Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

DMA Controller for LEON3 SoC:s Using AMBA

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Emelie Nilsson

LiTH-ISY-EX--13/4663--SE

Linköping 2013

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

DMA Controller for LEON3 SoC:s Using AMBA

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

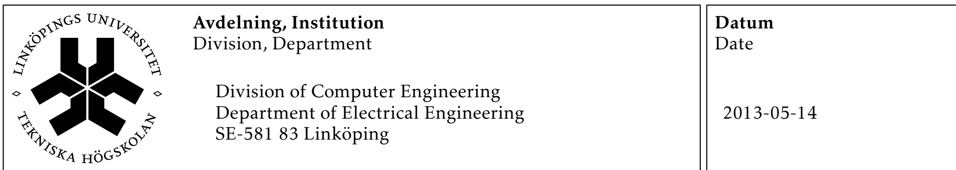
Emelie Nilsson

LiTH-ISY-EX--13/4663--SE

Handledare: **Andreas Ehliar**
ISY, Linköpings universitet
Jan Andersson
Aeroflex Gaisler

Examinator: **Olle Seger**
ISY, Linköpings universitet

Linköping, 14 maj 2013



Avdelning, Institution
Division, Department

Division of Computer Engineering
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2013-05-14

Språk
Language

□ Svenska/Swedish
⊠ Engelska/English
□ ____________

Rapporttyp
Report category

□ Licentiatavhandling
⊠ Examensarbete
□ C-uppsats
□ D-uppsats
□ Övrig rapport
□ ____________

ISBN
—

ISRN
LiTH-ISY-EX--13/4663--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version

http://www.ep.liu.se

Titel
Title DMA Controller for LEON3 SoC:s Using AMBA

Författare
Author Emelie Nilsson

Sammanfattning
Abstract

A DMA Controller can offload a processor tremendously. A memory copy operation can be initiated by the processor and while the processor executes others tasks the memory copy can be fulfilled by the DMA Controller.

An implementation of a DMA Controller for use in LEON3 SoC:s has been made during this master thesis. Problems that occurred while designing a controller of this type concerned AMBA buses, data transfers, alignment and interrupt handling.

The DMA Controller supports AMBA and is attached to an AHB master and APB slave. The DMA Controller supports burst transfers to maximize data bandwidth. The source and destination address can be arbitrarily aligned. It supports multiple channels and it has interrupt generation on transfer completion along with interrupt masking.

The implemented functionality works as intended.

Nyckelord
Keywords AHB, AMBA, APB, DMA, LEON3

Abstract

A DMA Controller can offload a processor tremendously. A memory copy operation can be initiated by the processor and while the processor executes others tasks the memory copy can be fulfilled by the DMA Controller.

An implementation of a DMA Controller for use in LEON3 SoC:s has been made during this master thesis. Problems that occurred while designing a controller of this type concerned AMBA buses, data transfers, alignment and interrupt handling.

The DMA Controller supports AMBA and is attached to an AHB master and APB slave. The DMA Controller supports burst transfers to maximize data bandwidth. The source and destination address can be arbitrarily aligned. It supports multiple channels and it has interrupt generation on transfer completion along with interrupt masking.

The implemented functionality works as intended.

Acknowledgments

Thanks to Aeroflex Gaisler for giving me the opportunity to fulfil this master thesis. A special thanks to my supervisor at the company, Jan Andersson who gave me lot of guidance.

I would like to thank my examiner at Linköping University, Olle Seger. My supervisor at the university, Andreas Ehliar also deserves thanks for all of his good advice.

Great thanks to Tomas Frostensson for giving me valuable feedback as opponent.

Göteborg, June 2013
Emelie Nilsson

Contents

Abbreviations **ix**

1 Introduction **1**
1.1 Background 1
1.2 Purpose 2
1.3 Disposition 2

2 Theory **3**
2.1 DMA 3
2.2 LEON3 4
2.3 AMBA 4
2.3.1 AHB 5
2.3.2 APB 8
2.4 Address Incrementation 9

3 Problem Identification **11**
3.1 AMBA 11
3.2 Data Transfers 11
3.3 Alignment 12
3.4 Channels 12
3.5 Interrupt Handling 12
3.6 Delimitations 12

4 Implementation **13**
4.1 Overview 13
4.2 Registers 14
4.2.1 Description 15
4.3 Channels 17
4.4 Descriptors 17
4.4.1 Description 18
4.5 Registers, Channels and Descriptors 19
4.6 Programming Languages 19
4.6.1 VHDL 19

4.6.2 C 20
4.7 AMBA 20
4.7.1 Adapter Interface 20
4.7.2 AHB 20
4.8 Bus Operations 21
4.9 Finite State Machines 22
4.9.1 AMBA 22
4.9.2 Control 23
4.10 Alignment 25

5 Results **27**
5.1 Simulation 27
5.1.1 Tools 27
5.1.2 Results 28
5.2 Hardware 29
5.2.1 Hardware Flow 29
5.2.2 Results 29
5.3 Summary 33

6 Conclusions **35**
6.1 Future Work 35
6.1.1 Optimizations 35
6.1.2 Additional Functionality 36

Bibliography **37**

Abbreviations

Abbreviation	Meaning
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
BSD	Berkeley Software Distribution
CPU	Central Processing Unit
DMA	Direct Memory Access
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose Input Output
GPU	General Public License
I2C	Inter Integrated Circuit
LUT	Look-Up Table
RAM	Random Access Memory
SoC	System on Chip
SMP	Symmetric Multi Processing
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Introduction

This chapter describes the background and purpose of the master thesis. It will also give a description of how the thesis is divided into several chapters.

1.1 Background

When a processor is performing a read or write operation, it is fully occupied and will not be able to perform more or other types of operations. The brilliance of having a DMA Controller is that the processor can be effectively offloaded. The processor can initiate a memory copy and then execute other tasks while the DMA Controller takes full responsibility for the memory copy operation and signals the processor with an interrupt when it is finished with the operation. The feature of using a DMA Controller is useful when there is no time for the processor to wait for this type of data transfer or when the pace of performing data transfers is greater than what the processor can handle.

The DMA Controller should support these specific features:

1. AMBA
2. Multiple channels
3. Memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral data transfers
4. Scatter-gather
5. Data widths of more than 32 bits
6. Use licenses such as the BSD license and/or be compatible with GPL

[Xilinx, 2011] is a DMA Controller that supports number 1, 3, 4 and 5. Another DMA Controller is [ARM, 2005] and this one supports number 1, 2, 3 and 4. [ARM, 2007] supports number 1 and 4.

No DMA Controller of the mentioned ones can fulfil the requirements. No other DMA Controllers that has been found can fulfil all of the requirements. This is a reason for implementing a new DMA Controller.

1.2 Purpose

The purpose of this master thesis is to implement a DMA Controller for use in LEON3 SoC designs. The main reason for implementing this type of controller is to be able to perform memory copy operations and data moving among peripherals without intervention from the processor.

As mentioned in Section 1.1, there are some specific features that should be supported by the DMA Controller. It must be connected to an AHB master and an APB slave. Memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral are the different data transfers that the DMA Controller should support. The implementation should be descriptor based and scatter-gather must be supported. It should be able to handle multiple channels and interrupt handling must be implemented.

When implementation and testing of the DMA Controller is finished, the performance of the DMA Controller will be compared with existing software solutions.

1.3 Disposition

A description of how the thesis is organised into different parts can be seen in Table 1.1.

Chapter	Description
1	Contains an introduction to the problem.
2	Consists of related theory.
3	Presents the identified problems.
4	Describes the implementation.
5	Contains an analysis of the results.
6	Conclusions are described.

