the DMA transmits the appropriate requests which are 2 write and one read requests alternately. After

the constant delay of the CHI-Responder the responses arrive in consecutive cycles with one DBID response on the inbound Response channel, one Data response on the inbound Data channel, and one extra DBID response on the inbound Response channel respectively. When the first DBID and Data responses have been received, the DMA adjusts the inserted data and transmits the first data for the write transaction on the outbound Data channel. At the same time when the DMA transmits the data, it also receives the second DBID response. For this reason and as the data that have been read are enough for both writes, the DMA generates on the next cycle the last data transmission with the remaining data. In the same way as the last phase, there are no credits exchanges between the units before the transmissions as the maximum number of credits have been sent from both sides on every channel and all the consumed credits are re-transmitted one cycle after the consumption.

Phase 4:

The next phase that follows is phase 4 where a large transfer is inserted in DMA by the pseudo-CPU. The size of the transfer is 6402 bytes which is generated randomly, so it will need more than 100 read and write transactions for its completion. The $SrcAddr \mod 64$ happens to be smaller than the $DstAddr \mod 64$ as they also produced randomly, but it would not be a big difference if they were in reverse. In this phase will be verified that the transfers which are divided and scheduled in multiple chunks are executed correctly.

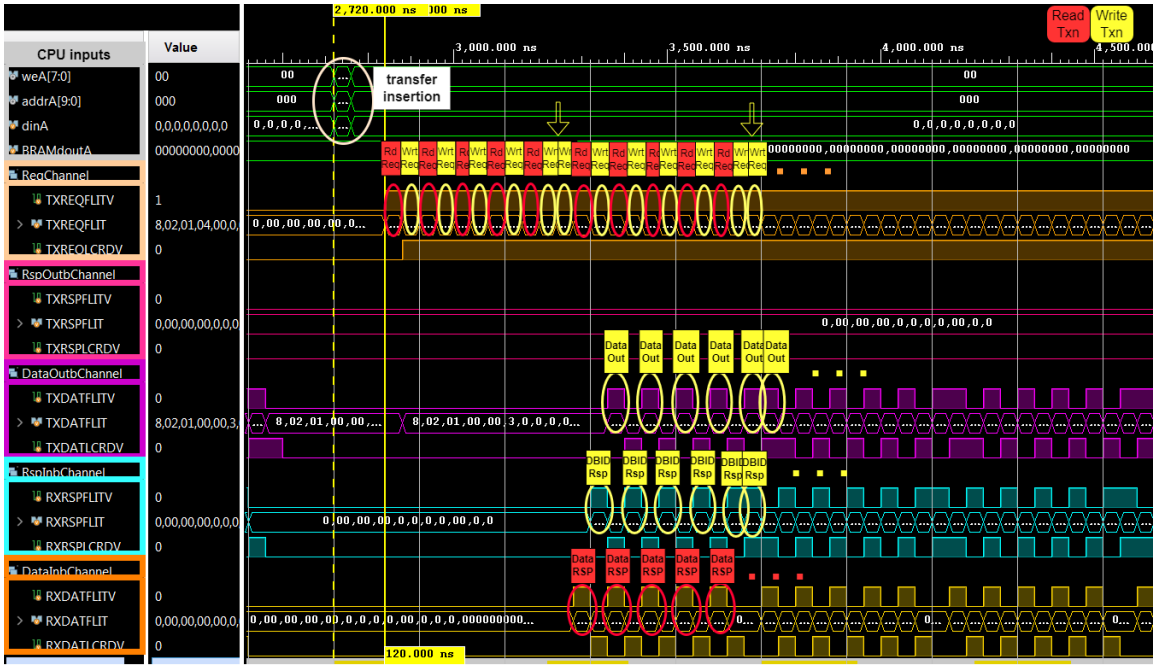

FIGURE 4.14: phase 4

Tree cycles after the insertion of the transfer, the DMA start the transmissions of new requests. The number of request transmissions is much larger compared to the previous phase, as the size of the transfer is much bigger. The transfer is scheduled in chunks of 5 which means that the Length of the command can be up to 320 bytes. For this reason, as shown in Figure 4.14, the DMA generates 5 read and 5 write requests for every command alternately. However, as the addresses of the transfer are misaligned and the data must be shifted left, the DMA needs to request one more write transaction at the end of every command to write the remaining read data. Therefore, the system generates a total of 5 read and 6 write transactions for each command. This phase would be very similar even if the shift was right, with the only difference being that there would be a total of 5 read and 5 write requests for the first command of the entire transfer because the number of needed bytes would be bigger only for the first command. After the constant delay time, the Data and DBID responses arrive on the corresponding channel in the order they requested. The 2 last DBID responses of every command arrive at consequent cycles as there is one extra write request for each command. When one DBID and one Data response are received, the DMA generates the outbound data transmission on the next cycle. Nevertheless, for the last write transaction of each command, DMA doesn't need a Data response to generate a data transmission as there are unsent bytes from the last response.

For this reason, after DMA receives the last DBID response, it generates directly the data transmission, which results in 2 consequent transmissions at the end of every command on the outbound Data channel.

Phase 5:

In this phase, DMA receives a large number(250) of small transfers which need one read and one write transaction to be completed. These transfers have the same form as the transfer in phase 1 but with this phase, we can observe the behavior of DMA when it handles a large number of them, as their small length will prevent the system from performing the optimal scheduling. This is happening because the scheduler of the system is not able to schedule commands with the expected length due to the minor size of every transfer, and the process flows in the following way.

