FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Design of multi-channel DMA controller with interrupt and flexible descriptor configuration
Student:	Olha Harielina
Supervisor:	Ing. Ondrej Ille
Study program:	Informatics
Branch / specialization:	Computer engineering
Department:	Department of Digital Design
Validity:	until the end of summer semester 2024/2025

Instructions

Research open source implementations of DMA controllers and compare the typical feature sets of generic DMA controllers as well as the capabilities of DMA engines in common microcontrollers.
Propose DMA controller block diagram for modular implementation and configuration of the IP e.g., number of channels, presence of hardware interfaces or interrupts per channel, descriptor storage size etc.
Define structure of DMA descriptor configuration data. Define DMA descriptors mapping to particular system interfaces of choice e.g. AHB, AXI or RISC-V LSU.
Implement the DMA controller in SystemVerilog RTL code supporting both ASIC and FPGA target technology.
Demonstrate DMA controller functionality on a FPGA board or in RTL simulations.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 31 May 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

DESIGN OF MULTI-CHANNEL DMA CONTROLLER WITH INTERRUPT AND FLEXIBLE DESCRIPTOR CONFIGURATION

Olha Harielina

Faculty of Information Technology
Department of Digital Design
Supervisor: Ing. Ondrej Ille
January 11, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Harielina Olha. *Design of multi-channel DMA controller with interrupt and flexible descriptor configuration.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

This thesis is based on designing a part of a new product being developed by Tropic Square s. r. o , who I would like to thank first of all for providing me with the topic with future practical application. I would like to thank the entire team for their support and guidance during my work, especially Ing. Ondrej Ille for supervising this thesis and giving me countless advice during consultations. I would also like to thank Dr.-Ing. Martin Novotný for guidelines and recommendations regarding this work. Not least of all I would like to thank my father for immense help during my studies and my mother for support and providing an opportunity to study abroad.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on January 11, 2024

Abstract

This thesis proposes a versatile multichannel DMA controller with flexible channel and block configuration. It describes the basic and modern functionality of these controllers. It also compares several DMA engines to pick the optimal combination of frequently used features to make sure the design stays simple yet offers a wide enough range of configurations. The DMA controller was written in SystemVerilog targeting ASIC technology and supporting FPGA application as well. The module was tested in RTL simulations using mostly UVM test benches. It is capable of handling up to 32 concurrent data transfers and each transfer size can be configured for up to 2^{32} words at once. The proposed design finds an application in an IP core by Tropic Square s. r. o., and thanks to generic variables and its flexibility the controller is not limited to one application. Due to complete hardware implementation, it offers data transferring at high speed without the use of additional cycles for any instructions to be processed.

Keywords multichannel DMA, scatter-gather, generic design, digital design, AHB-Lite, channel arbitration

Abstrakt

Tato práce navrhuje univerzální vícekanálový řadič DMA s flexibilní konfigurací kanálů a modulu. Popisuje základní a moderní funkce těchto řadičů. Porovnává také několik DMA enginů s cílem vybrat optimální kombinaci s často používanými funkcemi, aby návrh zůstal jednoduchý a přitom nabízel dostatečně širokou škálu konfigurací. Řadič DMA byl napsán v jazyce SystemVerilog se zaměřením na technologii ASIC a podporuje i použití v FPGA. Modul byl testován v simulacích RTL převážně pomocí testů UVM. Je schopen zpracovat až 32 souběžných přenosů dat a každou velikost přenosu lze nakonfigurovat až pro 2^{32} slov najednou. Navržený design našel uplatnění v IP jádře od Tropic Square s. r. o. a díky generickým proměnným a své flexibilitě není řadič omezen na jedno konkrétní použití. Díky své kompletně hardwarové implementaci nabízí přenos dat vysokou rychlostí bez použití dalších cyklů pro zpracování případných instrukcí.

Klíčová slova vícekanálové DMA, scatter-gather, obecný návrh, digitální návrh, AHB-Lite, arbitráž kanálů

Abbreviations

ASIC	Application Specific Integrated Circuit
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
AXI	Advanced Extensible Interface
CPU	Central Processing
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
IC	Integrated Circuit
IP	Intellectual Property
M2M	Memory-to-Memory
M2P	Memory-to-Peripheral
P2M	Peripheral-to-Memory
P2P	Peripheral-to-Peripheral
PC	Personal Computer
RTL	Register Transfer Level
SV	SystemVerilog
SW	Software
TB	Test-Bench
TSDMA	Tropic Square DMA
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Introduction

The modern world heavily relies on computers, yet most users are unaware of the sophisticated data exchanges that occur within these systems. When we think of computers, the Personal Computer (PC) typically comes to mind, so the next example is using it. Even before users begin actively using the PC, it is already engaged in various data transfers, such as tracking mouse movements, registering keyboard inputs, and displaying static images. These transfers are continuously taking place in the computer's memory, often without the user's knowledge. In systems other than PCs, the process is similar – various components within the system constantly communicate with each other, exchanging vital data. What many fail to recognize is the significant demand on the Central Processing Unit (CPU) to handle these transfers while simultaneously juggling other tasks. Moreover, crucial data transfers can interrupt ongoing activities, potentially hindering task completion or rendering the computer unresponsive, despite the user's attempts to interact through clicks and commands.

Data transfer is a process of relocating data from one place in memory to another. There are various modes for it and they can be categorized into two groups: those controlled by the CPU and those that operate independently. A brief overview of their main difference is the following.

The first group – managed by the CPU – already gives a hint that the main processor's active participation in data transfer is required. Two similar modes only differ in the way of triggering a transfer. Either the processor keeps checking on the Input/Output (I/O) device for availability and then proceeds to complete the transfer or I/O device sends a request to CPU when it is ready. In the latter case, the peripheral has to wait until the CPU saves the context of its current task and switches to data transfer. In both modes processing unit postpones any activity and fully commits itself to transferring the data.

The second group consists of Direct Memory Access (DMA) mode that does not need the CPU to intervene in data transfers at all. A separate controller – DMA controller (DMAC) – is connected directly to memories through the system interface. All data necessary for a transfer is stored in DMAC's configuration registers. Since the CPU neither initiates nor performs the transfer – it can complete more important and complicated tasks.

When it comes to chip design – choosing a better option between these modes should not seem complicated. DMA undoubtedly overcomes modes where transfers are controlled and run by the CPU. To sum up, DMAC is a control unit that speeds up data transfers in computer systems and enhances the performance of the entire system by enabling DMA mode. Nowadays they play an inevitable role in microchips by handling most of the data transfers without loading the CPU.

The particular DMA controller, that is being created according to the assignment and from now on referred to as TSDMA (Tropic Square DMA), is intended to be placed in a microcontroller. The chip will be designed by Tropic Square s. r. o. – company developing in the niche of chip design. Their goal is to produce products that are open-source and can be utilized in

crypto wallets, with a focus on security and resilience against vulnerabilities such as Meltdown and Spectre, as well as other micro-architectural attacks. The decision to create a new DMAC specifically for the company, rather than purchasing from a third party, was made after careful consideration of various factors. The most significant factor is the avoidance of signing a non-disclosure agreement with the IP core provider, which would prohibit the ability to expose the code to the public.

By designing the DMAC in-house, the company will have greater control over its verification process, allowing any potential issues to be identified and addressed before they become major problems. Other reasons are additional expenses and lack or excess of functions and flexibility of controller, which most likely could be integrated into chips with equivalent configuration only.

Over the years numerous examples of DMA controllers were introduced and there is a difference in the purpose they serve and, hence, the set of features and configurability. Despite many of them being quite general-purpose and suitable for use by different systems, having your own DMAC designed specifically for the case yet with a flexible interface is a good solution when it comes to open-sourceness and narrow use. Currently, TSDMA is meant to fulfill the requirements of one exact product only, however, in the future, it has the potential to be adapted and utilized by others according to their specific needs.

In this thesis, we will commit to the analysis and review of a DMAC architecture, a brief overview of existing DMACs, and the design of such a controller. Following synthesis, tests, and verification of the circuit will not be fully covered in the text since it reaches out of the main goals of the thesis, test-benches will be provided by the company's verification engineers.

Goals of the thesis

The main goal of the thesis is to design and implement a generic, configurable DMAC for a new product by Tropic Square s.r.o. company. Next, to investigate the topic and select the most useful aspects among open-source implementations of DMA controllers to resolve in the circuit. As the given microchip aims to be as open-source, small, and efficient as possible, one of the solutions is to devise TSDMA with the most appropriate set of features and configurations to fulfill the goals of the chip.

TSDMA shall support all kinds of data transactions, e.g. Memory-to-Memory (M2M), Peripheral-to-Memory (P2M), Memory-to-Peripheral (M2P), and Peripheral-to-Peripheral (P2P) through a particular type of system interface. It also should be configurable in terms of a number of channels and hardware interrupts by having generic variables. The implementation shall contain all key features of DMA controllers and at the same time not be congested by excessive traits. Lastly, design should respect future implementation not only in Field Programmable Gate Array (FPGA) but also in Application Specific Integrated Circuit (ASIC), hence, it should minimize the use of area on the chip.

This thesis will be organized in the following chapters:

Chapter State of the art covers the fundamental definition and theoretical background of computer structure and architecture related to the topic. Within this chapter, DMA controllers are introduced, and both their basic and advanced features are thoroughly described.

Chapter Analysis This section of the thesis is devoted to evaluating and analyzing publicly available open-source DMACs for microcontrollers with configurations similar to the aimed chip for DMAC integration.

Chapter Design guides the reader through every step of the design of TSDMA block.

Chapter Testing explains the main points of TSDMA testing using basic and UVM tests.

Chapter Conclusion contains results of the work done in this thesis and a description of possible future improvements.

Chapter 1

State of the art

This chapter provides theoretical background for DMA controllers, their basic features and operation principles are described in Section 1.2. Advanced functionality and configurations are described in following sections. Related definitions and terms that are used in context of DMAC will also be introduced below. Following text should help with better understanding of context of the thesis and briefly explain all necessary terms used for following analysis and design of DMAC.

1.1 System Bus

For better understanding of integrating inside the system and communication between the components we need to introduce system bus shown at Figure 1.1.

▶ **Definition 1.1** (System bus)**.** *A bus is a set of wires that acts as a shared but common datapath to connect multiple subsystems within the system. A system bus provides a communication path for the data and control signals moving between the major components of the computer system.*

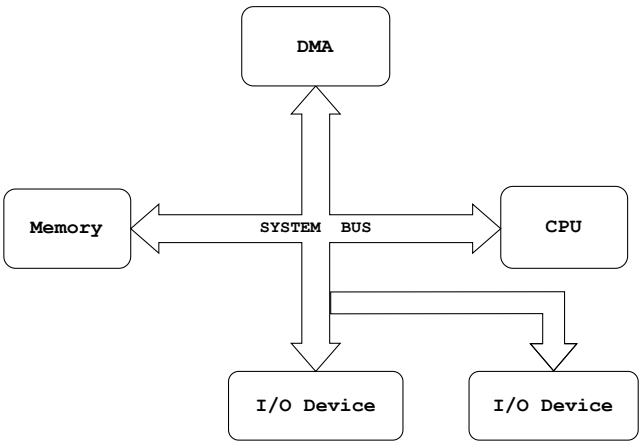

Figure 1.1 System bus connecting elements of computer.

The system bus works by combining the functions of the three main buses: namely, the data, address and control buses. Each of the three buses has its separate characteristics and responsibilities. The system bus combines the functions of the three main buses, which are as follows:

- The control bus carries the control, timing and coordination signals to manage the various functions across the system.
- The address bus is used to specify memory locations for the data being transferred.
- The data bus, which is a bidirectional path, carries the actual data between the processor, the memory, the DMA controller and the peripherals.

A bus can be point-to-point or a common pathway, however only one device at a time may use the bus. Often devices are categorized as *masters*, the ones who initiate transfers, and *slaves* – responding to these transfers. Operation of multiple devices on a bus is solved with bus protocols [1, p. 179-182].

1.2 Basics of DMAC

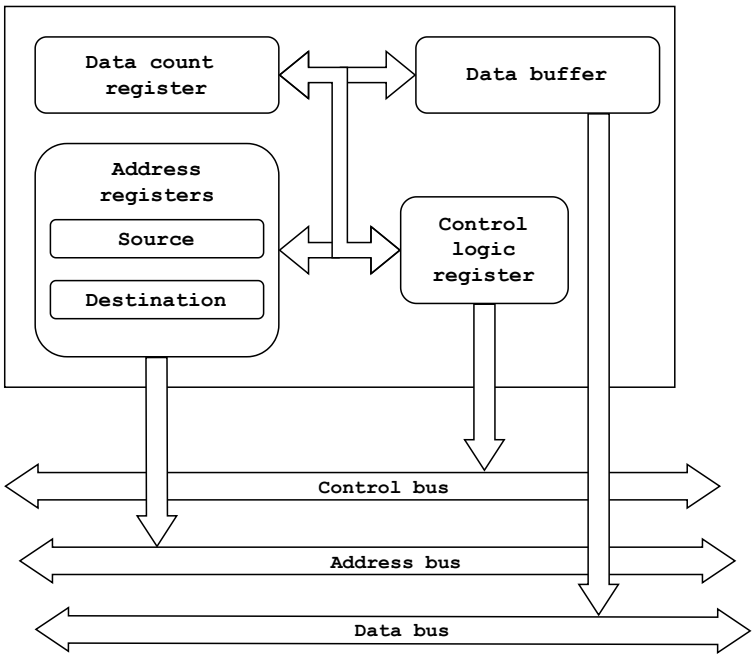

Figure 1.2 Block diagram of simple DMAC

To understand structure of DMA controller and how this memory access mode operates we will look at the most simple controller demonstrated at Figure 1.2. *Data count* register is responsible for amount of transferred data. It is initially loaded with total amount of data to transfer and serve as watchdog. With each transfer it decreases number of them left to the end of transaction. When the value reaches zero, *control logic* generates interrupt to notify other

components in system about finished transaction. *Address* registers store source and destination addresses of data to be transferred. When no configuration is available, their value is being increased inversely to *data count*. Last register is *control logic*. It plays the role of a DMAC's brain, coordinating internal events and operations. Besides controlling inner state of DMAC this register generates output signals for communication on system bus and processes input signals. DMA data transfer is executed in following way:

- Software(SW) configures necessary registers as source and destination addresses, number of data blocks to transfer.
- DMAC gets triggered, reads data from source address into local buffer, then writes buffered data to destination address.
- DMAC repeats previous step until size of transferred data equals number of data blocks set by SW.
- DMAC sends interrupt after transfer was completed.

