DMA Controller for a Custom Embedded System

Technological Educational Institute of Crete

Department of Informatics Engineering

by

Spyros Chiotakis

30 October, 2015

Abstract

A Direct Memory Access (DMA) Controller offloads a processor from tasks that involve transferring of data inside the computing system. The processor commands the DMA controller to initiate the appropriate transactions. While the transactions are done by the DMA Controller on the background the Central Processing Unit (CPU) is free to return to the tasks it has to complete until it gets interrupted by the DMA when the transfers finish.

An implementation of a DMA Controller was done during this Bachelor thesis on a Zynq-7000 System on Chip (SoC). It's intended use is for systems that support Advanced Microcontroller Bus Architecture (AMBA) and it's Advanced eXtensible Interface (AXI).

Features of the controller include an AXI4-Lite slave interface in order to be programmed by the processor and an AXI4-Full master interface for maximum bandwidth in the transactions. Additionally a Scatter-Gather interface is included for descriptor-based transfers from scattered memory addresses. Furthermore, multiple channels are implemented with priority scheduling in order to accommodate more than one transactions requests. Lastly, implementation of interrupt support to inform the CPU when transactions finish.

Thesis Supervisor: George Kornaros
Title: Assistant Professor at Department of Informatics Engineering, TEI of Crete

Σύνοψη

Ένας ελεγκτής άμεσης προσπέλασης μνήμης έχει ως στόχο, την ελάφρυνση του επεξεργαστή, από τις μεταφορές δεδομένων μέσα στο σύστημα. Ο επεξεργαστής διατάζει τον ελεγκτή άμεσης προσπέλασης μνήμης να ξεκινήσει της μεταφορές που πρέπει να γίνουν. Όσο γίνονται οι μεταφορές από τον ελεγκτή ο επεξεργαστής είναι ελεύθερος να κάνει άλλες δουλειές που του έχουν ανατεθεί μέχρι ο ελεγκτής να τον διακόψει και να τον ενημερώση ότι οι μεταφορές έγιναν επιτυχώς.

Μία υλοποίηση ενός τέτοιου ελεγκτή έγινε κατά την διάρκεια αυτής της πτυχιακής εργασίας πάνω στο σύστημα Zynq-7000 System on Chip. Η προβλεπόμενη χρήση του είναι για συστήματα που υποστηρίζουν το πρωτόκολλο AMBA και το AXI4 interface του.

Τα χαρακτηριστικά του ελεγκτή περιλαμβάνουν μία διεπαφή AXI4-Lite slave με σκόπο τον προγραμματισμό των καταχωριτών της μέσω του επεξεργαστή, και μία διεπαφή AXI4-Full master για μέγιστο εύρος ζώνης στις μεταφορές. Επιπλέον περιλαμβάνεται μία διεπαφή για Scatter-Gather μέσω της οποίας γίνονται μεταφορές από διάσπαρτες θέσεις μνήμης με την βοήθεια των descriptors. Ακόμη, υλοποιήθηκαν πολλαπλά κανάλια με προγραμματισμό προτεραιότητας για να μπορεί ο ελεγκτής να λαμβάνει εντολές για παραπάνω από μία μεταφορές. Τέλος, υλοποιήθηκε υποστήριξη για διακοπές για να ενημερώνεται ο επεξεργαστής για το πότε τελείωσαν οι μεταφορές από τον ελεγκτή.

Επιβλέπων Πτυχιακής: Γεώργιος Κορνάρος
Τίτλος: Επίκουρος Καθηγητής του τμήματος Μηχανικών Πληροφορικής, ΤΕΙ Κρήτης

Acknowledgments

This bachelor thesis is my first academic milestone and looking back I would like to thank all the people, that are not listed below, for their support and encouragement during my student years.

Firstly I would like to thank my thesis supervisor George Kornaros, for providing much-needed advice and patient tutelage, during my thesis.

My colleague Othon Tomoutzoglou for helping and supporting me in critical moments of my thesis.

My friend Fotis Koutoulakis for inspiring me to always do my best no matter the situation, ever since the day we met.

Most of all I would like to thank my family, who always supported my aspirations and endeavors throughout my life.

Contents

List of Figures

List of Tables

Abbreviations

AMBA ..... Advanced Microcontroller Bus Architecture

AXI ..... Advanced eXtensible Interface

CPU ..... Central Processing Unit

DMA ..... Direct Memory Access

FIFO ..... First In First Out

FPGA ..... Field Programmable Gate Array

FSM ..... Finite State Machine

ILA ..... Integrated Logic Analyzer

IP ..... Intellectual Property

INTR ..... Interrupt

IRET ..... Interrupt Return

ISR ..... Interrupt Service Routine

NoC ..... Network On Chip

PCIe ..... Peripheral Component Interconnect express

SDK ..... Software Development Kit

SoC ..... System On Chip

VHDL ..... VHSIC Hardware Description Language

VHSIC ..... Very High Speed Integrated Circuit

Chapter 1

Introduction

This chapter describes the knowledge background to accomplish this thesis. Also, motivation and related work for the completion of this thesis is discussed. Lastly, follows a brief overview of the thesis structure.

1.1 Background

DMA controller offloads the processor from tasks that involve memory transfers. CPU initiates transfers by programming the DMA controllers registers. DMA performs the memory transfers on the background, which liberates the CPU to operate on other assigned tasks. When transfers complete, DMA informs the CPU with an interrupt generation.

Efforts to design such IP's are known by ARM DMA-330[1], Xilinx CDMA v4.1[2], Xilinx AXI DMA v7.1[3], Lattice DMA[4] and Nilsson's LEON3 DMA[5].

Summary of features that these DMA include are:

1. Interrupt Generation
2. Scatter-Gather Interface
3. Multiple Channels

The features above are supported by the DMA of this thesis. It also supports the ARM AMBA protocol[6] and it's AXI4 interface[7], to get validated and debugged on a Zynq SoC[8]. The boards, which are used throughout this thesis, are ZyBo[9] and Zedboard[10].

None of the AXI4 compliant DMA controllers [1],[2],[3],[5] supports a dynamic burst length calculation. Burst length indicates the number of words (4 bytes each) transferred per AXI4 handshake for incremental bursts. The AXI4 lengths are 1,2,4,8,16,32,64,128,256 according to [7]. Current AXI DMA's are obliged to have one of the previous burst length's and cannot change throughout it's use. We proposed an algorithm that dynamically attributes the optimum burst length on the transaction depending on the bytes the DMA has to transfer.

1.2 Thesis Structure

Chapter 1 contains general information about DMA Controllers and the features which were implemented. Then follows discussion about the motivation and related work about this thesis.

Chapter 2 contains theory about DMA Controllers, how they evolved and are used nowadays.

Chapter 3 includes implementation details for the DMA Controller. Multiple channels, registers and finite-state machines.

Chapter 4 contains the results of the DMA Controller.

Chapter 5 contains the conclusions about the bachelor thesis and some thoughts about future optimizations and features.

Chapter 2

Theory

This chapter contains theory about DMA controllers. It also includes information about AMBA, the AHB bus that is connected to the DMA, and the AXI4 interface. Furthermore, Zynq SoC architecture will be covered and some brief coverage of the tools that are used to program this SoC's.

2.1 DMA Controller

DMA controllers were born when computer architects noticed that the CPU could either transfer data or execute tasks. They looked for ways to make these things to happen in parallel and that's how the DMA idea came. The concept back then was to let the CPU use the data that are in the cache and let the DMA perform the memory transfers. That's how both the memory cache and the system's single bus was exploited and parallelized.

The problem was when the CPU needed the bus the DMA was using to fetch data for the memory cache, resulting to a system halt until the memory transactions where finished. Techniques where invented to avoid this problems (i.e. DMA cycle stealing mode) temporarily until new technology was available.

Introduction of Network on Chips and evolution of PCIe turned the system from a single bus - that all peripherals arbitrarily used to communicate - to a packet-switched, point-to-point architecture network inside the system (similar to how packet-switched computer networks operate). Concurrent full duplex DMA transfers of multiple devices are possible through NoC's and PCIe without occupying the bus that another peripheral is using.

This resulted to a shift in the structure of the DMA controllers. They changed from a DMA that serves all peripherals - Third-Party DMA - to a DMA that is embedded inside peripherals in need of heavy data transfers - First-Party DMA. Both are explained in the below sections.

