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

January 1999

By

Chatchai Jantaraprim

Department of Computer Science

Contents

List of Tables

List of Figures

Abstact

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

Dedication

To HangChee & HYingKlang, with love.

Acknowledgements

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

Thanks being friends, correcting my English, playing games, opening the world of asynchronous logic design to me, and most of all giving me the most enjoys 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 inspires my juggling, Doug.

Thanks a lot to Chung for those several interesting discussions, even though many of them I don'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 companion.

Chapter 1

DMA Controllers

1.1 Introduction

In a computer system, the processor spends considerable time moving data around, 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 in addition to performing the transfer. A simple data transfer operation such as read/write 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 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, while DMA transfers are in progress, 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 thus increase system performance and reduce power consumption (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 micro-computer, or engineering workstation [3, 5]; for larger computer systems, such as a super-computer, mainframe, or mini-computer, other kind of controllers or input/output processors are used to control data transfers, or handle input/output [16, 2, 18, 12].

Typically the DMA controller would be used to transfer data between a peripheral device and memory device, 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 with every device which would impose a high wiring cost when the number of devices is large. The bus simplifies connection between devices on 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 initiator and target devices. When a processor programs a DMA controller, the DMA 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. 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 do not have data locally, e.g. communication devices. To read from or write to the slow target device, the initiator must wait until the target device is ready to transfer data or use some mechanism such as polling or interrupt 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 (usually called an 'interrupt' when sending to the processor) to the initiator device when it is ready to perform a transfer.

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

To perform a DMA transfer, the DMA controller must be programmed with a source, a destination, an amount of data to be transfered, and other transfer control specifics such as whether the DMA controller should interrupt the processor or not when the transfer is finished, or the type of source and destination devices. The DMA controller can then perform the DMA transfer autonomously.

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

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

When both source and destination device 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 matched,

and will notify the processor if required.

To transfer 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 interrupt signal from a device to the DMA controller is usually called '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 that will be discussed 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 for an 8-bit data bus; a newer one supports a 16-bit data bus too. An 8237 has four independent DMA channels which are cascadable; a channel could be used to connect to another 8237 and expand the number of DMA channels in the system. The number of channels which can be used by cascading this DMA controller 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. Transfer between memory devices is 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.

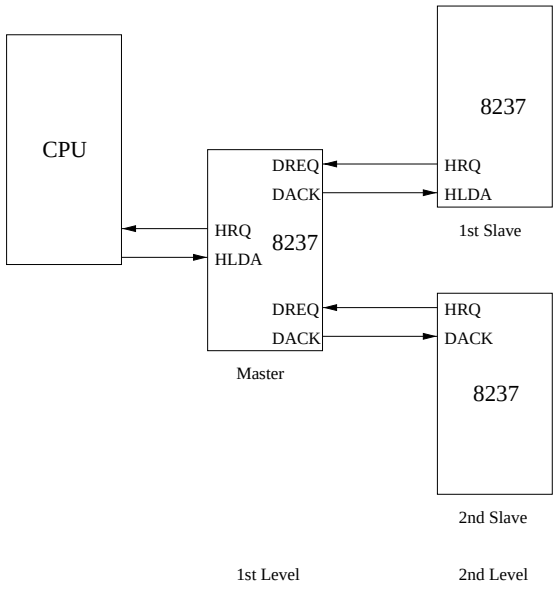


Figure 1.1: 8237 in cascade mode

This DMA controller contains two set of registers which are used by the processor to program the DMA transfer characteristics. While a channel being transferred, the primary set of register is used, when the transfer finishes and if the secondary set of register was programmed by the processor, the DMA controller copy registers values from the secondary set to the primary set and continues. This allows the processor to program a DMA channel even though that channel is being used.

Each peripheral request signal is permanently fixed to a particular channel. The 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 number of data it can transfer is 65536 items. Data transfer can be done only through the bus; data is read from source device to the DMA controller during the read-cycle and write from the DMA controller to the destination device during the write-cycle.

There are several variant of 8237 DMA controllers that are based on 8237 with some enhancements. In most recent PCs which use 80x86 processor and its variants, the 8237 DMA controller is part of the chipset on the mainboard [4].

1.2.2 National's NS32230

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

Features of the NS32230 DMA controller:

- Transfer data between memory devices, between peripheral devices, as same as 'traditional' transfer data between memory and peripheral device.

- 8/16 bit transfer

  Bus width for applications used NS32230. Transfer to/from 8-bit peripheral devices also support assembly/disassembly data 8-bit $< - >$ 16-bit. Two 8-bit data reads from the peripheral device can be combined into 16-bit transfer for single write to 16-bit memory or peripheral device.

- Remote/local configuration

  In remote configuration, the DMA controller and devices are connected to a dedicated bus and data transfer is performed on this bus. 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. However 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 transaction on the bus; a read followed by a write. Another mechanism is fly-past transfer in which the DMA controller initiates both read and write requests simultaneously and the data is transferred directly from source device to destination device. Fly-past mechanism is faster at transferring data, but requires extra decode logic and cannot be used for memory to memory transfer on the same memory device. This DMA controller can support either mode of these.

- Command chaining

  A channel can be chained to the next channel, when the data transfer is finished on the first channel, register values from the second channel will be copied to the first channel and the DMA controller can resume transfers on the first channel, allowing continuity in transfer for a specific device.

- Search capability

  Data transfer functionality can be adapted to be used for other functions such as "on-the-fly" comparisons without adding much resource.

- Interrupt vector generation

  Common features in a mechanism to notify the processor after the transfer is finished.

1.3 Summary

DMA controller can provide a useful enhancement to a microprocessor system, relieving the CPU of consideration 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 controller are :

- Transfer function which can transfer between peripheral and memory, and between memory devices (even though it needs two channel to perform memory to memory transfer for 8237).

- Multiple-channels

- Several data sizes, to support different data width devices.

- Interrupt request to the processor when transfer is finished.

These functions of DMA controller are used as guideline to design 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 almost equally used at the beginning of digital logic design era, but synchronous logic design had been more favoured in the last few decades and become the accepted design style for complex circuits.

These two design styles differs from each other by 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; 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 any parts in the system must be done before 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 because of several reasons [17]:

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

- It is widely taught and understood
- Parts that operates with clocks are widely available
- System noise has died away by the time a clock event occurs

Simplicity in timing models make 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.

With this simplicity, the development in circuit design has long been concentrated on the synchronous design style, and asynchronous design style has been virtually ignored.

As opposed to the synchronous logic design counterpart, the asynchronous logic design style is :

- Harder to design
- Not much taught or understood until recently
- Lacking components in standard libraries
- Short of development tools
- Overhead for simple circuit is higher

While the synchronous logic design uses a global signal for synchronization, the asynchronous logic design let circuits that need to communicate 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 also worsens the power dissipation problem in some applications.

- Worst case performance

  Since sub-circuits which can work at higher 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-modularity design

  A complex circuit designed for a specific clock speed could not reuses for a different clock speed, redesign or major adaptation is 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 processor have been successfully implemented using different styles of asynchronous logic design [14, 13].

2.2 Asynchronous logic design

The design methodologies for asynchronous logic could be categorized by delay models, a period of time needed for digital signal transition from one logic to another in wires or circuit elements, to two models: bounded delay, and unbounded delay models. The bounded delay model assumes that delay of a circuit element or wire is bounded. The opposite is assumed in the unbounded delay model.

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

- Timed circuits

  Correct operation of this circuit is dependent on the delays in circuits elements and wires.

- Delay-insensitive circuits

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

- Speed-independent circuits

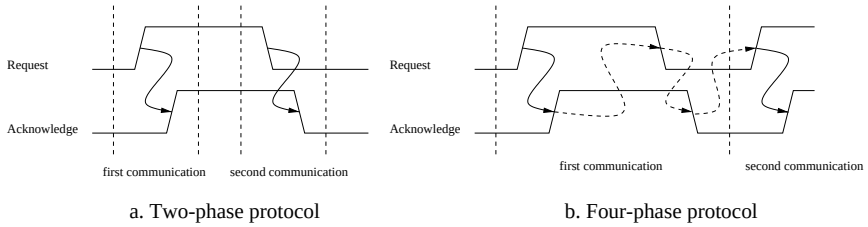
  Correct operation of this circuit is independent of the delays in both circuit elements and wires.

- Quasi-delay-insensitive circuits

  This circuit is delay-insensitive with '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 as single-rail and dual-rail encoding. In the single-rail encoding one wire is needed for each bit of information or data value to represent logic '1' and '0'. 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'. Signal transition in one of this pair of wires represents



a. Two-phase protocol b. Four-phase protocol

Figure 2.1: Signalling protocols

the data bit value for every communication. Note that signal transition could not occur in both wires (from a pair) in the same communication, because it represents both logic '0' and '1' in the same time for one data bit which is invalid.

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 bit wire-pair. Four possible states are possible on a wire-pair: '00' indicating that that bit is idle, '01' and '10' indicating data zero and one and '11' which is an invalid state which is never seen. By ORing together the two bits we can tell whether a data bit is being signalled or not. By ANDing (or more commonly the use of 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 with is used as timing information for the data communication. Data could be "pushed" from the sender to the receiver by sender setup data and initiates 'request' signal to the receiver, the sender must keep the data value until it receives 'acknowledge' signal from the receiver which means data has been received. Data could be "pulled" from the sender to the receiver by the receiver initiates 'request', the sender setup the data and 'acknowledge' when the data is ready to sent.

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

In a two-phase signalling protocol the information is transmitted by a single transition 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 two phases of the communication. Rising and falling transitions are equivalent in the two-phase protocol.

In a four-phase signalling protocol the information is transmitted by 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' transition on both wires contains no important meaning in communication but is treated as recovery state that is used to set the signal to predefined states, as shown in figure 2.1 b.

Several asynchronous logic design styles has been developed and used so far. The favourite styles that had been applied to real circuit and implement to a very complex circuit are discussed else where [10]. One of these style that used by the AMULET group in develop its processors is Micropipelines [17].

2.3 Micropipelines

The Micropipeline technique, with some modifications in timing and signalling protocols, is used in the design of the AMULET processors. The Micropipelines design style, proposed by Ivan E. Sutherland, could be described as event-driven elastic pipeline (a pipeline in which the amount of data can vary). It has been used and proved to be successful as frame work of asynchronous logic design style which can be classified as bounded-delay bundled data for the data-path with delay insensitive control path [14].

