AN ASYNCHRONOUS DMA CONTROLLER

A thesis submitted to the University of Manchester
for the degree of Master of Philosophy
in the Faculty of Science and Engineering

April 2000

By
Chatchai Jantaraprim
Department of Computer Science

Contents

Abstract **10**

Declaration **11**

Copyright **12**

Dedication **13**

Acknowledgements **14**

1 DMA Controllers **15**

1.1 Introduction 15

1.2 Existing DMA controllers 18

1.2.1 8237 Intel DMA controller 18

1.2.2 National Semiconductor NS32230 DMA controller 19

1.3 Summary 21

2 Asynchronous Logic Design **22**

2.1 Introduction 22

2.2 Asynchronous logic design 24

2.3 Micropipelines 27

2.4 The AMULET Processors 28

2.5 Summary 29

3 The AMULET3i Subsystem **30**

3.1 Introduction 30

3.2 The MARBLE Bus 31

3.2.1 Initiator Interface 32

3.2.2 Target Interface 33

3.2.3 Type of data transfer 34

3.3 Processor-Memory Subsystem 35

3.3.1 The AMULET3 Processor Core 35

3.3.2 Local RAM 36

3.3.3 The Processor Local Buses 36

3.4 Other Devices 37

3.4.1 On-chip ROM 37

3.4.2 External Memory Interface 37

3.4.3 The Asynchronous Peripheral Devices 37

3.4.4 Test Interface Controller 38

3.5 The DMA Controller 38

4 Top Level Design **39**

4.1 Introduction 39
4.2 Specifications 39
4.3 Terms Definition 40
4.4 DMA operation overview 42
4.5 The DMA controller internal structure 42
4.6 The Register Unit 43
4.6.1 The Global Register Unit 44
4.6.2 The Channel Registers 46
4.7 The Transfer Engine Unit 47
4.7.1 The Channel Arbiter Unit 49
4.7.2 The Transfer Control Unit 50
4.8 Summary 51

5 Design Issues **52**
5.1 Introduction 52
5.2 DMA Registers 52
5.2.1 Transfer Control Configurations 53
5.2.2 Processor Interrupt Control Configuration 54
5.2.3 Registers Access Conflict 54
5.3 DMA Operation 55
5.3.1 DMA Operation Overview 55
5.3.2 Channel Selection 55
5.3.3 Data Transfer Operation 57

5.3.4 Register Communication Models 57
5.3.5 End of Run Indication Mechanism 59
5.4 Shared resources in the DMA controller 61
5.4.1 Dual-Ported Memory 62
5.4.2 Registers locking 62
5.4.3 Arbitration 63
5.5 Arbitration 63
5.5.1 MARBLE Arbitration 64
5.5.2 DMA Registers Arbitration 64
5.5.3 Channel Arbitration 65
5.6 Problems 70
5.6.1 Deadlock 70
5.6.2 Race condition 71
5.7 Synchronization 73
5.7.1 Peripheral Synchronization 74
5.7.2 Processor Synchronization 75
5.8 Summary 75

6 Behavioural Models **76**

6.1 Introduction 76
6.2 Modelling tools 77
6.3 Simplified models 78
6.4 AMULET3i DMA Controller Model 79

6.5 Model of the Registers 80
6.5.1 Global Registers 81
6.5.2 Channel Registers 81
6.6 Model of the Transfer Engine 82
6.6.1 Transfer Control Module 82
6.6.2 Channel Arbiter 82
6.7 Simulation Schemes 83
6.8 Simulation Results 83
6.8.1 Deadlock 84
6.8.2 Early Interrupt Request Sending 84
6.8.3 Race Condition 85
6.9 Summary 89

7 Conclusions **90**
7.1 The AMULET3i DMA Controller 90
7.2 Conclusions 91

Bibliography **94**

List of Tables

List of Figures

1.1 8237 in cascade mode 19

2.1 Signalling protocols 26

2.2 Structure of a simple micropipelines 27

3.1 The AMULET3i Subsystem 31

3.2 The MARBLE interfaces 32

3.3 Initiator interface data transfer operations 33

3.4 Target interface data transfer operations 34

4.1 Block diagram of the DMA controller 43

4.2 Register Unit 44

4.3 Global Registers 45

4.4 Channel Registers 46

4.5 Control Register 48

4.6 Channel Request Mapping Unit 48

4.7 Internal structure of Channel Request Mapping Block 48

4.8 Transfer Engine Unit 49

4.9 Channel Arbiter Unit 50

5.1 Sequence of the DMA operation 56

5.2 Register communication with wide bus 58

5.3 Register communication with narrow bus 60

5.4 Global Registers Arbitration 66

5.5 Balanced arbitration tree 67

5.6 Balanced tree order of gaining access 68

5.7 Unbalanced arbitration tree 68

5.8 Unbalanced arbitration tree 69

5.9 System Deadlock Problem 72

5.10 Peripheral Request Handshake 74

6.1 Internal Modules of the DMA Controller 80

6.2 Deadlock cause by atomic register access by transfer engine 84

6.3 Non-atomic register access solve the deadlock problem 85

6.4 Early interrupt sending 86

6.5 Delay registers write back for last data transfer 86

6.6 Race Condition 87

6.7 Using a flag to prevent Race Condition 88

Abstract

An asynchronous Direct Memory Access (DMA) controller for an asynchronous microprocessor subsystem has been designed. Behavioural modelling and simulation of the DMA controller was performed using LARD, a hardware description language for asynchronous logic design. Problems in designing the DMA controller using an asynchronous logic design methodology are discussed, and solutions to these problems are presented.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of Department of Computer Science.

Dedication

To HangChee & HYingKlang, with love.

Acknowledgements

Thanks a lot for being my supervisor and not given up reading/correcting/ trying to understand the rubbish I have written in my thesis and for showing me the interesting bits of the asynchronous logic design, thank you Jim.

Thanks for being friends, correcting my English, playing games, opening the world of asynchronous logic design to me, and most of all giving me the most enjoyable moment of living here, my dear friends and brothers, Andrew and John.

Thanks for advising, giving the 'stucture' to my thesis, running, orienteering, trailquest, (un-successful) uni-cycling, and inspiring my juggling, Doug.

Thanks a lot to Chung for a number of interesting discussions, even though for many of them I didn't agree with you, but it is still interesting anyway.

Thanks a lot to Phil, for LARD, that is where I started.

Thanks a lot to all of AMULET people for these more than two years of companionship.

Chapter 1

DMA Controllers

1.1 Introduction

In a computer system the processor spends a considerable amount of time moving data around, even though it does not perform this very efficiently. In order to transfer a single datum, a processor has to fetch instructions from memory, decode and execute these instructions and perform the transfer. A simple data transfer operation such as reading/writing a block of data from/to a peripheral device can burden the processor in fetching, decoding and executing instructions most of the time instead of performing the actual data transfer.

An autonomous computer device which could be used to relieve the processor from this task is a Direct Memory Access (DMA) controller. A DMA controller is a simplified processor which can only perform the function of data transfer between devices. It need not be able to perform any other functions, such as reading and interpreting instructions, so it can work faster and more efficiently than the central processor.

A DMA controller, after being programmed, can perform data transfers without processor intervention. The processor is then free to perform other functions. The data transfer process may also be faster and use less power since the DMA controller need not fetch, decode and execute instructions to perform the transfer.

A DMA controller can increase system performance and reduce power consumption in this way (although the latter factor was not of much concern in the design of existing DMA controllers). A DMA controller would usually be used in small computer systems, e.g. a micro-computer, or engineering workstation [3, 5]; for larger computer systems, such as a super-computers, mainframes or mini-computers, other kinds of controllers or input/output processors are used to control data transfers, or handle input/output [18, 2, 22, 14].

Typically the DMA controller would be used to transfer data between a peripheral device and memory, although memory to memory and peripheral to peripheral transfers are possible.

Data transfer between devices is usually done via a bus. This allows a device to transfer data to/from other devices without a direct connection which would impose a high wiring cost when the number of devices is large. The bus simplifies connection between devices in the computer system.

Devices can be divided by behaviour into initiator devices and target devices. An initiator device is a device that can initiate a request to read/write data from/to another device, e.g. the processor or the DMA controller. A target device is one that sends or receives data when it receives read/write requests from another device, such as memory or peripheral device.

DMA controllers are both bus initiator and target devices. When a processor programs a DMA controller, the controller receives values as a target device; when it performs autonomous DMA transfers it behaves as an initiator.

Target devices can be divided by speed of response into “fast” and “slow” devices. A fast device, e.g. a memory device, responds immediately when it receives a request from the initiator device. This kind of device stores data locally and usually is one built from electronic components instead of electro-mechanical components.

Slow devices might be built from electro-mechanical components, e.g. a hard disk, or not have data locally, e.g. communication devices. To read from or write

to a slow target device the initiator must wait until it is ready to transfer data or use some mechanism such as polling or interrupts to perform the transfer.

Waiting for the transfer is inefficient because this will occupy the bus for an indefinite period, bringing other system activity to a halt. In the polling scheme, the initiator device checks the status of the target device periodically until the target device is ready before initiating the data transfer. With the interrupt scheme the target device sends a notification signal to the initiator device when it is ready to perform a transfer.

In this approach the notification signal from the target device can be regarded as a transfer request, but it is up to the initiator to initiate read or write transfers; alternatively the initiator can ignore this request.

To perform a DMA transfer the DMA controlller must first be programmed with necessary information by the processor. The DMA controller can then perform the DMA transfer autonomously.

To perform each individual transfer the DMA controller must wait until transfer conditions are reached. These conditions are:

- The DMA controller was programmed and enabled by the processor
- Both source and destination devices are ready for transfer data

When both source and destination devices are ready the DMA controller reads data from the source device, writes it to the destination device, updates its pointers and counter, and checks for conditions to stop the transfer. The DMA controller will normally repeat this process until a termination condition is reached and will notify the processor if required.

To exchange data with a slow peripheral device an interrupt mechanism is used to notify the DMA controller when devices are ready to receive or transmit data. The request signal from a device to the DMA controller is usually called a ‘DMA request’.

1.2 Existing DMA controllers

DMA controllers have been used in computer systems for many years. One of these designs [3] has been used in several generations of computers without any significant change in its specifications, except some speed enhancement to match faster system bus speeds.

Some of these designs' specifications have influenced the design of the asynchronous DMA controller discussed in this thesis. Two examples are given below.

1.2.1 8237 Intel DMA controller

The Intel 8237 DMA controller was designed to be used with Intel 80x86 microprocessor family, as used in the IBM PC and compatible computers. The original 8237 design was designed for an 8-bit data bus; a newer one also supports a 16-bit data bus. An 8237 has four independent DMA channels which are cascadable; a channel may be used to connect to another 8237 and expand the number of DMA channels in the system. The number of channels which can be implemented by cascading these DMA controllers together is virtually unlimited. As shown in figure 1.1.

This DMA controller is designed specifically to transfer data between a peripheral device and a memory device. Transfers between memory devices are also supported, but two channels must be used together for this type of transfer. Peripheral to peripheral transfer is not supported. If the DMA controllers are cascaded one of channel is used for cascading and can't be used for DMA transfer.

This DMA controller contains two set of registers which are used by the processor to program the DMA transfer characteristics. While a channel is in use the primary set of register is used; when the transfer sequence finishes, and if the secondary set of register was programmed by the processor, the DMA controller copies its register values from the secondary set to the primary set and continues. This allows the processor to program a DMA channel for a subsequent transfer

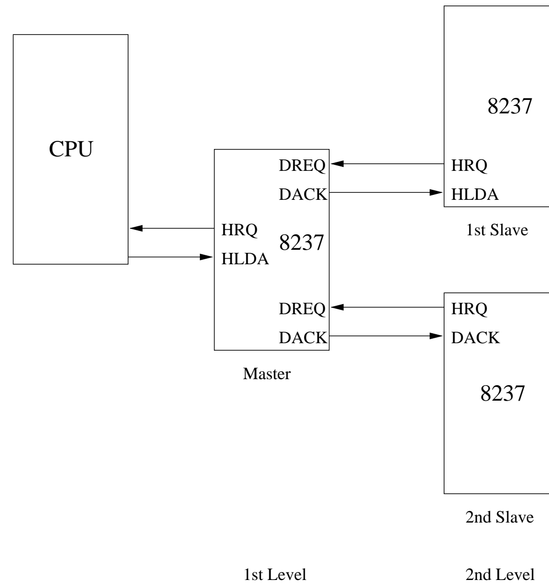

Figure 1.1: 8237 in cascade mode

even though that channel is still in use.

Each peripheral request signal is permanently fixed to a particular channel. The identity (number) of a peripheral device supported in the DMA transfer is hard-wired to a specific channel. Each channel has a 16-bit counter, so the maximum amount of data it can transfer is 2^{16} items. Data transfer can be done only through the bus and occupies two cycles; data is read from the source device to the DMA controller during the read-cycle and written from the DMA controller to the destination device during the write-cycle.

There are several variants of 8237 DMA controllers with different enhancements. In most recent PCs which use an 80x86 processor or one of its variants, the 8237 DMA controller is part of the chipset on the mainboard [4].

1.2.2 National Semiconductor NS32230 DMA controller

The NS32230, National Semiconductor DMA controller, is designed to be used with the NS32000 processor family.

Features of the NS32230 DMA controller:

- Able to transfer data between memory devices and between peripheral devices in the same way as 'conventional' transfers between memory and a peripheral device.

- 8/16-bit transfer

  The DMA controller can be used with an 8- or 16-bit system bus. The transfer to/from 8-bit peripheral devices also supports assembly/disassembly of data words, i.e. 8-bit $\leftrightarrow$ 16-bit. Two 8-bit data reads from the peripheral device can be combined into 16-bit data for a single write to a 16-bit memory or peripheral device.

- Remote/local configuration

  In its 'remote' configuration the DMA controller and devices are connected to a dedicated bus on which data transfer is performed. In 'local' configuration the DMA controller and devices share the bus with the processor. In remote configuration the processor can perform other functions which use the bus while the DMA transfer is in progress on its dedicated bus. This needs a two bus system.

- Fly-past and store-and-forward transfers

  In the store-and-forward data transfer the operation is separated into read and write operations: the DMA controller performs two separate transactions on the bus, a read followed by a write. In fly-past transfer the DMA controller initiates both read and write requests simultaneously and the data is transferred directly from source device to destination device. The fly-past mechanism is faster at transferring data, but requires extra decode logic and cannot be used for memory to memory transfers in the same memory device. This DMA controller can support either of these modes.

- Command chaining

  A DMA channel can be chained to the next channel. When a transfer sequence is complete on the first channel register values from the second

channel will be copied to the first and the DMA controller can resume, allowing continuity in transfer for a specific device.

- Search capability
  Additional data transfer functionality — such as "on-the-fly" comparisons — can be added with a little extra resource.

1.3 Summary

A DMA controller can be a useful enhancement to a microprocessor system, relieving the CPU of considerable load and moving data faster and more efficiently than the CPU could alone. This has been exploited commercially for a long time and some standard characteristics of DMA controllers have emerged.

The common features in these DMA controllers are:

- Transfers between a peripheral and memory and between memory devices (even though this needs two channels in an 8237).
- Multiple-channels
- Several data sizes to support different data width devices.
- Interrupt request to the processor when transfer is finished.

These generalised functions of DMA controllers are used as guidelines in the design of the asynchronous DMA controller in this thesis.

Chapter 2

Asynchronous Logic Design

2.1 Introduction

In the field of digital circuit/logic design, two different design styles have been developed. These design styles, synchronous and asynchronous, were used almost equally at the beginning of digital logic design era, but synchronous logic design has been more favoured in the last few decades and has become the accepted design style for complex circuits.

These two design styles differ from each other in the timing assumptions that are made. Synchronous logic assumes that time is discrete; each part of the system has to start and finish its tasks in the same period of time, whereas asynchronous logic does not make any assumptions about timing and each part starts/stops working on its own.

In the synchronous logic design style, a clock, a periodic square wave signal, is used for global synchronization to specify when to start and finish tasks. Communication between parts of the system must be done before the end of a clock period when the whole system is synchronized. Synchronous logic design has been favoured by the chip design community for several decades for several reasons [19]:

- It offers a simple way to design and test computing equipment

- It is widely taught and understood
- Parts that operate with clocks are widely available
- System noise such as switching surges or glitches has died away by the time a clock event occurs

Simplicity in timing models makes synchronous logic design easier, especially when the circuits are not very complex or no other constraints such as power consumption, performance or electro-magnetic interference need to be considered. Because of this simplicity development in circuit design has long been concentrated on the synchronous design style and asynchronous design style has been virtually ignored.

