Design and Development of a DMA

A Project Report

submitted by

PRATHAM SONAYYA

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

AND

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **Design and Development of a DMA**, submitted by **Pratham Sonayya**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Masters 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.

Prof. V Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT Madras, 600 036

Prof. Nitin Chandrachoodan
Project Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: June 9, 2022.

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my project guide Prof. V Kamakoti for giving me this opportunity to work on a project under him and SHAKTI lab. I would also like to express my deepest gratitude towards my project mentor Arjun, for his constant support and guidance throughout the course of this project. I would like to acknowledge Prof. Nitin Chandrachoodan for being my co-guide in this project.
Lastly, I would like to thank my parents for providing my with this education and supporting me throughout.

ABSTRACT

KEYWORDS: Direct Memory Access ; ARM DMAC; BURST Transfer;
SINGLE Transfer; SHAKTI

It is imperative for a processor to have an efficient DMA that can take the responsibility of handling the input/output transactions so that the processor can focus on other tasks. The SHAKTI processor has a DMA of its own. However it is has its own limitations that prevents it from catering to requests from a variety of peripherals. This project focuses on looking for a better alternative and implementing the same. Amongst the considered options, we decided to implement the ARM DMAC.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

ARM	Advanced RISC Machines
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
ISA	Instruction Set Architecture
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
API	Application Programming Interface
AXI	Advanced eXtensible Interface
PC	Program Counter
BRAM	Block Random Access Memory
FIFO	First In First Out

CHAPTER 1

INTRODUCTION

This chapter gives a brief introduction of a DMA. It also mentions the limitations of the current SHAKTI DMA and the need for a new and more flexible version.

1.1 General Description of DMA:

Direct memory access (DMA) is the process of transferring data without the involvement of the processor itself. It is often used for transferring data to/from input/output devices. Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller (DMAC) when the operation is done. This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer. A separate DMA controller is required to handle the transfer.

1.2 Working of a DMA:

Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request to the DMA controller. DMA controller accepts this and asks the CPU to hold for a few clock cycles by sending it the Hold request.

CPU receives the Hold request from DMA controller and relinquishes the bus and sends the Hold acknowledgement to DMA controller.

After receiving the Hold acknowledgement, DMA controller acknowledges I/O device that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory.

When the data transfer is accomplished, the DMA raises an interrupt to let the processor know that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left.

1.3 SHAKTI DMA and its limitation:

From the above description we can understand the importance of a DMA for every processor. Since a DMA's major function is to facilitate input/output transactions, it must be compatible with a wide variety of peripherals. The SHAKTI processor currently has a DMA of its own. However, it has a very naive architecture and is not very flexible. A detailed description of the SHAKTI DMA is provided in following chapter. The main limitation is that it will not work for any peripheral that requires polling of certain registers or sending control signals between data transfers. Thus peripherals like PCI express, ethernet, etc. won't be compatible with it.

Hence we need a new DMA, whose architecture will be more flexible and will be able to cater a wider variety of peripherals.

A variety of DMA architectures were studied and browsed. A few of them are described in the future chapters. The one with the best architecture was chosen and implemented for the SHAKTI processor.

CHAPTER 2

Background

2.1 SHAKTI DMA OVERVIEW

This section gives an overview of the current DMA present for the SHAKTI processor. We look into its working, features and architecture.

2.1.1 Introduction:

The direct memory access (DMA) controller is a bus master and system peripheral. DMA is used for data transfer from peripheral to memory and vice versa and also from memory to memory and peripheral to peripheral. This takes the load off the CPU and it can focus on other work.

This DMA features single AHB master architecture.

It comprises of 2 DMAs. Each DMA has 7 channels. Each channel has multiple peripherals linked to it and which peripheral is currently active on the channel is given by CSELR register. Priority among the channels is set by the arbiter and can be set by software (low, med, high) or by hardware (e.g. channel 1 has priority over channel 2).

It can perform Peripheral-to-memory, memory-to-peripheral, memory-to-memory and peripheral-to-peripheral data transfers.

2.1.2 Working:

- The software configures the DMA controller at channel level, in order to perform a block transfer, composed of a sequence of AHB bus transfers.
- Each channel will receive a DMA request from the peripheral of a software trigger request from the memory. The following happens in such a case:
  1. The peripheral sends a single DMA request signal to the DMA controller.
  2. The DMA controller serves the request, depending on the priority of the channel associated to this peripheral request.
  3. As soon as the DMA controller grants the peripheral, an acknowledge is sent to the peripheral by the DMA controller.

4. The peripheral releases its request as soon as it gets the acknowledge from the DMA controller.
5. Once the request is de-asserted by the peripheral, the DMA controller releases the acknowledge.

This is for both peripheral to memory transfer as well as the other way round.

- For a given channel x, a DMA block transfer consists of a repeated sequence of a single DMA transfer, encapsulating two AHB transfers of a single data, over the DMA AHB bus master. These two AHB transfers are -
  1. A single read from peripheral/memory
  2. A single write from memory/peripheral

- The first address of the transfer locations is stored in one of the DMA registers. And the CNDTR register stores the total no. of bytes needs to be transferred. It is post decrementing, so the transfer will happen till the register is NULL.

2.1.3 DMA Arbitration:

- It manages priorities between channels
- Software priority (present in CCRx register):
  – very high
  – high
  – medium
  – low
- Hardware priority: If 2 channels have same priority the lower numbered channel will have preference.
- For memory-to-memory, there will be an arbiter check after every single AHB transfer. If there is a peripheral request it will get preference even if it is on a lower priority channel.

2.1.4 DMA Channels:

Each channel may handle a DMA transfer between a peripheral register located at a fixed address, and a memory address.

Each channel will have a CCRx register that will hold the following info for that particular channel:

- **PSIZE [1:0], MSIZE [1:0]:** sets the size of data it'll have for a single transfer, if these are not aligned then the DMA does transfer according to a set of alignment rules.

- **PINC, MINC:** this is the increment mode, if ON then the address will automatically increment to the next value depending upon the PSIZE and MSIZE mentioned.
- **Circular mode:** If ON then, after the last data transfer, the CNDTRx, CMARx, CPARx registers get automatically reloaded to the initial programmed value. If OFF, the above does not happen. No DMA requests are served. The channel would have to be disabled and then the registers must be reset.
- **DIR:** tells the direction of transfer.
  - DIR = 1: It is memory to peripheral transfer. Hence memory bits -> source, peripheral bits -> destination.
  - DIR = 0: It is peripheral to memory transfer. Hence memory bits -> destination, peripheral bits -> source.

Channel Configuration:

1. Set the peripheral register address in the DMA_CPARx register.
2. Set the memory address in the DMA_CMARx register.
3. Configure the total number of data to transfer in the DMA_CNDTRx register.
4. Set the DMA_CCRx register.
5. Activate the channel by setting the EN bit in the DMA_CCRx register.

Once the channel is configured and enabled, it can then serve any peripheral requests coming to it.

2.1.5 DMA Error Management:

A DMA transfer error is generated when reading from or writing to a reserved address space. In this case the enable bit of the corresponding channel is disabled.

2.1.6 DMA interrupts:

An interrupt can be generated on a half transfer, transfer complete or transfer error for each DMA channel x.
The DMA has interrupt registers (DMA_ISR) whose bits are used to facilitate this for different channels.

2.1.7 DMA Registers:

DMA interrupt status register (DMA_ISR):

It has the four interrupt bits for every channel. If a particular interrupt is occurred, that corresponding bit is set in this register.

DMA interrupt flag clear register (DMA_IFCR):

When a bit in this register is set, it is indication to the software to clear that corresponding interrupt bit in ISR, indicating that interrupt has been handled. Also if the GIFx bit is set then the corresponding GIFx bit of ISR is cleared along with any other interrupt bits that might be ON for that channel.

DMA channel x configuration register (DMA_CCRx):

It is the configuration register for every channel; each element has been already discussed. The TEIE, TCIE, HTIE, bits are for enabling the interrupt system for that channel. If these are OFF, that corresponding interrupt won't be active for that channel.