Table 1.1: *Description of how the thesis is divided*

2

Theory

This chapter contains theory about DMA and DMA Controllers. It also includes information about AMBA along with the buses that are connected to the DMA Controller, the AHB and APB. A brief description of address incrementation is presented.

2.1 DMA

DMA is an abbreviation of direct memory access and this is a feature that allows systems to access the main memory without any help from the processor. As described in Section 1.1, a DMA Controller can offload the processor tremendously. A DMA Controller can fulfil a memory copy without intervention from the processor and when it is finished with the specified operation, it signals the processor with an interrupt.

To gain knowledge about what to take into consideration when implementing a DMA Controller and valuable information about how to implement a DMA Controller, earlier implementations of DMA Controllers was studied. Technical references concerning both DMA Controllers and AMBA was more thoroughly studied.

Explanations about how to implement an AHB master was found in [ARM, 2005]. This technical reference was considered a well defined supplement to the AMBA specification [ARM, 1999]. This DMA Controller supports data transfers such as memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral. It supports multiple channels. Information about interrupt handling and which registers that may be needed was also found in this technical reference.

[ARM, 2007] is a technical reference that contained visualisations of how the DMA Controllers logic could be separated into different FSM:s. It handles the same data transfers as the [ARM, 2005] and it also supports scatter-gather. The DMA Controller is connected to an APB slave.

The technical reference [ARM, 2012] was much like the [ARM, 2007] and therefore contained similar facts.

The report [Morris, 2000] contained information about both DMA Controllers and AMBA. It included explanations about the performance of the DMA Controller. The report also had illustrations that showed how the DMA Controller performed and the results that was given.

A DMA Controller that supports AMBA and is connected both to AHB and APB is [Ma and He, 2009]. This DMA Controller also supports multiple channels.

[Sharma, 2011] is a DMA Controller that is connected to the AHB. It can handle data transfers between memory addresses and peripherals such as I2C and USB. An interrupt is generated when the data transfer has been performed.

2.2 LEON3

The LEON3 is created by Aeroflex Gaisler [Aeroflex Gaisler, 2013]. It is a VHDL model of a 32-bit processor produced for the SPARC V8 architecture [SPARC International Inc, 1992]. The LEON3 is highly configurable and specified to be used for SoC designs. The processor is a Harvard architecture containing a 7-stage pipeline with configurable caches. It has an AMBA AHB interface and contains SMP support. It is distributed as part of the GRLIB IP Library [Aeroflex Gaisler, 2013-01-31] which allows fairly simple integration into more complicated SoC designs. Source code of the LEON3 can be found at the homepage of Aeroflex Gaisler [Aeroflex Gaisler, 2013-05-07].

2.3 AMBA

AMBA is used as an on-chip bus for various SoC designs. The AMBA protocols are de facto standard for 32-bit embedded processors, much thanks to their broad documentation [ARM, 1999].

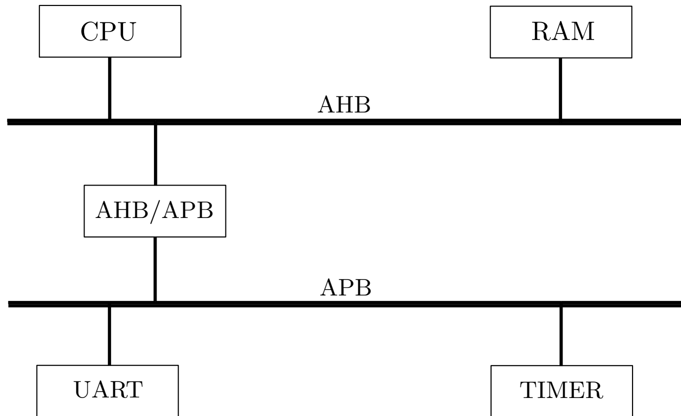


Figure 2.1: *Overview of a general AMBA system*

Figure 2.1 shows an overview of a general AMBA system attached to an AHB and APB. An AMBA system is often constructed of a processor, a RAM that is an on-chip memory, a bridge between the AHB and APB called AHB/APB and some peripherals, such as a timer and an UART.

2.3.1 AHB

The AHB is used in high performance systems. A bus transfer can be a read or write operation and these operations can take one or several bus cycles to complete. This bus supports up to 128-bit data transfers. It supports multiple bus masters.

A typical AHB system consists of the following components:

- AHB master
- AHB slave
- AHB arbiter
- AHB decoder

The bus master initiates the read or write operation by obtaining control information and an address. There can only be one operative bus master at any given time. The bus master must be granted access to the bus to be able to perform a transfer. If the bus master asserts a request signal, this grant can be received. The main task of the bus slave is to report the operating masters success, failure or waiting of the data transfer. The bus arbiter assures that only one bus master can initiate data transfers at any given time. The bus decoder provides a select signal for the involved slave and it also decodes the address that is being used at the data transfer.

There is one bus dedicated for read operations and another bus to handle write operations, the reason for this is to be able to implement an AHB system with-

out tristate drivers. The read data bus is utilized when data should be moved from a slave to the master while the write data bus is used when data should be transferred from the master to a slave.

Each transfer on the AHB consists of an address and control cycle followed by one or several data cycles. Pipeline behaviour is fulfilled thanks to that the address cycle of a transfer occurs during the data cycle of the previous data transfer. A slave has the ability to insert wait states into a data transfer, the result from this is that the data transfer will be prolonged until the slave is ready. When such an extension is made, the address cycle of the following transfer will also be extended.

Name	**Description**
HCLK	System clock.
HRESET	Reset. Active low.
HBUSREQ	Used by the master to request the bus.
HGRANT	Indicates which master that has been granted the bus.
HADDR[31:0]	Address.
HTRANS[1:0]	Type of transfer. Idle, nonsequential, sequential or busy.
HWRITE	Specifies write or read transfer. High indicates a write transfer and low a read transfer.
HSIZE[2:0]	Size of transfer. Byte, halfword or word.
HBURST[2:0]	Indicates if the transfer forms part of a burst.
HPROT [3:0]	Indicates if the transfer is an opcode fetch or data access.
HWDATA [31:0]	Used for transferring data from master to slaves during write operations.
HRDATA [31:0]	Used for transferring data from slaves to master during read operations.
HRESP[1:0]	Status of transfer. Okay, error, retry or split.
HREADY	High indicates a finished transfer.
HSEL	Slave select.

Table 2.1: *AHB signals*

Signals that can be used on the AHB are presented in Table 2.1.

HTRANS

Idle indicates no required data transfer. Nonsequential indicates the first transfer of a burst or a single transfer. Remaining transfers in a burst are sequential. Busy is used when the master is unable to complete the next transfer immediately.

HBURST

There are several different types of bursts, only incremented burst will be handled in this master thesis.

HRESP

Okay indicates that the data transfer was successful. Error shows that an error occurred during the data transfer. Retry indicates that the data transfer is not completed. Split is used when the data transfer has not been successfully completed.

Read and Write Operations

Timing diagram for read operations can be found in Figure 2.2.