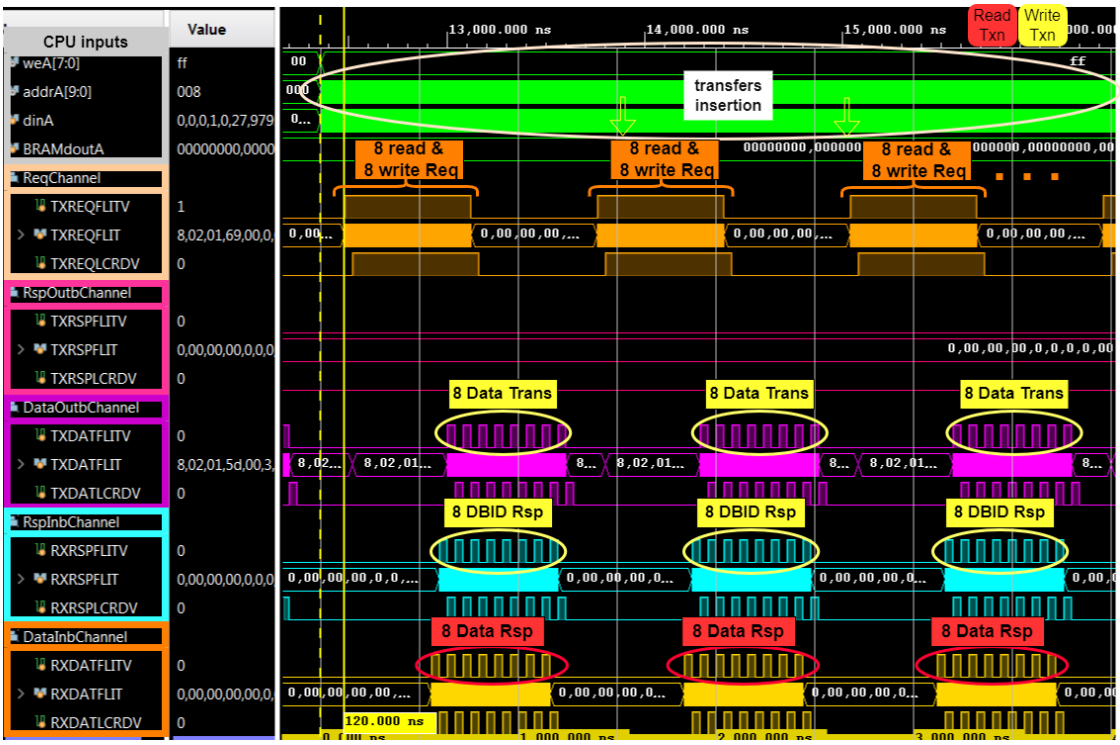

FIGURE 4.15: phase 5

When the phase begins, the pseudo-CPU inserts in DMA a new small transfer on every cycle, Fig.4.15. A little time after the first insertion, the DMA starts generating the appropriate requests. The requests for the first 8 transfers are transmitted normally, with each write after the corresponding read request and the transmission of all requests from every transfer in consecutive cycles. Subsequently, for the next few cycles, the DMA stops requesting and the TXREQFLITV signal deactivates. Afterward, the system activates

the valid signal for 16 cycles which generates the appropriate requests for the next 8 transfers, and then it deactivates again. This pattern continues periodically for the rest of the phase until all the transfers have ended. The reason for this strange behavior of the DMA is that all the transfers are pretty small and the Scheduler of the system is not in time to schedule enough bytes for transfer before it loses access to BRAM and can't read the next transfers to schedule them. The access to BRAM for this module is lost after the system has received the first data and DBID responses and sends the first data transmission. At that moment, one transfer has been fully completed (as all transfers are very small)and the completer of CHI-Converter must update the Status field of the corresponding Descriptor in BRAM. For this reason and as the Arbiter that controls the second port of BRAM always gives a priority to Completer, the Scheduler loses the ability to read the next Descriptor, and thus it can not schedule a new command. In addition, as the CHI-Responder responds after a constant number of cycles, by the time the Status of the Descriptor is updated the DBID and Data responses for the next transfer have arrived, the corresponding data transmission has been sent and the Status for the next Descriptor must be updated. This situation occurs 8 continuous times as the number of requested transfers and for this reason, Scheduler loses the access to BRAM for a significant amount of time which results in commands shortage in CHI-Converter and there is an idle time on the request channel every 16 requests. In conclusion, this phase reveals the inability of the DMA to transmit new requests on every cycle due to the small size of a large number of the inserted transfers, which causes a constant conflict between the Scheduler and the CHI-Converter for the occupation of BRAM's port. Worth noting is that even if the arbiter of BRAM switches the access priorities of modules, the behavior of the system is similar and the results are not better. Nevertheless, the Design of the system could be improved and the Status of the Descriptors could be stored in a different BRAM or array that would abolish the Arbiter of the BRAM and would allow both the Scheduler and CHI-Converter to operate simultaneously which would maximize the performance of DMA in this phase. This optimization can be implemented in future work.

Phase 6:

In this phase, DMA receives a large number(250) of small transfers which need one or two read and one or two write transactions to be completed. The inserted transactions have the same form as the transaction in phases 1,

2, or 3. In the same way as the last one, this phase is important to observe the behavior of the DMA to handle a large number of small transfers but this time they may need a different number of read than write transactions which changes the pattern of waveforms.