DMA channel x number of data to transfer register (DMA_CNDTRx):

This gives that amount of data to be transferred in bytes. It will decrement by 1, 2 or 4 depending on the transfer unit being byte, half word or word.

DMA channel x peripheral address register (DMA_CPARx):

To put the start address of the peripheral.

DMA channel x memory address register (DMA_CMARx):

To put the start address of the memory.

DMA channel selection register (DMA_CSELR):

To write out, of all the peripherals mapped to a channel which one is currently mapped.

2.2 uDMA

(An Autonomous I/O Subsystem For IoT End-Nodes)

This paper was on an autonomous I/O Subsystem in which they have introduced a DMA to overcome the bandwidth and power management limitations. In the proposed architecture, a lightweight multi-channel I/O DMA is tightly-coupled to a multi-banked system memory rather than communicating through the system interconnect, allowing to improve the bandwidth by 2.2x with respect to traditional approaches when operating at a given frequency, or saving the same amount of dynamic power leveraging frequency scaling to achieve a given bandwidth.

2.2.1 Introduction

This is a lightweight multi-channel I/O DMA. On one side, the DMA has connections with the peripherals. On the other side, it has 2 dedicated ports connecting to the system interconnect. One of them is the Receive Channel (RX) and the other is the Transmit Channel (TX). These two channels are completely decoupled and not synchronised and hence can act in parallel.

Both the channels have their own FIFO to ensure continuous flow of data.

The software has to program the source or target pointer, the transfer length in bytes

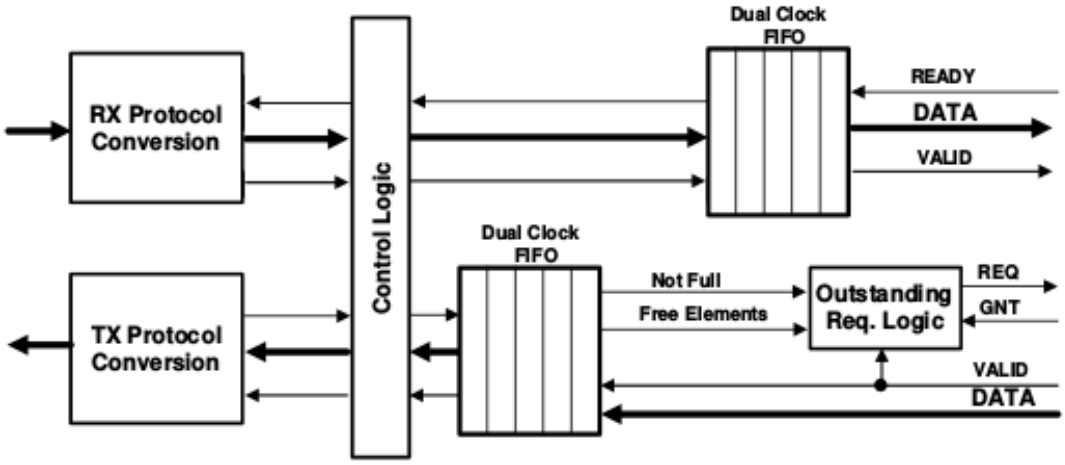

Figure 2.1: uDMA Peripheral Architecture

and send the start command.
At the end of each transaction on each channel the uDMA generates a dedicated event to notify the fabric controller that is possible to queue another transaction.
One main architectural aspect about uDMA that is different from SHAKTI-DMA (section 2.1) is that, the former has a different set of registers for each of its peripheral.

2.2.2 RX Channel:

This channel is used to **write** (only) from peripheral to memory.

All (i.e. from all peripherals) the RX channels share the same port towards the memory.

When a data word is available to be transferred from the peripheral to the memory, the peripheral raises the valid signal and notifies the DMA

The DMA performs an arbitration between all the active channels and acknowledges (with the ready signal) the data transfer to the winning peripheral.

The DMA logic stores the ID of the winning channel among with the data and the data-size of the channel so that it can revisit the same channel for the next data transfer.

The RX channel will have certain registers associated with it for every peripheral. These registers have data like memory address, data, data size, channel ID etc.

All these once selected by the DMA are pushed in the FIFO. On the other side of the FIFO the memory interface logic pops the transfer information, generates the byte enable and performs the transaction to the memory.

The DMA logic is fully pipe-lined and capable of handling 1 transfer per cycle when there is no contention on the memory banks.

2.2.3 TX Channel:

This channel is used to **read** (only) from memory to peripheral.

This channel has separate request and response paths, i.e. the peripheral will first raise a request to read from a particular memory address and then the actual transfer will happen. Since there are two steps so can be pipe-lined.

Even here, in case of multiple requests, the arbiter of the DMA will decide the winning peripheral.

The TX channel will have certain registers associated with it for every peripheral. These registers have data like memory address, data, and data-size, channel ID etc.

So once the winning peripheral is decided, its contents are pushed in the TX FIFO. At the output of the FIFO the memory transaction logic pops an element from the FIFO and performs the memory read. The received data is then sent to the proper channel that is selected by looking at the stored ID.

2.2.4 Scheduler:

The scheduler is a run-to-completion task scheduler with no preemption. This allows using a single stack for all the tasks and avoids storing the saved contexts in the memory.
The scheduler takes the first available task from a FIFO, executes it until it returns, and then continues with the next one, until the FIFO is empty, in which case he goes to sleep.

2.2.5 Drivers interactions with scheduler:

All asynchronous events like end of transfer, re-enqueue a transfer when one has just finished etc. are attached with a task or a handler.
Tasks will be enqueued and executed by the scheduler when the task is scheduled while the handler is executed immediately from the interrupt handler.

The interrupt handler of the DMA is called when the transfer is finished. It sees there is one task attached to the channel of the transfer, and thus enqueues it to the scheduler and leaves.
This task will generally involve setting the conditions for the next transfer.

2.2.6 Results:

The paper mentions that the maximum bandwidth that can be transferred from I/O to the system memory for a system with a single ported system memory, a CPU and a central DMA is equal to 32 × fsys bit/s. With a frequency of 50MHz, this means 1.6Gbit/s.
However, their uDMA can go up to 2.6Gbit/s including all the run-time overheads.
The limit of this uDMA is the amount of data it can transfer in one shot and since it has different registers for each peripheral, adding a new peripheral will entail extra memory overhead. Thus this DMA cannot be used for large systems.

2.2.7 Datasheet:

The data-sheet essentially sums up all the registers associated with the uDMA.
It has a different set of registers for each peripheral, for both TX and RX channels.

CHAPTER 3

ARM DMAC

3.1 Introduction

The DMAC is an Advanced Micro-controller Bus Architecture (AMBA) compliant peripheral.

It has AXI Master-Slave interface, with 2 Slave interfaces- one for secure state and one for non-secure state. It has a programmable security state for each DMA channel.

The DMAC includes a small instruction set that provides a flexible method of specifying the DMA operations.

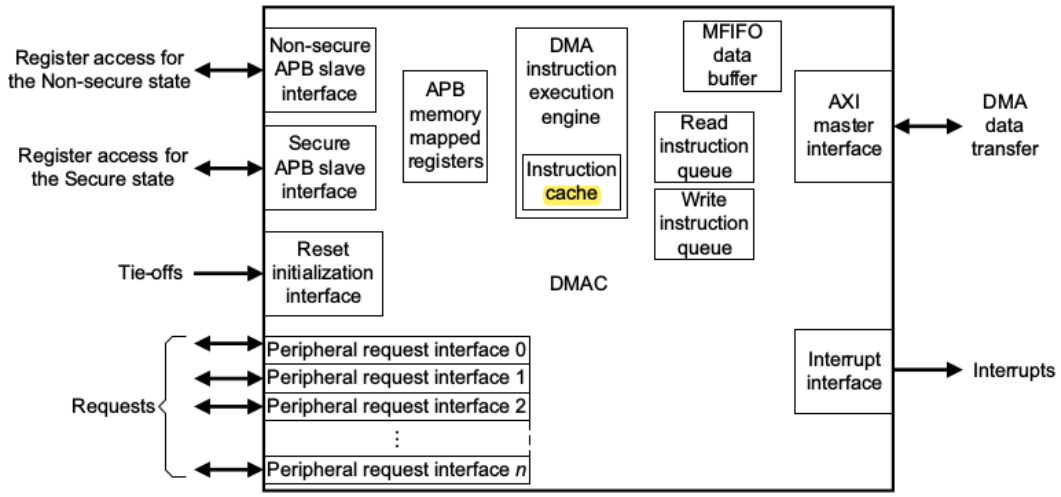

Figure 3.1: DMAC Block Diagram

It has its own cache, read/write queues, a data buffer, different secure and non-secure, API slave interfaces, program counters,etc.

