DESIGN AND IMPLEMENTATION OF DIRECT MEMORY ACCESS CONTROLLER

A Project Report

submitted by

N SASIDHAR

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **DESIGN AND IMPLEMENTATION OF DIRECT MEMORY ACCESS CONTROLLER**, submitted by **N Sasidhar**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr V. Kamakoti
Research Guide
Professor
Department of Computer Science
and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 8th May 2017

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards several people who enabled me to reach this far with their timely guidance, support and motivation.

First and foremost, I offer my earnest gratitude to my guide, Dr. V. Kamakoti whose knowledge and dedication has inspired me to work efficiently on the project and I thank him for motivating me, allowing me freedom and flexibility while working on the project.

I would like to thank my co-guide Dr.Nitin Chandrachoodan and faculty advisor Dr.Shreepad Karmalkar, who have guided me through out the program.

My special thanks and deepest gratitude to Rahul Bodduna who has been very supportive. He has enriched the project experience with his active participation and invaluable suggestions.

My thanks goes to my fellow labmates Zaid,Debpratim,Vishvesh and Arjun for their help and support.

ABSTRACT

KEYWORDS: Direct Memory Access (DMA), Advanced Microcontroller Bus Architecture (AMBA), Advanced eXtensible Interface(AXI), Advanced Peripheral Bus (APB).

As the two essential modes of data transfer, Programmed I/O and Interrupt I/O involves Central Processing Unit (CPU), the time taken for the data transfer is large. So, there is a need for mode of transfer which does not involve the CPU. Direct Memory Access (DMA) allows the input and output devices to read/write data from the main memory without the interference of CPU. The DMA controller, controlled by the CPU, handles the data transfer in this mode of transfer. With DMA, the CPU first initiates the data transfer, then does other operations while the transfer is in progress. When the transfer is complete, the DMA controller sends an interrupt to signal the end of transfer. This optimizes the data transfer allowing high speed transfer of large blocks of data.

This thesis describes the design and implementation of Direct Memory Access (DMA) controller and integrating it in a System on Chip. The design is based on ARM Corelink DMA Controller DMA-330. DMAC is an AMBA compliant peripheral.It is compliant with AXI and APB protocols. The code for the entire project is written in a HDL namely Bluespec System Verilog (BSV).Most of the parameters in the design are configurable which makes it flexible.DMAC provides AXI master interface to perform the DMA transfers and APB slave interface through which a processor can control the operation of DMA Controller by accessing the control registers. It includes a small instruction set with variable lengths.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DMAC	Direct Memory Access Controller
AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
APB	Advanced Peripheral Bus
TLM	Transaction Level Modeling
HDL	Hardware description language
BSV	Bluespec System Verilog
CLB	Configurable Logic Block

CHAPTER 1

Introduction

1.1 About the DMAC

The Direct Memory Access Controller (DMAC) is a hardware feature that enables movement of blocks of data from peripheral-to-memory, memory-to-peripheral or memory-to-memory. This movement of data by a separate entity significantly reduces the load on the processor.

DMA Controller is an AMBA(Advanced Microcontroller Bus Architecture) compliant peripheral.It is compliant with AXI3(Advanced eXtensible Interface) and APB3(Advanced Peripheral Bus) protocols.

DMAC provides one AXI master interface to perform the DMA transfers and two APB slave interfaces through which a processor can control the operation of DMA Controller. It includes a small instruction set with variable lengths.
A example system with DMAC and its interfacing with peripherals,memory and processor is shown below:

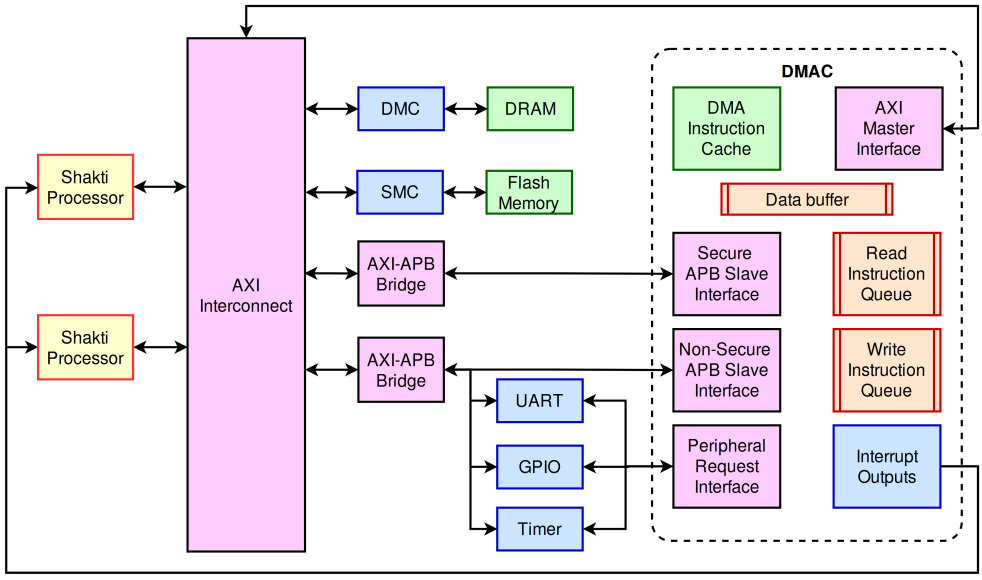

Figure 1.1: Example system

The example system contains:

- AXI bus masters:
  - A DMA Controller
  - Two Shakti processors.
- An AXI interconnect and two AMBA protocol bridge components.
- AMBA slaves:
  - A Dynamic Memory Controller (DMC).
  - A Static Memory Controller (SMC).
  - A Timer.
  - A General Purpose Input-Output (GPIO).
  - A Universal Asynchronous Receiver-Transmitter (UART).

The AXI interconnect enables each bus master to access the slaves.Shakti processors can access the APB interfaces by using appropriate AXI-APB bridge.

1.2 Features

The DMAC provides the following features:

- ARM CoreLink DMA-330 based instruction set.
- Parametrized Peripheral request interface.
- DMA Manager thread to initialize dma transfer(channel thread).
- Parametrized DMA instruction cache.
- Supports multiple transfer types:
  - Memory-to-memory
  - Memory-to-peripheral
  - Peripheral-to-memory

1.3 Configurable Options

We can configure the following parameters:

1. AXI bus width.
2. Number of active AXI read transactions.
3. Number of active AXI write transactions.
4. Number of DMA channels.
5. Depth of the internal data buffer.
6. Number of lines in the instruction cache and how many words a line contains.
7. Number of peripheral request interfaces.
8. Request acceptance capability of a peripheral request interface.
9. Number of interrupt output signals.

CHAPTER 2

Functional overview

2.1 Overview

Block diagram of the DMA Controller design is shown below:

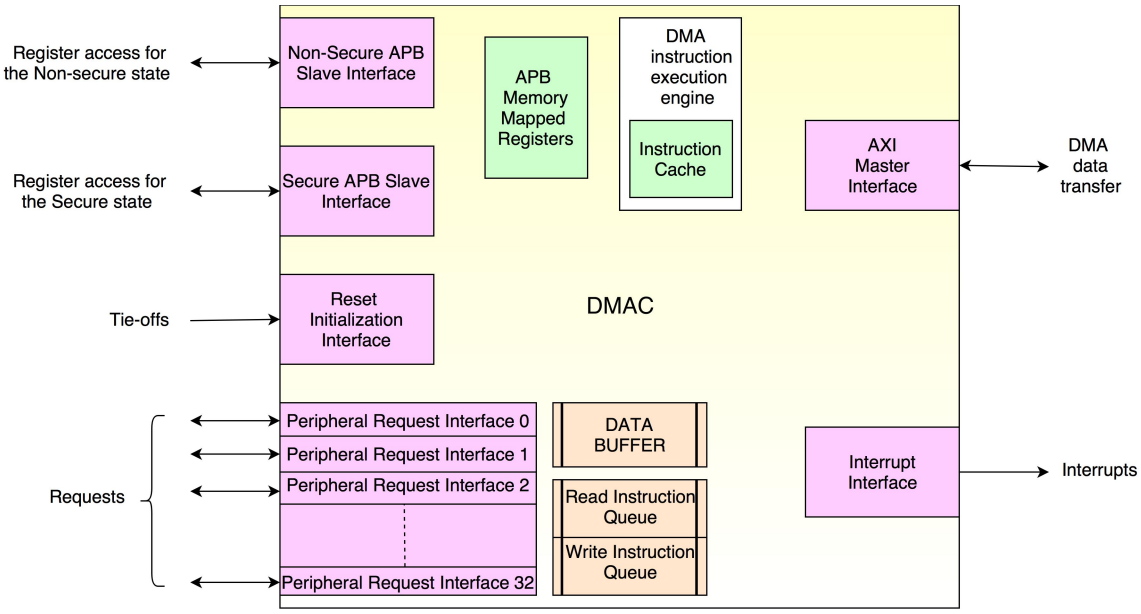

Figure 2.1: Block diagram of DMA Controller

DMA Controller contains a configurable Multi First-In-First-Out(MFIFO) data buffer to store the read and write data for dma data transfer.It includes a read instruction and write instruction queues to store the load and store instructions.A configurable internal instruction cache and an instruction processing block are available to process the variable-length instructions that consists of one to six bytes.