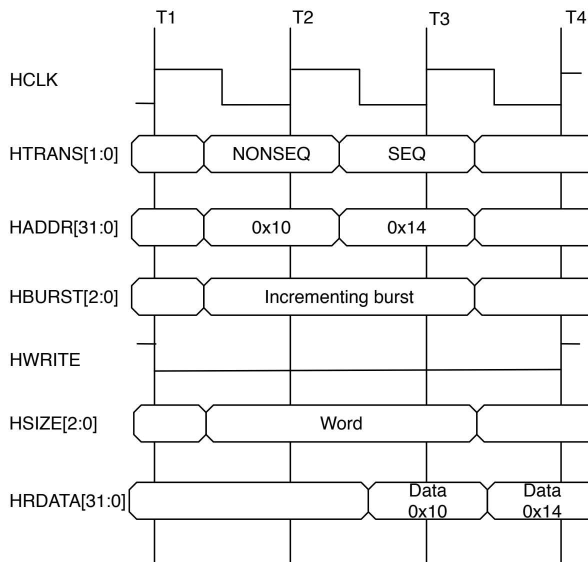


Figure 2.2: *Read mode, two word transfer starting at address 0x10*

Figure 2.2 shows how a read operation looks like once access to the bus has been granted. A write operation looks the same, except that the HWRITE signal is high and the HRDATA signal is replaced by the HWDATA signal. It is important that the address is available before the data, which can be seen in Figure 2.2.

For more information and understanding of how a read or write operation is made on the AHB and how the above mentioned signals interact, see [ARM, 1999, Figure 3-11].

2.3.2 APB

APB is a single master bus, optimized for low power consumption. This bus is used by peripherals which does not need a high performance pipelined bus interface. It always takes two clock cycles to terminate a read or write operation. This bus is connected to an AHB slave by an AHB/APB bridge that is used when converting AHB transfers into an adapting format for the slave devices on the APB. The APB bridge is the APB master, all of the other devices on the APB are APB slaves.

Name	Description
PCLK	System clock.
PRESET	Reset. Active low.
PADDR[31:0]	Address.
PSEL[15:0]	Indicates the selected slave and that a data transfer is desired.
PENABLE	Used to time all accesses on the peripheral bus. Indicates the second cycle of an APB transfer, it occurs in the middle of an APB transfer.
PWRITE	Specifies write or read transfer. High indicates a write transfer and low a read transfer.
PWDATA [31:0]	Driven by the peripheral bus bridge unit during write operations.
PRDATA [31:0]	Driven by the slave during read operations.

Table 2.2: *APB signals*

Signals that can be used on the APB are presented in Table 2.2.

The functionality of the APB is described in a fairly simple FSM, [ARM, 1999, Figure 5-2]. It has three states, idle, setup and enable. The APB remains in the setup state for one clock cycle and it is in this state that the PSEL signal is asserted. Following state is always the enable state. PENABLE is asserted in the enable state which also lasts for one clock cycle. If more transfers than one, the bus will move from the enable state to the setup state otherwise to the idle state.

See [ARM, 1999, Figure 5-3 and 5-4] for more information and understanding of how a read or write operation is made on the APB and how the above mentioned

signals interact.

2.4 Address Incrementation

Since data moving between memory spaces should be implemented, address incrementation is essential for this master thesis.

Transfer width	Increment in bytes
BYTE	1
HALFWORD	2
WORD	4

Table 2.3: *Address incrementation*

Table 2.3 shows how many bytes that the incrementation should be made with depending on the transfer width.

When accessing memory the linearly addresses will be incremented by 1, 2 or 4 bytes after each access, depending on the transfer size of the data.

If current address is a byte address, current address modulo 4 = 1 or current address modulo 4 = 3, a byte move must be made and the address incrementation will therefore be a byte. On the other hand, if current address is a word or halfword address, current address modulo 4 = 0 or current address modulo 4 = 2, a byte, halfword or word move can be made. More about this in Section 4.10.

3 Problem Identification

Features that should be supported was mentioned in Section 1.2. Here follows an identification of these features along with delimitations.

3.1 AMBA

The AHB is the far most complicated bus of the two buses that the DMA Controller will be attached to. Both the AHB and APB are well defined in the AMBA specification [ARM, 1999]. A way of receiving knowledge about how to integrate these buses along with the DMA Controller can be found in source code when connecting other cores to these specific buses that has been implemented by Aeroflex Gaisler.

3.2 Data Transfers

Data transfers that should be supported are memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral. This means that the DMA Controller must be able to move data from a memory to the same memory or to another memory. It has to support moving from a peripheral to a memory or from a memory to a peripheral and also from a peripheral to another peripheral. The DMA controller should be able to control the following peripherals in a useful way, an UART and I2C along with a SPI controller and GPIO.

The DMA Controller must have support for scatter-gather. To be able to read data from a data stream to multiple buffers or write data from multiple buffers

to a single data stream is called scatter-gather. It can be seen like gather data from or into a specified set of buffers.

A first implementation of the DMA Controller will contain simple memory copy along with scatter-gather and it will later be expanded to handle peripherals.

3.3 Alignment

It must be possible to move data from any source address to any destination address. For this, data alignment is essential. One restriction that will be made while moving data is that the source address and destination address are considered to have an offset of at least transfer size, where transfer size is the number of bytes that should be moved. When a computer reads from or writes to a memory address, it will do this in chunks that is of length word. Data alignment puts the data at a memory offset equal to a multiple of that word size. This makes it possible for the DMA Controller to move data from any given source address to any mentioned destination address.

3.4 Channels

Since the DMA Controller should be able to manage multiple channels, it must also support channel priority. At first the implementation will be made using only one channel. If the DMA Controller can support one channel in a correct manner, it should be fairly simple to expand it with multiple channels.

3.5 Interrupt Handling

The DMA Controller is supposed to send an interrupt to the processor after a finished transaction, therefore interrupt handling must be implemented. If any kind of error occurs the processor will receive an interrupt. A mask register will be used so that it is possible to choose which interrupt to generate and therefore know what kind of interrupt that will be sent.

3.6 Delimitations

All DMA operations will be performed in burst mode. All of the systems works synchronously. Only big endian must be supported by the DMA Controller, therefore there will be no endian issues. Reading and writing at the same time will not be supported. As mentioned in section 3.3, one restriction regarding the source and destination address has been made.

4 Implementation

This chapter describes how the DMA Controller was implemented. It contains information about the registers and descriptors that the DMA Controller utilizes. It also describes how the buses are connected to the DMA Controller. Explanations of the two FSM:s that provides the functionality of the DMA Controller are included. A description of how the data alignment was implemented is presented at the end of this chapter.

4.1 Overview

This section includes an overview of how the DMA Controller operates and an illustration of the DMA Controller. An enumerated list with steps of how the DMA Controller processes can be found below. Everything in this list is done in the order that it is written.

1. The core registers are configured by the processor
2. The processor enables the DMA Controller
3. One channel is read by the DMA Controller
4. Descriptor fields associated with the channel are read by the DMA Controller
5. Moving of data is performed by the DMA Controller
6. If at least one more channel exist, go to step 3 else step 7
7. The DMA Controller is disabled

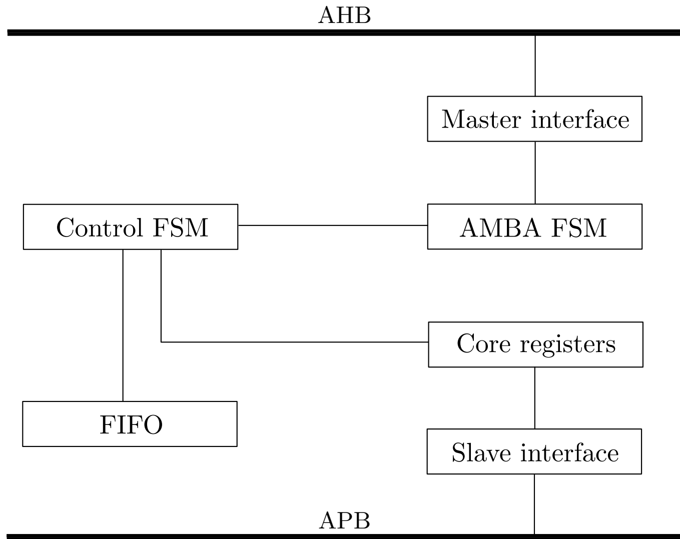


Figure 4.1: *Overview of the DMA Controller*

Figure 4.1 shows an overview of the DMA Controller. It is attached to an AHB master and APB slave. There are two FSM:s that provides the functionality of the DMA Controller, one that handles the AMBA communication and another that takes care of the control logic. The core registers are configured by the processor before the DMA Controller is enabled. The FIFO buffer is used when moving data.