The DMAC instructions are present in system memory accessible by it. However, cache also stores instructions; size of the cache in configurable. Each instruction takes 1 AXI clock cycle to execute.

It has 8 channels, with each channel having a program counter When a DMA channel

thread executes a load or store instruction, the DMAC adds the instruction to the relevant read or write queue.

3.2 DMAC Interfaces

It has the following interfaces: (can elaborate if want)

1. 2 Slave Interface - The DMAC provides with two APB slave interfaces; secure and non-secure.
2. 1 master interface - The DMAC contains a single AXI master interface that enables it to transfer data from a source AXI slave to a destination AXI slave.
3. Peripheral request interface - The peripheral request interface supports the connection of DMA-capable peripherals.
4. Interrupt interface - The interrupt interface enables efficient communications of events to an external microprocessor.
5. Reset initialization interface - This interface enables you to initialize the operating state of the DMAC as it exits from reset.

3.3 Operating States

A DMAC thread can be in the following states :

Initially when the DMAC comes out of reset, all the threads are in stopped state and the manager thread is in either stopped or executing state depending on the value of boot_from_pc. Following is a short description of each state:

- **Stopped:** The thread has an invalid PC and it is not fetching instructions.
- **Executing:** The thread has a valid PC and therefore the DMAC includes the thread when it arbitrates.
- **Cache miss:** The thread is stalled and the DMAC is performing a cache line fill.
- **Updating PC:** The DMAC is calculating the address of the next access in the cache.
- **Waiting for event:** The thread is stalled and is waiting for the DMAC to execute DMASEV using the corresponding event number.

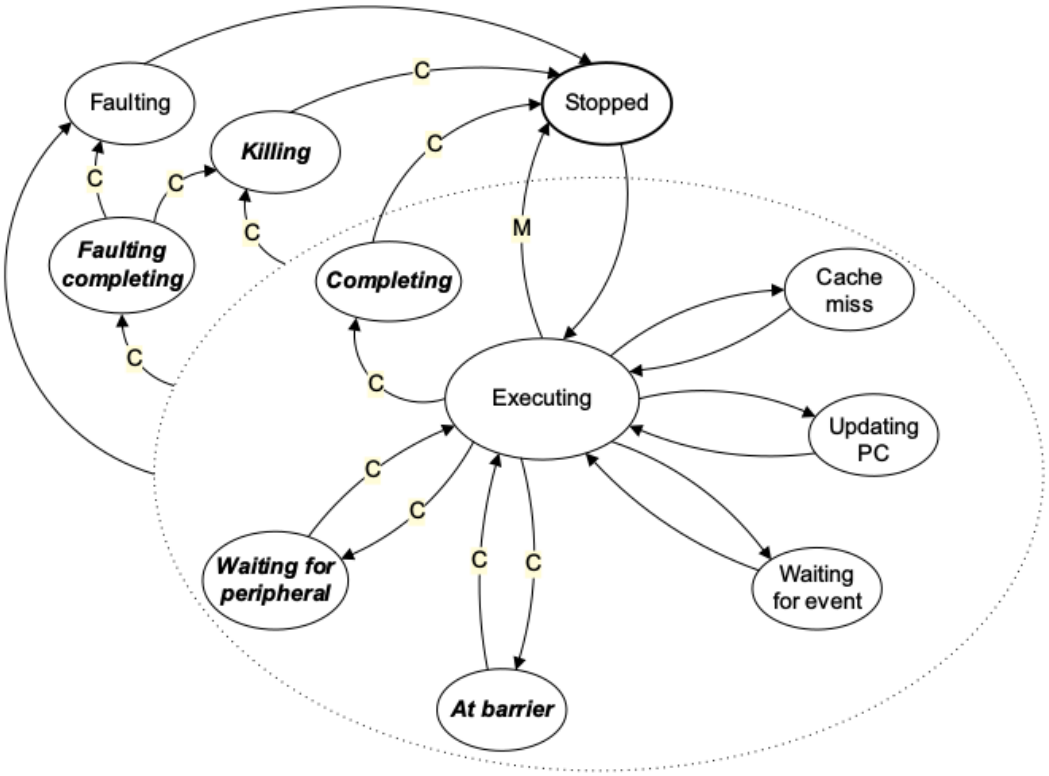

Figure 3.2: Channel Operating States

- **At barrier:** A DMA channel thread is stalled and the DMAC is waiting for transactions. This happens when we call DMARMB, DMAWMB, which are basically read before write type instructions
- **Waiting for peripheral:** A DMA channel thread is stalled and the DMAC is waiting for the peripheral to provide the requested data.
- **Faulting completing:** A DMAC thread goes here in case of a fault and stays here till is completes an outstanding operation after which it will move to the Faulting state.
- **Faulting:** The thread is stalled indefinitely and then goes to stopped state when DMAKILL is executed.
- **Killing:** A DMA channel thread is waiting for the AXI master interface to signal that the outstanding load or store transactions are complete.
- **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.

3.4 Initializing the DMAC:

The boot_manager_ns signal is the only method to set the security state of the DMA manager.

The boot_from_pc gives the state of the manager thread. If in executing state then the boot_addr[31:0] gives the DPC value for the DMA to start.
The DMAC provides the boot_irq_ns[x:0] signals using which we can assign each irq[x] signal (interrupt signal) to a security state
The DMAC provides the boot_periph_ns[x:0] signals using which we can assign each of the peripheral requests to a security state.

3.5 Peripheral request interface

The peripheral request interface consists of a peripheral request bus and a DMAC acknowledge bus that use the prefixes:
dr - The peripheral request bus.
da - The DMAC acknowledge bus.

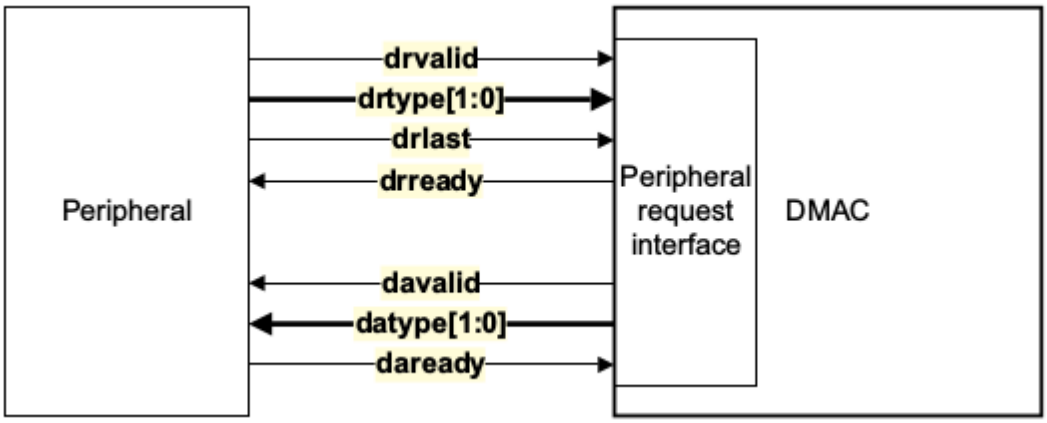

Figure 3.3: DMAC Peripheral Request Interface

The peripheral uses drtype[1:0] to either:

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

Others are handshake signals, for communication between the peripheral and the DMAC during the start and end on a transfer.