A DMAC contains a single manager thread to initialize channel threads.Peripheral operations like data transfer happens through a channel.Number of channels are configurable and each channel is capable of supporting a single concurrent thread of DMA operation.Each channel has a separate program counter which directs to

system memory. The DMAC executes upto one instruction for each AXI clock cycle. To ensure that it regularly executes each active thread, it alternates by processing the DMA manager thread and then a DMA channel thread.The program code for manager thread and channel thread are stored in system memory which can be accessed by DMAC by using AXI interface.

The DMAC provides multiple interrupt outputs to enable efficient communication of events to external microprocessors. The peripheral request interfaces support the connection of DMA-capable peripherals to enable data transfer to occur, without intervention from a microprocessor.

Dual APB interfaces enable the operation of the DMAC to be partitioned into the Secure state and Non-secure state. You can use the APB interfaces to access status registers and also directly execute instructions in the DMAC.Interfaces will be explained in detail in further sections.

2.2 Operating states

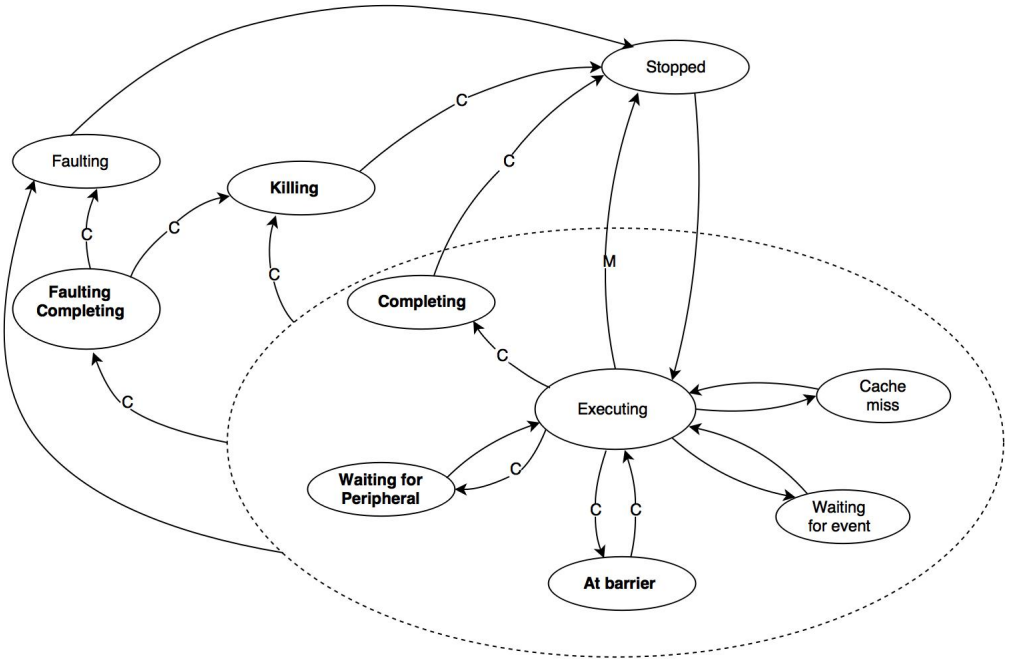

Figure 2.2: Operating states of DMAC

Figure 2.2 shows the operating states of manager thread and channel threads.Each thread has a separate state machine.

In figure 2.2,the DMAC permits that:

- Only DMA channel threads can use states in bold italics.
- Arcs with no letter designator indicate state transitions for the DMA manager and DMA channel threads, otherwise use is restricted as follows:
  - **C** DMA channel thread only
  - **M** DMA manager thread only
- states within the dotted line can transition to the Faulting completing, Faulting, or Killing states.

After the DMAC exits from reset,it sets all channel threads to stopped state and manager state is controlled by **boot_from_pc** signal.If it is **HIGH**,manager moves to Executing state and if it is **LOW**,manager stays in STOPPED state. The following sections describe the states:

2.2.1 Stopped

The thread has an invalid PC and it is not fetching instructions. Depending on the thread type,you can cause the thread to move to the Executing state by:
DMA manager thread: With boot_from_pc HIGH and aresetn LOW,the DMA manager thread moves to the Executing state after aresetn goes HIGH.
DMA Channel thread: Programming the DMA manager thread to execute DMAGO for a DMA channel thread in the Stopped state.

2.2.2 Executing

The thread has a valid PC and therefore the DMAC includes the thread when it arbitrates. The thread can then change to one of the following states under the following conditions:

stopped

When the DMA manager thread executes DMAEND

Cache miss

When the instruction cache does not contain the next instruction for either the DMA manager thread or the DMA channel thread.

Waiting for event

When a thread executes DMAWFE.

At barrier

When a DMA channel thread executes DMAWMB,DMARMB or DMAFLUSHP.

Waiting for peripheral

When a DMA channel thread executes DMAWFP.

Killing

When a DMA channel thread executes DMAKILL.

Faulting completing

For a DMA channel thread either:

- The thread executes an undefined or invalid instruction.
- An AXI bus error occurs during an instruction fetch or data transfer.

Faulting

For a DMA channel thread either:

- The thread executes an undefined or invalid instruction.
- An AXI bus error occurs during an instruction fetch.

For a DMA manager when an abort occurs.

Completing

When a DMA channel thread executes DMAEND.

2.2.3 Cache miss

The thread is stalled and the DMAC is performing a cache line fill. After it completes the cache fill, the thread returns to the Executing state.

2.2.4 Waiting for event

The thread is stalled and is waiting for the DMAC to execute DMASEV using the corresponding event number. After the corresponding event occurs, the thread returns to the Executing state.

2.2.5 At barrier

A DMA channel thread is stalled and the DMAC is waiting for transactions on the AXI bus to complete. After the AXI transactions complete, the thread returns to the Executing state.

2.2.6 Waiting for peripheral

A DMA channel thread is stalled and the DMAC is waiting for the peripheral to provide the requested data. After the peripheral provides the data, the thread returns to the Executing state.

2.2.7 Faulting completing

A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete. After the transactions complete, the thread moves to the Faulting state.

2.2.8 Faulting

The thread is stalled indefinitely. The thread moves to the Stopped state when you use the DBGCMD Register to instruct the DMAC to execute DMAKILL for that thread.

2.2.9 Killing (or) Completing

A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete. After the transactions complete, the thread moves to the Stopped state.

The design is based on ARM Corelink DMA-330 DMAC in which there is a separate state for updating PC.As PC update can be done in executing state,this state is not used in our design.

2.3 Initializing DMAC

2.3.1 Setting the location of the first instruction for the DMAC to execute

Initially when DMAC exit from reset,boot_from_pc signal controls the state of the DMA manager.

If boot_from_pc is HIGH,DMA manager moves to executing state and updates the dma pc register with the value in boot_addr[31:0].Once PC gets updated,DMAC fetches and executes the instruction in DMA PC register.You must ensure that the state of the boot_addr[31:0] bus points to a region in system memory that contains the start address for the DMAC boot program.Usually the first instruction in manager thread(boot program) will be DMAGO which moves the particular channel defined in the instruction to executing state.

