An SoC Infrastructure for Embedded RISC-V Cores

Callum Thomas McWilliam

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

The aim of this project is to improve the viability and performance of an existing implementation of an RV32IM core by designing, implementing, and evaluating a System-on-Chip (SoC) infrastructure. The project focuses on the memory subsystem of an SoC, and in particular an L1 data cache, as it allows the core to make more efficient use of memory accesses which, in turn, makes it more usable for a wider array of applications. The design involves a 1kB, direct-mapped data cache with a 32B block size and a write-back policy. Its implementation allows the core access to larger amounts of memory, while preserving its original speed, with negligible increases in resource and power usage. There are also other design considerations for cache performance factors, and the addition of other types of cache. The project has much room left for future work, such as making the design more configurable, and evaluating it with industry standard benchmarks such as CoreMark.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Callum Thomas McWilliam*)

Acknowledgements

I would like to express my thanks to my supervisor, Nigel Topham. This project wouldn't have been possible without his continued support, guidance, and expertise.

To my wonderful parents, Thomas McWilliam and Wendy Scaife, thank you for your continuous love and support throughout the last four years.

Lastly, I want to thank my incredible friends. These last four years wouldn't have been full of so much joy without each and every one of you.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The instruction set architecture (ISA) is a key part of a computer. The ISA used by a core defines the set of instructions that a processor can execute, along with the format and encoding of those instructions. Many early ISAs used a complex instruction set computer (CISC) architecture, where each instruction is packed with functionality and can carry out multiple lower level instructions. However, several studies found that 95% of the execution time was taken up by around 25% of the instructions used in a CISC architecture [15]. On top of this, many ISAs are proprietary meaning that a license is required which makes it nearly impossible for many parties to gain access [4]. Hence, there was a need for an open-source ISA with a reduced number of simpler instructions. Enter RISC-V (pronounced ”risk-five”), an ISA from the reduced instruction set computer (RISC) family of ISAs.

These ISAs use a reduced number of simpler instructions, using the idea that they can be used in sequence to achieve the function of a single CISC instruction. These instructions are easier to decode and most are executed in a single clock cycle. Furthermore, their simplicity makes them easier to use in a technique called pipelining; where multiple instructions are executed in one cycle since they use different resources.

RISC-V is the most recent of these ISAs, and one of its most important qualities is that it's open-source which allows a larger number of developers and researchers to contribute to the improvement of the ISA. This leads to strong competition and, in turn, a thriving eco-system for the ISA which will be supported for years to come.

While it was designed to be usable for a wide array of practical applications, RISC-V still has a level of simplicity which makes it the go-to ISA for academic purposes, such as student exercises. For students, this allows easier access to a field that is typically difficult to break into. RISC-V has many other qualities, some of which will be touched upon in the Background 2 section of this dissertation.

Cores which use RISC-V ISA (RISC-V cores) are extremely fast, but without access to external memory they lack viability for larger applications. You may be wondering what the trouble is then, why not just give the core access to external memory? The

issue is that accesses to external memory are extremely slow when compared to the cores speed, and so carrying out a memory access slows it down immensely. This is where a System-on-Chip (SoC) infrastructure comes in, as one of the additions it provides is a memory subsystem and, in particular, a cache component. A cache acts as an intermediate memory between the core and external memory, storing data and/or instructions that have been recently used or are likely to be used next. This memory is still slower to access than the cores own memory, but is much faster than accessing external memory. More details of cache types, including hierarchies and replacement policies etc., will be given in the background 2.

Thus, the importance of the addition of an SoC infrastructure is clear, as it simply amplifies many of the already existing qualities of a RISC-V core. The core can be used for a wider array of larger applications while being nearly as fast as it was before, more contributors will be attracted due to this increased viability which will further extend the ISAs lifetime, and students are able to learn about more complex designs which are still fairly straightforward.

1.2 Aims and Contributions

In this project, I work with a version of a RISC-V core called the RV32IM core, the name signifies that it uses a 32-bit address space (32) with the base integer instruction set (I) and the standard extension for integer multiplication and division (M). This core currently only has ICCM and DCCM memories, which are 8kB and 4kB respectively. The core requires a larger memory hierarchy to be more viable.

The key objectives of this project are:

1. To design, implement and evaluate an SoC infrastructure for the RV32IM core.
2. Discuss further improvements to the core, as well as provide some possible designs for these improvements.

The following has been done to achieve these objectives:

1. A data cache which interfaces with external memory has been designed, implemented and evaluated.
2. Further discussion on how the design would vary, due to changes in cache performance factors and the addition of more cache memories, has been discussed.

1.3 Overview

Chapter 1 introduces the motivation behind the project, the main aims, and the contributions made.

Chapter 2 gives background information on key concepts and literature required to understand the project.

Chapter 3 describes the design of the data cache, with details of the cache itself and how it fits into the existing core.

Chapter 4 describes the methodology behind designing, implementing and then evaluating the data cache, so that the results can be reproduced.

Chapter 5 describes the implementation of the data cache in Verilog.

Chapter 6 is an evaluation of the resource utilisation, timing and power usage of the design as well as key cycle timings and a benchmark evaluation.

Chapter 7 summarises the results, describes limitations and future work as well as details on the personal challenges encountered and lessons learned from them.

Chapter 2

Background Research

2.1 Literature Review

2.1.1 RISC Architectures and RISC-V

RISC architectures initially seemed to emerge in the 1960s, but the term itself wasn't coined until the early to mid 1980s when it emerged as a competitor to CISC architectures and started the RISC vs CISC debate [1].

CISC was designed to minimise the number of instructions required to execute a given task, this leads to large instruction sets where each instruction can perform a complex operation. Due to the difference in CPU and memory speeds, more instructions were added to CISC architectures as the years went by to improve performance and reduce the number of accesses to main memory [1, 21]. This leads to instructions that are variable in length (making them difficult to pipeline) and require multiple clock cycles as they are packed with as much functionality as possible. It was found that, in many CISC architectures, a small subset of instructions account for the majority of the CPU time, meaning that a lot of the chip area used by the ISA is redundant [15].

On the other hand, RISC aims to have a reduced and simplified instruction set, with the idea being that a sequence of RISC instructions can achieve the same operation of a single CISC instruction. The simplified instruction set has a number of benefits [21]:

- reduced hardware complexity
- more chip area available for other additions such as on-chip caches
- instructions can be pipelined due to their simplicity and fixed format, leading to faster execution
- quicker to design and implement

CISC places more of an emphasis on hardware, while RISC places more of an emphasis on software. This makes RISC perfect for an SoC architecture used in a device such as a smart phone where the hardware is more compact and not as powerful compared to a desktop, for example.

There was still one problem in the ISA market however, they're all proprietary. This is where RISC-V comes in, an open-source ISA that was conceived to oppose proprietary ISAs and their limitations, such as lack of innovation, huge costs and the loss of an ISA if the company supporting it goes bankrupt [3, 4]. The open-source nature of RISC-V allows a large number of researchers and developers to contribute to the improvement of the ISA, leading to a competitive and thriving environment for it to be supported in for years to come.

On top of this, RISC-V was designed to be modular and extensible, meaning designers can pick and choose what is required for their specific application e.g. if the application does not require integer multiplication or division, they can use an RV32I core instead of RV32IM core. The base version is simple enough to be used for educational and research purposes, while the extensions can be used to shape it for specific purposes like high-performance computing [28, 29].

2.1.2 SoC Infrastructure

2.1.2.1 Overview

To give a brief summary, an SoC infrastructure is defined as an integrated circuit (IC) that integrates most or all of the components of a computer system into a single chip. Most often, this includes a CPU, memory subsystems, GPU and I/O etc., with some other, less common, components such as FPGA and sensors [7, 30]. The components used in an SoC can be customised for it's specific purpose, e.g. a smartphone will usually have sensors such as a gyroscope and accelerometer as well as image processing components. SoCs are important as they provide numerous benefits including reduced power consumption, form factor and cost while providing increased performance and flexibility [18]. As such, they continue to drive technological progress across the tech industry. The component of an SoC that I focus on in this dissertation is the memory subsystem.

2.1.2.2 Memory Wall

One of the key challenges in the design of computer systems, is the memory wall. The time to access main memory is many magnitudes higher than that of a CPUs on-chip memory. The memory wall is the gap between CPU and memory performance, both are increasing but CPU performance is increasing at a much greater rate. Many early papers on this topic [6, 17] note that while the gap was an issue for computer performance back then, it would become a much bigger issue in the future, due to the gap increasing at a rate of roughly 50% every year [20].

This leads to memory speeds acting as a bottleneck for overall system performance, making it difficult to take full advantage of the 60% increase of CPU performance that is seen year on year. The wall will be hit when an increase in CPU speed causes little no overall performance improvement, and the speed of the computer is purely determined by memory access speeds.

In fact, [31] gave, to put it lightly, a fairly damning assessment of the memory wall

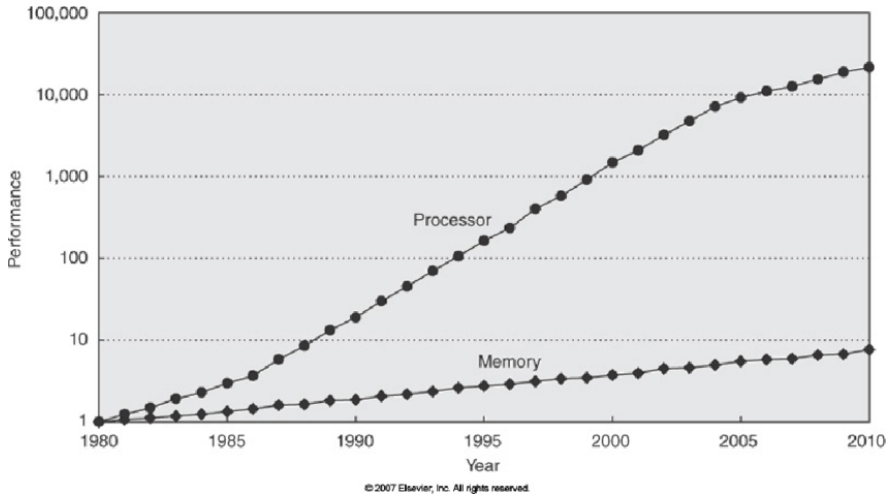


Figure 2.1: Processor-Memory Performance Gap from [13]

where even assuming the best case scenario, the wall would be hit within a few years and concluded "It would appear that we do not have a great deal of time". This highlights the importance of an SoC infrastructure, and in particular an effective memory subsystem, including cache memory.

2.1.2.3 Memory Subsystems and Cache Memory

The memory subsystem is a key part of an SoC, as it can heavily impact the overall performance of the computer by reducing memory access times. Main memory is extremely slow to access, especially in comparison to the CPUs on-chip memory and slows down the overall speed of a system significantly. To combat this, memory hierarchies are introduced consisting of the CPUs on-chip registers followed by varying levels of cache memory, and then followed by main memory. There is also usually secondary storage above this, but I will not go into that in this project [13]. By setting up memory in this way, the average time required for a memory access can be brought down, giving the illusion of a memory that is both large and fast to access at the same time [27].

A key component in this hierarchy which bridges the gap in performance is cache memory. Cache memory is located on, or very close to, the CPU itself and is smaller than main memory. The idea of cache is based on the principle of locality, specifically temporal locality, meaning that a recently accessed memory location will likely be accessed again soon, and spatial locality, meaning that memory locations near a recently accessed memory location are also likely to be accessed soon [13]. A cache will store data based on these principles, so when the CPU wants to access main memory it can first look in the cache for what it's looking for. If it is present in the cache then the CPU will access the cache instead of main memory which is much faster due to the smaller size and proximity of the cache, improving overall system performance [24].

Typically, multiple levels of cache are implemented, called L1, L2 and L3 caches where each level has increased memory size but longer access times when compared to the previous. Each of these cache usually serve a different purpose in the hierarchy and have

a number of different factors that impact their performance. More technical information on cache performance metrics, factors and enhancement techniques can be found in the technical background 2.2 section.

2.1.3 Related Work

In this section, I discuss related work for RISC-V SoC designs. Particularly, I focus on recent designs and implementations, specifically their memory hierarchies, in order to provide insight into the current landscape of the field. A comprehensive list of cores and SoCs can be found at [19], which was maintained up until March of 2021.

An early entry in the area of RISC-V SoCs is the Rocket Chip [8], a RISC-V based SoC generator. It contains the Rocket core [9], which can be configured with L1 data and instruction caches. The Rocket Chip is highly configurable, which is great for optimising it for different applications, but this configurability must come at increased complexity, power consumption and resource utilisation. This isn't good for very specific applications such as high performance computing or embedded systems.

A very relevant SoC to my project, is IOb-SoC [16]. This is an open-source SoC template which uses a picorv32 core and other SoC components, the most notable of which is an optional external memory interface. The picorv32 core [10] implements the RV32IMC instruction set, and is highly configurable. The RV32E, RV32I, RV32IM, RV32IC or RV32IMC instruction set can be chosen depending on the application. Like my design, this SoC implements an L1 data cache but, unlike my design, also has an L1 instruction cache, and L2 unified cache. It has very low resource and power consumption, and is an ideal SoC for teaching and research due to its modularity which allows for varying levels of complexity [14]. It would benefit from more configurability in terms of its memory hierarchy, in terms of associativity, block size etc.

The Parallel Ultra Low Power (PULP) platform aims to develop energy efficient and customisable SoCs for applications such as IoT and high-performance computing [22]. PULP provides single-core SoC designs for IoT applications, such as PULPino [2] and PULPissimo [25]. PULPino is the simplest of the two, and has a very simply memory subsystem, as it only contains separate instruction and data memories, each of which are 32kB. The lack of memory subsystem hinders PULPino when it comes to executing larger tasks. PULPissimo on the other hand, has a much more complex memory subsystem. It includes a tightly coupled data memory (TCDM, the same as the DCCM in the RV32IM core) for quick data access, and 512kB L2 memory which can be configured as a cache. These memories are also shared with the peripherals. In addition to all of this, PULPissimo also has a memory controller to connect to external memory. This does of course have limitations, as it is specifically for low-power applications and may not give the best performance for other applications. However, it is very successful in achieving what it is made for.