Entire transfer then happens without CPU intervention and significantly increases performance of entire system by doing so.

Such module needs to be configured before every separate data transfer and it sends DMA request as far as it's data count register does not equal zero. That can work with minor systems and not so loaded transfer traffic where both source and destination are memory locations.

From these simple controllers to modern DMACs there is a huge step up in their configurability and capability. Multiple channels and transfer trigger sources, different kinds of memory to transfer from and to, bigger choice of address modification and much more.

1.3 Address modification

Addresses in memory point to separate blocks of data which could be of different size. For convenience, we will consider 32-bit memory organization. Each data block contains one word – 32 bits – of information stored. To avoid various problems including creating disorder in memory – addresses should be word-aligned as shown at Figure 1.3. One address holds 1 byte (or 8 bits) and to be 32 bits aligned it should be divisible by 4.

Additional detailed information and clarification regarding memory organization and addressing can be found in [1, p. 186-189]. Some of the possible configurations regarding address modification within data transfer can be seen at the end of this section on Figure 1.5.

1.3.1 Increasing and Decreasing mode

At minimalistic DMA controllers source and destination addresses are being changed inversely to data count. At the time it was the only possible mode and is called increasing mode. When a chunk of data is stored in the main memory, it is saved sequentially, block by block from lower to higher address. Thus to read or write such data in memory the knowledge of the starting address, width of block, and number of blocks is sufficient. The source or destination address shall be increased after each read or written block by its size divided by 8 – in this case, it is $32 : 8 = 4$. In other words, we add the number of bytes in the data block after each operation on a given address.

In various systems, data is stored starting from the end address. Provided that information increasing mode should be reversed. Operation of address increment turns into decrement and, hence, increasing mode becomes decreasing. In decreasing mode size of one block in bytes is subtracted from the address.

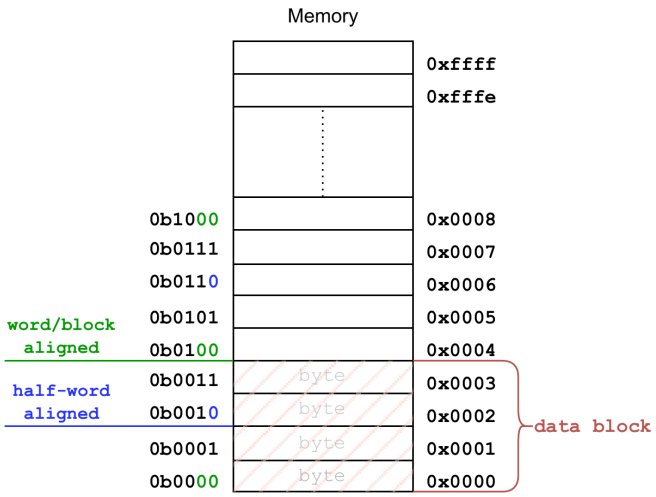

Figure 1.3 Memory organization

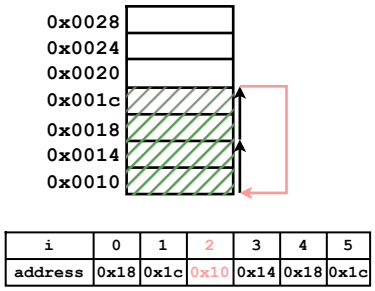

Figure 1.4 Address values for wrap16 addressing mode example.

1.3.2 Wrap mode

In a world full of PCs and other computers using I/O devices – it became unavoidable to perform data transfers with peripherals on either side. The main difference between data in main memory and data in peripherals is the speed of data changing and limited space in the latter. Usually, storage space in peripherals is restricted to one block size or buffer with a small amount of data blocks that could be stored there.

From a peripheral point of view, it is possible to implement buffers in different ways. The main distinction is whether it is a circular buffer or a (multi-byte) shift register. In a circular buffer, it iterates through addresses and wraps back to the starting address after reaching the end of the buffer. Shift register shifts its content and writes to and reads from one point at memory – one address.

Using the wrap mode for a circular buffer is a practical choice. It functions similarly to the increasing mode, with one variation. Instead of simply increasing the address, the last address will have the same value as the starting address, effectively wrapping back to the beginning when it reaches the end of the memory block (Figure 1.4). This mode is usually used with a number of bytes in its name, according to the size of a memory block.

For example, wrap32 wraps on addresses divisible by 32. A buffer with such size contains 8 words – 8 data blocks. Iterating through memory addresses for a 16-bit sector of data would look like shown in Figure 1.4.

You can observe the way addresses in the DMAC's configuration registers will circulate in closed memory space and not overflow out of buffer bounds. In a scenario where the size of the buffer is one data block, there is a more elegant solution in the following part.

1.3.3 Fixed mode

Specific case of peripheral buffers can lead to another addressing mode in data transfers. When buffer size is one data block there is no need to change the address since every time read or write will be executed at the same place in peripheral memory. Thus address in fixed mode remains unchanged during the entire data transaction. This mode can be used for different cases such as copying data from one data block to multiple locations in memory or expanding the value.

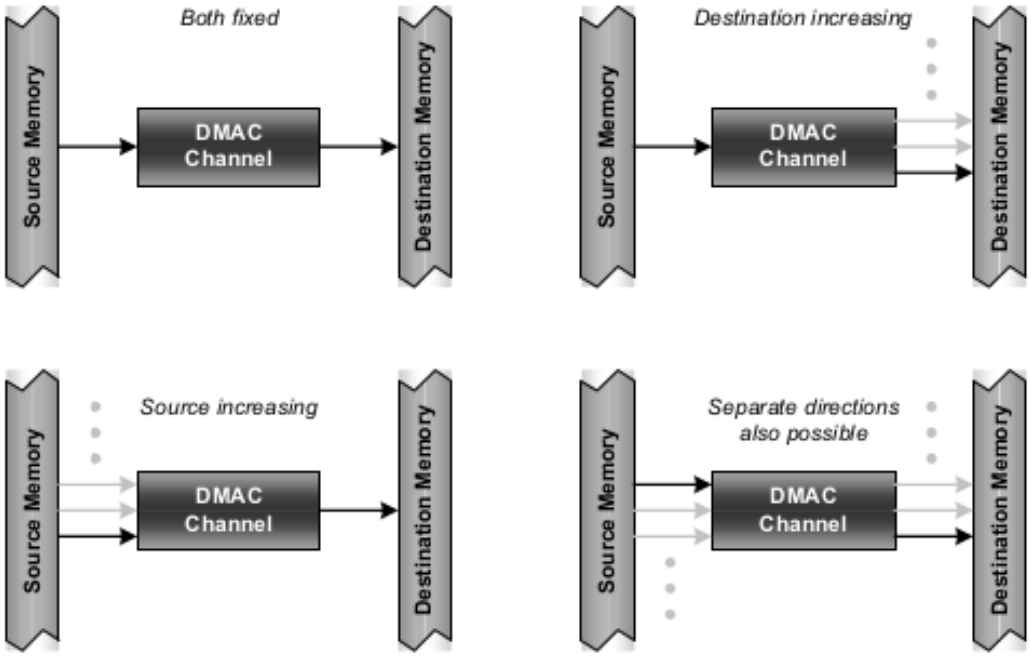

Figure 1.5 Addressing modes examples [2].

1.4 Scatter-Gather

Scatter-Gather represents an advanced method of data handling within computer systems. Rather than using a linear data transfer approach from a single source to a single destination, scatter-gather operates by efficiently gathering data from multiple distributed locations in memory or scattering it across various destinations.

This technique is highly beneficial in cases where data exists non-contiguously across separate memory segments or when the processing of multiple data streams is required without disrupting CPU operations. By allowing the DMA controller to access and combine scattered data blocks into a single, uninterrupted stream, this approach reduces overhead and greatly improves data transfer efficiency. This functionality is especially valuable in systems with disparate memory layouts or fragmented data, as it allows for one channel to gather data in a continuous block and other channels to utilize it for various operations at a later time.

The utilization of Scatter-Gather is crucial in storage systems, particularly in the management of file systems or databases. This technique handles the distribution of data among different segments in memory. In multimedia processing, Scatter-Gather plays a key role in the efficient handling and transmission of multimedia data streams, such as audio, video, or graphics, which may be located in various memory locations. Ultimately, its implementation leads to improved data transfer operations and provides an essential approach for managing complex computing environments with non-linear data.

1.5 Transfer modes

DMA does not require CPU intervention in data transfer, but this does not necessarily mean that the CPU can operate completely independently. Whenever data needs to be transferred, the system bus (see Definition 1.1) is actively used, which restricts the main processor's ability

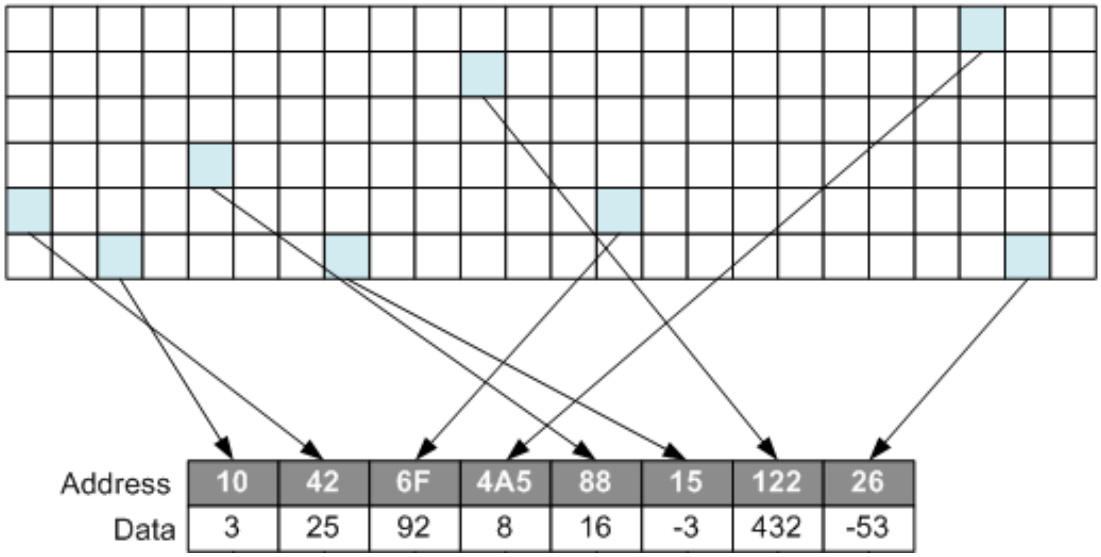

Figure 1.6 Scatter-Gather principle [3].

to execute tasks that rely on bus communication. This means that the CPU may be blocked by DMA if it needs to use the bus. Additionally, DMA offers different transfer modes with varying transfer and bus loading rates. Essentially, these modes determine how long the CPU will be blocked and when it will regain control of the bus. It is difficult to determine which mode is better or worse, as they each have distinct characteristics and are suited for different purposes.

1.5.1 Cycle stealing mode

Peripheral devices take some time to prepare or process data in a buffer, hence, it would not be nice for DMAC to block the bus while waiting for data to be ready. DMAC gets control of the bus for one transfer cycle each time data are ready to be transferred. After that CPU takes back ownership of the system and uses it for other tasks until the next chunk of data is prepared. Then this loop continues as far as DMAC's data counter is positive [4].

This method is not the most optimal from DMA's nor CPU's point of view yet it combines reasonable data transfer rate, while still allowing the bus not to be clogged with DMA and let CPU use the bus consistently. Most suitable use for P2M or M2P type of transfers.

1.5.2 Burst mode

In *burst mode* entire block, whose size is defined by a number of bytes in the data count register, is being transferred continuously. CPU is being blocked by DMA from having access to the bus during the entire burst of data. Burst mode is advantageous for transferring significant amounts of data as it minimizes the overhead associated with frequent resource acquisition and release. It gives DMA a very high transfer rate since it transfers multiple data at once maximizing utilization of sources without overhead caused by the constant obtaining and releasing of bus and sources. Hence, there is no need to wait for CPU to assign the ownership of the bus for every single data block.

Data transfer's throughput is at its highest in burst mode, however, CPU remains blocked from bus operation for longer periods of time.

1.5.3 Interleaving/Transparent mode

Interleaving, in some sources *Transparent*, the mode of data transfer is based on using a data bus for transfers only when the CPU completes tasks without the need to access the bus. In other words, DMAC will wait for as long as it can take for the CPU to finish a current task and free the bus. Once the bus is requested by CPU - DMAC stops the transfer and relieves bus ownership until it is free again.

This mode has slowest transfer rate, but is arguably the most efficient from system's point of view, since CPU is not getting blocked by DMA yet transfers are still being completed. It can be used for M2M transfers as they usually do not require high speed or immediate completion.

1.6 Data Transfer

Nowadays, as the range of options for transfers and their purposes increase – it is natural that it can be useful to control how much of the total data is needed to be transferred at one trigger impulse. A trigger is a specific signal or event that initiates or activates the start of a data transfer process. It is used as a directive prompting a device or system to start either sending or receiving data. This signal's generation can occur through different methods, including manual input, a preset timing system, or triggered by specific conditions, tailored to the unique demands of the data transfer operation.

1.6.1 Beat transfer

Computers can vary in the size of the data block they have, the size of data transfer at once can be as big as the width of the data bus. *Beat* in this text represents the smallest unit of data transferred between components within a system and varies based on the system's data bus width. For instance, in an 8-bit microcontroller, a beat typically corresponds to a byte transfer, while in a 32-bit system, a beat refers to a transfer of 4 bytes due to the wider data bus, reflecting the amount of data moved in a single transfer.

Thus, not to confuse things talking about the smallest unit of transfer regardless of it's size, it is called beat. It takes one transfer cycle and cannot be divided into smaller items.