In contrast to synchronous logic design, asynchronous logic is:

- Harder to design
- Not commonly taught or understood
- Lacking components in standard libraries
- Short of development tools
- Higher in overheads (for simple circuits)

While synchronous logic uses a global signal for synchronization asynchronous logic allows circuits that need to communicate to perform a local synchronization between themselves.

As circuits grow bigger and become more complex because of the need for higher performance and functionality, several problems caused by the timing assumption in synchronous logic design become more obvious. These problems are:

- Clock skew

  As the chip size increases, and higher clock speed is needed for employed

higher performance, the phase difference between clocks in each part of circuit becomes great enough to cause problems.

- High power consumption

  With global synchronization parts of the circuit that are not performing any function at that time must be activated and consume power without any useful result. This worsens the power dissipation problem in some applications.

- Worst case performance

  Since sub-circuits which can work at high speed must be slowed down to work with the slowest one, the cycle time of the overall circuit is that of the slowest circuit instead of the average case.

- Design transfer to new technology

  In a design with global synchronization a circuit design for one technology could not be used with a different technology without major adaptation.

- Non-modular design

  A complex circuit designed for a specific clock speed may not be reusable with a different clock speed; redesign or major adaptation may be required.

One solution to these problems is to give up this timing assumption. Several research groups have been successful in using asynchronous logic design to implement asynchronous circuits at different levels of design complexity. Several circuits as complex as a microprocessor have been successfully implemented using different styles of asynchronous logic design [16, 15, 20].

2.2 Asynchronous logic design

The design methodologies for asynchronous logic could be categorized into two models: bounded delay and unbounded delay models. A bounded delay model

assumes that delay of a circuit element or wire is known within certain limits; the opposite is assumed in the unbounded delay model.

The circuits implemented by using these delay models can be classified into:

- Timed circuits

  Correct operation of circuits of this class may be dependent on the delays in circuits elements and wires.

- Delay-insensitive circuits

  Correct operation of the circuit is independent of the delays in both circuit elements and wires.

- Speed-independent circuits

  Correct operation of the circuit is independent of delays in circuit elements and wire delays are assumed to be zero.

- Quasi-delay-insensitive circuits

  This circuit is delay-insensitive with an 'isochronic forks' assumption. Isochronic forks are sets of interconnecting wires where the delay difference between the branches is zero or negligible compared to the circuit element delays.

Data passing between two circuit components in asynchronous system could be encoded in a single or a pair of wires, known respectively as single-rail and dual-rail encoding. In single-rail encoding one wire is needed for each bit of information. In the dual-rail encoding two wires are required for every bit of information: one wire is used to represent logic '1', another is used for represent logic '0'. A signal transition or prespecified level on one of this pair of wires represents the data bit value for each communication. Note that the active condition must not occur in both wires (from a pair) in the same communication because it would represent both logic '0' and '1' for one data bit at the same time, which is invalid.

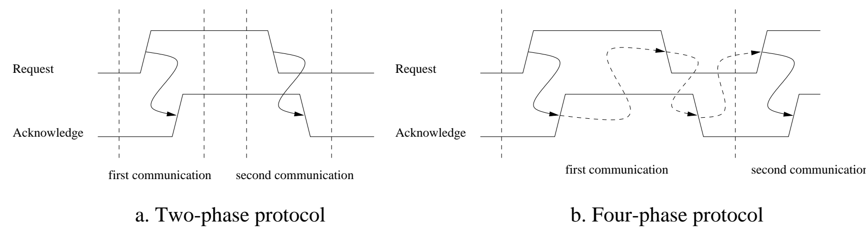

Figure 2.1: Signalling protocols

When using dual-rail encoding it is possible for the receiver to determine the validity of the data sent from the sender (entire data word is sent) by detecting transitions in one of the wires of each wire-pair. Three of the four possible states of a wire-pair are legal: in a level sensitive implementation '00' indicates that the bit is idle, '01' and '10' indicate data zero and one respectively and '11' is an invalid state which is never seen. By ORing together the two signals it can be determined whether or not a data bit is being sent. By ANDing (or, more commonly, using a C-element tree) each of these ORed bits across a bundle a single request signal, which indicates the validity of the data on the whole bundle, can be generated. To acknowledge a communication a single acknowledge wire is used for the whole bundle.

With single-rail encoding, explicit timing information is required for the data transfer. Data word is "bundled" with 'request' and 'acknowledge' signals which are used as timing information for the data communication. Data can be "pushed" from the sender to the receiver by the sender setting up the data and initiating a 'request' to the receiver; the sender must keep the data value stable until it receives an 'acknowledge' signal from the receiver which means data has been received. Alternatively data may be "pulled" from the sender to the receiver by the receiver initiating a 'request'; the sender sets up the data and 'acknowledge's when the data is ready.

Signal transitions in the data or the request/acknowledge signal pair of wires may be two-phase or four-phase. As shown in figure 2.1.

In a two-phase signalling protocol a condition is signalled by a single transition

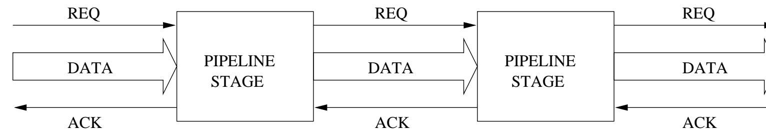

Figure 2.2: Structure of a simple micropipelines

of signal as shown in figure 2.1 a. The sender initiates the communication by making a single transition on the request wire; the receiver responds by making a single transition on the acknowledge wire completing the two phases of the communication. Rising and falling transitions are equivalent in the two-phase protocol.

In a four-phase signalling protocol a condition is signalled by a level (i.e. two transitions). The sender initiates the communication by making a transition from low to high on the request wire; the receiver responds by the same transition on the acknowledge wire. The second 'return to zero' pair of transition on both wires contains no important meaning in communication but is treated as recovery state that is used to set the signal to their idle states, as shown in figure 2.1 b.

Several asynchronous logic design styles have been developed and used. A number of these styles have been used to design and implement complex circuits [11].

2.3 Micropipelines

The Micropipeline technique [19], with some modifications in timing and signalling protocols, is used in the design of the AMULET processors. The Micropipeline design style, proposed by Ivan E. Sutherland, could be described as event-driven, elastic pipelining (i.e. a pipeline in which the amount of data can vary). It has been used and proved to be successful as framework for asynchronous logic design and can be classified as bounded-delay, bundled data for the datapath with delay insensitive control [16]. The structure of a simple micropipeline is shown in figure 2.2.

A Micropipeline uses a simple data bundle to transfer data between pipeline stages. Each stage is composed of storage and processing units; communication between stages uses a request and acknowledge signal pair. Data validity is indicated by the handshake signals [19].

2.4 The AMULET Processors

The AMULET group research interest is in low power circuit design. Since the asynchronous logic design has potential for low power (among other things) when compared with synchronous logic, it has been used by the AMULET group to design the AMULET processors.

The AMULET3i is an asynchronous microprocessor subsystem developed by the AMULET group at the University of Manchester [8, 10, 13]. Previous research by this group also included two asynchronous processors, the AMULET1, and AMULET2e [9, 16].

AMULET1, the first asynchronous processor developed by this group, was a feasibility study in using asynchronous logic to build a very complex circuit. The commercial, 32-bit, ARM RISC architecture [12] was chosen mainly because it is small, simple, and a low power design; in addition some AMULET group members had familiarity with this processor architecture and support for developing the processor was available [7].

AMULET1 was comparable in functionality to the synchronous ARM6 processor which was built on the same process technology. Even though the AMULET1 was no better in both performance or power-efficiency when compared with ARM6 it met its primary goal, to show the feasibility of designing very complex asynchronous circuits [7, 9, 16].

With successful results from the design of the AMULET1 the AMULET group continued working on the second asynchronous processor, the AMULET2e. Because of experiences of interfacing problems between the AMULET1 processor

and other chips at board level, to gain performance and to reduce power consumption, the AMULET2e was designed to be an asynchronous embedded microcontroller. It incorporated the AMULET2 processor core, on-chip memory and an external memory interface. This memory interface can be used with ROM, RAM, or peripheral chips.

Several new features were introduced to the AMULET2 processor core, e.g. register forwarding, a branch target cache, but the most useful is the 'halt' feature which made the power consumption drop to near zero when the processor was running in an idle loop. Both performance and power-efficiency of the AMULET2e were competitive with ARM710 and ARM810, the contemporary synchronous processors. The AMULET2e was a highly usable asynchronous embedded system chip [9].

The AMULET3i is the latest design. It is suitable for commercial embedded applications. To gain performance, reduce power consumption and make asynchronous logic more useful in a system the AMULET3i subsystem incorporates the AMULET3 processor core, on-chip RAM, ROM, an asynchronous bus, external memory interface, synchronous bridge, test interface and a DMA controller. The AMULET3i subsystem will be discussed in more detail in the next chapter.

2.5 Summary

Asynchronous logic design has been used to solve problems that exist in a very complex circuit such as a processor subsystem. Micropipelines, one of specific asynchronous design style, have been adapted to be a successful method in which to implement three AMULET processors. The AMULET3i processor should achieve commercial viability; the DMA controller is one important component in this subsystem which will improve the system in handling data transfer more effectively.

Chapter 3

The AMULET3i Subsystem

3.1 Introduction

The AMULET3i, an asynchronous processor subsystem, is a set of macro-cells which is composed of the AMULET3 processor core and several other asynchronous components. It is designed as a part of larger chip which is intended to be used as a base unit in the low power applications.

The main components in the AMULET3i subsystem (figure 3.1) include :

- Processor-Memory Subsystem, which include :
  - AMULET3 Processor Core
  - Local RAM
  - Processor Local Bus
- ROM
- MARBLE Bus (see section 3.2)
- DMA Controller
- Peripheral Devices
- External Memory Interface

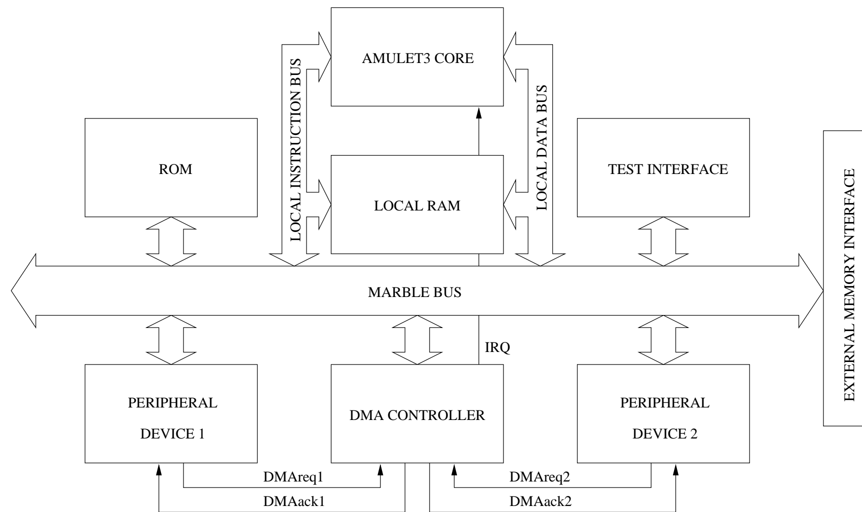

Figure 3.1: The AMULET3i Subsystem

- Test Interface Controller

3.2 The MARBLE Bus

The MARBLE bus [1] is an asynchronous on-chip bus for connecting macro-cells. It supports high speed data transfer, atomic transactions and multiple initiators with central arbitration and address decoding.

Most communication between devices in the AMULET3i subsystem can be performed across the MARBLE bus. (There are some exceptions such as processor interrupt (IRQ) or peripheral request (DRQ) which are passed directly between devices). A data transfer from a one device to another device is done in one cycle via the MARBLE device interfaces.

The MARBLE bus has two types of interface : an initiator interface and a target interface. Each interface presents one address bundle and two unidirectional data bundles to its subsystem. A single data bundle from the MARBLE bus is forked to input/output data bundles on the device interfaces. The data bundle consist of a 32-bit wide data bus and a request/acknowledge signal pair. The address bundle consist of the 32-bit wide address bus and other command

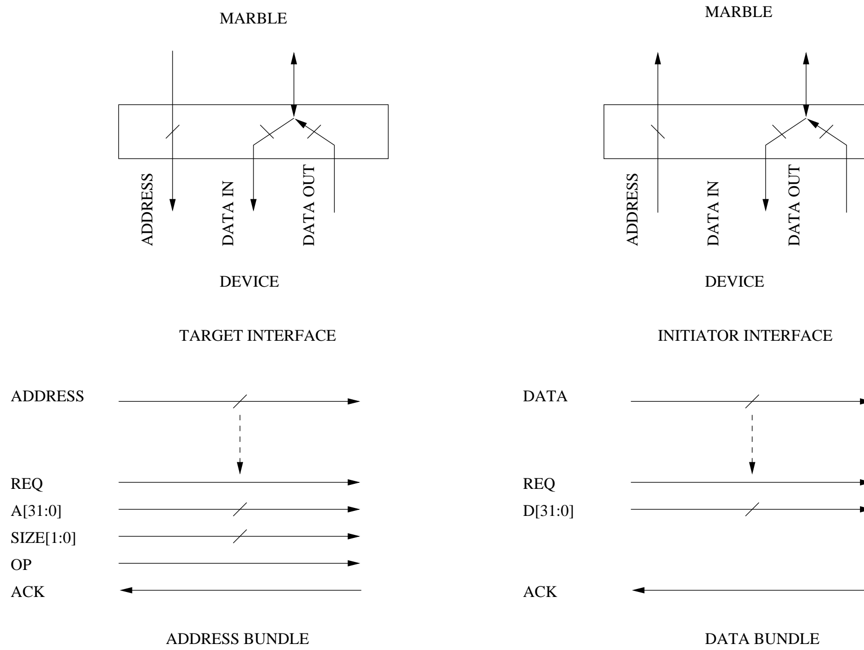

Figure 3.2: The MARBLE interfaces

buses such as data size and direction (read/write) of data transfer, as shown in figure 3.2.

3.2.1 Initiator Interface

The initiator interface is used by the initiator device to read or write data from or to the target device across the bus. The initiator device set up target device's address, size of data, type of the operation (read or write) on the initiator interface and sends them to MARBLE using request/acknowledge signals. The MARBLE bus receives address/command and passes it to the target device via target interface (see next section) of that device.

In a write operation data is sent across the bus from the initiator device to the target device. The data item is sent independently from the address. Data can be sent before, after, or simultaneously with the address to the bus. However the corresponding address and data of the same transfer must be sent before next transfer may be started, data and address of different transfer cannot not overlapped in sending/receiving.

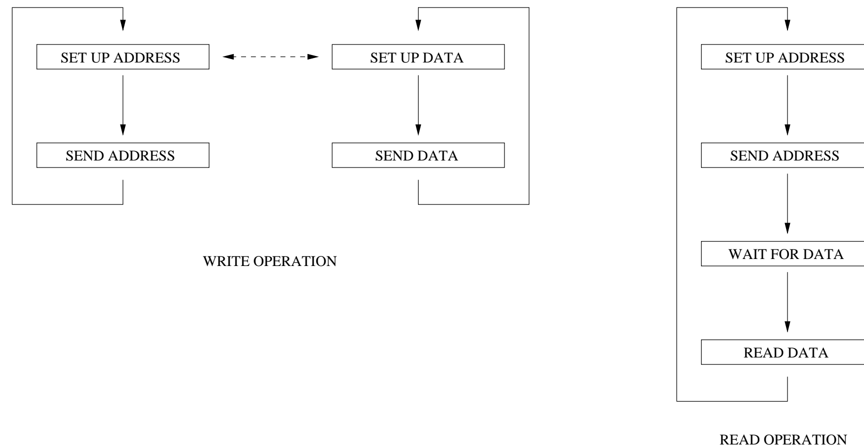

Figure 3.3: Initiator interface data transfer operations

In a read operation, a data item is transferred from the target device to the initiator device across the bus. The initiator device send address to the target device. Upon receives the address the target device can then determine which data should be send back to the initiator device.

A diagram of data read/write operations performed by the initiator device is shown in figure 3.3.

3.2.2 Target Interface

The target interface is used by the target device to receive addresses and read or write data to or from the initiator device across the bus. The target device determines the operation (read or write) from address/command sent by the initiator device before performing the corresponding operation.

In a write operation the target device waits for data from the initiator; in a read operation a data item is returned from the target device to the initiator device.

A diagram of data read/write operations performed by a target device is shown in figure 3.4.