One of the most prominent providers of RISC-V based SoC designs, which are then manufactured and sold, is SiFive. Their cores cover a wide range of applications such as high performance computing, low-power systems, and machine learning. SiFive divide these cores into different families, allowing the consumer to choose an appropriate

design for their needs. Along with this, they also provide their own core designer for a consumer to create their own configuration for a specific application [23].

SiFive and other company's designs, as well as the fact that 10 billion cores have been shipped worldwide [11], serves as an example of the wide range of real-world applications of RISC-V cores and RISC-V based SoC designs with complex memory subsystems.

2.2 Technical Background

2.2.1 RV32IM Core

The RV32IM core [26] is fairly simple. The 5-stage pipeline diagram provided in figure 2.2 shows how instructions progress through the pipeline.

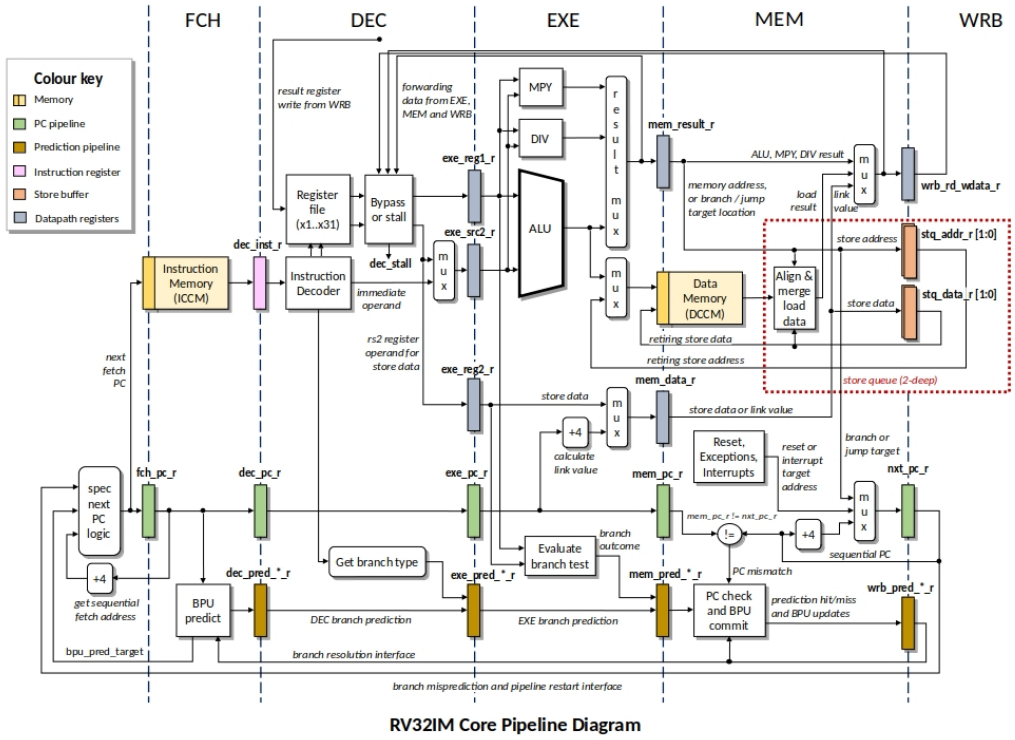


Figure 2.2: The pipeline of the RV32IM Core

The stages can be described as follows:

- **FCH (fetch)**: In this stage, the instruction is fetched from the ICCM using the program counter (PC) as an index. The PC is also incremented in this stage.
- **DEC (decode)**: It then progresses to this stage where the instruction is decoded and any required values are taken from the register file.
- **EXE (execute)**: In this stage, the core carry out any arithmetic or logic operations required. If the instruction is a LOAD or STORE then it compute the address to

be loaded from or stored to and set the appropriate memory access signals for DCCM and ICCM.

- **MEM (memory)**: This stage is only relevant for memory access instructions e.g. a LOAD or STORE. If the instruction is a LOAD then the address computed in EXE is used to read from the DCCM, if it is instead a STORE then the data and address are committed to the store queue (STQ) and then later retired in the WRB stage. The reason it must commit to the STQ first is that there could be data that becomes invalidated by an instruction at an earlier stage in the pipeline, so the core waits until it knows that the data is valid before retiring it to DCCM. The existence of the STQ allows the core to run uninterrupted when a STORE occurs, hence increasing performance. There is one structural hazard accounted for when the STQ wants to retire an entry and there is a LOAD instruction in the EXE stage, as both cannot access the memory at the same time.
- **WRB (write-back)**: In this stage, the core writes back to the register file, or the DCCM in the case of a STORE. Note that the STQ cannot retire to the DCCM if there is a DCCM load at the exe stage as they can't both access the DCCM at the same time, hence the STQ waits.

The core uses a memory map, as seen in table 2.1, to differentiate between ICCM and DCCM accesses. The ICCM is 8192B in size, hence bit 13 of the address will be 0 if the access is for ICCM, and 1 if it's for DCCM. Note that, when talking about address bits, I will be counting from 0, hence for a 32-bit address the most significant bit will be referred to as bit 31, and not bit 32.

Memory	Memory Area (hex range / size)
ICCM	0x00000000-0x00002000 / 8kB
DCCM	0x00002000-0x00003000 / 4kB

Table 2.1: RV32IM Core Memory Map

In its current state the core is very efficient, but it only has two relatively small memories in ICCM and DCCM, which are 8kB and 4kB respectively. This means that while the core is extremely efficient, it lacks viability when it comes to executing larger tasks. The addition of the memory subsystem of an SoC infrastructure will allow the core to execute larger tasks due to having access to main memory, while still retaining a high throughput due to the data cache. While it is preferred for to understand the pipeline to its full extent, it doesn't need to be in order to implement the data cache, as will be seen when discussing the design of it.

2.2.2 Cache Design Principles

This section begins by describing cache performance metrics. These are essential to understand before moving onto cache performance factors and enhancement techniques.

Hit Rate and Miss Rate

The two most basic performance metrics are hit rate and miss rate. Hit rate is simply a percentage measure of how many of the accesses result in a hit, whereas

the miss rate is the opposite. The aim is to maximise hit rate and minimise miss rate [13].

That naturally leads to the question of what causes a cache miss, and hence affects the miss rate. Cache misses can be divided into three categories, which are crucial to understand in order to improve performance:

- Compulsory misses - These are also known as cold misses and occur when the cache blocks are empty and a data block is read from main memory for the first time [13]. These misses must happen, and more will occur depending on the number of blocks in the cache e.g. block size gets smaller or cache size gets bigger.
- Conflict misses - These misses occur when multiple data blocks from main memory map to the same block in the cache, which causes the old one to be evicted. Lower associative caches are more susceptible to this type of miss.
- Capacity misses - These misses occur when a cache isn't large enough to hold all of the blocks required to run a program, and so a miss occurs because a block that was previously in the cache has been evicted due to capacity.

Cache Access Time and Miss Penalty

Cache access time is a measure of how long an access to the cache takes, and is important to understand how much is gained from accessing the cache vs a longer access to main memory. Miss penalty is the time that will be required for the memory access if it cannot be found in the cache, and main memory has to be accessed. This metric highlights the importance of reducing misses [24, 13].

Average Memory Access Time

Average memory access time (AMAT) is a more extensive metric that combines the previous four metrics into one and is calculated using the following formula from [13]:

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

AMAT is a good metric to quickly and comprehensively assess the effectiveness of a cache.

Now that the metrics are understood, let's look at the factors that impact performance.

Associativity

Perhaps the most important factor to consider for a cache, is its associativity. There are three main types of associativity; direct mapped, set associative and fully associative which can be seen in figure 2.3.

Fully associative provides the lowest miss rate of the three, since a block from memory can go anywhere in the cache, hence eliminating conflict misses entirely. However, this comes at an increased cost of higher access time and power consumption as the entire cache needs to be searched to find a block.

Direct mapped is the complete opposite of fully associative. It has the highest miss rate of the three since a block of memory has only one possible location in

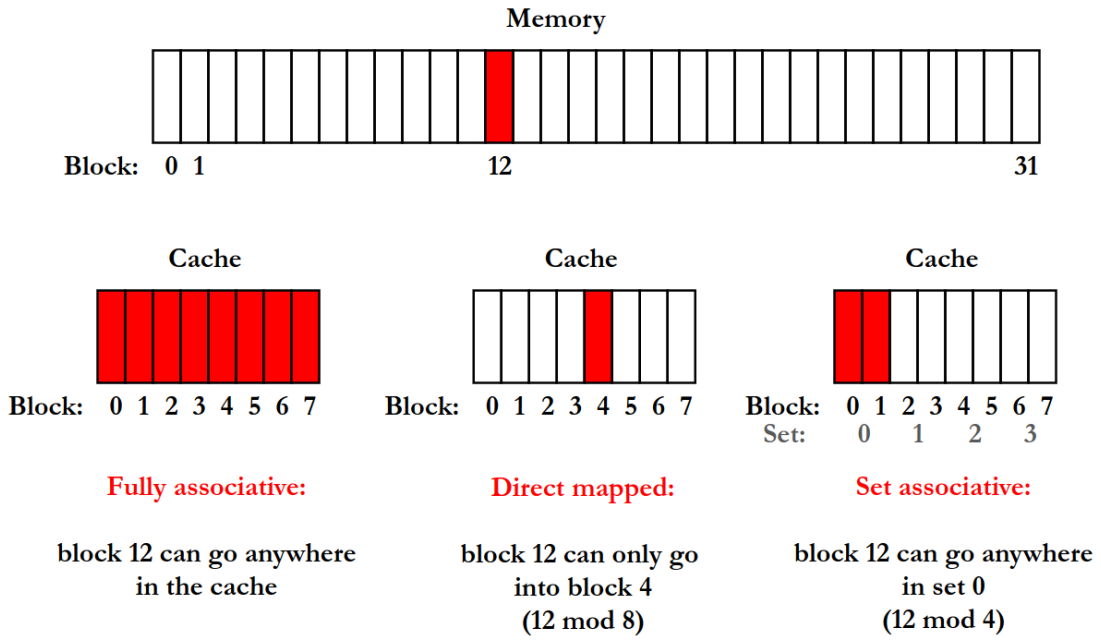


Figure 2.3: The Three Types of Cache Associativity [27]

the cache leading to more conflict misses. Although, this approach does have the quickest access time since there is only one location the block could be in. It is also the most straightforward to implement, which is why I have chosen it for my design and implementation.

Set associative is a mix of fully associative and direct mapped. The cache is split into a number of sets and a block is mapped to only one of these sets (similar to direct mapped), but can be placed anywhere within that set. The degree of associativity of a cache e.g. 2-way or 4-way set associative can play a large role in the overall performance of a cache.

Block Size

The block size of the cache is the number of bytes that are fetched from the main memory and stored together. Increasing the block size improves both temporal and spatial locality, as well as reduces the number of compulsory misses, however it can cause cache pollution. Cache pollution is where useful data in the cache is displaced by other data that isn't used. Hence, increasing the block size can cause more conflict misses [13]. Increasing the block size has positives and negatives and often the optimal size depends on the specific application.

Cache Size

Naturally, increasing the size of a cache will increase its hit rate as it can hold more blocks from main memory, hence reducing the number of extremely slow accesses to main memory. On the other hand, increasing the size comes at the cost of increased cache access time and complexity, since the larger size takes more time to search and manage [13]. Similar to block size, this can be specific to each application.

Replacement Policies

When the cache, or a set in the cache, is full, there needs to be a way to decide which block to evict and replace with a block from main memory when a miss occurs. This is where cache replacement policies come in. Note that, these policies do not apply to a direct mapped cache, which has been implemented in this paper, but they are still relevant to talk about for future work.

Least Recently Used (LRU) is one of the most common policies. When a block must be evicted from the cache, this policy will have kept a history of when each was last used, and will evict the one that hasn't been used in the longest time, replacing it with a block from memory [5].

Another is First In, First Out (FIFO) where the usage of a block doesn't matter, it simply evicts the oldest block no matter what.

A very low effort policy is Random Replacement (RR). RR provides a completely unbiased method of replacing blocks, which is good in some situations and is very easy to implement. However, the performance of RR is, in most situations, very unpredictable.

The most efficient replacement algorithm would be one that works by replacing the block that will not be needed for the longest time. This is known as Bélády's algorithm, and would obviously be the most optimal. However, it is not practically implementable as it requires knowledge of future memory accesses [5].

Write Policies

The write policy of a cache determines how the cache acts when performing a write operation. There are two main write policies that are used. The first is a write-through policy which means that whenever the cache is written to, the cache writes this data back to main memory also. This ensures that main memory always has the most up to date data, however it leads to increase access times and more traffic between the cache and main memory since it must be written to on every write [24, 13]. The second is a write-back policy where write operations only write to the cache. Main memory is only updated when a cache block is evicted, decreasing access times and the traffic between main memory and the cache. The only problem is that this can introduce coherency problems between the cache and main memory as it might not contain the most up to date data, this is only a problem in multi-core systems however [13]. The policy used in this project is a write-back one as I am only working with a single core, and so there are no coherency problems to deal with.

There are also a number of enhancement techniques that can be used to reduce specific types of miss.

Prefetching

Prefetching is a powerful technique which can significantly reduce the number of compulsory misses if implemented correctly. It attempts to predict future memory accesses, and brings in blocks that are likely to be used later. There are a number of ways the blocks can be chosen and this must be carefully considered, as otherwise prefetching could cause cache pollution. [27, 13].

Victim Caching

A victim cache is a small and fully-associative cache that is placed between two cache levels, such as between the L1 and L2 cache in a memory hierarchy. When a block is evicted from the L1 cache, it is stored in the victim cache, which essentially gives the block a second chance of being accessed before being fully evicted. This targets conflict misses, as the victim cache acts as a buffer for the evicted blocks.