If boot_from_pc is LOW,DMA manager thread stays in stopped state.In this case,we have to provide the first instruction by using one of the APB slave interfaces.Most of the time it will be DMAGO while initializing the DMA.Channel

thread contains instructions related to peripherals like wait for peripheral,load and store.For a data transfer to occur,a channel should be in executing state.

2.3.2 Setting security state

A DMAC can be operated in secure and non-secure states.In our design we implemented DMA in a non-secure state in which any processor can interact with the DMAC by both secure and non-secure APB slave interfaces.The security state of DMAC are set by boot_manager_ns(manager),boot_irq_ns(interrupts), boot_periph_ns(peripheral) signals.For non-secure operation,all these signals are set to HIGH.

2.4 DMAC Interfaces

The DMA Controller contains following interfaces:

- Reset initialization interface
- APB slave interface
- Peripheral request interface
- Interrupt interface
- AXI master interface

2.4.1 Reset initialization interface

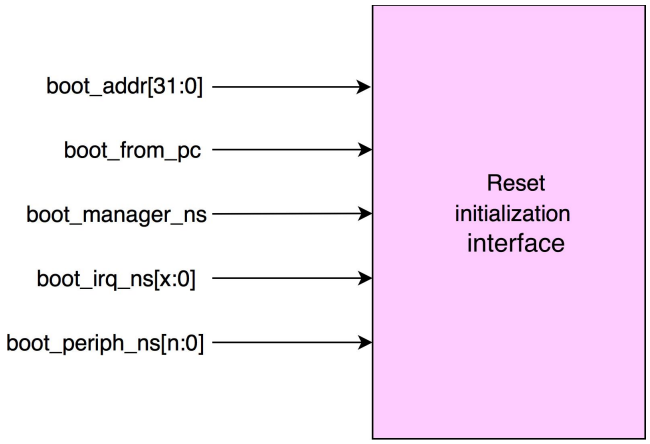

Figure 2.3: Reset initialization interface

This interface is used to initialize the DMAC when it exits from reset.

boot_from_pc controls the initial state of DMA manger.

boot_addr[31:0] gives the starting address of the boot program.DMA PC register should be updated with this address.

boot_manager_ns,**boot_irq_ns[x:0]**,**boot_peripheral_ns[n:0]** are the security states of manager,interrupts and peripheral respectively.HIGH represents non-secure state.

x represents number of interrupts,**n** represents number of peripheral interfaces.

2.4.2 APB slave interface

The DMAC provides secure and non-secure APB interfaces.
Figure 2.4 shows the signal connections for both interfaces.

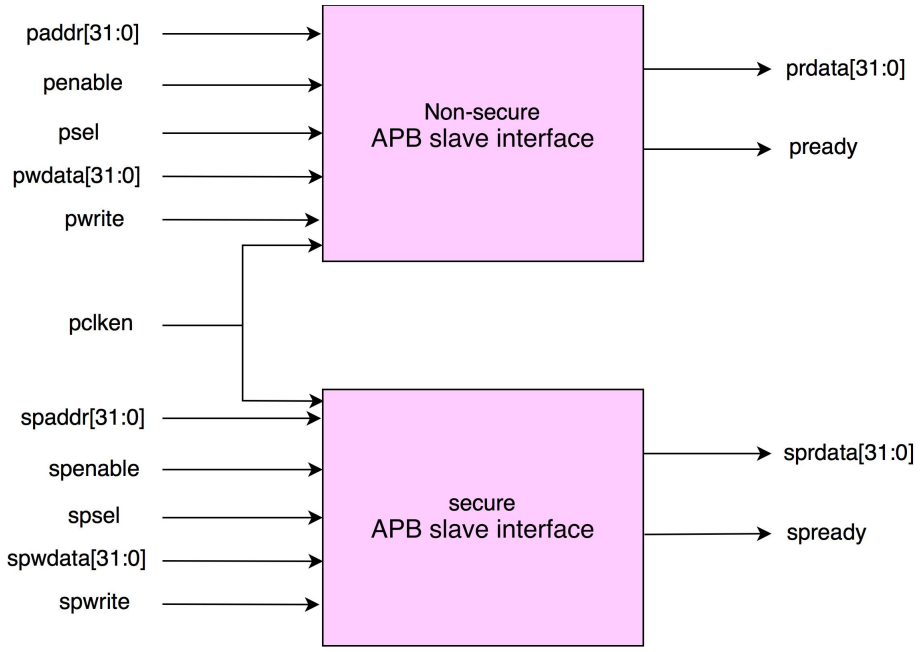

Figure 2.4: APB slave interfaces

The DMAC allocates 4KB of memory for each APB interface.The same clock as the AXI domain clock, aclk, clock the APB interfaces. However, the DMAC provides a clock enable signal, pclken, that enables both APB interfaces to operate at a slower clock rate. The clock enable signal must be an integer divisor of aclk.

APB interface connects the DMAC to the APB and enables a microprocessor to access the registers through which a microprocessor can:

- Access the status of the DMA manager thread.
- Access the status of the DMA channel threads.
- Enable or clear interrupts.
- Enable events.
- Issue an instruction for the DMAC to execute by programming the following debug registers:
  - DBGCMD Register.
  - DBGINST0 Register
  - DBGINST1 Register

Before you can issue instructions using the debug instruction registers or the DBGCMD Register, you must read the DBGSTATUS Register to ensure that debug is idle, otherwise the DMAC ignores the instructions.

When the DMAC is operating in real-time, you can only issue the following limited subset of instructions:

- **DMAGO :** Starts a DMA transaction using a DMA channel that you specify.
- **DMASEV :** Signals the occurrence of an event or interrupt, using an event number that you specify.
- **DMAKILL :** Terminates a thread.

Below example shows the necessary steps to start a DMA channel thread using the debug instruction registers.

1. Create a program for the DMA channel.
2. Store the program in a region of system memory.
   Use one of the APB interfaces on the DMAC to program a DMAGO instruction as follows:
3. Poll the DBGSTATUS Register to ensure that debug is idle, that is, the dbgstatus bit is 0.
4. Write to the DBGINST0 Register and enter the:
   - Instruction byte 0 encoding for DMAGO.
   - Instruction byte 1 encoding for DMAGO .
   - Debug thread bit to 0. This selects the DMA manager thread.
5. Write to the DBGINST1 Register with the DMAGO instruction byte [5:2] data.You must set these four bytes to the address of the first instruction in the program, that was written to system memory in step 2.
6. Writing zero to the DBGCMD Register. The DMAC starts the DMA channel thread and sets the dbgstatus bit to 1.
   After the DMAC completes execution of the instruction, it clears the dbgstatus bit to 0.

2.4.3 Peripheral request interface

Figure 2.5 shows the signals that a single peripheral request interface provides.

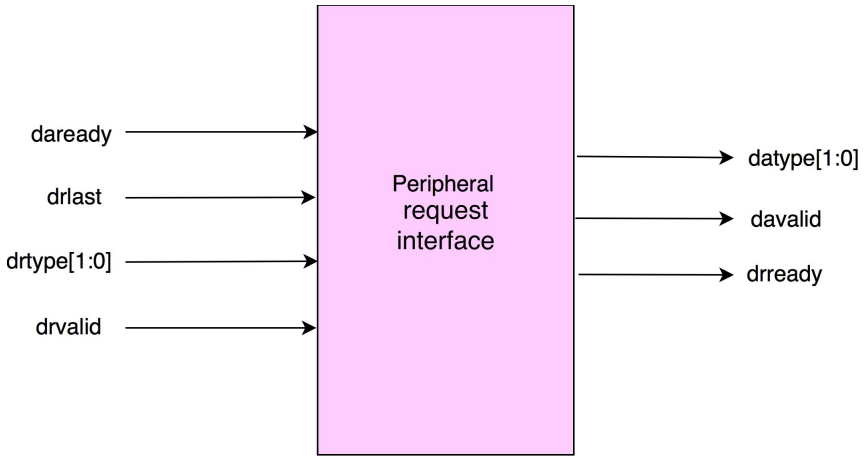

Figure 2.5: Peripheral request interface

We can configure number of peripheral request interfaces.It supports the connection of peripherals to DMA. Description of the signals provided by the interface is shown below:

Peripheral uses **drtype[1:0]** to either:

- Request a single transfer.
- Request a burst transfer.
- Acknowledge a flush request.

drlast is used by peripheral to notify DMA that the request on drtype[31:0] is the last one.Both this signals should be transferred at the same time.
drvalid and **drready** signals are used to notify that a valid request is there by peripheral and DMAC is ready to take the request respectively.
davalid and **daready** signals are used to notify that DMAC sent a valid acknowledgement and peripheral is ready to take the acknowledgement respectively.
valid and **ready**are the handshake signals that AXI protocol describes:
The DMAC uses **datype[1:0]** to either:

- Signal when it completes the requested single transfer.
- Signal when it completes the requested burst transfer.
- Issue a flush request

Mapping to a DMA channel

A peripheral request interface can be connected to any DMA channel.When a channel thread executes DMAWFP instruction,the value mentioned in peripheral[4:0] field of instruction defines the peripheral associated with that channel.