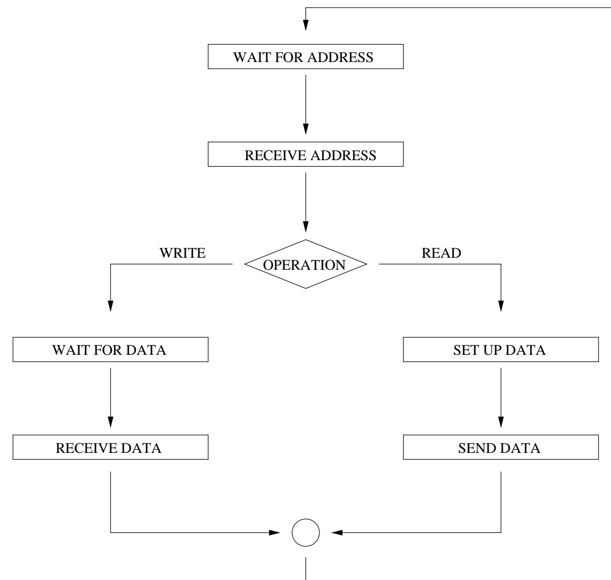

Figure 3.4: Target interface data transfer operations

3.2.3 Type of data transfer

The bus supports data transfers between an initiator and a target device. In DMA, transfers between two target devices must be achieved. There are two general methods of approaching this:

- Store-and-Forward

  Data transfer between two different target devices is controlled by an initiator device. The transfer is done in two cycles : a read-cycle followed by a write-cycle. In the read-cycle, the initiator reads data from the source device; in the write-cycle, the data is written from the initiator to the destination device.

- Fly-past

  Data transfer between two different target devices is transferred in one cycle. In this type of transfer the read request to the source device and the write request to the destination device are initiated simultaneously. The data item is transferred directly from the source device to the destination device in one cycle.

Fly-past transfer is more efficient than store-and-forward transfer, however it cannot be used for transferring data between two different regions in the same device; e.g. memory to memory transfer on the same memory device could not done by this method. The current implementation of the MARBLE bus supports only store-and-forward data transfers.

3.3 Processor-Memory Subsystem

Unlike the other components of AMULET3i which are linked by the MARBLE bus, the processor and local RAM form a close-coupled group with local buses. Separating the processor core and local RAM from other parts of the system allows the processor to access local RAM more effectively, since it avoids the impediment of the full, multi-master bus control. These local buses also reduce traffic on MARBLE since instructions and data which the processor use frequently can be stored in local RAM, leaving MARBLE free to be used by other devices.

There is some penalty when the processor needs to access other parts of memory outside the local RAM or other devices on MARBLE, however some of this penalty can be overcome by using the DMA controller to transfer data from that part of memory to local RAM.

The processor-memory subsystem connects to MARBLE as a single component with two initiator interfaces (one for the instruction bus and another one for the data bus) and one target interface. The initiator interfaces allow the processor access to instructions/data from other devices on MARBLE while the target interface allows other initiator devices access to data in the local RAM.

3.3.1 The AMULET3 Processor Core

The AMULET3 processor core is implemented using ARM architecture version 4T [8]. It is functionally compatible with the ARM8 processor. Although to get the best performance the binary code could be compiled and optimized specifically

for AMULET3.

One feature of the ARM architecture which does not exist in many RISC processors are instructions to load/store multiple register values to/from memory. These instructions allow the processor to transfer several data items between device and registers with higher performance than simple load/store instructions. However, these instructions are still not applicable for transferring large amounts of data between devices, which is a functional performed more efficiently by the DMA controller.

3.3.2 Local RAM

The local RAM on AMULET3i subsystem is an 8KB static memory which is divided into smaller memory blocks. The fragmenting of the local RAM into interleaved blocks means that different blocks can be accessed simultaneously on the two local buses without the expense of dual-port RAM.

In theory the local RAM could be made to work as a cache, but because of limited development time this feature is not realized in the first implementation of AMULET3 which has simply a directly memory mapped local RAM.

3.3.3 The Processor Local Buses

The processor local buses separate instruction and data traffic into two separate buses. The local buses allow high bandwidth memory accesses and simplify the memory-address interface. The instruction bus is connected to MARBLE via an initiator interface which is used by the processor when it fetches instructions from an external memory device. The local data bus, however, is more complex; it can be used by the processor to access data on other devices on MARBLE and by the initiator devices on MARBLE to access data on the local RAM. Therefor the local data bus is connected to MARBLE by both initiator and target interfaces. The DMA controller can access data in the local RAM via this target interface on the local data bus.

3.4 Other Devices

3.4.1 On-chip ROM

The AMULET3i ROM is an 16KB on-chip ROM connected to the MARBLE bus via a target interface. It provides application software and routines for testing the AMULET3i subsystem. Routines for testing and initializing the DMA controller could be stored on this ROM and be executed after system power up.

3.4.2 External Memory Interface

The external memory interface allows the AMULET3i subsystem to connect with off-chip devices, so conventional synchronous devices could be used via this interface. The interface supports direct connection of external memory and peripheral devices with 8/16 bits data width. The timing of the accesses to these off-chip devices is defined by a timing reference delay, which is only activated when an off-chip access is required. The timing characteristics and bus width are programmable separately for different memory regions. The DMA controller could be used to perform DMA transfers with these devices, however peripheral device connected via this interface could only be treated as memory device, because a peripheral request signal (DRQ) is not supported by this interface.

3.4.3 The Asynchronous Peripheral Devices

At the time that this thesis is written, no real asynchronous peripheral devices are planned for inclusion in the AMULET3i subsystem. However, the behaviour and interface of any such devices must conform to the following specifications to work with the system and the DMA controller properly.

- Interface to MARBLE via target interface
- One or more fixed address in memory address space

- Data transfer size can be 8/16/32 bits
- Initiate data transfer by DRQ signal when it is ready to transmit or receive data

The device address is fixed and predefined, each device has a specific DRQ signal pair.

3.4.4 Test Interface Controller

The test interface controller provides an external interface to a MARBLE initiator. This allows external access to all MARBLE targets for test or debug purposes. The interface is designed to suit conventional VLSI production test equipment and it therefore uses a clocked protocol.

3.5 The DMA Controller

The DMA controller is another device on the MARBLE bus, it is both an initiator device and target device at the same time. Design of the DMA controller will be discussed in the next chapter.

Chapter 4

Top Level Design

4.1 Introduction

The AMULET3i DMA controller is designed for a general MARBLE based system. The framework of the design are mostly taken from the requirements and its environments. Some are influenced by the design of existing DMA controllers. This chapter discusses the top level design of the DMA controller.

4.2 Specifications

The DMA controller specifications are:

- Four independent programmable channels

  The number of channels is arbitrarily chosen, but is expandable without significant change in the structure of design. Also four DMA channels are used in the design of many existing DMA controllers, e.g. Intel 8237 DMA controller and National Semiconductor NS32230.

- 8/16/32 bit transfer sizes

  As required by devices on the subsystem, these transfer sizes are supported by the MARBLE bus and allow DMA transfers with appropriate data size for specific devices.

- Data transfer can be performed between:
  - Peripherals and memory
  - Memory and memory
  - Peripherals and others peripherals
- Store-and-forward transfer

  This method of data transfer is the only one supported by the MARBLE bus; fly-past transfers are not supported. These two types of transfer were described in chapter 3.
- Interrupt enable, configurable for each channel

  The IRQ signal can be used to notify the processor when data transfer for a specific channel is finished.
- Peripheral synchronization with DRQ signal

  The peripheral device initiates data transfer when it is ready to transmit/receive data by using DRQ signal. Devices may be mapped onto arbitrary channels, so more than four devices can use DMA, but only four devices can be active at a given time.

4.3 Terms Definition

- DMA Channel

  The DMA controller is designed to support concurrent transfer of more than one pair of devices. Data transfer between a pair of deivces is assigned to a DMA channel. After being configured that DMA channel will handle the transfer of a given device pair until finished.

  Even though only a single data item can be transfered at a time since there is one data bus, but a datum of a DMA channel can interleave with other DMA channel so concurrent transfer can be performed.

- Channel Registers

  To support multiple DMA channel, a set of registers for each DMA channel is required to store state of data transfer. This set of registers is called Channel Registers.

- Devices : Memory and Peripheral

  The DMA controller supports two types of device: a memory device and a peripheral device. Memory device is a storage device; an address is used to store a fixed size data value, synchronization is not required to access data. Peripheral device is a channel to an external device; an address is assigned as a port to communicate with that device, synchronization to access data maybe required.

- Peripheral Request Signal (DRQ)

  DRQ (Data transfer ReQuest signal) is a signal used by 'some' peripheral device for synchronize with the DMA controller for transfer a data item.

- Channel Request Mapping Block

  To allow the DMA controller to expandable in number of peripheral device that can be used for DMA transfer. An peripheral device is not fixed to a DMA channel. Each DMA channel can be programmed to perform DMA transfer with any device. The Channel Request Mapping block is used to provide this feature to the DMA controller. It maps a DRQ signal to any DMA channel as configured by the channel registers.

- Channel Transfer Request Signal (CHANREQ)

  CHANREQ is an internal signal of the DMA controller uses by the control unit in each channel to send request to the transfer engine. Since the is only one transfer engine to service all DMA channel the CHANREQ is used to indicate that the devices pair is ready for transfer a data item.

4.4 DMA operation overview

When a DMA channel is enabled, conditions of devices are used to activate the DMA controller to perform data transfer. Both devices must be 'ready' before data transfer can be performed. For the memory device it is assumed to be always ready; for some peripheral device the ready state is indicated by peripheral request signal (DRQ) which sent from device to DMA controller.

The DMA controller uses single datum transfer basis. After a data item is transfered the devices of that DMA channel need to re-assert 'ready' state for the next data item to be transfered. Data transfer of a channel can be interleaved with other channel.

Details of the DMA transfer operation are discussed in the next chapter.

4.5 The DMA controller internal structure

In the DMA controller, the internal structure can be divided into two major units: the register unit and the transfer engine unit, as shown in figure 4.1.

Each unit has its own interface to MARBLE. The register unit contains a target interface which allows the DMA controller to be programmed by the processor. The transfer engine unit uses an initiator interface to transfer data between source and destination devices.

The transfer engine reads/writes data from/to the register unit via an internal bus. The register unit not only contains DMA registers and handles register access from the transfer engine and processor but also handles the interrupt signal (IRQ) and initiates requests for data transfer to the transfer engine with CHANREQ signals when a DMA device is ready.

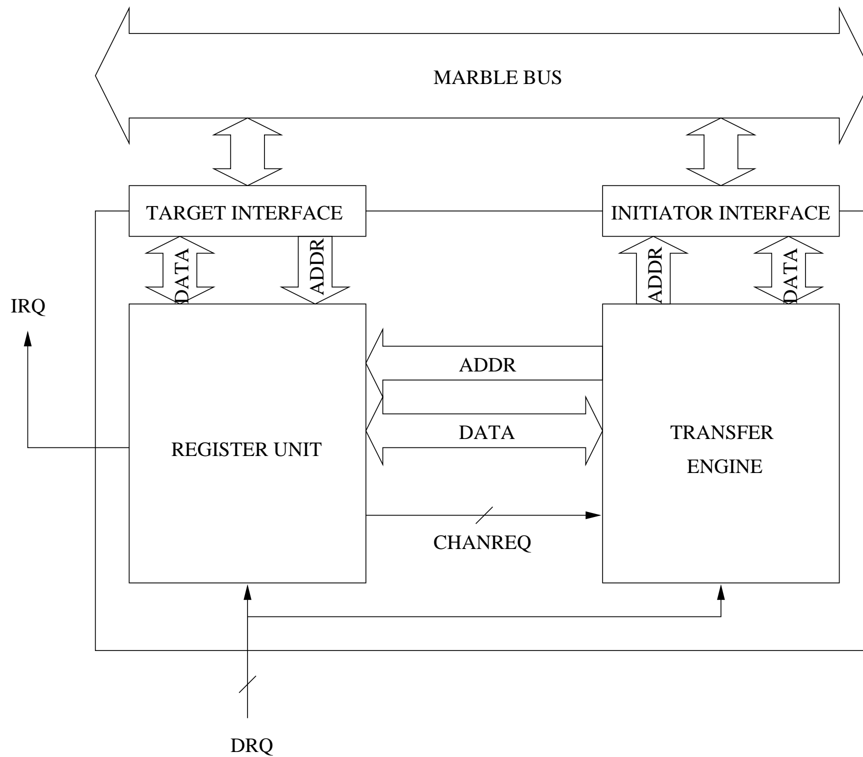

Figure 4.1: Block diagram of the DMA controller

4.6 The Register Unit

The register unit contains the DMA registers and other control circuits for handling register access, IRQ, DRQ and CHANREQ signals. The registers are divided into two groups: global registers and channel registers. The global registers are used to control the interrupt request signal (IRQ); the channel registers are used to keep the states of the transfer operation for each channel. There are three groups of control circuits in the register unit: one is used to control access requests from the processor (via the bus interface) and the transfer engine, one to control the IRQ signal and one to map the peripheral request signals (DRQ) and enable state (set in the control registers) onto channel requests (CHANREQ) to the transfer engine. A diagram of the register unit is shown in figure 4.2.

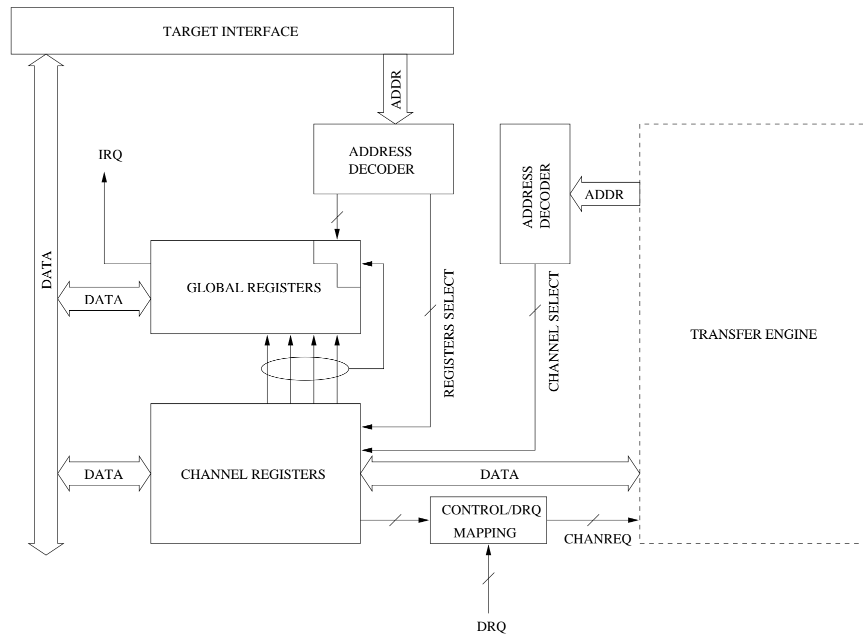

Figure 4.2: Register Unit

4.6.1 The Global Register Unit

The global registers comprise:

- ChanStatus
- IRQMask
- IRQRequest

registers. Access from the processor to one of these registers is independent from any access to channel registers. These registers are used for interrupt control and generate the interrupt request (IRQ) signal to the processor. The IRQMask register is read/writable by the processor but the other two registers are read-only. Each bit in the IRQMask register corresponds to a DMA channel which is used to specify whether, when the data transfer sequence for that channel has finished, the DMA controller needs to interrupt the processor or not. Each bit in the ChanStatus register also corresponds to a DMA channel which is set by the channel registers when the transfer for that channel is finished. All bits in

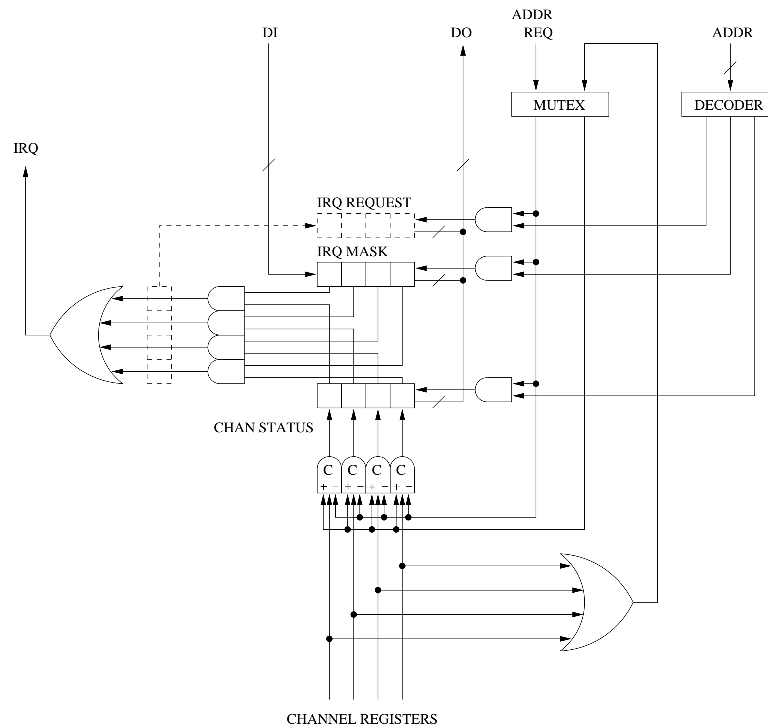

Figure 4.3: Global Registers

ChanStatus are reset when the processor performs a data read from ChanStatus or IRQRequest register. The IRQRequest register is not an actual register but is an AND operation between the ChanStatus and IRQMask register, this register provides a convenient mechanism for the processor to determine which channel(s) caused the interrupt request.