Simple structure of the Micropipelines is shown in figure 2.2.

A Micropipeline uses a simple data bundle to transfer data between pipeline

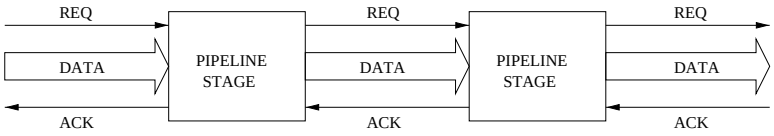


Figure 2.2: Simple structure of Micropipelines

stages. Each stage could be composed of storage and processing units; communication between stages uses a request and acknowledge signal pair. Data validity is indicated by the handshake signals. Both two-phase and four-phase protocols were presented in the Micropipelines paper [17].

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 being developed by the AMULET group at the University of Manchester. Previous research by this group also included two asynchronous processors, the AMULET1, and AMULET2e [14, 9].

AMULET1, the first asynchronous processor developed by this group, was a feasibility study of using asynchronous logic design in building a very complex circuit. The 32-bit RISC commercial processor, ARM architecture [11], was chosen mainly because it is small, simple, and a low power design; also AMULET group members had familiarity with this processor architecture and could get support for developing the processor [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, 14, 9].

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 micro-controller. 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 synchronous processors. The AMULET2e was a highly usable asynchronous embedded system chip [9].

The AMULET3i is currently being developed by the AMULET group. It is designed to be suitable for commercial embedded applications. To gain performance, reduce the 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; the DMA controller is the subject of this thesis.

2.5 Summary

Asynchronous logic design has been used to solve problems that exist in a very complex circuit, such as a processor subsystem. The Micropipelines, one of specific asynchronous design style has been proved to be successful method in implement two AMULET processors. The AMULET3i processor will be the latest which tries to 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

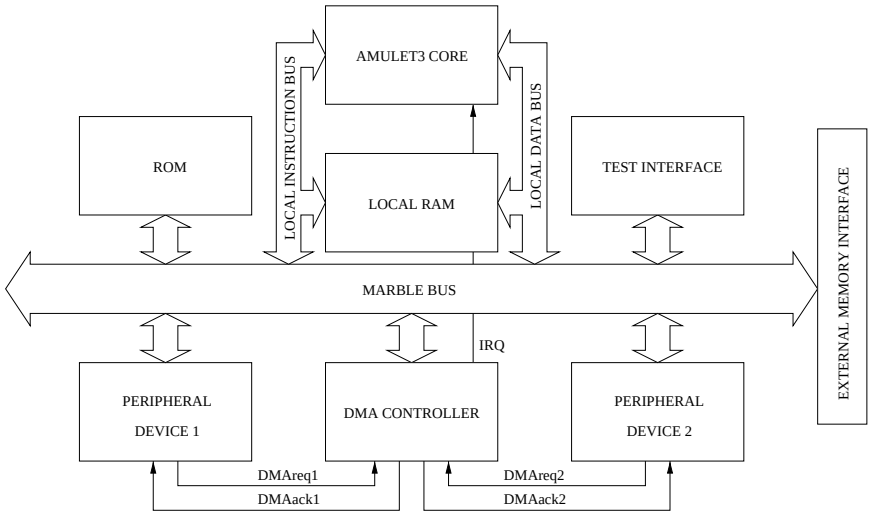


Figure 3.1: The AMULET3i Subsystem

- 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

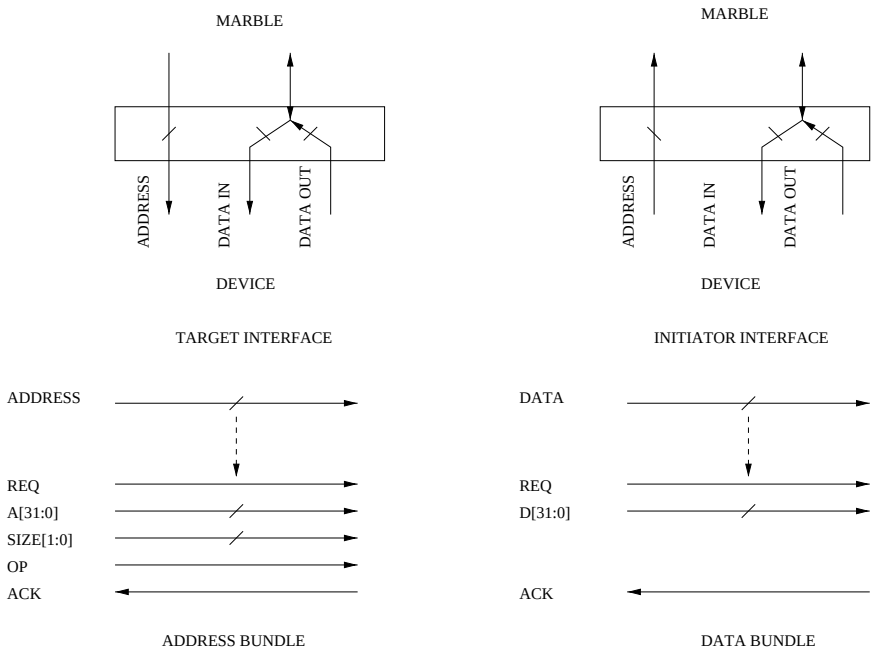


Figure 3.2: The MARBLE interfaces

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.

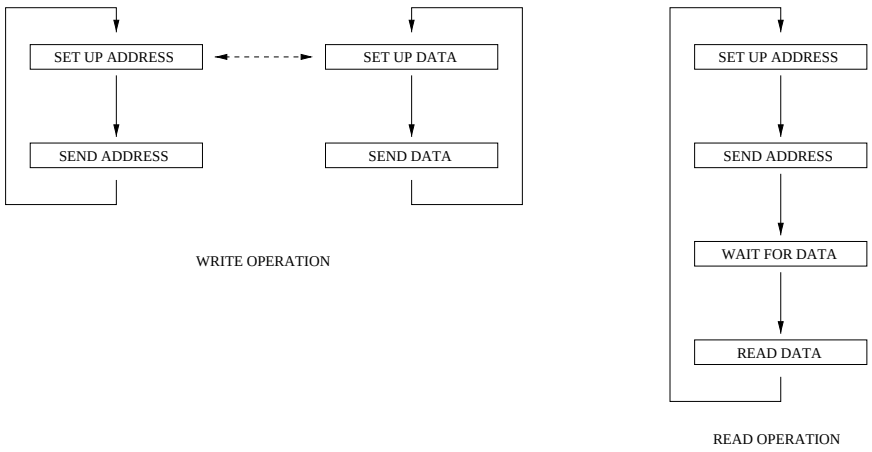


Figure 3.3: Initiator interface data transfer operations

In a read operation, a data item is transferred from the target device to the initiator device across the bus. The target device is finishes receiving the address before it can determine the address and command and send the data 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.

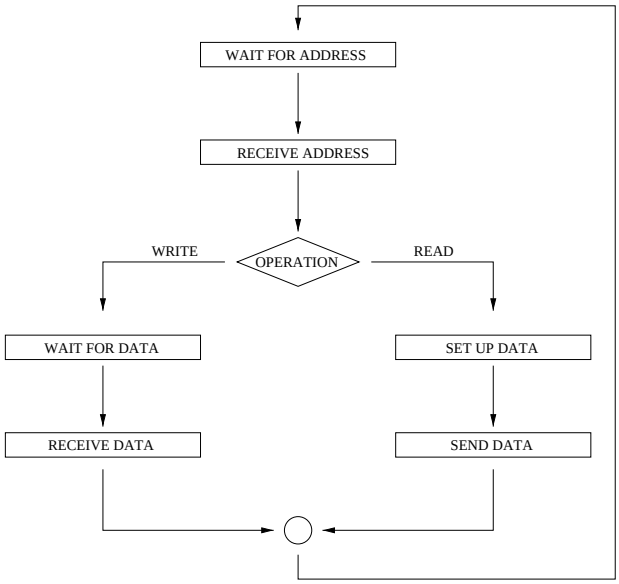


Figure 3.4: Target interface data transfer operations

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 and a write-cycle consequently. 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 support 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 specifications of the DMA controller which are used as a framework of the design are mostly taken from the requirements and its environment. Those that does not in the requirements are arbitrarily chosen, some are influenced by the design of existing DMA controllers. The reasons for arbitrarily chosen specifications will be discussed in the next chapter. This chapter discusses the top level design of the DMA controller.

4.2 Specifications

The DMA controller specifications are :

- Four independent programmable channels

  This 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
  - Peripheral and memory
  - Memory and memory
  - Peripheral and peripheral

  Required by the subsystem to support three types of data transfer.

- Store-and-forward transfer
  This method of data transfer is supported by the MARBLE bus, as opposed to fly-past transfer method, which is 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 at a time.

4.3 The DMA operation overview

Arriving of the peripheral request signal (DRQ), after a channel is enabled activates the channel request mapping block in the register unit to generate the channel transfer request signal (CHANREQ) to the transfer engine. The CHANREQ activates the transfer engine to read the corresponding channel registers

from the register unit and performs data read/write with MARBLE using addresses and data transfer configurations in the channel registers. Simultaneously acknowledgement to the DRQ signal is also performed by the transfer engine after reading/writing with the peripheral device. For memory to memory transfer, no DRQ signal is needed, enable state in control registers is mapped directly by the channel request mapping block to sent CHANREQ to the transfer engine. For peripheral to peripheral DMA transfer, two DRQ signals from both source and destination devices are needed.

Detail the DMA transfer operation is discussed in next chapter.

4.4 The DMA controller internal structure

In the design of 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 all allows the DMA controller to be programmable by the processor via this interface, the transfer engine unit uses an initiator interface to transfer data between source and destination devices.

The transfer engine reads/writes data to/from 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 processor request signal (IRQ) and initiates request for data transfer to the transfer engine with CHANREQ signals when a DMA channel is ready.

