TÉCNICO LISBOA

Development Environment for a RISC-V processor: Cache

João Vieira Rodrigues de Almeida Roque

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering (MEEC)

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Mário Pereira Véstias

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Abstract

Despite the recent advent of open-source hardware, the available open-source caches have low configurability, limited lack of support for single-cycle pipelined memory accesses, and use non-standard hardware interfaces and Hardware Description Languages. In this work IOb-Cache, a high-performance configurable open-source Verilog cache is proposed and developed.

The cache is designed modular and composed of 3 modules. The Cache-Memory module contains the memories and the cache's main controller. The Front-End and Back-End modules isolate the cache design from the processor and memory interfaces, respectively, which enables fast adoption of new processors or memory controllers. Currently, the Front-End module supports the native interface and the Back-End module supports the native and the standard Advanced eXtensible Interface (AXI) interfaces.

The cache can be configured to define the number of ways (k) in set-associative designs (k = 1 selects a direct-mapped design), the number of lines and words per line, the replacement policy, etc. The write-policy is currently fixed to Write-Through Not Allocate policy with an internal buffer, limiting the write accesses to word-sized data. The back-end can be configured to read bursts of multiple words per transfer to take advantage of the available memory bandwidth.

To the best of our knowledge, IOb-Cache is currently the only configurable Verilog cache that supports pipelined Central Processing Unit (CPU) interfaces, and the popular AXI memory bus interface. IOb-Cache is integrated into the IOb-SoC Github repository, which has 16 stars and is being used in 38 projects (forks).

Keywords: Open-source, Cache, Highly Configurable, Pipeline, AXI, Native.

Resumo

Apesar do recente advento do hardware de código aberto, as caches disponiveís em código aberto têm baixa configurabilidade, falta de suporte para ciclo único acessos de memória em pipeline e usam interfaces e Linguagens de Descrição de Hardware não-padrão. Neste trabalho IOb-Cache, uma cache configurável de alto desempenho em Verilog de código aberto é proposta e desenvolvida.

A cache tem design modular e é composto por 3 módulos. O módulo Cache-Memory contém as memórias e o controlador principal. Os módulos Front-End e Back-End isolam o design do cache do processador e da memória interfaces, respectivamente, o que permite a adoção rápida para novos processadores ou controladores de memória. Atualmente o módulo Front-End suporta a interface nativa e o módulo Back-end oferece suporte às interfaces nativa e AXI.

A cache pode ser configurado para definir o número de vias (k) no conjunto projetos associativos (k = 1 seleciona um projeto mapeado direto), o número de linhas e palavras por linha, a política de substituição, etc. A política de escrita é atualmente a política *Write-Through Not Allocate* com um buffer interno, limitando ao tamanho da palavra durante acessos de escrita. O back-end pode ser configurado para ler de várias palavras por transferência para aproveitar a memória disponível largura de banda.

Tanto quanto é do nosso conhecimento, IOb-Cache é a única cache Verilog configurável que suporta interfaces de CPU em pipeline e a popular interface de barramento de memória AXI4. IOb-Cache está integrado no repositório IOb-SoC Github, que tem 16 estrelas, e está sendo usado em 38 projetos (bifurcações).

Palavras-chave: Código aberto, cache, altamente configurável, pipeline, AXI Nativa.