The interrupt control circuit generates the interrupt request signal to the processor when the AND operation between IRQMask and ChanStatus registers has a non-zero value. The IRQ signal may change when either IRQMask or ChanStatus register value is changed; if the AND operation returns a zero value then the IRQ signal is deasseted.

A diagram of the global registers is shown in figure 4.3.

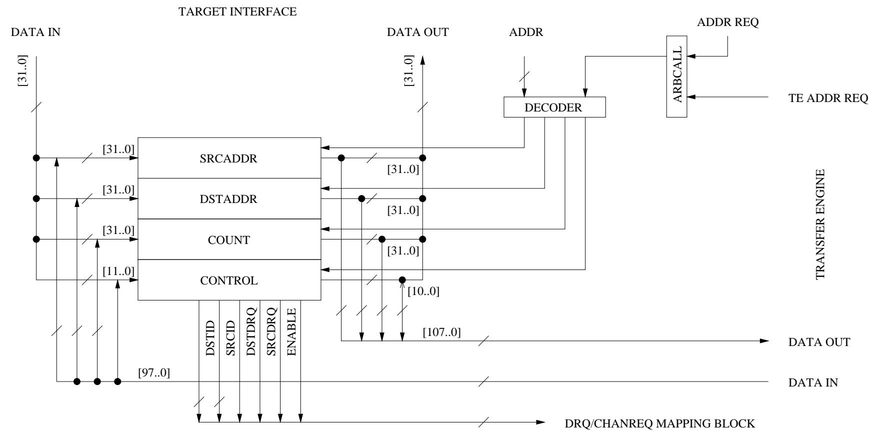

Figure 4.4: Channel Registers

4.6.2 The Channel Registers

Each channel register is independent from the other channel registers and the global registers. The processor and the transfer engine can access registers for different channels simultaneously.

Each set of channel registers comprises:

- SrcAddr
- DstAddr
- Count
- Control

registers as shown in figure 4.4.

Every register is read/writable by the processor and the transfer engine. The processor can access one register at a time, via the bus interface; this is limited by the size of the bus. The transfer engine uses a wider bus for access to the channel registers allowing the transfer engine to access all of a channel's registers at once.

The SrcAddr and DstAddr are 32-bit address registers for keeping source and destination addresses respectively; this allows the DMA controller to perform data transfers across the full range of the address space. The counter is also 32

bits wide so the DMA controller can transfer up to 2^{32} data items in one DMA transfer sequence.

The control register is used to control the DMA transfer operation. The control functions in the the control register include:

- Enable/Disable
- Source Address increment
- Destination Address increment
- Counter decrement
- Source device uses DRQ
- Destination device uses DRQ
- Source device ID
- Destination device ID
- Data size

The ‘device uses DRQ’ and ‘device ID’ bits in the control register are used to map the peripheral requests signal for generating transfer requests to a transfer engine channel. A memory device (which is always ready) does not use DRQ.

The control bit configurations in the control register is shown in figure 4.5. The channel request mapping unit and channel mapping block are shown in figure 4.6 and 4.7.

4.7 The Transfer Engine Unit

The transfer engine unit is composed of the channel arbiter unit and the transfer control unit. The transfer engine unit, when idle, waits for requests to transfer from DMA channels in the register unit and performs data transfers for a selected

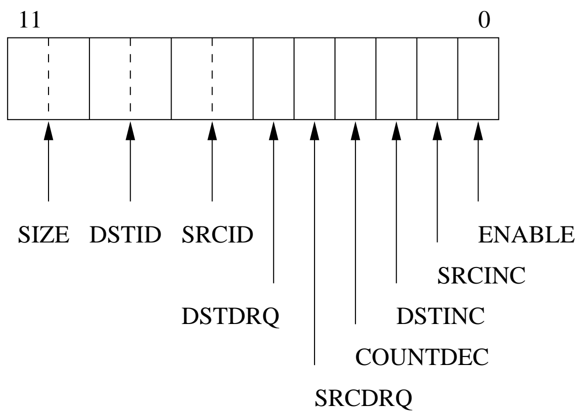

Figure 4.5: Control Register

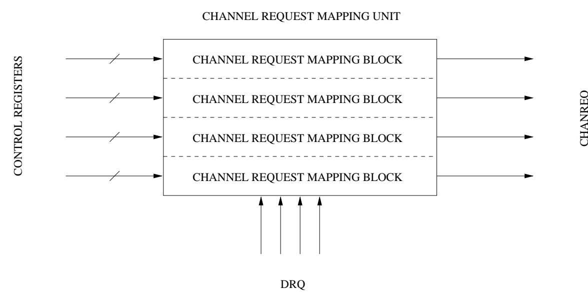

Figure 4.6: Channel Request Mapping Unit

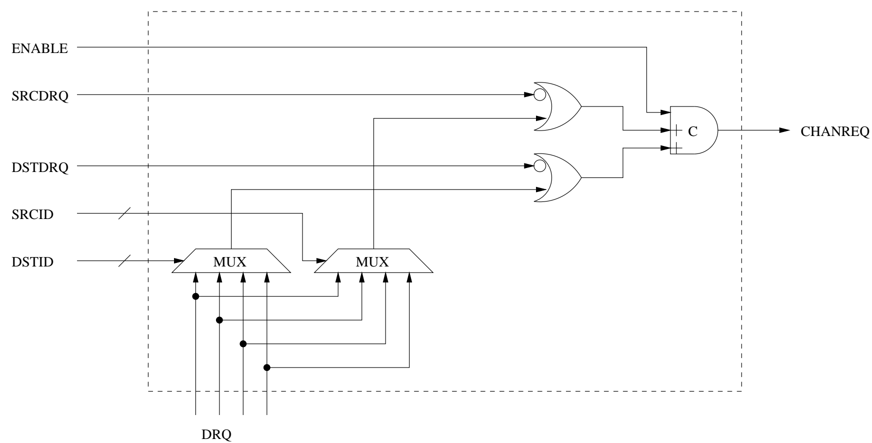

Figure 4.7: Internal structure of Channel Request Mapping Block

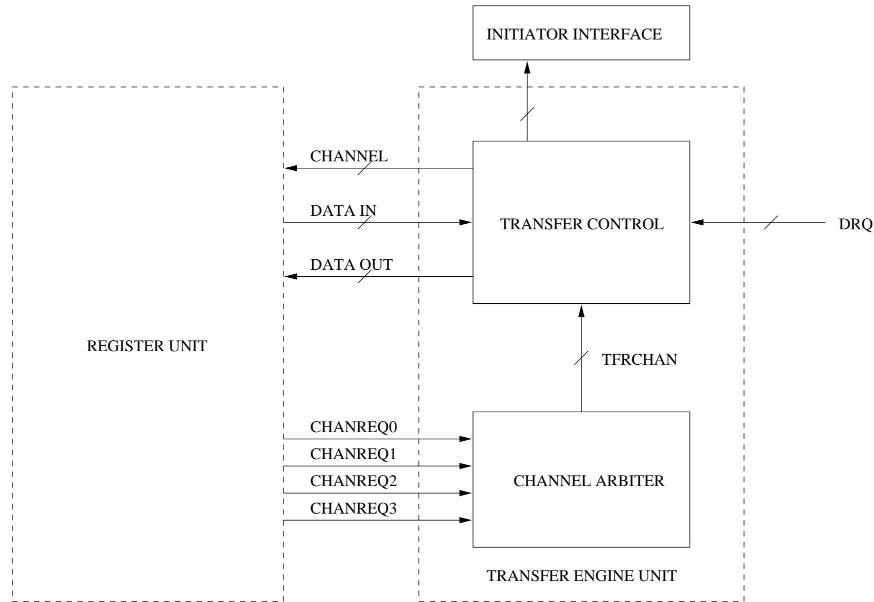

Figure 4.8: Transfer Engine Unit

channel by initiating a data transfer to the bus. Data items are transferred singly. Each data item transfer with the peripheral device may needs synchronization which will be done by using DRQs request/acknowledge signals pair. When a transfer is finished the transfer engine becomes idle again and waits for the next request.

A diagram of internal structure of the transfer engine is shown in figure 4.8.

4.7.1 The Channel Arbiter Unit

It is possible that more than one DMA transfer is requested at the same time. The channel arbiter unit arbitrates the transfer request signals sent from the channel registers in the register unit. It sends the selected channel number to the transfer control unit to perform a data transfer.

The channel arbiter uses a tree of arbiter-call elements to choose a channel from multiple channel requests. Two types of arbiter tree can be used: a balanced tree, or an unbalanced tree. Details of the arbiter call elements and arbiter tree will be discussed in next chapter.

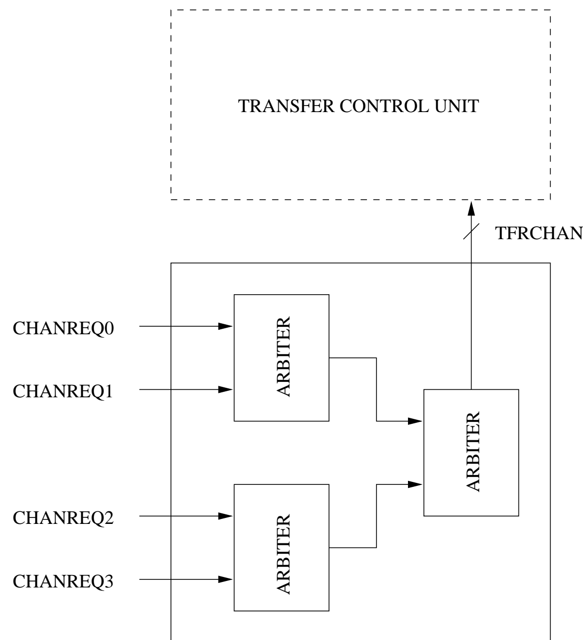

Figure 4.9: Channel Arbiter Unit

A diagram of the channel arbiter is shown in figure 4.9.

4.7.2 The Transfer Control Unit

The transfer control unit controls the actual data transfer between the source device and the destination device. It needs to perform four separate functions :

1. Synchronization with the peripheral device using the DRQ signal as required (controlled by 'UseDRQ' and 'DeviceID' bits in the control register).
2. Read/write registers from/to the register unit.
3. Initiate data read/write transfer with the bus.
4. Increment/decrement and write back registers.

When it receives an arbitrated channel number from the channel arbiter, the transfer control unit reads the selected channel's registers from the register unit.

Control register bits are used to determine data size, requirements for synchronization with peripheral devices during data transfer, source and destination address increment, and counter decrement. The transfer control unit then performs a data read from the source device by initiating an address request across MARBLE; this is followed by the corresponding write cycle. While performing data transfer with the device DRQ signal is synchronised by the transfer engine if synchronization is required. Concurrently the transfer control values are updated and written back to the channel registers. When the operation is finished it waits for next transfer request from the channel arbiter.

4.8 Summary

From the top level design the DMA controller is functionally divided into smaller parts. The register unit contains all the registers and handles requests for transferring data for each DMA channel. It also controls any interrupt requests to the processor. The transfer engine performs the actual data transfers when it receives requests from the register unit. DMA operations are performed with tightly coupled cooperation between these two major parts of the DMA controller.

Detailed design issues and problems will be discussed in the next chapter.

Chapter 5

Design Issues

5.1 Introduction

This DMA controller is designed (see section Specifications in previous chapter) to be a multi-channel DMA controller in which each channel supports transfers between any combination of memory and peripheral devices. There are several issues concerned in allowing the DMA controller to function correctly and efficiently as specified. This chapter discusses various important design issues, their problems, and the solutions chosen to solve these problems.

5.2 DMA Registers

To program the DMA controller a number of registers are made CPU visible. For each DMA channel the following values are needed:

- Transfer Source/Destination addresses
- Number of data items to transfer
- Type of source/destination devices (memory or peripheral)
- Data transfer size

32-bit registers are used to store these values for two main reasons:

- The processor word size is 32 bits, so a read/write operation with one DMA register can be performed in one access.
- The 32-bit address register allows the DMA controller to perform data transfers with any device in the address space and for lengths of the whole address space.

5.2.1 Transfer Control Configurations

The control configuration specifies whether the device's address should increment after the transfer and whether synchronization with a peripheral request signal (DRQ) is required. For a memory device an address increment is required and no synchronization is needed; for a peripheral device the address is fixed and synchronization is needed, in which case a device ID must be supplied so the transfer engine can synchronize with the right device. So 'source/destination address increment' and 'device uses synchronization' control bits are used instead of the 'type of devices'.

To allow a DMA channel to interact with a specific peripheral device the device number needs to be programmable and is used when peripheral synchronization is specified. 'Source/Destination ID' is used for this configuration.

The counter is used to control the number of data items to transfer, but it can also allow a 'free run' type of transfer in which the DMA controller performs data transfers indefinitely until it is stopped by the processor. A 'counter decrement/free run' control bit is used to tell the DMA controller to decrement the counter after each data transfer or just to ignore the value of the counter. An 'enable/disable' mechanism to start and stop the DMA transfer is required for the 'free run' type of transfer.

To reduce the number of registers and allow the processor to program a DMA channel more conveniently all transfer control configurations are stored as bits

field in a single register, called the control register. The bit field configuration in the control register is shown in figure 4.5.

5.2.2 Processor Interrupt Control Configuration

After the data transfer for a channel has finished a mechanism to signal the processor is necessary. An interrupt is used for this purpose. However an interrupt costs processor time in context switching to an interrupt service routine, so a mechanism to allow the processor to specify which channels should interrupt is also desirable.

The 'interrupt enable' control bit is used for this. This could be a bit field stored in the control register like other control bits; however if the processor needs to disable all interrupts from the DMA controller, it has to write to every control register, which is inconvenient. Instead, the 'interrupt enable' bits for all DMA channels are stored together in one register called 'IRQMask' which allows the processor to disable/enable all channels with one register access.

5.2.3 Registers Access Conflict

Both the processor and the transfer engine require access to the DMA register and, though unusual, it is possible that these accesses may be attempted simultaneously. For example the processor may be programming one DMA channel while another is transferring. As these accesses have no fixed timing relationship they must either use separate ports on the register file — necessitating a dual-port register file — or undergo arbitration to sequence them in a mutually exclusive fashion. A third option would be to separate the registers into separate area for each channel, but this would still not guarantee the absence of conflicts — for example a program might read the registers of an active channel to find how far a transfer has gone.

Conflicting accesses are relatively rare and not time critical, so the second option is preferred rather than the expense (in silicon area) of a large dual-port

register bank. Details of the arbitration performed will be discussed later.

5.3 DMA Operation

The DMA controller, after being programmed and enabled by the processor, can start performing data transfers when the source/destination devices are ready to transmit/receive data.

5.3.1 DMA Operation Overview

The DMA operation starts at the channel request mapping block in the register unit (figure 5.1). This sends a CHANREQ signal to the channel arbiter-call unit in the transfer engine when both source and destination devices are ready to transfer data.

In the (unlikely) case of two channels requesting at the same time the channel arbiter-call and selects one of the active requests and sends the chosen channel number to the transfer control unit. The transfer control unit uses the channel number to read the appropriate channel registers and performs a data transfer for that channel. It updates all corresponding registers with respect to the control bits in the control register and performs necessary functions to indicate the end of the data transfer if the last data item has been transferred.