4.5 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

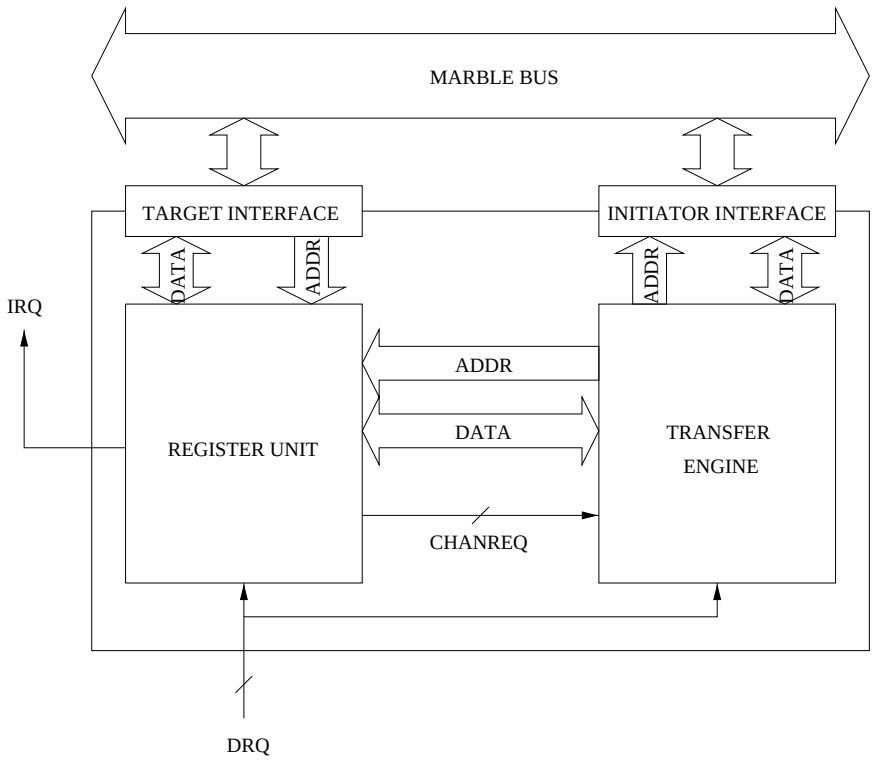


Figure 4.1: Block diagram of the DMA controller

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 used to handle 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) to channel requests (CHANREQ) sent to the transfer engine. A diagram of the register unit is shown in figure 4.2.

4.5.1 The Global Register Unit

The global registers comprise:

- ChanStatus
- IRQMask
- IRQRequest

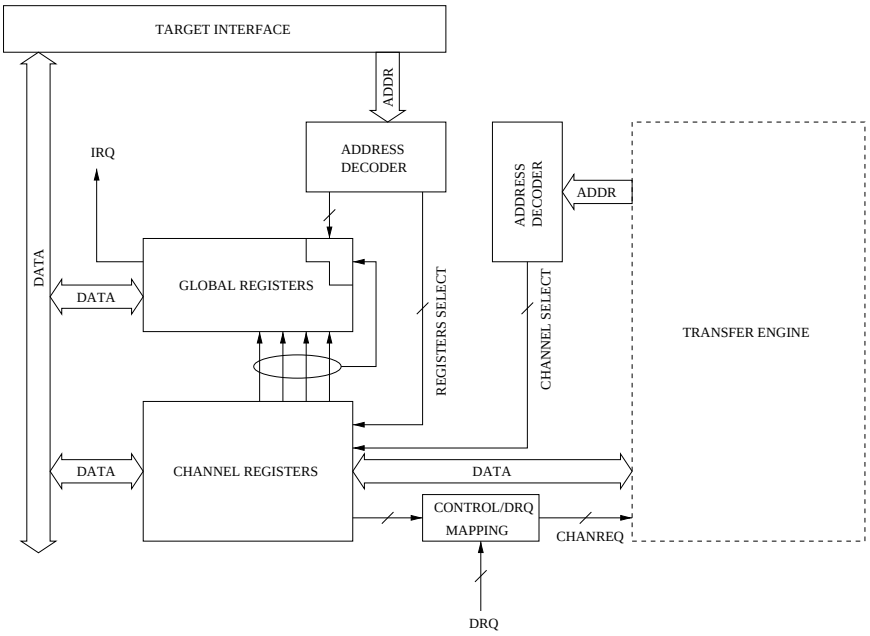


Figure 4.2: Register Unit

registers. Access from the processor to one of these registers is independent from access to channel registers. These registers are used in 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 is corresponds to a DMA channel which is used to specify whether, when the data transfer 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. Every bit in ChanStatus will 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 channels 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 have

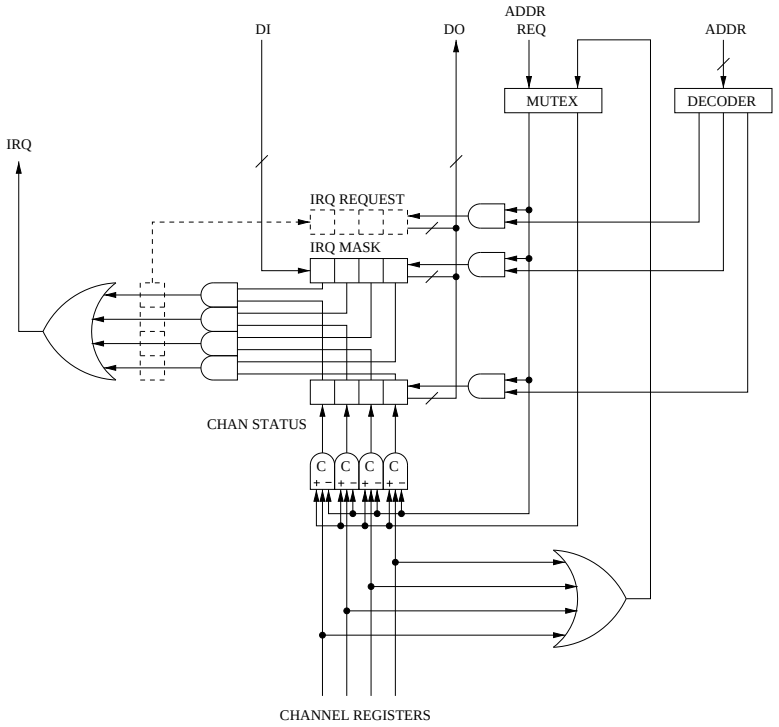


Figure 4.3: Global Registers

non-zero values. The IRQ signal will change when either IRQMask or ChanStatus register value is changed, if the AND operation returns a zero value then the IRQ is removed.

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

4.5.2 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 on different channels simultaneously.

Each set of channel registers comprises:

- SrcAddr
- DstAddr
- Count

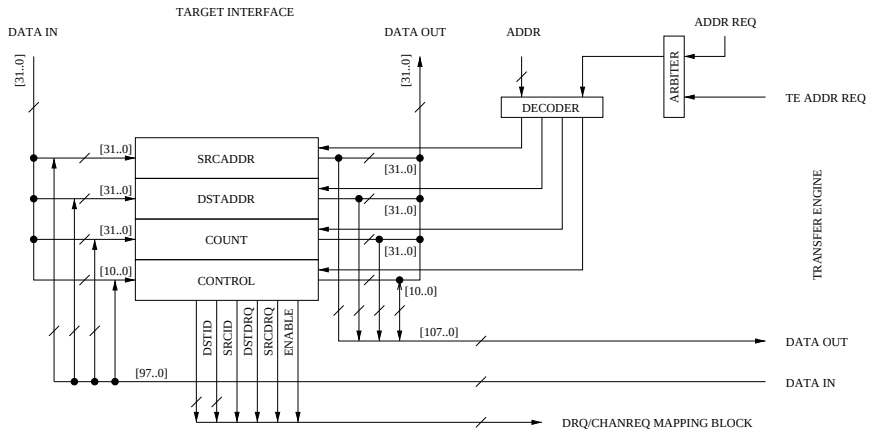


Figure 4.4: Channel Registers

- 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 channel registers, allowing the transfer engine to access all registers in a channel 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 transfer across the full range of the address space. The counter is also 32 bits wide, 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

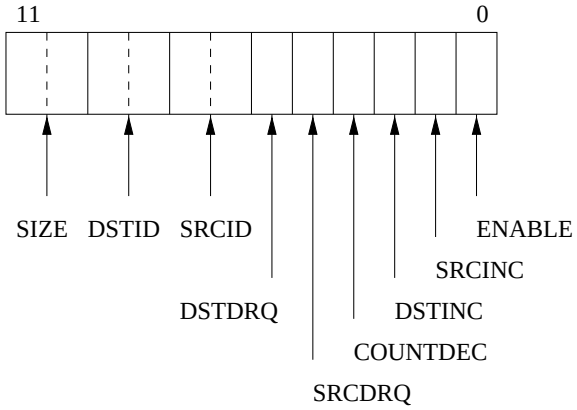


Figure 4.5: Control Register

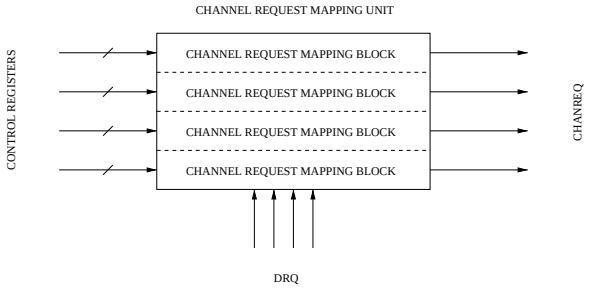


Figure 4.6: Channel Request Mapping Unit

- 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 generate transfer request to the transfer engine.

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.

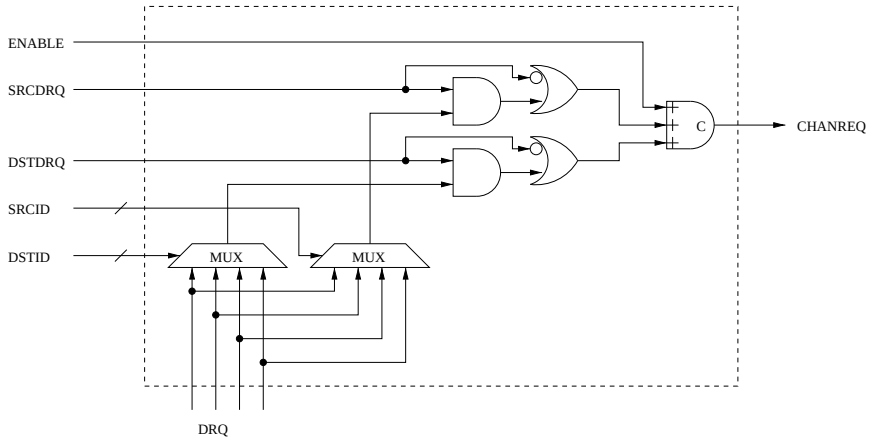


Figure 4.7: Internal structure of Channel Request Mapping Block

4.6 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 channel by initiating a data transfer to the bus. Data are transferred singly. When a transfer is finished, the transfer engine becomes idle again and waits for next request.