Contents

List of Tables

List of Figures

List of Acronyms

IP Intellectual Property

FPGA Field-Programmable Gate Array

LUT Look-Up Table

HDL Hardware Description Language

CPU Central Processing Unit

CPI Cycles per Instruction

SoC System-on-a-chip

ROM Read-Only Memory

RAM Random-Access Memory

BRAM Block Random-Access Memory

DRAM Dynamic Random-Access Memory

SRAM Static Random-Access Memory

SDRAM Synchronous Dynamic Random-Access Memory

AXI Advanced eXtensible Interface

FSM Finite State Machine

MIPS Mega Instructions per Second

RAW Read-After-Write

AXI4 Advanced eXtensible Interface 4^{th} generation

MSB Most Significant Bit

LRU Least-Recently-Used

MRU Most-Recently-Used

PHY Physical Interface

PLRUm Pseudo-LRU: MRU-based

PLRUm Pseudo-LRU: binary tree-based

BRAM Block Random-Access-Memory

CPI Clocks-per-Instruction

FF Flip-Flop

PLRUm Pseudo-Least-Recently-Used: Most-Recently-Used based

PLRUt Pseudo-Least-Recently-Used: (binary) tree-based

LUTRAM Look-Up-Table Random-Access-Memory

PoC Pile-of-Cores

VHDL VHSIC Hardware Description Language

PLRU Pseudo-Least-Recently-Used

Chapter 1

Introduction

1.1 Motivation

As open-source processors such as the RISC-V architecture become adopted by the industry and compete with commercial solutions such as ARM, the community rushes towards creating the ecosystem for these CPUs to thrive on. These include not only different CPU architectures with different performance, size, and power consumption, but also efficient memory systems, peripherals, and interfaces of all sorts. The software part is even more important as, without compelling user applications and programming tools, no sustainable business can be built with open-source CPUs.

One such key component is a truly configurable cache module, able to support multiple architectural trade-offs. After analysis of the available open-source caches, one finds limitations of the interfaces (no support for de facto standards such as AXI [1]), lack of support for single-cycle pipelined memory accesses, and use of exotic and non-standard Hardware Description Language (HDL), limiting the use of simulation and synthesis tools.

Hence, the need to develop a high-performance configurable cache using a standard bus interface and a standard HDL such as Verilog [2] became evident for IObundle, a company started in 2018 with the aim of developing domain-specific RISC-V Intellectual Property (IP) systems for low power embedded devices. The candidate carried out this work both in the scope of his master's dissertation and as a trainee at IObundle.

1.2 Objectives

The development of a high-performance configurable open-source cache in Verilog with the popular Advanced eXtensible Interface 4^{th} generation (AXI4) interface [1] in the back-end, and with better features than the existing ones is the main objective of this dissertation. To attain this objective the following sub-objectives are pursued:

- Support pipeline architectures. The cache must fulfill 1 request per clock cycle while keeping stalls to a bare minimum. This requires a well designed datapath to correctly implement the cache

operation while guaranteeing that loads and store instructions execute in one cycle. A new request and the response to the previous request must be superimposed, given the 1-cycle latency of the RAM modules that constitute the cache memories.

- Modular design. The cache must be composed of 3 modules: front-end; cache core and back-end. This makes it easy to replace the front-end and back-end interfaces, while keeping the core functionality intact, if needed.

- The back-end must implement the Native and AXI interfaces. The flexibility of the back-end interface is indispensable as it can be connected to higher level caches using the Native interface or to 3rd party memory controllers which are likely to be using an AXI interface.

- The back-end must be configured with a different data-width from that of the front-end (asymmetric memory) in order to take advantage of the available memory bandwidth. Many memory controllers allow wide data buses from designs that need to work at a lower frequency while using a much higher frequency to communicate with the external memory.

1.3 Author's work

The author fully developed all aspects of IOb-Cache [3], with the guidance of his thesis supervisor and advice from his industry colleagues from IObundle Lda. So this work was developed both as an IST student and IObundle intern.

This project was started at the same time as the IObundle's IOb-SoC RISC-V platform [4], in March 2019, and has been a key component of the system which was used by quite a few other dissertations and has now various users worldwide. IOb-SoC kept evolving, so the cache also evolved with it, offering new features to fulfill needs or simply keeping up to date with the the system, which constitutes advanced training for a master student.