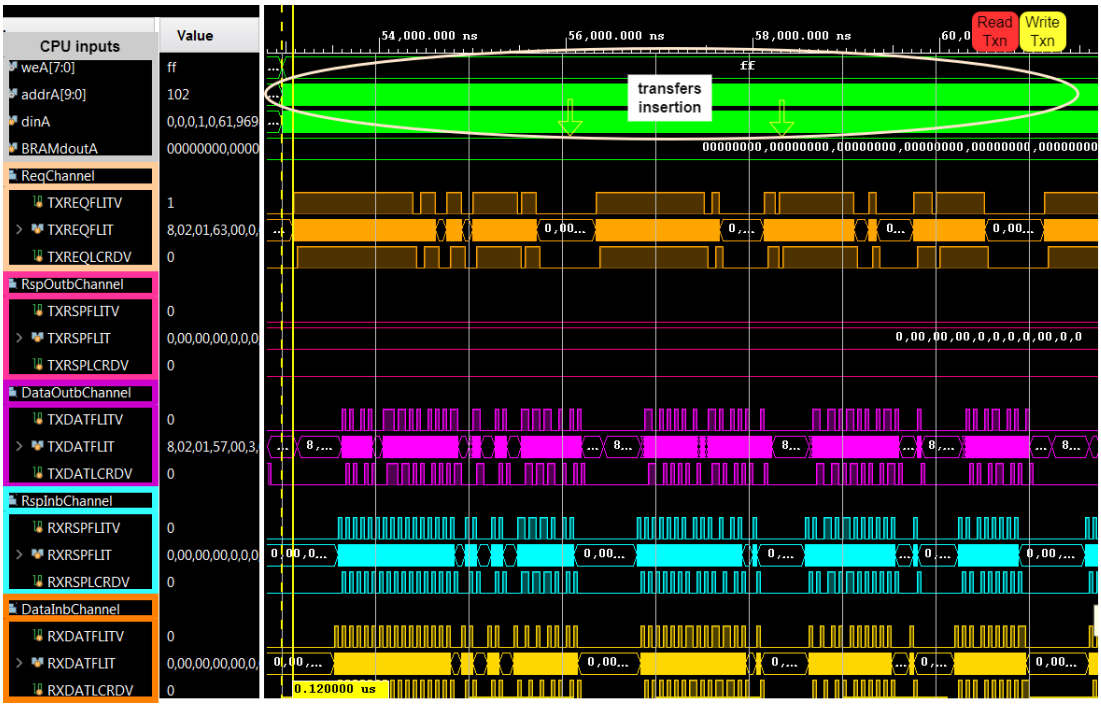

FIGURE 4.16: phase 6

As shown in the waveform Fig.4.16, the DMA in this phase suffers from the same problem as in the last phase. Due to the conflict of the Scheduler with the Completer of CHI-Converter for the control of the second port of BRAM, the CHI-Converter runs out of commands, which causes temporary inability from the module to generate new CHI-Requests. This is represented by the deactivation of the TXREQFLITV signal of the Request channel in the waveform for some time. In addition, by comparing this waveform with the last one it is easy to be noticed that the activity of the request channel is chaotic in contrast with the previous phase where it was periodic because the number of reads and writes transactions in this phase is random for every inserted transfer (it can be one or two read or writes). However, even if there is chaotic behavior the dead time of the request channel is less than phase 5 because even so, the number of overall executed transfers is the same the number of requests is greater due to the probable multi-read-write necessity for each transfer. The problem of the DMA for this phase can be solved in the same way as for the last phase, that is by the implementation of an extra

BRAM or array where it would store only the Statuses of Descriptors, and it would allow the CHI-Converter and Scheduler to operate normally at the same time.

Phase 7: Phase 7 is performed to check the ability of the DMA to handle efficiently multiple large transfers. For this reason, pseudo-CPU assigns 15 transfers to DMA with more than 3000 bytes in length. The Source and destination addresses are chosen to not overlap, but the addresses are random so the shift which is needed for every transfer is different.

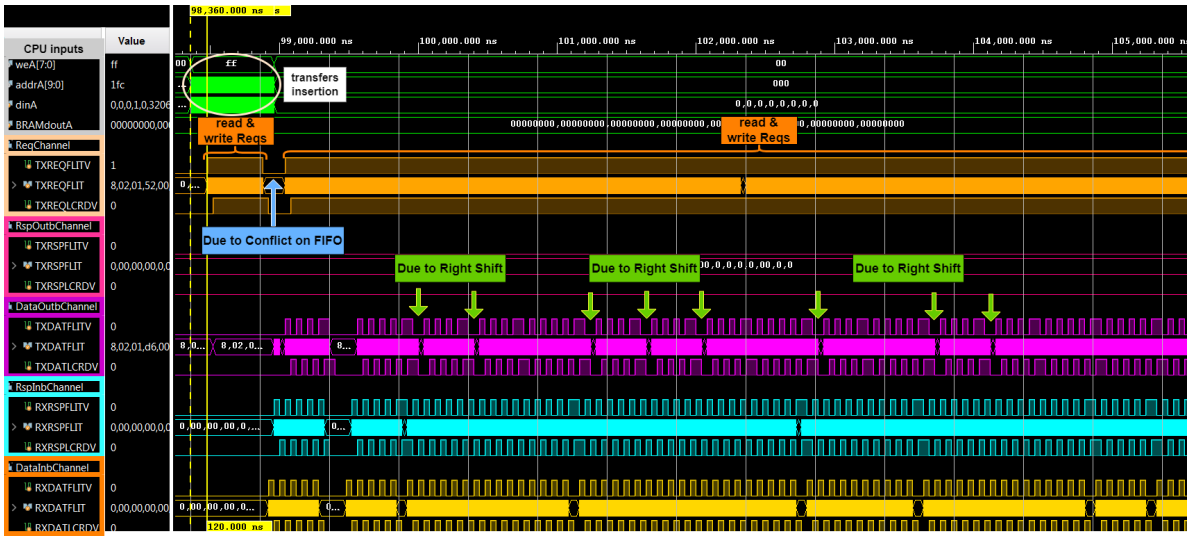

FIGURE 4.17: phase 7

The phase begins with the insertion of all the transfers in continuous cycles, Fig.4.17. Three cycles after the first assignment, the DMA starts to generate the requests for the first command. However, after these requests, there is a small gap in the request channel where the DMA does not generate any requests for a small time. This is happening because the pseudo-CPU is inserting in DMA new transfers non-stop and hence it uses the system FIFO continuously, which prevents the Scheduler from writing the old Descriptor pointer and hence from scheduling new commands. For this reason, the CHI-Converter runs out of commands and stops requesting until all of the transfers have been assigned, and then the operation continues normally. Nevertheless, this behavior of the CPU is not realistic, it is safe to assume that such a case will never happen in a real application and this issue does not constitute a problem. When CHI-Converter receives commands again, the request channel stays active all the time until all the requests are sent. A constant time after each request is transmitted, the DBID or Data response is received on the corresponding channel. Finally, for each write request, the

appropriate received data are modified to be transmitted and written in the correct location. Noteworthy is the fact that there are a few small delays in the outbound transmissions of some data. This is happening on the first command of the transfers which their data must be shifted right because there are not enough data from the first read and the second read must arrive to generate the first outbound data transmission. Except for the first command of each transfer, there is no other command which needs a right shift and creates a delay in the outbound transmission because the Scheduler manages all read addresses after the first command to be aligned(multiple of 64) and hence all the shifts will be left.