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

4.6.1 The Channel Arbiter Unit

The channel arbiter unit arbitrates the transfer request signals sent from the channel registers in the register unit. It sends the arbitrated 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 : balanced tree, or unbalanced tree. Details of the arbiter call elements and arbiter tree will be discussed in next chapter.

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

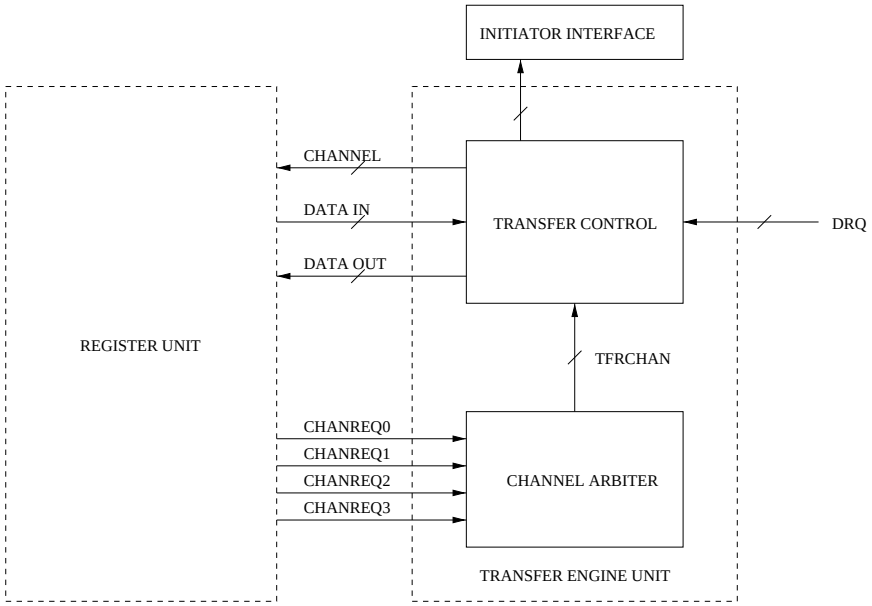


Figure 4.8: Transfer Engine Unit

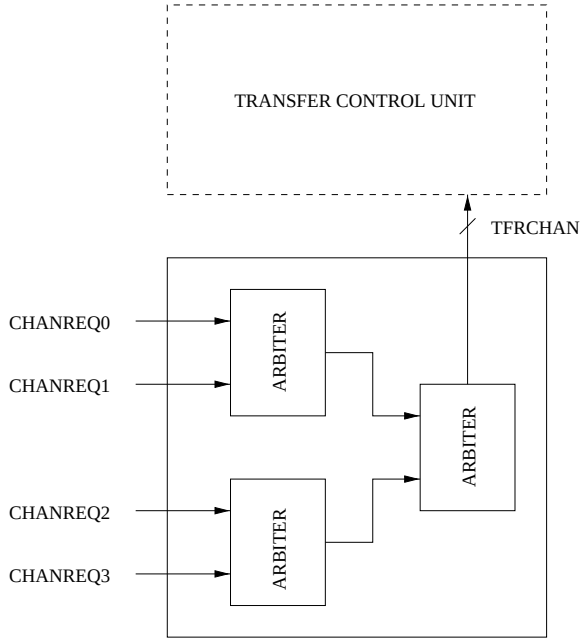


Figure 4.9: Channel Arbiter Unit

4.6.2 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 if required (controlled by 'usedrq' and 'device ID' bits in the control register).
2. Read/write registers to/from the register unit.
3. Initiate data read/write transfer with 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 channels registers from the register unit. Control register bits are used to determine data size, requirements for synchronization with peripheral device during data transfer, source and destination address increment, and counter decrement. The transfer control unit then performs data read/write from/to the devices by initiating address request with buses. If peripheral synchronization is required, it synchronizes with the device with DRQ signals. Simultaneously with data read/write operation, the transfer control update registers and write the updated values back to the channel register. When the operation is finished it waits for next transfer request from the channel arbiter.

4.7 Summary

From top level design, the DMA controller is divided into smaller parts by its functionality. The register units keep all registers and handle request for transferring data for each DMA channel and send requests to the transfer engine. It also handles interrupt request to the processor. The transfer engine performs the actual data transfer when receives a request from a DMA channel. The DMA

operation are performed by tightly coupled cooperation between these two major parts of the DMA controller.

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

Chapter 5

Design Issues

5.1 Introduction

This DMA controller is designed from its specifications (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, the problems, and solutions chosen to solve these problems.

5.2 DMA Registers

For programmability, registers are used. For each DMA channel, these values are needed :

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

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

- The processor word size is 32 bits, 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 transfer with any device in the address space and for lengths of the whole address space.

5.2.1 Transfer Control Configurations

The control operation specifies by whether the device's address is required to 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' are used as configurations instead of the 'type of devices'.

To allow a DMA channel to perform data transfer with a non-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 also to allow a 'free run' type of transfer, in which the DMA controller performs data transfers continuously until it is stopped by the processor. A 'counter decrement/free run' configuration is used to tell the DMA controller to decrement the counter after each data transfer or just to ignore the value of the counter. The 'enable/disable' mechanism to start and stop the DMA transfer is required for the 'free run' type of transfer.

To save registers and allow the processor to program a DMA channel more

conveniently all transfer control configurations are stored as bits field in one 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 is finished, a mechanism to notify the processor is necessary. An interrupt mechanism is chosen for this purpose. However an interrupt costs the processor with a context switch to an interrupt service routine, so a mechanism to allow the processor to specify which channels should send an interrupt when the transfer is finished is also necessary to eliminate unwanted interrupts. The 'interrupt enable' configuration is used. This could be a bit field stored in the control register like other control configuration; however if the processor needs to disable all interrupts from the DMA controller, it has to write to every control register which is not convenient. Instead, the 'interrupt enable' bits for all DMA channels are stored together in one register called 'IRQ Mask' which allows the processor to disable/enable every channel with one register access.

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 channel transfer request signal (CHANREQ) to the channel arbiter unit in the transfer engine when both source and destination devices are ready to transfer data. The channel arbiter then arbitrates and selects a channel; it sends the channel number to the transfer control unit. The transfer

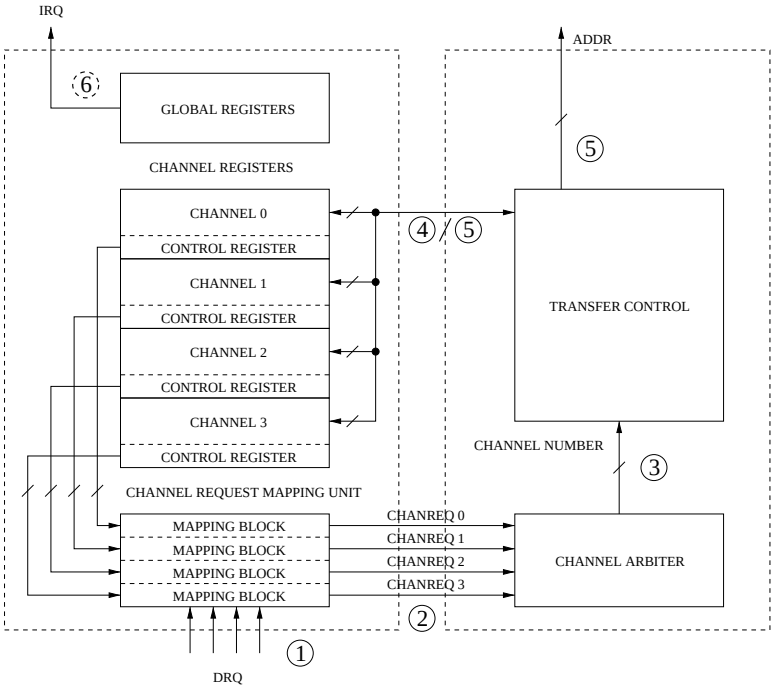


Figure 5.1: Sequence of the DMA operation

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 configurations in the control register and performs necessary functions to stop the data transfer operation and notify the processor if the last data item for that channel has been transferred.

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

- Channel selection operation
- Data transfer operation
- Stop transfer operation

5.3.2 Channel Selection

Each channel request mapping block works independently if a DMA channel is enabled, and source and destination devices for that channel are ready, a CHANREQ signal will be sent from the register unit to the channel arbiter unit in the

transfer engine. If several requests occur, one must be selected from these. Arbitration is used to grant exclusive access to the single transfer engine to just one request at a time.

In the 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 the simultaneous 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 section Arbitration) can be used to select a channel; this gives a fairly small circuit and allows the transfer engine to start the data transfer as soon as possible.

5.3.3 Data Transfer Operation

The transfer between devices is performed on the bus, the DMA controller acting as an initiator device. A read followed by a write cycle is performed.

The transfer control unit starts the data transfer operation when a channel is selected by the channel arbiter unit. The operation starts by the transfer control reading the channel registers from the register unit. These register values are used to control the data transfer operation.

The data transfer operation can be divided into two subsequence operations: 'read operation' and 'write 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 the device 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', but destination address and other destination device's configurations are used instead of the source device configurations; so 'data write' transfer is performed with the destination device.

During the 'read/write operations' which the transfer control performs with the bus, register update and write back operation can be performed concurrently with the register unit. 'Source/destination addresses increment' and 'Counter decrement' configurations 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 channels register in the register unit could use two 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 bus can be used.

Wide Bus Communication

This model uses a wide bus for communication, all registers in a channel are read/written at once. This allows register value transfer between these two internal units in the DMA controller 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.