As IOb-SoC became the focus of multiple projects, multiple RISC-V processors have been adopted by IObundle. The author helped with developing wrappers for these new processors, to make sure that they seamlessly interfaced with IOb-Cache. After the adoption of the multi-issue superscalar SSRV processor [5] by IObundle, the cache evolved to support a pipelined architecture with single-cycle loads and stores, making it the most advanced open-source cache system we are aware of.

1.4 Thesis Outline

This dissertation has four additional chapters. Each is briefly described below:

- **Chapter 2**: This chapter describes the landscape of open-source caches, their advantages and limitations. It also explains the basic concepts of a processor cache for self-sufficiency reasons. An overview of the different types of memory mappings, policies, and configurations are described.

- **Chapter 3**: This chapter describes the developed cache. Each of its modules is individually and thoroughly described.
- **Chapter 4**: This chapter gives a short description of IOb-SoC, the system where the cache has been integrated, as well, some available configurations for implementing multi-level cache systems.
- **Chapter 5**: This chapter presents the results of tests applied to the developed cache: performance measurements in simulation and Field-Programmable Gate Array (FPGA) emulation, and synthesis results for FPGA. The resources consumed by the cache are analyzed, for different types of configuration. IOb-Cache is also compared to other open-source caches.
- **Chapter 6**: This chapter contains the project's achievements and conclusions, as well as perspectives for future work.

Chapter 2

Background

In this chapter the existing state-of-the-art on open-source cache systems is sketched and the theory of cache systems is briefly introduced. Being the the central problem that this thesis solves the difficulty in finding a comprehensive cache design to accompany the growing trend of open-source hardware, the author set off to investigate the existing ones, which are described in the first of the two sections of this chapter. The second section provides an overview of cache systems, describing structure and implementation policies.

2.1 Open-Source Caches

In search of HDL open-source cache designs, the most relevant ones are found on the Github platform. The caches on Github are chosen based on their popularity (stars and forks): airin711's Verilog-caches [6]; prasadp4009's 2-way-Set-Associative-Cache-Controller [7]; and PoC.cache, which is part of the Pile-of-Cores (PoC)-Library [8], one of the most popular HDL libraries.

The airin711's Verilog-caches repository houses 3 different set-associative caches: 4-way with Least-Recently-Used (LRU) replacement policy; 8-way with Pseudo-Least-Recently-Used (PLRU) and a run-time configurable 2-to-8-way with PLRU replacement policy. All caches have 4 words per line and only allow configuring the number of lines.

The prasadp4009's Verilog cache repository is a 2-way set-associative cache that uses the LRU replacement policy. Unlike the airin771's caches, it allows configuration of the number of cache lines and words per line, as well as the width of both address and data.

Both caches the airin711's and prasadp4009's caches use write-back write-allocate policy, native memory interface and are unable to either invalidate or flush a cache line. The biggest difference between the two is the fact that airin711's caches require the data memory to be 128-bit wide so that the entire line or memory block can be accessed in a single word transfer. The prasadp4009's cache requires the data memory width to be word-sized, using a counter to receive the memory block or transfer the cache line.

Unfortunately, there is a big issue, which makes these two caches poor choices with more advanced architectures: they need at least 2 clock cycles to process requests, even if the data is already in the cache. These caches implement a Finite State Machine (FSM) that controls all their functions, including the communication with the processor and to the main memory. They only allow requests when their FSM is in the initial state, and have special states for the read and write accesses because of the RAM's 1 cycle read latency upon a cache hit. In the read or write access, if a hit occurs, they acknowledge the request and go back to the initial state. This causes the undesirable 1 clock-cycle latency.

The third cache that was investigated, PoC.cache [8], does allow 1 read-access per clock cycle, which is a big improvement over the others. It is also highly configurable, with various synthesis parameters that characterize the cache dimensions, even allowing 3 types of mapping: direct, set-associative, and full-associative. It uses the LRU replacement policy, an effective but costly policy, especially when compared to others that are cheaper but closely effective. Despite only having a native memory interface, it has access to adapters for other commonly used memories, especially SDRAMs.