4.2 Registers

This section presents the registers that has been implemented and a description of what each of the bits in the registers represent.

31 … 0
Status
Control
Base address
Current address
Interrupt
Mask

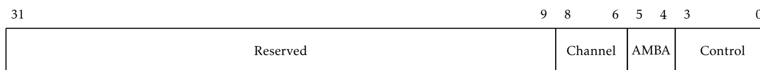
Figure 4.2: *Overview of the core registers*

Figure 4.2 shows that the existing registers contains information about the status, control, base address, current address, interrupt and mask.

4.2.1 Description

Each of the visible registers are described here.

Status



[31:9] Reserved

[8:6] Active channel
000: No channel is active
001: Channel 1 is active
010: Channel 2 is active
011: Channel 3 is active
100: Channel 4 is active
101: Channel 5 is active
110: Channel 6 is active
111: Channel 7 is active

[5:4] Current state of the AMBA FSM
00: Idle
01: Read
10: Write
11: Done

[3:0] Current state of the control FSM
0000: Idle
0001: Read channel
0010: Check channel
0011: Read descriptor
0100: Process descriptor
0101: Start read data
0110: Read data
0111: Start move data
1000: Move data
1001: Check if finished
1010: Done

More about these FSM:s in Section 4.9.

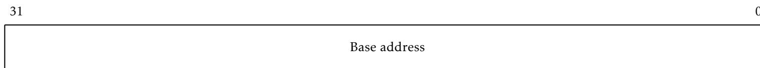
Control

[31:2] Reserved

[1] Channel enable
0: No channel is enabled
1: One or several channels are enabled

[0] DMA Controller enable
0: Disabled
1: Enabled

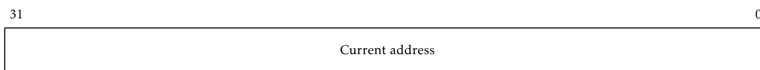
Base Address



[31:0] Base address

The channel list will always start at the base address given in this register. Therefore it is important to be sure that the address that is written in this register is correct.

Current Address



[31:0] Current address

The current address is used to show where the DMA Controller is reading or writing from.

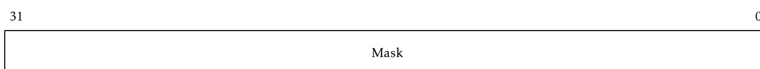
Interrupt

[31:2] Reserved

[1] Status of transfer
0: Okay
1: Error, retry or split

[0] Descriptor finished
0: Not finished
1: Finished

Mask



[31:0] Mask

Which interrupts that are granted to be sent to the processor is decided by the mask register. For example, if bit 0 is 1 and the other bits are 0 in the mask register and bit 0 and 1 are 1 in the interrupt register, only the interrupt called ”Descriptor finished” in the interrupt register will be sent and no others.

4.3 Channels

Similar to [ARM, 2005] and [Ma and He, 2009], the DMA Controller should be able to manage multiple channels and also have channel priority. The first implementation only utilized one channel. When the DMA Controller worked as expected with one channel, it was not hard to make an implementation that supported multiple channels. As described in Section 4.1, one channel is read by the DMA Controller and moving of data is performed. When the first channel is finished, next channel is read and so on. This makes it easy to know the priority of the channels. The first channel has the highest priority and the last channel has the lowest priority. It is assumed that the channels has an offset consisting of a word. For example, if the first channel has address offset 0x00, next channel will have address offset 0x04. Figure 4.4 shows how the channels are represented in main memory.

4.4 Descriptors

The implementation of the DMA Controller is descriptor based. A descriptor represents a single memory transaction from a source address to a destination address. Since the implementation is descriptor based, scatter-gather is supported, just like [ARM, 2007]. Multiple descriptors can be connected through a linked list, this can be seen in Figure 4.4. The main reason of having a descriptor based implementation is to make it as configurable as possible. The descriptors resides in main memory and is configured by the user.

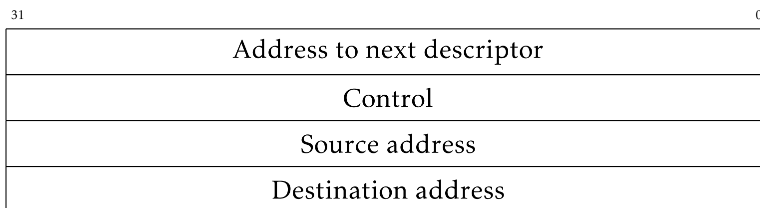


Figure 4.3: *Overview of a descriptor*

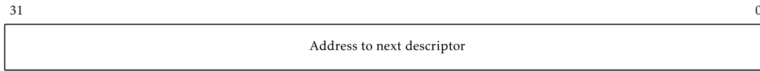
As shown in Figure 4.3, while configuring a descriptor, data fields containing the address to next descriptor in the linked list, control word, source address

and destination address of the transfer are written. The end of a linked list must be terminated with 0 in the field "Address to next descriptor".

4.4.1 Description

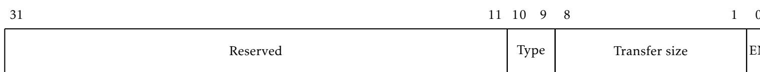
This section contains a description of the fields contained in the descriptors.

Address to next descriptor



[31:0] Address to next descriptor

Control



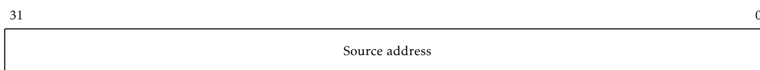
[31:11] Reserved

[10:9] Data transfer type
00: Memory-to-memory
01: Memory-to-peripheral
10: Peripheral-to-memory
11: Peripheral-to-peripheral

[8:1] Transfer size (in bytes)

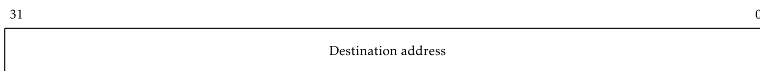
[0] Descriptor enable
0: Disabled
1: Enabled

Source address



[31:0] Source address

Destination address



[31:0] Destination address

4.5 Registers, Channels and Descriptors

Figure 4.4 shows how the registers, channels and descriptors are attached to each other.