3.6 Mapping to DMA Channel

The DMAC enables you to assign a peripheral request interface to any of the DMA channels. When a DMA channel thread executes DMAWFP, the value programmed in the peripheral [4:0] field specifies the peripheral associated with that DMA channel.

We can set the number of simultaneous active requests that a DMAC is able to accept, for each peripheral request interface. For each peripheral interface, the DMAC has a request FIFO that it uses to capture the requests from a peripheral. When a request FIFO is full then the DMAC sets the corresponding drready LOW to signal that the DMAC cannot accept any requests sent from the peripheral.

3.7 DMA Manager

A single DMA manager thread exists for the DMAC which you can use it to initialize the other DMA channel threads.
It is a section of the DMAC that manages the operation of the DMAC by executing its own program thread.
When the DMA manager thread accesses the AXI master interface, the DMAC signals the AXI identification tag to be the same number as the number of DMA channels that the DMAC provides. For example, if the DMAC is configured to provide eight DMA channels, when the DMA manager performs a read operation, the DMAC sets ARID[3:0] to 0b1000.
DMAGO instruction is only executed by the manager. 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.

3.8 Events and Interrupts

Using the INTEN register we can specify whether an event-interrupt resource should be an event or an interrupt.
The DMAC executes the instruction DMAWFE before DMASEV; in this the former makes the thread wait for an event and latter sets the event. This is to restart a channel. For example, if four DMA channels have all executed DMAWFE for event 12, then when another DMA channel executes DMASEV for event 12, the four DMA channels all restart at the same time.
The DMAC provides the irq[x] signals for use as active-high level-sensitive interrupts to external microprocessors.

3.9 Aborts

There are two types of aborts precise and imprecise. In the former, The DMAC updates the PC Register with the address of the instruction that created the abort; this does not happen in case of the latter.
Precise aborts usually happen when something in the non-secure state tries to access or execute something in the secure state. Imprecise aborts happen when DMAC receives error messages from the AXI master bus or if the MFIFO is too small to satisfy all load store instructions.

3.10 DMA channel arbitration

The DMAC uses a round-robin scheme to service the active DMA channels. It always services the manager before the other channels. All channels have equal priority.

3.11 Registers

The DMAC has a wide list of registers that it uses for various purposes.
It has registers for various interrupt functions, fault status and type, channel status,

channel program counter, source address, destination address, loop counters, debug status, DMAC configurations, etc.

3.12 Instructions

The DMAC has its own ISA that provides a flexible method of specifying the DMA operations.
This ISA includes various instructions for load, store, wait for peripheral, add half-words, flush, move, send event, loops, read/write memory barrier etc.
These instructions allow the requests for transfer of data to be sent in flexible manner as well as help in communicating with the peripheral.

3.13 Why ARM DMA?

After looking into the architectures of different DMAs described in the previous sections, we decide to go with the **ARM DMAC** and implement a version of that for the SHAKTI processor.

The main reason for the same is that the ARM DMAC has its own ISA which makes its architecture much more flexible and compatible with a variety of peripherals than the other architectures we have seen. Apart from your normal load and store instructions, its ISA also contains instructions like -

- **DMAWFP (wait for peripheral)** that halts the data transfer until the DMAC receives a signal from the peripheral.
- **DMAWFE (wait for event)** that halts the execution until the specified event has occurred,
- **DMALDP (Load and notify Peripheral)** that notify the peripheral each time the load has occurred,
- **DMASTP (store and notify Peripheral)** that notify the peripheral each time the store has occurred.

All such instructions help to facilitate interactions like polling of registers and sending of control signals in between data transfers with the peripherals.

It also has instructions like **DMAADDH** and **DMALP** that provide adding of immediate value and looping facilities respectively, making the transfer requests less rigid.

Also the DMAC has two APB slave interfaces. One APB operates in the secure state and the other in the non-secure state. DMAC provides programmable security state for each DMA channel. When a DMA channel thread is in the Non-secure state, and an instruction attempts to program the channel to perform a secure AXI transaction, the DMAC executes a DMANOP (no operation) and sends the thread to faulting completing state. Hence this DMAC also provides a good security support.

The above reasons make the ARM DMAC an obvious choice for our DMA implementation for the SHAKTI processor.

The following sections have a detailed description of the DMAC code implemented, the test bench and the various test cases performed on it.

CHAPTER 4

Implementation of DMAC

This chapter explains each section of the DMAC code written for this project in detail. The DMAC module of this code has been inspired by the ARM DMAC. In some of the sections, along with the code we have also explained the theory behind it that would help understand the code better.

4.1 Interface and Methods

The interface for the DMAC module contains a vector of peripheral interface, an AXI4 master interface and a method for reset initialization.

1. **interface Vector(numPeripherals, Periph_ifc) peripherals;**
   This is a vector of interface "Periph_ifc". This interface is mainly for linking the DMAC with peripheral and for sending and receiving ready and acknowledgement signals from both ends. It contains the following methods -
   - method Action periph_input(Bit(2) drtype, bit drvalid, bit daready); - This action method takes the valid, ready signals from the peripheral as well as drtype that essentially tells the type of transfer requested by the peripheral.
   - method Bit(2) get_datype(); - This method returns datype to the peripheral indicating an acknowledgement of the type of transfer, that the DMAC signals.
   - method bit get_davalid(); - This method returns davalid indicating when the DMAC provides valid control information.
   - method bit get_drready(); - This method returns drready indicating whether the DMAC can accept the information that the peripheral provides on drtype.

   We have a vector of this peripheral interface because, we need to have one interface for each peripheral. Hence when multiple peripherals try to call these methods to send there signals, there wont be any clashes. Once we get in input from multiple peripherals we put only one of it into the DMAC local variable at a time. Once that request is served the next one will be assigned. The priority order for this is that the lower peripheral id gets the higher priority.

2. **interface Ifc_axi4_master(id_width, addr_width, data_width, user_width) master;**

This is the AXI4 master interface that is required for performing various read-/write transactions from one memory to the other for the transfer of data. It takes in the above parameters which we set when we configure the DMA. The DMAC contains a single AXI master interface that enables it to transfer data from a source AXI slave to a destination AXI slave.

3. **method Action reset_initialization (int boot_addr, bit boot_from_pc, bit boot_manager_ns, Bit(32) boot_irq_ns, Bit(32) boot_periph_ns);**
   This methods is for the reset conditions of the DMA. It takes in all the values needed in case of a reset which are then assigned to the respective registers.

4.2 Memory Mapped DMAC Registers

Following are the memory mapped DMAC registers used in the code:

1. **Channel Program Counter Register (CPC):** Provides the value of the program counter for the DMA channel thread. The DMAC provides a CPCn Register for each DMA channel that it contains.

2. **Source Address Registers (SAR):** Provides the address of the source data for a DMA channel. The DMAC provides a SARn Register for each DMA channel that it contains. The DMAC writes the initial source address value to the Source Address Register when the DMA channel thread executes a DMAMOV SAR instruction.

3. **Destination Address Registers (DAR):** Provides the address of the destination data for a DMA channel. The DMAC provides a DARn Register for each DMA channel that it contains. The DMAC writes the initial destination address value to the Destination Address Register when the DMA channel thread executes a DMAMOV DAR instruction.

4. **Channel Control Registers (CCR):** Controls the AXI transactions that the DMAC uses for a DMA channel. The DMAC writes to the corresponding CCR Register when a DMA channel thread executes a DMAMOV CCR instruction. The DMAC provides a CCRn Register for each DMA channel that it contains.

5. **Loop Counter 0 Registers(LC0):** Provides the status of loop counter zero for the DMA channel. The DMAC updates this register when it executes DMALPEND[S|B], and the DMA channel thread is programmed to use loop counter zero. The DMAC provides a LC0_n Register for each DMA channel that it contains.

6. **Loop Counter 1 Registers(LC1):** Provides the status of loop counter one for the DMA channel. The DMAC updates this register when it executes DMALPEND[S|B], and the DMA channel thread is programmed to use loop counter one. The DMAC

provides a LC1_n Register for each DMA channel that it contains.