2.1.1 Third-Party DMA

Third-Party DMA is an IP that performs data transfers for IP's that don't have a DMA engine on their own. An example of such IP is the 8237 DMA controller[11] in

intel South Bridge and the DMA of this thesis. These DMA controllers have multiple channels that can be programmed to transfer data between peripheral devices and system memory.

2.1.2 First-Party DMA

First-Party DMA are IP's which reside inside peripheral devices that wish to perform transfers. They are also called bus-mastering DMA. These peripherals take the control of the system bus to perform the transfer and don't go through a third-party DMA to ask for transfers. Examples of first-party DMA are a wireless card, a PCI Express device and the Gigabit Controller.

2.2 AMBA

The AMBA specification [ARM, 1999] defines an on-chip communications standard for designing high-performance embedded microcontrollers.

Three distinct buses are defined within the AMBA specification:

- the Advanced High-performance Bus (AHB)
- the Advanced System Bus (ASB)
- the Advanced Peripheral Bus (APB)

2.2.1 AHB

AHB is recommended by ARM for all new designs not only because it provides a higher bandwidth solution, but also because the single-clock-edge protocol results in a smoother integration with design automation tools used during a typical ASIC development. As a result this bus was used for the implementation of the DMA controller.

A typical AMBA AHB system design contains the following components:

AHB master

A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.

AHB slave

A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

AHB arbiter

The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. An AHB would include only one arbiter, although this would be trivial in single bus master systems.

AHB decoder

The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

2.2.2 AXI4

For the master and slave part of the AHB components we have the AXI4 interface. AXI4 is part of ARM AMBA [Xilinx, 2011], a family of micro controller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second version of AXI, AXI4.

There are three types of AXI4 interfaces:

- AXI4-Full for high-performance memory-mapped requirements.
- AXI4-Lite for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream for high-speed streaming data.

The DMA controller uses AXI4-Lite interface for the slave part of the design and AXI4-Full interface for it's master part. The slave part has no need for high-performance and as such it uses the AXI4-Lite interface which is less complex and takes up less space on the FPGA board.

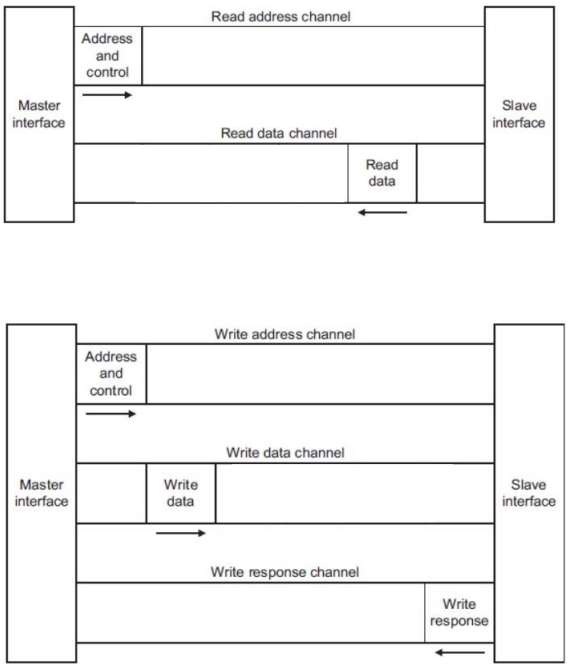

Figure 2.1: AXI4-Lite interface

On the other hand the master part is responsible for the data transfers and high-performance is beyond questioning. The price to be paid resides in the complexity of the design and space consumption on the FPGA board.

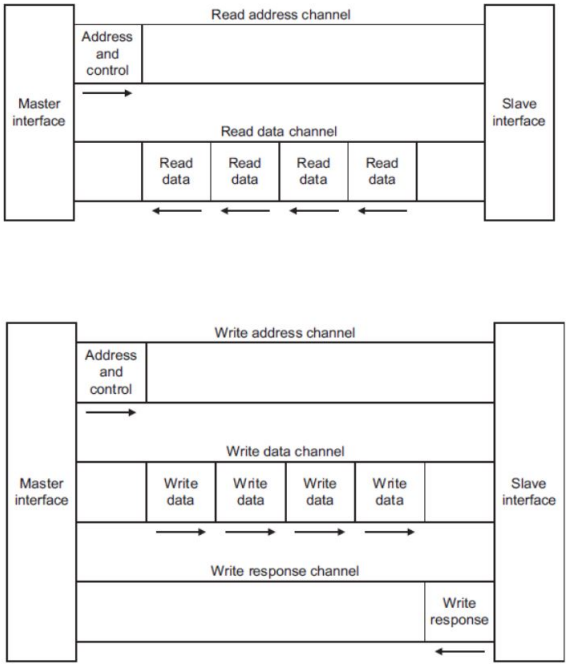

Figure 2.2: AXI4-Full interface

2.3 Zynq

Zynq SoC was used to implement the DMA controller. It consists of one ASIC part that contains a dual ARM Cortex-A9 MPCore CPUs with ARM v7, and a programmable logic part where the user can create his own IP cores. The two parts communicate with each other through the usage of AMBA interconnects.

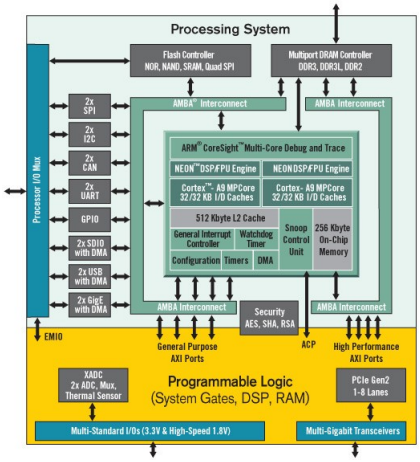

Figure 2.3: Zynq SoC insiders

Chapter 3

Implementation

This chapter contains details for the implementation of the DMA controller. The details that will be discussed here are registers, finite-state machines, descriptor internals, multiple channel operation. Lastly a small presentation of the tools and language used to implement the IP and the internals above will be given.

3.1 DMA Channels

The DMA Controller consists of 7 channels each one with it's own register address space. The processor can program and initiate transfers from whichever channel it wants. Each register has size of 4 bytes.

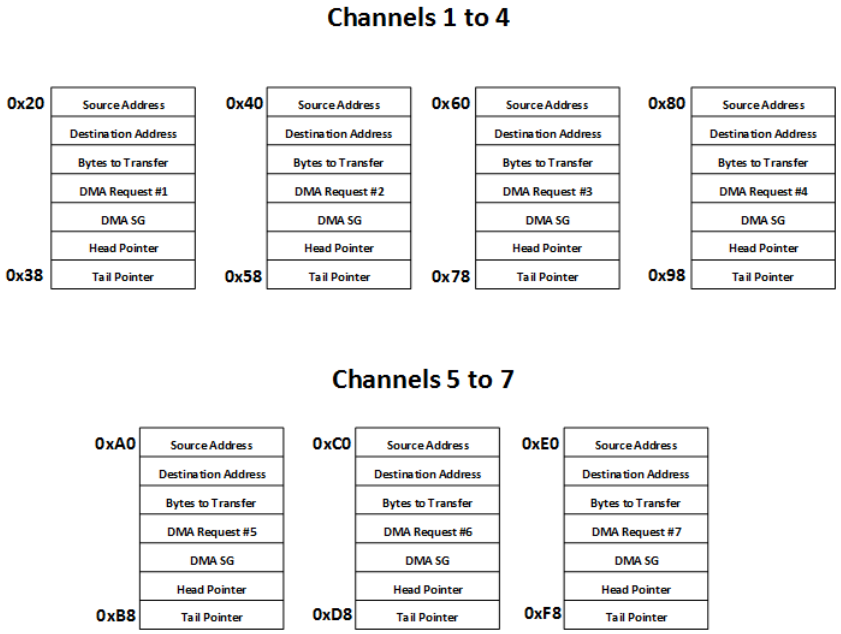

Figure 3.1: DMA Channels

3.1.1 DMA Channel Arbitration Algorithm

The algorithm for the multiple DMA channels is displayed on figure 3.2. It checks if any channel has requested a DMA transfer always starting the checks from channel 1 going downto channel 7. The less the number of the channel the greater priority it has to be serviced (channel 1 highest / channel 7 lowest).