Chapter 3

Design

3.1 Memory Hierarchy

The initial consideration for my design was what would it look like at a high-level. Many choices were possible for memory hierarchies, ranging from a simple L1 data cache that interfaces directly with main memory, to more complex configurations such as L1 data and instruction caches, with an L2 unified cache. Of course, it is clear that the more complex designs would be better in terms of making the core viable for larger tasks while still maintaining its current speed, but it makes the most sense to start with a simple design and then work up to a more complex one. For this reason, I chose to go with a simple L1 data cache design which will interface directly with main memory. Given more time, this hierarchy would have simply been a stepping stone to a more complex design and implementation. Despite this, I have still discussed how the design would change if I changed different aspects of the cache, or added new cache memories to the hierarchy, towards the end of this chapter.

3.2 Data Cache

Before beginning the cache design, there were some considerations. The first of these was the specification of the cache itself. For simplicity, I chose to go with a 1kB, direct-mapped cache with a write-back policy and a 32B block size.

I could then calculate what the effective memory address for a cache access would look like. I needed to calculate three values, the block offset, the index and the tag. The block size is $32 = 2^5$ bytes, so the first 5 bits (0-4) of the address will be for the block offset and will allow the access to choose a specific byte within the block. Next there's the index, which selects the block in the cache being accessed. The cache has $\frac{1024}{32} = 32 = 2^5$ blocks, so the next 5 bits (5-9) will be for the index. The rest of the bits are allocated to the tag, so there are $32-5-5 = 22$ bits (10-31) for the tag. This effective memory address can be seen in figure 3.1.

The layout of a block in the cache can be seen in figure 3.2. The block has 256 bits (32B) of data since that is the block size, it then has 22 bits dedicated to the tag as

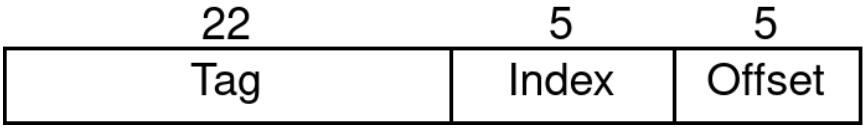


Figure 3.1: Effective Memory Address

discussed before. Finally, there are 2 flag bits. These flags are the valid and dirty bit flags. The valid bit indicates if the data for that cache block is valid. The dirty bit is required since the cache uses a write-back policy, so main memory may not have the most up to date data for a block. If a block is dirty then that means that main memory doesn't have the most up to date version and the block must be written back to main memory before being evicted from the cache.

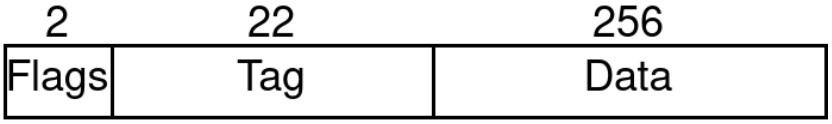


Figure 3.2: Cache Block

It then made sense to split each piece of the cache block into its own memory. The tag and the data would be their own separate block RAMs, with the tag memory having 32 entries, one for each block, and the data memory having 256 entries, one for each 4 byte word. The flag bits could then be simple arrays of length 32, which indicate if a specific block is valid or dirty. The original core uses a memory map to differentiate between ICCM and DCCM accesses, so I chose to extend this and add a cacheable memory area of the map, as seen in table 3.1. This can be accessed by looking to see if bit 31 of the address is asserted.

Memory	**Memory Area (hex range / size)**
ICCM	0x00000000-0x00002000 / 8kB
DCCM	0x00002000-0x00003000 / 4kB
Cacheable Data Memory	0x80000000-0xFFFFFFFF / 2GB

Table 3.1: RV32IM Core with Data Cache Memory Map

The next consideration was where in the pipeline the cache would fit. Since it's a data cache it makes sense to have it alongside the DCCM in the MEM stage as this is where data is loaded and stored in the pipeline. The two memories are now in parallel, so there needs to be extra logic in the EXE stage to decide which memory a load or store is for as well as the inputs for each memory. Looking to the inner workings of the cache itself, it is required to:

1. calculate if the access is a hit or miss
2. in the case of a miss, handle it appropriately by:
    - stalling the pipeline until the miss has been dealt with

- reading data from cache memory and writing it to main memory, if it is dirty
- reading data from main memory and writing it to cache memory

First, I added a tag check to decide if the access is a hit or miss and then a miss handling finite state machine (FSM) both in the MEM stage after the memories themselves. If a hit, the FSM will not do anything, it will just allow the pipeline to continue as normal. In the case of a miss, the FSM will cycle through its states to read and write from the memories, depending on which dirty bits are set etc. Finally, to interface with main memory, I added registers between the MEM and WRB stages which will be passed along to main memory. I will also required some logic to set the memory enable and write signals for the main memory based on the outputs of the registers from the FSM.

The final considerations were where inputs should come from (STQ, ALU etc.) and where the outputs should go (read data, stall signal etc.). Inputs will mainly come from the ALU and STQ, similar to how DCCM works in the original pipeline. The only addition to this is inputs that come from main memory and the FSM in the case of a cache miss. Outputs such as read data will work similar to DCCM, with the addition of the stall output signal, which will just be added as an additional signal to freeze the MEM stage.

With these considerations taken into account, the initial design of the data cache was created, and is shown in figure 3.3. This design was altered a number of times during implementation, either because the change in design made the implementation simpler, or because the design contained errors. I will go into detail on these changes and why they were made in the implementation chapter. For now, the final design is shown in figure 3.4.

Everything within the red dotted border is within the cache module itself. As discussed, it is implemented across the EXE, MEM and WRB stages. Let's go through each stage and the components involved to understand the organisation of the system and how the components work together.

3.2.1 EXE Stage

In the EXE stage the result from the ALU and the STQ retire data and address are passed to the cache as well as the DCCM. The cache is obviously more complex than the DCCM, so I have three muxed inputs, two for the data memory and one for the tag. The results for each MUX for various memory requests are shown in table 3.2

Mem Req / MUX	MUX result			
	FSM	EXE LOAD	EXE STORE	STQ REQ
dd_addr	FSM address	ALU result	ALU result	STQ address
dd_wdata	Mem data	X	X	STQ data
dt_addr	FSM address	ALU result	ALU result	X

Table 3.2: MUX result for each MUX when a specific memory request is made

Of course, all of these memory requests can't happen at the same time as they introduce

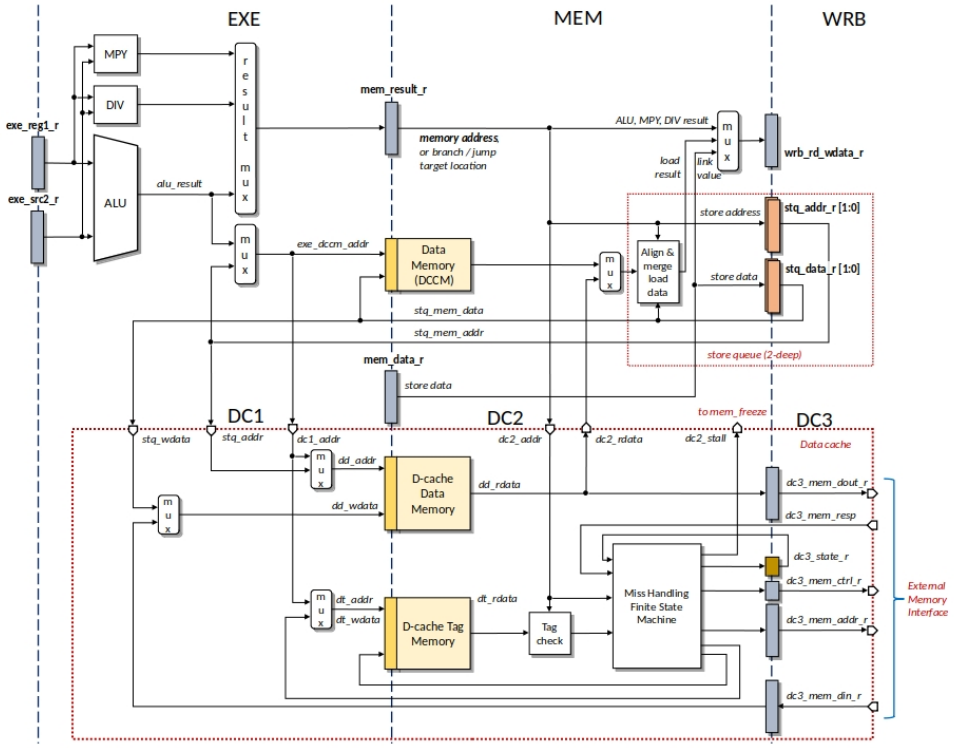


Figure 3.3: Initial Data Cache Pipeline

structural hazards. To get around this, I defined an order of priority and decide which combinations of requests are allowed.

The main hazard is when the STQ has something to retire while there is a load in EXE for the same memory e.g. if the STQ wants to retire to the cache data memory it can only do so if there is not a load for cache data memory in the EXE stage. This is because they use the same signals and the core can only handle one of them accessing the memory at a time. In this case, the core always give the EXE load priority, all other combinations of memory accesses are allowed. The only other exception for our priority is when the cache is busy dealing with a miss. In this case, the pipeline is frozen so no EXE loads or stores are given access during this time. The STQ will be given access briefly so it can empty any data it holds before the cache deals with the miss, otherwise the data read from or written to could be out of date.

3.2.2 MEM Stage

Most of the action in the MEM stage happens in the cache module. Below is a list of events along with a description of what happens in response to that event.

Note that the event where the block is dirty is the same as the one where it isn't dirty, it just has to do write the old block back to main memory before it can read the new one. To go into more depth, if a block is dirty the FSM must go through what is called a "write-back", where it reads the old block from cache memory and writes it to main memory. It does this by initiating a sequence of reads from cache memory closely

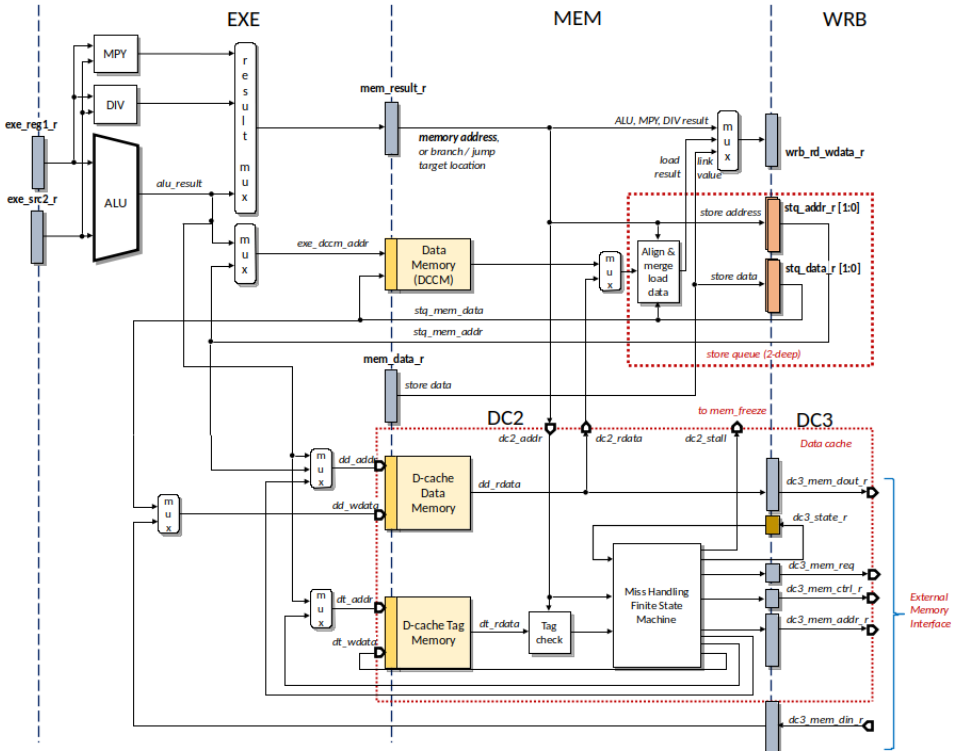


Figure 3.4: Final Data Cache Pipeline

followed by a sequence of writes to main memory by asserting the appropriate signals. Note that the RV32IM core is 32-bit, hence only 4 bytes of data can be transferred in a single cycle, so for a 32B block size the FSM must carry out 8 reads and writes to transfer the whole block. The number of cycles required would change with block size.

If the block is not dirty, or after it has completed the write-back if the block was dirty, the FSM moves on to read the new block from main memory and write it to cache memory, going through what is called a ”refill”. It does this similarly to a write-back, except this time the signals being asserted are the other way around and it’s initiating a sequence of reads from main memory closely followed by a sequence of writes to cache memory.

3.2.3 WRB Stage

There isn’t too much to say about the WRB stage, except that it carries the signals from the FSM and the data memory to the main memory. There will be a small amount of logic here to translate the signals into memory enable and write enable signals for main memory. As can be seen in the pipeline diagram for the final design, the signals carried change slightly during implementation.

Event	Description
Tag Check	The tag given by the tag memory is compared to the tag from the effective memory address to decide if the access is a hit or a miss, this is then passed on to the miss handling FSM
Hit	The access was a hit, so the FSM will do nothing and the load or store access will continue as normal
Miss	The access was a miss, so the FSM will assert the `dc2_stall` signal to freeze the MEM stage until the miss is dealt with and then waits until the STQ is emptied
STQ Empty	The STQ has been emptied so now the FSM carries out a check to see if the block it is evicting is dirty or not
Block is Dirty	The block is dirty so the FSM must write the block back to main memory before it can be evicted and replace by the new block from main memory
Block is not Dirty	The block is not dirty so the FSM can evict and replace the block with the new one from main memory without having to first write it back to main memory

Table 3.3: Memory Stage Events

3.3 Further Design Considerations and Discussions

In this section, I discuss possible changes/additions for the memory hierarchy, and how they would change the design. I begin with performance factors that could be changed for the L1 data cache, but these also apply to the cache memories that are discussed later in the section.

3.3.1 Performance Factors

Associativity

One of the most obvious changes that could be made to the current data cache design is it's associativity to be set-associative or fully-associative. As discussed in the background 2, as the associativity increases so does the hit rate, but this comes at the cost of increased power consumption and access time. Different levels of associativity could be implemented and then analysed using different evaluation benchmarks to determine which is best for a specific use case.