4.3 Instruction Functionality Rule

This rule takes in a DMAC instruction and decodes it according to the opcode specified in the ARM manual and then performs various assignments and functions that that instruction is meant to do. Here are the following instructions -

1. **DMAADDH (Add Halfword)**
   Description - adds an immediate 16-bit value to the SAR[n] (source address) Register or DAR[n] (destination address) Register, for that DMA channel thread. This enables the DMAC to support 2D DMA operations
   Opcode -
   ra = 0; Add the immediate value to SAR[n] register

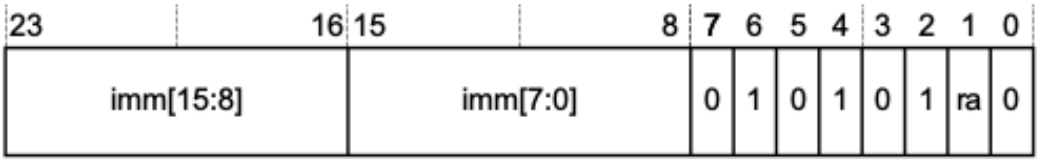

Figure 4.1: Encoding for DMAADDH

   ra = 1; Add the immediate value to DAR[n] register
   imm -> the immediate value to be added

2. **DMAADNH (Add Negative Halfword)**
   Description - adds an immediate negative 16-bit value to the SAR[n] Register or DAR[n] Register, for that DMA channel thread. This enables the DMAC to support 2D DMA operations, or reading or writing an area of memory in a different order to naturally incrementing addresses.
   Opcode -
   ra = 0; Add the immediate value to SARn register

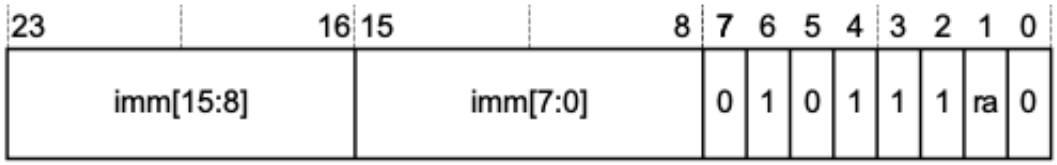

Figure 4.2: Encoding for DMAADNH

   ra = 1; Add the immediate value to DARn register
   imm -> the immediate value to be added; The immediate value here is in two's compliment form.

3. **DMAEND**

Description - 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.

Opcode -

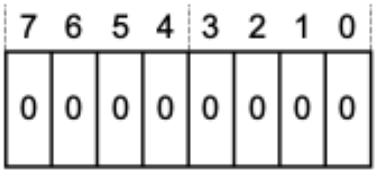

Figure 4.3: Encoding for DMAEND

4. **DMAGO**

Description - 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.

Opcode -

cn[2:0] – the channel that will be set to execution state

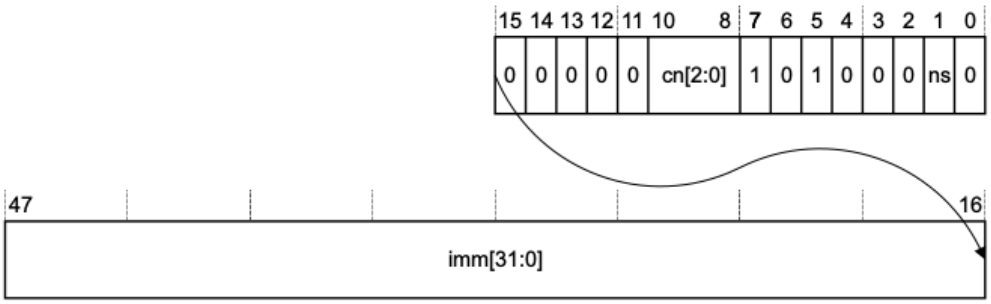

Figure 4.4: Encoding for DMAGO

ns = 1; the DMA channel operates in the Non-secure state.
ns = 0; the execution of the instruction depends on the security state of the DMA manager. If DMA manager is in the Secure state then DMA channel operates in the Secure state else the DMAC aborts.
32-bit immediate - The immediate value that is written to the CPC[n] (Channel Program Counter) Register for the selected channel number.

5. **DMALD[S|B] (Load)**

Description - instructs the DMAC to perform a DMA load, using AXI transactions that the Source Address Registers (SAR[n]) and Channel Control Registers (CCR[n]) specify. It places the read data into the MFIFO (responseDataFS) 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.
Opcode -
Case1: bs=0, x=1: Single load instruction [S]. If the peripheral request signal

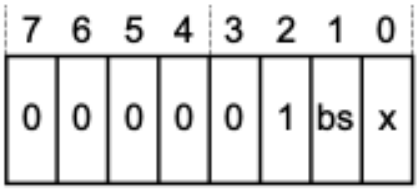

Figure 4.5: Encoding for DMALD[S|B]

also requests for a single transfer then perform the load instruction else perform NOP instruction.
Case2: bs=1, x=1: Burst load instruction [B]. If the peripheral request signal also requests for a burst transfer then perform the load instruction else perform NOP instruction.
Case3: bs=0, x=0: Perform load operation unconditionally accordingly to the entries of the Channel Control Registers.

6. **DMAST[S|B] (Store)**
Description - instructs the DMAC to transfer data from the FIFO to the location that the Destination Address Registers (DAR[n]) specifies, using AXI transactions that the Destination Address Register (DAR[n]) and Channel Control (CCR[n]) Registers specify. DMAST is an unconditional instruction but DMASTS and DMASTB are conditional on the state of the request_type flag.
Opcode -
Case1: bs=0, x=1: Single store instruction [S]. If the peripheral request signal

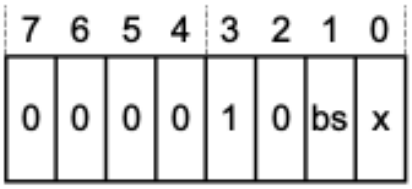

Figure 4.6: Encoding for DMAST[S|B]

also requests for a single transfer then perform the store instruction else perform NOP instruction.
Case2: bs=1, x=1: Burst store instruction [B]. If the peripheral request signal also requests for a burst transfer then perform the store instruction else perform NOP instruction.
Case3: bs=0, x=0: Perform store operation unconditionally accordingly to the entries of the Channel Control Registers.

7. **store zero**
Description- instructs the DMAC to store zeros, using AXI transactions that the Destination Address Registers (DAR[n]) and Channel Control Registers (CCR[n]) specify.
Opcode -

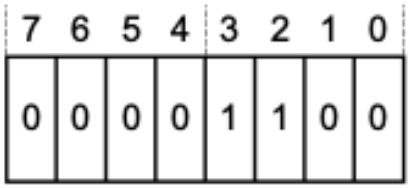

Figure 4.7: Encoding for DMASTZ

8. **DMALP (Loop)**
Description- instructs the DMAC to load an 8-bit value into the Loop Counter Register you specify. This instruction indicates the start of a section of instructions, and you set the end of the section using the DMALPEND instruction. The DMAC repeats the set of instructions that you insert between DMALP and DMALPEND until the value in the Loop Counter Register reaches zero.
The DMAC saves the value of the PC for the instruction that follows DMALP . After the DMAC executes DMALPEND , and the Loop Counter Register is not zero, this enables it to execute the first instruction in the loop.
Opcode -
iter[7:0] - Specifies the number of loops to perform, range 1-256.

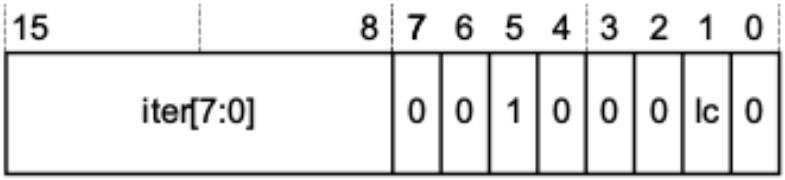

Figure 4.8: Encoding for DMALP

lc = 0: DMAC writes the value loop_iterations minus 1 to the Loop Counter 0 (lc0) Register.
lc = 1: DMAC writes the value loop_iterations minus 1 to the Loop Counter 1 (lc1) Register.