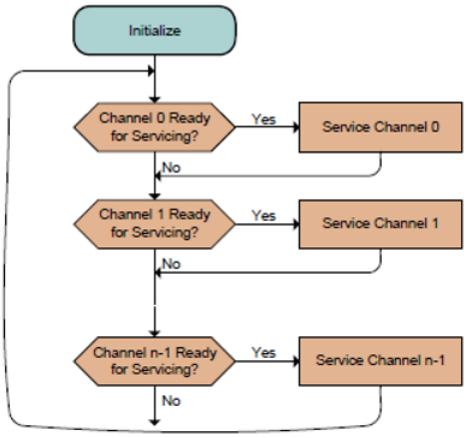

Figure 3.2: DMA Channel Arbitration Algorithm

3.2 DMA Registers

In this section the operation of DMA registers will be discussed. Control register affects the operation of the DMA, and status register is affected by the current DMA state. Lastly the register organization of Channel #1 will be discussed. The remaining channels operate the same as channel #1 with the only difference being the address offset.

3.2.1 Control Register (Offset 0x00)

This register controls operations of the DMA controller.

Figure 3.3: Control Register

Table 3.1: Control Register Details

Bits	Field Name	Default Value	Description
[31:9]	Reserved	0	These bits are reserved for future use.
[8]	Acknowledgement Bit	0	This bit when set to 1 acknowledges (by the routine that is written in software) that the processor received the interrupt and that the DMA can continue servicing the next channels or descriptors.
[7:1]	Active Channels	bin(1111111)	These bits control which channels are active. For example if the bits are 0000001 then only Channel #1 is active (By default all channels are active).
[0]	Reset Bit	0	When this bit is set to '1' it sets all channel registers values to '0'.

3.2.2 Status Register (Offset 0x04)

This register displays the status of the DMA controller.

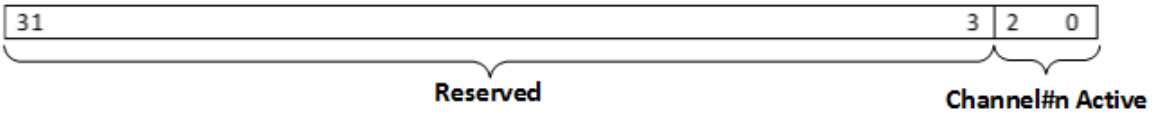

Figure 3.4: Status Register

Table 3.2: Status Register Details

Bits	Field Name	Default Value	Description
[31:3]	Reserved	0	These bits are reserved for future use.
[2:0]	Channel#n Active	0	These bits display which channel is getting serviced by the DMA. If for example the value is 010 then channel 2 is active. 000 means no channel is active (DMA is Idle).

3.2.3 Source Register (Offset 0x20)

This register holds the source address for where the DMA should fetch data from.

Figure 3.5: Source Register

Table 3.3: Source Register Details

Bits	Field Name	Default Value	Description
[31:0]	Source address register Ch#N	0	Source address for the DMA controller

3.2.4 Destination Register (Offset 0x24)

This register holds the destination address for where the DMA should sent the data to.

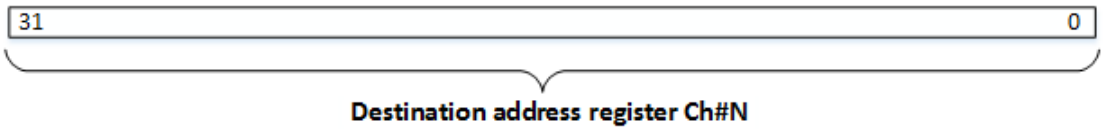

Figure 3.6: Destination Register

Table 3.4: Destination Register Details

Bits	Field Name	Default Value	Description
[31:0]	Destination address register Ch#N	0	Destination address for the DMA controller

3.2.5 Bytes to Transfer Register (Offset 0x28)

This register holds the number of bytes to be transferred.

Figure 3.7: Bytes to Transfer Register

Table 3.5: Bytes to Transfer Register Details

Bits	Field Name	Default Value	Description
[31:0]	Bytes to transfer register Ch#N	0	Number of bytes to be transferred from source to destination address

3.2.6 DMA Request Register (Offset 0x2C)

This register is responsible for DMA transfer requests from the particular channel.

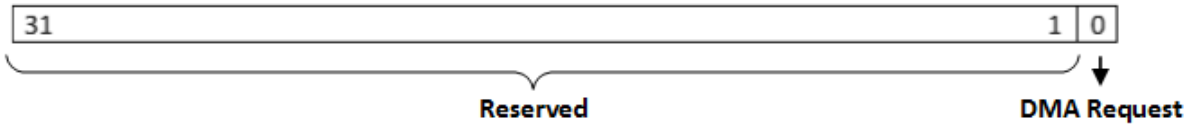

Figure 3.8: DMA Request Register

Table 3.6: DMA Request Register Details

Bits	Field Name	Default Value	Description
[31:1]	Reserved Ch#N	0	These bits are reserved for future use.
[0]	DMA Request	0	If this bit is set to '1' a DMA transaction is requested from the specific channel

3.2.7 DMA Scatter Gather Mode Register (Offset 0x30)

This register holds the number of bytes that the user wants to transfer.

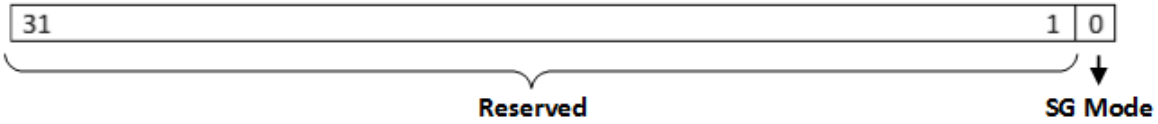

Figure 3.9: DMA Scatter Gather Mode Register

Table 3.7: DMA Scatter Gather Mode Register Details

Bits	Field Name	Default Value	Description
[31:1]	Reserved	0	These bits are reserved for future use.
[0]	SG Mode	0	If this bit is set to '1' and DMA requests for a transfer the transfer will be a Scatter Gather. If the bit is '0' then on DMA request the transfer will be a simple DMA one.

3.2.8 DMA Scatter Gather Head Pointer Register (Offset 0x34)

This register holds the number of bytes that the user wants to transfer.

Figure 3.10: DMA Scatter Gather Head Pointer Register

Table 3.8: DMA Scatter Gather Head Pointer Register Details

Bits	Field Name	Default Value	Description
[31:0]	DMA Scatter Gather Head Pointer Ch#N	0	Address of the first node on the linked list of descriptors in memory.

3.2.9 DMA Scatter Gather Tail Pointer Register (Offset 0x38)

This register holds the number of bytes that the user wants to transfer.

Figure 3.11: DMA Scatter Gather Tail Pointer Register

Table 3.9: DMA Scatter Gather Tail Pointer Register Details

Bits	Field Name	Default Value	Description
[31:0]	DMA Scatter Gather Tail Pointer Ch#N	0	Address of the last node on the linked list of descriptors in memory.

3.3 Finite State Machines

The DMA has 2 FSM's. The "Master Interface FSM" is responsible to make the transfers depending on the bytes it has to transfer. The "Master Scatter Gather Interface FSM" works in conjunction with the master FSM, by bringing the descriptors and feeding them to the master FSM to make the transactions described by the descriptor. Both FSM's are explained in more detail below.