2.4.4 Interrupt interface

Interrupt interface used to notify the processor that data transfer is completed by sending an interrupt.It is basically used for efficient communications of events to an external microprocessor.Number of interrupts are configurable.

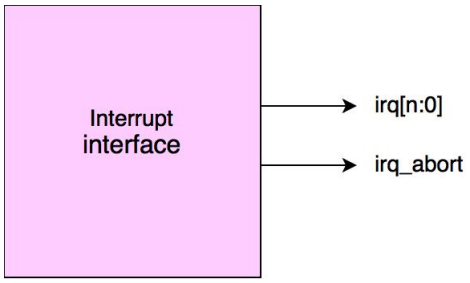

Figure 2.6: Interrupt interface

Figure 2.6 shows signals provide by this interface.**irq[n:0]** and **irqabort** signals generate and abort interrupts respectively.

Using events and interrupts

The number of events and interrupts that the DMAC can support is configurable.After that program the INTEN Register to control if each event-interrupt resource is either an event or an interrupt.
When the DMAC executes a DMASEV instruction,it modifies the event-interrupt resource that we specify.If the INTEN Register sets the event-interrupt resource to be an:

Event

The DMAC generates an event for the specified event-interrupt resource.When the DMAC executes a DMAWFE instruction for the same event-interrupt resource, it clears the event.

Interrupt

The DMAC sets irq<event_num> HIGH, where event_num is the number of the specified event-resource. To clear the interrupt you must write to the INTCLR Register.

Using an event to restart DMA channels:

When we program the INTEN Register to generate an event, you can use the DMASEV and DMAWFE instructions to restart one or more DMA channels.

- **DMAC executes DMAWFE before DMASEV:**

  To restart a single DMA channel:

  1. The first DMA channel executes DMAWFE and then stalls while it waits for the event to occur.
  2. The other DMA channel executes DMASEV using the same event number. This generates an event, and the first DMA channel restarts. The DMAC clears the event, one aclk cycle after it executes DMASEV .

  We can program multiple channels to wait for the same event. For example, if five DMA channels have all executed DMAWFE for event 18, then when another DMA channel executes DMASEV for event 18, the five DMA channels all restart at the same time. The DMAC clears the event, one clock cycle after it executes DMASEV.

- **DMAC executes DMASEV before DMAWFE:**

  If the DMAC executes DMASEV before another channel executes DMAWFE then the event remains pending until the DMAC executes DMAWFE.When the DMAC executes DMAWFE it halts execution for one aclk cycle,clears the event and then continues execution of the channel thread.

Interrupting a microprocessor:

The DMAC provides the **irq[x]** signals for use as active-high level-sensitive interrupts to external microprocessors.When we program the INTEN Register to generate an interrupt,irq[x] will be set HIGH after the DMAC executes DMASEV. An external microprocessor can clear the interrupt by writing to the Interrupt Clear register. If we use DMASEV instruction to notify microprocessor when the DMAC completes DMALD or DMAST,it is recommended to insert a memory barrier instruction before DMASEV.Otherwise the DMAC might signal an interrupt before the AXI transfers complete.

2.4.5 AXI master interface

The DMAC contains a single AXI interface to transfer data from source AXI slave to destination AXI slave.
The AMBA AXI protocol is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed sub micron interconnect.The key features of the AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes
- burst-based transactions with only start address issued
- separate read and write data channels to enable low-cost Direct Memory Access

The AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write data channel to the slave or a read data channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI protocol has an additional write response channel to allow the slave to signal to the master the completion of the write transaction.
Importing AXI package in Bluespec System Verilog provides interface, transactor, module and function definitions to implement the Advanced eXtensible Interface (AXI).The BSV AXI library groups the AXI data and protocols into reusable, parametrized interfaces, which interact with TLM interfaces. An AXI bus is implemented using AXI transactors to connect TLM interfaces on one side with AXI interfaces on the other side. The TLM interfaces used by the Axi package are defined in the TLM package.

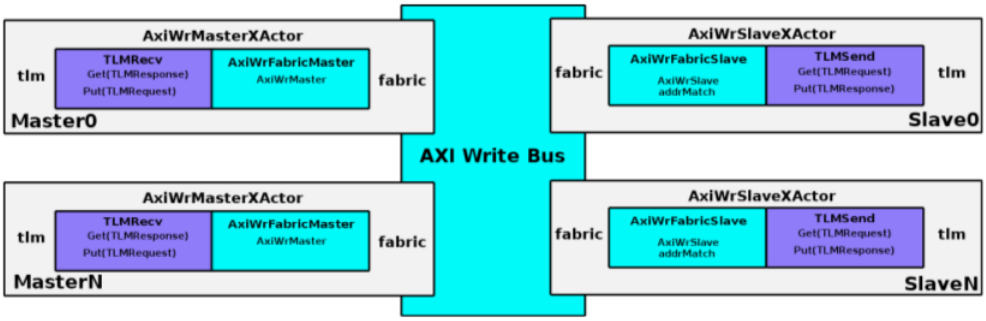

Figure 2.7: BSV AXI Write Bus Implementation Using TLM Transactors

The corresponding BSV AXI implementation is shown in Figure 2.7. TLM Write requests are received via the TLMRecvIFC interfaces of the master transactors. The request is then transmitted via the AxiWrMaster interface out onto the AXI bus and on to the appropriate slave transactor. The slave transactor receives the request via the AxiWrSlave interface, translates the request back into a stream of TLM objects, and then transmits those objects via the TLMSendIFC interface. The TLM response from the write operation follows the same path in reverse.

CHAPTER 3

DMA Controller Registers

3.1 Register summary

DMA Controller contains 32 bit registers and the register map consists of the following sections:

- **Control registers**

  These registers are used to control the DMAC. This include:

  - DMA Status Register
  - DMA Program Counter Register
  - Interrupt Enable,Status and Clear Registers
  - Fault Status and Fault Type Registers

- **DMA Channel thread status registers**

  These include DMA channel status registers and DMA channel Program Counter registers.Each channel will have a separate status and PC registers.

- **AXI and loop counter status registers**

  These registers provide the AXI bus transfer status and the loop counter status for each DMA channel thread. This includes the following registers:

  - Source Address Registers
  - Destination Address Registers
  - Channel Control Registers
  - Loop Counter Registers

  Each channel will have separate AXI and loop counter registers.

- **DMAC debug registers**

  These registers are used to program the DMA controller by processor and enables us to send instructions to a thread when debugging the program code.This includes the following registers:

  - Debug Status Register
  - Debug Command Register
  - Debug Instruction-0 and Instruction-1 registers

- **DMAC configuration registers**

  These registers enable system firmware to discover the configuration of the DMA Controller.This includes:

  - Five Configuration Register(CR0,CR1,CR2,CR3 and CR4)
  - DMA Configuration Register

3.2 Register Description

Description of some registers which are often used in the design are explained in next few sections.

3.2.1 DMA Manager Status Register

It returns information about the status of the DMA manager thread.

Figure 3.1: DMA Status Register bit assignments

Table 3.1: DSR Register bit assignments

Bits	Name	Description
[31:10]	-	Read UNDEFINED
[9]	DNS	Provides the security status of the DMA manager thread: 0 = DMA manager operates in the Secure state 1 = DMA manager operates in the Non-secure state.