More associativity would mean significant changes in logic for components like the tag check and the FSM as there are multiple blocks that data could be in, and all of them must be checked. Overall increasing the associativity could be beneficial, but does have drawbacks of increased power consumption, access time and complexity.

Block Size

A simpler change to make would be block size. Little change would need to be made to components like the FSM as it would only require more data transfers if the size was increased, and less if it was decreased. As discussed in background 2,

this would also be dependent on the application of the core and its memory access patterns. For example, a larger block size may end up causing cache pollution for an application where accesses are more random, but would be more efficient for an application with lots of spatial locality with data.

Cache Size

Another simple change would be the size of the cache itself. Increasing would obviously increase the power consumption and form factor, but would obviously have a higher hit rate.

Replacement Policies

An aspect that isn't considered for the current design, because it isn't applicable, is a replacement policy. The easiest of these to implement would be RR, as it only requires the ability to generate a random number, so little would change with the current design. It can be unpredictable in terms of performance however as it doesn't keep track of how recently or frequently a block has been used.

Then there's FIFO which is also fairly easy to implement as it only requires the cache to keep track of insertion order and can work well with predictable accesses. On the other hand, FIFO also doesn't take into account how recently a block has been used which results in blocks being evicted when they are likely to be used again.

LRU is better than RR and FIFO when it comes to access patterns that have good temporal locality. However, it can sometimes perform poorly when there isn't good temporal locality, or for very specific loops of data e.g. imagine a loop which runs through a sequence of data blocks where the number of data blocks is slightly larger than the cache, then a fully-associative cache with an LRU policy would be constantly evicting a block that it's just about to use, leading to constant misses during that loop. This is an example of cache thrashing, where the memory locations being accessed are larger than the cache memory, so data that is about to be used is constantly displaced. The complexity of implementing LRU increases with our caches associativity as it becomes more and more difficult to maintain a history of accesses as it must be updated on hits and misses, which can also introduce slower performance.

Write Policies

The data cache currently uses a write-back policy which has increased complexity compared to write-through as it keeps track of dirty bits. Write-through would be simpler in terms of the design as these dirty bits could be removed, but it would increase the number of writes to main memory as well as power consumption.

3.3.2 L1 Instruction Cache

Perhaps one of the more obvious changes that could be made is the addition of more cache. In particular, the next step would be to allow the core to access main memory to retrieve instructions instead of just using the 8kB on-chip ICCM. An L1 instruction cache could then be added to allow the core to execute larger programs while still maintaining its speed by reducing fetch latency.

This instruction cache would sit at the start of the FCH stage, like ICCM. The core is already setup to handle separate instruction and data memory accesses, and it would just be a case of changing the memory mapping to have a specific section of main memory setup for instructions and the instruction cache, while another would be for data and the data cache. An example mapping can be seen in table 3.4, where main memory takes the upper half of the memory space. Then, instructions take up the lower 256MB of that space, and data the upper 1792MB. The core would look at bit 31 to decide if the access was for main memory, and then bit 28 to decide if it was for instruction or data memory. There are many ways this mapping could be arranged and this is just an example.

Memory	**Memory Area (hex range / size)**
ICCM	0x00000000-0x00002000 / 8kB
DCCM	0x00002000-0x00003000 / 4kB
Main Memory	0x80000000-0xFFFFFFFF / 2GB
Cacheable Instruction Memory	0x80000000-0x90000000 / 256MB
Cacheable Data Memory	0x90000000-0xFFFFFFFF / 1792MB

Table 3.4: Example Mapping with added Instruction Cache

This instruction cache would then build upon the current set of signals used for the ICCM in the core, just like the data cache does for the DCCM, and would also require its own miss handling FSM controller. The complexity of this FSM, like the data cache FSM, would depend on several factors such as associativity and write-policy which have been discussed previously. It would make sense to have a larger block size to take advantage of spatial locality for the instruction cache, such as 64B, since there is the assumption that instructions near each other in memory are likely to be executed in sequence.

This would of course come with increased complexity and power consumption, but the gains in viability while maintaining the cores efficiency would offset this. It would also require its own functional verification, but this verification can be done completely separately to the already existing data cache and so would be fairly straightforward.

3.3.3 L2 Unified Cache

After an L1 instruction cache, the next logical step is to add an L2 unified cache to the design. This cache would hold both data and instructions and would require significant modifications to be integrated into the design. It would fit in between the L1 caches and main memory, and the L1 caches would now send their request to the L2 cache instead of main memory directly. This would require another cache controller/miss handling FSM of sorts and it may be best to merge the existing controllers for the data and instruction caches into one since they now share a cache. This would likely be more straightforward to understand and reduce the chance of implementation errors as all the signals would be sent to one place.

Misses should still be fairly straightforward to deal with. If there is a miss in an L1 cache, then the data or instruction would be requested from the L2 cache. If it isn't found then the L2 cache is now the one that requests the data or instruction from main

memory itself. The L2 cache would fetch the data for that miss and store it before then responding to the L1 cache that requested in the first place. The logic for the L1 caches would stay mostly the same since, to them, they are just waiting on the memory above them to give a response.

This cache has a higher chance of causing cache thrashing since data and instructions are contending for the same space in the cache. For example, a program may have lots of data memory accesses which could evict instructions from the cache that are soon to be executed. The cache will then need to request these from main memory, which could then evict a data block that is soon to be used and so the two are constantly displacing each other and slowing down the system.

A problem that could arise with this design is cache consistency. Cache consistency ensures that all cache in the hierarchy have a consistent view of memory. For example, let's say that the L1 data and L1 instruction caches have write-through policies and both hold the same block for address A, if an instruction modifies block A, then it gets written to the L1 data cache, which then writes through to the L2 unified cache. The problem here is now that the L1 instruction cache has an old version of block A. Hence, cache consistency mechanisms would be an important design consideration. It is important to note that this would only be a problem if the design moved away from using a memory map, like in table 3.4, since a block can only be in the L1 data or L1 instruction cache with a memory map, not both.

3.3.4 L2 Split Caches

An alternative to an L2 unified cache would instead be split caches. In this configuration, instructions and data wouldn't share a cache, and the already existing L1 caches would just have an extra cache above them, an L2 data cache and an L2 instruction cache. These caches would be larger than the L1s and hence slower to access, with their own associativity, block size etc.

Like the unified cache, they would also fit into the design between the L1 caches and main memory and service requests from the L1s and send their own requests to main memory. Unlike a unified cache, it makes sense here to keep the controllers separate since it is just building upon the already existing L1 caches, but there is still the option to have a shared controller if it makes sense to do so.

There are problems with a unified cache that either don't have to be dealt with for split caches, such as cache thrashing, or are made significantly easier to deal with, such as consistency. The two caches could also be optimised separately which leads to better performance. However, there is potential for higher complexity as there is now two separate caches instead of one unified cache to deal with which also leads to higher power consumption.

The choice between the two comes down to the specific requirements of the systems application.

3.3.5 Configurability

The configurability of a design is an extremely important aspect. When I refer to configurability I'm talking about how easy it is to modify parameters of the design such as changing a caches associativity, block size, replacement policy or even changing the memory hierarchy itself.

This has numerous benefits. It makes the design adaptable to different applications as each will have their own resource constraints and desired performance. For example, for high performance computing fast memory accesses are required, with a configurable design the hierarchy and caches could be modified to be fully associative with larger sizes. This would have higher power consumption, but would ensure the needs of the application are met.

It also makes it easier to test different configurations of the memory hierarchy and cache memories and compare them with each other. This would allow me to further optimise the design and find out which configuration suits each application best.

Chapter 4

Methodology

This chapter details the overall approach taken to achieve the goal of this dissertation, so that the results are reproducible later on.

4.1 Vivado

Xilinx's Vivado is the main software that has been used to realise the design, as it provides an extensive set of tools to implement, simulate and synthesis designs. The first of these tools is Vivado's IDE which allows the design to be implemented at a high level using a hardware description language (HDL), which in this case is Verilog. It provides useful syntax and error checking.

Xilinx's Vivado is the main software that has been used to realise the design as it provides an extensive set of tools. The first of these tools is just the IDE provided by Vivado, where an HDL can be used to implement a design at a high level. Vivado also provides a strong set of simulation tools which can be used to simulate the behaviour of the data cache and core. These simulations are invaluable for two reasons, 1) they can be used to verify the functionality of the cache by simulating in conjunction with Vivado's debugging tools which allow waveforms to be viewed, and 2) they can be used to run benchmark programs which measure various performance metrics for the cache.

Another extremely useful tool Vivado provides is synthesis and implementation for designs, which can be programmed onto an FPGA board. This would be the logical next step in this process, and even thought I don't put the design onto an FPGA, the synthesis and implementation still provide valuable feedback with error and warning reports as well as reports on resource utilisation, timing and power usage.

4.2 RISC-V GNU Toolchain

Next, there's have the RISC-V GNU toolchain to compile RISC-V assembly files into hex files which can be used in simulation for functional verification and evaluation.

4.2.1 Verification and Evaluation

To verify the design of the cache, I used two hex files, one for general debugging, and another which tested various aspects of the caches functionality, for a more complete verification. The same was done to evaluate the performance of the cache. The RISC-V assembly for these hex files can be found in listings A.1,A.2,A.3. These were compiled using a script which ran a sequence of commands from the GNU toolchain:

- `riscv64-unknown-elf-as -march=rv32im -o init.o init.s`, this creates the '`init.o`' object file based on the '`init.s`' assembly file. '`-march=rv32im`' specifies the target RISC-V architecture, which in our case is the standard rv32i architecture with the M extension for integer multiplication and division. This file is for initialising the program.
- `riscv64-unknown-elf-as -march=rv32im -o $prog.o $prog.s`, this does the same as the previous command, but with the file specified as $prog in the command line when the script is run. This file is the main body of the program.
- `riscv64-unknown-elf-ld -m elf32lriscv -o $prog.out -T link_map.txt init.o $prog.o`, links the two object files, '`init.o`' and '`$prog.o`' into an executable program '`$prog.out`'. '`-m elf32lriscv`' specifies the format for the output file, which in this case is a 32-bit little-endian ELF format for RISC-V. '`-T link_map.txt`' is the link script which decides how the two object files should be linked to produce the output file.
- `riscv64-unknown-elf-objdump -D -s $prog.out > $prog.dump`, generates a disassembly '`$prog.dump`' from the executable '`$prog.out`', this is particularly useful for debugging.
- `riscv64-unknown-elf-elf2hex --bit-width 32 --input $prog.out --output $prog.hex`, takes an ELF executable file '`$prog.out`' and converts it into a hex file '`$prog.hex`' to be used with the simulation. '`--bit-width 32`' indicates the word size for the hex file.

Chapter 5

Implementation

This section delves into the practical aspects of how I implemented the data cache design in Vivado, including testing and verification. To make the implementation easier, I decided to divide it logically into sections that could be implemented in isolation, with the only knowledge of other sections being inputs given to them or outputs taken from them. The design already lays out where important signals come from and go to, so I already had this knowledge going into the implementation. My only concern was possible discrepancies between the sections that I've isolated, but I was confident in my ability to solve these discrepancies as they arose.

5.1 EXE and MEM Stages (Outside Cache Module)

I began with the EXE and MEM stages outside of the cache module, and decided to setup the straightforward signals in the MEM stage first.

- `mem_result_r` added as an input for the cache module
- `dc2_rdata` muxed with DCCM read data
- `dc2_stall` added as input for `mem_freeze` signal

The decision to extend the already existing memory map was made in the Design 3 chapter, hence bit 31 will be looked at to decide if the access is cacheable or not. If bit 31 is not asserted, then the core will then look at bit 13 to make the choice between DCCM and ICCM.

I then moved onto look at inputs to the first stage of the cache module. As I began to add these I realised that a lot of signals were having to be input into the cache module just to control the MUXes for the memories, I felt this was unnecessary and it would be much more straightforward to do have the logic for the MUXes outside of the cache module, where the signals required to control them were already present, and simply propagate the outputs from the MUXes into the cache module. This change can be see in the final design in 3.4 as the red dotted box for the cache module has been reduced to begin at the end of the EXE stage.

Next to implement was the larger task of the MUXed inputs to the cache. For this, I required new signals which would be used to enforce the order of priority and stop structural hazards from occurring while the data cache is integrated into the current design. A list of new signals and the reason for them being defined can be seen in table 5.1. The `req` signals indicate that there is a memory request coming from a specific location, for a specific memory e.g. `exe_dd_req` indicates a request from the EXE stage to access the cache data memory. The `ack` signals indicate that an access from a specific location, for a specific memory is allowed to go ahead e.g. `stq_dd_ack` indicates that the STQ is allowed to retire data to the data cache memory.

While defining these signals, I realised two errors in the design. The first being that the ALU result first went through a MUX for the DCCM before getting to the MUX to decide `dd_addr`, which doesn't make sense. This was rectified by simply taking the ALU result directly to the MUX for `dd_addr`. The second was that there was actually no input from the FSM for the `dd_addr` MUX, which is required for writing data that is read from main memory into the data memory for the cache. This was also easily rectified by adding an output from the FSM and plugging it directly into that MUX.

Signal	**Description**
`exe_dccm_req`	indicates there is an instruction in EXE that would like to access DCCM
`exe_dd_req`	indicates there is an instruction in EXE that would like to access cache data memory
`exe_dt_req`	indicates there is an instruction in EXE that would like to access cache tag memory
`stq_dccm_req`	indicates the STQ wants to retire to the DCCM
`stq_dd_req`	indicates the STQ wants to retire to the cache data memory
`exe_dccm_ack`	indicates the EXE instruction is being allowed to access the DCCM
`exe_dd_ack`	indicates the EXE instruction is being allowed to access cache data memory
`exe_dt_ack`	indicates the EXE instruction is being allowed to access cache tag memory
`stq_dccm_ack`	indicates the STQ is allowed to retire to the DCCM
`stq_dd_ack`	indicates the STQ is allowed to retire to the cache data memory
`fsm_dd_ack`	indicates the FSM is being allowed to access cache data memory
`fsm_dt_ack`	indicates the FSM is being allowed to access cache tag memory

Table 5.1: Signals used to control MUXes at EXE stage

These signals made the process of controlling the MUXes fairly straightforward:

1. The `req` signals are set by using whether there is a load or store in EXE, STQ req and which bits of the address are asserted e.g. `exe_dccm_req = exe_load_r & !alu_result[31] & alu_result[13]`
2. Next, the `ack` signals are set based on the `req` signals and other signals such as `exe_freeze`. This is where the structural hazards are handled by making sure only `ack` one ack signal is asserted for each memory.