1.6.2 Burst transfer

Burst transfer refers to a method of consecutive transferring blocks of data between components in a system without interruption. It provides moving a sequence of beats in rapid series within a single operation. Burst transfer is often supported directly by bus interfaces. This type of transfer is introduced in multiple bus architectures, e. g. IBM CoreConnect, ARM AHB/APB/AXI architectures, and others, which enable uninterrupted continuous sequence of data transfer cycles.

1.6.3 Block transfer and Repeat mode

Block transfer specifies the entire data transfer of a certain amount of data defined by the data count register. Can be divided into burst or beat transfers and be completed in several data transfers triggered by multiple interrupts. It is also possible to configure DMAC to perform an entire block transfer at once.

Size of the block here is restricted by the configuration register, in order to perform bigger transactions a *Repeat mode* was introduced. It allows to perform the same block transaction continuously without reconfiguring DMAC to the same setting several times. A number of repetitions is either given by a number in the corresponding register or is infinite which allows for a continuous and constant stream of data. It is good for constantly renewing data as in video or sound stream.

1.6.4 Transaction transfer

Repetition of the block transfer can come in handy in cases of renewable data in the same memory area. To ease the process of such transfer and its configuration next register that appears in controllers is the repeat counter. It is used to perform the same block transfer several times.

Transaction transfer stands for all operations with data, i. e. performing block transfer number of times defined by repeat counter.

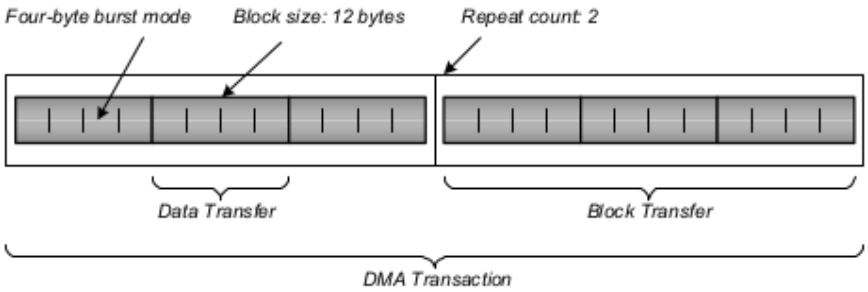

Figure 1.7 Beat, block and transaction DMA transfers [2].

1.7 Channels

Data channels are essential components within a DMA controller, functioning as specialized lanes that streamline data movement without requiring constant intervention from the main processor. These channels operate as dedicated pathways, each assigned with specific tasks for independent data transfer. By utilizing these pathways, the DMA controller can efficiently handle multiple data transfers concurrently. This means it can simultaneously oversee data being transferred from various devices and peripherals, resulting in a faster and more effective process overall.

The number of channels available varies depending on the design of the DMA controller. Some systems may have a small number of channels, while others may offer more, allowing for a higher capacity of simultaneous data transfer tasks. These channels are crucial in enhancing the efficiency of data movement. By enabling multiple data transfers to take place concurrently without taxing the main processor, they significantly contribute to the overall system performance, particularly in tasks involving large amounts of data.

1.8 Interrupts and triggers

An interrupt is a crucial signal that is sent from one component of a system or program to the CPU to signify an occurrence of an event. It serves as a way to notify the main processor about a particular event that has taken place and requires immediate attention. One common way interrupts are implemented in HW is by setting the interrupt signal active when both the interrupt flag and interrupt enable are active.

When an interrupt is received, the current program in operation temporarily pauses to allow for the execution of an interrupt procedure. This procedure, usually a smaller subprogram, is responsible for executing the necessary steps to handle the data or event that triggered the interrupt. Additionally, it is common for the interrupt flag to be cleared during this procedure to let the other side acknowledge that the interrupt has been successfully processed. A single

system component may generate multiple interrupt events, each with its own signal specifically assigned to one event. Alternatively, there may be just one signal with multiple flags indicating different events. In either case, the CPU's interrupt procedure must retrieve the source of the interrupts and determine which event has occurred.

Similarly, not only the CPU but other components within computer systems as well can respond to incoming signals, similar to interrupts, and perform specific tasks. These signals are called triggers. In terms of DMAC, they are often responsible for initiating data transfer. Triggers and interrupts may arise from both software and hardware sources. In other words, they can be activated through manual signal inputs or be automatically triggered by events occurring in various hardware components.

In addition to the CPU, other components within a computer system can respond to incoming signals, similar to interrupts, and perform specific tasks. These signals, known as triggers, are often responsible for initiating data transfer within the Direct Memory Access Controller (DMAC). Whether generated by manual signals or events within various hardware components, triggers, and interrupts can be triggered by both software and hardware sources.

1.9 Arbitration

Arbitration is a process of choosing one of the sources requesting for grant to have access to shared resources, rejecting all other sources at the moment. When faced with multiple data channels, the dilemma arises of how to choose and prioritize among them, especially when more than one transfer is ready to be completed simultaneously. This is similar to bus arbitration, where multiple bus masters request for bus access, but within a single component: which channel will be given priority at the current moment.

Many solutions and variations exist for bus arbitration algorithms, but they can generally be classified into four types [1, p. 385]:

- Round robin: the first type is round-robin, where all sources are granted access in a circular rotation and are given a limited time balanced with respect to other requesting devices.
- Daisy chain: Another solution is fixed priority, or daisy chaining, which grants access to the first encountered requesting device, usually starting with the lowest index. See example in Figure 1.8.
- Centralized parallel arbitration: The third type involves using a custom priority system in which the algorithm for determining the first master to be granted access is determined by the settings of the specific system. Each device is equipped with a request control line and a centralized arbiter responsible for determining which one gains access to the bus. However, this arbitration process usually involves more complicated logic in the arbiter compared to the first two mentioned methods and may lead to bottlenecks.
- Distributed arbitration : This scheme is similar to centralized arbitration, but instead of a central authority selecting who gets the bus, the devices either determine who has the highest priority and who should get the bus or request the bus only when it is not busy. See example in Figure 1.9.

1.10 Control logic

The control logic unit is a vital component responsible for directing and coordinating the flow of data in a system. It is responsible for interpreting input signals and executing operations based on specific conditions or instructions. This unit coordinates the timing and functioning of other key components, including arithmetic units, memory, and input/output devices, to ensure that

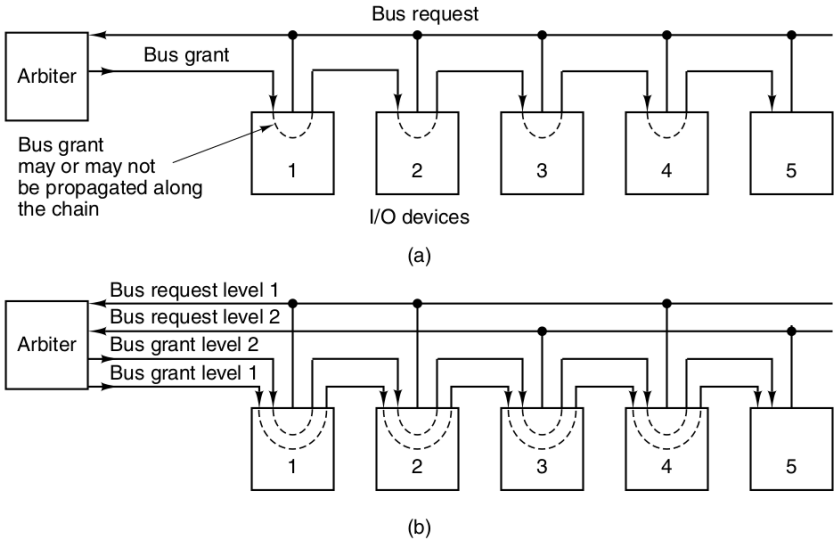

Figure 1.8 (a) A centralized one-level bus arbiter using daisy chaining. (b) The same arbiter, but with two levels [5, p. 197].

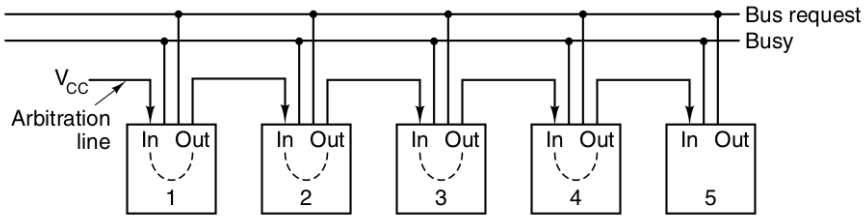

Figure 1.9 Distributed bus arbitration [5, p. 198].

data processing occurs accurately and efficiently. In short, the control logic unit is responsible for directing and controlling the overall behavior of a circuit, making sure that all operations occur in line with the defined instructions or program.

There are two ways we can guarantee the correct setup of control lines. The first is hardwired control, which physically links the control lines to the specific machine instructions. These instructions are then divided into fields, with individual bits being connected to input lines that control different digital logic components.Alternatively, we can use microprogrammed control, which utilizes software made up of microinstructions that execute the microoperations of each instruction [1].

1.10.1 FSM

A Finite-State Machine is a mathematical representation of computation. Essentially an abstract machine, that represents sequential circuit. It consists out of states and transitional function from pair of state and input to next state. The FSM is able to transit from one state to another in reaction to outside influences or when certain criteria are met. To define an FSM, one must specify its states, initial state, and the requirements for each transition [6].

The FSM provides output and their timing is based on the type of the machine – Mealy or Moore. Mealy FSM forms its output based on the current state and input, meanwhile, the second option, Moore FSM, generates output depending solely on the current state. For capturing all the information about FSM it is often being represented as a state transition diagram in Figure 1.10. In design then it is represented with one register for the current state and combinational logic taking the current state and input to FSM as an input and providing the next state as an output, see Figure 1.11.

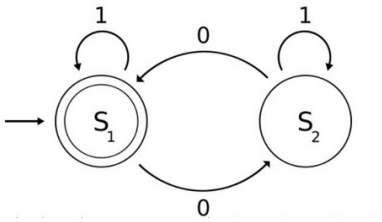

Figure 1.10 FSM with two states, where valid input are binary strings with even number of zeros

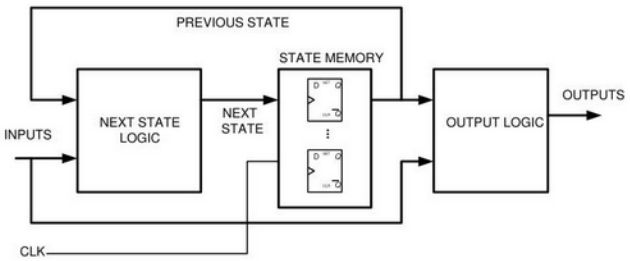

Figure 1.11 Block design of Mealy FSM

1.10.2 Microprogrammed controller

A microprogrammed controller is a control unit that stores binary control values in memory as words. It arranges a series of signals with each clock cycle, which gives the ability to generate the necessary instructions for execution. These output signals trigger a specific micro-operation, such as a register transfer. Ultimately, this process creates distinct micro-operations that can be saved in memory through a set of control signals.

An advantage of this technique lies in its flexibility, as it allows for a customizable definition of the system's operations. Furthermore, it streamlines the execution of complex tasks by breaking them down into smaller, more manageable instructions stored as microcode. These microinstructions are conveniently accessed from a control memory, functioning as the conductor of operations and ensuring the smooth and efficient functioning of the system.

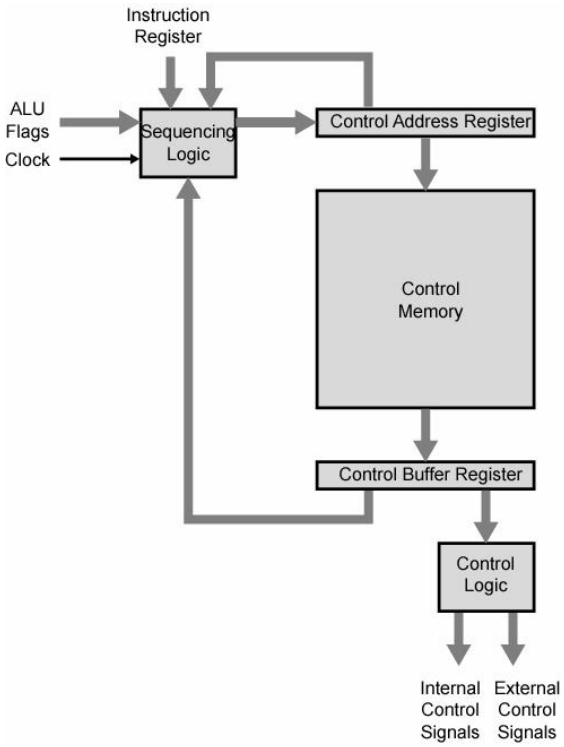

Figure 1.12 Microprogrammed Control Unit Organization [7].

Chapter 2
Analysis

In Chapter 1, we have provided a theoretical overview of DMA controllers and data transfers. We explored the widely utilized functionality of DMACs and gained an understanding of where these separate features are most useful and applicable. In this chapter, we will delve into potential approaches for designing and reasoning behind the inclusion or exclusion of functions in the chosen set of features for TSDMA.

To complete a given task we need to decide a strategy that will be used for design. To refresh our memory: we need to make a design suitable for FPGA and ASIC applications, fast and versatile, generic and with flexible descriptor configuration.

2.1 FPGA vs ASIC

When it was mentioned that the design shall target both technologies it is good to understand what are the differences and how to change design depending on technology and whether it is needed at all. Both ASIC and FPGA are microchips that are used for electronics design.

The main difference is that FPGA is an already manufactured chip with programmable logic that can be reprogrammed and is dedicated for general purpose utilization. Such board is usually equipped with a large amount of flip-flops – registers for keeping the data – so they can be easily overused. This allows us to make design easier by not implementing hard combinational logic into it and just keeping things easy and mostly sequential.

Meanwhile, ASIC is a microchip designed for one specific application and cannot be reprogrammed or somehow modified once it is manufactured. The cost of flip-flops on ASIC is bigger than other logic gates, so it is more common to focus more on combinational logic instead of sequential and using registers to save values.