The DMA operation can be divided into three sub-operations:

- Channel selection operation
- Data transfer operation
- End of run indication mechanism

5.3.2 Channel Selection

Each channel request mapping block works independently. If a DMA channel is enabled, and both source and destination devices for that channel are ready, a

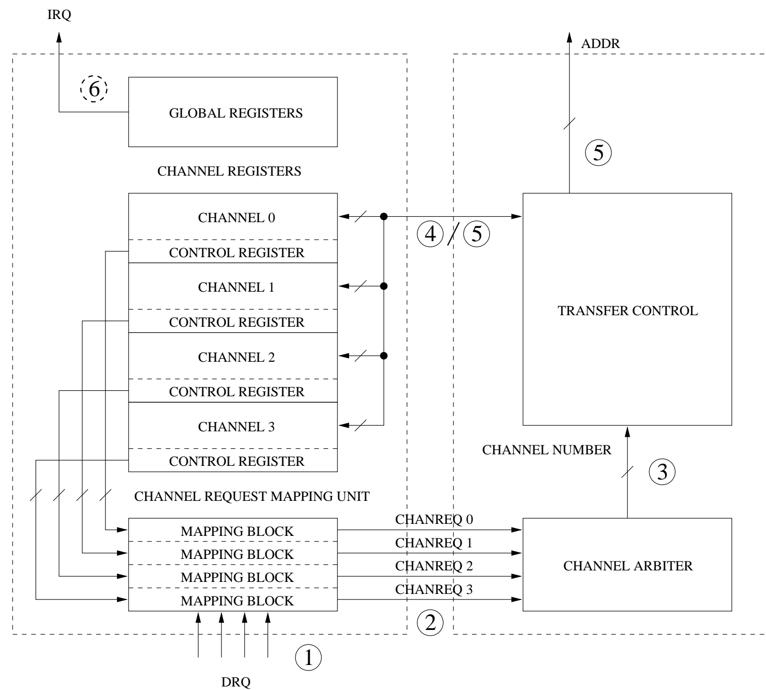

Figure 5.1: Sequence of the DMA operation

CHANREQ signal will be sent from the register unit to the channel arbiter-call unit in the transfer engine. If several requests occur arbitration is used to grant exclusive access to the single transfer engine to one at a time.

In existing synchronous DMA controllers both prioritized and non-prioritized schemes can be applied to select a requesting channel. However, the nature of asynchronous circuit is different; no global clock is used to judge as 'simultaneous' the arrival of events.

Unlike a synchronous system where the stability of signals is assured before an active clock edge an asynchronous system cannot sample incoming requests as there is no safe time period during which all the inputs remain stable. Instead an arbitration tree (discussed later in the section "Arbitration") can be used to select a channel.

5.3.3 Data Transfer Operation

In the read operation, the transfer control initiates a data read transfer from the source device. The source device address is taken from the SrcAddr register and data size is taken from the control register; these values are required to initiate the transfer. If the source device needs peripheral synchronization the transfer control also acknowledges it when data is received.

After getting the data from the source device, the transfer control unit performs the write operation sending the data to the destination device in the same manner as it performs the read operation although the destination address and the other destination device's configurations are used instead of the source device configurations; so data write transfer is performed with the destination device.

Concurrently with the read/write operations which the transfer control performs with the bus, the register update and write back operation can be performed with the register unit. Control bits in the control register are used to determine the necessity of specific register updating. The additional operation channel disable in which the enable bit in the control register is reset to stop data transfer for that channel is also performed if the counter value is decremented to zero.

5.3.4 Register Communication Models

Communication between the transfer control unit in the transfer engine and the channel's registers in the register unit could use different models with respect to size of data bus connecting them. This is different from communications between the DMA controller and MARBLE which is restricted by the MARBLE interfaces; only a 'narrow' (i.e. 32 bit) bus can be used.

Wide Bus Communication

In this model all registers in a channel are read/written at once. This allows register value transfer between these two internal units in the DMA controller

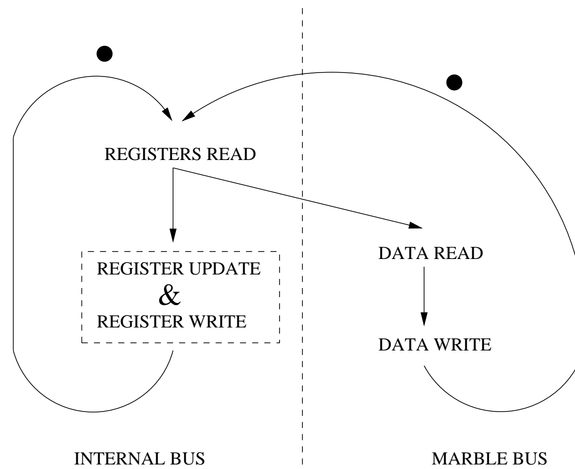

Figure 5.2: Register communication with wide bus

with high efficiency, although it also costs a number of wires and the transfer control needs extra storage to store the register value needed in the later sequence of the 'data write' operation.

A diagram of the register communication using wide internal bus is shown in figure 5.2.

Narrow Bus Communication

Here only one register can be read/written at a time. The transfer control unit starts the communication by reading the control register to determine the transfer configurations — initiating a data read from the source device requires 'data size', 'source device uses synchronization' and 'source device ID' control bits from the control register. The values of the control register are stored for later use.

The transfer control then reads the SrcAddr register and performs a 'data read' operation (with MARBLE); if a "source address increment after the data transfer" is required the 'data size' will be added to the source address and the new value written back to the register unit. The 'data read' operation and 'source address update/write-back' operation can be performed concurrently.

After the data is received from the source device the 'data write' operation is performed (after the transfer control unit has read the DstAddr register from

the register unit). Data is written to the destination device concurrently with DstAddr updating and write-back in the same manner with the read operation.

If the 'counter decrement' control bit is set the transfer control updates the counter register by reading the Count register, decrementing it and writing the new value back to the register unit. If the new counter value is zero, the channel disable is also performed when the Count register is written back. A diagram of data transfer operation using narrow internal bus width is shown in figure 5.3.

By comparing figures 5.2 and 5.3 it can be seen that the control operation when using a wide bus is much simpler than when using a narrow bus. Because the DMA registers and their processing unit will be in close proximity on-chip a large number of wires is not a particular handicap. Therefore in this design the wide bus is chosen.

The independence of the MARBLE bus read/write operations and register read/write operations could lead to a problem when the last data item is transferred. If a channel is set to interrupt the processor this could happen before the data transfer is finished, since the operation performed within the DMA controller (writing new register value to the register unit) will probably be faster than the data transfer performed with the bus. To solve this problem the register updating/writing operation for the last transfer must take place *after* the transfer operation is completed.

5.3.5 End of Run Indication Mechanism

The end of run indication is performed after the last data item of a DMA sequence is transferred. Since the enable bit in the DMA channel control register is used for sending the request for data transfer to the transfer engine this bit must be changed before the DMA operation is completed, otherwise the transfer engine may receive another request for transfer even though all data items have been transferred.

The end of run indication is performed when the value of a counter for the

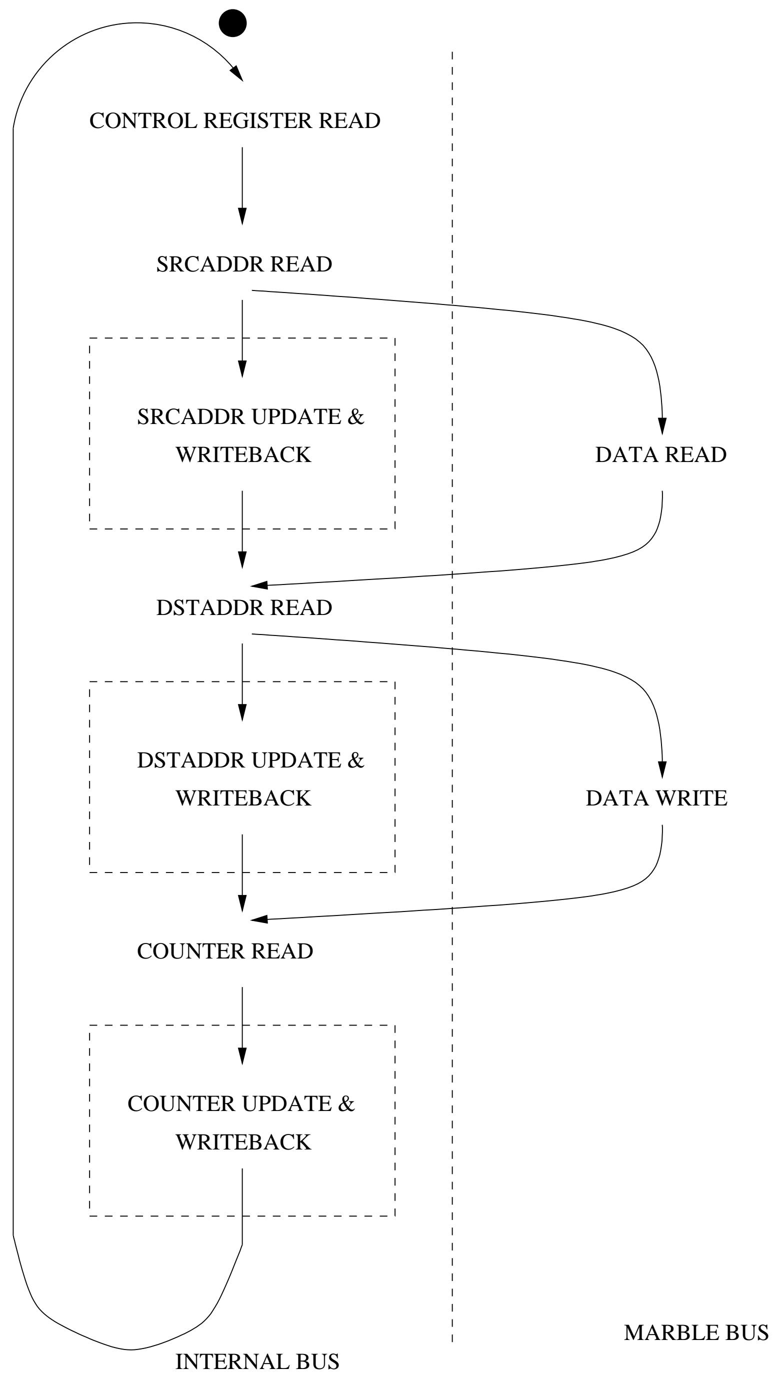

Figure 5.3: Register communication with narrow bus

current channel reaches zero. The transfer control unit clears the channel enable bit in the control register.

After the enable state in a channel's control register is changed to disable, the corresponding bit in the ChanStatus register is set by a request from the control unit in the channel register. The global register unit which contains ChanStatus and IRQMask registers and handles the interrupt request signal performs an AND operation between these two register values and raises the IRQ signal if the result of this AND operation is not zero. A diagram of the global registers and how they relate to the IRQ signal is shown in figure 4.3.

This mechanism simplifies the operation of handling the IRQ signal. When the processor reads from the ChanStatus or IRQRequest registers the ChanStatus value will be reset and the IRQ signal will be dropped. The processor writing to the control register (when programming a DMA channel) will cause the corresponding bit in the ChanStatus register to be reset.

5.4 Shared resources in the DMA controller

To perform DMA transfer functions the DMA controller needs to act as both an initiator and a target device. The DMA channel needs to be programmed by the processor before the DMA transfer operation can start. During this time the DMA controller is a target device which receives access requests from the processor. Subsequently, when the DMA controller performs a data transfer, it initiates access requests to source/destination devices to transfer data.

In both cases the DMA registers are accessed, either by the processor via the target interface or by its transfer control unit via its own internal bus. Since these operations occur in an asynchronous system, nothing can guarantee that both the processor and the transfer control unit will not request access to the registers simultaneously. A mechanism to handle these requests is required.

Some mechanisms that had been investigated are:

- Dual-ported memory
- Register locking
- Arbitration

5.4.1 Dual-Ported Memory

Dual-ported memory could be used to implement the DMA registers. This would allow two units to perform operations on the registers simultaneously. However if one unit performs a write operation and the other unit performs either a read or write operation, some additional mechanism is still needed to prevent a data conflict. Also as the probability of the registers being requested by both units simultaneously is low, it is not necessary — in performance terms — to perform both these operations in parallel; one unit could wait until the other finishes without much effect on the performance of the DMA controller. Dual-porting also significantly increases the register unit's silicon area when compared with other methods.

5.4.2 Registers locking

To enforce mutual exclusion the channel registers could be locked by the DMA controller after being programmed and enabled by the processor. This could use a flag which can be read by the processor and the processor must not read or write that channel's registers until this flag is cleared. Whilst the DMA controller performs data transfers it can access this channel's registers safely; when the data transfer is finished it will clear the flag and let the processor have access to that channel's registers again. This mechanism allows both the processor and the DMA controller's transfer control unit safe access.

However this causes a problem in that the DMA operation can't be interrupted by the processor; once the DMA operation has started it must continue until it is finished.

This behaviour is not desirable since, if something goes wrong, the DMA controller could hog the bus and the processor would have difficulty in recovering the system. The 'free run' type of data transfer could not be used if register locking mechanism is used on this DMA controller because it could never be stopped. Also the locking flag which is read by the processor and written by the DMA controller still needs mutually exclusive access.

5.4.3 Arbitration

To solve this problem in the asynchronous logic design, arbitration is unavoidable and the only soultion when two asynchronous activities must interact in this way. It will be discussed in more detail in next section.

5.5 Arbitration

When a shared resource is required to be mutually exclusively accessed by two or more other units, some form of arbitration and temporary locking is necessary. For example if resource C is shared by A and B, and if A sends a request before B, A should be granted exclusive access and B must wait until A finishes its operation before B get its turn.

In a synchronous system requests from A and B could be separated in time or could be considered as simultaneous if they are received in the same clock cycle. In the latter case prioritization must be used to determine which unit should gain access first. Such methods are possible in a synchronous system because of the discrete timing model used by the synchronous logic design.

In asynchronous logic two events never occur 'simultaneously', however the problem of determining which request arrived first and how long the resource is granted to the first request before it is released can still be a problem; an arbiter is used to make an arbitrary decision to grant mutual exclusive access to these requests.

In the 'real world' arbitration is not desirable because it leads non-deterministic behaviour, with consequent difficulties in design verification, testing and performance prediction. The number of arbiters should therefore be minimized. However they are unavoidable in certain circumstances and some examples are given below.

5.5.1 MARBLE Arbitration

The MARBLE bus is a multi-initiator bus; two or more initiators can send requests simultaneously. For example, the processor could send a request to read a value from a DMA controller register and the DMA controller send request to read data from the memory. Since MARBLE can service only one device at a time an arbitrated decision as to which device should be served first must be made. MARBLE uses a central arbiter to handle this situation. The particular mechanism used [1] is not of direct relevance to this thesis however.

5.5.2 DMA Registers Arbitration

As explained in the previous section, the DMA registers could be accessed by the processor and the transfer control unit. Arbitration is required to let both units access the registers safely. The whole DMA register unit could be treated as a single unit in which either the processor or transfer control unit has access to a register while all other registers would be unavailable.

On the other hand, each DMA register could be treated as separate unit, access to each register being independent. These two methods are extremes of a scheme which divides the registers into arbitrary regions. Separating all registers allows the processor and the transfer control unit more freedom of access, but requires arbiter for each register. Treating DMA registers as a single unit needs only one arbiter but registers access could be done less freely.

Channel Registers' Arbitration

In this design *channel* based arbitration is used on a wide internal bus which allows the transfer control unit access to all registers in a DMA channel at once. Each channel's registers has one arbiter; access from the transfer control unit to these channel registers or access from the processor to one register from this channel uses the same arbiter and is independent from other channel registers. This scheme lies between the schemes outlined above.

Global Registers' Arbitration

The DMA global registers are assigned with arbitration separated from the channel registers. The global registers allow no access from the transfer control unit, but only from the processor and from the channel registers unit. The global registers unit contains IRQMask and ChanStatus registers.

The IRQMask is accessed by processor to enable/disable IRQ for each DMA channel. The ChanStatus register can be read by the processor to obtain end of run status of each channel. The status is set by the control unit in the channel register. The transfer control unit needs no access to the global register unit; arbitration is required for access from processor and channel registers only.

As shown in figure 5.4 the processor can read/write the global registers via the target interface and each single bit in the ChanStatus register could be set/reset by the channel registers. The ChanStatus bit set/reset operation will be performed one bit at a time, so only one arbitration is required to arbitrate requests from the processor and the channel registers.

5.5.3 Channel Arbitration

A single transfer control unit is shared by a number of DMA channels. Since each DMA channel is independent, several 'requests to transfer' from the DMA channel could arrive either at the same time or while the transfer control unit is

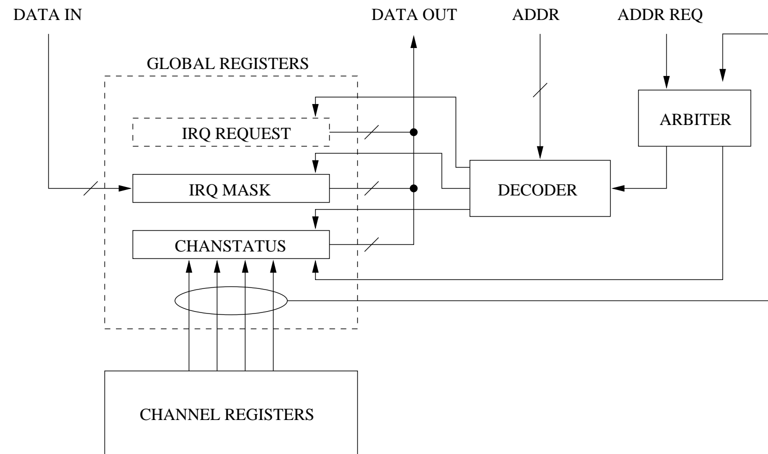

Figure 5.4: Global Registers Arbitration

still performing an earlier data transfer for another DMA channel. Arbitration then is needed to select the next DMA channel to transfer data.