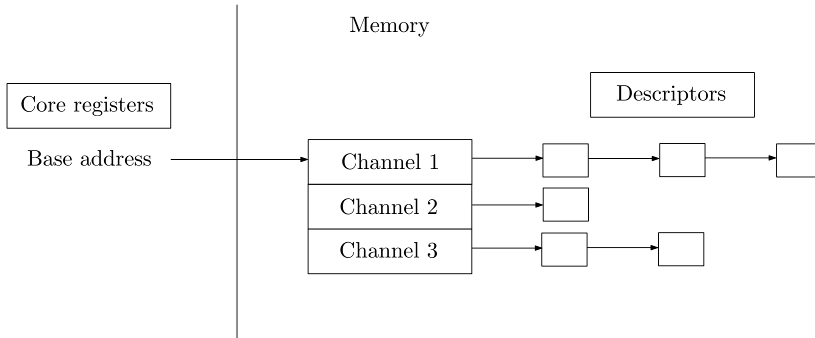


Figure 4.4: *Overview of the core registers, channels and descriptors*

The connection between the core registers, channels and descriptors can be seen in Figure 4.4. The base address, which resides in the core registers, points to the beginning of the channel list. The channels points to the descriptors that are associated with that specific channel. Both the channels and descriptors are placed in main memory.

4.6 Programming Languages

The hardware implementation is written in VHDL and the software implementation has been written in C.

4.6.1 VHDL

VHDL supports execution of concurrent processes triggered by events, the execution has nothing to do with in which order the processes are written. It will be difficult to understand and follow these processes, when the number of concurrent processes increases. In enters the two-process design method.

Two-process Design Method

Instead of writing VHDL code according to the common data flow design method, Aeroflex Gaisler uses a two-process design method [Gaisler]. This method has two processes per entity, one process containing the combinational logic and another process including the sequential logic. Thanks to this, the algorithm can be coded in sequential statements in the combinational process while the registers can be found in the sequential process.

4.6.2 C

Descriptors and core registers was configured in a C program. Memory had to be allocated to be able to create these. The fields in the descriptors was filled with appropriate data and same procedure applied for the core registers. Section 5.1 describes more about how the C program was used during testing.

4.7 AMBA

The DMA Controller is integrated with AMBA and attached to an AHB and APB.

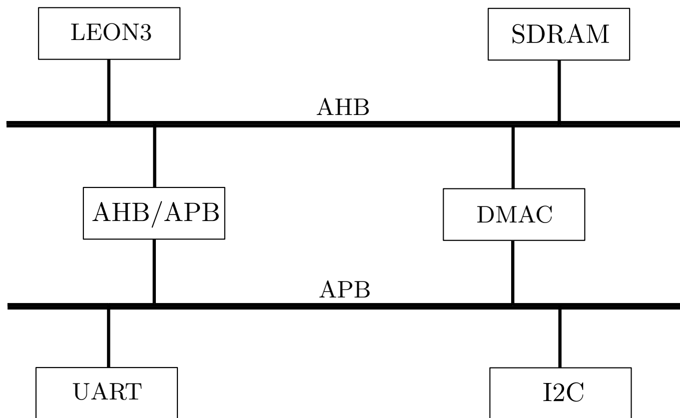


Figure 4.5: *Overview of the system and the attached buses*

Figure 4.5 shows an overview of the system and the attached buses. LEON3 is the processor that is being used. SDRAM is the memory used. AHB/APB is a bridge between the AHB and APB, the main purpose of this is to translate AHB requests to a format that is readable by the APB. The APB is connected to peripherals, such as an UART and I2C. DMAC is the DMA Controller.

4.7.1 Adapter Interface

Aeroflex Gaisler has an easier adapter interface towards the AMBA buses. This adapter interface has no documentation or specification so to be able to understand how it worked further research had to be made. The source code was therefore investigated thoroughly.

4.7.2 AHB

During the implementation signals mentioned in Table 2.1 was used. AHB signals with the adapter interface that has been used during the implementation can be seen in Table 4.1.

Name	Description
ADDRESS[31:0]	Address.
START	High indicates that data is available.
ACTIVE	High indicates that data is being processed.
READY	High indicates a finished transfer.
SIZE[2:0]	Size of transfer. Byte, halfword or word.
BURST	Indicates if the transfer forms part of a burst.
WRITE	Specifies write or read transfer. High indicates a write transfer and low a read transfer.
WDATA[31:0]	Used for transferring data from master to slaves during write operations.
RDATA[31:0]	Used for transferring data from slaves to master during read operations.
IRQ	Interrupt.

Table 4.1: *AHB signals with the adapter interface*

4.8 Bus Operations

When the connection between the DMA Controller and the AMBA buses was made, the APB was the first bus to be connected. The APB has an easier interface for read and write operations than the AHB and this was the main reason for starting with the APB. When the APB managed read and write operations, the task of connecting the AHB started. The connection between the DMA Controller and the AHB was not complicated to perform, but to be able to understand how to perform correct read and write operations, a thorough reading of the AMBA specification [ARM, 1999] was needed. This gave an explanation of how the AHB signals are dependent of each other, information about how a burst should be made on the AHB, transfer types and other important aspects to take into consideration.

The first problem that occurred during the implementation was to perform proper read and write operations over the AHB. It was hard to implement this along with the AMBA standard. The main problem while trying to implement correct read and write operations was that the AMBA standard was complicated to understand and that the information about the adapter interface was to insufficient.

4.9 Finite State Machines

There are two FSM:s that provides the functionality of the DMA Controller. One FSM handles the AMBA communication and the other FSM manages the control logic. This separation gives a better understanding of the implementation.

4.9.1 AMBA

The AMBA FSM is the least complicated FSM and it handles the bus communication.

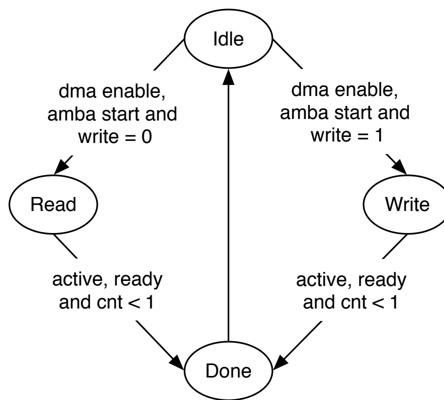


Figure 4.6: *State diagram for the AMBA communication*

Figure 4.6 shows the state diagram for the AMBA communication with the existing states idle, read, write and done.

Idle

This is the start state. When the DMA Controller is enabled and the control FSM is enabled, the AHB will be requested. If the write signal is enabled, next state will be write, otherwise read.

Read

While in the read state, if the signals active and start are enabled, a data counter will be decremented and an index counter will be increased. These counters are a way of keeping track of how much data that has been received and which position of a specified FIFO buffer that is active. An internal register called current address is increased with the transfer type, byte, halfword or word, to give information about which address that is being processed at the moment. If the data counter reaches 0, the request of the bus stops. When the signal ready is enabled, reading of data from the AHB is made. When everything has been read, done will be the next state.

Write

A state that is much like the read state is the write state, but when data is read from the AHB in the read state, data is instead being written.

Done

This state is only present for having a finishing state, the next state will immediately be idle.

4.9.2 Control

A more complex FSM is the control FSM that handles the control logic.