The choice of design style is heavily influenced by the intended technology, and their respective approaches can vary significantly. In the case of TSDMA design, the use of FPGA is not a primary objective and is solely utilized for the final verification of the overall IC. Because of the smaller size of TSDMA, there is no need for a separate FPGA-specific version, as the design created for ASIC will not exceed the limitations of the FPGA board.

The decision not to adjust the design for FPGA was made during one of the consultations with the team. For verification and validation purposes of TSDMA itself, RTL simulations based on UVM testbenches are sufficient.

2.2 Set of features

There is no exact microcontroller yet for TSDMA to be placed in, but there are certain requirements for a system that we will follow and consult with the team of designers to get to better decisions when there are several equally functional options according to given requirements.

Information given about the chip is its 32-bit memory organization and system bus width, AHB-Lite protocol used for communication within the microcontroller, and targeting ASIC technology. Even though the number and names of components of the systems are not known yet, it is expected that the IP core has a similar range of components and basic functions as STM32 ICs, a list of them can be found in [8].

To know which features to choose it is necessary to understand where the versatility starts and ends and which aspects can be solved as generic variables. The TSDMA operates solely through hardware, without the involvement of firmware, which limits its inherent versatility. Nevertheless, the TSDMA can still offer a considerable degree of adaptability through its extensive range of configurable features in both channel descriptors and DMA configuration.

We will compare the features of DMAC in several microcontrollers and determine if they would be a valuable addition to our design. After consideration and studying several open-source DMA controllers it has been determined that the functionality of the following ICs is closest to the expected functionality of TSDMAC:

1. STM32Fxx
2. Atmel AVR 8-bit (XMEGA DMA Controller)
3. PIC33/PIC24

Their features are compared side by side in Table 2.1. revealing subtle distinctions between them. However, overall, all three groups of microcontrollers offer similar DMAC functionality that we will take into consideration for our controller.

In determining which features should be included in the final implementation, we must prioritize maintaining an uncomplicated design. This means considering if certain functionalities can be achieved through alternative configurations. We should prioritize implementing the simplest and most commonly used features, while more complex ones that encompass multiple functions may not need to be directly incorporated, as they can be accessed through the basic ones.

- Address modification

  Fixed and incrementing address modes play a crucial role in data transfers, hence, we confidently select them as our primary options. Decrementation of addresses is determined not to be necessary for TSDMAC due to their low usability in a system. However, by utilizing Scatter-Gather mode and having each address in the descriptor be one block lower than the previous one, we can achieve the same effect with certain limitations. If a decrementing mode becomes necessary in the future, we are fully prepared to expand our addressing modes without any difficulties.

  The inclusion of address reload after block transfers seems implicit, making it a non-configurable value for now. Conversely, enforcing address reload at any point other than the end of a block may only serve to complicate matters unnecessarily. Thus, another solution would be to implement a wrap mode of fixed size that enables the system to remain within a fixed memory section, regardless of the amount of transferred data.

- Transfer modes

  While it's true that two out of three controllers offer single and burst transfers, these are not necessarily essential features. Initially, the mentioned transfer modes were considered to be a part of the configuration, but after numerous discussions and reviews, we ended up

	STM32F	Atmel AVR	PIC33/PIC24
Address modes			
Fixed address	yes	yes	yes
Address incrementation	yes	yes	yes
Address decrementation	–	yes	yes
Address reload	yes	yes	yes
Peripheral indirect addressing	–	–	yes
Transfer modes			
Single transfer	–	yes	yes
Burst transfer	–	2/4/8 byte	yes
Block transfer	yes	yes	yes
Repeated transfer	–	yes	yes
Stream transfer	yes	yes	yes
Channel arbitration modes			
Round Robin	–	yes	yes
Fixed priority	lowest index first	yes	lowest index first
Custom priority	up to 4 levels of priority	channel 0 or both 0 and 1 – highest priority, round robin for the rest	–
Triggers and interrupts			
Disable/Enable interrupt	yes	yes	yes
Completion interrupt	yes	yes	yes
Half-done interrupt	yes	–	yes
Error interrupt	yes	yes	yes
Additional interrupts	–	–	yes
HW triggers	fixed set per channel	configurable per channel	configurable per channel, up to 128 sources
SW trigger	yes	yes	yes
Other			
Channels	12	4	n
Size of transfer	8/16/32-bit, source/destination independently	8-bit	8/16-bit
Scatter-Gather	–	–	–
Size of data	up to 65536	up to 65536	up to 65536

Table 2.1 Features of DMAC in STM32Fxx, Atmel AVR 8-bit and PIC24/PIC33 IC's

eliminating these modes from the feature set of TSDMA. This is because the same result can be achieved by using block transfers of corresponding sizes. It is not worth it for us to include these as separate features yet. However, it is more important for our design to have repeated and stream transfers as they have a high potential for practical use in peripherals.

- Channel arbitration

  Fixed priority is a fundamental basic feature that may be a reliable method for resolving conflicts, yet it alone cannot fully meet the objectives of the TSDMA strategy. After assessing the strengths and characteristics of different arbitration modes, we have concluded that a mix of Round Robin and Fixed priority, along with a Custom priority mode, is the relevant approach. Offered by us Custom mode guarantees that each channel is assigned a unique priority value, allowing users to have complete control over channel priorities. Furthermore, in situations where multiple channels hold the same priority value, the channel with the lowest index is given precedence and will be served first.

- Interrupts

  Undoubtedly beneficial interrupts are completion and error interrupts. These two signals provide essential information about a transfer, indicating whether it was successful or not, or if it even finished at all. On the other hand, the Half-done interrupt only informs the system that the transfer has reached the halfway point, which may be useful in advanced systems, but for our specific scenario, it appears unnecessary and will not be incorporated. However, despite the profit of Error interrupt it will not be included in TSDMA for now.

  As an additional interrupt in the PIC family, there is an interrupt signal that indicates when data written to the internal buffer has not yet been successfully transferred to the destination memory. This feature is particularly beneficial for systems with larger buffers and the ability to put the DMAC in sleep mode. This allows for uninterrupted re-enabling and resuming of the same transfer process.

- Triggers

  Reviewing triggers, based on several other subsystems in a mentioned early Tropic's IP core shown in Figure **??**, we decided to include up to 31 distinct hardware sources for event triggers in TSDMA. Since the exact number of external trigger sources will be known at the time of integration of DMAC in the system, this will be one of the generic variables. Hence, it can be altered accordingly to accommodate the appropriate number of hardware trigger sources lately. The significance of software triggers in M2M data transfers has been taken into account and was included in our design without bigger consideration. TSDMA will have a maximum of 32 diverse trigger sources, with each channel having the capability to independently assign any of them, thus giving users complete control over their priorities.

- Other features.

  As required by the assignment, to ensure versatility the number of channels can be a generic variable. Taking into account the size of the microcontroller, it has been determined that the maximum value will be set at 32.

  The size of the transfer will be fixed, 32 bits, with very simple reasoning – targeted microcontroller has a 32-bit architecture, and all transfers within the system are planned to be of the same size.

  The block size will range from 1 to 65535 as in reviewed ICs, and when combined with repeat mode, it will enable the transfer of up to 2^{32} words, providing ample flexibility for data management.

  One feature that is not included in any of the compared controllers but can be seen in Table 2.1 is Scatter-Gather. It was mentioned during one of the consultations and after a discussion

with Ondrej Ille, we decided to include this feature in TSDMA controller. Scatter-Gather is beneficial for DMA in numerous ways, which are described in Section 1.4.

2.3 Controller

There are various approaches to implementing control logic, with two key contenders being the *Micro-programmed controller* and *Finite State Machine* (FSM). These two solutions each have their unique strengths and weaknesses. We'll analyze these traits to understand where they excel and where they might have limitations.

Using FSMs allows for a simpler and faster design and implementation process compared to micro-programmed controllers. This is due to their direct hardware-based approach, making them well-suited for smaller ICs. Micro-programmed controllers, on the other hand, tend to be more complex and demand additional hardware resources due to the need for microcode storage and control logic. As a result, their execution may be slower because of cycles for micro-instruction sequencing and interpretation. Additionally, the larger chip area and resource utilization required for microcode storage can limit their feasibility for smaller ICs with limited resources. FSMs have minimal hardware overhead, making them a viable option for applications where chip area and resource utilization are critical. They also exhibit predictable behavior, making them suitable for applications requiring straightforward sequential logic control.

Through the use of microcode sequences, micro-programmed controllers offer great flexibility, allowing for easy adaptation to changes. Their ability to handle complex logic makes them suitable for different applications, without the need for hardware modifications. With the simple modification of the microcode, adjustments can be made to the logic and instructions without direct hardware changes. In contrast, FSMs rely on hardwired sequential circuits, which can make it challenging to adjust control logic and implement modifications. As the control logic increases in complexity, FSMs may prove difficult to design and may encounter limitations. To alter their behavior or functionality, a significant redesign is often required.

2.4 Datapath

In order to effectively manage data movement and processing, it is necessary to include supplementary modules that are controlled by the central processor. Despite the fact that standard DMACs feature four additional registers in addition to the control logic, this may not be sufficient given the potential for multiple channels of varying numbers. Consequently, it is not feasible to allocate address, buffer, and counter registers for each channel within a single block. As the development of TSDMA continues, it becomes clear that the datapath will require more than just these four registers to successfully accomplish its objectives.

2.4.1 Channel Descriptor

Each channel will be assigned a descriptor that contains all or most of the features relevant to that specific channel and its transfer. This includes the source and destination addresses, as well as a data counter register. Additional features pertaining to these registers or specific modes of the channel/transfer may also be present in this descriptor. These descriptors will be saved in memory and their starting address will be included in the DMA configuration. Since all descriptors are equal in size, we can effortlessly retrieve any channel descriptor by knowing the starting point.

When a channel is operating in normal mode with sequential memory addresses it is enough to store all related information in one descriptor. However, it is not sufficient for Scatter-Gather mode where different locations in memory are involved in one transfer. One of the possible solutions is to describe a list of descriptors in memory through the channel's descriptor.

2.4.2 Buffer

From the very start, the use of a buffer has been essential in storing data when transferring from the source address to the destination address. Naturally, TSDMA will require one as well. However, since there is more to be temporarily stored in our block, we will introduce a descriptor buffer. This will allow us to effectively work with the data transfer within the TSDMA module. As each descriptor is loaded from memory, it will be updated locally throughout the transfer process. Finally, at the end of the triggered activity, the updated descriptor will be written back to memory.

2.4.3 Trigger Register

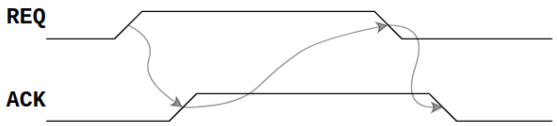

Figure 2.1 Four-phase handshake.

The next thing to consider is how the transfers will be set off. The external triggers represented as interrupts from other components of a computer system may arise at any moment and it is not guaranteed that the DMA controller can process them right away. Hence, it is necessary to capture them in a register. The first idea was to acknowledge a signal with a four-phase handshake (Figure 2.1), but it could be too complicated taking into account that it should be implemented on both sides and involve modifying existing interfaces. The second and for now last idea was to capture the rising edges of incoming interrupts into an inner buffer for triggers.

2.4.4 Arbiter

Once triggers are captured channel is ready for data transfer, with a greater number of channels available at the same time comes the need to choose which channel to serve first. Arbiter will be the module responsible for this. As discussed in Section 2.2 it will support three different algorithms to choose a highest priority channel. Meanwhile, it seems easy to understand the design of *Round Robin* and *Fixed* priority algorithms, *Custom* priority mode has a few uncertainties in it. It has to pick the highest priority channel amongst all active and triggered ones. It is a linear problem if sequential logic is involved, but it could then take up to 32 clock cycles, so we will try to design a combinational solution in the following chapters.

2.4.5 Register Map

In order to communicate with DMA and effectively configure the block's inner registers, it is crucial to arrange them in a clear and logical manner and map them to the system memory. The register map must then be assigned an address in memory space before the controller is integrated into the chip. Additionally, the register map will feature a slave bus interface, allowing it to be accessed by other components. This interface will enable all necessary read and write operations for configuration registers not included in channel descriptors. It could manage such registers as flag or DMA status registers, customizable priority values, or the external trigger source for the channels. A major part of the register map including signals for bus communication will be generated by the DMAC's description using scripts.

Chapter 3
Design

The previous chapter provided the necessary information on which features should the TSDMA consist of and how these components function. After analyzing the system, it is necessary to design how to implement it in practice. First, the technologies to be used for the implementation shall be chosen. Then we will go through all the components and how they will be designed. The chapter serves as an overview of the design decisions that arose during the process.

3.1 HDL

When it comes to choosing a language for the design itself, there are two front-runners in the world of Hardware Description Languages (HDL) for Register Transfer Level (RTL) design: VHDL and SystemVerilog (SV). While the usage of SV for this design was dictated by the company, it's still worth considering and comparing these two options in order to gain a deeper understanding of why VHDL may not have been the preferred choice.

VHDL is a language that is characterized by its strong type system and verbose nature. While writing code in VHDL may result in longer lines of code, it also makes it easier to identify and rectify errors. On the other hand, SystemVerilog, which is reminiscent of C language in terms of syntax, is more concise and condensed. This allows for the compilation of "illegal" scenarios, such as comparing vectors of different lengths, and offers packed types that are advantageous for managing multidimensional buses. In the world of ASIC, SV is often preferred over VHDL, which has a more intuitive and straightforward structure. However, SystemVerilog boasts of a synthesizable subset that strikes a balance between brevity and expressiveness, surpassing what VHDL has to offer.

For instance, SV offers a range of blocks - such as always_ff, always_latch, and always_comb - that enable designers to easily distinguish between blocks implementing various types of logic. Furthermore, for always_comb and always_latch blocks, the signals to be included in the sensitivity list are automatically inferred, avoiding the potential for extensive bugs often encountered in both VHDL and SV always blocks. That being said, VHDL remains a viable option in this realm. Many software tools geared towards ASIC development have shifted towards supporting SV over VHDL, making it the preferred choice for companies and their design workflows, meanwhile, FPGA-targeted tools have a preference for VHDL. Given that the ultimate objective of this design undertaking is an ASIC microcontroller, it makes sense for the company's development approach and tool selection to lean towards SV, without completely disregarding VHDL [9].