3. Memory enable (cen) signals are now decided based on which ack signals are asserted.
4. Finally, the ack signals make it fairly easy to use if statements for the MUXes themselves to decide the input address and data along with the write-enable (wenb) signal for each memory.

5.2 Memory Interface

For the second part of the implementation I focused on the memory interface, as I knew that it would likely have an effect on the states and signals used by the miss handling FSM in the cache module. My original plan for this was to use the AXI bus protocol to interface with the on-board DRAM in the processing system of the Z7020 chip. However, by this time in the project it was fairly close to the deadline and I hadn't used AXI protocol before, so learning would be a time commitment. Because of this I chose to forgo the AXI interface and DRAM memory in the PS, and instead opted to implement my own simple block RAM in the programming logic which could act as main memory. This meant I would have to implement my own protocol to interface with memory, but this was extremely straightforward as the main memory would just be a variation of the existing modules for DCCM and D-cache data memory.

This changed the design as there was no longer any need for a memory response signal as I was working on the assumption that the data will always be found in main memory. I also added a new request signal to indicate that the cache wanted to initiate a main memory access. These changes can be seen in the final design.

The signals used for the memory interface along with an explanation of them can be found in table 5.2.

Signal	Description
dc3_mem_dout_r	carries read data from cache data memory to be written to main memory
dc3_mem_din_r	carries read data from main memory to be written to cache data memory
dc3_mem_addr_r	carries the address which dc3_mem_dout_r will be written to in main memory
dc3_mem_req	indicates to main memory that the miss handling FSM wants to access it
dc3_mem_ctrl_r	indicates if the access is a read or write (0 = read, 1 = write)

Table 5.2: Interface Signals

Some extra logic was also added to translate the req and ctrl signals into the memory and write enable (wenb) signals for the main memory.

The only problem with having memory implemented this way, is that all accesses are uniform in length i.e. they will always take the same number of cycles. This is not as realistic as a real system, and so made evaluation slightly less realistic, but changing

the interface at a later date to interface with a more realistic memory would not be too difficult.

5.3 Cache Module

Lastly, there's the inner workings of the cache module, namely the miss handling FSM as it will control the core when a miss is being handled. The tag check is the most straightforward part of this section as it is simply a comparison of the tag given by the tag memory and bits 31 to 10 of `mem_result_r`, as well as a check that the given cache block has its valid bit asserted.

The FSM itself is less complicated than you would think and only requires 4 states which were briefly touched upon in the design section. The FSM itself can be seen in figure 5.1.

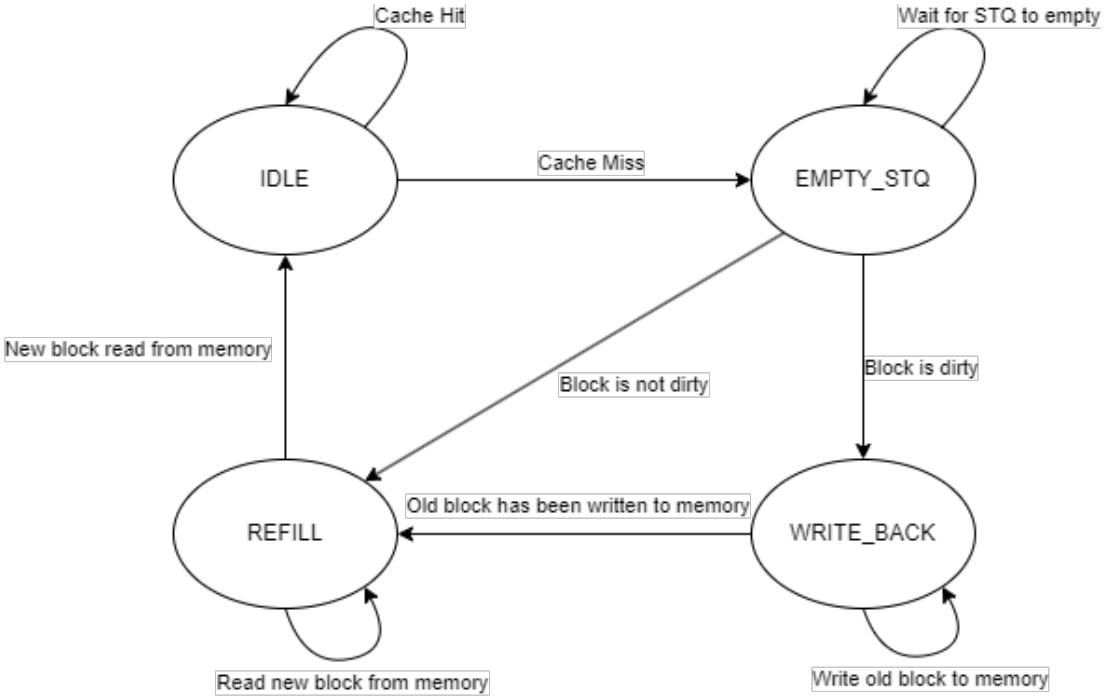


Figure 5.1: Miss Handling FSM

IDLE
: This state simply waits for a cache request, and if it's a miss will assert the `dc2_stall` stall signal, and the `dc2_miss_stq` signal to allow the STQ to empty. It sets the next state to the **EMPTY_STQ** state. If the access is a store, regardless of hit or miss, this state will set the dirty bit for that block.

EMPTY_STQ
: Here the FSM waits for the STQ to empty as it could contain stores that need to be retired before it reads to or writes from main memory. Once the STQ is empty, this state checks if the dirty bit for the block is asserted, if it is then it transitions to the **WRITE_BACK** state, otherwise it transitions to the **REFILL** state. It also deasserts `dc2_miss_stq`.

WRITE_BACK
The FSM is expected to stay in this state for numerous cycles. It reads 4B of the old block from cache memory every cycle, and initiates a write of that data to the main memory in the next cycle. It does this by asserting the signals required to read 4B of the old block in cycles 1-8, then asserts the signals to write to main memory in cycles 2-9. Note that in a real system, this wouldn't be done in a set amount of cycles, and the FSM would instead be waiting for a response from memory to know that the previous data had been successfully written, so the next 4B block can be sent. Once the data has been written to main memory, the FSM transitions to the **REFILL** state. `dc2_miss_dd_r` is used to read from cache memory.

REFILL
This state is always the last in the FSM before it returns to **IDLE**. It works similarly to the **WRITE_BACK** state, but in the opposite direction. It asserts the signals required to read the new block from main memory in cycles 1-8 and then the signals to write it to cache memory in cycles 4-11. Note that here the signals to write to cache memory are delayed in their assertion, as it takes 3 cycles between a memory read being initiated and the data reaching the CPU. Once this new block has been written to cache memory, it deasserts the stall signals and transitions back to the **IDLE** state. `dc2_miss_dd_w` is used to write to cache memory, and `dc2_miss_dt` is used to write to tag memory.

5.4 Testing and Debugging

My approach to testing and debugging was to first test the basic functionality using a simple test file, as this would expose bugs arising from anything I hadn't accounted for or very obvious mistakes I had made. With these out of the way, I could then use a more comprehensive test file which could verify the full functionality of the data cache and catch any issues that were more hidden.

5.4.1 General Debugging

To carry out general testing, I compiled a simple RISC-V assembly file into a hex file (Listing A.1). This file is fairly straightforward, it allocates space on the stack and loops while doing a sequence of loads and stores, making it perfect for testing the basic functions of the cache memory.

A number of issues were found and fixed during this testing, to name a few:

- Address and Data widths - some signals didn't have the correct widths, causing undefined signals, and incorrect loads and stores
- Valid and dirty bit arrays the wrong size, aswell as being indexed incorrectly
- Main memory clock disconnected - clock was not connected to main memory initially, causing the module to not function at all

These were found by first checking if the correct end result was given (e.g. correct data loaded into cache and main memory), and then examining waveforms related to any incorrect result.

5.4.2 Functional Verification

With these basic bugs found and fixed, it was time to move onto a full functional verification of the implementation, as would be done in industry. Below are the functions that I verified:

Basic Functionality

- Load/Store to non-cacheable memory (does it end up in DCCM?)
- Load/Store to cacheable memory (does it end up in cache?)

Hits and Misses

- Loads/Stores which results in hit
- Loads/Stores which results in miss

Replacement and Write-Back

- Loads/Stores which result in eviction of a block that is not dirty (data is already in main memory before eviction)
- Loads/Stores which result in eviction of a block that is dirty (data is not already in main memory before eviction and is updated)

Cache Reset

- Valid and dirty bits are cleared

Specific Test Cases

- STQ Cache Retire
  - Waits when cache load in EXE
  - Retires when any other instruction in EXE
- STQ DCCM Retire
  - Waits when DCCM load in EXE
  - Retires when any other instruction in EXE

To carry out this functional verification I created a RISC-V assembly test file `funct_ver.s`, which can be found in listing A.2, and compiled it into a hex file for simulation. The file carries out a series of different loads and stores in a specific order to cover each of the criteria listed above.

Evidence

This section contains evidence of some of the criteria listed in the test plan.

Figure 5.2a shows one of the most basic criteria, a cache hit. A request comes in and the hit signal is asserted. There are actually two hits in the screenshot, as the pipeline has a store, followed by a load. It can be seen that, two cycles after the store is in the MEM stage, the data is retired into the data memory from the STQ, as expected.

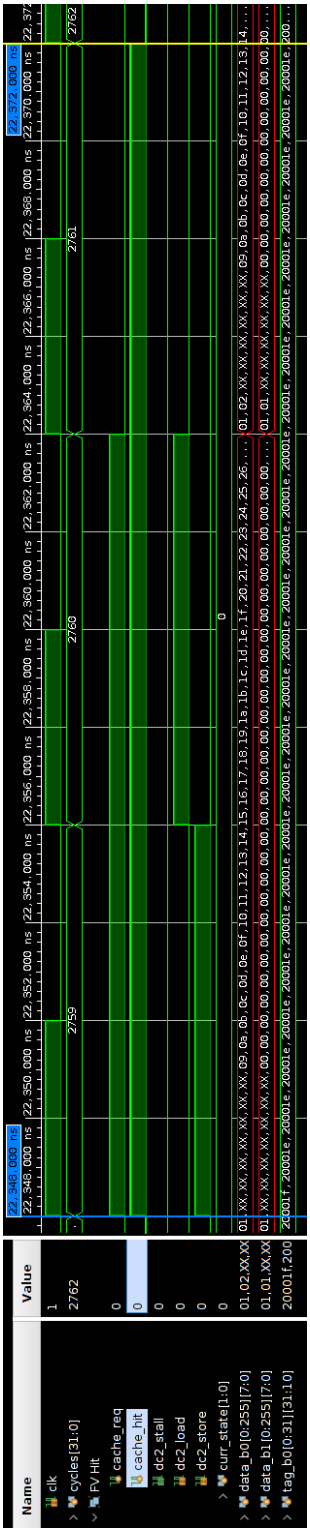
A more complicated criteria is a dirty miss, as shown in figure 5.2b. The first blue marker shows when the request comes in, and the `dc2_stall` signal is immediately asserted as the cache has determined this is a miss, indicated by `cache_hit` being deasserted. The FSM then transitions to the **EMPTY_STQ** state, where it has nothing to empty, so transitions to the **WRITE_BACK** state, which ends at the second blue marker. In this time, it can be seen that signals are being asserted to read from the data memory, and write to the main memory.

After this, the FSM transitions to the **REFILL** state which can be seen between the second and third blue markers. Signals are asserted to read from main memory and write to data memory, as well as write to tag memory at the end of the state to update the tag. Note that `mem_din` is the read data from main memory, and is undefined in this screenshot since that part of main memory is yet to be initialised. It can be seen that the data memory is being written to as the data becomes undefined, and the tag memory is being written to, as the tag changes.

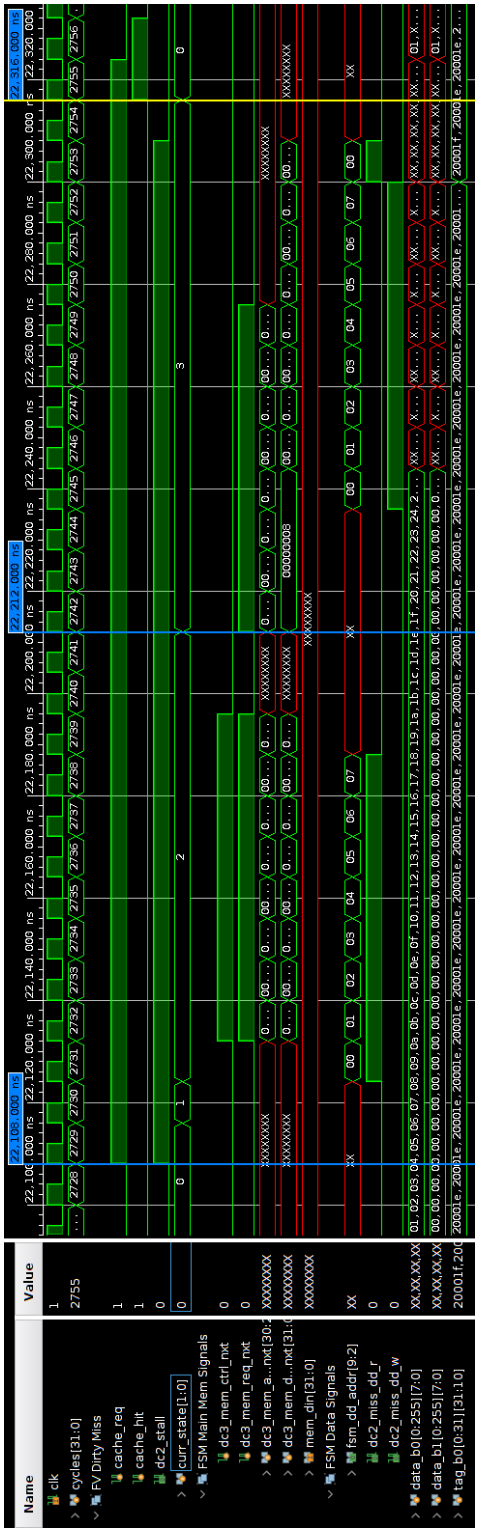
After dealing with the miss, the store instruction retires from the STQ, and hence writes into the data memory, as can be seen after the third blue marker, at the end of the waveform.