With more than two DMA channels, the arbitration needs to accept more than two requests (four for this design) and grant access to one of these requests. The simple two input arbiter can be used to build an arbitration tree which can receive request from more than two contenders.

Two types of arbitration tree could be used to select a channel.

Balanced Tree

If the number of request inputs is a power of two, a balanced arbitration tree (figure 5.5) could be built from two input arbiters. At the first level of the tree, pairs of signal are connected to an arbiter, every pair of first level arbiters is connected to the second level arbiter, and so forth until the last level which leaves only one output which is connected to the shared resource.

In the case of a single active request, that request will be serviced after winning each stage in the tree. Contention at any level will be resolved arbitrarily. In theory this could mean that, if both inputs to a particular arbiter were continually stimulated, one could be excluded from access. In practice this will not occur, due to a property of the arbiter used – if one request is active at the time that the

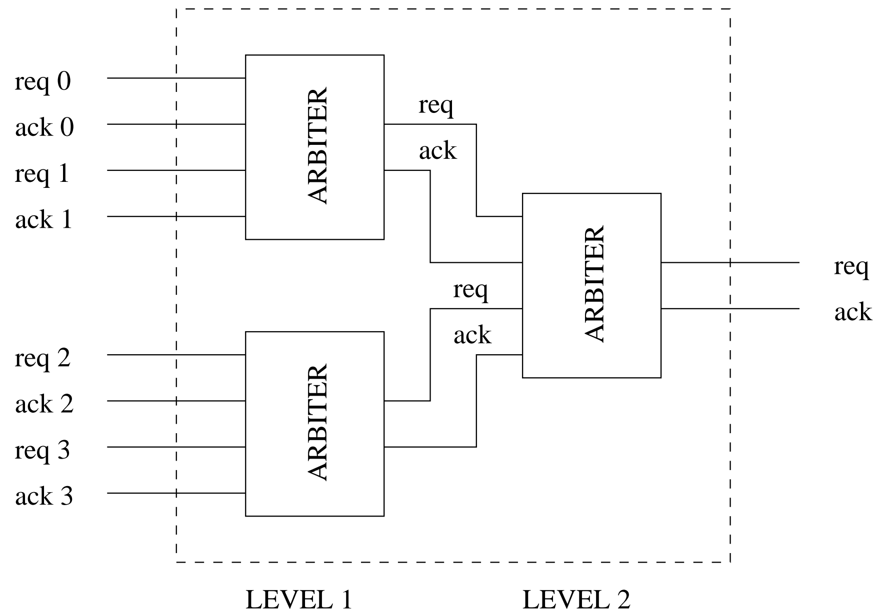

Figure 5.5: Balanced arbitration tree

other completes its transaction the former will be granted before the first request can be reasserted. This means that, if all the inputs are continuously active, each will be guaranteed an equal share of the shared resource.

Even though this arbitration tree can guarantee that each request has equal chance to access the shared resource, the order of gaining access might not be the same as the order of arrival of the requests. For example if contenders A, B, C, D send requests in that order and the time to get services or use the shared resource takes longer than the propagation of request to pass through the second level of the arbiter unit, the order the units gains access will be A, C, B, D. Since A and C will get into the second level of the arbiter tree, before B and D respectively, A will gain access before C, when A finishes and releases the resource C is the next unit to gain access, as shown in figure 5.6.

Unbalanced Tree

To give all contenders the same chance of access to a resource using balanced tree, number of contenders must be a power of two; for any other number of contenders an arbitration tree cannot be built to give the same chance to access

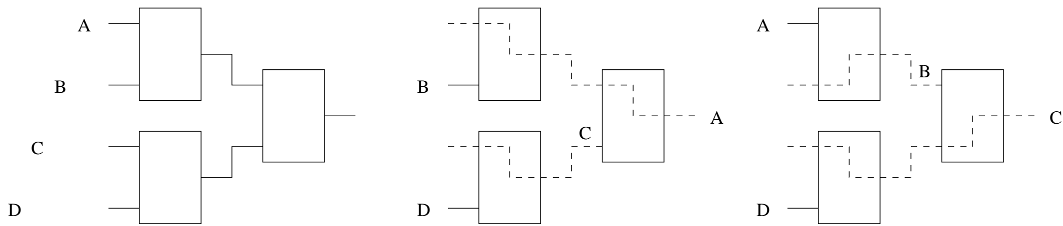

Figure 5.6: Balanced tree order of gaining access

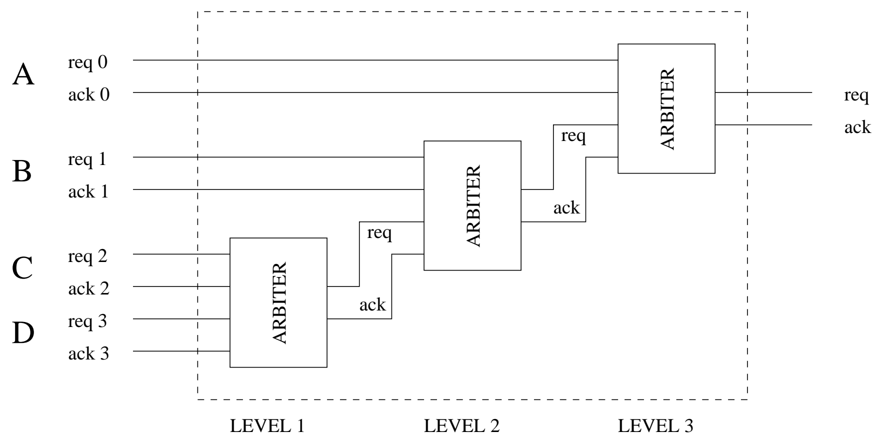

Figure 5.7: Unbalanced arbitration tree

for all contenders. However this kind of arbitration tree, an unbalanced tree, has some advantages that could be used in an asynchronous system.

Even though strict prioritization cannot apply to an asynchronous system because request arrival never occurs simultaneously, if all contenders are always wanting to access the shared resource, i.e. when a contender finishes accessing the resource it sends the next request immediately, this unbalanced tree would gives different 'bandwidth' to the contenders (figure 5.7).

C and D have equal chance to gain access to their arbitration, B has equal chance to (C+D), and A has equal chance to (B+(C+D)). If all contenders are always busy to access the resource, A gets 50%, B gets 25%, C and D get 12.5% of bandwidth. The sequence of gaining access is shown in figure 5.8.

In typical DMA operations, in which a DMA transfer is used to transfer data between a peripheral device and memory, the data transfer does not saturate the transfer engine even though more than one channel is used. Here the arbitration

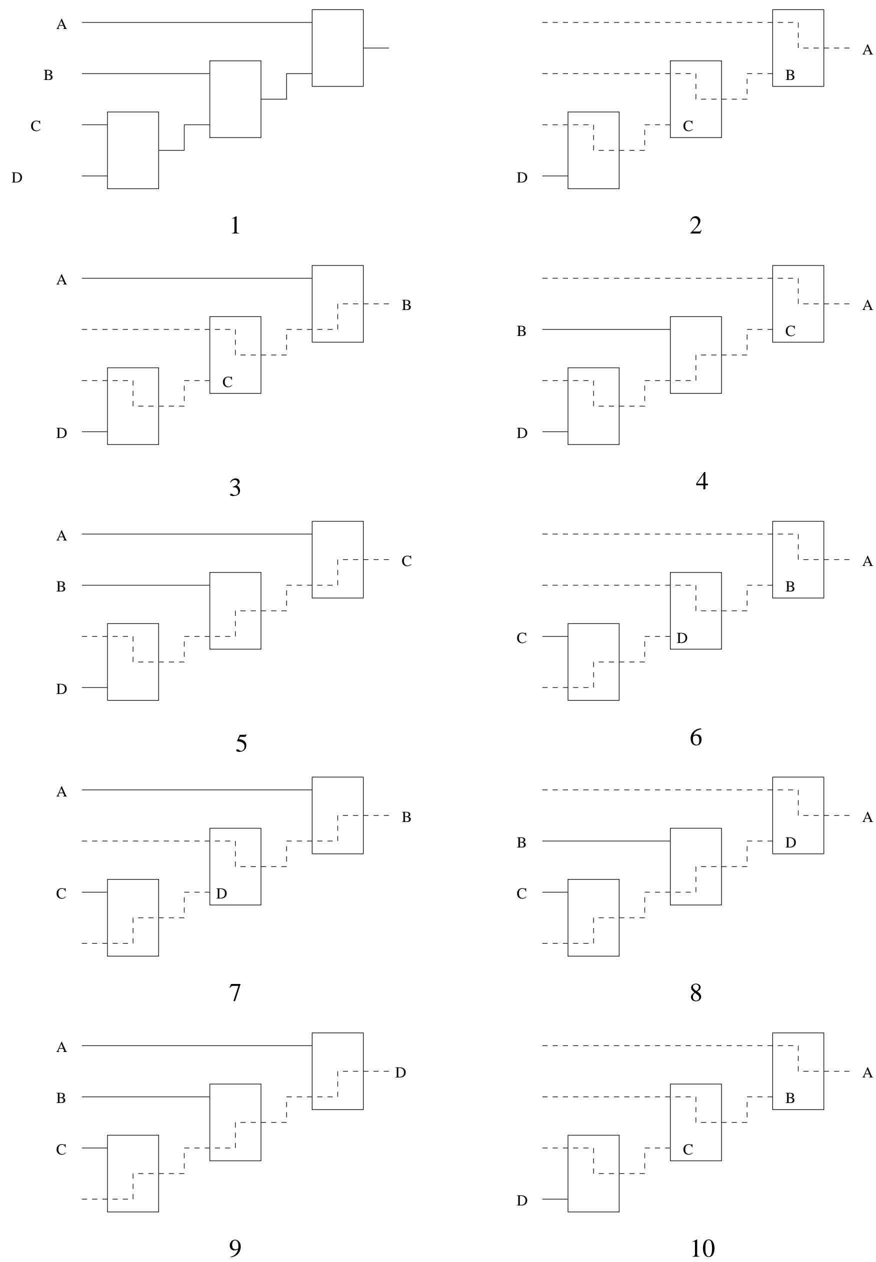

Figure 5.8: Unbalanced arbitration tree

tree will behave merely as a channel encoder which encodes the channel request signal to a channel number and sends it to the transfer control unit.

The possibility that the unbalanced arbitration tree could be used as 'bandwidth allocator' for DMA channels occurs when more than one channel is programmed to perform memory to memory transfers in which requests from DMA channels would saturate the transfer engine. However this situation is unlikely because this will first saturate MARBLE. In practice it is more appropriate to program the DMA controller to perform memory to memory transfers sequentially.

The bandwidth allocation could be useful when one DMA channel is used for a memory to memory transfer and others are used for peripheral to memory or peripheral to peripheral transfers. If the lower bandwidth is programmed for memory to memory transfer the other channel will have its request latency lowered somewhat, since memory to memory transfer are usually faster than peripheral to peripheral or peripheral to memory data transfer.

5.6 Problems

Using arbitration to allow the processor and the transfer engine access to the DMA registers safely could lead to a deadlock problem. This is caused by two (or more) units trying to access the same two (or more) shared resources to complete their operations; when one unit gains access to one resource and another unit gain access to another resource both wait indefinitely for other to free the remaining resource they need and the system deadlocks.

5.6.1 Deadlock

Deadlock is a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something.

In a DMA transfer a deadlock can be caused by the following situation: the

processor attempts to read a DMA register at the same time as a DMA request from a peripheral is asserted. The processor wins the bus arbitration and addresses the DMA controller but, by this time, the DMA request has caused the transfer engine to win the register arbiter and take control of the channel registers. The processor must then wait, but in doing so remains in control of MARBLE.

If the transfer engine insists on obtaining MARBLE before releasing the register arbiter neither operation can proceed and the system is deadlocked. As the processor cannot be deferred this situation must be avoided by the transfer engine relinquishing its hold on the registers before requesting MARBLE. The processor may then complete its cycle, MARBLE will be freed and the DMA transfer may proceed.

A diagram of this deadlock problem is shown in figure 5.9.

1. Peripheral asserts request.

2. Channel request is mapped and sent to transfer engine.

3. Transfer control reads register and win register arbitration.

4. The processor wins bus arbitration, occupying the bus, but is blocked by the register arbiter.

5. Transfer control tries to access bus without releasing the register arbiter, but is blocked by bus arbiter.

5.6.2 Race condition

Race condition in the DMA transfer operation can be occured in the channel registers. Although it is unlikely, it is possible that both the transfer engine and the processor could attempt to write different values to the same register. This could occur if the processor gains access between the read and write phases of the transfer engine's register updates.

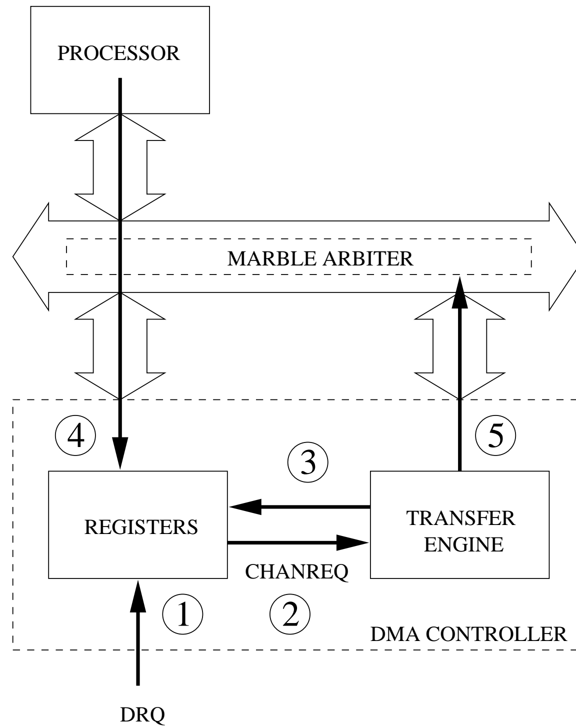

Figure 5.9: System Deadlock Problem

Normally this would not happen, but — for example — the processor could write to disable a channel and then begin to reprogram that channel before the DMA read-write operation completes. In this case the transfer engine would overwrite the newly programmed value.

Whilst practically improbable it is theoretically possible for this to occur in an asynchronous system where cycles *could* happen in any order and simulation (see chapter 6) has revealed this flaw. It is therefore necessary to ensure that the values written by the processor are safe from being overwritten by the transfer engine if this happens during the DMA read-write operation.

To avert this problem the transfer engine is prevented from writing if the processor has begun to modify a channel's registers. It is thus necessary to detect this circumstance. This is done by setting a flag each time the processor *writes* to the channel; the flag is cleared when the transfer engine reads these registers. Because the transfer engine always alternates between reads and writes the flag should always be clear when it attempts a write operation; if this is not the case

the processor has intervened and the write operation can be suppressed.

5.7 Synchronization

Synchronization is an important function in a computer system that makes two different units in the system able to work together. Asynchronous systems synchronize using handshake signals (as described in chapter 2). Synchronization is also used in synchronous systems when an infrequent, non-periodic communication between two independent subsystems is needed.

Two of these synchronizations are concerned in the design of this DMA controller: the first is the DRQ signal pair for synchronizing with a peripheral during the data transfer operation, the second is the interrupt request used by the DMA controller to notify the processor when the DMA transfer is finished.

Both synchronizations have the same purpose, the 'client' uses the synchronization signal to request the 'server' to perform some function when it is ready. The synchronization is performed directly between the 'client' and the 'server' but the function is not.

In the first case, synchronization between peripheral devices and the DMA controller, the server is a DMA controller and the client is a peripheral device; the function requested by the client is to transfer data which is interaction between bus and DMA controller and between bus and peripheral device, not directly between peripheral device and the DMA controller.

In the second case, synchronization between DMA controller and the processor, the server is a processor and the client is the DMA controller. Also in this case function of data transfer is performed through bus.