3.3.1 DMA Master Interface FSM

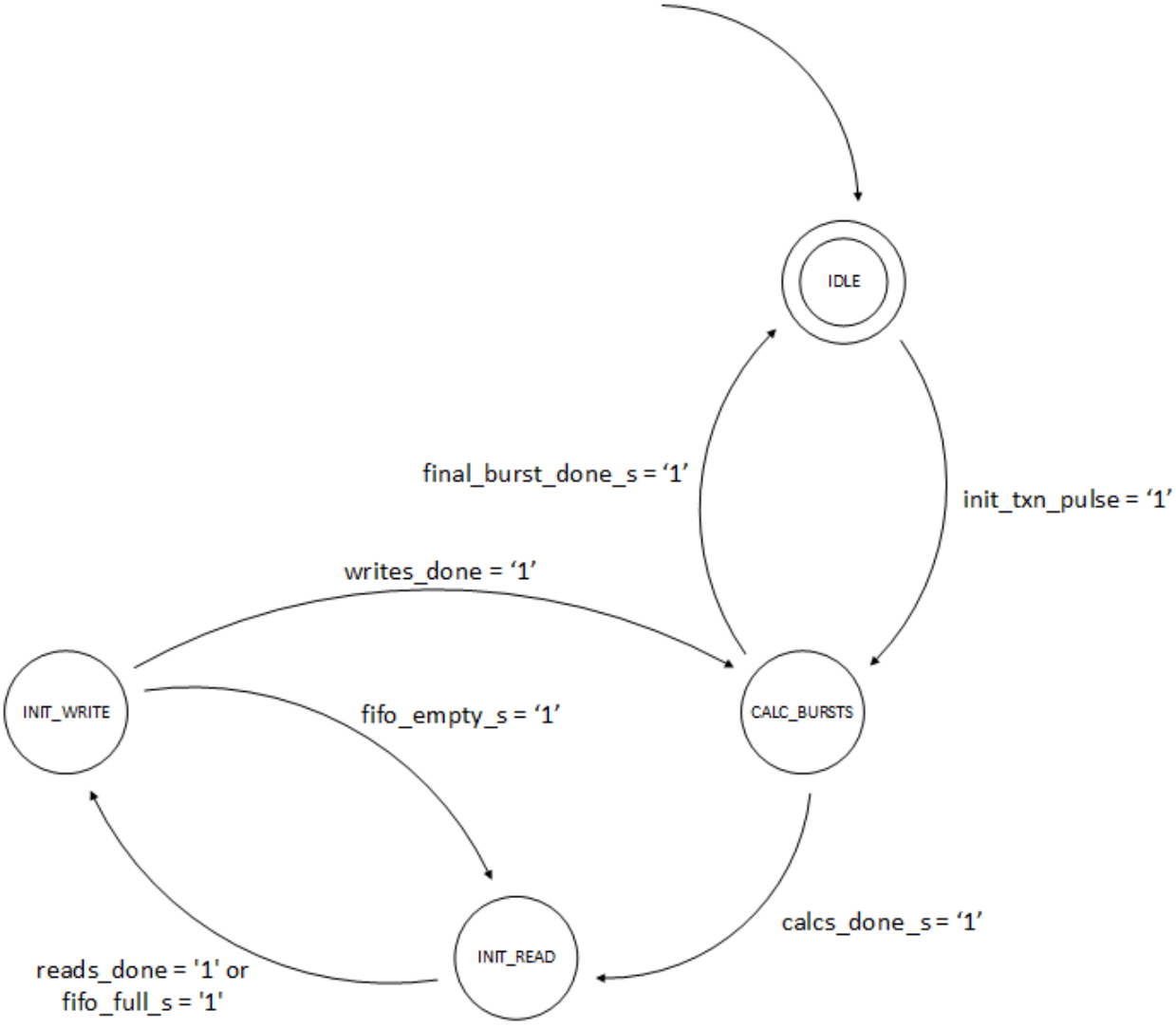

Figure 3.12: DMA Master Interface FSM

IDLE

This is the starting state of the FSM following a reset from the system or the finish of a transaction. If a channel wants to initiate a DMA transaction then the state moves to CALC_BURSTS.

CALC_BURSTS

This state is responsible to calculate the optimum burst for the bytes that have to be transferred. The burst lengths available from AXI4 are 1,2,4,8,16,32,64,128,256.

Example: The user wants to transfer 5908 bytes. These bytes get translated to words (4 bytes = 1 word) 5908 / 4 = 1477 words. The number 1477 translates in 10111000101 in binary and the rest from the 32 bits are 0. Then we choose burst length according to figure 3.13.

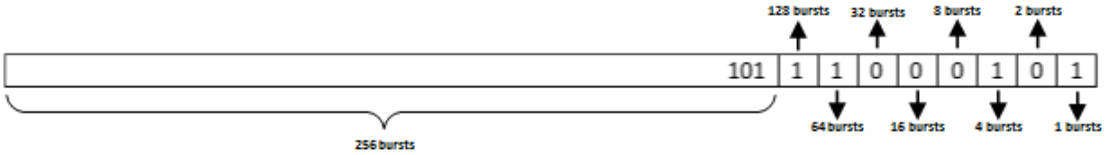

Figure 3.13: Optimum Burst Example

The table shows that we have to do 5 (101 binary = 5 decimal) transfers of burst length 256; 1 transfer of 128, 64, 4, 1 burst length. Adding all these bursts together 256 * 5 + 128 + 64 + 4 + 1 = 1477 words are transferred.

The algorithm checks if any of the first 8 bits are 1 and makes a transfer according to the length shown by the figure above. The top 24 bits display the number of 256 burst length transfers that have to be done.

INIT_READ

This state starts reading data from the source register given by the user or descriptor. The number of data to be read are known from the CALC_BURSTS state before. When all data are read and stored in the FIFO then the FSM state changes to INIT_WRITE.

INIT_WRITE

This state starts writing the data existing in the FIFO to the destination indexed by the destination register. When FIFO empties the state changes to either CALC_BURSTS or INIT_READ. CALC_BURSTS when transfers of a specific burst length (1,2,4,8,16,32,64,128,256) are finished. INIT_READ if more specific burst length transfers need to be completed.

3.3.2 DMA Master Scatter Gather Interface FSM

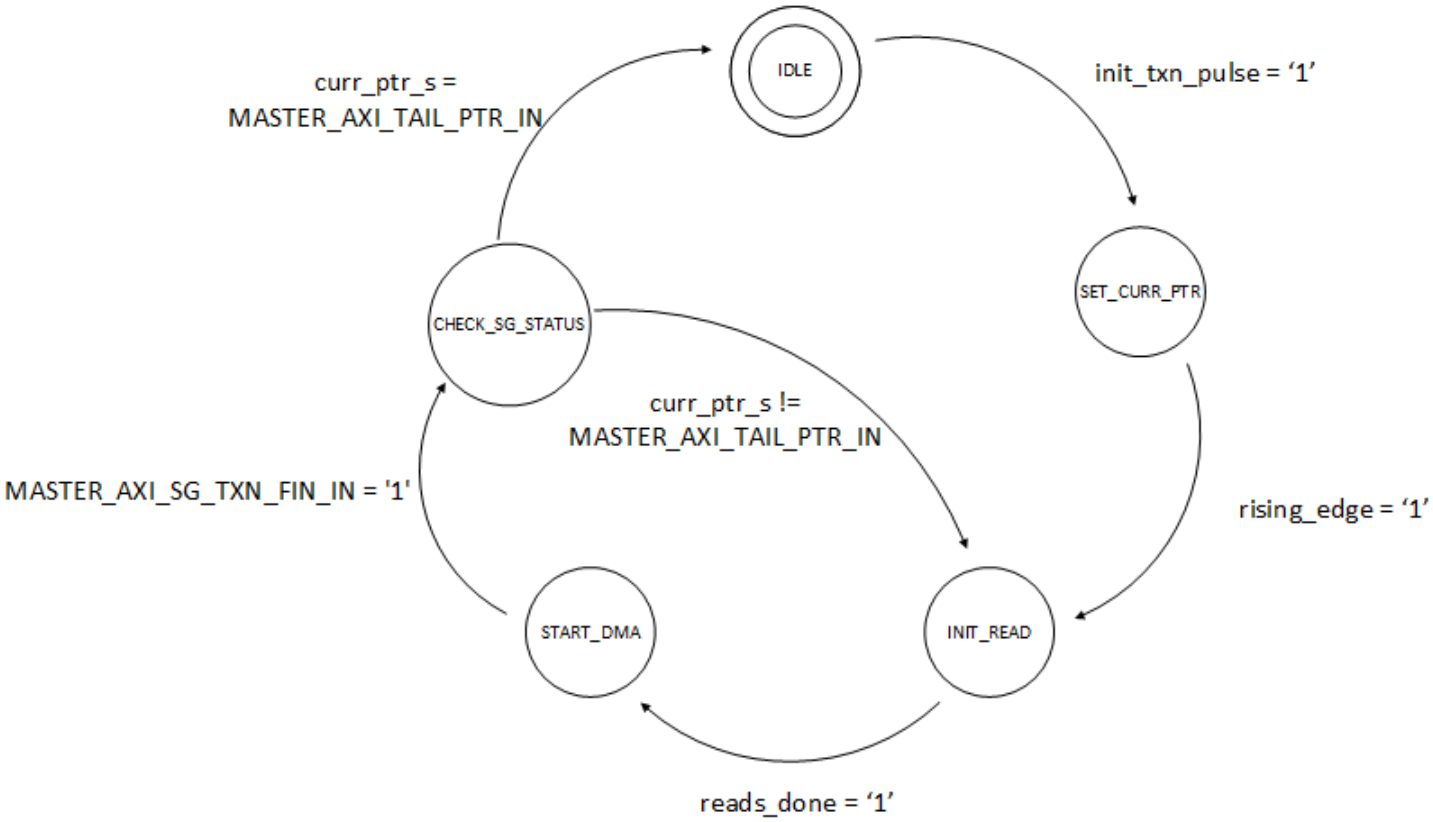

Figure 3.14: DMA Master Scatter Gather Interface FSM

IDLE

This is the starting state of the FSM following a reset from the system or the finish of descriptor fetching. The initiation of a Scatter-Gather transaction moves this state to SET_CURR_PTR.