Finally, figure 5.2c shows two important criteria; first, the STQ waiting to retire, because there is a load in EXE, and second, the STQ retiring in parallel with a store in EXE accessing the tag memory to complete a tag check in the next cycle.

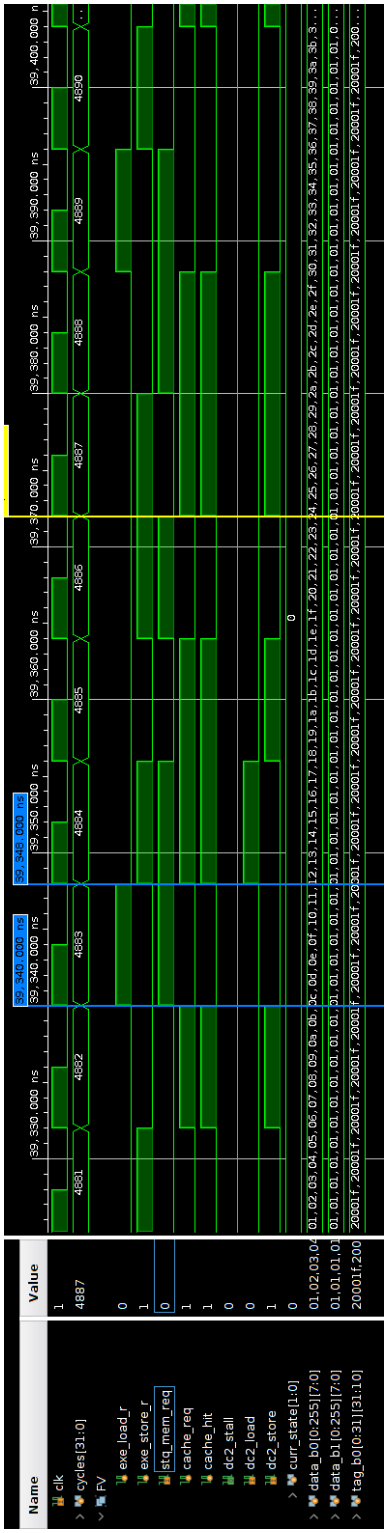
At the first blue marker, it can be seen that there is a load in EXE as well as a STQ retirement request. One cycle later, at the second blue marker, the load has progressed while the STQ has held its retirement, and there is now a store instruction in the EXE stage. Between the second blue marker, and the final yellow marker, it can be seen that the STQ retires its entry and is then empty. At the same time, the store instruction progresses and gets committed to the STQ, resulting in another retirement request.



(a) Signals showing what happens in the event of a hit



(b) Signals showing what happens in the event of a dirty miss



(c) Signals showing STQ waiting for a load to pass, then STQ retiring while EXE store progresses

Figure 5.2: Waveform evidence of some functional verification criteria

Chapter 6

Evaluation

6.1 Vivado Analysis

As previously mentioned, Vivado provides powerful analysis tools which allows me to evaluate the design based on resource utilisation, timing and power usage. Xilinx's PYNQ-Z2 FPGA board [12] was targeted throughout the analysis as this was the board targeted for the original RV32IM design. The Z7020 chip on this board has two sections, the programmable logic (PL) and the processing system (PS). The PL is where my design can be modeled as it contains useful components such as look-up tables (LUTs), flip flops (FFs) and multiplexers among others. These will allow the design to be modeled in order to evaluate the resource, timing and power usage to ensure it fits within the constraints.

For the following analyses, the cache is a 1kB, direct-mapped cache with a write-back policy and a 32B block size as discussed previously. This configuration could of course be changed, and is discussed below where relevant.

6.1.1 Resource Utilisation

Starting with utilisation of hardware resources, it can be seen in table 6.1 that both the number of LUTs and registers have increased, but not by a huge amount. These increases are caused by the addition of components such as tag comparison, FSM, memory control, and valid and dirty bits, among others. The usage of these resources is still well within acceptable standards, so there is a lot of room to work with when considering changes to cache performance factors such as associativity, write policy and replacement policy.

The number of BRAMs used has increased significantly by 10.5, but it should be noted that 8 of these are for the main memory module, which would not be present if the design interfaced with the DRAM in the PS section of the Z7020 chip. 2.5 of these are for the cache itself, which is not a huge increase, so there is still a lot of room to work with to make the cache larger, or add additional cache memories.

The increase in the number of muxes is mainly due to the extra logic for selecting

memory inputs as well as the cache controller logic. This is a very insignificant increase and is still well within the constraints of the chips resources. Many changes could be made to the design to better utilise these muxes.

Resource	Base Core	With Cache
Slice LUTs	2777 (5.22%)	2982 (5.61%)
Slice Registers	2941 (2.76%)	3130 (2.94%)
BRAMs	51.5 (36.79%)	62 (44.29%)
F7 Muxes	257 (0.97%)	263 (0.99%)
F8 Muxes	72 (0.54%)	74 (0.56%)

Table 6.1: Resource Utilisation of Each Design, seen in Figures B.1, B.2

6.1.2 Timing Report

When it comes to timing constraints, it is most important to look at the critical path through the design. The critical path is the longest path through our design, and hence determines our maximum possible operating frequency. As can be seen in 6.2, the original core has a slack of 1.036ns, but the modified one had a slack of 0.721ns. The critical path in the original core design was an instruction which progresses through the pipeline, and performs a load from ICCM. As seen in figures B.3, B.4, the addition of the cache hasn't changed the critical path which is ideal, as the core is now much more viable with the addition of main memory while still retaining its speed due to the data cache.

Metric	Base Core	With Cache
Clock Frequency	75Mhz	75Mhz
Slack	1.036ns	0.721ns
Total Delay	11.870ns	11.841ns

Table 6.2: Timing for Critical Path of Each Design, seen in Figures B.5, B.6

6.1.3 Power Usage

For power consumption, much has stayed the same, as can be seen in table 6.3. The most significant change is BRAM at 52mW, up from 43mW. However, as in the utilisation report, most of this is accounted for by the main memory BRAM and so the actual change from the cache itself is rather small, but still an increase. If the design had instead used the DRAM in the PS section of the Z7020 chip, then I/O power usage would be expected to increase.

Despite the increase in power consumption, even with the BRAMs as main memory, the overhead is still relatively small while still providing the performance benefits of the main memory and data cache.

Metric	Base Core	With Cache
Total On-Chip Power	538	550
Dynamic Power	422	433
Logic	10	9
Signal	14	14
Clock	9	10
BRAM	43	52
MMCM	209	209
I/O	138	139
Static Power	116	117

Table 6.3: Power Usage (mW) from Vivado, seen in Figures B.7, B.8, B.9, B.10

6.2 Cycle Timing

Table 6.4 shows how many cycles it takes to run through each state in the FSM.

State	Cycles
EMPTY_STQ	1-2
WRITE_BACK	11
REFILL	13

Table 6.4: Cycle Timing for Different States in FSM

An optimisation for the **EMPTY_STQ** state, is to skip it entirely when there is nothing in the STQ. The current implementation always transitions from **IDLE** to **EMPTY_STQ**, even if it's already empty. This could save a cycle in a lot of situations. For the **WRITE_BACK** and **REFILL** states, 9 of the cycles are used for reading and writing the block, hence the number of cycles required scales as a function of the block size:

$$Data\ Transfer\ Cycles = \frac{Block\ Size}{4} + 1$$

Table 6.5 makes it clear how large of a factor block size is in transfer time.

Block Size	Cycles To Transfer
16B	5
32B	9
64B	17
128B	33

Table 6.5: Cycles to Transfer Scaling with Block Size

REFILL has an initial 3 cycle delay, as once a memory read is initiated, it takes 3 cycles to reach the cache. This could be better optimised, but with an AXI interface to

external memory, the times would vary as the cache would wait on a memory response. Both states set signals to transition to the next state a few cycles after their final data write, this was done to avoid any possible conflicts that could arise with subsequent accesses being too close. In hindsight, I don't think this would be an actual issue and the states can have their lengths reduced by 2 cycles by setting the transition signals on the same cycle as the final write. Table 6.6 shows the optimised cycle timings.

State	Cycles
EMPTY_STQ	0-2
WRITE_BACK	9
REFILL	11

Table 6.6: Optimised Cycle Timing for Different States in FSM

The cycle timing of full events in the cache can be seen in table 6.7.

Event	Cycles	Optimised Cycles
Hit	1	1
Dirty Miss	25-26	20-22
Non-Dirty Miss	14-15	11-13

Table 6.7: Regular and Optimised Cycle Timing for Cache Events

6.3 Benchmark

My original plan for this section was to run the well known dhrystone benchmark with different cache size configurations. Unfortunately, due to time constraints, I have been unable to properly compile this benchmark to work with my design, so I have created my own simple benchmark file, found in listing A.3, which initialises 16kB of main memory, and then performs 4000 random memory accesses. These random accesses were analysed in terms of cache performance metrics such as hit rate and miss rate, as mentioned in the background. While it isn't as good as running an industry standard benchmark, the one I've created should still be a strong test of the design as it represents a worse-case scenario, where temporal and spatial locality are weak. Ideally, the benchmark would test a wide-range of scenarios and access patterns to give a more realistic view of performance, but evaluating the worst case does give a good indication of the minimum performance.

I also wanted to run this benchmark for multiple configurations of cache size e.g. 1kB, 2kB, 4kB, 8kB, but I was unable to get the design to work in these configurations, also due to time constraints. This raises an important point about the configurability of my design, as had it been created with configurability as a main aim then I would be able to quickly change the parameters to change the cache size, and hence run these benchmarks to compare them.

The current design was still benchmarked using my test program, with the cache in its base 1kB configuration. Table 6.8 shows the hit, miss and dirty miss rate of the design when ran with the benchmark.

Random Accesses	Hit Rate	Miss Rate	Dirty Miss Rate
4K	93.35%	6.65%	6.57%

Table 6.8: Hit and Miss Rate for Benchmark Program

It's important to differentiate between the regular miss rate and the dirty miss rate, as a dirty miss incurs a larger penalty which factors into the calculation for AMAT. The penalty for a non-dirty miss is 14 cycles, while the penalty for a dirty miss is 25 cycles. The AMAT using this benchmark is then:

$$AMAT = 1\ cycle + (0.0657 \times\ 25\ cycles) + (0.0008 \times 14\ cycles) = 2.65\ cycles$$

These are good results considering the simplicity of the cache and the small amount of overhead it adds in terms of resource utilisation, timing and power usage. Of course, the results could be better, but as previously discussed there is a lot of room to build upon this design by changing performance factors.

All of the misses are conflict and capacity misses. The capacity misses can be reduced by simply increasing the size of the cache so it can hold more of the programs data at one time. The conflict misses on the other hand, could be reduced by increasing the associativity of the cache which would allow data to be stored in multiple blocks, instead of just one. Compulsory misses are not accounted for here, but some would have occurred during the initialisation of main memory in the benchmark. These could be reduced by increasing the block size, but this would increase the miss penalty as well as possibly introduce more conflict misses due to data being displaced. Prefetching could also be introduced to reduce compulsory misses.

Overall, this initial design serves as a proof of concept in making the RV32IM core more viable while maintaining its speed. The low cost incurred by the addition of the cache gives full freedom to explore many avenues with different cache performance factors and enhancement techniques.

Chapter 7

Conclusions

7.1 Summary of Findings

To recap, the key objectives of this project were to design, implement and evaluate an SoC infrastructure, including further improvements to the infrastructure and how they would change the initial design.

The design of the data cache is simple, being 1kB, direct-mapped with a 32B block size and a write-back policy. The simplicity of the design is logical, as it allows it to serve as a building block to more complex and efficient designs, as was discussed at the end of chapter 3. The design was divided up to make implementation more straightforward, and then iterated on during implementation, either due to errors or for ease of implementation. This involved changes to the original pipeline, outside of the cache module added, where new signals and logic were required to select between different memories. There was also the addition of the cache module which involved memory modules with BRAMs for the data and tag memories, a tag check, and an FSM to handle misses. A new main memory interface was added at the end of the module for the cache to communicate with main memory.

The design and implementation was evaluated in chapter 6. First, I performed analysis of the resource utilisation, timing paths and power usage using Vivado's analysis tools. Compared to the original core, I found that they had the same critical path, meaning that the data cache had not slowed down the core. I also found that the core had minimal increases in resource and power usage. This makes the design an overall success as it has retained the speed of the original core, while significantly increasing it's viability to complete larger tasks with minimal resource and power usage increase.

I also analysed the specific cycle timings of events that happen during a cache miss, and found that the design could be made more efficient by removing unnecessary cycles. Lastly, I ran a benchmark program and used it to analyse cache performance metrics, such as hit rate, miss rate and AMAT, and found that the cache has a high hit rate and good AMAT, despite its simple design.

7.2 Limitations and Future Work

The work has several limitations, which presents opportunities for future iterations. I wasn't able to provide any concrete designs other than the one that was implemented, given more time I would've liked to look into adding an L1 instruction cache, like IOb-SoC [16], as well as made the data cache design more configurable. Making the design more configurable would allow for cache performance factors to be altered quickly, and hence the design could be more easily optimised for specific applications. These are the next steps for future designs, after these an L2 unified cache or L2 split caches could be added, along with other SoC infrastructure components.

While the design was verified functionally, and evaluated well in terms of Vivado analysis and cycle timings, the benchmark analysis was rather limited. Running my own benchmark did provide some insight, but in future it would be ideal to run an industry standard benchmark such as Dhrystone, or better yet, CoreMark. It would also be ideal to run the benchmark while varying the cache performance factors, which feeds back into the importance of making the design configurable, as it would make this task significantly easier.