Phase 8 and 9:

Phase 8 and 9 is used to verify the DMA in situations where the CPU behaves realistically. In phase 8 the pseudo-CPU inserts in DMA 45 transfers with random length, Source, and Destination addresses which are stored in continuous descriptor addresses. Each insertion in phase 8 happens after a small random delay, which is more realistic as a real CPU would not assign many new transfers on continuous cycles. Similarly, in phase 9 the pseudo CPU assigns random transfers after random delay but this time in random Descriptor addresses. To achieve this, the pseudo-CPU has to read a random Descriptor of BRAM and only if its Status is Idle or Error it can assign the transfer in DMA, else it has to read a different Descriptor. In phase 9, a total of 450 different transactions are assigned in DMA, which forces the DMA to operate for a long time non-stop and its behavior can be verified for long-term bugs.

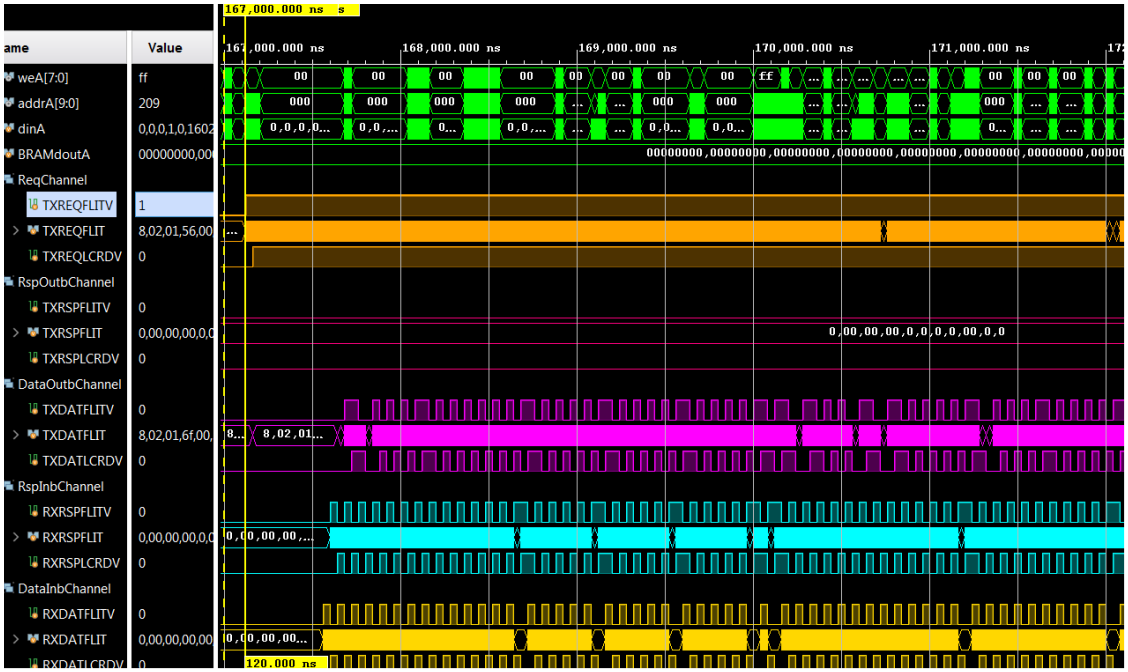

FIGURE 4.18: phase 8,9

For both phases 8 and 9, the DMA operates normally and executes all the transfers successfully. The request channel is always active and there are no gaps as expected for normal cases, Fig.4.18. The responses come after the constant delay that the CHI-Responder creates, and the data transmission is executed when the necessary responses have been received. The small delay that happens occasionally on the outbound data channel is due to the first commands of the transfers which need a right data shift, and it is normal behavior. Finally, as in all other phases, the self-checking mechanism that the Testbench implements prints a message when the phases are over which verifies the correctness of the system.

4.3.2 Stress Testing

To further verify the behavior of the DMA, and to ensure that it will operate correctly in all conditions, the previous simulation where the CHI-Responder provided infinite credits to DMA and responds after constant time is not enough. For this reason, several different tests were applied to the system to check its reliability under extreme cases. The tests that were performed are :

1. Random data transmission and Response time:

In this test, CHI-Responder doesn't give infinite credits to the DMA but it counts the number of issued credits on every channel, and if it is less than the maximum number that CHI protocol allows then it provides a credit on the corresponding channel after a small random delay. This behavior creates a situation where the DMA doesn't have always enough credits to generate new requests and it has to wait until it receives a new credit. In this way, the ability of the system to manage and use the credits properly can be verified in every phase. In addition, in this test CHI-Responder generates the appropriate response to every request that the DMA generates after a small random number of cycles. This operation creates situations where the DMA obtains slightly more DBID than Data responses or in reverse at particular periods of time which allows the verification of the system when the responses are not one-to-one.

2. High latency of data responses:

   In this test, CHI-Responder provides DBID responses approximately ten times faster than the Data responses (there is some randomness in the response time, so the time delay won't be constant). This behavior causes the DBID and command FIFOs of the CHI-Converter to be FULL most of the time, while the Data FIFO is empty. In addition, the Scheduler of the system operates match slower as there is no free space in the FIFO of CHI-Converter to schedule new commands. Also as the Data responses are received at a much slower rate than the DBID responses, the CHI-Converter runs out of TxnIDs for read requests very fast. At the same time its TxnIDs for write requests are plenty that force the module to transmit read and write requests with different rates. Another corner case that appears in some phases, is that the command FIFO of the Barrel Shifter gets full. This occurs because there are not enough data for the module to operate, but the CHI-Converter doesn't stop requesting and passing more commands. In conclusion, this huge difference in the response time by the type of request results in a load imbalance on many units of the system. With this testing, we can ensure that the DMA can handle a probable bad case that could happen with this form.