9. **DMALPEND[S|B] (Loop End)**
Description- Loop End indicates the last instruction in the program loop. If the loop starts with the instruction DMALP then -
The loop has a defined loop count and DMALPEND[S|B] instructs the DMAC to read the value of the Loop Counter Register. If a Loop Counter Register returns:
Zero - The DMAC executes a DMANOP and therefore exits the loop.
Non-zero - The DMAC decrements the value in the Loop Counter Register and updates the thread PC to contain the address of the first instruction in the program

loop, that is, the instruction that follows the DMALP .
Opcode -
backwards_jump[7:0] - Sets the relative location of the first instruction in the

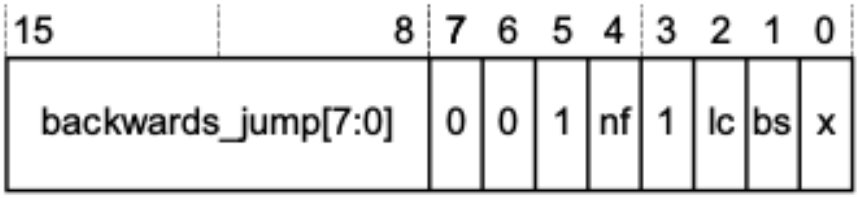

Figure 4.9: Encoding for DMALPEND

program loop. The assembler calculates the value for backwards_jump[7:0] by subtracting the address of the first instruction in the loop from the address of the DMALPEND instruction.
nf = 0: DMALPFE started the loop
nf = 1: DMALP started the loop
lc = 0: the Loop Counter 0 Register (lc0) contains the loop counter value.
lc = 1: the Loop Counter 1 Register (lc1) contains the loop counter value.

Case1 [S]: bs=0, x=1: If the peripheral request signal requests for a single transfer then perform the DMALPEND instruction else perform NOP instruction and exit the loop.
Case2 [B]: bs=1, x=1: If the peripheral request signal requests for a burst transfer then perform the DMALPEND instruction else perform NOP instruction and exit the loop.
Case3: bs=0, x=0: Perform DMALPEND operation unconditionally.

10. **DMAMOV (Move)**
Description: instructs the DMAC to move a 32-bit immediate into the following registers:
• Source Address Registers
• Destination Address Registers
• Channel Control Registers
Opcode - rd[2:0] = 0: move the immediate value to Source Address Register

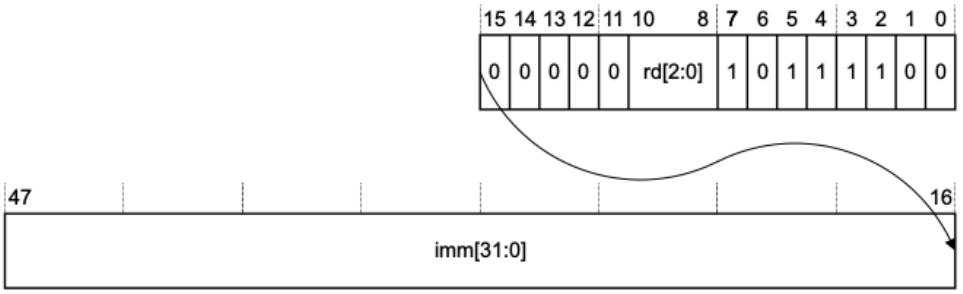

Figure 4.10: Encoding for DMAMOV

(SAR[n]).
rd[2:0] = 1: move the immediate value to Channel Control Register (CCR[n]).
rd[2:0] = 2: move the immediate value to Destination Address Register (DAR[n]).
imm = 32bit immediate value to be put into on of these registers.

11. **DMANOP (No Operation)**
Description: No Operation does nothing. You can use this instruction for code alignment purposes.
Opcode -

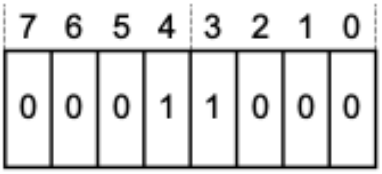

Figure 4.11: Encoding for DMANOP

4.4 Read/Write Rules

These are the rules that actually perform the transaction of data from the load address to the store address using AXI4 requests. The following are the rules and what each of them do:

startread:

In this rule we create a read request and enqueue it in the input AXI4 read address FIFO. The AXI4 will then send it to the output read FIFO. All the entries needed for this request are present in the CCR (channel config register) register of the DMA. This rule will only fire if the a load instruction has been given to the DMA.

rl_finishRead:

For this rule to fire we first check that whether the data coming out of the output read FIFO has the same id as the one enqueued and that its response (rresp) is "axi4_resp_okay" indicating that there are no AXI4 issues. Once these conditions are met we pop out the data from the AXI4 output data FIFO and enqueue it into one of our DMAC's internal FIFO (responseDataFs) so that we can convey this data for a write request in the store address. It also increments the read counter variable so that we know how many of the

requested read transfers have been done.

read_done:

This rule is fired when the read request is complete i.e. all the read data has been popped out of the AXI4 read FIFO and been enqueued in the DMAC's internal FIFO (responseDataFs). This rule reduces the number of pending read requests by 1.

startwrite:

In this rule we create an AXI4 write request by enqueuing write data and write address structures into the respective AXI4 FIFOs. We get the write data from the DMAC's internal FIFO (responseDataFs), the write address from the DMAC's register (store address) and all the other parameters from the CCR (channel config register) register of the DMAC. This rule will only fire if store instruction has been given to the DMA. In case of SINGLE mode transfer, only this rule will fire (once) since only one data transfer is needed. In case of BURST mode transfer this rule will fire once and then the next rule (rl_send_burst_write_data) will keep firing depending on the number of burst transfer mentioned as it needs to enqueue those many write data. All this is handled using the variable "rg_burst_count" that counts the burst size and accordingly fires the respective rules. It also sets the flag "lv_last" high when the last transfer is taking place.

rl_send_burst_write_data:

This rule will fire only when BURST mode transfer is happening so that we can enqueue in the remaining write data into the AXI4 FIFOs. For the last transfer "wlast" entry of the axi_write_data will be set high and the rule will fire for the last time after that.

rl_finishWrite:

For this rule to fire, we first check that whether the data coming out of the output write FIFO has the same id as the one enqueued and that its response (rresp) is "axi4_resp_okay" indicating that there are no AXI4 issues. We then pop out that entry from the AXI4 FIFO. The control flow reaching this rule means that that write request has been ful-

filled and reduces the value of pending write requests by 1.

TransferDone:

This rule is fired when both number of pending read and write requests are zero as well as the instruction set has reached the DMAEND (last instruction). This rule marks the end of all the transfer request sent by that particular peripheral. Hence it sets the ready and valid signals of the DMAC high indicating that its transfer is complete and it is ready for the next one. It also sets da_type to type of transfer it has completed. Along with that, we free the peripheral input variable so that the next transfer can be initiated.

4.5 Function Returning Rules

Since the DMA will be having multiple channels it can use to transfer data for multiple peripherals parallely, we will be needing the above instruction decode rule and the read/write rules for every channel. In order to do this we create a function called "generatePortDMARules" that returns rules when we call it for each channel number. Among these rules, we set the rule will a lower channel number to have a higher urgency.

4.6 DCBus Modules

We integrated the DMAC module with DCBus to create memory mapped registers. So now we can configure them by sending AXI4 requests to write into that memory space. (The DCBus is itself the slave for this DMAC module.) We thus make DMAC slave modules for AXI4, AXI4l and APB that can be used to link this DMAC module and thus call it from a test bench that has the master side. We also used this DCBus module to create memory mapped registers. We create various structures to facilitate these registers having different bit-to-bit mapping.