SET_CURR_PTR

This state reads the head pointer register and puts it's value in a signal which declares the start of a linked list. When this process finishes the state changes to INIT_READ.

INIT_READ

This state starts reading the descriptors from the address pointed by the head pointer register. When the descriptor is read and decoded then the state changes to START_DMA.

START_DMA

This state initiates the "DMA Master Interface FSM" by feeding the source, destination and bytes to transfer that were read by the descriptor. When the "DMA Master Interface FSM" finishes the transaction it informs "DMA Master Scatter Gather Interface FSM" and the state changes to CHECK_SG_STATUS.

CHECK_SG_STATUS

This state keeps track of the progress in the linked list of descriptors. If the current head pointer matches the tail of the list then the FSM changes to IDLE state. If the current head pointer fails to match the tail pointer then the current head moves to the next node in the linked list and the FSM starts reading the descriptor found in that address by changing the FSM state to INIT_READ.

3.4 VHDL

VHDL (VHSIC Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. This programming language differs from a conventional one. A hardware description language is inherently parallel, i.e. commands which correspond to logic gates are executed (computed) in parallel, on new input arrival. A HDL program mimics the behavior of a physical, usually digital system. It allows incorporation of timing specifications (gate delays) as well as, a description of a system as an interconnection of different components. Every HDL language resides in the Register Transfer Level (RTL) of the abstraction hierarchy.

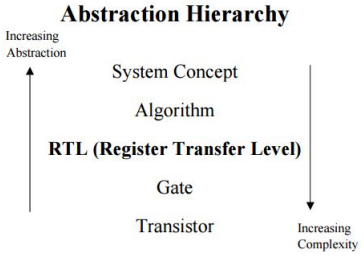

Figure 3.15: Abstraction Hierarchy

3.5 Xilinx Vivado

Vivado allows the developers to create, debug, simulate and integrate IP's on FPGA. A selection of proprietary IP's is readily available from Xilinx, but it also allows the creation of custom IP's that can integrate with the previous ones. To glue the components together Vivado uses the AXI4 interface. This interface is generated automatically through Vivado IP packager and allows the developer to encapsulate the IP inside the interface. The fusion of the IP and AXI4 interface allows intercommunication with the whole programmable system - given that the rest of the IP's are AXI4 compatible - and also with the ARM processor.

The DMA was first created using only a AXI4-Lite Slave interface (the less complicated one). AXI4-Lite contains the registers which control the operations of the DMA controller. After that, the DMA was placed an a Zynq SoC and it's register where stimulated, checked and debugged with the use of ILA. When everything run bug-free both master interfaces where added. Their actions are controlled by the

AXI-4 Lite slave registers. Lastly, each master interface has it's own interrupt generation on transaction completion. The routine that the ARM CPU will follow after the interrupt is written in Xilinx SDK tool.

3.6 Interrupts

When the DMA generates an interrupt it asserts the INTR pin of the processor. The assertion of the bit is done indirectly through an intermediate interrupt router which asserts the INTR CPU pin. This router handles a lot of interrupt requests and each request has different priority depending on the design and user choices.

When CPU sees the INTR pin asserted, it finishes the instruction it was executing, and stops processing any further instructions. Then based on the interrupt number, it goes out and reads some values from a table, called interrupt vector table. The interrupt vector table contains the address where the ISR is stored in memory. The program execution jumps to the ISR starting address. Each interrupt has its own separate ISR. After ISR takes care of the interrupting device, it executes IRET instruction. The CPU comes back to its normal execution flow and resumes operation from where it had stopped.

Two interrupt routines are implemented in software using Xilinx SDK: The first routine activates the acknowledgement bit of the control register to let the channel know that the request for DMA is acknowledged and completed, and as a result the DMA requesting device turns it's requesting bit to '0'. The second routine is for the scatter gather channel. It checks if the channel is requesting a scatter gather and prints a message to the user each time a descriptor is processed and finished. When the scatter gather finishes with all the descriptors it informs the user with a message and it activates the acknowledgement bit like the first routine.

3.7 Abstract Schematic

On the figure below there is an abstract scenario of how the DMA worked with the rest of the peripherals. This scenario is presented in order to get an idea of how the transactions are made.

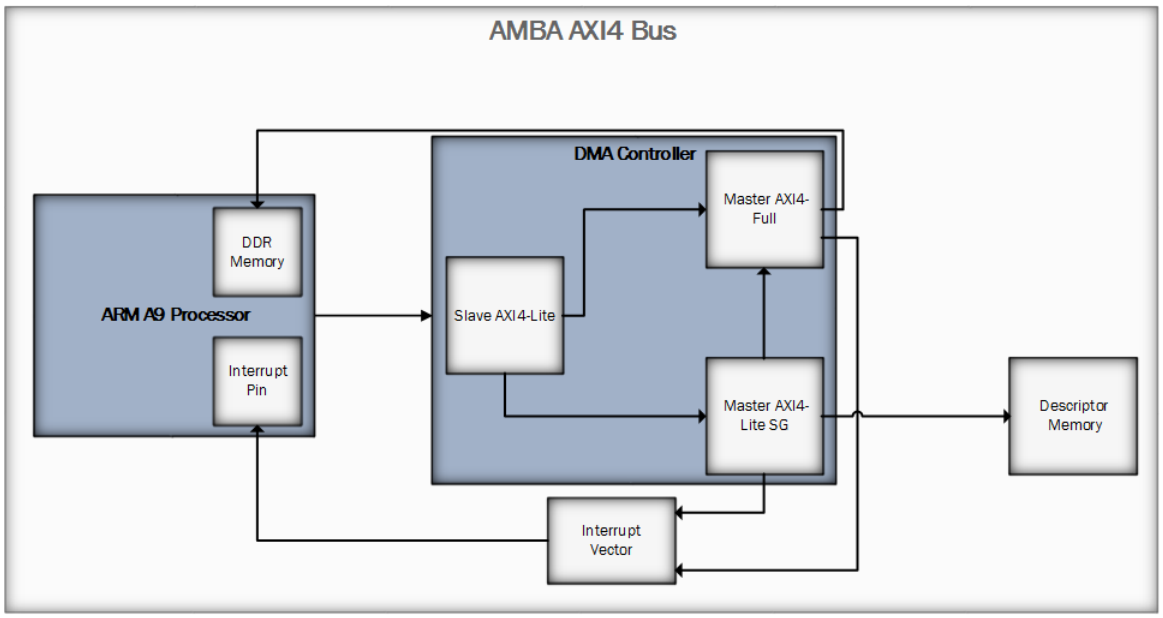

Figure 3.16: Abstract implementation of the DMA Controller

3.7.1 Example of a DMA operation

The steps followed in order to initiate a transaction are the following:

1. Processor programs DMA registers and initiates transfer.
2. DMA checks if it is in Simple DMA or Scatter Gather mode.
3. DMA performs the transactions depending on the mode.
4. When the transactions finish there is an interrupt generation to inform the processor.

The actual implementation in Vivado exists in Appendix A.

Chapter 4

Results

This chapter verifies that the DMA operates as expected. Code and simulations are covered to prove how the DMA works. Results derived are compared to other DMA controllers. Lastly, area of DMA is covered depending on the features it supports.