The disadvantages of PoC.cache are described in this paragraph. PoC.cache uses a control FSM, which during a read-access, only changes state on a miss. This requires the hit to be checked in the same cycle a request is made. The tag and valid memories are therefore implemented with distributed RAM and registers, respectively. In the presence of a hit, PoC.cache acknowledges the request, but the data is only available in the next clock cycle. In the following clock cycle, when the data is available, it can receive a new request. This allows the cache to operate with the 1 read per clock cycle. Write accesses on the other hand require a change of state in the FSM, resulting in a minimum of 1 write per 2 clock cycles. The cache uses a write-through write-not-allocate policy but does not have a write-through buffer. Instead, it accesses the main memory directly. This means each write-access is dependent on the write-access time of the memory interface controller, which is a big issue given that the write-through policy is expected to generate significant traffic.

Despite being highly configurable, its main memory interface is limited to the size of the cache line. The cache expects to load a line in a single transfer, meaning the memory's data-width needs to be line-sized. This is not a negative point since it maximizes the main memory's bandwidth, but may limit the memory options. One severe limitation is when implementing a multi-level cache, as the higher-level cache needs to have a word-size of the lower-level's line width. The lack of a write-through buffer is also a big limitation since this cache needs to stall during a write-access while the higher-level cache is fulfilling another request.

PoC.cache is written in VHSIC Hardware Description Language (VHDL), unlike the other caches that are written in Verilog. Depending on the synthesis and simulation tools, one language can be advantageous over the other but they are semantically equivalent. Most open-source tools only allow one HDL, generally Verilog, so the entire project needs to use the same language.

Compared to the presently developed IOb-Cache system, PoC.cache also lacks (1) a front-end module to avoid the need to implement an FSM for processor-cache communication; (2) a configurable back-end module that controls the communication with the main memory, freeing the main-controller of

unnecessary delays; (3) a universally adapted memory interface like AXI. All these lacking features, plus the fact it written in VHDL, have motivated lObundle to develop a new open-source cache in Verilog.

2.2 Cache: Basic Overview

With the development of integrated systems, there is a necessity for larger memories and speed. While processors keep getting faster, the increase of capacity in memories results is progressive slower accesses[9, p. 634]. This causes a bottleneck effect[10, p. 4], as the processor performance is dependent on the speed of the memory accesses.

A cache is a smaller memory (compared to the main memory), installed close to the processor with faster access time. Instead of accessing the main memory, the processor accesses the cache most of the time, drastically decreasing the average access time. Usually, the large main memory is implemented with Dynamic Random-Access Memory (DRAM) technology, which is cheaper per byte stored but slower- On the other hand, caches are implemented with Static Random-Access Memory (SRAM) technology, which is more expensive but considerably faster to access.

The cache works under two principles of locality: Temporal Locality and Spatial Locality [9, p.633]. Temporal Locality describes that if a specific memory address is accessed, it is likely it will be accessed again. Spatial Locality states that after a specific memory address is accessed, it is likely that an address close to it will be accessed.

A cache's main structure is composed of lines and words. A cache line represents a memory block from the main memory, which is a sequential data block from a specific location. Each line is composed of words, instructions, or data requested by the processor, which is limited by the size of its registers (a 32-bit processor has 32-bit words). The cache line storing a recently accessed memory block capitalizes on both spatial and temporal localities.

When the cache is accessed, the requested data can be stored in it or not. This access is called a hit or miss access, respectively. There are 3 types of misses: Compulsory, Capacity, and Conflict [10, p. 9].

A compulsory miss happens when a cache-line is accessed for the first time, since the cache was empty or invalidated.

A capacity miss occurs because the cache is smaller than the main memory and is not big enough to store all the necessary data. After filling, the next access will result in a capacity miss, as the data in some line must be discarded to load the requested data from the main memory.