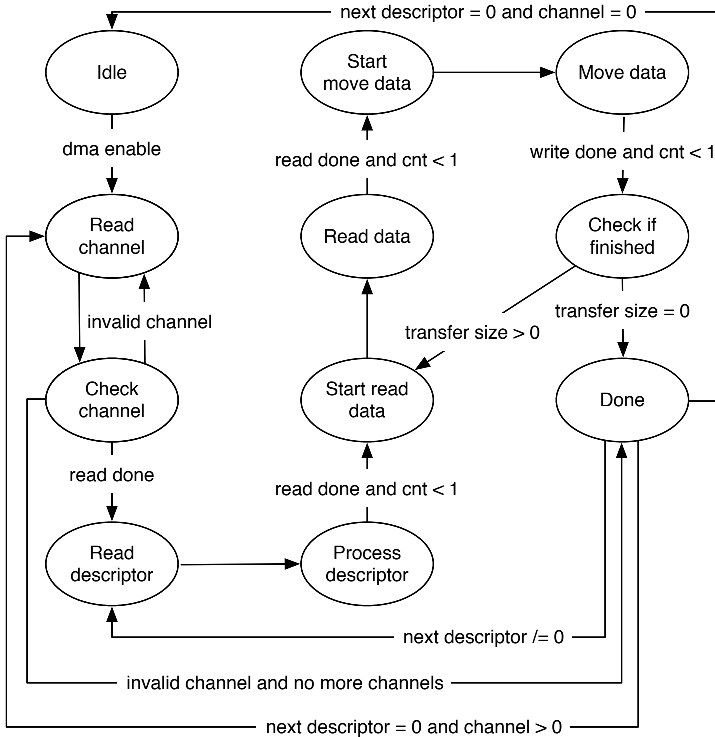


Figure 4.7: *State diagram for the control logic*

Figure 4.7 shows the state diagram for the control logic with the existing states idle, read channel, check channel, read descriptor, process descriptor, start read data, read data, start move data, move data, check if finished and done.

Idle

The idle state is the start state. When the DMA Controller is enabled, the current address is set to the base address. A counter is set to a constant value and this value represents the number of channels that will be used. The state that follows is read channel.

Read Channel

The AMBA FSM will be started and reading of one channel will therefore start. The state changes to check channel.

Check Channel

Wait until one channel has been read. If it was an invalid channel and there is at least one more channel, next state will be read channel. It it was an invalid channel and no more channels, next state is done. If the channel was valid, the number of channels decreases and next state is read descriptor.

Read Descriptor

A counter is set to a constant value and this value is the number of fields that will represent a descriptor. The AMBA FSM is started so that it will be possible to read the fields that the descriptor consists of. The following state is process descriptor.

Process Descriptor

This state saves all of the information that a descriptor has into an internal register. When all of the information is saved it is time to move on to the state that starts to read the data.

Start Read Data

A counter is set to a constant value and this value represents the transfer size. Current address will be the source address and the AMBA FSM is started so that the data can be read. Next state is read data.

Read Data

When all of the data that should be moved has been read it is saved into a FIFO buffer. Data alignment has been performed to give knowledge about which data that should be saved and then moved. More about the implementation of the data alignment in Section 4.10. Next state is start move data.

Start Move Data

Wait until all of the data has been saved correctly. Current address will be the destination address. The AMBA FSM is started, this makes it possible for the data to be written and next state is move data.

Move Data

The data is moved and after each specific move of data, the source address, transfer size and destination address are updated. It is possible to see if all of the data has been moved in next state.

Check If Finished

If transfer size is larger than 0, next state will be start read data otherwise done.

Done

If there is at least one channel and the address to next descriptor is 0, next state will be read channel and current address will be set to next channel. The DMA Controller is disabled if the number of channels are 0 and the address to next descriptor is 0. Next state will be idle if the DMA Controller is disabled. If the address to next descriptor is anything but 0, internal registers will be set to 0, current address will be set to the address to next descriptor and next state will be read descriptor.

4.10 Alignment

Data alignment had to be implemented for the possibility to move data from and to more addresses than only word addresses.

If the source address or destination address starts at a byte address or if transfer size is one byte, the size of the data that will be moved is a byte. If the source address or destination address starts at a word address or a halfword address or if the transfer size is less than four bytes, the size of the data that will be moved is a halfword. One burst can manage to move a constant set of maximum words at a time. This constant can of course be configured to any number, but a number that is a power of two is preferred since it dictates the size of the FIFO buffer used. If transfer size is larger than the maximum transfer size that can be moved, the size will be a word and the transfer size will be set to the maximum transfer size. In other cases, the size of the data that will be moved is a word.

Following expressions are made as an attempt to simplify the pseudo code in Algorithm 4.1:

N = 1, 2 and 3
srcN := source address modulo 4 = N
destN := destination address modulo 4 = N

Algorithm 4.1 Pseudo code for data alignment

```
if (src1 or src3 or dest1 or dest3 or (transfer size = 1)) then
    size := byte;
    transfer size := 1;
else if (src2 or dest2 or (transfer size < 4)) then
    size := halfword;
    transfer size := 2;
else
    if transfer size > transfer max then
        size := word;
        transfer size := transfer max;
    else
        size := word;
    end if
end if
```

Algorithm 4.1 shows pseudo code for how the data alignment was implemented.

5 Results

This chapter describes results from the testing during the implementation of the DMA Controller. Verifications has been made in simulation and on the specified hardware.

5.1 Simulation

Simulations was performed during the implementation to verify that the DMA Controller functioned properly.

5.1.1 Tools

Different tools was used during the simulation. It was possible to see if the implementation processed as expected in simulation with help from these tools.

C-program

A program in C for testing was developed continuously during the implementation. Arrays with predefined values was allocated and created. The source address and destination address in each descriptor was set to these arrays. The transfer size was also chosen. Since the source address, transfer size and destination address was predefined, it was easy to see if the DMA Controller moved correct data from the expected source address to the correct destination address.

ModelSim

While simulating, ModelSim SE version 10.0c was used. A test bench was used which included LEON3, so both the processor and the DMA Controller could be seen in the simulation.

AHB Trace

Aeroflex Gaisler has created a trace tool that can be used on the AHB. While simulating in ModelSim, the transcript window is filled with output data. This shows which master that made a transfer on the AHB, at which time, if it was a read or write operation, number of bytes that was transferred, source address and destination address. This was a perfect tool for testing the DMA Controller on the AHB since it shows everything that is needed to be able to verify that correct data was moved from the expected source address to the intended destination address.

5.1.2 Results

Here are some test examples that was tested along with the given results. All of the results could be seen in ModelSim and with help from the AHB trace.

5.1 Example

Number of bytes to transfer = 9
Source address = 40FFFE64
Destination address = 40FFFE30

Both the source and destination address are word addresses. One word is moved and the source and destination address are incremented with four bytes. Number of bytes to transfer is decreased with four bytes. Same thing will happen again and after that only one byte will be moved. To sum up, two words was moved followed by one byte.

5.2 Example

Number of bytes to transfer = 5
Source address = 40FFFE60
Destination address = 40FFFE32

The source address is a word address and the destination address is a halfword address. One halfword will be moved and therefore both addresses will be increased with two bytes while the number of bytes to transfer will decrease with two bytes. The source address is now a halfword address while the destination address becomes a word address. One more halfword will be moved and finally one byte is moved. In other words, one halfword was moved followed by another halfword and at last one byte was moved.

5.3 Example

Number of bytes to transfer = 7
Source address = 40FFFE61
Destination address = 40FFFE31

Both the source and destination address are byte addresses. One byte is moved and both addresses increases with one byte. Both addresses becomes a halfword address. Number of bytes to transfer has been decreased with one byte. Next move that is performed will be a halfword. The source and destination address are increased with two bytes and number of bytes to transfer is decreased with two bytes. The source and destination address are now both word addresses and the last move will be a word. To recap, one byte followed by one halfword and finally a word was moved.

More tests has been performed to verify that the DMA Controller works properly in simulation but only a few are presented here since the overall logic for the DMA Controllers behaviour will be the same for all of the transfers that it performs.

5.2 Hardware

When the DMA Controller worked properly according to the simulation, it was time to try it on hardware. The FPGA that has been used is Xilinx ML501, Virtex-5 XC5VLX50 with a speed grade of -1. During these tests, data was moved from an address in the DDR-memory to another address in the same memory using the AHB.