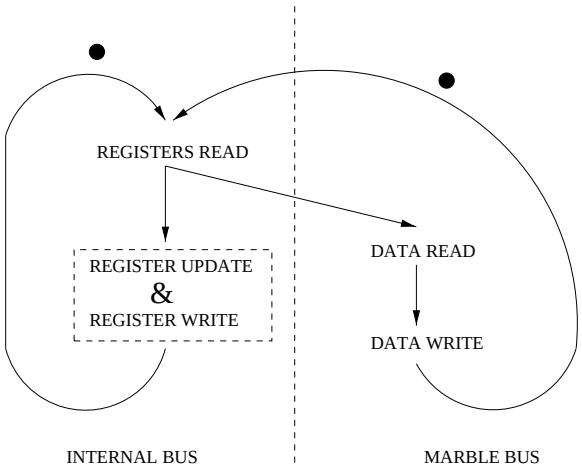


Figure 5.2: Register communication with wide bus

Narrow Bus Communication

This model uses a narrow bus for communication, only one register can be read/written at a time. The transfer control unit would start the communications by reading control register to determine the transfer configurations first. Initiating a data read from the source device requires 'data size', 'source device uses synchronization', and 'source device ID' configurations from the control register. The values of the control register are stored for later used.

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 source address will be added to the 'data size' and the new value is 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' configuration is set, the transfer control updates the counter register by reading the Count register, decrementing it and writing the new counter value back to the register unit. If the new counter value is zero,

the channel disable is also performed when the counter 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. 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 (depending on the corresponding bit of IRQ Mask register) the processor could be interrupted before the data transfer is finished, since the operation performed within the DMA controller (writing new register value to the register unit) will be faster than the data transfer performed with the bus (read from source device to DMA controller and write from DMA controller to destination device). 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 Stop Transfer Operation

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

The stop transfer operation is performed when the values of counter equals zero after updating. The transfer control unit writes a disable channel value to the control register. Since the only configuration bit that needs to be changed is the enable bit, writing to the control register will automatically set the enable bit to its 'disable' value.

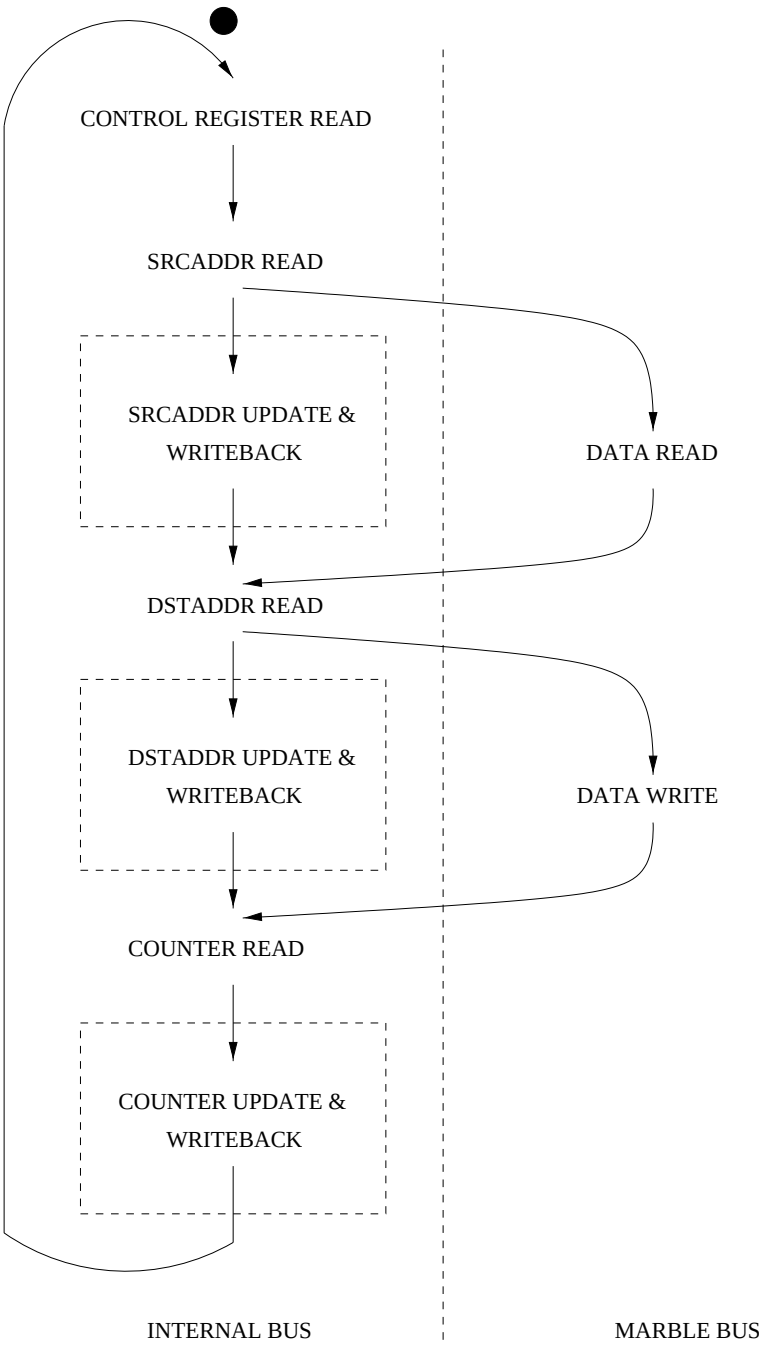


Figure 5.3: Register communication with narrow bus

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 channel's control 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 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 ChanStatus or IRQRequest registers; the ChanStatus register 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 reset.

5.4 Shared resources in the DMA controller

To perform DMA transfer functions, the DMA controller needs to act as both initiator device and 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 acts as a target device which receives read/write requests from the processor. When the DMA controller performs a data transfer, it initiates read/write requests to source/destination devices to transfer data. In these operations the DMA controller acts as an initiator device.

In both cases, the DMA registers are accessed by either 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 handling 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. It allows two units to perform read operations with 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 the probability of the registers being requested by both units simultaneously is low, so it is not necessary to perform these operations in parallel; one unit could wait until the first one finishes before it gets its turn without much effect on the performance of the DMA controller. This method also needs quite a large area of silicon to implement when compared with other methods.

5.4.2 Registers locking

When a DMA channel is programmed and enabled by the processor, the channel registers could be locked by the DMA controller using a flag which can be read by the processor, 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 that channel's registers again. This mechanism allows both 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 has 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 kind of problem in the asynchronous logic design, arbitration is the favourite mechanism. 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, a mechanism to handle this is necessary. For example if a resource C is shared by A and B, if A sends a request before B, A should be granted access and B must wait until A finishes its operation with the shared resource before B get its turn to access.

In a synchronous system requests from A and B could be considered sent simultaneously if they are received in the same clock cycle, and prioritization or other mechanisms, such as round-robin, could be used to determine which unit should gain access first. These methods are possible on the 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 access 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 can lead to non-deterministic behaviour, with consequent difficulties in design verification, testing and performance prediction. The number of arbiters should therefore be

minimized, however these 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 the MARBLE can service only one device at a time, an arbitrary decision as to which device should be served first must be made. The MARBLE bus uses a central arbitration to handle this kind of 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, other registers would be unavailable to the other unit. On the other hand each DMA register could be treated as separated unit, access to a register is independent to others. These two methods are extremes of a scheme which divides the registers into arbitrary regions. Separating each register allows the processor and the transfer control unit more free access, but requires arbiters 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, since the transfer control unit uses a wide internal bus which allows it to access all registers in a DMA channel at once, so *channel* based arbitration

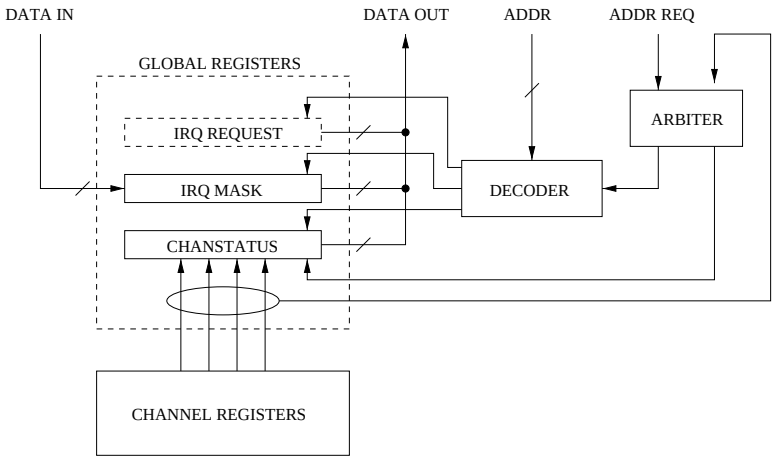


Figure 5.4: Global Registers Arbitration

is used. 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 outlines 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 from the processor and from the channel registers unit.

As shown in figure 5.4, the processor can read/write the global registers via the target interface, and each single bit in 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 'request to transfer' from the DMA channel could arrive at the same time or while the transfer control unit is performing a

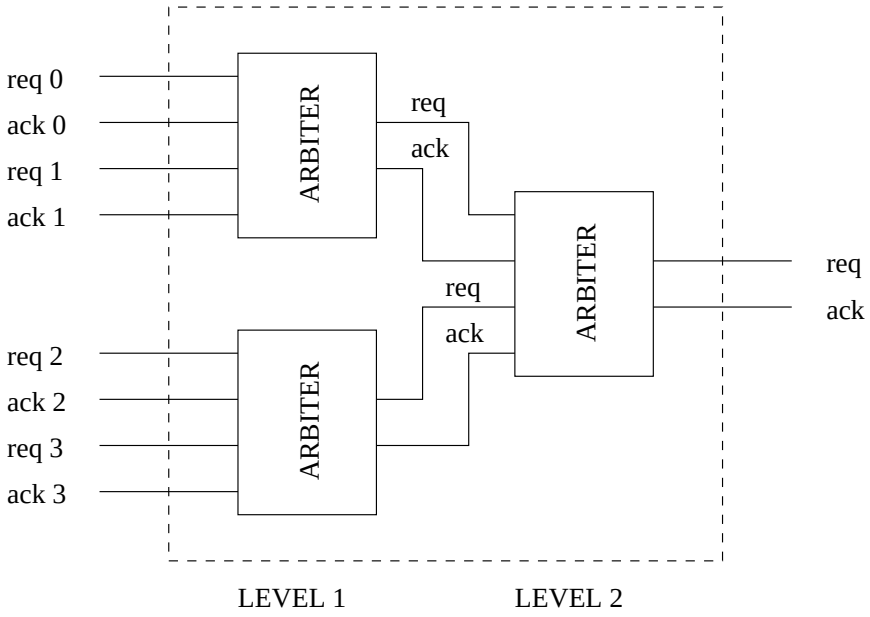


Figure 5.5: Balanced arbitration tree

data transfer for another DMA channel. Arbitration is needed for selecting a DMA channel to transfer data.