The peripheral device sends a request signal to the DMA controller to notify it that it is ready to transmit or receive data. The DMA controller sends the IRQ signal to interrupt the processor when the transfer sequence is finished; the processor reads the IRQ Request register across the bus to acknowledge the

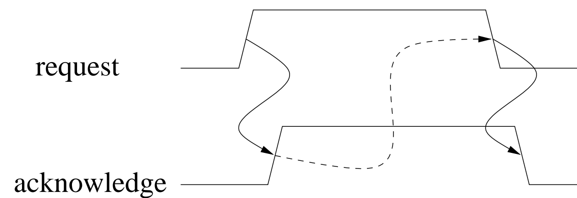

Figure 5.10: Peripheral Request Handshake

interrupt request.

5.7.1 Peripheral Synchronization

A synchronizing signal from the peripheral device to the DMA controller is normally used in DMA transfer. The device uses this signal to notify the DMA controller when it is ready to receive/transmit data. The DMA controller does not begin the DMA transfer operation until it knows that it can be completed. Note that, in the case of peripheral to peripheral transfer, both devices must be ready; the DMA controller must receive a request from both devices before data transfer can begin.

In a synchronous system, acknowledgement of the request signal could be done by the DMA controller reading/writing data from/to that peripheral, so that the request signal is inactived before the cycle is completed. The peripheral device can send the next request signal immediately after the DMA transfer, if it is ready for the next transfer.

The synchronization signal is usually a level-sensitive signal where an active state activates a DMA transfer. The signal must therefore return to its inactive state when the transfer occurs to prevent accidentally repeated operations. There is therefore a form of handshake protocol which interleaves the request signal with an acknowledge signal, as shown in figure 5.10.

5.7.2 Processor Synchronization

The processor interrupt request is a synchronization mechanism between the DMA controller and the processor. The DMA controller notifies the processor when the data transfer is finished and the processor reads the ChanStatus register to determine which DMA channel caused the interrupt.

Unlike the peripheral synchronization handshake there is no explicit acknowledgement that the interrupt has been serviced. The interrupt request is cleared in software by the processor and the interrupt service routine is responsible for ensuring the IRQ is inactivated before it is re-enabled. This is less ‘clean’ than a handshake protocol, but is the same model as a synchronous ARM system (and happens at software speeds).

5.8 Summary

There are several design issues and problems in the design of the DMA controller. Using an arbiter to solve simultaneous access problems to the registers causes several other consequent problems. However these problems can be solved, even though these make the design of the DMA controller harder.

Chapter 6

Behavioural Models

6.1 Introduction

To prove that the design of a complex system is working correctly without using formal methods, which is not practical for a very complex system such as a VLSI design, or building the actual system a system model and methods to test that model are necessary.

System modelling can be done at many levels of complexity. An abstract model shows the definition and purpose of the system. For example the abstract model of the DMA controller can be described as:

```
repeat
    READ;
    WRITE;
until END_TRANSFER=true;
```

The READ operation reads data from the source device, and the WRITE operation writes data to the destination device. The operation starts by READ followed sequentially by WRITE and repeats until the END_TRANSFER condition is true.

This abstract model of the DMA controller could be modelled and simulated by any general programming language without difficulty, since there is no great complexity in the model.

The behavioural model is more complicated; it models how the system behaves and interacts with its environment. Such a model contains more accurate details of the system than the abstract model. It could describe/show interactions between the model and its environment better than the abstract model, which is more useful in finding problems in the design. However the more complex the model becomes the more time is needed for modelling and testing.

Behavioural models of a system usually expose the design problems before the actual system is built so the design can be changed and the problems can be rectified. It can also be used to predict performance, efficiency or other aspects of the system that are interesting and need close inspection so the design of the system can be improved.

6.2 Modelling tools

Modelling a complex circuit such as a processor can be done by using VHDL[17], Verilog[21] or other hardware description languages. However these languages were designed for synchronous logic circuits and are not appropriate when used with asynchronous logic design. For high level modelling, where synchronous and asynchronous system do not differ, these languages can be used. However at the implementation level differences between synchronous and asynchronous circuits are well distinguished and these languages are not appropriate for modelling the asynchronous system.

LARD[6] is a hardware description language, based on CSP-like channel communication, which has been developed for behavioural modelling of asynchronous VLSI systems. Channel based communication in LARD allows each module, which represents a component in the system, to communicate with others asynchronously. This communication does not need to specify the exact handshake protocol but leaves it to be chosen during implementation; this simplifies the models and reduces the time needed to develop and maintain them.

The model developed in LARD can be simulated and tested using the LARD toolkit. The execution environment of the LARD toolkit includes an interpreter, library and runtime modules which allow the designer to debug behavioural models at source level, watch for activity on each communication channel, control each variable during runtime if necessary, and so on.

Most components in the AMULET3i subsystem were designed as behavioural models using LARD, this also includes the DMA controller.

6.3 Simplified models

The DMA controller was first designed by implementing LARD modules of a basic DMA controller unit. Other simplified modules for the processor, bus, memory and peripheral devices were implemented to make the complete system necessary for testing the DMA transfer operation and functionality of the DMA controller. The simplified models (both the DMA controller and its environment modules) allowed the DMA operation and function to be tested repeatedly without taking too much simulation time for this simple operation.

This simplified models of the DMA controller comprise:

- Processor

  The simplified processor model which has enough functionality to read/write the DMA registers to program a DMA channel to perform data transfer.

- Bus

  The bus module functions as a middle-man to exchange data between other devices connected to it. The bus receives requests from the processor and the DMA transfer engine and performs read/write data with the memory devices, peripheral devices or the DMA registers.

- Memory

  The memory is a read/writable storage device which receives requests from

the bus.

- Peripheral devices

  The peripheral device model is a read/writable storage device connected to the bus which can synchronize with the DMA controller using a peripheral request signal when transmitting/receiving data.

The DMA controller itself is modelled as single module with two concurrently running sub-modules: respectively the registers and the transfer engine. Each module has its own interface to the bus, allowing separate communication. However DMA register access by the transfer engine and the bus interface (within the register unit) is done using the register locking mechanism since the registers are defined as internal variables which can be accessed by both modules.

The simplified model of the DMA controller is made to ensure that its functionality and interactions with its environment are correct. The behaviour and function of its internal components are not considered at this point.

6.4 AMULET3i DMA Controller Model

After the functionality of the simplified model has been tested and proven a more complicated model that reflects the design of the DMA controller was produced.

The internal modules of this model comprise:

- Registers
  - Target interface and Transfer Engine decoders
  - Global Registers
  - Channel Registers
  - Channel Request Mapping
- Transfer Engine

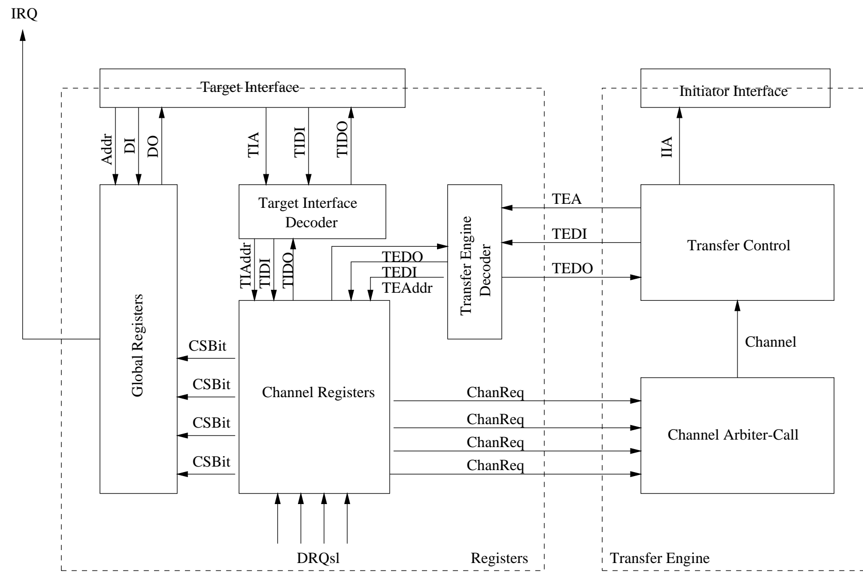

Figure 6.1: Internal Modules of the DMA Controller

- Transfer Control
- Channel Arbiter

In the behavioural model these are independent modules in LARD, each module communicating with the other via channels and signals as independent components in the circuits communicate via asynchronous data bundles and control signals.

6.5 Model of the Registers

The registers in the DMA controller store the values needed for the DMA transfer operation. These values are the addresses of the devices, the number of data items to transfer and other control information that controls the data transfer operation. Most DMA registers can be modelled with simple variables which represent these values. The registers unit includes these variables and interfaces to the actual registers for other modules to access.

Several decoders (one for the target interface, one for the transfer engine, and each group of registers also containing an internal decoder) and arbiters (one for

global registers, and one for each set of channel registers) are also incorporated to the registers unit. The decoder is needed to select a particular register; the arbiter forces two or more units that could attempt to access the registers simultaneously to take their turn.

6.5.1 Global Registers

The global registers handle read/write requests from the target interface and set/reset the ChanStatus bit from the channel registers; they also control the interrupt request to the processor. Each bit of the ChanStatus register can be independently set/reset. Also two bits of ChanStatus can be concurrently set/reset. All ChanStatus bit set/reset requests are ORed together as one request to the global register arbiter.

6.5.2 Channel Registers

The channel registers are independent from each other; they can be accessed by the transfer engine and the target interface. Each channel has its own arbiter to grant access to these two units.

In the target interface only a single register can be read/written at a time. A flag, the 'write_protect_flag', is set to prevent the 'race condition' (discussed in the previous chapter) when a value is modified by the processor in that channel. Writing a value to the control register by the processor also causes the channel register to send a request to clear the corresponding bit in the ChanStatus register.

When using the transfer engine interface every register in a channel isi read/written at once. In the read operation every register is read and the 'write_protect_flag' bit is cleared, allowing the registers to be written in the write-cycle of the transfer control.

In the write operation, the 'write_protect_flag' is checked before values are written to the registers; if the flag is set the processor has intervened and the 'new' values will be ignored. If the write operation resets the enable bit in the

control register, a request will be sent to the ChanStatus register to set the corresponding bit.

6.6 Model of the Transfer Engine

The transfer engine is composed of two major units: the transfer control unit and channel arbiter unit, as shown in figure 4.8.

The Channel Arbiter unit receives requests from each control register and, if there is more than one request, makes an arbitrated decision in choosing a channel to be services by the transfer engine. Simple two input arbcalls are used in a tree to create an n-input arbiter.

6.6.1 Transfer Control Module

The transfer control module performs data transfers with MARBLE using the channel number supplied by the channel arbiter; it also reads/writes channel registers from the register unit to update the transfer states. The MARBLE read/write operations are performed concurrently with the register update/write-back operations to increase DMA performance. To prevent the processor interrupt request being sent before the data transfer is finished (for the last data item transferred) register write back is delayed until the MARBLE operations are finished.

6.6.2 Channel Arbiter

An arbiter is a non-deterministic component in the real circuit implementation. However when modelling by software "non-deterministic" functions depend on the 'random' function of the computer system that is running the software, which is usually not non-deterministic. This therefore does not reflect the real behaviour of the circuit. However as any choice is arbitrary it is believed that the simulation cover is adequate.

6.7 Simulation Schemes

To check that the behaviour of the DMA controller model is correctly designed the model has been tested with the simplified model of processor and other modules. Even though these modules do not behave identically to the real AMULET3 modules they are accurate enough to test the DMA controller behaviour.

Testing with the real 'environment' was also done using the AMULET3 processor core, MARBLE bus, memory and the simplified peripheral device modules for which the real modules don't yet exist.

Tests on all of types of data transfer, i.e.:

- Peripheral to Memory
- Memory to Peripheral
- Memory to Memory
- Peripheral to Peripheral

have been done with every DMA channel using both the 'free run' and a definite number of data items transfer (i.e. self-terminating). During the 'terminating' test the operation was also interrupted and the channel reprogrammed in an attempt to find deadlocks or other problems. Running multiple channels concurrently was also tested with both the same and different types of data transfer.

6.8 Simulation Results

The simulation results show that interaction between the DMA controller and other modules in the processor subsystem works as expected. However testing revealed some problems with the initial design, both a deadlock and more subtle problems such as a race condition and the early interrupt request. These were subsequently corrected and verified.

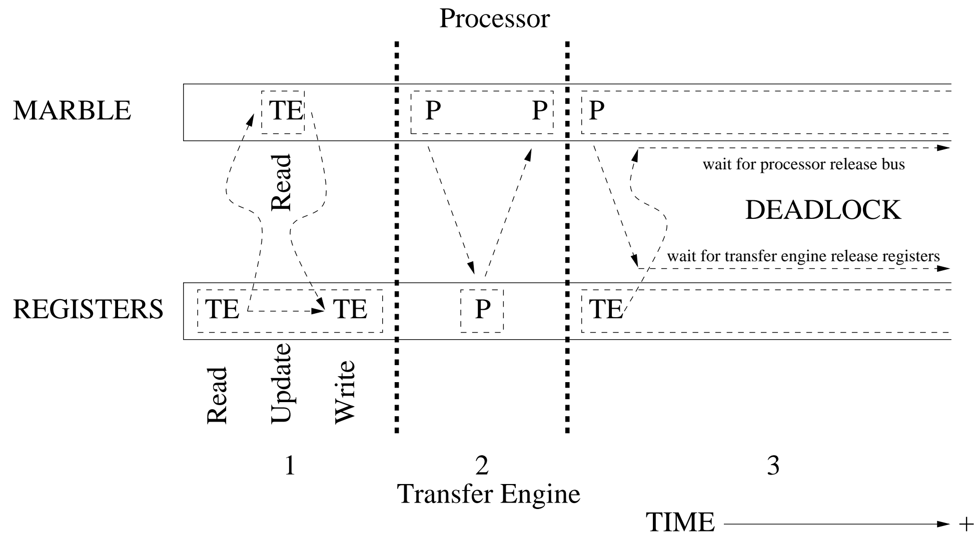

Figure 6.2: Deadlock cause by atomic register access by transfer engine

6.8.1 Deadlock

The deadlock problem is an obvious one when compared with other problems in the design. As shown in figure 6.2; the transfer engine accesses a register *atomically*, in which, when access is granted to the transfer engine, the register is not released until register write back is finished. This causes deadlock if the processor tries to access the same register.

This problem could be easily solved by not allowing the transfer engine to use atomic access to the register unit, as shown in figure 6.3. The transfer engine requests access separately for each operation it performs with the registers.

6.8.2 Early Interrupt Request Sending

This problem was found when a DMA channel was programmed to transfer data between two peripheral devices. When the processor receives an interrupt request from the DMA controller it reads the IRQRequest register to determine which DMA channel caused the interrupt. The processor could receive the interrupt before the data transfer was finished, as shown in figure 6.4.

Even though, in practice, this is not a real problem because the transfer should

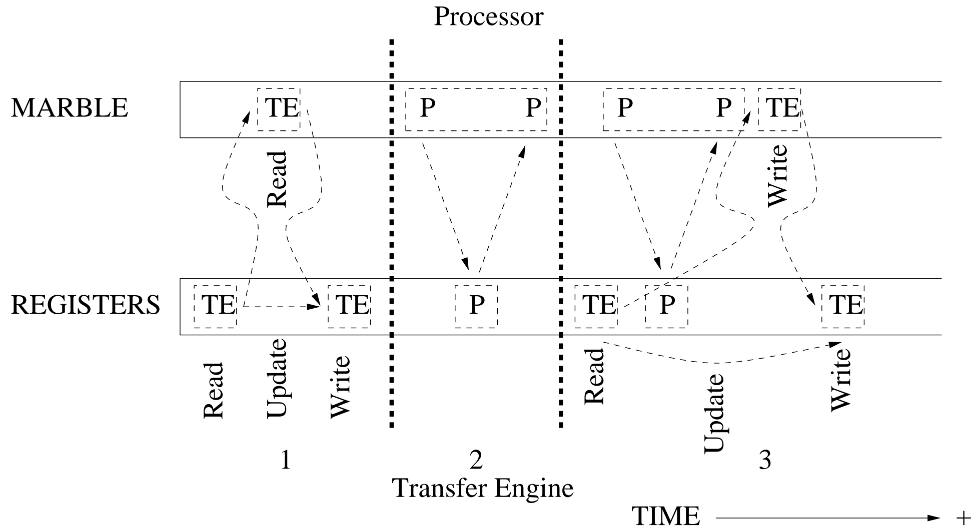

Figure 6.3: Non-atomic register access solve the deadlock problem

be finished before the processor enters its interrupt service routine, in theory the processor could have begun to reprogram the DMA controller before the last transfer had taken place. To ensures that a problem cannot be caused on the last data transfer the register write back operation is delayed until data transfer with MARBLE is finished, as shown in figure 6.5.

6.8.3 Race Condition

In normal circumstances the race condition described in previous chapter would be extremely unlikely. This problem was found when the solution for early interrupt request was introduced and the simulation was done with peripheral to peripheral data transfer. As shown in figure 6.6.