Last but not least, a significant advantage of SystemVerilog in the context of verification and simulations of bigger designs is a lower chance for many unwanted and hard-to-find bugs

to arise, than in VHDL. This can be attributed to the nondeterministic scheduling of blocking and non-blocking assignments in VHDL, as well as the utilization of delta cycles in simulations, which may result in unexpected behavior on actual hardware. More information regarding delta cycles and scheduling in VHDL can be found at [10]. In terms of different types of assignments and events, SV has deterministic scheduling of processes of different natures to provide proper interaction between them within the RTL, Test Benches(TB), and assertions [11].

3.2 Design tools

Design flow tools are primarily defined by Tropic Square. The IDE selected for managing and editing source files is Visual Studio Code, a versatile cross-platform open-source editor that offers a broad amount of features and supports a lot of programming languages. It can be customized and expanded in numerous ways to suit individual preferences. For the purposes of TSDMA design, it was used for writing RTL source codes and test benches as well as different configuration files necessary for both simulation and synthesis processes.

For version control and tracking the progress of project Gitlab – web-based Git repository. Within this software, it is easy to handle every aspect of a project, from initial planning and source code management to ongoing monitoring and security measures. Designed to foster strong collaboration and drive superior software development with streamlined processes and increased efficiency, GitLab empowers teams to speed up production timelines and provide exceptional value to clients.

Synopsys VCS and DVE serve as simulation and verification tools. Synopsys VCS, also known as the Verilog Compiler Simulator, is a top-performing tool used for functional verification and debugging of digital designs. Its speed and precision make it a favorite among engineers for effectively simulating sophisticated designs. The Synopsys DVE – Discovery Visual Environment – is integrated with VCS and offers advanced visualization and debugging features. With DVE, engineers can analyze simulation results visually, trace signals, and efficiently debug designs, making it an invaluable aid in the verification of complex digital systems.

Finally, for synthesis, once more solution by Synopsys was used. Design Compiler RTL was specifically designed to simplify the optimization of timing, area, power, and test processes all at once. Equipped with topographical technology, it ensures a smooth and efficient design flow, providing faster results. This technology boasts an impressive accuracy rate of predicting timing and area within a 10% margin of post-layout outcomes, significantly reducing the need for repetitive adjustments between the synthesis and physical implementation stages.

While all of the above tools are undoubtedly powerful, it's important to note that in our TSDMA design, we will not be utilizing the full extent of their capabilities. However, if we were not planning on integrating this particular design into a larger environment later on, we could have taken advantage of more straightforward and cost-effective tools instead.

3.3 Datapath

3.3.1 Register map

All configurations not related to a data transfer should be stored in the register map. These configurations are being accessed during different states of DMA and it would not be rational to put them outside of the module. The following registers are going to be included in the register map:

- BLOCK ID – contains identification and revision codes
- DMA CONFIG – contains DMA enable bit, priority modes configuration, and software reset

- DESCRIPTOR ADDRESS – has a starting address of channel descriptors in memory
- CHANNEL ENABLE – contains channel enable bits
- SW TRIGGER – similarly to CHANNEL ENABLE register contains bits for indicating the presence of software trigger
- HW TRIGGER A-F – 6 registers used for configuring the hardware interrupt source separately for each channel
- CPU PRIORITY A-F – 6 registers for custom CPU set priorities
- TRANSFER COMPLETION – flags for every channel, indicating the completion of the entire transfer (data counter in the descriptor is zero)
- INTERRUPT ENABLE – allows interrupt generated by channel.
- SCATTER GATHER – defines whether the transfer on a channel is Scatter-Gather or not.

  More detailed information about the register map can be found in Appendix A.

3.3.2 Channels descriptor

Descriptors are stored sequentially in memory starting from the address contained in the register map. Number of descriptors in memory corresponds with the number of channels. A descriptor is loaded to the inner buffer before each transfer to get the information about the transfer's configuration. Hence, it has to contain all transfer-related instructions. It was already mentioned that it obtains source and destination addresses as well as counter register. The data counter contains two separate values being size of the block and the repeat counter. These are already three registers in a descriptor. Certainly, there will be one more for configurations itself. As mentioned a descriptor should be adjustable to either a channel's main descriptor when the transfer is happening within one sequential memory block on each side, or a Scatter-Gather descriptor pointing to a list of ordinary descriptors for non-contiguous data locations.

After consultations, it was decided that the following settings are going to be included in a configuration register of channel descriptor:

- Source and destination address modes – controls modification of the addresses during the transfer initiated by a trigger.
- Trigger action – determines which transfer should take place on a single trigger impulse.
- Scatter-Gather trigger action – regulate whether the entire list of descriptors or only the first one is being processed on a trigger impulse.

When the Scatter-Gather bit is set for a channel in a corresponding configuration register, then the descriptor is treated as a pointer to the list of descriptors, which are being executed sequentially from the top based on the Scatter-Gather trigger action bit in the descriptor. The source address register is considered as a starting address of the descriptors list and the data counter register contains the number of descriptors in the list. The destination address register is not being used in this case. The appearance of a descriptor in the list is the same as in ordinary sequential transfer.

All registers are used as described before with one exception – trigger action considered to have transaction transfer value, hence, it is impossible to configure a block transfer in this case. The entire transfer defined by this descriptor is executed at once. More details about the descriptor map can be found in Appendix B.

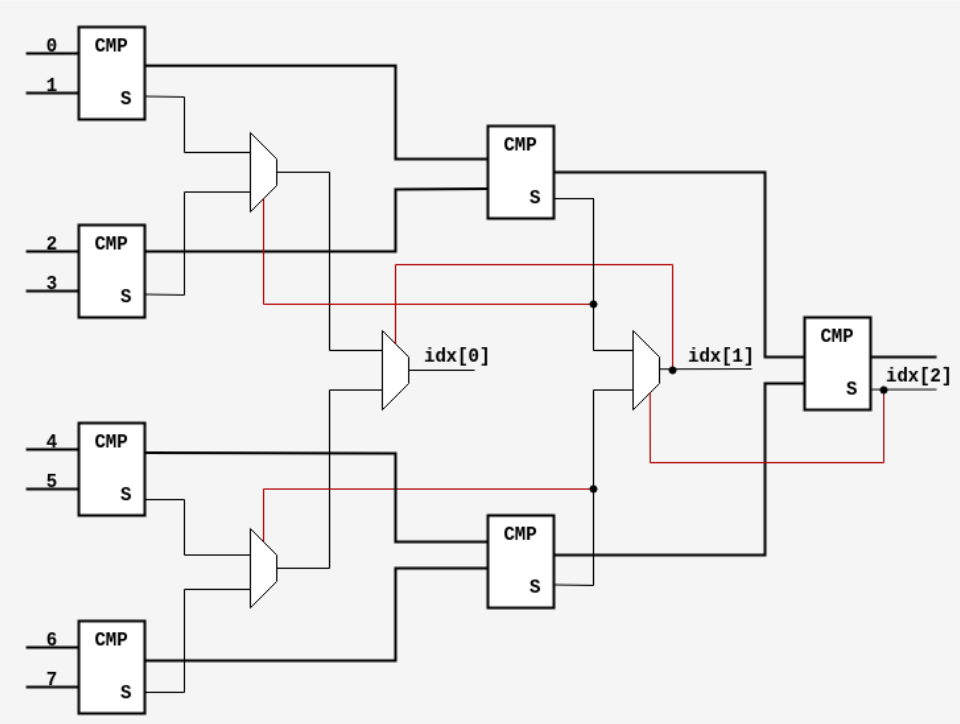

Figure 3.1 Block scheme of priority comparators for 8 channels.

3.3.3 Arbiter

The arbiter has to choose the next channel to be processed, so it needs to have information from the register map and trigger status about channel status and priority. It also should maintain the value of the chosen channel till the next arbitration cycle. The number of the channel is required by other components throughout the transfer, to calculate the address of the descriptor, for example, or to know which trigger buffer has to change its value because the trigger is being processed.

In understanding Fixed and Round Robin modes, the algorithms seem straightforward. However, looking into CPU custom priorities requires a discussion of options. As mentioned earlier, one approach involves sequential processing, where we save the channel number with the currently highest priority and compare each subsequent channel with this value. If the new channel has a higher priority, we replace the previously saved channel with the current one and continue the comparison. While this method is clear, it could consume a considerable number of clock cycles.

Alternatively, a hybrid solution combines sequential and combinational approaches. For instance, we could reduce the total number of channels for comparison by choosing the highest priority between each pair of channels in a combinational manner. This approach aims to optimize efficiency by decreasing the number of channels to compare, but it still maintains a linear solution.

For completely combinational logic we can use binary tree form to compare each pair of channels each time cutting in half, but the results of it are not being stored anywhere. For example, in Figure 3.1 the hierarchy of comparators for 8 channels is shown. Each CMP has data lines, taking on input priorities of two channels and propagating the highest of them – the lowest value – to the next stage of comparator blocks. The result of the comparison is propagated to hierarchies of multiplexors from which the index of the channel with the highest priority can

be retrieved. Comparators of the first level – the left one – get input priorities from the register map. Next states get output from the previous stage as an input.

3.3.4 Trigger register

The trigger register is designed to capture incoming interrupts from various hardware components. Given the possibility of multiple channels triggering simultaneously, it's important to note that only one channel can be addressed at a time, posing a risk of losing trigger impulses.

To partially compensate for this we can store several triggers into a buffer. For the initial version of TSDMA, the number of 4 stored triggers was chosen. This storage can be implemented either as a counter or an actual buffer.

Considering details and errors that can appear when dealing with the matter, the preference was given to a buffer option. Represented as a 4-bit shift register it can easily complete the given task of storing impulses.

The software trigger on the other hand is captured in the register map and cannot have multiple entries, thus, the trigger register will only take care of clearing the corresponding register in DMA memory.

3.3.5 Descriptor buffer and address

The descriptor buffer serves as a temporary storage for the currently processed descriptor during data transfers. It is loaded with descriptor after a process of arbitration is finished and the next served channel is known. Configuration register of descriptor does not change the values since they are only giving the information to the controller to determine the next behavior.

Address and data counter registers are changed after each cycle. After the read cycle, the source address is either incremented, wrapped to the starting address of the block, or remains unchanged based on the value in the configuration register. Similarly, after the write cycle destination address is modified. Both values in the data counter register are modified separately. The size of the block is decreased after each read-write cycle, meanwhile repeat counter is decreased each time the block counter reaches a zero until the repeat counter equals zero as well.

The descriptor address is a separate register that is used for loading the descriptor from and to memory. In the case of Scatter-Gather mode first loaded descriptor is a pointer to descriptors, so the descriptor address register is loaded with the source address from the buffered descriptor followed by reading the actual transfer descriptor from the obtained address. After the transfer, both descriptors are modified in memory.

3.4 Controller

Based on previous analysis better option for the controller representation for our design is FSM – as it is more suitable for small designs, takes less area on a chip, and is faster, despite the limitation in modifications. States of FSM are representing two modes of TSDMA operation:

- IDLE – mode when the TSDMA is disabled and performs no external activity.
- RUNNING – operational mode for data transfer and active state.

The controller is responsible for coordinating all datapath modules by using control signals. Furthermore, the controller is designed to be an AHB Master on the bus, handling all communication processes for data transfers and descriptor operations(Figure 3.2). It also watches the number of enabled channels and puts DMAC in IDLE mode where it performs no activity if there is no presence of an active channel.

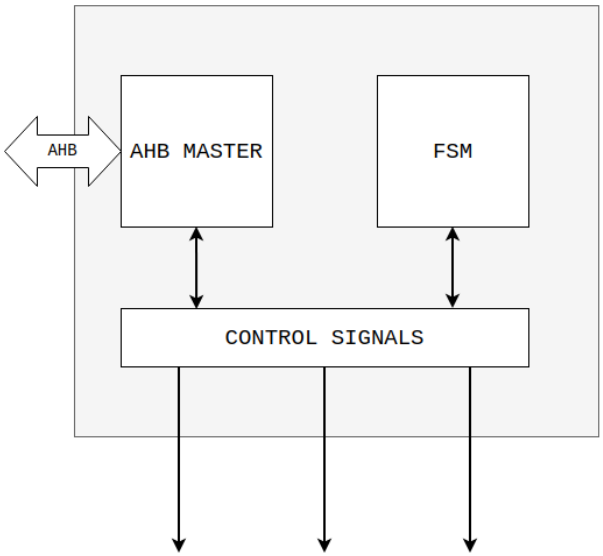

Figure 3.2 Control logic block diagram.

3.5 TSDMA

After all modules of the design were proposed it is suitable to look at the top module combining it all. The generic variables will be defined at the top level as well and all components are instantiated and connected accordingly to signals they require for communication inside the block. It receives input values for both AHB Slave and Master interfaces redirecting them to register map and controller correspondingly. Other input signals are clock, synchronous reset, and hardware interrupts – *hw_int*. Outputs are composed of transfer completion interrupts – *tc_int* and output signals for both bus interfaces.

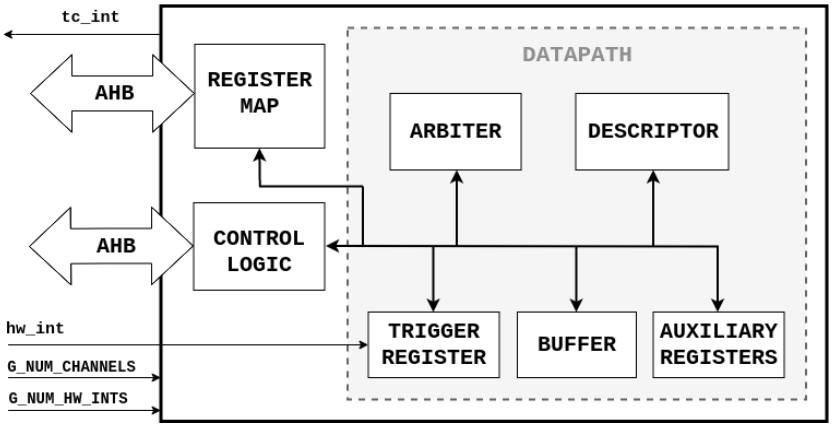

Figure 3.3 TSDMA block diagram

Chapter 4
Implementation

In Chapter 2 we described the design chosen for TSDMA. In this chapter, we discuss the implementation of its submodules and their details.