A conflict miss happens when two memory blocks are mapped to the same line, causing the most recently accessed to replace the older one. The replaced line can be required later eventually causing another conflict miss.

A fourth miss type exists for multiprocessing systems: the coherency miss. This miss happens when the main memory block is updated by an external source. If that block is stored in the cache(s), it needs to be invalidated. When requested, the newly updated block is fetched.

Instead of having a fixed main memory access time, a cached system has the following average memory access time[10, p. 6]:

$$\begin{aligned} \text{average\_access\_time} = \text{hit\_ratio} \times \text{average\_cache\_access\_time} + \\ + \text{miss\_ratio} \times (\text{average\_cache\_access\_time} + \text{average\_main\_memory\_access\_time}) . \end{aligned} \quad (2.1)$$

The hit-ratio is the percentage of cache-hits of all accesses, while the miss-ratio the percentage of cache-misses of all accesses. Of course they are complementary: miss-ratio = 1 - hit-ratio.

2.2.1 Mapping

Mapping represents the cache's internal organization. There are 3 used mappings: direct-mapped, set-associative, and fully associative.

Direct-mapped represents the basic organization of a cache. The address is divided into tag, index, and offset. The index addresses the cache lines, while the offset addresses their respective words. Because of the smaller capacity, not all address bits are used to address the cache and are used for validation. These bits are called the Tag and are used to verify if the data present in the cache has the same address as the requested data.

Besides the data memory, two additional memories are used, the Tag memory and Valid memory. The Tag memory stores the tag of that respective block, while the Valid memory indicates that that position contains valid data for the main memory. Initially, all positions are invalid until they are filled with valid content. One may want to invalidate a certain memory position if a third entity alters the contents of that memory position, which happens in multi-processor systems. The requested address' tag is then compared with the Tag memory output. If they match then a cache hit happens; otherwise, a cache miss occurs. In case of a miss, the accessed line needs to be replaced with the block that contains the requested data.

The set-associative mapping is based on the direct-mapped but uses multiple ways to access data blocks in the same set. Each way is in theory an individual direct-mapped cache. This means the same cache-line stores as many tags as the number of ways. This reduces conflict-misses. During a line-replacement, it uses a replacement policy to select the way to be replaced.

The fully associative mapping is different in that does not need an index to address a line since it uses a single set. During access, every line is verified for a tag match. This means all lines will have their tag compared and validated. This totally removes conflict-misses. Since all tags are required to be compared, cascaded comparators need to produce a hit, resulting in an expensive hardware structure seldom used in practice. Like the set-associative mapping, it uses a replacement policy to select the line to be replaced.

The most common replacement policy is the LRU, which keeps track of how recently each way has been accessed, replacing the least recent one. Some effective but less costly (requiring less information)

replacement policies can be used, such as the Pseudo-LRU policies. Of these, the most common are the Most-Recently-Used (MRU)-based and tree-based Pseudo-LRU policies.

The MRU-based PLRU keeps track of the accessed ways. When all of them have been accessed, all but the most recent way are reset to non-accessed. The lowest indexed non-accessed way is the one elected to be replaced.

The tree-based PLRU uses a binary tree to keep track of the less recently used way. Starting from the root node, each node points to the next level node, creating a path towards a leaf that represents the least recently accessed way. The nodes' pointers are updated (or not) when access happens, which may change or not the path to point to a more recently used way. Each node uses 1 bit to point to two halves of a subset of ways, which may need to be toggled upon an update.

2.2.2 Write-Policy

During reads, if the data is not available in the cache, it is fetched from the main memory. Writes on the other hand have 2 different policies called Write-Through and Write-Back.

With the Write-Through policy, the data is written to both the cache and the main memory. This policy is normally associated with the Write-Not-Allocate co-policy, meaning that, if the data is not available in the cache, it is simply not stored.