4.7 BRAM

The DMA needs a memory block that has the instruction set for the particular transfer mapped to it. The PC (program counter) of that channel can then index that memory block to access this instruction set. For this, we use a BRAM module. As of now, the instruction are pre-written into the BRAM module and meant for a single channel use. In order to make this compatible for multiple channels, we have to assign different sections of this BRAM memory to different channels. The instruction sets for a particular channel would be in that section. We can also create a C code that writes these instructions in that particular memory space depending on the peripheral requirement.

CHAPTER 5

Test Bench For DMAC

We create a test bench for performing standalone testing of the DMAC module before we integrate it with the SHAKTI processor. We create an instance of the AXI4 slave module of the DMAC in this test bench as well as an AXI4 master of this test bench module. We then connect them via a connector.
Now we can essentially send requests from the test bench master to the DMAC slave.

5.1 Configuring DMAC Registers via Test Bench

The first task is to see if we can send AXI4 write requests to the memory mapped register of the DMAC module via test bench. This needs to be done before actually sending the transfer requests because we would want to configure the DMAC registers.
For this we create a rule write in which we set some arbitary value to be written into the memory mapped DMAC register with relative address ‘h00. This is the DMA manager status register (DSR).
*Note: here we are not actually configuring the register but rather just checking that we can correctly write to it.
We set up the other AXI4 parameters as well and then enqueue the data and the address into the respective AXI4 write input FIFOs. Once we get an axi_okay response for this write request, we send out a read request in the read rule for the same address to verify if it has correctly written the data into the specified address.
We also then print out the value of this DSR register in the DMAC module to verify that the entry of the register has been successfully modified.
On running this test, we achieve the output as expected. Hence we can set the memory mapped registers to the appropriate values required for transfer.

5.2 Setting Up Environment for Transfer Requests

We now move on to setting up the environment for transfer requests to test if the DMAC can actually function correctly.
For this we first create two OCM modules which we will use to transfer data to and from. We set their memory ranges as follows-

- OCM1 Base -> 'h00000300
- OCM1 End -> 'h00000500
- OCM2 Base -> 'h00000600
- OCM2 End -> 'h00000800

We even set the memory range for our DMAC module-

- DMAC Base -> 'h00000000
- DMAC End ->'h00000200

**these are relative address values*

So now we have 2 AXI4 master and 3 AXI4 slaves -

- 1 dmac master
- 1 test bench master
- 1 dmac slave
- 2 ocm slaves.

We connect all these using an AXI4 crossbar and create AXI4 map functions for read-/write (fn_axi4_rd_map, fn_axi4_wr_map) that tells which slave connection maps to what address range.

Now we have set up the environment and are thus ready for sending transfer requests to the DMAC for sending data from OCM1 to OCM2 and vice versa. We write our instruction set into the BRAM module and the send a peripheral input to the "periph_input" method of the DMAC signalling to start reading from the BRAM module and hence initiate the transfer. We do this in a rule called run_dmac. Once this rule is fired, the input is sent to the DMAC module and the transfer process begins.

Finally we create a rule called read_ocm2, that will get fired only when we receive a ready signal from the DMAC signalling the transfer request has been completed. We then send AXI4 requests to read the contents of OCM2 in order to check whether the transfer was conducted successfully or not.
(we will essentially be transferring data from OCM1 to OCM2 in all our test cases and hence here we are only reading OCM2 values)

The following chapter contains test cases and results that will essentially tell you about all the diverse test runs that were performed in this environment.

CHAPTER 6

Test Cases and Results

Our main aim for testing is to check that the DMA can perform one or multiple SINGLE and BURST transfer requests as well as to do a functionality check for all the instruction written.

6.1 Overview

As seen in the test-bench section, we have created two OCM slaves. We will thus be running various test cases requesting data transfer in different ways from OCM1 to OCM2. We set the memory of these OCMs with certain value.

For testing, we are currently writing a set of binary instruction into the BRAM memory, which could be directly decoded by our DMAC module.
In future we can always write an assembly level code that directly gives these binary instruction instead of us hard-coding it.
Also all the test cases listed below are for numChannels = 1 and numPeripherals = 1.
We also need to give an input to the periph_input method of the peripherals interface of the DMAC module via the test bench. For each test case we need to provide three inputs - Bit(2) drtype, bit drvalid, bit daready. We set the drvalid and daready bits to 1 for all cases, and set drtype as 0 for single transfer and 1 for burst transfer test case respectively.

Once the transfer is complete from the DMAC side, it sends a signal (drready) to the test bench indicating that transfer is done and is now ready to accept other requests. Once the test bench receives that signal, we make an AXI4 read request for OCM2 so that we can verify if the transfer was performed successfully.

6.2 General Flow of code:

Whenever we want to send a transfer request to the DMAC, we first set the configure register (CCR) and the source (SAR) and destination (DAR) address registers (either directly writing into the memory mapped register via test bench or by sending instructions to the DMAC).
Then we send the load and store instructions. These instructions will the enable the rules that send the AXI4 read and write requests. Once the read request is fulfilled it will go to read_done rule and decrement the number of pending requests; similar for the write request.
Once all requests are served and there are no more incoming requests, the flow will enter the transferDone rule that will reset the inputs from peripheral and set the output peripheral signals.

6.3 Test Cases

Following are the different test cases that were run on this DMA.

1. **Instruction set for a single transfer from OCM1 (0300) to OCM2 (0600)**
   0001000401BC
   0000030000BC
   0000060002BC
   000000000005
   000000000009
   000000000000

   Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
   Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
   Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
   Instruction 4 – DMALD giving the load instruction for single transfer.
   Instruction 5 – DMAST giving the store instruction for single transfer.
   Instruction 6 – DMAEND marking the end of transfer requests.

   Expected Output: The first entry of OCM2 must change to the first entry of OCM1.

2. **Instruction set for a burst transfer (4 words) from OCM1 (0300) to OCM2 (0600)**
   000D403501BC
   0000030000BC
   0000060002BC
   000000000007
   00000000000B
   000000000000

   Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for burst transfer) to the CCR register.
   Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
   Instruction 3 - DMAMOV for moving an immediate value (value of destination, 0600 address) to the DAR register.
   Instruction 4 – DMALD giving the load instruction for burst transfer.
   Instruction 5 – DMAST giving the store instruction for burst transfer.
   Instruction 6 – DMAEND marking the end of transfer requests.

   Expected Output: The first four entries of OCM2 must change to the first four entries of OCM1.

3. **Instruction for multiple SINGLE transfer, using DMAADDH Instruction from OCM1 (0300) to OCM2 (0600)**
   0001000401BC
   0000030000BC
   0000060002BC
   000000000005
   000000000009
   000000000454
   000000000456
   000000000005
   000000000009
   000000000000

   Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
   Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
   Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
   Instruction 4 – DMALD giving the load instruction for single transfer.
   Instruction 5 – DMAST giving the store instruction for single transfer.
   Instruction 6 – DMAADDH adding the immediate value (04) to the SAR register.
   Instruction 7 – DMAADDH adding the immediate value (04) to the DAR register.
   Instruction 8 – DMALD giving the load instruction for single transfer.
   Instruction 9 – DMAST giving the store instruction for single transfer.
   Instruction 10 – DMAEND marking the end of transfer requests.

Expected Output: The first two entries of OCM2 must change to the first two entries of OCM1.

4. **Instruction set for multiple single transfer, using DMAADNH Instruction**
   0001000401BC
   0000030800BC
   0000060802BC
   000000000004
   000000000008
   000000FFFC5C
   000000FFFC5E
   000000000004
   000000000008
   000000000000

   Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
   Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
   Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
   Instruction 4 – DMALD giving the load instruction performing unconditionally.
   Instruction 5 – DMAST giving the store instruction performing unconditionally.
   Instruction 6 – DMAADNH adding the immediate value (2's compliment of 04) to the SAR register.
   Instruction 7 – DMAADNH adding the immediate value (2's compliment of 04) to the DAR register.
   Instruction 8 – DMALD giving the load instruction performing unconditionally.
   Instruction 9 – DMAST giving the store instruction performing unconditionally.
   Instruction 10 – DMAEND marking the end of tranfer requests.

   Expected Output: The second and third entry of OCM2 must change to the second and third entry of OCM1.