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

In the case of a single active request, that request will 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

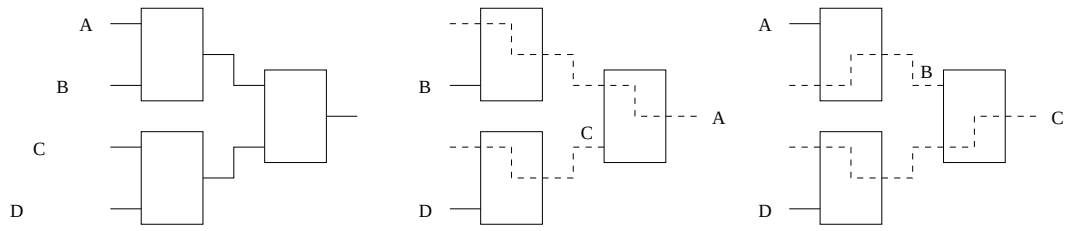


Figure 5.6: Balanced tree order of gaining access

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 other completes it transaction it will be granted before the first request can be reasserted. This means that, if all the inputs are continuously active, each will be guaranteed a share of the shared resource. This share will be equal in the balanced tree.

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 with order of request sending. For example if A,B,C,D contenders 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 their arbiter unit, the order the units gains access will be A,C,B,D instead of A,B,C,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 release 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 to access a resource using balanced tree, number of contenders must be 2^N; for any other number an arbitration tree cannot be built to give the same chance to access 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 prioritization cannot apply to an asynchronous system because

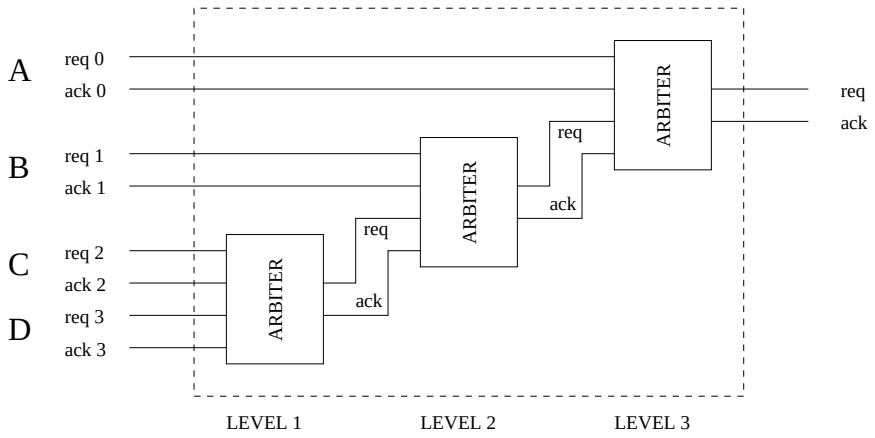


Figure 5.7: Unbalanced arbitration tree

request arrival never occurs simultaneously, if all contenders are always wanting to access the shared resource, i.e. when a contenders 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 normal DMA operation in which DMA transfer is used to transfer data between peripheral device and memory device, the data transfer does not saturate the transfer engine even though more than one channel is used. These arbitration trees 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 occur when more than one channel is programmed to perform data transfer between memory devices in which requests from DMA channels could saturate the transfer engine. However this situation is unlikely because this will saturate the MARBLE bus. In practical it is more appropriate to program the DMA controller to perform memory to memory transfers sequentially.

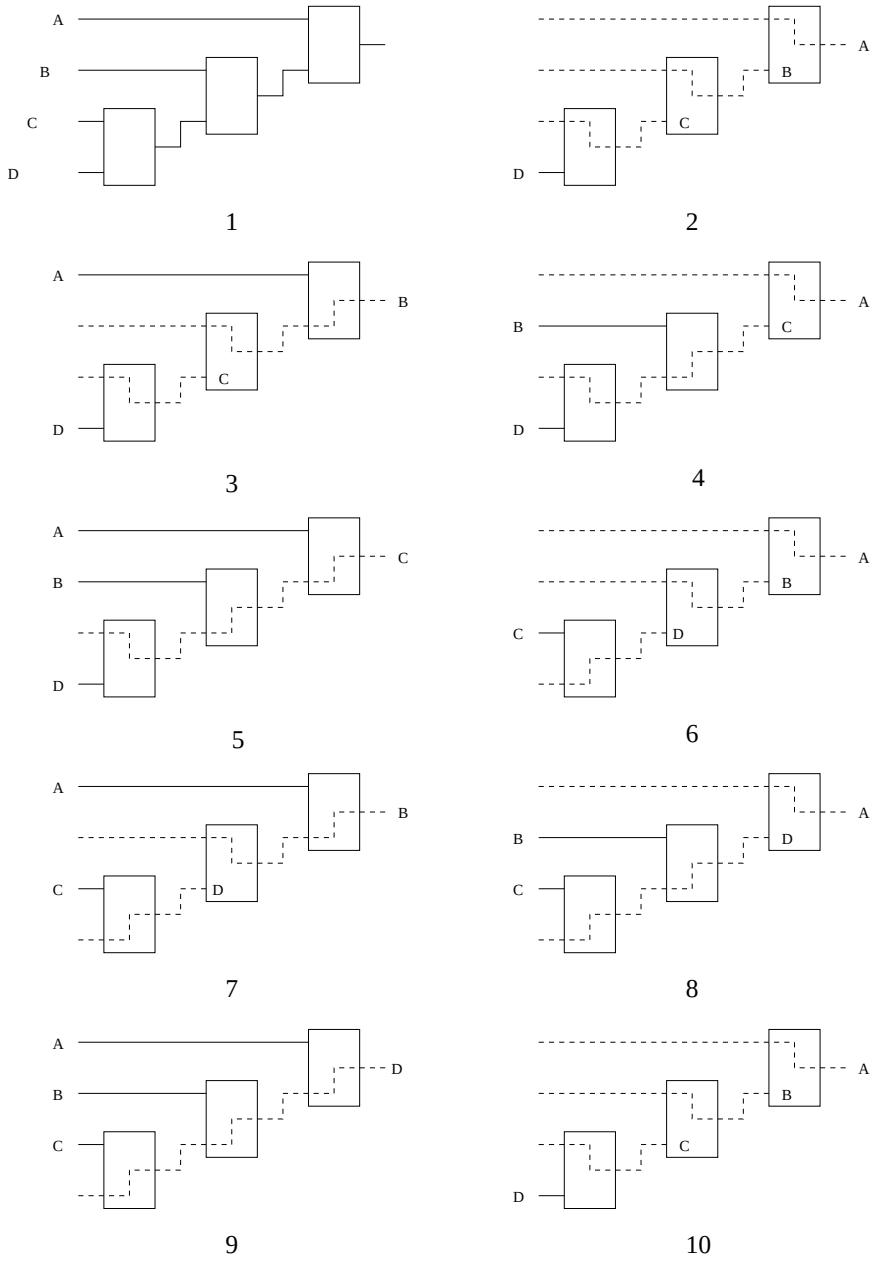


Figure 5.8: Unbalanced arbitration tree

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 in which if the lower bandwidth is programmed for memory to memory transfer, the other channel will gain more chance to be transferred by the transfer engine; since memory to memory transfer is usually faster than peripheral to peripheral or peripheral to memory data transfer.

5.5.4 Arbitrated-Call

The arbitration unit discussed in the section above is not the same arbiter unit described by Sutherland in his Micropiplelines paper [17], but is equivalent to an 'arbitrated-call', the combination between arbiter and call unit. As described in Micropipelines paper :

> "The CALL element remembers which of its inputs most recently received an event, and returns an event to the matching output terminal after a called procedure has finished. The memory in the CALL element serves the role of subroutine return address. The CALL element operates properly only if each call completes before a subsequent call occurs. The ARBITER decides cleanly between two events whose arrival sequence is unknown, producing a grant event for only one of them even if they arrive at very nearly the same time. Like a semaphore in programming, it delays sub-sequent grants until after receiving an event on the done wire corresponding to an earlier grant so that only one grant at a time is ever outstanding. An ARBITER can be connected directly to a CALL element to permit two entirely independent processes to call on a single shared procedure."

It is not stated explicitly in the sections above because this is the only form of the arbitration that is used in this design of the DMA controller.

5.6 Problems

Using arbitration to allow the processor and the transfer engine to access the DMA registers safely, can lead to a deadlock problem. The deadlock problem 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 units wait for other to free the resource they need and both of them will stuck in deadlock.

5.6.1 Deadlock

An example of a deadlock could 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, occupying the bus. If the transfer engine insists on obtaining the bus 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 the registers before requesting the bus. The processor may then complete its cycle, the bus 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.

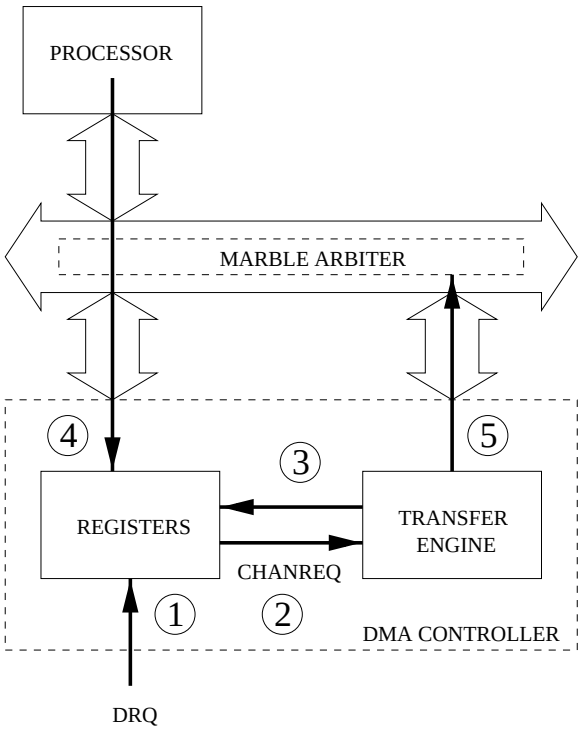


Figure 5.9: System Deadlock Problem

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

5.6.2 Race condition

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. 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 operation completes. In this case the transfer engine could 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 Behavioural Models) has revealed this flaw. It is therefore necessary

to ensure that this achieves the desired result.