4.1 Register map

Contains 20 32-bit registers. The three first registers contain information for TSDMA as a block. First register – BLOCK ID contains the 16-bit ID and 4-bit revision code of TSDMA, DMA CONFIG has DMA enable bit for block activation, software reset, and 2-bit mode of arbitration. Next register DESCRIPTOR ADDRESS contains the starting address of channel descriptors stored sequentially from this address. CHANNEL ENABLE has a bit per channel for its activation, where every *x-th* bit of register corresponds to *x-th* channel. Similarly to this register SW TRIGGER, TRANSFER COMPLETION, INTERRUPT ENABLE, and SCATTER GATHER registers are organized. Following registers are made of 6 32-bit registers – CPU PRIORITY A-F and HW TRIGGER A-F. Each of these contain 5-bit vectors for value per channel, according to Appendix A. RTL files are generated from rdl source contain all logic covering AHB Slave activity and mentioned registers. Top file *tsdma_reg_map* is manually written module covering all wires of *dma_pio* into more organized way for following utilization in the top module.

Controller The controller is implemented as a 13-state FSM. The FSM states used for the TSDMA are represented in Figure 4.1. After reset TSDMA is inactive and remains in IDLE state, performing no activity, until it gets activated. After it is enabled it goes to PREPARE state controlling the presence of the active channels. If there is at least one active channel the controller proceeds to ARBITRATION state, otherwise it returns to IDLE mode. In ARBITRATION state it activates the arbiter module and waits for a response signaling the arbitration was done.

It continues with a more or less linear sequence of states performing a transfer. States READ DATA and *WRITE DATA* loop until the trigger transaction is finished. Then it updates the descriptor in memory and proceeds to FINISH state, where the channel is disabled if all transfers for the configuration are completed. Depending on whether it is Scatter-Gather transaction or not it can switch back to the next transfer – when the trigger initiates the transfer of the entire list of descriptors at once – or goes back to PREPARE state, checking the presence of other active channels.

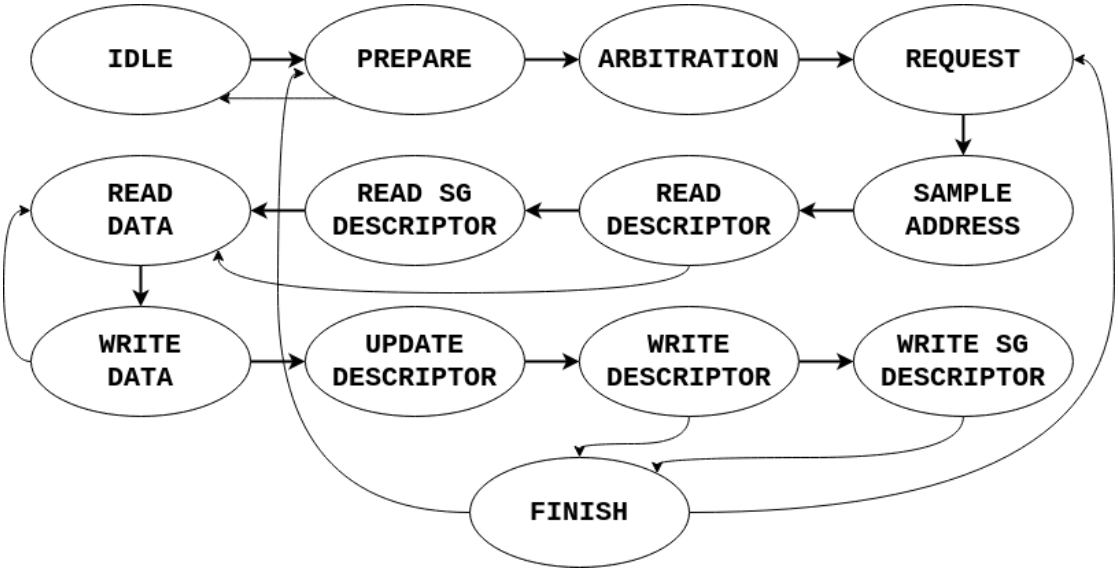

Figure 4.1 TSDMA controller's states

4.1.1 Arbiter

Controls the arbitration of channels depending on priority mode and is enabled by input control signal *arbiter_enable*from the controller. It contains a register for the last chosen channel, which is being held for the entire transfer triggered for this channel until the controller sets of new arbitration cycle. Arbitration is a combination of both sequential and combinational logic.

```
generate
    for(i = 0; i < P_NUM_STAGE; i++) begin: block_stage
        for(j = 0; j < P_IN_STAGE_1/(2**(i+1)); j++) begin: comparator_num
            tsdma_pr_cmp i_cmp_(
                .ch_a_en    (enables    [i][2*j]),
                .ch_a_pr    (priorities [i][2*j]),
                .ch_b_en    (enables    [i][2*j+1]),
                .ch_b_pr    (priorities [i][2*j+1]),
                .hi_en      (enables    [i+1][j]),
                .hi_pr      (priorities [i+1][j]),
                .sel        (sel        [i][0][j])
                );
        end
        for(k = 0; k < P_NUM_STAGE-i; k++) begin: mux_stage
            for(l = 0; l < (P_IN_STAGE_1/(2**(i+k+2))); l++) begin:block_num
                tsdma_mux sel_muxed(
                    .in1    (sel[i][k][l*2]),
                    .in2    (sel[i][k][l*2+1]),
                    .sel    (sel[k+i+1][P_NUM_STAGE-2-i-k][0]),
                    .out    (sel[i][k+1][l])
                );
            end
        end
    end
endgenerate
```

It contains a sub-module *tsdma_cpu_hi_pr* for the CPU custom priority mode, which generates submodules according to the number of channels as shown in the example above. The comparator block *tsdma_pr_cmp*takes two channels' priorities and outputs the higher priority alongside with *sel* signal which serves later for getting the index of the channel with the highest priority. The value is then propagated back to the arbiter module and is used for the result of arbitration.

4.2 Trigger register

Is implemented as a packed 2D array of 4-bit vectors serving as a buffer for triggers where valid values are 0b0001, 0b0011, 0b0111, 0b1111. Each 1 represents a trigger impulse. The implementation respects the possibility of catching invalid values and they will lead back to valid ones. If the channel is disabled both hardware and software triggers are cleared. Below is an example of incoming interrupt impulses. Decreasing value in buffer due to processed trigger is implemented similarly.

```
    case(hw_trg[i])
        0: hw_trg[i]  <= 1;
        1: hw_trg[i]  <= 3;
        2: hw_trg[i]  <= 3;
        3: hw_trg[i]  <= 7;
        4: hw_trg[i]  <= 3;
        5: hw_trg[i]  <= 7;
        6: hw_trg[i]  <= 7;
        7: hw_trg[i]  <= 15;

        8: hw_trg[i]  <= 3;
        9: hw_trg[i]  <= 7;
        10: hw_trg[i] <= 7;
        11: hw_trg[i] <= 15;
        12: hw_trg[i] <= 7;
        13: hw_trg[i] <= 15;
        14: hw_trg[i] <= 15;
        15: hw_trg[i] <= 15;
    endcase
end
```

4.3 Descriptor

Descriptors are implemented as 4 32-bit registers. The source and destination addresses take up the entire registers, data counter is split in half for repeat counter and block size. The configuration register uses only 6 lower bits, the rest of the register is remaining reserved. During TSDMA operation upper 16 bits are used for storing the initial block size which is not supposed to be reached or exposed to a user. The registers are being conducted by control signals from the controller.

Chapter 5
Testing

In Chapter 4 we discussed the implementation of individual components of the design. This chapter will provide information about the TSDMA testing flow. The testing process was broken down into two stages:

- The initial debug. There were primary test benches (TB) written alongside the design phase. They served to test the basic functionality of each separate part of the design. This approach allowed us to detect any errors or mismatches in control signals before integrating the sub-modules of TSDMA. By ensuring that each block worked as expected individually, we minimized the chances of encountering errors later on.
- Proper tests using TBs written in UVM. The Universal Verification Methodology is the verification standard for the verification of the digital system, more information could be found at [12]. These TBs were provided by Marek Santa from Tropic Square. These tests were aimed at covering the widest possible range of configurations and cases that can be encountered in the utilization of the block. With Marek's help and provided testing environment we debugged the TSDMA controller and fixed encountered bugs that will be described in Section 5.2.

This process is standard and crucial practice in design verification, as it is not safe to have the same person responsible for both testing and the design itself. This could potentially lead to errors being overlooked, which could have negative consequences in the future process of integrating DMAC into a bigger microchip.

5.1 Basic functionality tests

Basic tests were not written in UVM, as they primarily focused only on verifying the fundamental capabilities of individual components. Initially, the controller and AHB master components were written and tested separately, but they were eventually merged into one comprehensive module. To effectively test this united module, an additional module was created to simulate a slave on the bus, complete with its own memory. This allowed for the loading of descriptor values into the memory, which were then retrieved and executed by the controller. The outcomes of the TSDMA operation were then stored in the same slave memory. Ultimately, in order to ensure that everything was functioning as expected, the primary focus during testing was on monitoring the memory registers.

By subjecting the arbiter and descriptor register to similar tests, we were able to effectively identify and correct any significant malfunctions before integrating them with the controller and other components.

5.2 UVM tests

5.2.1 Structure of tests and TB

UVM tests used for TSDMA have the following structure. At the beginning of the test random yet valid descriptors are being generated and located at random starting address sequentially. Next, TSDMA is configured and enabled. Configuration is mostly randomly generated, but the DESCRIPTOR ADDRESS register has to correspond to a previously generated value. After the TSDMA set active, random SW and HW interrupts are being generated. Test runs until all enabled channels have the TRANSFER COMPLETION flag. For the test to have relevant runtime, descriptors are generated with certain limitations.

The TB itself is organized as shown in Figure 5.1. Any transfer that appears on the interface is reported to the predictor – a model of TSDMA. It generates outputs based on received inputs that are sent to Scoreboards – simple comparators. Transfers on TSDMA interfaces are reported to Scoreboards as observed and later are compared to expected outputs. The number of Scoreboards in TB corresponds with the number of monitored interfaces.

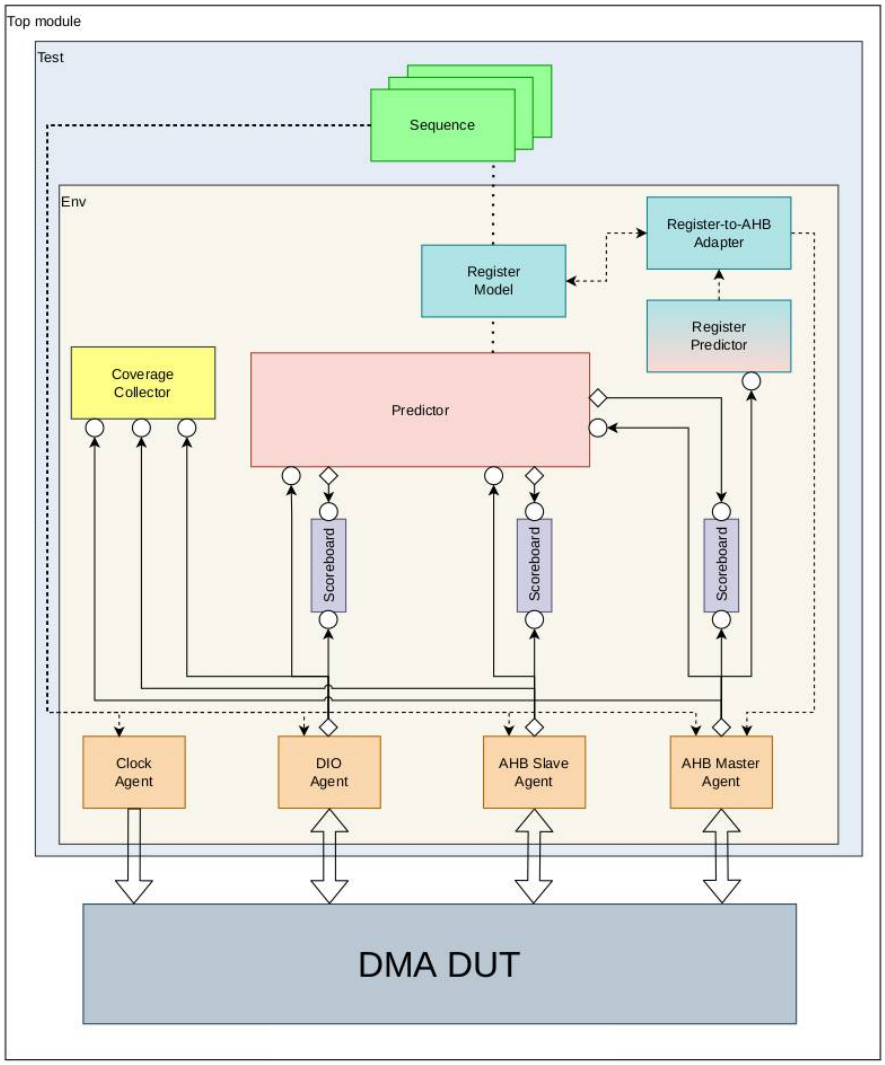

Figure 5.1 TB block diagram.

5.2.2 Observed bugs

The first thing revealed during UVM tests was a mismatch in the bus interface – it was expected to restrict several types of transfer (e. g. burst or undefined length), and signal *ahb_hsel* was supposed to be generated by bus master contrasted to arbiter in interface standard. After reduction to AHB-Lite 3 interface according to other components in the system, UVM testing could be commenced.

The next thing to fix was the method of capturing incoming triggers to a buffer, which resulted in capturing multiple entries of the trigger on one impulse that lasted for more than one clock cycle, shown in Figure 5.2. The solution to this is simple. It was enough to add an edge detector to capture a trigger on the rising edge only (Figure 5.3).