3. High latency of DBID responses:

   This test is very similar to the previous one with the difference that the

response times are in reverse. This time the DBID responses are produced approximately ten times slower than the Data responses which cause the corresponding problems with the previous test. More specifically, when CHI-Converter has generated enough requests, its DBID FIFO is almost always empty while the Data FIFO of Barrel Shifter is getting full very quickly. Subsequently, the CHI-Converter eventually runs out of TxnIDs for write due to the big response time, and again it generates read and write requests with different rates. Also, in some cases, the CHI-Converter is unable to request because the command of Barrel Shifter gets full, and the system has to wait for the arrival of the DBID response to continue its operation. To conclude, this test was necessary as supplementary to the previous one, and it creates extreme cases in which the system's behavior must definitely be tested.

4. Instant responses:

   In this test, CHI-Responder if there are enough credits on the corresponding channel, transmits the responses on the next cycle after it receives the requests. In this way, the DMA is able to generate requests and data very fast. The test is executed many times with different credit production rates from CHI-Responder which results in CHI-Convert transmitting non-stop or waiting to receive the next credit according to the case. This test is useful to check if the DMA operates correctly when the responses are received in the fastest possible way

5. Sudden changes in response time:

   This test was simulated many times with different credit rates. The CHI-Responder in this scenario responds to the requests very slowly for a period of time, and then it changes its behavior and responds very fast for some time. This process repeats periodically until all phases and hence the simulation is finished. In this way, many components of the system, every time the response rate changes, transition rapidly from being overloaded to empty or unused. This test was performed to verify the reliability of the system when the external factors push the DMA to different extreme cases very fast.

6. Lack of credits:

   In this test, CHI-Responder provides very rarely credits to DMA to check its behavior under this extreme condition. More specifically, the credits are transmitted on each channel after a random but very long

time which is many times larger than the response time. This situation forces the DMA to wait a long time until it obtains new credit for requesting or transmitting data, although it has received the appropriate responses. Also, the lack of credits in combination with the faster responses causes most of the FIFOs of the system to be FULL at the same time for many cycles. As the credits are received so slowly, the simulation needs much time to complete as expected. In conclusion, this test creates multiple corner cases in which the operation of the system was necessary to be verified.

7. BRAM overload:

   This simulation aimed to verify that the system operates properly if the CPU fills the whole BRAM with (1024) transfers. To achieve this, a new transfer was assigned in DMA in continuous cycles until there was no other free space in BRAM as all Descriptors had an Active Status. In this test, the credits and the responses were received after a reasonable amount of time and the Testbench verified that all the transfers were successfully executed.

Chapter 5

Results

5.1 Latency and Throughput

A necessary step in the design process is the performance measurement of the system in order to ensure that the DMA meets the required specifications, is optimized for its specific application, and is cost-effective. Also, this process assists in identifying potential bottlenecks or areas for optimization in the design. In this part, the performance of the DMA is evaluated by measuring its latency which is the time it takes for the core to complete a single operation, and the throughput which is the amount of data it can process in a given period of time. Those quantities were measured by using the waveforms of the simulation, where CHI-Responder provided unlimited credits to the DMA, and it responded after a constant number of cycles by the time it received each request. The reason for the selection of this simulation is that the transmission time of the CHI requests and data is not affected by latencies from external factors, and the speed of the design depends only on its performance and the boundaries of the protocol.

Latency:

First of all, the simulation was executed with a 40 nanoseconds clock, which is a reasonable speed that the DMA could clearly perform, and subsequently the results will be converted to the maximum possible clock frequency. As shown in the waveforms of the last chapter, the DMA starts the transmission of the requests in every phase 120 nanoseconds after the first transfer insertion without having credit dependencies. This behavior implies that the latency of the DMA for generating requests is constant, and it is 3 cycles that equal 120 nanoseconds. In the same way, it is easy to calculate the latency of the outbound data transmission. This latency depends on the response time

of the CHI-Responder as both responses of read and write transaction are needed for one data transmission and the type of the transfer. More specifically, the delay time until the outbound data transmission is 640 ns for the transfers which need a left shift and 680 ns for those that need a right shift. These times result from the request latency (120) + 12 or 13 cycles, which is the response time for the first 2 or 3 requests + 1. The extra cycle in the latency for the right shift is expected, as the DMA needs one more response to generate the first data transmission. Finally, the latency until the finish of all requests or the whole transfer is not constant, as it depends on the size of the transfer.

Throughput:

The throughput of the system was measured by counting the amount of requested or transmitted data within a period of time. However, the behavior of the DMA and the amount of processed data are different according to the phase. For this reason, the throughput is measured separately for each phase of the simulation. There is no point in measuring the throughput in the first 3 phases as the transfers are very small and only a few transactions are needed for their completion, hence the measurement begins in phase 4.

By looking at the waveform of phase 4, it is easy to notice that the requesting throughput is the maximum possible, as the DMA requests on every cycle until all requests are sent. This makes the requesting throughput: $1\frac{req}{cycle} = \frac{10^9}{40}\frac{req}{sec} = 25 \cdot 10^6 \frac{req}{sec}$ Although the request throughput is maximum, the throughput of the overall transferred data doesn't depend only on requests, but it also relies on the type of transfer. If the source and destination addresses of the transfer are aligned then the transmission of data throughput is the maximum possible which is 64 bytes every 2 cycles. Unfortunately, the transfer can't be executed faster than this because one data transmission needs 2 requests (1 read and 1 write) and therefore at least 2 cycles. Hence, the throughput of phase 4, if the data was aligned, would be: $\frac{64}{2}\frac{bytes}{cycle} = \frac{64}{80}\frac{bytes}{ns} = \frac{64}{80 \cdot 10^{-9}}\frac{bytes}{sec} = 8 \cdot 10^8 \frac{bytes}{sec} = \frac{80 \cdot 10^8}{1024^2}\frac{MegaBytes}{sec} = 762,9 Mbps$. However, the transfer in this phase is not aligned which creates the need for one extra write transmission in every command, and for this reason, $5 \cdot 64$ bytes which is the amount of data that is transferred for each command are transmitted every 11 cycles. Therefore, the useful throughput is: $\frac{5 \cdot 64}{11}\frac{bytes}{cycle} = \frac{320}{440}\frac{bytes}{ns} = 0,727 \cdot 10^9 \frac{bytes}{sec} = \frac{727 \cdot 10^6}{1024^2}\frac{Mb}{sec} = 693 Mbps$