7.3 Challenges Encountered

The initial challenge for the project was understanding the original core, as this was key to creating the design and then implementing it. Fortunately, I had worked with the core before, but only in a small capacity, so much of the design was still a mystery to me. A good amount of time went into this in the first weeks of the project. There was also the task of working with Verilog, an HDL language. I had worked with this before as well, but also in a small capacity and so it was a large task to get to grips with Verilog and then understand the code for the original design as well.

Other challenges encountered included learning how to use different Vivado tools effectively, such as the waveform viewer, as well as the RISC-V GNU toolchain for verification and evaluation. I've written assembly before, and the RISC-V assembly files I used were fairly straightforward, but I had to learn about how to compile them properly, which I had never done before. Unfortunately, my lack of knowledge in how to properly compile programs hindered evaluation, as I was unable to get an industry standard benchmark running.

7.4 Lessons Learned

On a project level, I learned important practises when coding in Verilog such as double checking signal widths, as this saves a lot of unnecessary debugging time. I also learned key skills about managing a project of this size. At a higher level, I've learned to better appreciate the importance and difficulty of computer architecture and systems design, as it ultimately lays the foundation of how computers function and how powerful they can be. On top of this, I've learned a lot about the importance of having a simple, open-source, but very powerful ISA like RISC-V to forward the industry.

Bibliography

[1] Samuel O Aletan. An overview of RISC architecture. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, pages 11–20, 1992.

[2] Michael Gautschi Andreas Traber. PULPino: Datasheet. https://github.com/pulp-platform/pulpino/blob/master/doc/datasheet/datasheet.pdf, 2017. Accessed: April 11, 2023.

[3] K. Asanovic. The Case for Open Instruction Sets. https://www.youtube.com/watch?v=Qq1nMNVCRg8, 2016.

[4] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[5] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[6] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[7] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SoC revolution*. Springer, 1999.

[8] chipyard. Rocket Chip. https://chipyard.readthedocs.io/en/stable/Generators/Rocket-Chip.html, 2023. Accessed: 12th April, 2023.

[9] chipyard. Rocket Core. https://chipyard.readthedocs.io/en/stable/Generators/Rocket.html, 2023. Accessed: 12th April, 2023.

[10] Claire Wolf, et al. PicoRV32 Core. https://github.com/YosysHQ/picorv32, 2022. Accessed: April 11, 2023.

[11] eeNewsEurope. Europe steps up as RISC-V ships 10bn cores, 2015. Accessed: April 11, 2023.

[12] TUL Embedded. PYNQ-Z2 Product Page. https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html, 2021. Accessed: October, 2022.

[13] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[14] IObundle. IObundle. https://www.iobundle.com/. Accessed: April 11, 2023.

[15] Tariq Jamil. Risc versus cisc. *IEEE Potentials*, 14(3):13–16, 1995.

[16] Jose T de Sousa, et al. IOb-SoC. https://github.com/IObundle/iob-soc. Accessed: April 11, 2023.

[17] Nihar R Mahapatra and Balakrishna V Venkatrao. The processor-memory bottleneck: problems and solutions. *XRDS*, 5(3es):2–es, 1999.

[18] Grant Martin and Henry Chang. System-on-Chip design. In *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No. 01TH8549)*, pages 12–17. IEEE, 2001.

[19] Michael Gielda, et al. RISC-V Cores List. https://github.com/riscvarchive/riscv-cores-list, 2021. Accessed: April 11, 2023.

[20] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[21] David A Patterson and David R Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[22] PULP Platform. PULP Platform. https://www.pulp-platform.org/. Accessed: April 11, 2023.

[23] SiFive. Sifive. https://www.sifive.com/risc-v-core-ip, 2023. Accessed: April 11, 2023.

[24] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[25] The PULP Team. PULPissimo: Datasheet. https://github.com/pulp-platform/pulpissimo/blob/master/doc/datasheet/datasheet.pdf, 2021. Accessed: April 11, 2023.

[26] Nigel Topham. RV32IM Core.

[27] Nigel Topham. Computer Architecture and Design. University of Edinburgh, 2021.

[28] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014.

[29] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[30] Marilyn Wolf. *Computers as components: principles of embedded computing system design*. Elsevier, 2012.

[31] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

Appendix A

RISC-V Assembly Files

```
    .file   "main.c"
    .option nopic
    .section    .text.startup,"ax",@progbits
    .align  2
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-400
    mv  a5,sp
    mv  a3,a5
    li  a4,0
  sub a6,x0,a4
    li  a2,100
.L2:
    sw  a6,0(a3)
    sb  a4,2(a3)
  lw  a1,0(a3)
    addi    a4,a4,1
    addi    a3,a3,4
    bne a4,a2,.L2
    addi    a3,a5,400
    li  a0,0
.L3:
    lw  a4,0(a5)
    addi    a5,a5,4
    add a0,a0,a4
    bne a5,a3,.L3
    lui a5,%hi(x)
    sw  a0,%lo(x)(a5)
    addi    sp,sp,400
    jr  ra
    .size   main, .-main
    .comm   x,4,4
```

```
    .ident  "GCC: (GNU) 7.2.0"
```

Listing A.1: RISC-V assembly file used for general debugging

```
    .file   "main.c"
    .option nopic
    .section    .text.startup,"ax",@progbits
    .align  2
    .globl  main
    .type   main, @function
main: # Sets up stack pointer and registers for initialisation
    of main memory and cache
    addi    sp,sp,-2044
    mv a0, sp
    li a1, 0
    li a2, 512
.initialise: # Sets up main memory and cache initially (covers
    non-dirty store eviction)
    sw  a1, 0(a0)
    lw a3, 0(a0)
    addi a0, a0, 4
    addi a1, a1, 1
    bne a1, a2, .initialise

# Tests dirty and non-dirty evictions for loads and stores
    # dirty store eviction
    mv a0, sp
    sw a1, 0(a0)

    # dirty load eviction
    addi a0, a0, 32
    lw a1, 0(a0)

    # non-dirty load eviction
    addi a0, a0, 1024
    lw a1, 0(a0)

# Tests stq cache retire priorities
    mv a0, sp
    li a2, 8192
    li a1, 100
    li a3, 200

    # cache load in EXE (STQ should wait)
    sw a1, 0(a0)
    addi a1, a1, 1
    lw a4, 0(a0)

    # cache store in EXE
    sw a1, 0(a0)
```

```
    addi a1, a1, 1
    sw a3, 0(a0)

    # dccm load in EXE
    sw a1, 0(a0)
    addi a1, a1, 1
    lw a4, 0(a2)

    # dccm store in EXE
    sw a1, 0(a0)
    addi a1, a1, 1
    sw a3, 0(a2)

# Tests stq dccm retire priorities
    # cache load in EXE
    sw a1, 0(a2)
    addi a1, a1, 1
    lw a4, 0(a0)

    # cache store in EXE
    sw a1, 0(a2)
    addi a1, a1, 1
    sw a3, 0(a0)

    # dccm load in EXE (STQ should wait)
    sw a1, 0(a2)
    addi a1, a1, 1
    lw a4, 0(a2)

    # dccm store in EXE
    sw a1, 0(a2)
    addi a1, a1, 1
    sw a3, 0(a2)

# Tests DCCM store by filling half of it
    li a0, 8192
    li a2, 10240
    li a1, 0
.dccm_store:
    sw a1, 0(a0)
    addi a0, a0, 4
    addi a1, a1, 256
    bne a0, a2, .dccm_store

# Tests DCCM load by loading from the half that were just
   stored
    li a0, 8192
    li a2, 10240
.dccm_load:
```

```
    lw a1, 0(a0)
    addi a0, a0, 4
    bne a0, a2, .dccm_load

    addi    sp,sp,1024
    addi    sp,sp,1024
    jr  ra
    .size   main, .-main
    .comm   x,4,4
    .ident  "GCC: (GNU) 7.2.0"
```

Listing A.2: RISC-V assembly file used for functional verification

```
    .file   "main.c"
    .option nopic
    .section    .text.startup,"ax",@progbits
    .align  2
    .globl  main
    .type   main, @function
main: # Sets up stack pointer and registers for initialisation
    of main memory and cache
    lui a7, %hi(-16380)
    addi a7, x1, %lo(-16380)
    sub sp, sp, a7
    mv a0, sp
    li a1, 0
    li a2, 4096
    li a5, 0
.initialise: # Sets up main memory and cache initially (covers
    non-dirty store eviction)
    sw  a1, 0(a0)
    addi a5, a5, 1
    lw a3, 0(a0)
    addi a0, a0, 4
    addi a1, a1, 1
    bne a1, a2, .initialise

# Random memory access section
    mv a0, sp
    li a1, 0
    li a2, 4000
    li a6, 16380 # 16KB range

    # Load random seed value into t0 and t1 register
    lui t0, %hi(1519139625)
    addi t0, x1, %lo(1519139625)
    lui t1, %hi(2641621294)
    addi t1, x1, %lo(2641621294)

.random_access:
    # Generate a random offset within 16KB range
    xor a4, t0, a1
    and a4, a4, a6
    add a4, a4, a0

    # Perform load access
    lw a3, 0(a4)

    # Generate a random offset within 16KB range
    xor a4, t1, a1
    and a4, a4, a6
    add a4, a4, a0
```

```
    # Perform store access
    sw a3, 0(a4)

    # Loop update
    addi a1, a1, 1
    bne a1, a2, .random_access

    # Return
    lui a7, %hi(16380)
    addi a7, x1, %lo(16380)
    add sp,sp,a7
    jr  ra
    .size   main, .-main
    .comm   x,4,4
    .ident  "GCC: (GNU) 7.2.0"
```

Listing A.3: RISC-V assembly file used as a benchmark for evaluating cache performance metrics

Appendix B

Vivado Analysis

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	Block RAM Tile (140)	Bonded IOB (125)	OLOGIC (125)	BUFGCTRL (32)	MMCME2_ADV (4)
N top	5.22%	2.76%	0.97%	0.54%	8.69%	5.22%	36.79%	21.60%	6.40%	18.75%	50.00%
u_cpu (cpu)	4.59%	2.49%	0.94%	0.54%	7.79%	4.59%	2.50%	0.00%	0.00%	0.00%	0.00%
u_control_unit (control_unit)	0.33%	0.19%	0.05%	0.00%	0.65%	0.33%	0.00%	0.00%	0.00%	0.00%	0.00%
u_exec_unit (exec_unit)	3.21%	1.63%	0.64%	0.48%	5.89%	3.21%	1.43%	0.00%	0.00%	0.00%	0.00%
u_alu (alu)	0.71%	0.00%	0.12%	0.00%	0.92%	0.71%	0.00%	0.00%	0.00%	0.00%	0.00%
u_bypass_or_stall (bypass_or_stal	0.27%	0.00%	0.00%	0.00%	0.43%	0.27%	0.00%	0.00%	0.00%	0.00%	0.00%
u_dccm_ram (dccm_ram)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.43%	0.00%	0.00%	0.00%	0.00%
u_decoder (decoder)	0.10%	0.00%	0.00%	0.00%	0.25%	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%
u_divider (divider)	0.05%	0.04%	0.00%	0.00%	0.05%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%
u_regfile (regfile)	1.28%	0.93%	0.48%	0.48%	3.31%	1.28%	0.00%	0.00%	0.00%	0.00%	0.00%
u_store_queue (store_queue)	0.17%	0.09%	0.00%	0.00%	0.35%	0.17%	0.00%	0.00%	0.00%	0.00%	0.00%
u_fetch_unit (fetch_unit)	1.05%	0.67%	0.26%	0.06%	1.85%	1.05%	1.07%	0.00%	0.00%	0.00%	0.00%
u_cpu_clock_gen (cpu_clock_gen)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.38%	25.00%
u_dvi_display (dvi_display)	0.57%	0.24%	0.00%	0.00%	0.86%	0.57%	34.29%	0.00%	6.40%	0.00%	0.00%
u_resync (resync)	<0.01%	0.01%	0.00%	0.00%	0.05%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%
u_ssd_driver (ssd_driver)	0.05%	0.02%	0.03%	0.00%	0.09%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%
u_video_clock_gen (video_clock_gen)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.38%	25.00%

Figure B.1: Resource utilisation of the original core

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	Block RAM Tile (140)	Bonded IOB (125)	OLOGIC (125)	BUFGCTRL (32)	MMCME2_ADV (4)
N top	5.61%	2.94%	0.99%	0.56%	8.93%	5.61%	44.29%	21.60%	6.40%	18.75%	50.00%
u_cpu (cpu)	4.98%	2.67%	0.96%	0.56%	7.98%	4.98%	4.29%	0.00%	0.00%	0.00%	0.00%
u_control_unit (control_unit)	0.33%	0.19%	0.05%	0.00%	0.73%	0.33%	0.00%	0.00%	0.00%	0.00%	0.00%
u_exec_unit (exec_unit)	3.60%	1.81%	0.66%	0.50%	5.91%	3.60%	3.21%	0.00%	0.00%	0.00%	0.00%
u_alu (alu)	0.71%	0.00%	0.12%	0.00%	0.89%	0.71%	0.00%	0.00%	0.00%	0.00%	0.00%
u_bypass_or_stall (bypass_or_stal	0.27%	0.00%	0.00%	0.00%	0.40%	0.27%	0.00%	0.00%	0.00%	0.00%	0.00%
u_data_cache (data_cache)	0.23%	0.11%	0.02%	0.02%	0.44%	0.23%	1.79%	0.00%	0.00%	0.00%	0.00%
u_data_mem (data_mem)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.43%	0.00%	0.00%	0.00%	0.00%
u_tag_mem (tag_mem)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.36%	0.00%	0.00%	0.00%	0.00%
u_dccm_ram (dccm_ram)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.43%	0.00%	0.00%	0.00%	0.00%
u_decoder (decoder)	0.10%	0.00%	0.00%	0.00%	0.21%	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%
u_divider (divider)	0.05%	0.04%	0.00%	0.00%	0.05%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%
u_regfile (regfile)	1.28%	0.93%	0.48%	0.48%	2.73%	1.28%	0.00%	0.00%	0.00%	0.00%	0.00%
u_store_queue (store_queue)	0.19%	0.13%	0.00%	0.00%	0.42%	0.19%	0.00%	0.00%	0.00%	0.00%	0.00%
u_fetch_unit (fetch_unit)	1.05%	0.67%	0.26%	0.06%	1.78%	1.05%	1.07%	0.00%	0.00%	0.00%	0.00%
u_cpu_clock_gen (cpu_clock_gen)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.38%	25.00%
u_dvi_display (dvi_display)	0.57%	0.24%	0.00%	0.00%	0.93%	0.57%	34.29%	0.00%	6.40%	0.00%	0.00%
u_main_mem (main_mem)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	5.71%	0.00%	0.00%	0.00%	0.00%
u_resync (resync)	<0.01%	0.01%	0.00%	0.00%	0.06%	<0.01%	0.00%	0.00%	0.00%	0.00%	0.00%
u_ssd_driver (ssd_driver)	0.05%	0.02%	0.03%	0.00%	0.11%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%
u_video_clock_gen (video_clock_gen)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.38%	25.00%