5.2.1 Hardware Flow

Before the implementation can be run on the FPGA, synthesis and place and route has to be run. The synthesis creates a netlist. The purpose of a netlist is to connect gates or flip-flops together. With help from the process place and route, the netlist is physically placed and mapped on the FPGA. After these steps a binary file is generated and this is used for configuring the FPGA.

Timing analysis was made to verify that the DMA Controller operated as expected. At first it did not. The code needed reconstruction to eliminate these timing problems. It was fairly simple to remove the timing issues, but because of the elimination of these, other problems occurred. The FSM:s needed minor adjustments and after this, all of the problems was resolved.

To synthesize and place and route the design, tool version 14.2 was used. The core has only been tested while running with frequency 80 MHz on the mentioned board in Section 5.2. Some other cores that are attached to the system can not handle a higher frequency than 80 MHz and that is why the DMA Controller has been tested only with that specific frequency.

5.2.2 Results

Aeroflex Gaisler has a tool much like the AHB trace, mentioned in Section 5.1.1, but for hardware. Thanks to this, it was easy to verify that the DMA Controller operated as expected on the FPGA. After this verification, it was time to look closer on the given results.

Comparison With Software Solutions

A program in C was implemented so that the processor would move a specified number of bytes. This program works in such a way that the built in function memcpy in C is used to move a specified number of bytes. The implementation of the memcpy function is optimized in terms of alignment. The time that it took for the processor and DMA Controller to move a smaller amount of bytes was compared which can be seen in Figure 5.1. Comparison when moving several words can be seen in Figure 5.2.

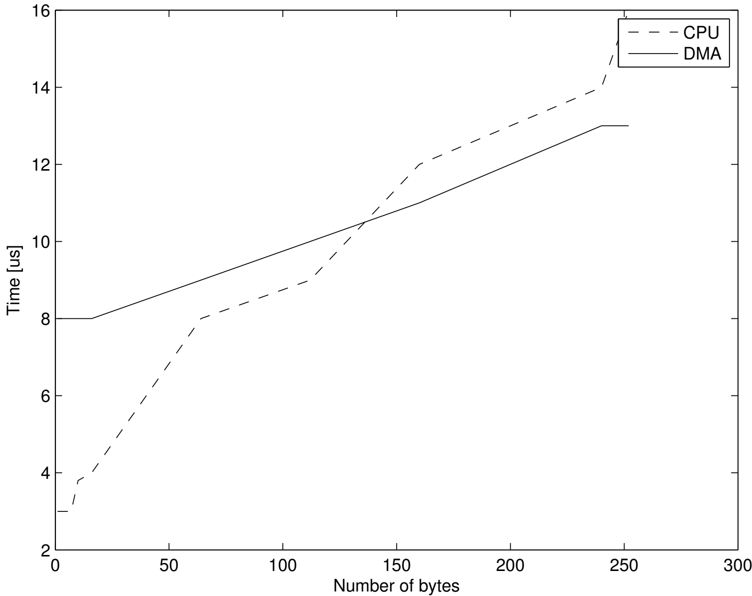


Figure 5.1: *Comparison between the processor and DMA Controller including descriptor configuration*

During the comparison that can be seen in Figure 5.1, the descriptor configuration was included. The processor starts at a lower time than the DMA Controller and the processor's curve increases more rapidly than the DMA Controller's which can be seen in Figure 5.1. Before the DMA Controller can start to move data, its descriptors needs to be configured. This configuration takes considerable time compared to the actual moving of a smaller number of bytes. When moving a smaller number of bytes, the processor is faster than the DMA Controller because of this configuration.

As can be seen in Figure 5.1, the measured time is of small magnitude and varies. One conclusion that can be drawn from the information given in Figure 5.1 is that the DMA Controller is more efficient for moving of data than the

processor after approximately 35 words.

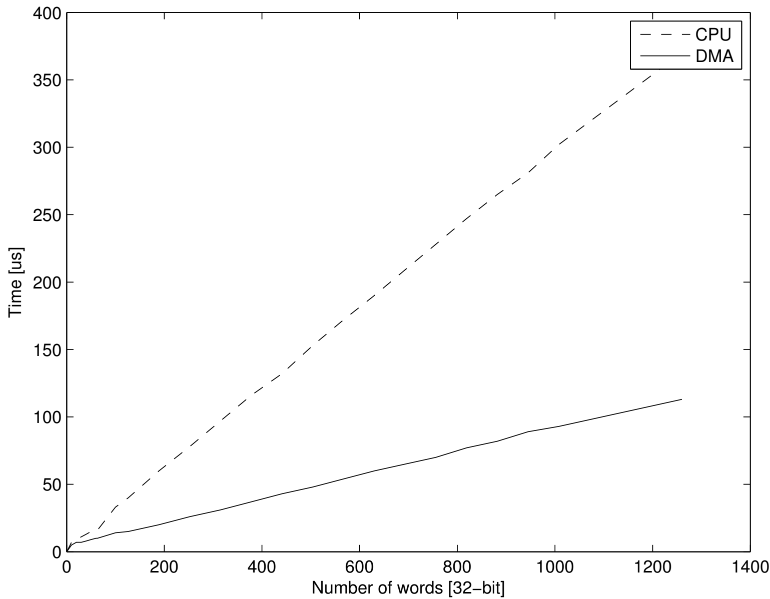


Figure 5.2: *Comparison between the processor and DMA Controller without descriptor configuration*

Figure 5.2 shows how long time it took for the processor and DMA Controller to move a specified number of words. The descriptor configuration was not included during this comparison. The processor's curve increases more rapidly than the DMA Controller's.

Alignment

Data alignment has been implemented so that the DMA Controller can move data from and to other addresses than just word addresses.

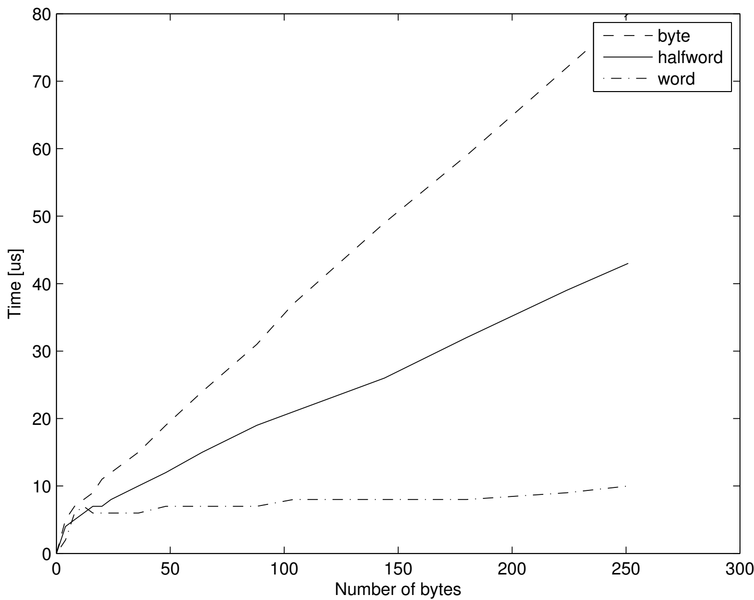


Figure 5.3: *The DMA Controller moves only bytes, only halfword and one byte, one halfword and then only words*

In Figure 5.3, the line named byte shows how the DMA Controller operates when only moving bytes. The reason that the DMA Controller only moves bytes is because at least one of the source and destination addresses will be a byte address. Alignment of the source and destination addresses to halfword or word addresses is impossible. The line called halfword shows how the DMA Controller operates when only moving halfwords. The source and destination addresses will always be a word address and a halfword address or a halfword address and a word address. It is impossible to align the source and destination addresses to word addresses. Finally, the line named word shows how the DMA Controller operates when first moving one byte, one halfword and then only words. Alignment of the source and destination addresses to halfword and/or word addresses is possible.