In the same way, the throughput of the next phase can be calculated. In phase 5 neither the request nor the data transmission throughput is the maximum as there is the conflict between the Scheduler and CHI-Converter about the control of BRAM. The consequence of this problem is that the DMA generates requests for the first 16 cycles, and then it stops for the next 16 cycles. This behavior continues periodically until all the requests are sent. With this in mind, it is easy to measure the requesting throughput by calculating the number of requests that are generated in a certain amount of time. The throughput of requests in this phase is : $\frac{16}{32}\frac{requests}{cycle} = 0.5\frac{req}{cycle} = \frac{5\cdot10^8}{40}\frac{req}{sec} = 125\cdot10^5\frac{req}{sec}$. As the sent requests are 16 in each period and as one data transmission needs exactly 2 responses to be generated because each transfer in this phase will be moved in one line respectively, it is logical that the outbound data responses are also transmitted periodically with 8 transfers in each period. More specifically, for the first 16 cycles in one period, the system generates 1 data transmission every 2 cycles. Subsequently, the system doesn't generate any transmissions on the next 16 cycles, and then the same process repeats again. Hence, the throughput of the system in this phase is: $\frac{8\cdot64}{16+16}\frac{bytes}{cycle} = 16\frac{bytes}{cycle} = \frac{16\cdot10^9}{40}\frac{bytes}{sec} = \frac{4\cdot10^8}{1024^2}\frac{Megabytes}{sec} = 381,5Mbps$. It is easy to observe that the throughput in this phase is half of the maximum throughput that was calculated for the last phase which makes sense as the DMA in this phase requests and transmits half of the time.

Phase 6 is basically a worse phase 5 as there is the same problem of the conflict on BRAM but also the transfers may need more than 2 transactions to be completed. For this reason, the data throughput is expected to be worse. Nevertheless, the throughput of the requests seems to be better than phase 5 because the number of requests for each transfer is bigger and the corresponding idle time of the channel of the last phase can be used to perform extra requests. The throughput of the requests is approximately: $\frac{97}{133}\frac{requests}{cycle} = \frac{97}{5320}\frac{requests}{ns} = 18,23\cdot10^6\frac{requests}{sec}$. In the same way, the number of data transmissions per time is bigger than in the last phase. However, each transmission does not contain the maximum possible useful information due to the bad scenario of this phase and the throughput ends up being worse. The useful data throughput of this phase is : $\frac{250\cdot64}{1116}\frac{bytes}{cycle} = 14,33\frac{bytes}{cycle} = \frac{14,33\cdot10^9}{40}\frac{bytes}{sec} = \frac{0,36\cdot10^9}{1024^2}\frac{Megabytes}{sec} = 343Mbps$.

Finally, for the last three phases the DMA operates with the maximum possible throughput. This is obvious by looking at the activity of the request and the outbound data channel. A first intuition for the maximum throughput is

that, in all of these 3 phases, a new request is transmitted on every cycle until all requests have been sent. The small dead time of the channel in phase 7 due to continuous insertion by CPU is ignored as it is not a realistic behavior. For this reason, the requesting throughput in these phases is the maximum which is: $1\frac{req}{cycle} = \frac{10^9}{40}\frac{req}{sec} = 25 \cdot 10^6\frac{req}{sec}$. Subsequently, the data throughput is calculated. By the waveforms, it can be observed that the DMA generates one data transmission every 2 cycles, and at the end of each command one extra data transmission. This behavior implies that the data are transmitted in the most efficient way on the outbound data channel as it is impossible to generate more useful transmissions due to the limitation of one request per cycle and the need of two requests for one transmission. Therefore, the data throughput is: $\frac{5 \cdot 64}{11}\frac{bytes}{cycle} = \frac{320}{440}\frac{bytes}{ns} = 0,727 \cdot 10^9 bytessec = \frac{727 \cdot 10^6}{1024^2}\frac{Mb}{sec} = 693Mbps$. The first command of each transfer that needs a right shift of data has one less transmission than the rest of the commands, hence there is one cycle where the system doesn't transmit but its effect is negligible on the overall throughput.

Phases	Request Latency	Transmission Latency	Request Throughput	Transmission Throughput
phase 1	120 ns	640 ns	x	x
phase 2	120 ns	680 ns	x	x
phase 3	120 ns	640 ns	x	x
phase 4	120 ns	640 ns	$25 \cdot 10^6\frac{req}{sec}$	693 Mbps
phase 5	120 ns	640 ns	$12,5 \cdot 10^6\frac{req}{sec}$	381 Mbps
phase 6	120 ns	640 ns	$18,2 \cdot 10^6\frac{req}{sec}$	343 Mbps
phase 7	120 ns	680 ns	$25 \cdot 10^6\frac{req}{sec}$	693 Mbps
phase 8	120 ns	680 ns	$25 \cdot 10^6\frac{req}{sec}$	693 Mbps
phase 9	120 ns	680 ns	$25 \cdot 10^6\frac{req}{sec}$	693 Mbps

TABLE 5.1: Performance

From the table, it can be seen that the system's latency is constant in contrast with the throughput, which is different according to the situation. In most cases, the system approaches the maximum possible throughput which is 693 Mbps, but its weakness is revealed in phases 5 and 6 where the transfers are very small, and the throughput is about half of the maximum with the worst case being phase 6 with 343 Mbps where all transfers are very small and need multiple transactions. It is worth reiterating that, as previously stated in the

simulation overview of phase 5 and 6, the problem could be resolved in the future by preventing Scheduler and Completer trying to access the same port of BRAM, through arbiter, and then the throughput could be the maximum one in every case. It is important to emphasize that the results showed in table 5.1 is a minimum boundary of the realistic results, as the clock used in the simulation is relatively slow (40 ns). However, these results are utilized to calculate the realistic results described in the subsection 5.1.1.