To avert this 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 attempt 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 describe in Asynchronous Logic Design chapter); this 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 this design of the DMA controller. The first is the DRQ signal pair for synchronizing 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 synchronize signal to notify 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.

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

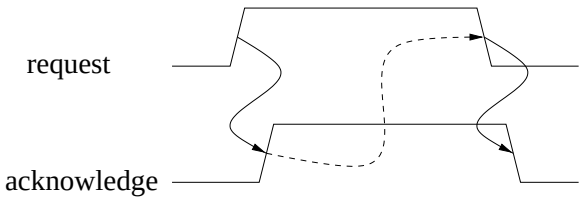


Figure 5.10: Peripheral Request Handshake

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 the DMA transfer 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 could be started.

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 becomes 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.

This form of handshake is also used in the synchronous system [3].

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, the processor reads the DMA 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 is cleared by the processor and the interrupt service routine is responsible for ensuring the IRQ is cleared 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 register causes several other consequent problems. However these problems can be solved, even though these make the design of the DMA controller harder. The arbitration method is more efficient and simpler than others, so it is chosen in this design of the DMA controller.

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 is necessary.

System modelling can be done at many levels of complexity. An abstract model shows the definition and purposes 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 'specifies' how the system behaves and interacts with its environment. The complicated 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 could 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[15], Verilog 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 to model asynchronous systems as well as their synchronous counterparts. 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 and leaves it to be chosen when implementing; this simplifies the models

and reduces the time to develop and maintain the models.

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 by 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 a 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) allow the operations and functions 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 could perform only enough function 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 device, peripheral device, 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: register and transfer engine respectively. Each module has its own interface to the bus, allowing bus communication separately. 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 still not being concerned at this point.

6.4 AMULET3i DMA Controller Model

After the functionality of the simplified model has been tested and proven. The more complicated model that reflected the design of the DMA controller was modelled.

The internal modules of this model comprise:

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

  - Channel Request Mapping

- Transfer Engine
  - Transfer Control
  - Channel Arbiter

In the form of a behavioural model, these modules in the DMA controller are independent modules in LARD, each module communicates 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. Usually the DMA registers can be modelled with various type of variables which represent these values. The registers unit includes these register variables and interfaces to the actual registers for other modules to access. Several decoders (one for the target interface, one for the transfer engine, each group of registers also contains an internal decoder) and arbiters (one for global register, and one for each channel registers) also are incorporated to the registers unit: The decoder is needed to select a particular register, the arbiter allows two or more units that could attempt to access the registers simultaneously to take their turn to access.

6.5.1 Global Registers

The global registers handle read/write request from the target interface and set/reset ChanStatus bit from the channel registers, and also interrupt request to the processor. The operation of the ChanStatus bit set/reset for each bit is

independent, two bits (one by the operation of the transfer control and another one by the target interface) could be set/reset concurrently, so every ChanStatus bit set/reset requests are ORed together as one request to the arbiter of global register.

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.

With the target interface only one single register could be read/written at a time. A flag, 'write_protect_flag', is set to prevent 'race condition' (discussed in previous chapter) when a value is written into a register 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 and the channel request mapping block is activated.

With the transfer engine interface, every register in a channel is 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, if the processor has not intervened. In the write operation, the 'write_protect_flag' is checked before values are written to the registers, if the flag is set those values will be ignored. If the write operation resets the enable bit in the control register, 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

makes an arbitration decision choosing a channel to be served by the transfer engine if there is more than one request. Simple two input arbiters are used in a tree to create an n-input arbiter.

6.6.1 Transfer Control Module

The transfer control module performs data transfer with MARBLE using the channel number given by the channel arbiter; it reads/writes channel registers from the register unit to update 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 in the 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. For simulation the non-deterministic behaviour this could not reflect the real behaviour of the circuit.

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 as 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 exist.

Test for any combination of type of data transfer:

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

has been done with every single DMA channel with both the 'free run' and a definite number of data items transfer. During the 'definite' amount of data items transfer, the operation is also interrupted and the channel is reprogrammed by the processor to find deadlock or other problems. Multiple channels concurrently also had been tested with 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 is working as expected. However testing revealed some problems with initial design – deadlock and more subtle problems : race condition, and early interrupt request. These were subsequently corrected and verified.

6.8.1 Deadlock

The deadlock problem is an obvious one when compared with other problems in the design. It could be figured out easily even without the simulation. As shown in figure 6.1; 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 could cause deadlock if the processor tries to access the same register; it occupies the bus while waiting the transfer engine to release the register, the transfer engine waiting for the processor to release the

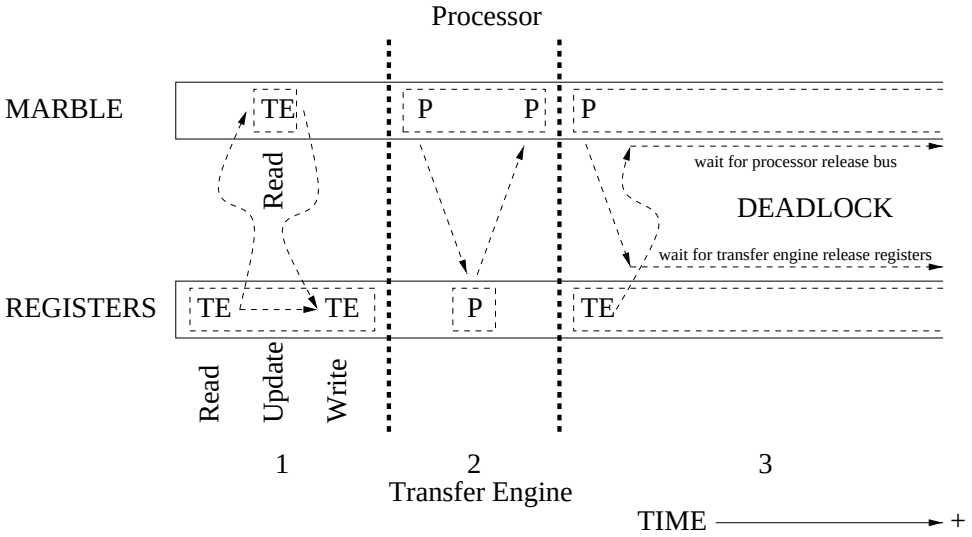


Figure 6.1: Deadlock cause by atomic register access by transfer engine

bus to complete its operation with the register. While the processor requires bus access before it can access the register, the transfer engine does not need to access register while it performs data transfer with bus. So this problem could be easily solved by not allowing the transfer engine to use atomic access to the register unit, as shown in figure 6.2. The transfer engine requests access separately for each operation it performs with the register.

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 actually finished, as shown in figure 6.3. Even though in practice, this is not a real problem because the transfer should be finished before the processor performs any function specified in its interrupt service routine. However to ensures that a problem cannot be caused by this effect, for the last data transfer the register write back operation will be delayed until data transfer with MARBLE is finished, as shown in figure 6.4.