4.1 Simulations

We simulated both simple and scatter-gather DMA transactions to verify correct operation. This was done using Xilinx Vivado ILA[13] core, and Xilinx SDK to write a C program that programs the DMA controller to perform a transaction.

4.1.1 Simple DMA Transaction Simulation

A C-program was written to program the DMA, figure 4.1, and the results are shown in figure 4.2.

```
// Fill memory location 0x4000_0000 with values
for (i = 0; i < 200; i++) {
    *(mem_ptr + i) = i;
}


/* Channel #1, offset 0x20 */
*(ptr + 8) = 0x40000000; // Source address
*(ptr + 9) = 0x40001000; // Destination address
*(ptr + 10) = 64;      // Bytes to transfer
```

Figure 4.1: Simple DMA Sample Code

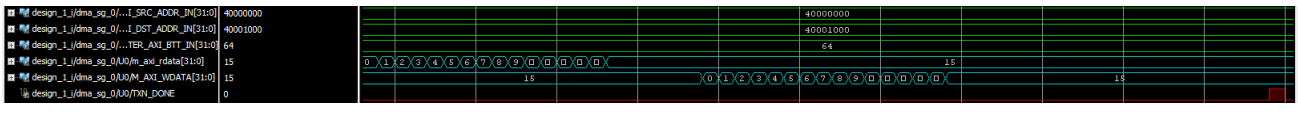

Figure 4.2: Simple DMA simulation

From figure 4.1 the C-code sets 0x40000000 address first 200 places to 0-199 so we know that the DMA is fetching the correct data in correct order. After we set the source address of the DMA to 0x40000000, and destination 0x40001000, and the bytes to transfer to 64. 64 bytes correspond to 16 words that's why we see in the aqua lines of figure 4.2, 16 reads (values 0-15) and 16 writes (values 0-15). Lastly,

the the red line from figure 4.2 declares an interrupt generation to the processor after all reads and writes are done.

4.1.2 Scatter-Gather DMA Transaction Simulation

Our DMA scatter-gather utility was checked for correct operation. The code snippet below fills FPGA's BRAM memory with incremental values from 0 to 199 in the corresponding addresses. Then we initialize 3 descriptors and use address 0x40000000 as source which contains the incremental values explained above. Then these values are copied to scattered destinations in memory (0x40001000, 0x40000200, 0x40000300). The bytes that each descriptors has to transfer from the source to the destination are 1024 (256 words) for the first descriptors; 512 (128 words) for the second descriptor; and 756 (189 words) for the third descriptor.

After a descriptor is processed the next one must be fetched until the linked list reaches it's end. Descriptors maintain a pointer in their structure which points to the next descriptor to be fetched. The first descriptor (address: 0xC0000000) points to the second descriptor (address: 0xC0000020). The second points to the third (address: 0xC0000040). The third points to the fourth (address: 0xC0000060).

The descriptor pointer can pointed anywhere in the memory, as long as, in the location pointed there is a valid descriptor. If not, the results are unpredictable. In our example code we use address 0xC0000000 as head of the list and 0xC0000040 as the tail. As a result, we start processing from the first descriptor then process the second, and stop when we process the third (because it matches the tail address).

```
// Fill memory location 0x4000_0000 with values
for (i = 0; i < 200; i++) {
    *(mem_ptr + i) = i;
}

/* Descriptor Initialization */

// Descriptors #1 0xC0000000 //
*(sg_mem_ptr + 0) = 0x40000000;
*(sg_mem_ptr + 1) = 0x40001000;
*(sg_mem_ptr + 2) = 1024;
*(sg_mem_ptr + 3) = 0xC0000020;

// Descriptors #2 0xC0000020 //
*(sg_mem_ptr + 8)  = 0x40000000;
*(sg_mem_ptr + 9)  = 0x40000200;
*(sg_mem_ptr + 10) = 512;
*(sg_mem_ptr + 11) = 0xC0000040;

// Descriptors #3 0xC0000040 //
*(sg_mem_ptr + 16)  = 0x40000000;
*(sg_mem_ptr + 17)  = 0x40000300;
*(sg_mem_ptr + 18)  = 756;
*(sg_mem_ptr + 19)  = 0xC0000060;

/* Channel #4, offset 0x80 */
*(ptr + 36) = 1;          // Scatter gather mode on
*(ptr + 37) = 0xC0000000; // Scatter gather list head pointer
*(ptr + 38) = 0xC0000040; // Scatter gather list tail pointer

// Start channel 4 scatter gather transaction
*(ptr + 35) = 1;
```

Figure 4.3: DMA Scatter Gather Sample Code

We simulated the results running the C code from figure 4.3. Figure 4.4 shows that the DMA fetched data from the source (address: 0x40000000) and stored them

to the FIFO before forwarding them to the destinations (address: 0x40001000, 0x40000200, 0x40000300). We also see the words that needed to be transferred 256, 128, 189 which correspond to the bytes of the C code 1024, 512, 756. For the 189 word transaction the DMA used the optimum burst algorithm described in Figure 3.13. It split 189 into burst length transfers of 128, 32, 16, 8, 4, 1 to comply with the AXI4 burst length protocol. Lastly, the red line is the interrupt that is generated after all 3 descriptors are processed to inform the CPU for the end of the Scatter Gather transaction.

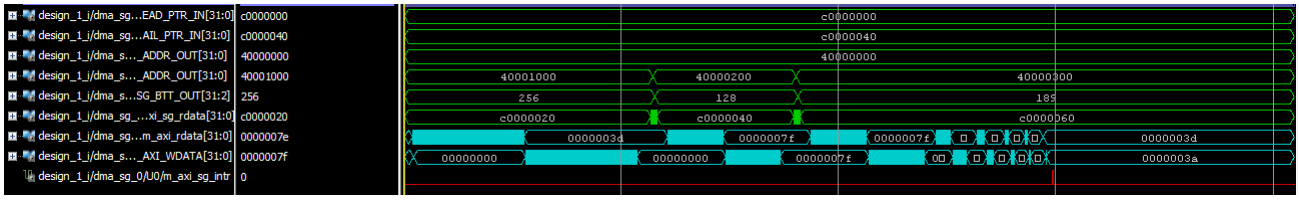

Figure 4.4: DMA Scatter Gather Simulation

4.2 Comparisons

DMA was compared against Zynq Cortex-A9 CPU in performing a memory transfer. Then it was compared against modern DMA solutions provided by Xilinx.

4.2.1 CPU vs Custom DMA

Figure 4.5 displays a C routine that emulates the way our DMA operates. It firsts reads the data from the source to a buffer and then moves the data from the buffer to the destination address.

```
for (i = 0; i < 256; ++i) {
        buffer[i] = *(mem_ptr + i);
}

for (i = 0; i < 256; ++i) {
         *(mem_ptr + i + 500) = buffer[i];
}
```

Figure 4.5: C Code for CPU Data Transfer

Figure 4.6 shows the assembly code for the first for-loop. The processor takes 19 cycles to copy one word (4 bytes) from source to the buffer.

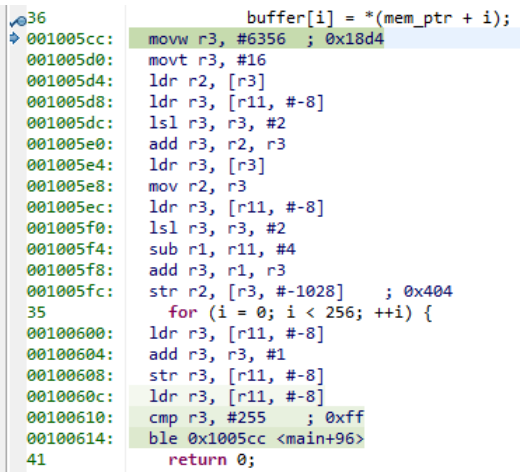

Figure 4.6: CPU Data Fetch Code Disassembly

Figure 4.7 shows the assembly code for the second for-loop. The processor takes 19 more cycle to copy one word (4 bytes) from the buffer to the destination address.