5.1.1 Synthesis results and realistic performance

The HDL implementation of the designed DMA engine is constructed to be synthesizable. The RTL-specified implementation was successfully synthesized and analyzed by the synthesis tool of vivado. The FPGA part used for the synthesis is "xcvu37p-fsvh2892-2L-e", which belongs in the Virtex UltraScale+ FPGA family of Xilinx manufacturer. This device is selected because it is one of the most high-end FPGA available in vivado tool where the design could be synthesized optimally. The FPGA is considered a large FPGA as it provides many resources, as 1.3 million CLB LUTs and 2.6 million CLB Regs. After synthesis was completed, timing summary results showed that the critical path of the system which was located in the communication between Barrel Shifter and CHI-Converter with 17 levels of logic and a 4.367 ns total delay. The critical path makes sense to be located in this position, as the control of the Barrel Shifter is quite big and needs to execute many combinational calculations to generate the appropriate signals for CHI-Converter. The total delay is an approximation of the minimum time that is necessary for the clock to spread to every endpoint of the circuit. With this specification it is easy to calculate the maximum clock frequency that can be applied to the circuit without affecting its behavior which is: $\frac{1}{4.367 \cdot 10^{-9}} Hertz = 0,22899 \cdot 10^{9} Hertz = 228,99 MHz$ and the minimum clock period which is: 4.367 ns. Also, by using the minimum clock period and adding the clock uncertainty which results in about 4.4 ns, compared with the clock used in the simulation: $\frac{4,4}{40} = 0.11$, the latency and throughput that would have been produced for every phase can be calculated, converting the results from simulation.

Phases	Request Latency	Transmission Latency	Request Throughput	Transmission Throughput
phase 1	13.2 ns	70,4 ns	x	x
phase 2	13.2 ns	74,8 ns	x	x
phase 3	13.2 ns	70,4 ns	x	x
phase 4	13.2 ns	70,4 ns	$227 \cdot 10^6 \frac{req}{sec}$	6300 Mbps
phase 5	13.2 ns	70,4 ns	$114 \cdot 10^6 \frac{req}{sec}$	3464 Mbps
phase 6	13.2 ns	70,4 ns	$165 \cdot 10^6 \frac{req}{sec}$	3118 Mbps
phase 7	13.2 ns	74,8 ns	$227 \cdot 10^6 \frac{req}{sec}$	6300 Mbps
phase 8	13.2 ns	74,8 ns	$227 \cdot 10^6 \frac{req}{sec}$	6300 Mbps
phase 9	13.2 ns	74,8 ns	$227 \cdot 10^6 \frac{req}{sec}$	6300 Mbps

TABLE 5.2: Maximum Performance

5.2 Resources

As already mentioned, the FPGA part which is used for this project is "xcvu37p-fsvh2892-2L-e". With this FPGA, the synthesis tool can provide an estimation of the resource utilization that the RTL design will commit. In the table below there are the resource requirements of the DMA and also an approximation for the individual utilization portion of each module.

As shown in the table5.3, the modules with the most demands for space are Barrel Shifter, CHI-Converter, and the system FIFO. The reason for the size of the FIFO is the large number of elements that it can store before it becomes full, which is necessary to be able to store as many pointers as the BRAM's addresses which justifies the result. Moreover, the CHI-Converter was expected to be expensive in resources as it contains many FIFOs and all the functioning logic that is necessary for interaction with the external CHI system. However, the most costly module in the DMA is the Barrel Shifter. Barrel Shifter is so demanding for space because it handles and manages the datapath. It contains the widest FIFO as it stores the received data which are 64 bytes and it has to perform operations on these data such as shifting, merging, and temporarily storing them which leads to the module being the most expensive in utilization. With this information, it is easy to conclude that the significant factor for the space requirements of the system is the datapath. The rest of the modules which are the 2 Arbiters, the Scheduler, and

BRAM occupy the least space as they perform simpler operations.

Resource	Utilization	Available	Utilization %
LUT	14262	1303680	1.09
FF	33693	2607360	1.29
BRAM	8	2016	0.40

FIGURE 5.1: Total Utilization(%)

The overall utilization of the DMA, as shown in Fig.5.1 which is produced by the synthesis tool, is very little compared to the resources of the FPGA. The percentage of the committed LUTs (Look Up Tables) and FF (Flip Flops) is close to one percent of the total availability, and the used BRAM Tiles are less than one percent of the maximum possible.

Name	CLB LUTs	CLB Regs	CARRY 8	F7 Muxes	F8 Muxes	Block RAM Tile	Utiliza-tion %
System FIFO	3965	10357	2	1360	680	0	LUTs 28% FF 30 % BRAM 0%
Scheduler	260	12	24	0	0	0	LUTs 2% FF 0,5 % BRAM 0%
CHI-Converter	1878	4407	42	528	0	0	LUTs 13% FF 13,5 % BRAM 0%
Barrel Shifter	8093	18917	36	2276	0	0	LUTs 56% FF 56,6 % BRAM 0%
Arbiter-FIFO	11	0	0	0	0	0	LUTs 0,1% FF 0 % BRAM 0%
Arbiter-BRAM	45	0	0	0	0	0	LUTs 0,3% FF 0 % BRAM 0%
BRAM	0	0	0	0	0	8	LUTs 0% FF 0 % BRAM 100%
TOTAL	14262	33693	104	4164	680	8	
Available	1303680	2607360	162960	651840	325920	2016	

TABLE 5.3: Utilization

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presents a comprehensive design and verification of a Direct Memory Access (DMA) engine IP-Core. The DMA controller intends to improve coherent High-Performance Computing systems(HPC), by taking over the responsibility for memory data movements. The proposed DMA has been designed to receive instructions for a parameterizable amount of transfers, schedule them in a generic way, and execute the transmissions in a CHI-compliant manner. This engine is an IO Coherent Requester which reads data from the last snapshot of memory, even if they are placed inside the cache of another component by exploiting the cache coherence feature that CHI provides, and write them back in a different memory location forcing the CHI interconnect to inform other devices for the changes if that is necessary. The design of the DMA also contains a Barrel-Shifter module capable of handling misaligned transfers by shifting the read data and creating the appropriate data for write transactions which makes the engine able to read and write in memory at any address byte offset. When one of the transfers is completed or there is an error in the operation, the DMA updates the corresponding status register, so the processor can be notified about the state of the transfer by polling the register. The DMA controller is implemented and verified in system-verilog HDL. For the verification, a series of tests were applied to the system using behavioral simulation to check most of the corner cases and confirm that the system is error-free. From the simulation, the latency and throughput that do not depend on the latency of the memory or interconnect were measured. In the majority of cases, the throughput was observed to reach its maximum potential. Finally, the engine was synthesized

by the vivado synthesis tool, and the minimum possible clock cycle and utilization were measured. The highest achievable clock frequency could be 228,99 MHz while utilization of FPGA resources by the design was found to be less than 1% of the total available resources.