The line named word in Figure 5.3 shows that the implemented data alignment works successfully. The source and destination addresses starts at byte addresses, which becomes halfword addresses and finally word addresses. When the source and destination addresses are word addresses, only words are moved. When at the end of the data transfer, the last move can be a byte, a halfword or a word, depending on the predefined transfer size.

Area

While looking at the area consumption, the most interesting resource units was the number of flip-flops, LUT:s and memories.

Unit name	Number of units	Percentage
Flip-Flop	656	6.6%
LUT	1197	6.7%
Memory	0	0%

Table 5.1: *Number of used resource units*

Table 5.1 shows that the implementation uses 656 flip-flops, 1197 LUT:s and 0 memories. The percentage is the percentage that the DMA Controller utilizes compared to the whole system.

The number of flip-flops and LUT:s can be decreased by optimizing the internal registers that are used. This optimization has not been done for the sake of readability and understanding of the implementation. It is a question of what the person that will be using it wants, an acceptable implementation that is fairly easy to understand or an optimized implementation that is harder to comprehend with the gain of less hardware.

The core registers mentioned in Section 4.2.1 must be present. The status and control registers are not fully utilized since there are some reserved bits. These two registers could be merged into one, but the separation gives better readability. The internal registers are fully utilized, but optimizations concerning resource sharing can be made, more about this is Section 6.1.

Section 4.4.1 presents the fields in a descriptor. These fields can not be more optimized. One descriptor was represented by 6 words at the beginning of the implementation. The status and transfer size words was removed. The transfer size is now represented by 8 bits in the control word, which can be seen in Section 4.4.1.

Mathematical operations are performed, these could perhaps be done in a more optimized way and thus minimize hardware resources. Instead of performing many different additions with multiple adders, one adder could be used for several or all additions.

5.3 Summary

The DMA Contoller supports AMBA and is attached to an AHB master and APB slave. The implementation is descriptor based and scatter-gather works properly. Each descriptor has the ability to move up to 255 bytes. Each transfer is made in burst mode, this will decrease time spent on bus accessing. It has data alignment, which means that the source and destination address can be

arbitrarily aligned. Interrupt handling along with interrupt masking is implemented. The DMA Controller can handle multiple channels.

The DMA Controller runs with frequency 80 MHz. Figure 5.1 shows that the DMA Controller is more efficient for moving of data than the processor after approximately 35 words. Figure 5.2 shows that the DMA Controller is much faster than the processor when a large set of 32-bit words are moved. Figure 5.3 shows how the DMA Controller operates when moving only bytes, only halfwords and one byte, one halfword and then only words. As described in Section 5.2.2, the DMA Controller utilizes 656 flip-flops, 1197 LUT:s and 0 memories.

6 Conclusions

This chapter describes future works that can be made on the DMA Controller. Thoughts about how to implement new features are presented.

The main conclusion that can be made from this master thesis is that a successfully working DMA Controller for use in LEON3 SoC:s integrated with AMBA could be implemented. Implemented functions works as expected and the used hardware presented in Table 5.1 is acceptable. The implementation should not need much adjustments in the future, but more features can of course be added.

6.1 Future Work

The DMA Controller can support up to 7 different channels. If this is not enough, expand the number of bits that represent the number of channels in the status register mentioned in Section 4.2.1.

6.1.1 Optimizations

To be able to make the type of optimization mentioned in Section 5.2.2, different design choices must be made. For example, the descriptor fields and the data that should be moved could might be saved in another way that requires less hardware. The data that should be moved is saved into a FIFO buffer. This FIFO buffer consists of 8 words, since the DMA Controller should be able to move at maximum 8 words in each burst. It is not ideal that the FIFO buffer has a predefined size of 8 words, since the DMA Controller could be used to move less than 8 words. Instead of thinking of it as a FIFO buffer implemented with flip-flops, it could be seen as a regular memory. Transform the FIFO buffer into a

specific memory could be a way of reducing hardware and making the implementation more efficient.

The implementation consists of several counters. One of these is used for keeping track of the number of channels, descriptors and to be able to know what the transfer size is. The counters that give channel information are not behaving like the first counter, they do not perform several tasks. These counters could be optimized. One counter decreases the number of predefined channels, another counter increases the number of channels so that it is easy to know which channel that is in progress and the last channel counter counts the number of invalid channels. It should not be difficult to merge these three channel counters into one, just like the first mentioned counter.

6.1.2 Additional Functionality

As mentioned in Section 4.4.1, there are two bits in the control word that describe what kind of data transfer that shall be performed. Knowledge of which kind of data transfer that is requested is essential if the DMA Controller will be able to control the peripherals described in Section 3.2. In other words, the idea is to use these two bits in future works while connecting the DMA Controller to the peripherals mentioned in Section 3.2. The DMA Controller can not handle these peripherals for now. To be able to control these peripherals with the DMA Controller, the first step is to read the source code and documentation for these specific cores. Source code for these cores can be found at the homepage of Aeroflex Gaisler [Aeroflex Gaisler, 2013-05-07] and documentation for the cores can be found in the GRLIB IP Library [Aeroflex Gaisler, 2013-01-31]. To be able to succeed with this, the hardware must be adapted to the new bus. This type of future work will require additional adjustments of the implemented DMA Controller.

The signal HRESP is described in Section 2.3.1. It has four different states: okay, error, retry and split. The DMA Controller becomes disabled when in states: error, retry or split. This is unnecessary for states retry and split. Instead of disabling the DMA Controller, the bus should be requested and attempts to complete the transfer could be made.

Bibliography

Aeroflex Gaisler. LEON3. 2013. URL http://gaisler.com/index.php/products/processors/leon3.

Aeroflex Gaisler. GRLIB IP Library User's Manual. 1.2.2 - B4123, 2013-01-31. URL http://gaisler.com/products/grlib/grlib.pdf.

Aeroflex Gaisler. 2013-05-07. URL http://www.gaisler.com.

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

ARM. PrimeCell DMA Controller (PL080), Technical Reference Manual. r1p3, 2005. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0196g/DDI0196.pdf.

ARM. PrimeCell DMA Controller (PL330), Technical Reference Manual. r0p0, 2007. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424a/DDI0424A_dmac_pl330_r0p0_trm.pdf.

ARM. CoreLink DMA-330 DMA Controller, Technical Reference Manual. r1p2, 2012. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424d/DDI0424D_dma330_r1p2_trm.pdf.

Jiri Gaisler. A structured VHDL design method. URL http://www.gaisler.com/doc/vhdl2proc.pdf.

Guoliang Ma and Hu He. Design and implementation of an advanced dma controller on amba-based soc. In *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, pages 419–422, 2009.

Gareth Morris. DMA Controller for AMBA Bus IP Core. 1.0, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.2986&rep=rep1&type=pdf.

Chetan Sharma. General Purpose AHB-DMA Controller. *International Journal of Computer Technology and Applications*, 02(02):248–252, 2011.

SPARC International Inc. The SPARC Architecture Manual. 8, 1992. URL http://www.sparc.com/standards/V8.pdf.

Xilinx. LogiCORE IP Direct Memory Access. 4.00.a, 2011. URL http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v4_00_a/ds852_axi_dma.pdf.

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`

© Emelie Nilsson