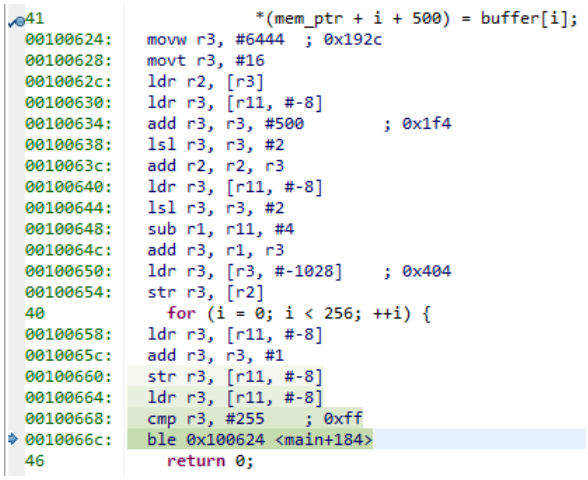

Figure 4.7: CPU Data Send Code Disassembly

As a result, every 38 cycles a word is copied from source to destination.

For the DMA controller a small overhead occurs. The DMA must be programmed first by the CPU before it initiates transfers. The cost of the CPU and the AXI4 protocol handshakes to program the DMA is about 88 clock cycles. After the DMA is programmed for every 2 clock cycles (one read one write) new data are fetched and sent to proper destination, through AXI burst transactions. Figure 4.8 shows a comparison between the CPU and the DMA controller.

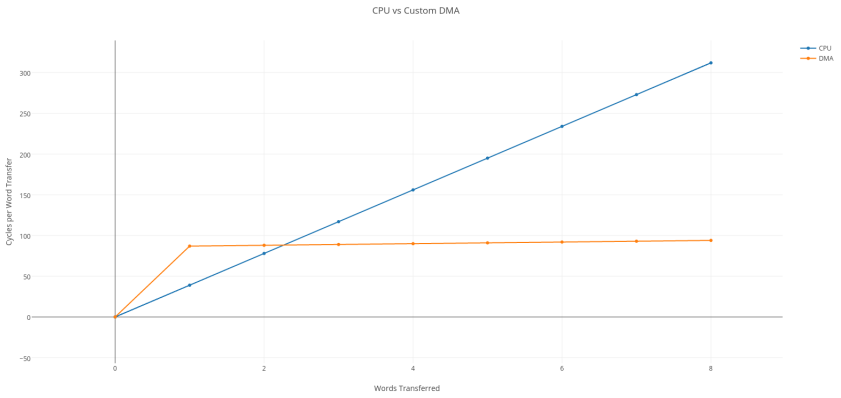

Figure 4.8: CPU vs Custom DMA Graph

The graph above shows that the CPU is a lot slower than the DMA. For example a 256 word transfer would take the CPU 256*39 cycles to complete while the DMA only needs 256*2+overhead(about 88 cycles). It also suggests that the DMA is more efficient than the CPU after 3 word transfers. That's half true because the CPU might be slower after 3 word transfers but the DMA also interrupts the processor after each transactions is completed, and this causes many context switches(which are costly). That adds more overhead to it's performance. Our suggestion is to use a DMA controller when our transactions constitute mainly from >8 words. Else, the CPU performs better without the DMA controller.

4.2.2 Xilinx CDMA vs Custom DMA

After the CPU was compared with the DMA of the thesis, the next comparison to be made was against Xilinx CDMA. Figure 4.9 shows this comparison.

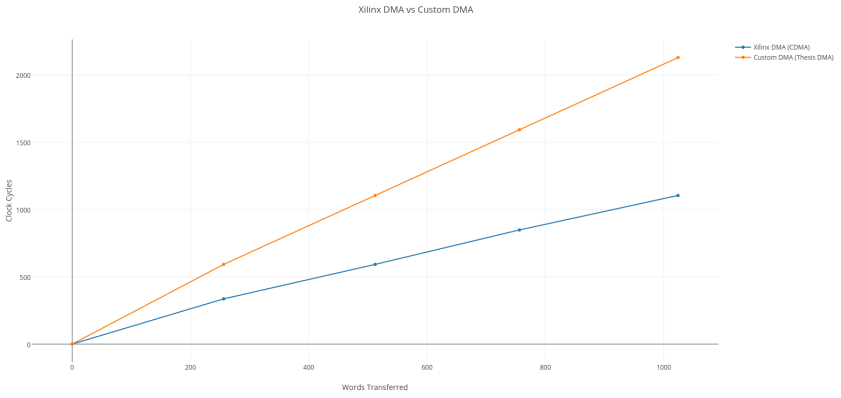

Figure 4.9: Xilinx DMA vs Custom DMA graph

We can see that the DMA of this thesis takes double the time to perform transactions with the same amount of bytes. The reason is the technique used by the CDMA of Xilinx which is to pipeline the data after they are read to the write bus. Figure 4.10 shows the pipelining used by Xilinx and figure 4.11 shows this thesis DMA technique which is to first read all data to FIFO and then write them to destination.

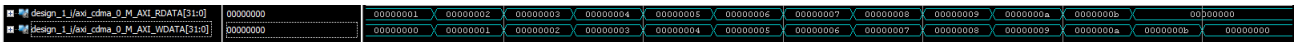

Figure 4.10: Xilinx DMA Transfer Method Simulation

Figure 4.11: Custom DMA Transfer Method Simulation

While performance is crucial our Custom DMA is able to respond in situations where Xilinx CDMA can't. An example is a BRAM controller[12] with one bus lane like figure 4.12.

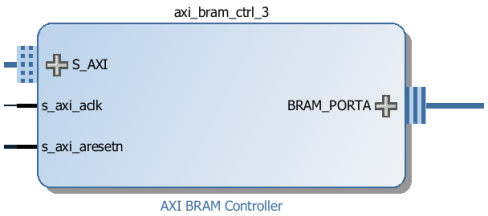

Figure 4.12: BRAM Controller

This BRAM controller has only one bus lane and it can either read or write. It can't perform both concurrently. When Xilinx CDMA was used to read and write using this BRAM controller it halted operation and the data where faulty. On the other hand our DMA made the transfer without any errors. A solution we propose is a bit which, when set, transforms the DMA transfer operations via pipeline and when it's not it removes the pipeline technique.

4.3 DMA Area Occupation

Four DMA controllers where created for this thesis. First a simple DMA with one channel; Second a DMA with Scatter-Gather utility; Third a DMA with multiple channel support; Fourth a DMA with all the above features supported. Each one can be used depending on the user needs. All the resulting tables in the next subsections are without Xilinx tool performing it's optimizations. A 10% less area occupation is expected if all the DMA's pass through the optimization phase.

4.3.1 Simple DMA Area

The simple DMA controller occupies the least area and it's simple in implementation and usage. It can transfer data from one memory region to another through it's channel.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	735	0	17600	4.18
LUT as Logic	479	0	17600	2.72
LUT as Memory	256	0	6000	4.27
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	524	0	35200	1.49
Register as Flip Flop	524	0	35200	1.49
Register as Latch	0	0	35200	0.00
F7 Muxes	128	0	8800	1.45
F8 Muxes	0	0	4400	0.00

Figure 4.13: Simple DMA area table

4.3.2 Scatter-Gather DMA Area

The Scatter-Gather DMA occupies more space than the simple because of the extra AXI4-Lite interface that fetches the descriptors. It supports both simple and Scatter-Gather transactions.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	935	0	17600	5.31
LUT as Logic	679	0	17600	3.86
LUT as Memory	256	0	6000	4.27
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	988	0	35200	2.81
Register as Flip Flop	988	0	35200	2.81
Register as Latch	0	0	35200	0.00
F7 Muxes	160	0	8800	1.82
F8 Muxes	0	0	4400	0.00

Figure 4.14: DMA with Scatter Gather area table

4.3.3 Multiple Channel DMA Area

DMA with multiple channels occupies almost twice the space of the simple DMA. Main reason is the registers that each channel contains. This enables peripherals to connect to different channels dedicated to them and ask for transfers. These channels don't support scatter gather utilities.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1549	0	17600	8.80
LUT as Logic	1293	0	17600	7.35
LUT as Memory	256	0	6000	4.27
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	1434	0	35200	4.07
Register as Flip Flop	1434	0	35200	4.07
Register as Latch	0	0	35200	0.00
F7 Muxes	256	0	8800	2.91
F8 Muxes	32	0	4400	0.73

Figure 4.15: DMA with multiple channels area table

4.3.4 Complete DMA Area

Complete DMA occupies the most space as it has the features of multiple and scatter gather DMA fused together. It also has interrupt support to inform the processor for the transactions. Each channel can do either simple DMA transaction or scatter gather and the channels - 1 to 7 - have different priority in getting serviced by the DMA depending on the number of the channel; 1st has the most and 7th the least.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2061	0	17600	11.71
LUT as Logic	1805	0	17600	10.26
LUT as Memory	256	0	6000	4.27
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	2410	0	35200	6.85
Register as Flip Flop	2410	0	35200	6.85
Register as Latch	0	0	35200	0.00
F7 Muxes	320	0	8800	3.64
F8 Muxes	96	0	4400	2.18

Figure 4.16: Complete DMA area table

Chapter 5

Conclusions and Future Work

This chapter contains the conclusion of this thesis and also future work and expansions for the DMA controller.