5. **Instruction set for multiple single transfer, using DMALP (loop) Instruction**
   0001000401BC
   0000030000BC
   0000060002BC
   000000000004
   000000000008
   000000000520
   000000000454
   000000000456
   000000000004
   000000000008
   000000000538
   000000000018

000000000000

Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
Instruction 4 – DMALD giving the load instruction performing unconditionally.
Instruction 5 – DMAST giving the store instruction performing unconditionally.
Instruction 6 – DMALP marking the start of loop as well as giving the value of number of iterations (05).
Instruction 7 – DMAADDH adding the immediate value (04) to the SAR register.
Instruction 8 – DMAADDH adding the immediate value (04) to the DAR register.
Instruction 9 – DMALD giving the load instruction performing unconditionally.
Instruction 10 – DMAST giving the store instruction performing unconditionally.
Instruction 11 – DMALPEND marks the end of the loop.
Instruction 12 - DMANOP, a no operation function is needed to be put after DMALPEND because the PC goes to the next instruction before the DMALPEND can change it to the instruction after DMALP. Hence to avoid mismatch we add this.
Instruction 13 – DMAEND marking the end of transfer requests.

Expected Output: The first six entries of OCM2 must change to the first six entries of OCM1.

6. **Instruction set for multiple SINGLE transfer, one given correctly and the other one incorrectly i.e. not matching with the request type given by the peripheral**
0001000401BC
0000030000BC
0000060002BC
000000000005
000000000009
0000030800BC
0000060802BC
000000000007
00000000000B
000000000000

Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
Instruction 4 – DMALD giving the load instruction for single transfer (in coherence with the request type).

Instruction 5 – DMAST giving the store instruction for single transfer (in coherence with the request type).
Instruction 6 - DMAMOV for moving an immediate value (value of source address, 0308) to the SAR register.
Instruction 7 - DMAMOV for moving an immediate value (value of destination address, 0608) to the DAR register.
Instruction 8 – DMALD giving the load instruction for burst transfer (inconsistent, instruction is for a burst mode transfer and peripheral request type is SINGLE)
Instruction 9 – DMAST giving the store instruction for burst transfer (inconsistent, instruction is for a burst mode transfer and peripheral request type is SINGLE)
Instruction 10 – DMAEND marking the end of transfer requests.

Expected output: the code will run the first request correctly and tell no operation for the second. So first entry of OCM2 must change to the first entry of OCM1.

7. **Instruction set for a request for zero store (2 requests) using DMASTZ instruction**
   **no need to have a source address and load instruction here*
   0001000401BC
   0000060802BC
   00000000000C
   0000061002BC
   00000000000C
   000000000000

   Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for single transfer) to the CCR register.
   Instruction 2 - DMAMOV for moving an immediate value (value of destination address, 0608) to the DAR register.
   Instruction 3 – DMASTZ, instructing to store a zero.
   Instruction 4 - DMAMOV for moving an immediate value (value of destination address, 0610) to the DAR register.
   Instruction 5 – DMASTZ, instructing to store a zero.
   Instruction 6 – DMAEND marking the end of transfer requests.

   Expected Output: The third and fifth entry of OCM2 must change to zero.

8. **Instruction set for multiple (two) BURST transfers**
   000D403501BC
   0000030000BC
   0000060002BC
   000000000007
   00000000000B
   000000001454
   000000001456

000000000004
000000000008
000000000000

Instruction 1 – DMAMOV for moving an immediate value (that sets the parameters for burst transfer) to the CCR register.
Instruction 2 - DMAMOV for moving an immediate value (value of source address, 0300) to the SAR register.
Instruction 3 - DMAMOV for moving an immediate value (value of destination address, 0600) to the DAR register.
Instruction 4 – DMALD giving the load instruction for burst transfer.
Instruction 5 – DMAST giving the store instruction for burst transfer.
Instruction 6 – DMAADDH adding the immediate value (h'14) to the SAR register.
Instruction 7 – DMAADDH adding the immediate value (h'14) to the DAR register.
Instruction 8 – DMALD giving the load instruction performing unconditionally.
Instruction 9 – DMAST giving the store instruction performing unconditionally.
Instruction 10 – DMAEND marking the end of transfer requests.

Expected Output: The first four and sixth to ninth entries of OCM2 must change to the respective entries of OCM1.

All these test cases were run on the dmac code and all of them gave the output that was expected. Thus the dmac module was able correctly to transfer data from one memory point to the other using all the coded instructions.

CHAPTER 7

Future Scope

In this chapter, we discuss the future possibilities of this project, what all features can be added to the existing DMAC code to make it more useful, efficient and flexible.

7.1 Expanding the Instruction Set

The current instruction set can be increased by adding more instructions that might be able to make the DMA more flexible and responsive. Few of the instructions present in the ARM manual that haven't been implemented yet need to be added. These include - Flush Commands (DMAFLUSH) - clears the state in the DMAC that describes the contents of the peripheral and sends a message to the peripheral to resend its level status.
Commands that notify the peripheral once a load/store has been issued. (DMALDP, DMASTP).
Read/Write Memory Barriers (DMARMB, DMAWMB) - forces the DMA channel to wait until all of the executed load/store instructions for that channel have been issued on the AXI master interface and have completed.
Waiting for Peripherals (DMAWFP) - instructs the DMAC to halt execution of the thread until the specified peripheral signals a DMA request for that DMA channel.
Commands for sending and waiting for interrupts and events (DMASEV, DMAWFE).
We can add the instructions that are not mentioned in the manual like a command for conditional transfers (parallel to if statements).

7.2 Testing for multiple Channels and Peripherals

Although the DMAC module is coded to handle number of channels and number of peripherals to be greater than 1, it has only been tested for a single channel and peripheral.

We will thus need to create an appropriate test bench that can send multiple requests that will be served by different DMAC channels, thereby verifying its working. Similar for multiple peripherals.
For testing with multiple channels, we also need to divide the BRAM memory space into sections. We then assign different sections of this memory to different channels. The instruction sets for a particular channel would be in that section and hence the DMAC channels would index their particular sections of the BRAM to fetch their instructions.
We can also create a C code that writes these instructions in that memory space depending on the peripheral requirement instead of having to hard-code the instructions into the memory.

7.3 Additional Features

Some extra features can be added to the existing DMAC code -
We can have read/write queues and cache for the DMAC. The instructions read from BRAM, source/destination addresses can go into the cache so that they can be accessed directly and in a much quicker way by the DMAC. The read/write requests can go into these read/write queues from where they can be performed in a correct and organised manner while parallely doing other tasks. These features will thus make the DMA handle large amount of requests and data more efficiently.
Implementing different secure and non-secure API slave interfaces will also help adding a security aspect to the DMA. So whenever a DMA channel thread is in the Non-secure state, and an instruction attempts to program the channel to perform a secure AXI transaction, the DMAC will have to execute a DMANOP (no operation), throw an error signal and send the thread to faulting completing state. This will provide a good security for that DMAC.

7.4 Testing on SHAKTI

As of now, we have done standalone testing for the DMA by creating a separate test bench. We still need to run it on the SHAKTI processor. Hence one of the last things that needs to be done is to link this DMA with the SHAKTI processor and test it out along with it.

REFERENCES

https://gitlab.com/shaktiproject/uncore/devices/-/tree/master/dma

https://www.st.com/resource/en/reference_manual/dm00151940-stm32l41xxx42xxx43xxx44xxx45xx
advanced-armbased-32bit-mcus-stmicroelectronics.pdf

https://ieeexplore.ieee.org/document/8106971

https://github.com/pulp-platform/pulpissimo/blob/master/doc/datasheet/datasheet.pdf

https://binaryterms.com/direct-memory-access-dma.htmlDMAController'sWorking

https://ijtech.eng.ui.ac.id/uploads/submission/attachment/795/R3-EECE-795-20190123175300.pdf

http://www-verimag.imag.fr/ maler/Papers/thesis-selma.pdf

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=7434907

https://people.eecs.berkeley.edu/ krste/papers/EECS-2016-1.pdf

https://developer.arm.com/documentation/ddi0424/d?lang=en