With the Write-Back policy, the main-memory is only written any data when the cache-line is replaced (or removed). This policy normally associated with the Write-Allocate co-policy, meaning that if the address to write is not present in the cache, the respective memory block is first fetched from the main memory to the cache before writing the new data onto it.

Since the write-through updates the memory for every write-access, it generates significant traffic to the main memory [10, p. 5], which is the main disadvantage of this method.

2.2.3 Cache-types

To improve performance, it is not enough simply to increase the cache size. A higher-level organization of the cache can achieve this goal without adding more on-chip memory. This organization can either exploit the type of accessed data, instructions or data, or expand the memory hierarchy with multiple levels of cache blocks.

Dividing caches into instruction or data caches enables parallel access to instruction and data words and parallel exploitation of the localities that exist in instruction and date address sequences. This greatly reduces unnecessary conflict-misses.

In a multi-level cache system, the idea is to reduce the miss penalty, that is, the cost in time of a cache miss. Accessing the external memory upon a cache miss represents the highest penalty. Placing a second cache in between the first-level cache and the main memory attenuates the miss penalty, as the time cost of accessing the second cache should be much lower than that of accessing the main memory. Normally the size of the cache increases with the level. A typical configuration used in this work is to use two level-1 (L1) caches for instructions and data and a larger level-2 cache of unified

instructions and data. Different replacement and write-policies can be used on different levels, which can be optimized to greatly improve performance.

Chapter 3

IOb-Cache

IOb-Cache [3] is a configurable open-source pipelined-memory cache, designed in synthesizable Verilog HDL [2], for System-on-a-chip (SoC) implementation.

IOb-Cache is a very configurable IP core: it offers 2 different interfaces for the back-end memory, Native and AXI (4th generation), whose can be different from that of the front end (asymmetric implementation); it can be implemented as Directly mapped or K-Way Set-Associative; there are multiples line replacement policies to choose from, depending on the performance-resources needs. It uses a fixed write-through not-allocate policy,

Performance-wise, it allows 1 request/clock-cycle (pipelined). Each of the following chapters will describe its respective modules, behavior, and implementation.

3.1 IOb-Cache: top-level

The top-level integrates all the IOb-Cache modules and is represented in the Fig. 3.1.

The Front-End connects the cache to a Master (processor). The ports always use the Native Interface, using a valid-ready protocol.

The Back-End connects the cache (master) to the main-memory (slave). Its interface depends on the choice of the top-level module: Native (iob_cache) or AXI (iob_cache_axi).

Cache-Memory is shown in between Front-End and Back-End and contains all the cache's memories and its main-controller.

Cache-Control is an optional module for an L1 cache that allows performing tasks such as invalidating a data cache, requesting its Write-Through Buffer's status, or analyzing its hit/miss performance. If the "CTRL_IO" macro is set, interfaces for invalidating the cache or observe the Write-Through-Buffer's status are implemented without the Cache-Control module, which is useful for cascading higher-level caches.

The many synthesis parameters that are available for IOb-Cache are shown in Table 3.1.

From the configurable parameters, a few derived parameters of interest are computed. They are explained in Table 3.2

Table 3.1: IOb-cache's configurable parameters.