5.1 Conclusions

A DMA controller is an integral part of modern computing systems. It offloads the processor dramatically from tasks that involve data transfers throughout the system and it also does them more efficiently. The DMA of this thesis is able to communicate with it's AXI4 counterparts in the system and perform transfers by it's programmable interface. It outperformed the CPU on transfers that were greater than 8 words, but it's performance was poor when compared to Xilinx's CDMA. Although, this degradation in performance makes it usable in every situation presented, while Xilinx's CDMA requires at least two bus lanes from the sender/receiver to perform error free.

5.2 Optimizations

The DMA Controller has a lot of room for optimizations by tweaking the design choices that were made. Firstly, the FIFO buffer can become significantly smaller (current size 256 words), by pipelining the read data to write data immediately. The current DMA first stores the data to the FIFO and then proceeds on writing them to the destination.

The Scatter-Gather interface is AXI4-Lite, so the descriptors are fetched a lot slower than an AXI4-Full interface. This change will make the process of fetching faster but at the cost of more complexity in the code and space allocated in the FPGA. Also, a FIFO can be implemented on the Scatter-Gather interface in order to fetch all the descriptors and then start the transactions. The current scenario is fetch one descriptor and when it finishes fetch the next.

5.3 Additional Functionality

The DMA supports only word-aligned transfers. An alignment algorithm [diva paper/chapter] can be implemented to support half-word and byte-aligned transfers.

DMA doesn't support error detection. If the user sends data to an address that is not AXI4 compliant then the DMA doesn't inform the user that the address is invalid. A possible method to solve this problem is to create a timer and wait for a response for a set period of time (i.e. 30 cycles). The time can be set on the control register, for example using 6 bits to indicate the max cycle time before signaling an interrupt and declaring a bit on the status register that the transaction to the specific address failed.

Appendix A

Vivado Simple DMA Schematic

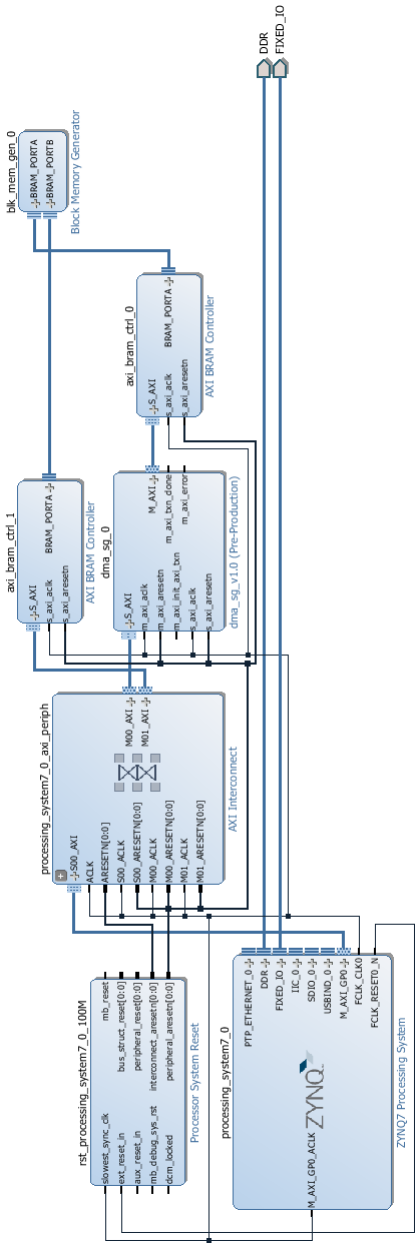

Figure 5.1: Simple DMA Schematic

Vivado DMA Scatter Gather Schematic

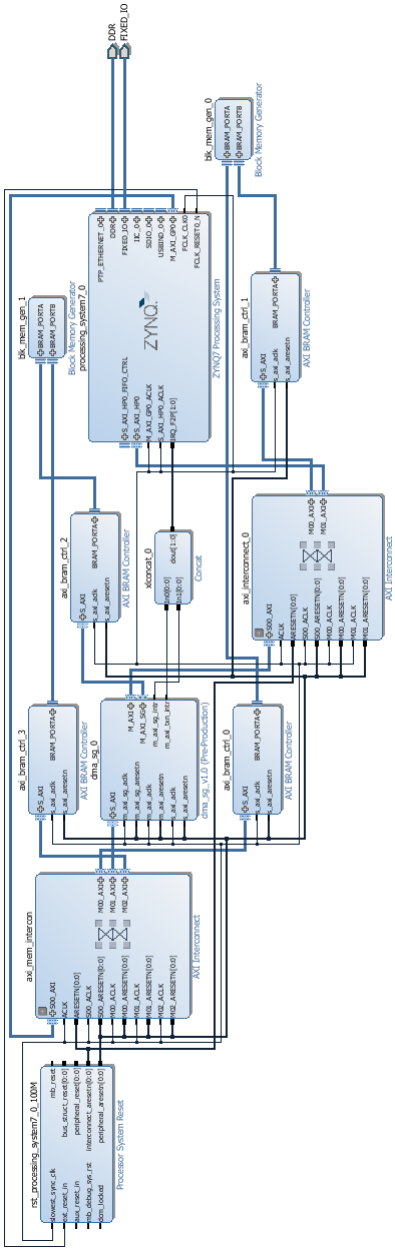

Figure 5.2: DMA Scatter Gather Schematic

Bibliography

[1] ARM, *Corelink DMA Controller DMA-330*, Technical Reference Manual, r1p1 release, 2010. URL
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424c/DDI0424C_dma330_r1p1_trm.pdf

[2] Xilinx, *AXI Central Direct Memory Access v4.1*, LogiCORE IP Product Guide, Vivado Design Suite, 2015. URL
http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf

[3] Xilinx, *AXI DMA v7.1*, LogiCORE IP Product Guide, Vivado Design Suite, 2015. URL
http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[4] Lattice, *Scatter-Gather Direct Memory Access (DMA) Controller*, IP Express, 2011. URL
http://media.digikey.com/pdf/Data%20Sheets/Lattice%20PDFs/DMA-SG_Overview.pdf

[5] Emelie Nilsson, *DMA Controller for LEON3 SoC: Using AMBA*, Linköpings University, Master Thesis, 2013. URL
http://www.diva-portal.org/smash/get/diva2:626214/FULLTEXT01.pdf

[6] ARM, *AMBA Specification*, Rev 2.0, 1999. URL
http://www-micro.deis.unibo.it/~magagni/amba99.pdf

[7] Xilinx, *AXI Reference Guide*, UG761 v13.1, 2011. URL
http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[8] Xilinx, *Zynq-7000 All Programmable SoC*, Technical Reference Manual, UG585 v1.10, 2015. URL
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[9] Digilent, *ZYBO Reference Manual*, ZYBO rev. B, 2014. URL
http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf

[10] AVNET, *Zedboard Getting Start Guide*, version 5.0, 2012. URL
http://zedboard.org/sites/default/files/GS-AES-Z7EV-7Z020-G-14.1-V5.pdf

[11] Intel Corporation, *8237/8237-2 High Performance Programmable DMA Controller*, 1993. URL
https://edge.edx.org/c4x/BITSPilani/EEE231/asset/Intel-8237-dma.pdf

[12] Xilinx, *LogiCORE IP AXI Block RAM (BRAM) Controller*, version 1.03a, 2012. URL
http://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v1_03_a/ds777_axi_bram_ctrl.pdf

[13] Xilinx, *Programming and Debugging*, version 2015.1, 2015. URL
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug908-vivado-programming-debugging.pdf