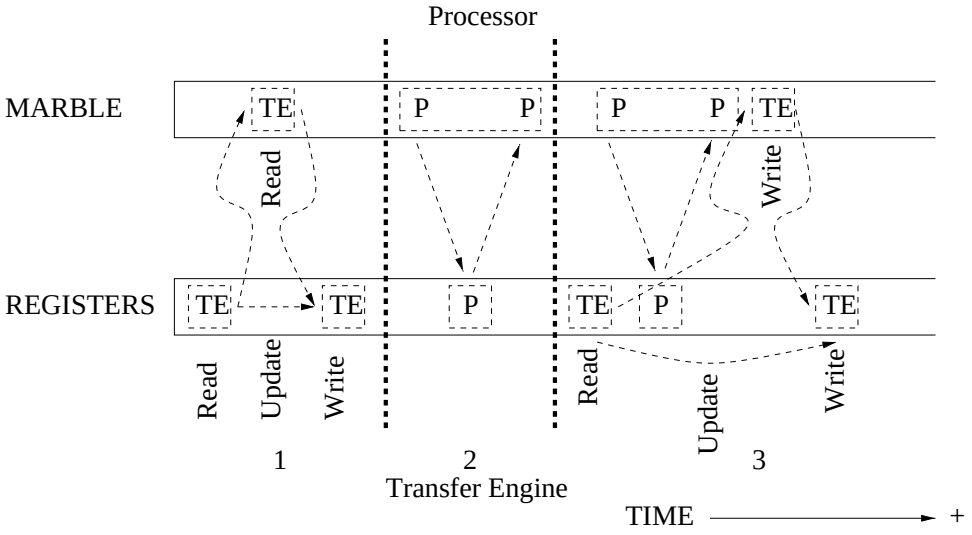


Figure 6.2: Non-atomic register access solve the deadlock problem

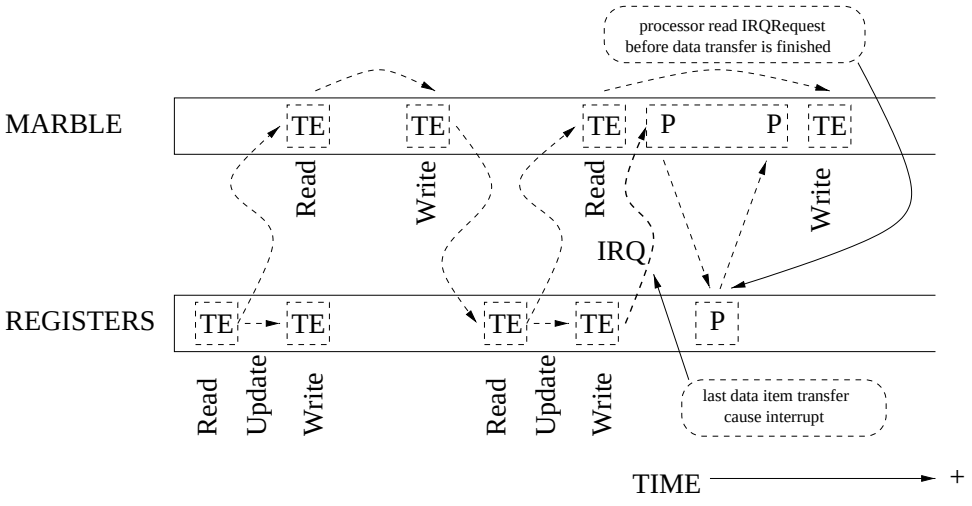


Figure 6.3: Early interrupt sending

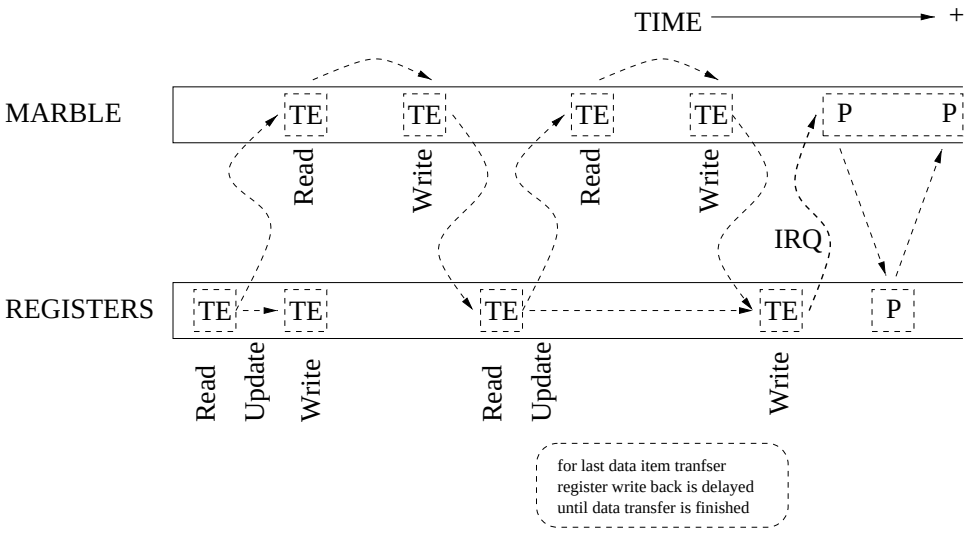


Figure 6.4: Delay registers write back for last data transfer

6.8.3 Race Condition

In normal circumstances the race condition hardly occurs as described before in previous chapter. This problem was found when the solution for early interrupt request was introduced and the simulation was done with peripheral to peripheral data transfer. The data transfer was interrupted and the same channel was reprogrammed to perform the next transfer exactly after the last data transfer request was sent to the transfer engine. Since this was the last data transfer, the transfer control waited for the data transfer to be completed before register values were written back to the register. There is a timing window for the processor to interrupt the transfer and program a register before the transfer control writes back register values to the register unit, and over writes the value written by the processor, as shown in figure 6.5.

This problem can be solved by adding a flag, which prevents the transfer control over writing register values if the process has written some value to the register unit (and set flag by doing that), as shown in figure 6.6.

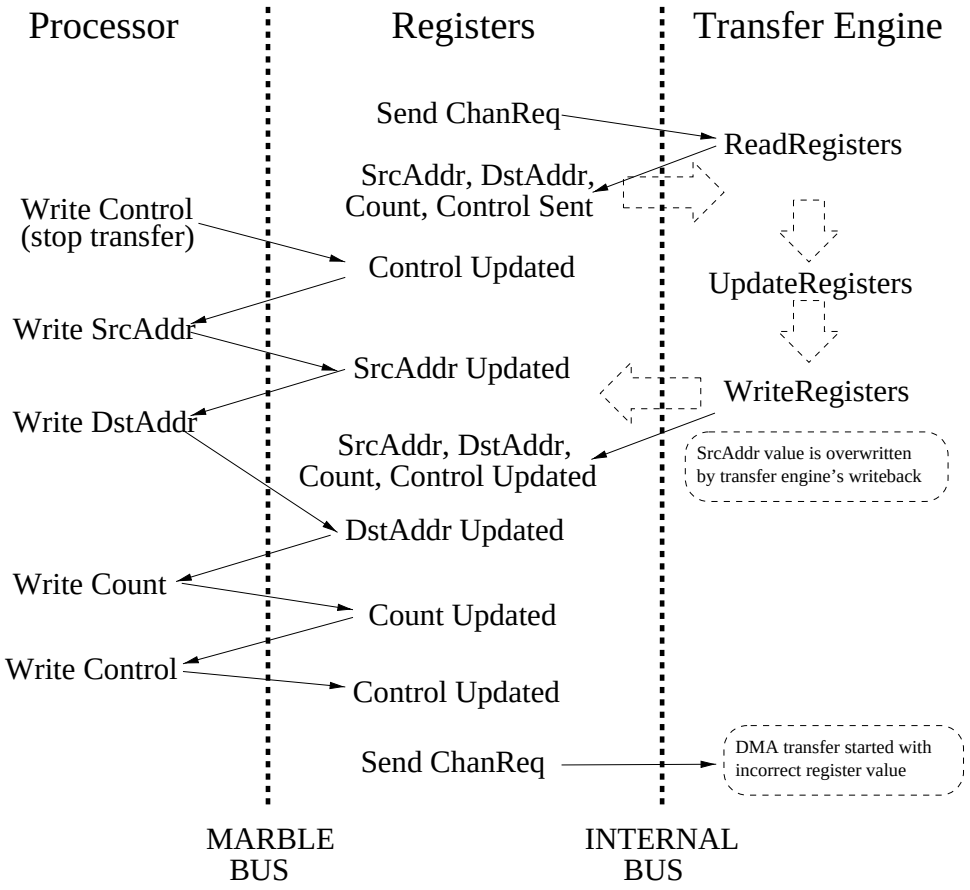


Figure 6.5: Race Condition

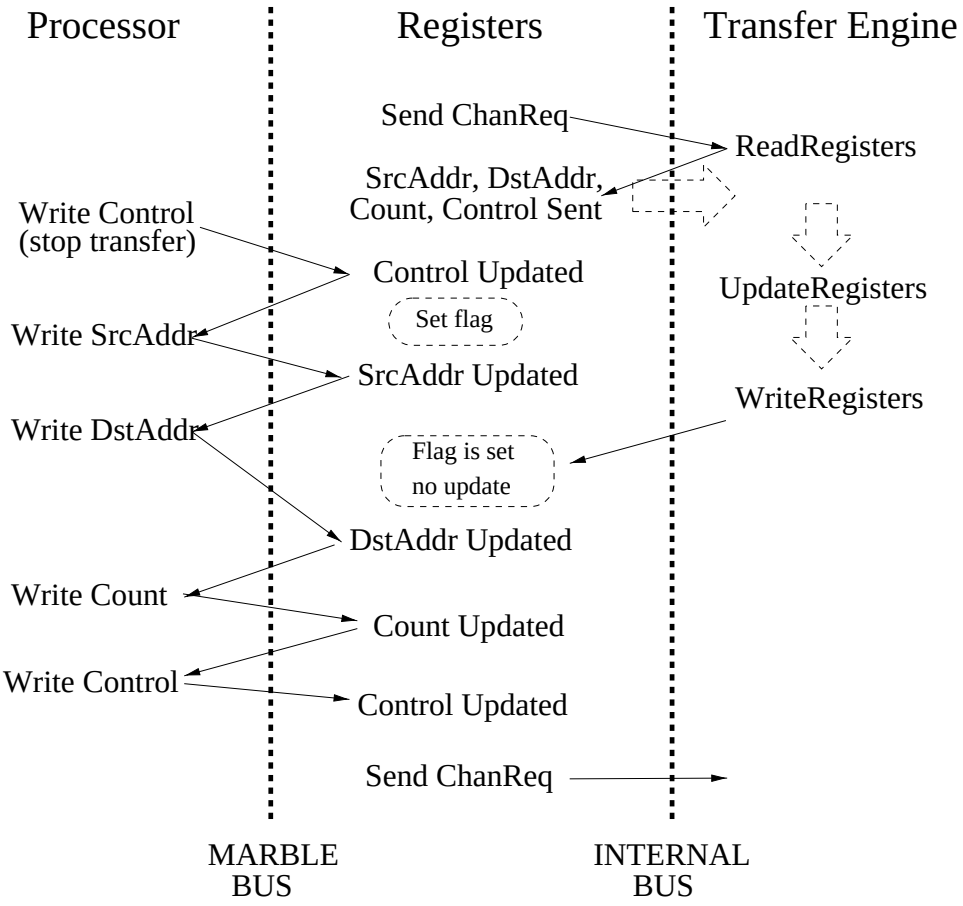


Figure 6.6: Using a flag to prevent Race Condition

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

Chapter 7

Conclusions

Design of an asynchronous DMA controller is possible. Even though there are several problems in the design that do not exist in the synchronous counterpart several techniques could be used to solve the problem. In this thesis, the real circuit of the DMA controller is not achieved, 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 which is enough to prove that it works.

7.1 The AMULET3i DMA Controller

The DMA controller which has been designed is a multi-channel DMA controller which could be used with 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 overall 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

An asynchronous DMA controller is feasible to design and use 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 can gives problems in 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 bus, fly-past transfer could increase the DMA transfer performance

in some situations, and it doesn't requires major changes in the DMA controller to support this function; only a part of the transfer control that needed to re-design. 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 the "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 turned down 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 request, 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 hardware description language and simulation tools provided, modelling the system and simulating it is much more easy.

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 support parallel scheduling, but the interpreter itself is a single-thread scheduler, modelling a non-deterministic circuit component, such as an arbiter could not be achieved realistically. LARD is intended for high-level modelling; low-level models of circuit components could be done but not very effectively.

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 to co-simulate with other applications, but with 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 of 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] J.W. 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] 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.

[8] S.B. Furber, J.D. Garside, and D.A. Gilbert. Amulet3: A high performance self-timed ARM microprocessor. *ICCD'98*, 1998.

[9] 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.

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

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

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

[13] 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. 1997.

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

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

[16] 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.

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

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