Parameter	Description
FE_ADDR_W	Front-End Address Width: defines how many bytes are accessible in the main memory ($2^{FE\_ADDR\_W}$ bytes).
FE_DATA_W	Front-End Data Width: defines the cache word-size. Needs to be multiple of 8 (byte). Independent of the Back-end's data bus width (BE_DATA_W).
N_WAYS	Number of (Cache) Ways: Direct Mapped(1); Set-Associative Mapped ($>$1). Needs to be power of 2.
LINE_OFF_W	Line Offset Width or Index Width: Defines the number of cache lines. $2^{LINE\_OFF\_W}$ lines.
WORD_OFF_W	Word Offset Width: Number of Words per cache line. $2^{WORD\_OFF\_W}$ words/line.
REP_POLICY	Replacement Policy: LRU (0); PLRUm (1); PLRUt (2). Requires N_WAYS $>$1.
WTBUF_DEPTH_W	Write-Through-Buffer Depth Width: Number of positions in write-through buffer's FIFO, $2^{WTBUF\_DEPTH\_W}$.
BE_ADDR_W	Back-End Address Width: defines the width of the back-end address port, additional bits aren't accessible (access depends on FE_ADDR_W).
BE_DATA_W	Back-End Data Width: Back-End's memory word-size. Needs to be multiple of FE_DATA_W. This can be used to increase bandwidth.
CTRL_CACHE	Implementation of Cache-Control: performance measurement, write-through buffer status and cache invalidate.
CTRL_CNT	Implements counters for performance measurement (requires CTRL_CACHE(1)).
AXI_ID	AXI-Identifier: Sets the value for the "axi_**id" signals for AXI back-end connections.
AXI_ID_W	Defines the Width of AXI-Identifier. "AXI_ID's" value needs to be in this range.

Table 3.2: IOb-cache's derived parameters.

Parameter	Description
FE_NBYTES	Front-End Number of Bytes: The number of bytes in FE_DATA_W. FE_NBYTES = $\frac{FE\_DATA\_W}{8}$
FE_BYTE_W	Front-End Byte Offset Width. FE_BYTE_W = $\log_2$(FE_NBYTES)
NWAY_W	Width of Number of Ways. NWAY_W = $\log_2$(N_WAYS)
BE_NBYTES	Back-End Number of Bytes: The number of bytes in BE_DATA_W. BE_NBYTES = $\frac{BE\_DATA\_W}{8}$
BE_BYTE_W	Back-End Byte Offset Width. BE_BYTE_W = $\log_2$(BE_NBYTES)
LINE2MEM_W	Line-to-Memory Word-Size Ratio Width: the width of the ratio between the size of the cache-line and the back-end's word. LINE2ME_W = $\log_2(\frac{WORD\_OFF\_W * FE\_DATA\_W}{BE\_DATA\_W})$