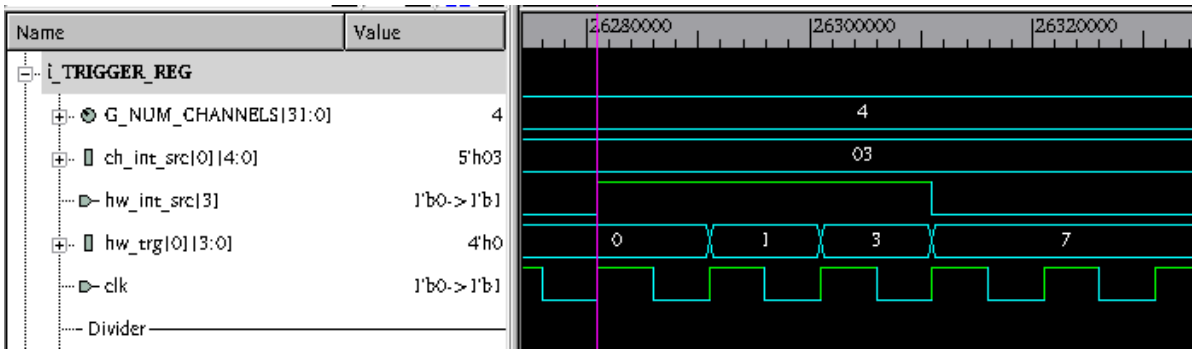

Figure 5.2 *hw_trg* capturing multiple interrupts on one impulse.

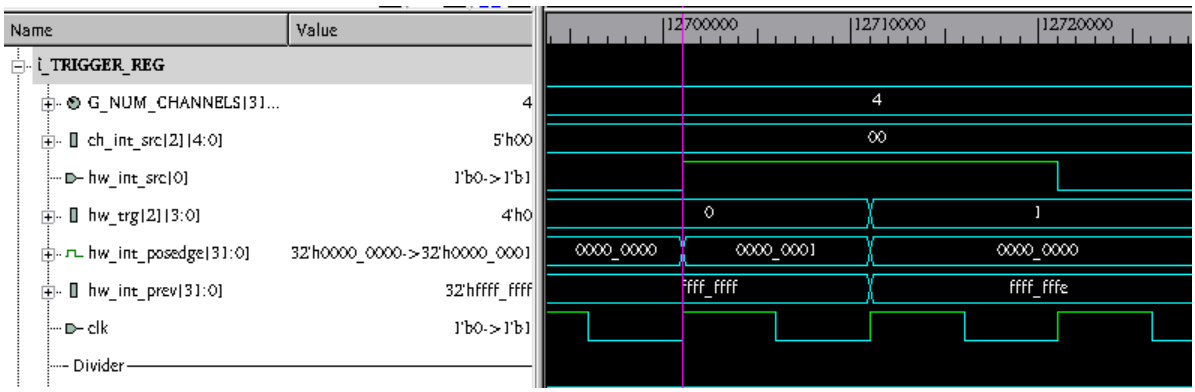

Figure 5.3 *hw_trg* capturing only positive edge of interrupt.

The next revealed error was the dropping of *ahb_hsel* signal with specific descriptor configuration between two blocks in repeat mode. It caused the invisibility of one read cycle which resulted in a distorted sequence of transfers on the bus and could lead to loose of bus ownership. In that case, transfer would be postponed until the bus is released and granted back to TSDMA

In addition, clearance of the software trigger was poorly written resulting in setting *sw_trg_clr* signal to a logical one forever. This restricted the write of multiple software triggers at once, since they were cleared right after.

Now one found bug still remains unsolved and will be fixed in future work. The problem is again with the software clear and the SW TRIGGER register has different values after identical configuration depending on whether it was configured involving a prior reset of the TSDMA or not.

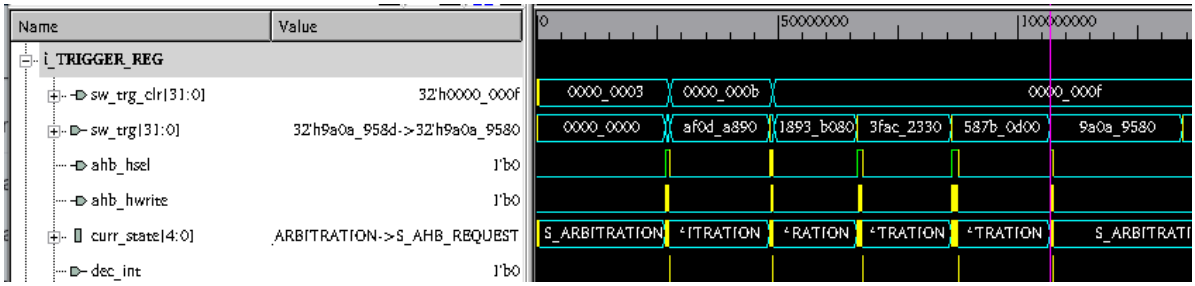

Figure 5.4 *sw_trg_clr* sets at 1 lower 4 bits and sticks to value.

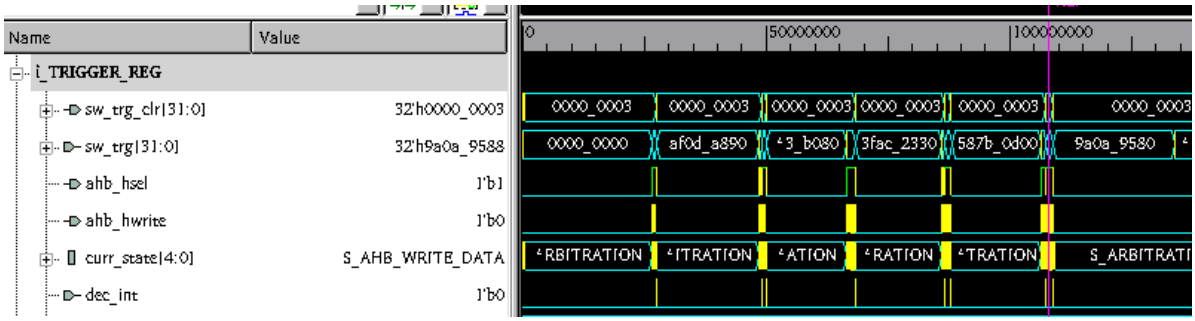

Figure 5.5 *sw_trg* has correct values.

Conclusion

In this thesis, we completed the main goal – designed TSDMA, a versatile DMA controller for future application in an ASIC-targeted microcontroller by Tropic Square s. r. o. It has up to 32 configurable channels and trigger sources which provide the flexibility to arrange them as desired using generic variables. Due to the flexibility of channels' descriptors and the DMA controller itself, it can be used for different projects.

The set of features was based on researched microcontrollers and their range of configuration while taking into consideration advice and recommendations received during consultations with Ing. Ondrej Ille – the supervisor of the thesis and company's design team lead. Resulting in versatile DMA, having separate descriptors for each channel with a possibility to extend it up to $2^{16} - 1$ descriptors for each channel, using Scatter-Gather.

Controlled by FSM, it has a register map acting as a slave on a bus with an AHB-Lite interface for accessing configuration registers of TSDMA. The controller acts at the same time as a bus master with equal interface handling data transfers initiated by one of the trigger sources. When triggers arrive while the controller is occupied with a previous transfer, they are kept in an internal buffer for later processing.

TSDMA was tested with RTL simulations using UVM test benches and works as expected, transferring data in the system with high speed thanks to a small number of auxiliary operations in the context of transfers and not wasting time. During the testing phase, TSDMA was put through RTL simulations with the help of UVM test-benches. These simulations proved that TSDMA functions accurately as intended in high-speed system data transfers. This success can be attributed to the minimal use of auxiliary time for side processes during data transfers, ensuring efficient and time-saving operations.

Due to the minimalistic yet wide functionality in the set of features, we made sure that the design would not take up much chip surface in ASIC implementation which was one of the thesis' goals.

5.3 Future work

TSDMA designed within this thesis undoubtedly brings valuable contribution to the field of DMA controllers. Several factors were not completed in the best possible way which gives room for future improvements of TSDMA. It combines high flexibility of channel and block configuration making it a multipurpose versatile IC. Besides its primary goal to be integrated into the next microcontroller to be designed by Tropic Square, it also has the potential to enhance other projects. After it eventually becomes an open-source design it can be picked up by others and improved in multiple directions.

Appendix A

Register map

Address Offset	Register Name	Reset Value
0x0	BLOCK_ID	—
0x4	DMA_CONFIG	0x00000000
0x8	DESCRIPTOR_ADDRESS	—
0xc	CHANNEL_ENABLE	0x00000000
0x10	SW_TRIGGER	0x00000000
0x14	HW_TRIGGER_A	0x00000000
0x18	HW_TRIGGER_B	0x00000000
0x1c	HW_TRIGGER_C	0x00000000
0x20	HW_TRIGGER_D	0x00000000
0x24	HW_TRIGGER_E	0x00000000
0x28	HW_TRIGGER_F	0x00000000
0x2c	CPU_PRIORITY_A	0x00000000
0x30	CPU_PRIORITY_B	0x00000000
0x34	CPU_PRIORITY_C	0x00000000
0x38	CPU_PRIORITY_D	0x00000000
0x3c	CPU_PRIORITY_E	0x00000000
0x40	CPU_PRIORITY_F	0x00000000
0x44	TRANSFER_COMPLETION	0x00000000
0x48	INTERRUPT_ENABLE	0x00000000
0x4c	SCATTER_GATHER	0x00000000

Register name:		BLOCK_ID		
Address:		0x0		
Field	**Type**	**Reset value**	**Bits**	**Description**
ID_CODE	RW		15:0	Identification code
REV_CODE	RW	0x0	19:16	Revision code

Register name:		DMA_CONFIG		
Address:		0x4		
Field	**Type**	**Reset value**	**Bits**	**Description**
DMA_EN	RW	0x0	0:0	Enable DMA
MODE	RW	0x0	2:1	Channel priority algorithm MODE_ROUND_ROBIN : 0x0 : ROUND ROBIN MODE MODE_FIXED : 0x1 : LOWEST INDEX FIRST MODE MODE_CPU_DEFINED : 0x2 : CPU DEFINED MODE
RESET	RW	0x0	3:3	0x0

Register name:		DESCRIPTOR_ADDRESS		
Address:		0x8		
Field	**Type**	**Reset value**	**Bits**	**Description**
ADR	RW		31:0	Starting address of descriptors block.

Register name:	CHANNEL_ENABLE
Address:	0xc

Field	**Type**	**Reset value**	**Bits**	**Description**
CH_EN0	RW	0x0	0:0	0th channel enable bit.
CH_EN1	RW	0x0	1:1	1st channel enable bit.
CH_EN2	RW	0x0	2:2	2nd channel enable bit.
CH_EN3	RW	0x0	3:3	3rd channel enable bit.
CH_EN4	RW	0x0	4:4	4th channel enable bit.
CH_EN5	RW	0x0	5:5	5th channel enable bit.
CH_EN6	RW	0x0	6:6	6th channel enable bit.
CH_EN7	RW	0x0	7:7	7th channel enable bit.
CH_EN8	RW	0x0	8:8	8th channel enable bit.
CH_EN9	RW	0x0	9:9	9th channel enable bit.
CH_EN10	RW	0x0	10:10	10th channel enable bit.
CH_EN11	RW	0x0	11:11	11th channel enable bit.
CH_EN12	RW	0x0	12:12	12th channel enable bit.
CH_EN13	RW	0x0	13:13	13th channel enable bit.
CH_EN14	RW	0x0	14:14	14th channel enable bit.
CH_EN15	RW	0x0	15:15	15th channel enable bit.
CH_EN16	RW	0x0	16:16	16th channel enable bit.
CH_EN17	RW	0x0	17:17	17th channel enable bit.
CH_EN18	RW	0x0	18:18	18th channel enable bit.
CH_EN19	RW	0x0	19:19	19th channel enable bit.
CH_EN20	RW	0x0	20:20	20th channel enable bit.
CH_EN21	RW	0x0	21:21	21st channel enable bit.
CH_EN22	RW	0x0	22:22	22nd channel enable bit.
CH_EN23	RW	0x0	23:23	23rd channel enable bit.
CH_EN24	RW	0x0	24:24	24th channel enable bit.

CH_EN25	RW	0x0	25:25	25th channel enable bit.
CH_EN26	RW	0x0	26:26	26th channel enable bit.
CH_EN27	RW	0x0	27:27	27th channel enable bit.
CH_EN28	RW	0x0	28:28	28th channel enable bit.
CH_EN29	RW	0x0	29:29	29th channel enable bit.
CH_EN30	RW	0x0	30:30	30th channel enable bit.
CH_EN31	RW	0x0	31:31	31st channel enable bit.

Register name:		SW_TRIGGER		
Address:		0x10		
Field	**Type**	**Reset value**	**Bits**	**Description**
SW_TRG0	RW	0x0	0:0	0th channel SW trigger bit.
SW_TRG1	RW	0x0	1:1	1st channel SW trigger bit.
SW_TRG2	RW	0x0	2:2	2nd channel SW trigger bit.
SW_TRG3	RW	0x0	3:3	3rd channel SW trigger bit.
SW_TRG4	RW	0x0	4:4	4th channel SW trigger bit.
SW_TRG5	RW	0x0	5:5	5th channel SW trigger bit.
SW_TRG6	RW	0x0	6:6	6th channel SW trigger bit.
SW_TRG7	RW	0x0	7:7	7th channel SW trigger bit.
SW_TRG8	RW	0x0	8:8	8th channel SW trigger bit.
SW_TRG9	RW	0x0	9:9	9th channel SW trigger bit.
SW_TRG10	RW	0x0	10:10	10th channel SW trigger bit.
SW_TRG11	RW	0x0	11:11	11th channel SW trigger bit.
SW_TRG12	RW	0x0	12:12	12th channel SW trigger bit.
SW_TRG13	RW	0x0	13:13	13th channel SW trigger bit.
SW_TRG14	RW	0x0	14:14	14th channel SW trigger bit.