This problem was solved by adding a flag which prevents the transfer control overwriting register values if the processor has written some value to the register unit (and set flag by doing that), as shown in figure 6.7.

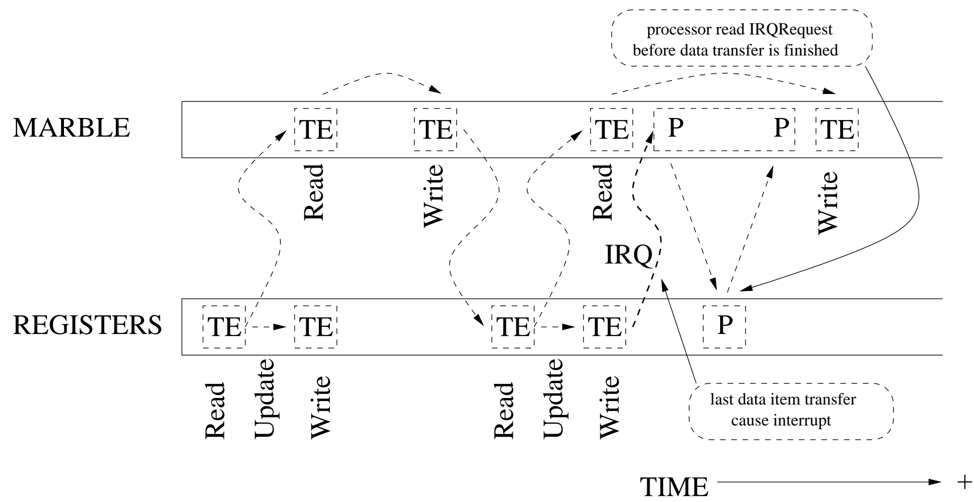

Figure 6.4: Early interrupt sending

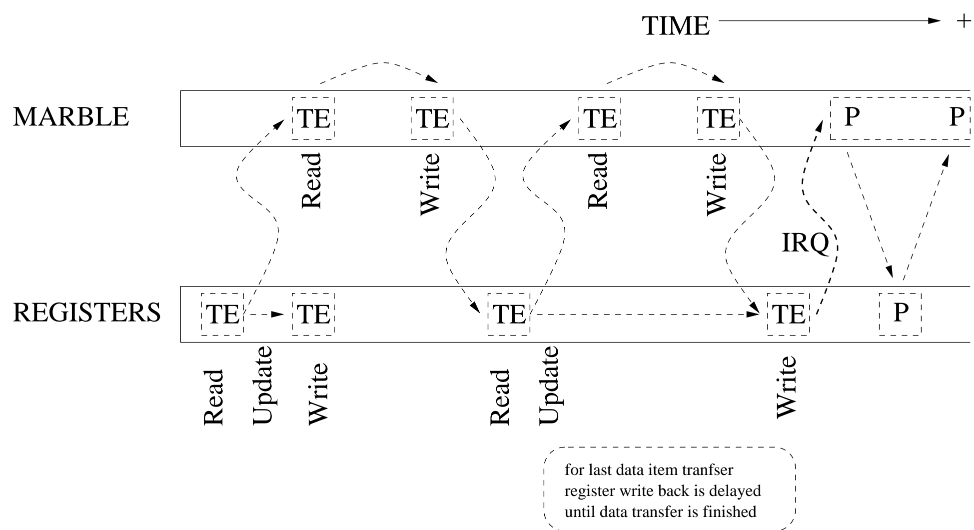

Figure 6.5: Delay registers write back for last data transfer

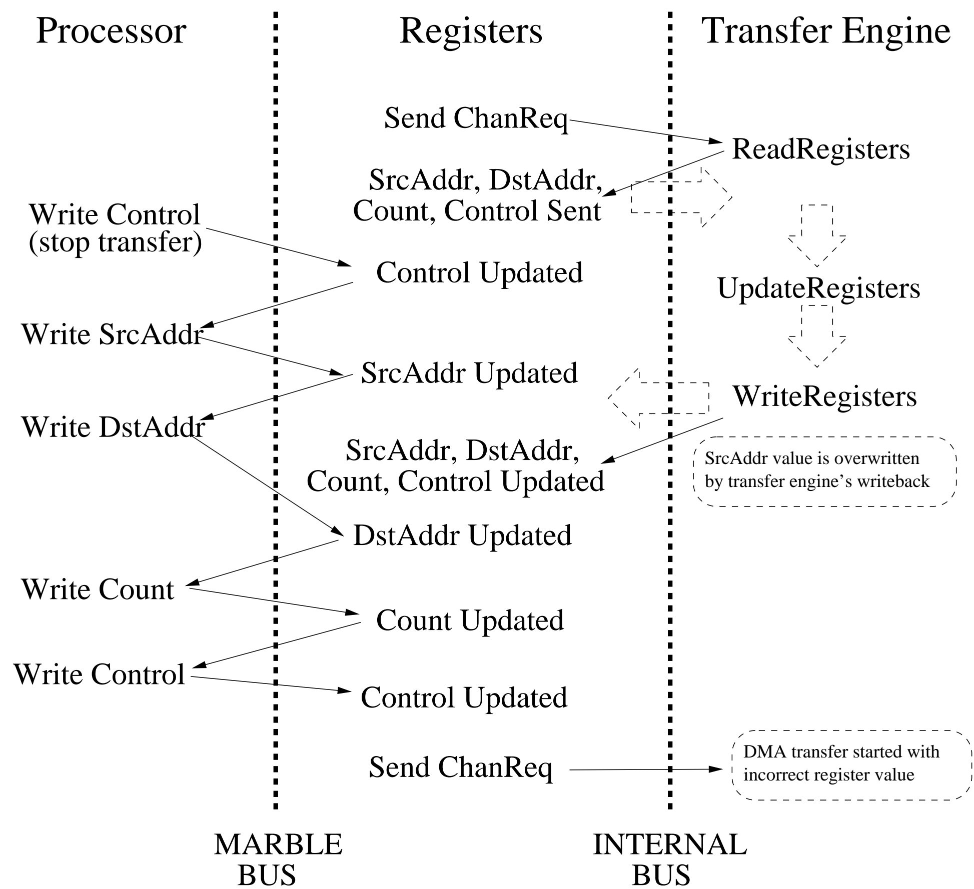

Figure 6.6: Race Condition

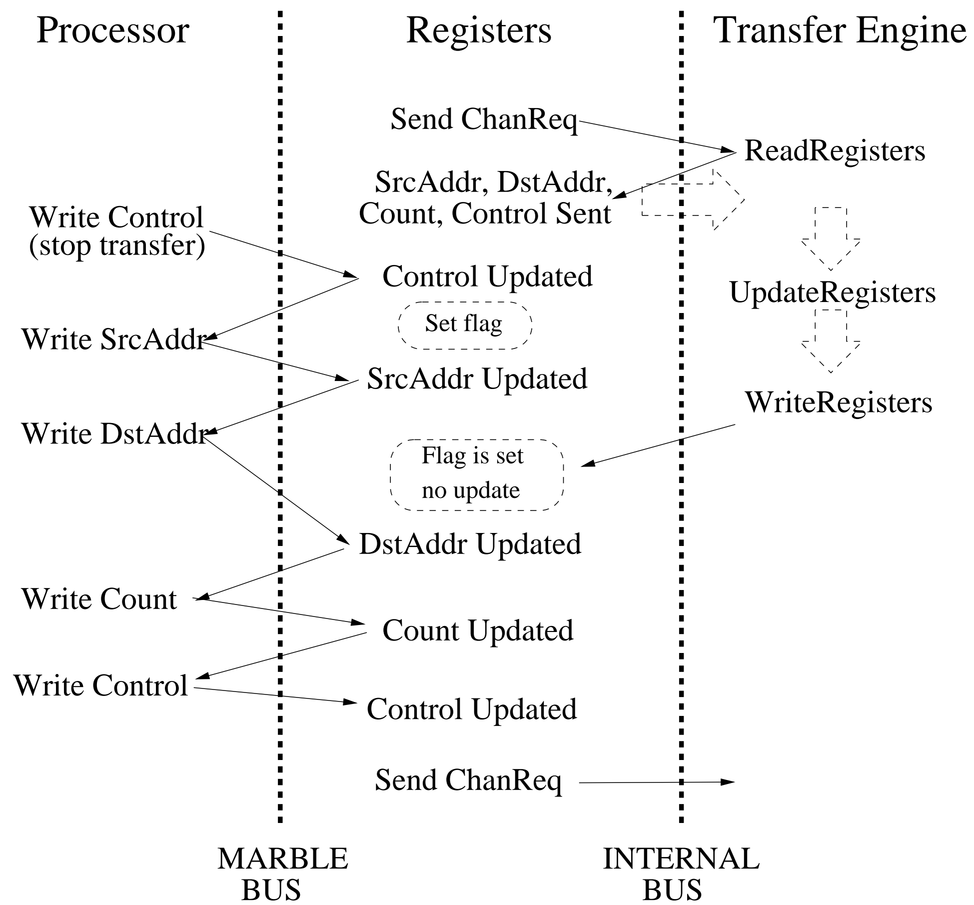

Figure 6.7: Using a flag to prevent Race Condition

6.9 Summary

Modelling and simulation can reveal problems in the design especially at the behavioural level. Some of these problems are not obvious from the design itself, but when modelling and simulating they are shown up and the designer can revise the design.

More detailed simulation can be done to estimate the power consumption and performance of the DMA controller, however this was not done because of time constraints in writing this thesis; this is left for future work.

Chapter 7

Conclusions

The design of an asynchronous DMA controller is possible. This thesis presents a possible design for such a DMA controller in which problems of asynchronous implementation are addressed and the practical features of existing, deployed, DMA controllers are provided. Even though there are several problems in the design that do not exist in synchronous designs, several techniques exist to solve the problems. This thesis does not present an implementation for the DMA controller described but a design has been produced, software modelling has been done, and the simulation has been tested and verified with the processor and other modules on the processor subsystem.

7.1 The AMULET3i DMA Controller

The DMA controller which has been designed is a multi-channel DMA controller which could be used in a general MARBLE application. It can transfer data between any combination of memory and peripheral devices. Even though this design has four channels and supports four peripherals it could be expanded without change to the conceptual design.

The design has been modelled and tested using LARD, the hardware description language for asynchronous logic design. Simulation and test has been done

with both a simplified model of the processor system and actual AMULET3i processor systems which have been used in design of the silicon.

7.2 Conclusions

It is feasible to design and use an asynchronous DMA controller with the asynchronous processor subsystem. There several particular issues in the design that make the asynchronous DMA controller different — and more difficult to design — than its synchronous counterpart. For example, the problem controlling access to shared resources must be addressed.

Arbitration is used in several places in this DMA controller. Even though non-determinism is undesirable because it makes the unit's behaviour unpredictable and thus harder to model and test, it is more appropriate than other mechanisms for sharing resources because it requires less hardware and imposes a negligible performance penalty.

In an asynchronous system communications can happen in an arbitrary order (unlike synchronous system where communications are simultaneous); this can cause the system to have more reachable states and more possibilities for errors. This is made worse by arbitration with its non-determinism and can introduce problems into the design, such as deadlock and race condition.

Behavioural models can show up many potential faults including some which would not manifest in a real system because of timing constraints which is adjustable in the models. Using behavioural modelling the problems can be detected and solved.

Application of the arbitration tree, which is built from two input arbiters, has been investigated. Multi-way arbiters or arbitration trees can be built in different ways to make them give different bandwidth to the requests. This feature was not utilized in this design of DMA controller because requests for data transfer rarely saturate the transfer engine without saturating the bus first. However both

kinds of arbitration tree might be useful elsewhere.

The mechanism to transfer data on MARBLE is limited to store-and-forward because this implementation of MARBLE does not support fly-past transfer. However it is planned to support this in the future version of MARBLE. With support from the bus, fly-past transfer could increase the DMA transfer performance in some situations; it does not require major changes in the DMA controller to support this function, only a part of the transfer control need be redesigned. This should be investigated in the future work.

A DMA feature that provided by both Intel's 8237 and National's NS32203 but not investigated in this thesis is "double buffering" in which a set of register is used to store pre-configured values of the registers. When the data transfer is finished the DMA channel re-load these values to its registers and continue the data transfer. This mechanism allows DMA transfer to continue without interruption.

This feature was rejected at the beginning of the design because it requires another set of register for each DMA channel (as 8237) or reduces the number of channels that could be used at once by half (as NS32203). However this could be a very useful function for several applications and should also be investigated for the future work.

Three major design problems: deadlock, race condition, and early interrupt requests, have been found, mainly as a result of simulations since some of them are very subtle. Formal methods or other mechanisms could be used to analyze these problems. However with the hardware description language and simulation tools provided, modelling the system and simulating it is much easier.

LARD, the hardware description language for asynchronous logic design, has been used for modelling this DMA controller. It provides support in the language, library, and tools for simulating and testing; the behavioural modelling could be done more easily than using other languages that do not provide support for asynchronous logic design. The channel based communication in LARD

abstracts the asynchronous data transfer and signalling protocol very efficiently when compared with other hardware description language such as VHDL.

There are some minor problems when using LARD to model a very complex system such as the AMULET3i processor subsystem. Simulation would be very slow if detailed modules of every unit are used — the current implementation of LARD is interpreted at run-time and is therefore slow. Error reporting when compiling still a bit cryptic. Even though, as programming language, LARD supports parallel scheduling the interpreter itself is a single-thread scheduler, therefore modelling a non-deterministic circuit component — such as an arbiter — cannot be achieved realistically. LARD is intended for high-level modelling; low-level models of circuit components could be done but not very efficiently.

There are several features of LARD that could be used in the design of the DMA controller to investigate performance and power efficiency when used in co-simulation with other applications, but due to time constraints this is left for future work.

A lot of work still needs to be done to implement the actual asynchronous DMA controller. Even though most of data-path of the DMA controller could be done without much problem, several diagrams and figures in this thesis have shown data-path of the DMA controller could be built. However, control circuit, especially in the transfer control unit still needs investigation.

Bibliography

[1] W.J. Bainbridge and S.B. Furber. Asynchronous macrocell interconnect using marble. *Async98*, 1998.

[2] R.J. Baron and L. Higbill. *Computer Architecture*. Addison Wesley, 1992.

[3] Intel Corporation. *8237A High Performance Programmable DMA Controller (8237A,8237A-4,8237A-5) Datasheet*, 1994.

[4] Intel Corporation. *Intel 430MX PCIset 82371MX Mobile PCI I/O IDE Xcellerator(MPIIX)*, 1996.

[5] National Semiconductor Corporation. *NS32203-10 Direct Memory Access Controller*, 1995.

[6] P.B. Endecott. Lard documentation home page. http://www.cs.man.ac.uk-/amulet/projects/lard, 1998.

[7] J.D. Garside et. al. Amulet3i - an asynchronous system-on-chip. In *Proceedings Asynchronous 1999*. IEEE Computer Society Press, April 2000.

[8] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A micropipelined arm. In *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[9] S.B. Furber, J.D. Garside, and D.A. Gilbert. Amulet3: A high performance self-timed ARM microprocessor. *International Conference on Computer Design*, 1998.

[10] S.B. Furber, J.D. Garside, S. Temple, P. Day, and N.C. Paver. AMULET2e: Asynchronous embeded control. In *Proceedings Asynchronous 1997*, pages 290–299, April 1997.

[11] J.D. Garside, S.B. Furber, and S-H. Chung. AMULET3 revealed. In *Proceedings Asynchronous 1999*, pages 51–59. IEEE Computer Society Press, April 1999.

[12] S. Hauck. Asynchronous design methodologies: An overview. Technical Report UW-CSE-93-05-07, University of Washington, April 1993.

[13] Dave Jaggar. *Advanced RISC Machines Architectural Reference Manual.* Prentice Hall, 1996.

[14] Edward J. Laurie. *Modern Computer Concepts: The IBM 360 Series.* South-Western Publishing Co., 1970.

[15] A.J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The design of an asynchronous MIPS R3000

microprocessor. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[16] Nigel C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, Computer Science Dept. The University of Manchester, 1995.

[17] Douglas L. Perry. *VHDL.* McGraw-Hill, Inc., 1991.

[18] D.P. Siewiorek, C.G. Bell, and A. Newell. *Computer Structures: Principles and Examples*, chapter 31 A Dual-Processor Desk-Top Computer: The HP9845A, pages 508–532. McGraw Hill, 1983.

[19] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[20] A. Takamura, A. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Uneo, and T. Nanya. Titac-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. *International Conference on Computer Design*, 1997.

[21] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic, 1998.

[22] J.E. Thornton. *Design of a computer: The Control Data 6600.* Scott, Foresman and Company, 1970.