6.2 Future Work

The development of a DMA IP Core is a complex and challenging process especially combined with a complex protocol like CHI that requires significant specification knowledge. Here we list potential optimizations and ideas for future development of the IP Core presented in this thesis that will improve the engine.

- Creation of a register space out of BRAM for the storing of Statuses so Scheduler and Completer modules will not conflict for the access on BRAM and perform the maximum throughput in all cases.
- Acknowledgement support from the DMA side, so it can be compatible with peripherals or devices that requires it.
- Support for out-of-order receipt of CHI responses as this version operates correctly only for in-order responses.
- Make the slave side of the DMA CHI compliant, so the processor can access BRAM and assign transfers to the engine via the CHI interface.
- Implementation of Scatter-Gather mechanism for more efficient transfer of data when they are stored in non-contiguous memory locations.
- Support for handling the Retry response that the CHI interconnect could potentially provide.
- Physical realization of the design(in FPGA, or ASIC, etc.) and testing within a real system with a processor, memory, and CHI interconnect.

References

[1] Yasha Jyothi M Shirur, Kritika M Sharma, and Aishwarya A. "Design and implementation of Efficient Direct Memory Access (DMA) Controller in Multiprocessor SoC". In: *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*. 2018, pp. 1–6. DOI: 10.1109/ICNEWS.2018.8903991.

[2] S. Gupta and L. Natarajan. "Optimizing Embedded Applications using DMA". In: *EE Times Design* 15 (Nov. 2010). Link: https://www.embedded.com/...

[3] Jason Andrews. *Optimization of Systems Containing the ARM CoreLink CCN-504 Cache Coherent Network*. Nov. 2014. URL: https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/posts/optimization-of-systems-containing-the-arm-corelink-ccn-504-cache-coherent-network.

[4] Synopsys. *Verifying ARM AMBA 5 CHI Interconnect-Based SoCs Using Next-Generation VIP*. URL: https://www.synopsys.com/designware-ip/technical-bulletin/verifying-arm-amba.html.

[5] ARM. *ARM CoreLink CMN-600*. URL: https://developer.arm.com/Processors/CoreLink%20CMN-600.

[6] Tiago Mück. *gem5 Ruby CHI Protocol Documentation*. URL: https://www.gem5.org/documentation/general_docs/ruby/CHI/.

[7] Wikipedia. *IBM Personal Computer — Wikipedia, The Free Encyclopedia*. [Online; accessed 22 June 2023]. 2023. URL: https://en.wikipedia.org/wiki/IBM_Personal_Computer.

[8] Wikipedia. *IBM System/360 — Wikipedia, The Free Encyclopedia*. [Online; accessed 22 June 2023]. 2023. URL: https://en.wikipedia.org/wiki/IBM_System/360.

[9] Chao Lu, Haosen Yang, and Qi Wu. "Design and Implementation of a Direct Memory Access ControllerBased on Microcontroller Unit". In: *Journal of Physics: Conference Series* 2221.1 (May 2022), p. 012016. DOI: 10.1088/1742-6596/2221/1/012016. URL: https://dx.doi.org/10.1088/1742-6596/2221/1/012016.

[10] Guoliang Ma and Hu He. "Design and implementation of an advanced DMA controller on AMBA-based SoC". In: *2009 IEEE 8th International Conference on ASIC*. 2009, pp. 419–422. DOI: 10.1109/ASICON.2009.5351258.

[11] Abdullah Aljumah and Mohammed Altaf Ahmed. "AMBA Based Advanced DMA Controller for SoC". In: *International Journal of Advanced Computer Science and Applications* 7.3 (2016). DOI: 10.14569/IJACSA.2016.070326. URL: http://dx.doi.org/10.14569/IJACSA.2016.070326.

[12] Altaf Ahmed, Abdullah Aljumah, and M Ahmad. "Design and Implementation of a Direct Memory Access Controller for Embedded Applications". In: *International Journal of Technology* 10 (Apr. 2019), p. 309. DOI: 10.14716/ijtech.v10i2.795.

[13] *AXI Central Direct Memory Access LogiCORE IP Product Guide (PG034)*. Xilinx. May 2022. URL: https://docs.xilinx.com/r/en-US/pg034-axi-cdma/AXI-DMA-v4.1-LogiCORE-IP-Product-Guide-PG034.

[14] Meet Dave and Santosh Jagtap. "Design and Verification of Configurable Multi-channel DMA controller". In: *International Journal of Advance Research and Innovative Ideas in Education* 3 (2017), pp. 4698–4704.

[15] Chetan Sharma and D. K. Chauhan. "High performance low power AHB DMA controller with FSM decomposition technique". In: *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*. 2017, pp. 456–461. DOI: 10.1109/ICPCSI.2017.8392337.

[16] Miquel Roset Julia. "Extending a modern RISC-V vector accelerator with direct access to the memory hierarchy through AMBA 5 CHI." B.S. thesis. Universitat Politècnica de Catalunya, 2022.

[17] Matheus Cavalcante et al. "Design of an open-source bridge between non-coherent burst-based and coherent cache-line-based memory systems". In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 2020, pp. 81–88.

[18] ARM Limited. *AMBA 5 CHI Architecture Specification (ARM IHI 0050C)*. Manual. 2018. URL: https://developer.arm.com/documentation/ihi0050/c/?lang=en.