SW_TRG15	RW	0x0	15:15	15th channel SW trigger bit.
SW_TRG16	RW	0x0	16:16	16th channel SW trigger bit.
SW_TRG17	RW	0x0	17:17	17th channel SW trigger bit.
SW_TRG18	RW	0x0	18:18	18th channel SW trigger bit.
SW_TRG19	RW	0x0	19:19	19th channel SW trigger bit.
SW_TRG20	RW	0x0	20:20	20th channel SW trigger bit.
SW_TRG21	RW	0x0	21:21	21th channel SW trigger bit.
SW_TRG22	RW	0x0	22:22	22th channel SW trigger bit.
SW_TRG23	RW	0x0	23:23	23th channel SW trigger bit.
SW_TRG24	RW	0x0	24:24	24th channel SW trigger bit.
SW_TRG25	RW	0x0	25:25	25th channel SW trigger bit.
SW_TRG26	RW	0x0	26:26	26th channel SW trigger bit.
SW_TRG27	RW	0x0	27:27	27th channel SW trigger bit.
SW_TRG28	RW	0x0	28:28	28th channel SW trigger bit.
SW_TRG29	RW	0x0	29:29	29th channel SW trigger bit.
SW_TRG30	RW	0x0	30:30	30th channel SW trigger bit.
SW_TRG31	RW	0x0	31:31	31th channel SW trigger bit.

Register name:		HW_TRIGGER_A		
Address:		0x14		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH0	RW	0x0	4:0	HW trigger source for channel 0.
CH1	RW	0x0	9:5	HW trigger source for channel 1.
CH2	RW	0x0	14:10	HW trigger source for channel 2.
CH3	RW	0x0	19:15	HW trigger source for channel 3.
CH4	RW	0x0	24:20	HW trigger source for channel 4.

CH5	RW	0x0	29:25	HW trigger source for channel 5.

Register name:		HW_TRIGGER_B		
Address:		0x18		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH6	RW	0x0	4:0	HW trigger source for channel 6.
CH7	RW	0x0	9:5	HW trigger source for channel 7.
CH8	RW	0x0	14:10	HW trigger source for channel 8.
CH9	RW	0x0	19:15	HW trigger source for channel 9.
CH10	RW	0x0	24:20	HW trigger source for channel 10.
CH11	RW	0x0	29:25	HW trigger source for channel 11.

Register name:		HW_TRIGGER_C		
Address:		0x1c		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH12	RW	0x0	4:0	HW trigger source for channel 12.
CH13	RW	0x0	9:5	HW trigger source for channel 13.
CH14	RW	0x0	14:10	HW trigger source for channel 14.
CH15	RW	0x0	19:15	HW trigger source for channel 15.
CH16	RW	0x0	24:20	HW trigger source for channel 16.
CH17	RW	0x0	29:25	HW trigger source for channel 17.

Register name:	HW_TRIGGER_D
Address:	0x20

Field	**Type**	**Reset value**	**Bits**	**Description**
CH18	RW	0x0	4:0	HW trigger source for channel 18.
CH19	RW	0x0	9:5	HW trigger source for channel 19.
CH20	RW	0x0	14:10	HW trigger source for channel 20.
CH21	RW	0x0	19:15	HW trigger source for channel 21.
CH22	RW	0x0	24:20	HW trigger source for channel 22.
CH23	RW	0x0	29:25	HW trigger source for channel 23.

Register name:		HW_TRIGGER_E		
Address:		0x24		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH24	RW	0x0	4:0	HW trigger source for channel 24.
CH25	RW	0x0	9:5	HW trigger source for channel 25.
CH26	RW	0x0	14:10	HW trigger source for channel 26.
CH27	RW	0x0	19:15	HW trigger source for channel 27.
CH28	RW	0x0	24:20	HW trigger source for channel 28.
CH29	RW	0x0	29:25	HW trigger source for channel 29.

Register name:		HW_TRIGGER_F		
Address:		0x28		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH30	RW	0x0	4:0	HW trigger source for channel 30.
CH31	RW	0x0	9:5	HW trigger source for channel 31.

Register name:		CPU_PRIORITY_A		
Address:		0x2c		
Field	Type	Reset value	Bits	Description
CH0	RW	0x0	4:0	CPU defined priority for channel 0.
CH1	RW	0x0	9:5	CPU defined priority for channel 1.
CH2	RW	0x0	14:10	CPU defined priority for channel 2.
CH3	RW	0x0	19:15	CPU defined priority for channel 3.
CH4	RW	0x0	24:20	CPU defined priority for channel 4.
CH5	RW	0x0	29:25	CPU defined priority for channel 5.

Register name:		CPU_PRIORITY_B		
Address:		0x30		
Field	Type	Reset value	Bits	Description
CH6	RW	0x0	4:0	CPU defined priority for channel 6.
CH7	RW	0x0	9:5	CPU defined priority for channel 7.
CH8	RW	0x0	14:10	CPU defined priority for channel 8.
CH9	RW	0x0	19:15	CPU defined priority for channel 9.
CH10	RW	0x0	24:20	CPU defined priority for channel 10.
CH11	RW	0x0	29:25	CPU defined priority for channel 11.

Register name:		CPU_PRIORITY_C		
Address:		0x34		
Field	Type	Reset value	Bits	Description
CH12	RW	0x0	4:0	CPU defined priority for channel 12.
CH13	RW	0x0	9:5	CPU defined priority for channel 13.
CH14	RW	0x0	14:10	CPU defined priority for channel 14.
CH15	RW	0x0	19:15	CPU defined priority for channel 15.
CH16	RW	0x0	24:20	CPU defined priority for channel 16.
CH17	RW	0x0	29:25	CPU defined priority for channel 17.

Register name:		CPU_PRIORITY_D		
Address:		0x38		
Field	Type	Reset value	Bits	Description
CH18	RW	0x0	4:0	CPU defined priority for channel 18.
CH19	RW	0x0	9:5	CPU defined priority for channel 19.
CH20	RW	0x0	14:10	CPU defined priority for channel 20.
CH21	RW	0x0	19:15	CPU defined priority for channel 21.
CH22	RW	0x0	24:20	CPU defined priority for channel 22.
CH23	RW	0x0	29:25	CPU defined priority for channel 23.

Register name:		CPU_PRIORITY_E		
Address:		0x3c		
Field	Type	Reset value	Bits	Description
CH24	RW	0x0	4:0	CPU defined priority for channel 24.
CH25	RW	0x0	9:5	CPU defined priority for channel 25.
CH26	RW	0x0	14:10	CPU defined priority for channel 26.
CH27	RW	0x0	19:15	CPU defined priority for channel 27.

CH28	RW	0x0	24:20	CPU defined priority for channel 28.
CH29	RW	0x0	29:25	CPU defined priority for channel 29.

Register name:		CPU_PRIORITY_F		
Address:		0x40		
Field	**Type**	**Reset value**	**Bits**	**Description**
CH30	RW	0x0	4:0	CPU defined priority for channel 30.
CH31	RW	0x0	9:5	CPU defined priority for channel 31.

Register name:		TRANSFER_COMPLETION		
Address:		0x44		
Field	**Type**	**Reset value**	**Bits**	**Description**
TC_INT	RW W1C	0x0	31:0	i-th bit corresponds to i-th channel transfer completion flag.

Register name:		INTERRUPT_ENABLE		
Address:		0x48		
Field	**Type**	**Reset value**	**Bits**	**Description**
INT_EN	RW	0x0	31:0	i-th bit corresponds to i-th channel interrupt enable.

Register name:	SCATTER_GATHER
Address:	0x4c

Field	**Type**	**Reset value**	**Bits**	**Description**
SG	RW	0x0	31:0	i-th bit corresponds to i-th channel scatter_gather enable bit.

Appendix B

Descriptor map

Address Offset	Register Name	Reset Value
0x0	CHx_CONFIG	0x00000000
0x4	CHx_SRC_ADR	0x00000000
0x8	CHx_DST_ADR	0x00000000
0xc	CHx_DATA_CNT	0x00000000

Register name:		CHx_CONFIG		
Offset:		0x0		
Field	**Type**	**Reset value**	**Bits**	**Description**
TRG_ACT	RW	0x0	1:0	Type of transfer triggered by SW or HW trigger. 0x0 : TRANSACTION TRANSFER 0x1 : BLOCK TRANSFER
SRC_ADR_MODE	RW	0x0	3:2	Type of modification of source address after each data transfer. 0x0 : FIXED MODE 0x1 : INCREMENTING MODE 0x2 : 16 BYTES WRAPPED MODE 0x3 : 32 BYTES WRAPPED MODE
DST_ADR_MODE	RW	0x0	5:4	Type of modification of destination address after each data transfer. 0x0 : FIXED MODE 0x1 : INCREMENTING MODE 0x2 : 16 BYTES WRAPPED MODE 0x3 : 32 BYTES WRAPPED MODE
SG_TRG_ACT	RW	0x0	6:6	Type of transfer triggered by SW or HW trigger in scatter-gather list. 0x0 : ENTIRE LIST TRANSFER 0x1 : SINGLE DESCRIPTOR TRANSFER
RESERVED		0x0	31:16	Reserved for inner use, can aquire different values.

Register name:	CHx_SRC_ADR
Offset:	0x4

Field	**Type**	**Reset value**	**Bits**	**Description**
SRC_ADR	RW	0x0	31:0	Source address.

Register name:		CHx_DST_ADR		
Offset:		0x8		
Field	**Type**	**Reset value**	**Bits**	**Description**
DST_ADR	RW	0x0	31:0	Destination address.

Register name:		CHx_DATA_CNT		
Offset:		0xc		
Field	**Type**	**Reset value**	**Bits**	**Description**
BLOCK_SIZE	RW	0x0	15:0	If SCATTER_GATHER = 1 - number of desciptors in scatter-gather source list, otherwise number of words from 1 to 65535.
REPEAT_CNT	RW	0x1	31:16	Number of times block of data will be transfered. 1 - single shot, 2-65535 - number of block transfers and 0 - 65536 times.

Appendix C

Interface

Figure C.1 Ports

Name	Direction	Width	Description
DMA channels signals			
hw_int	In	*G_NUM_HW_INTS*	HW interupts
tc_int	Out	*G_NUM_CHANNELS*	Transfer completion interrupt
AHB Master Signals			
ahb_m_hready	In	1	AHB Ready indication from slaves
ahb_m_hrdata	In	32	AHB Read data
ahb_m_haddr	Out	32	AHB Address
ahb_m_hwrite	Out	1	AHB Write enable
ahb_m_hsize	OUT	3	AHB Transfer size
ahb_m_hwdata	Out	32	AHB Write data
ahb_m_hburst	Out	3	AHB Burst type
ahb_m_htrans	Out	2	AHB Transaction type
AHB Slave Signals			
ahb_s_haddr	In	5	AHB Address
ahb_s_hwrite	In	1	AHB Write enable
ahb_s_hwdata	In	32	AHB Write data
ahb_s_hready_in	In	1	AHB Ready indication from slaves
ahb_s_hsize	In	3	AHB Transfer size
ahb_s_hsel	In	1	AHB Slave select
ahb_s_htrans	In	2	AHB Transaction type
ahb_s_hready	Out	1	AHB Ready indication to slaves
ahb_s_hrdata	Out	32	AHB Read data

Figure C.2 Generics

Name	Type	Description
G_NUM_CHANNELS	natural	Number of channels
G_NUM_HW_INTS	natural	Number of HW sources of interrupt

Bibliography

1. NULL, Linda; LOBUR, Julia. *The essentials of computer organization and architecture*. 2nd ed. Sudbury, MA: Jones and Bartlett, 2006.
2. CORPORATION, Atmel. *Atmel AVR1304: Using the XMEGA DMA Controller* [http://ww1.microchip.com/downloads/en/appnotes/atmel-8046-using-the-xmega-dma-controller_application-note_avr1304.pdf]. 2013. [Accessed 3-10-2023].
3. MOYER, Bryon. *How Does Scatter/Gather Work?* [https://www.eejournal.com/article/20170209-scatter-gather/]. 2017. [Accessed 02-11-2023].
4. *Modes of DMA Transfer* [https://www.geeksforgeeks.org/modes-of-dma-transfer/]. 2023. [Accessed 27-11-2023].
5. TANENBAUM, Andrew S; AUSTIN, Todd. *Structured Computer Organization*. 6th ed. Upper Saddle River, NJ: Pearson, 2012.
6. WANG, Jiacun. *Formal methods in computer science*. Philadelphia, PA: Chapman & Hall/CRC, 2019. Textbooks in Mathematics.
7. LAN, Nguyen Thi Hoang. *Computer Architecture* [https://archive.org/details/cnx-org-col10761/page/n125/mode/2up]. 2009. [Accessed 04-01-2024].
8. *STM32 32-bit Arm Cortex MCUs* [https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html]. 2023. [Accessed 31-12-2023].
9. CONTI, Francesco. *Revisions to VHDL vs. Verilog* [https://electronics.stackexchange.com/posts/163110/revisions]. 2015. [Accessed 01-01-2024].
10. JENSEN, Jonas Julian. *Delta cycles explained* [https://vhdlwhiz.com/delta-cycles-explained/]. 2018. [Accessed 29-9-2023].
11. *IEEE standard for SystemVerilog–unified hardware design, specification, and verification language*. Piscataway, NJ, USA, 2018. Tech. rep. IEEE.
12. HARSHITHA, N B; PRAVEEN KUMAR, Y G; KURIAN, M Z. An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC's): A Review. In: *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*. 2021, pp. 1710–1713. Available from DOI: 10.1109/ICAIS50930.2021.9396034.

Concents of the attachment

- `readme.txt` ..... a brief description of the content of the medium
- `licence.txt` ..... Apache2 licence
- `src`
  - `rtl` ..... source code
    - `controller` ..... RTL file with controller implementation
    - `datapath` ..... RTL files with datapath implementation
    - `outdated` ..... outdates RTL files used for tests
    - `regmap` ..... RTL files with regmap implementation and source .rdl file
  - `thesis` ..... thesis source code in LaTeX
  - `simulation` ..... demonstration of results in VCD
- `text`
  - `thesis.pdf` ..... text of the thesis in PDF