Table 3.1 Continued:

Bits	Name	Description
[8:4]	Wakeup_event	When the DMA manager thread executes a DMAWFE instruction, it waits for the following event to occur: 0b00000 = event[0] 0b00001 = event[1] 0b00010 = event[2] . . . 0b11111 = event[32]
[3:0]	DMA status	The operating state of the DMA manager: 0b0000 = Stopped 0b0001 = Executing 0b0010 = Cache miss 0b0011 = Reserved 0b0100 = Waiting for event 0b0101 - 0b1110 = reserved 0b1111 = Faulting.

3.2.2 DMA Program Counter Register

The DPC Register provides the value of the program counter for the DMA manager thread. Figure 3.2 shows the DPC Register bit assignments.

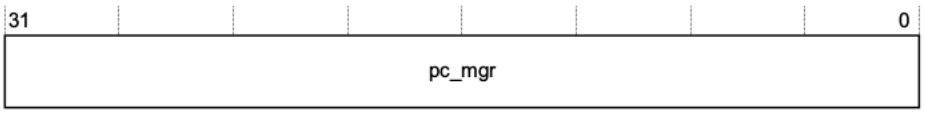

Figure 3.2: DPC Register bit assignments

3.2.3 Interrupt Enable Register

Figure 2.6 shows the INTEN Register bit assignments.

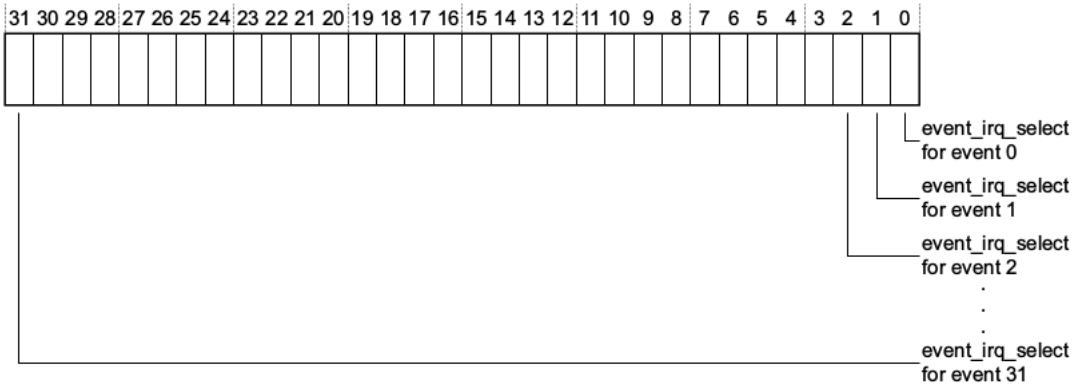

Figure 3.3: INTEN Register bit assignments

Table 3.2 shows the INTEN Register bit assignments.

Table 3.2: INTEN Register bit assignments

Bits	Name	Description
[31:0]	event_irq_select	Program the appropriate bit to control how the DMAC responds when it executes DMASEV : **Bit [N] = 0** If the DMAC executes DMASEV for the event-interrupt resource N then the DMAC signals event N to all of the threads. Set bit [N] to 0 if the system design does not use irq[N] to signal an interrupt request. **Bit [N] = 1** If the DMAC executes DMASEV for the event-interrupt resource N then the DMAC sets irq[N] HIGH. Set bit [N] to 1 if the system design requires irq[N] to signal an interrupt request.

When the DMAC executes a DMASEV instruction, each bit of the INTEN Register controls if the DMAC signals:

- The specified event to all of the threads.
- An interrupt using the corresponding irq.

Event-Interrupt Raw Status Register returns the status of the event-interrupt resources(status of DMASEV)with value 0–inactive and 1–active).

Interrupt Status Register and **Interrupt Clear Register** also works in the same way.

3.2.4 Channel Status Registers

The CSRn Register provides the status of the DMA program on a DMA channel n.
Figure 3.4 shows the CSRn Register bit assignments.

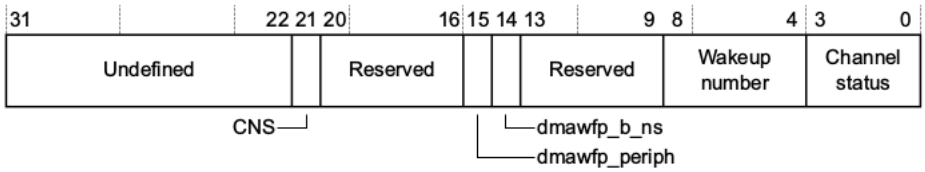

Figure 3.4: CSRn Register bit assignments

Table 3.3 shows the CSRn Register bit assignments.

Table 3.3: CSRn Register bit assignments

Bits	Name	Description
[31:22]	-	READ UNDEFINED.
[21]	CNS	The channel non-secure bit provides the security of the DMA channel with value 0 as secure and 1 as non-secure states.
[20:16]	-	READ UNDEFINED.

Table 3.3 Continued:

Bits	Name	Description
[15]	dmawfp_periph	When the DMA channel thread executes DMAWFP , this bit indicates whether the periph operand was set: 0 = DMAWFP executed with the periph operand not set 1 = DMAWFP executed with the periph operand set.
[14]	dmawfp_b_ns	When the DMA channel thread executes DMAWFP , this bit indicates whether the burst or single operand were set: 0 = DMAWFP executed with the single operand set 1 = DMAWFP executed with the burst operand set.
[31:9]	-	READ UNDEFINED.
[8:4]	Wakeup number	If the DMA channel is in the Waiting for event state, or the Waiting for peripheral state, then these bits indicate the event or peripheral number that the channel is waiting for: 0b00000 = DMA channel is waiting for event, or peripheral, 0 0b00001 = DMA channel is waiting for event, or peripheral, 1 . . 0b11111 = DMA channel is waiting for event, or peripheral, 31.

Table 3.3 Continued:

Bits	Name	Description
[3:0]	Channel status	The channel status encoding is: 0b0000 = Stopped 0b0001 = Executing 0b0010 = Cache miss 0b0011 = reserved 0b0100 = Waiting for event 0b0101 = At barrier 0b0110 = reserved 0b0111 = Waiting for peripheral 0b1000 = Killing 0b1001 = Completing 0b1010 - 0b1101 = reserved 0b1110 = Faulting completing 0b1111 = Faulting.

3.2.5 Channel Program Counter Registers

The CPCn Register provides the value of the program counter for the DMA channel thread.

Figure 3.5 shows the CPCn Register bit assignments.

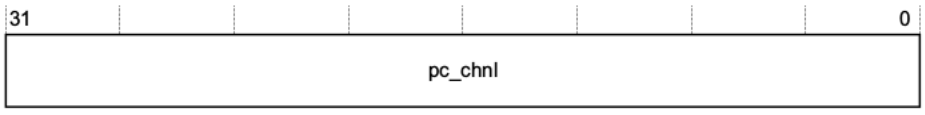

Figure 3.5: CPCn Register bit assignments

3.2.6 Source Address Registers

The SARn Register provides the address of the source data for a DMA channel. The DMAC writes the initial source address value to the SA Register when the DMA channel thread executes a DMAMOV SAR instruction. If a DMAMOV CCR instruction programs the source address to increment, each time the DMA channel executes DMALD, it updates the value to indicate the address that the next DMALD must use.

3.2.7 Destination Address Registers

The DARn Register provides the address for the destination data for a DMA channel.The DMAC writes the initial destination address value to the DA Register when the DMA channel thread executes a DMAMOV DAR instruction.If a subsequent DMAMOV CCR instruction programs the destination address to increment, then each time the DMA channel executes DMAST, it updates the value to indicate the address that the next DMAST must use.

3.2.8 Channel Control Registers

The CCRn Register controls the AXI transactions that the DMAC uses for a DMA channel.The DMAC writes to the corresponding CC Register when a DMA channel thread executes a DMAMOV CCR instruction.
Figure 3.6 shows the CCRn Register bit assignments.