Figure B.2: Resource utilisation of the core with data cache

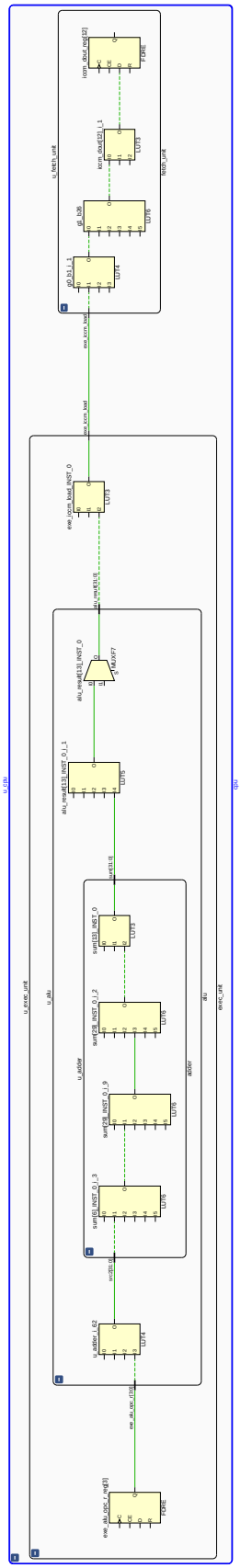


Figure B.3: Critical path of the original core

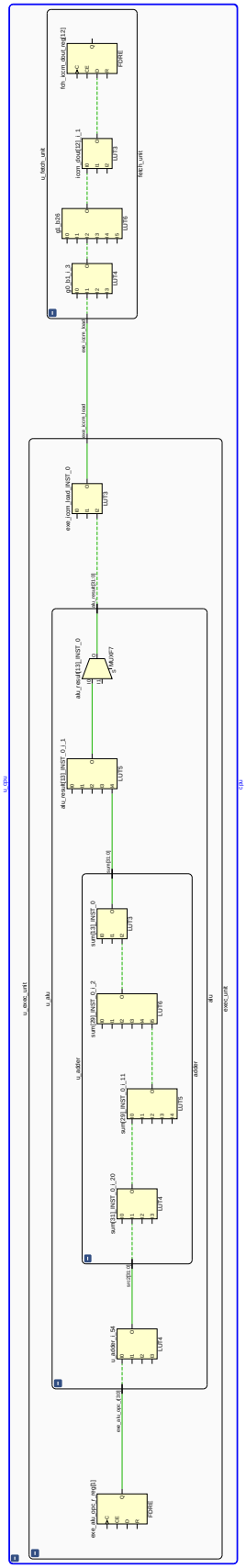


Figure B.4: Critical path of the core with data cache, the same as the original core

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	1.036	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/iccm_dout_reg[12]/D	11.870	2.596	9.274	13.3
Path 2	1.136	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/fch_iccm_dout_reg[26]/D	11.975	2.570	9.405	13.3
Path 3	1.199	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/iccm_dout_reg[26]/D	11.967	2.570	9.397	13.3
Path 4	1.205	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/iccm_dout_reg[0]/D	11.908	2.570	9.338	13.3
Path 5	1.331	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/fch_iccm_dout_reg[12]/D	11.612	2.596	9.016	13.3
Path 6	1.337	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/fch_iccm_dout_reg[18]/D	11.641	2.660	8.981	13.3
Path 7	1.361	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/fch_iccm_dout_reg[0]/D	11.803	2.570	9.233	13.3
Path 8	1.386	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/fch_iccm_dout_reg[29]/D	11.796	2.570	9.226	13.3
Path 9	1.400	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/iccm_dout_reg[29]/D	11.755	2.570	9.185	13.3
Path 10	1.413	11	85	u_cpu/u_exec_unit/exe_alu_opc_r_reg[3]/C	u_cpu/u_fetch_unit/iccm_dout_reg[18]/D	11.565	2.660	8.905	13.3

Figure B.5: 10 worst paths of the original core

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	0.721	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[12]/D	11.841	2.952	8.889	13.3
Path 2	0.724	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[12]/D	11.828	2.952	8.876	13.3
Path 3	0.804	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[31]/D	11.760	3.013	8.747	13.3
Path 4	0.841	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[13]/D	11.687	3.016	8.671	13.3
Path 5	0.911	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[20]/D	11.628	3.016	8.612	13.3
Path 6	0.948	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[29]/D	11.558	2.953	8.605	13.3
Path 7	0.987	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetc...ut_reg[31]/D	11.607	3.013	8.594	13.3
Path 8	1.008	10	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_exec...RARDADDR[5]	11.324	3.017	8.307	13.3
Path 9	1.052	11	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_fetch...out_reg[9]/D	11.508	3.016	8.492	13.3
Path 10	1.057	10	129	u_cpu/u_exec...c_r_reg[1]/C	u_cpu/u_exe...ARDADDR[10]	11.271	3.017	8.254	13.3

Figure B.6: 10 worst paths of the core with data cache

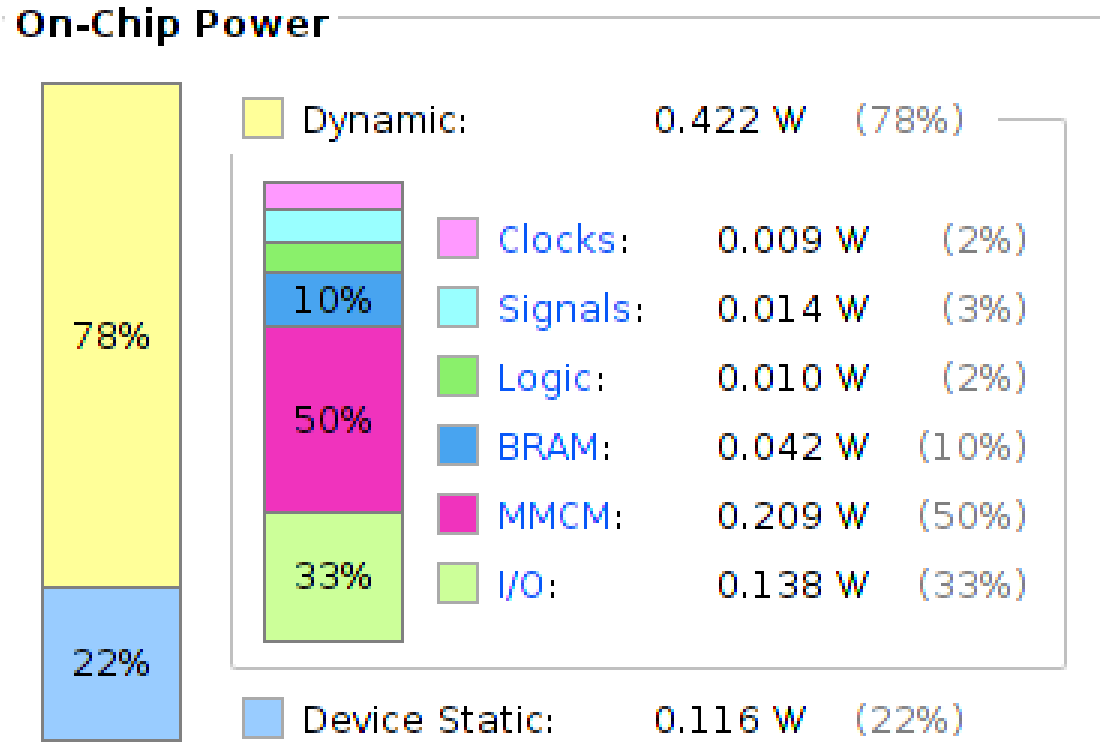


Figure B.7: Summary of power usage in the original core

Utilization	Name
0.422 W (78% of total)	top
0.175 W (32% of total)	u_dvi_display (dvi_display)
0.106 W (20% of total)	u_video_clock_gen (video_clock_gen)
0.104 W (19% of total)	u_cpu_clock_gen (cpu_clock_gen)
0.033 W (6% of total)	u_cpu (cpu)
0.02 W (4% of total)	u_exec_unit (exec_unit)
0.007 W (1% of total)	Leaf Cells (1073)
0.006 W (1% of total)	u_regfile (regfile)
0.004 W (1% of total)	u_alu (alu)
0.001 W (<1% of total)	u_dccm_ram (dccm_ram)
0.001 W (<1% of total)	u_bypass_or_stall (bypass_or_stall)
<0.001 W (<1% of total)	u_decoder (decoder)
<0.001 W (<1% of total)	u_store_queue (store_queue)
<0.001 W (<1% of total)	u_divider (divider)
0.012 W (2% of total)	u_fetch_unit (fetch_unit)
0.001 W (<1% of total)	u_control_unit (control_unit)
0.004 W (1% of total)	Leaf Cells (24)
<0.001 W (<1% of total)	u_ssd_driver (ssd_driver)
<0.001 W (<1% of total)	u_resync (resync)

Figure B.8: Power usage of specific modules in the original core

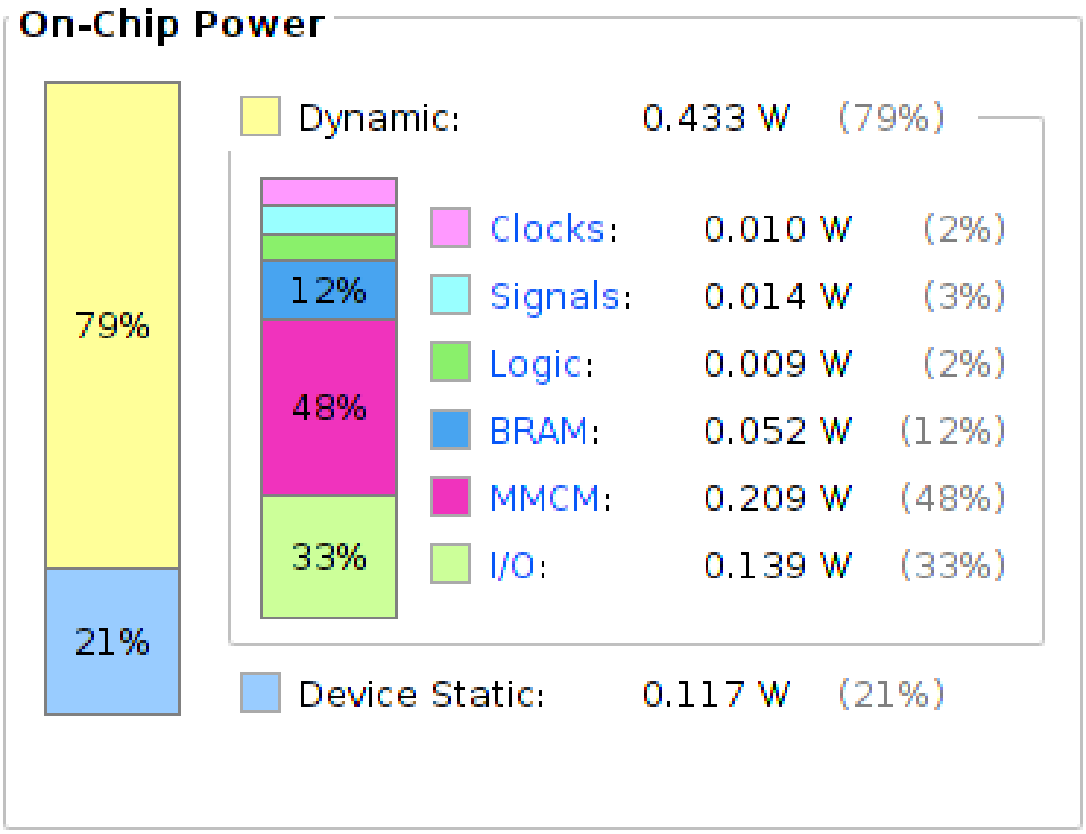


Figure B.9: Summary of power usage in the core with data cache

Utilization	Name
0.433 W (79% of total)	top
0.175 W (32% of total)	u_dvi_display (dvi_display)
0.106 W (19% of total)	u_video_clock_gen (video_clock_gen)
0.104 W (19% of total)	u_cpu_clock_gen (cpu_clock_gen)
0.033 W (6% of total)	u_cpu (cpu)
0.02 W (4% of total)	u_exec_unit (exec_unit)
0.006 W (1% of total)	Leaf Cells (1162)
0.006 W (1% of total)	u_regfile (regfile)
0.003 W (1% of total)	u_data_cache (data_cache)
0.003 W (1% of total)	u_alu (alu)
0.001 W (<1% of total)	u_bypass_or_stall (bypass_or_stall)
0.001 W (<1% of total)	u_store_queue (store_queue)
<0.001 W (<1% of total)	u_decoder (decoder)
<0.001 W (<1% of total)	u_divider (divider)
<0.001 W (<1% of total)	u_dccm_ram (dccm_ram)
0.012 W (2% of total)	u_fetch_unit (fetch_unit)
0.001 W (<1% of total)	u_control_unit (control_unit)
0.009 W (2% of total)	u_main_mem (main_mem)
0.005 W (1% of total)	Leaf Cells (24)
<0.001 W (<1% of total)	u_ssd_driver (ssd_driver)
<0.001 W (<1% of total)	u_resync (resync)

Figure B.10: Power usage of specific modules in the core with data cache