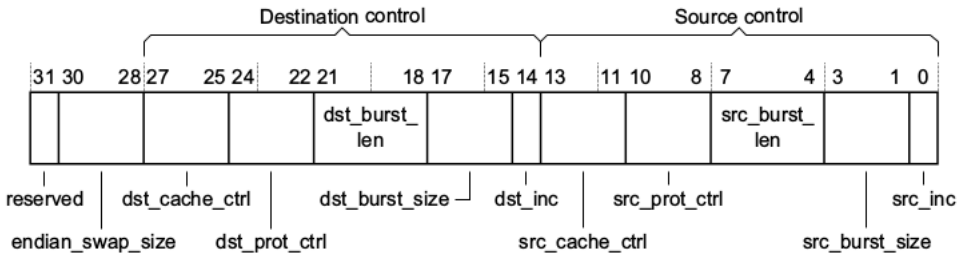

Figure 3.6: CCRn Register bit assignments

Table 3.4 shows the CCRn Register bit assignments.

Table 3.4: CCRn Register bit assignments

Bits	Name	Description
[31]	-	READ UNDEFINED
[27:25]	dst_cache_ctrl	Programs the state of AWCACHE when the DMAC writes the destination data.
[24:22]	dst_prot_ctrl	Programs the state of AWPROT when the DMAC writes the destination data.
[21:18]	dst_burst_len	For each burst,these bits program the number of data transfers that the DMAC performs when it writes the destination data: 0b0000 = 1 data transfer 0b0001 = 2 data transfers 0b0010 = 3 data transfers . . . 0b1111 = 16 data transfers. The total number of bytes that the DMAC writes out of the MFIFO when it executes a DMAST instruction is the product of dst_burst_len and dst_burst_size. These bits control the state of **AWLEN[3:0]**.

Table 3.4 Continued:

Bits	Name	Description
[17:15]	dst_burst_size	For each beat within a burst, it programs the number of bytes that the DMAC writes to the destination: 0b000 = writes 1 byte per beat 0b001 = writes 2 bytes per beat 0b010 = writes 4 bytes per beat 0b011 = writes 8 bytes per beat 0b100 = writes 16 bytes per beat 0b101 - 0b111 = reserved. These bits control the state of **AWSIZE[2:0]**.
[14]	dst_inc	Programs the burst type that the DMAC performs when it writes the destination data: 0 = Fixed-address burst.The DMAC signals AWBURST[0] LOW. 1 = Incrementing-address burst.The DMAC signals AWBURST[0] HIGH.

CCR[13:0] is for source control and is same as destination as described in table.

3.2.9 Debug Status Register

The DBGSTATUS Register provides the debug status of the DMAC.

Figure 3.7 shows the DBGSTATUS Register bit assignments.

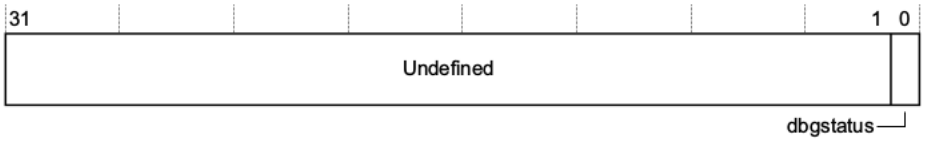

Figure 3.7: DBGSTATUS Register bit assignments

The debug status encoding is:

0 = Idle

1 = Busy

3.2.10 Debug Command Register

The DBGCMD Register Controls the execution of debug commands in the DMAC.Figure shows the DBGCMD Register bit assignments.

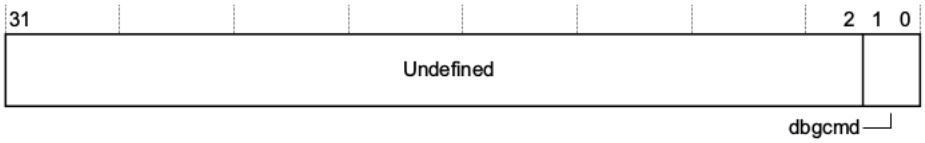

Figure 3.8: DBGCMD Register bit assignments

The dbgcmd[1:0] encoding is as follows:

0b00 = execute the instruction that the DBGINST [1:0] Registers contain

0b01-0b11 = reserved

3.2.11 Debug Instruction-0 Register

The DBGINST0 Register controls the debug instruction, channel,and thread information for the DMAC. Figure 3.9 shows the DBGINST0 Register bit assignments.

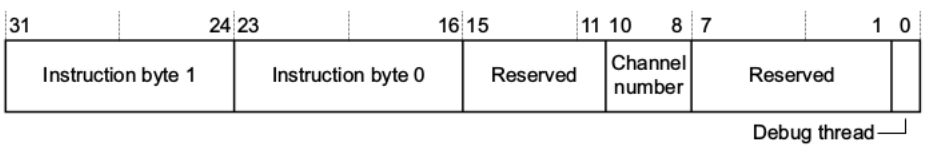

Figure 3.9: DBGINST0 Register bit assignments

Debug thread shows execution of manager(1'b0) or channel(1'b1).If it is channel,the Channel number field selects the DMA channel to debug.As instruction size varies from one to six bytes,**Debug Instruction-1 Register** is used for remaining bytes of debug instruction.

3.2.12 Configuration Registers

Configuration register-0(CR0) provides the status of the tie-off control signals.It contains the following information about the configuration of the DMAC:

- The number of DMA channels that it contains.
- The number of peripheral request interfaces it provides.
- The number of irq signals it provides.

Configuration register-1(CR1) Provides the instruction cache configuration.
Configuration register-2(CR2) Provides the value of the boot address that boot_addr[31:0] configures.
Configuration register-3(CR3) Provides the security state of the event-interrupt resources that are initialized when the DMAC exits from reset.
Configuration register-4(CR4) Provides the security state of the peripheral request interfaces that is initialized when the DMAC exits from reset.
The DMA Configuration Register Provides the configuration of the data buffer, data width, and read and write issuing capability of the DMAC.

CHAPTER 4

Instruction Set

Instruction syntax conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

< >	Any item bracketed by < and > is mandatory.
[ ]	Any item bracketed by [and] is optional.
spaces	Single spaces are used for clarity,to separate items.When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

4.1 Instruction set summary

The DMAC instructions:

- Use a DMA prefix, to provide a unique name-space
- Have 8-bit opcodes that might use a variable data payload of 0, 8, 16, or 32 bits
- Use suffixes that are consistent.

Table shows a summary of the instruction syntax.

Table 4.1: Instruction syntax summary

Mnemonic	**Instruction**	**Thread usage:** • **M = DMA manager** • **C = DMA channel**
DMAADDH	Add Halfword	C
DMAADNH	Add Negative Halfword	C
DMAEND	End	M,C
DMAFLUSHP	Flush and Notify Peripheral	C
DMAGO	Go	M
DMAKILL	Kill	M,C
DMALD	Load	C
DMALDP	Load and Notify Peripheral	C
DMALP	Loop	C
DMALPEND	Loop End	C
DMALPFE	Loop Forever	C
DMAMOV	Move	C
DMANOP	No operation	M,C
DMARMB	Read Memory Barrier	C
DMASEV	Send Event	M,C
DMAST	Store	C
DMASTP	Store and Notify Peripheral	C
DMASTZ	Store Zero	C
DMAWFE	Wait For Event	M,C
DMAWFP	Wait For Peripheral	C
DMAWMB	Write Memory Barrier	C

4.2 Instructions

4.2.1 DMAGO

When the DMA manager executes Go for a DMA channel that is in the Stopped state, it performs the following steps on the DMA channel:

- Moves a 32-bit immediate into the program counter.
- Sets its security state.
- Updates it to the Executing state.

If a DMA channel is not in the Stopped state when the DMA manager executes DMAGO then the DMAC does not execute DMAGO but instead it executes DMANOP. Figure 4.1 shows the instruction encoding.

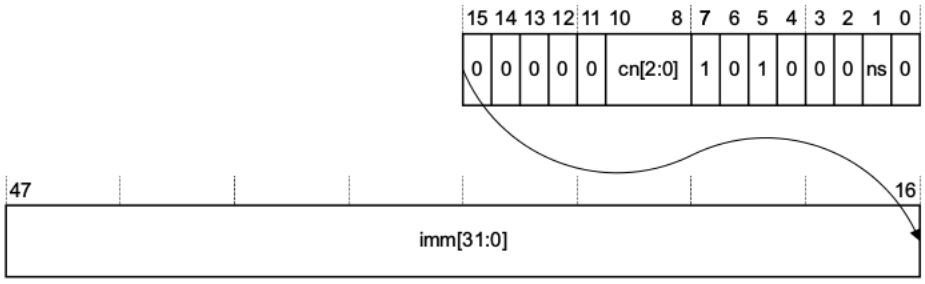

Figure 4.1: DMAGO encoding

Assembler syntax

DMAGO <channel_number>, <32-bit_immediate> [,ns]
where:

<channel_number>	Selects a DMA channel.
<32-bit_immediate>	The immediate value that is written to the CPCn Register for the selected <channel_number>
[ns]	• If ns is present, the DMA channel operates in the Non-secure state. • Otherwise, the execution of the instruction depends on the security state of the DMA manager: **DMA manager is in the Secure state:** DMA channel operates in the Secure state. **DMA manager is in the Non-secure state:** The DMAC aborts.

4.2.2 DMAEND

End signals to the DMAC that the DMA sequence is complete. After all DMA transfers are complete for the DMA channel, the DMAC moves the channel to the Stopped state. It also flushes data from the MFIFO and invalidates all cache entries for the thread.
Figure 4.2 shows the instruction encoding.

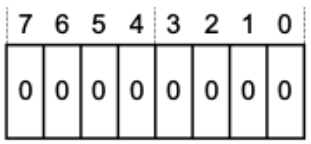

Figure 4.2: DMAEND encoding

Assembler syntax

DMAEND

4.2.3 DMAKILL

Kill instructs the DMAC to immediately terminate execution of a thread. Depending on the thread type, the DMAC performs the following steps:

DMA manager thread

1. Invalidates all cache entries for the DMA manager.
2. Moves the DMA manager to the Stopped state.

DMA channel thread

1. Moves the DMA channel to the Killing state.
2. Waits for AXI transactions, with an ID equal to the DMA channel number,to complete.
3. Invalidates all cache entries for the DMA channel.
4. Remove all entries in the MFIFO for the DMA channel.
5. Remove all entries in the read buffer queue and write buffer queue for the DMA channel.
6. Moves the DMA channel to the Stopped state.

Figure shows the instruction encoding.

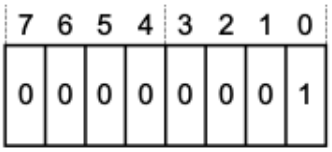

Figure 4.3: DMAKILL encoding

Assembler syntax

DMAKILL

4.2.4 DMAWFP

Wait For Peripheral instructs the DMAC to halt execution of the thread until the specified peripheral signals a DMA request for that DMA channel.
Figure 4.4 shows the instruction encoding.

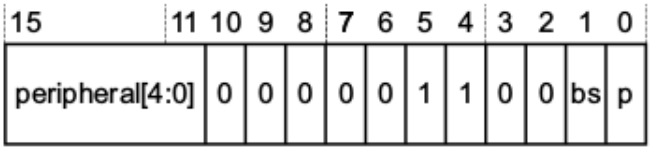

Figure 4.4: DMAWFP encoding

Assembler syntax

DMAWFP <peripheral>, <single|burst|periph>
where:

<peripheral> 5-bit immediate, value 0-31.The DMAC aborts the thread if you select a peripheral number that is not available for the configuration of the DMAC.

<single> Sets bs to 0 and p to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the request_type to Single, for that DMA channel.

<burst> Sets bs to 1 and p to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a burst DMA request. The DMAC sets the request_type to Burst.In this case,the DMAC ignores single burst DMA requests.

<periph> Sets bs to 0 and p to 1. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the request_type to:

Single When it receives a single DMA request.

Burst When it receives a burst DMA request.

4.2.5 DMAMOV

Move instructs the DMAC to move a 32-bit immediate into the following registers:

- Source Address Registers
- Destination Address Registers
- Channel Control Registers

Figure 4.5 shows the instruction encoding.

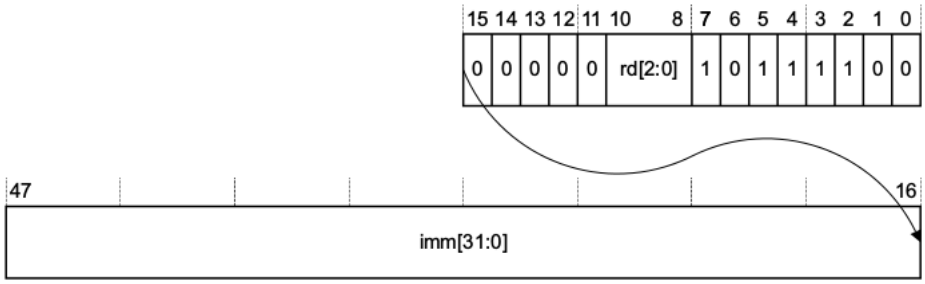

Figure 4.5: DMAMOV encoding

Assembler syntax

DMAMOV <destination_register>, <32 bit_immediate>
where:
<destination_register>

SAR	Selects the Source Address Registers and sets rd to 0b000 .
CCR	Selects the Channel Control Registers and sets rd to 0b001 .
DAR	Selects the Destination Address Registers and sets rd to 0b010 .

<32 bit_immediate>

A 32-bit value that is written to the specified destination register.

4.2.6 DMAWFE

Wait For Event instructs the DMAC to halt execution of the thread until the event, that event_num specifies, occurs. When the event occurs, the thread moves to the Executing state and the DMAC clears the event.
Figure 4.6 shows the instruction encoding.

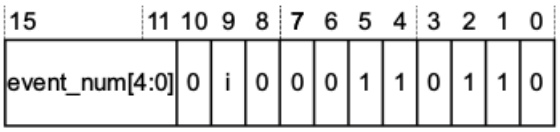

Figure 4.6: DMAWFE encoding

Assembler syntax

DMAWFE <event_num>[, invalid]
where:

<event_num>	5-bit immediate, value 0-31.
[invalid]	Sets i to 1. If invalid is present, the DMAC invalidates the instruction cache for the current DMA thread. If invalid is not present, then the assembler sets i to 0 and the DMAC does not invalidate the instruction cache for the current DMA thread.

4.2.7 DMARMB

Read Memory Barrier forces the DMA channel to wait until all of the executed DMALD instructions for that channel have been issued on the AXI master interface and have completed.This enables write-after-read sequences to the same address location with no hazards.
Figure 4.7 shows the instruction encoding.

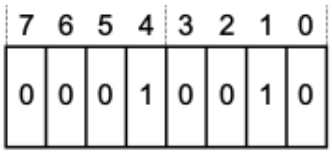

Figure 4.7: DMARMB encoding

Assembler syntax

DMARMB

4.2.8 DMAWMB

Write Memory Barrier forces the DMA channel to wait until all of the executed DMAST instructions for that channel have been issued on the AXI master interface and have completed.This permits read-after-write sequences to the same address location with no hazards.

Figure 4.8 shows the instruction encoding.

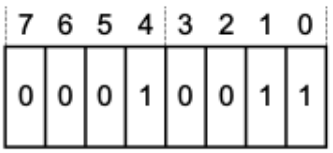

Figure 4.8: DMAWMB encoding

Assembler syntax

DMAWMB

4.2.9 DMASEV

Send Event instructs the DMAC to modify an event-interrupt resource. Depending on how you program the Interrupt Enable Register, this either:

- Generates event <event_num>.
- Signals an interrupt using irq<event_num>.

Figure 4.9 shows the instruction encoding.

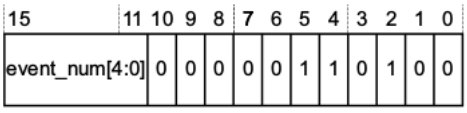

Figure 4.9: DMASEV encoding

Assembler syntax

DMASEV <event_num>

where: <event_num> is a 5-bit immediate value(0-31).

4.2.10 DMALD[S|B]

Load instructs the DMAC to perform a DMA load, using AXI transactions that the Source Address Registers and Channel Control Registers specify. It places the read data into the MFIFO and tags it with the corresponding channel number. DMALD is an unconditional instruction but DMALDS and DMALDB are conditional on the state of the request_type flag. If the src_inc bit in the Channel Control Registers is set to incrementing, the DMAC updates the Source Address Registers after it executes DMALD[S|B] .

The DMAC sets the value of request_type when it executes a DMAWFP instruction. Figure 4.10 shows the instruction encoding.

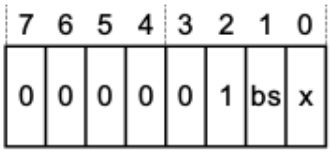

Figure 4.10: DMALD encoding

Assembler syntax

DMALD[S|B]

where:

[S] If S is present, the assembler sets bs to 0 and x to 1. The instruction is conditional on the state of the request_type flag:

request_type = Single

The DMAC performs a DMALD instruction and it sets arlen[3:0]= 0x0 so that the AXI read transaction length is one. The DMAC ignores the value of the src_burst_len field in the Channel Control Registers.

request_type = Burst

The DMAC performs a DMANOP instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.

[B] If B is present, the assembler sets bs to 1 and x to 1. The instruction is conditional on the state of the request_type flag:

request_type = Single

The DMAC performs a DMANOP instruction. The DMAC increments the channel PC to the next instruction. No state change occurs.

request_type = Burst

The DMAC performs a DMALD instruction.

If you do not specify the S or B operand, the assembler sets bs to 0 and x to 0, and the DMAC always executes a DMA load.

4.2.11 DMALDP<S|B>

Load and notify Peripheral instructs the DMAC to perform a DMA load, using AXI transactions that Source Address Registers and Channel Control Registers specify. It places the read data into a FIFO that is tagged with the corresponding

channel number and after it receives the last data item, it updates datype[1:0] to indicate to the peripheral that the data transfer is complete. If the src_inc bit in the Channel Control Registers is set to incrementing, the DMAC updates Source Address Registers after it executes DMALDP<S|B>. Figure 4-7 shows the instruction encoding.

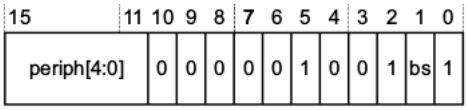

Figure 4.11: DMALDP encoding

Assembler syntax

DMALDP<S|B> <peripheral>
It is same as DMALD with bs = 0 for single and bs = 1 for burst and <peripheral> as a 5-bit immediate value 0-31.

4.2.12 DMAST[S|B]

Store instructs the DMAC to transfer data from the FIFO to the location that the Destination Address Registers specifies, using AXI transactions that the DA Register and Channel Control Register specify. If the dst_inc bit in the Channel Control Registers is set to incrementing, the DMAC updates the Destination Address Registers after it executes DMAST[S|B] .
Figure 4.12 shows the instruction encoding.

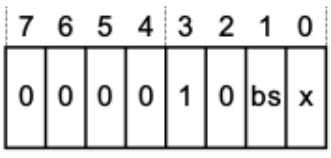

Figure 4.12: DMAST encoding

Assembler syntax

DMAST[S|B]

It is same as DMALD for single and burst transfers with store operation instead of load operation.

4.2.13 DMASTP<S|B>

Store and notify Peripheral instructs the DMAC to transfer data from the FIFO to the location that the Destination Address Registers specifies, using AXI transactions that the DA Register and Channel Control Registers specify. It uses the DMA channel number to access the appropriate location in the FIFO. After the DMA store is complete, and the DMAC has received a buffered write response, it updates datype[1:0] to notify the peripheral that the data transfer is complete. If the dst_inc bit in the Channel Control Registers is set to incrementing, the DMAC updates the Destination Address Registers after it executes DMASTP<S|B> . Figure 4.13 shows the instruction encoding.

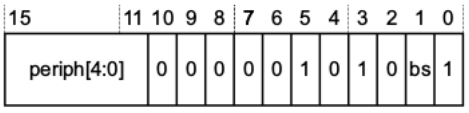

Figure 4.13: DMASTP encoding

Assembler syntax

DMASTP<S|B> <peripheral>

DMASTP is same as DMAST with with bs = 0 for single and bs = 1 for burst and <peripheral> defining the peripheral number.

CHAPTER 5

Design and Synthesis Results

5.1 Design

Design of DMAC is done in bluespec system verilog.Design is based on the FSM with states as mentioned in earlier chapters.Updating PC state in ARM core-link DMAC is not included and instead it is done in executing state itself for both DMA manager and channels.Cache miss state has to be added in the design.A top DMA interface contains all five DMA interfaces.Rules have been written to receive instruction either from APB interface or through DMA manager by using boot_from_pc signal.A set associate cache with configurable options is designed.Whenever a cache miss happens,data has to be fetched from main memory during which other channels can be made active. Instruction decoding is as per the instruction set in DMA-330 controller. This cache module is instantiated in dma module as it is part of the design.Instruction decoding is as per the instructions explained in previous chapter.Instruction fetch and decode are based on the operating states of manager and channel.

A core module is designed with dma module instantiated in it and a TLM send interface is used to connect to main memory.In the same way a TLM memory is used for TLM receive interface.An AXI module is used to connect this TLM send and receive interfaces.This AXI module acts as a top module.Data transfer happens through this AXI interface.

5.2 Synthesis

Synthesis of the design is done in Vivado xcvu095-ffva2104,a virtex ultrascale FPGA evaluation board.

Utilization report after synthesis and implementation are as follows:

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	5209	0	537600	0.97
LUT as Logic	4441	0	537600	0.83
LUT as Memory	768	0	76800	1.00
LUT as Distributed RAM	768	0		
LUT as Shift Register	0	0		
CLB Registers	3753	0	1075200	0.35
Register as Flip Flop	3753	0	1075200	0.35
Register as Latch	0	0	1075200	0.00
CARRY8	28	0	67200	0.04
F7 Muxes	313	0	268800	0.12
F8 Muxes	102	0	134400	0.08
F9 Muxes	0	0	67200	0.00

Figure 5.1: CLB Utilization after synthesis

* The Final LUT count, after physical optimizations and full implementation, is typically lower.

Site Type	Used	Fixed	Available	Util%
CLB LUTs	5193	0	537600	0.97
LUT as Logic	4425	0	537600	0.82
LUT as Memory	768	0	76800	1.00
LUT as Distributed RAM	768	0		
LUT as Shift Register	0	0		
CLB Registers	3753	0	1075200	0.35
Register as Flip Flop	3753	0	1075200	0.35
Register as Latch	0	0	1075200	0.00
CARRY8	28	0	67200	0.04
F7 Muxes	313	0	268800	0.12
F8 Muxes	102	0	134400	0.08
F9 Muxes	0	0	67200	0.00

Figure 5.2: CLB Utilization after Implementation

Timing report of the design is shown below:

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.068ns  (required time - arrival time)
  Source:                 core_channel_status_reg_4_reg[1]/C
                            (rising edge-triggered cell FDRE clocked
                             by CLK  {rise@0.000ns fall@2.400ns period=4.800ns})
  Destination:            core_int_event_ris_reg_reg[4]/D
                            (rising edge-triggered cell FDRE clocked
                             by CLK  {rise@0.000ns fall@2.400ns period=4.800ns})
  Path Group:             CLK
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.800ns  (CLK rise@4.800ns - CLK rise@0.000ns)
  Data Path Delay:        4.628ns  (logic 0.999ns (21.586%)  route 3.629ns (78.414%))
  Logic Levels:           11  (LUT3=1 LUT5=1 LUT6=8 MUXF7=1)
  Clock Path Skew:        -0.128ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.870ns = ( 7.670 - 4.800 )
    Source Clock Delay      (SCD):    3.440ns
    Clock Pessimism Removal (CPR):    0.442ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
  Clock Net Delay (Source):      1.752ns (routing 0.335ns, distribution 1.417ns)
  Clock Net Delay (Destination): 1.547ns (routing 0.309ns, distribution 1.238ns)
```

Figure 5.3: Timing Report

There are multiple register to register paths and the maximum delay path is between channel status register and interrupt status register as in figure 5.3. Above timing report is obtained with a clock of 4.80ns(208.3MHz),which provides a slack of 0.068ns.

The data path delay is 4.628ns which concludes that the design can operate at a maximum frequency of 216MHz.

References

1. CoreLink DMA-330 DMA Controller (Revision:r1p2) Technical Reference Manual
2. AMBA APB Protocol Specification Manual
3. AMBA AXI Protocol Specification Manual