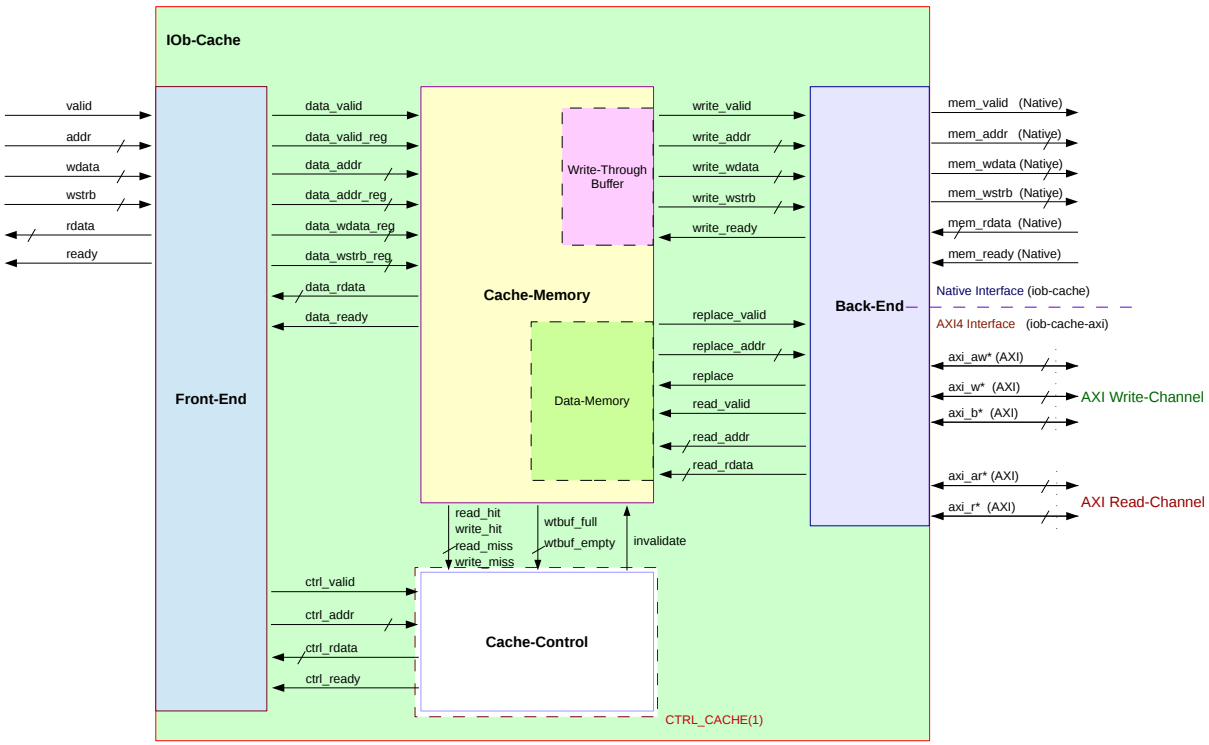

Figure 3.1: IOb-Cache top-level module diagram.

3.2 Front-End

The Front-End module interfaces the processor (master) and cache (slave). In the current design, it splits the processor bus to access the cache memory itself or the Cache-Control module (if present). It also registers some bus signals needed by the cache memory. Its interface is presented in Table 3.3 and Figure 3.2 details the internal structure.

Table 3.3: Front-End ports

Parameter	Width (bit)	Direction	Description
valid	1	input	Validates the request.
addr	FE_ADDR_W +(CTRL_CACHE)	input	Address signal, defines the requested location. When the Cache-Control is implemented , its width will increase by 1, the MSB used to access it.
wdata	FE_DATA_W	input	Write-data, data to be stored in cache (if available (hit)) and in main memory (write-through).
wstrb	FE_NBYTES	input	Write-Strobe, validates each respective byte of write-data. Signal is 0 (all bits) during read-accesses.
rdata	FE_DATA_W	output	Read-Data, the requested data from either Cache-Memory or Cache-Control (only if implemented).
ready	1	output	The acknowledge signal that validates the conclusion of the request.
Cache-Memory			
data_valid	1	output	Validates the Cache-Memory's access.
data_valid_reg	1	output	Registered 'data_valid' signal.
data_addr	FE_ADDR_W -FE_BYTE_W	output	Main-memory address.
data_addr_reg	FE_ADDR_W -FE_BYTE_W	output	Registered 'data_addr'.
data_wdata_reg	FE_DATA_W	output	Write-Data registered signal.
data_wstrb_reg	FE_NBYTES	output	Write-Strobe registered signal.
data_rdata	FE_DATA_W	input	Cache-Memory's read-data.
data_ready	1	input	Cache-Memory's ready signal. Validates the conclusion of the access.
Cache-Control (optional)			
ctrl_valid	1	output	Validates the Cache-Control access. Only if Cache-Control is implemented.
ctrl_addr	CTRL_ADDR_W	output	Selects the Cache-Control's task.
ctrl_rdata	FE_DATA_W	input	Cache-Control's requested read data (counters or buffer status).
ctrl_ready	1	input	Validates the conclusion of the Cache-Control's requested task.

The cache requires that during a request (valid), the master's inputs are maintained until the cache signals its conclusion by asserting the ready signal. During the assertion of the ready, a new access can be requested.

The signals required for memory writing and the ready's combinational path are registered. This way, the necessary input data is still available while ready is asserted.

The cache always returns entire words since it is word-aligned. This means the access is word-addressable, so the byte-offset of the CPU address signal (last FE_BYTE_W bits) is not connected to the cache.

In a system with a different CPU interface, only this module requires modification to allow compatibility.

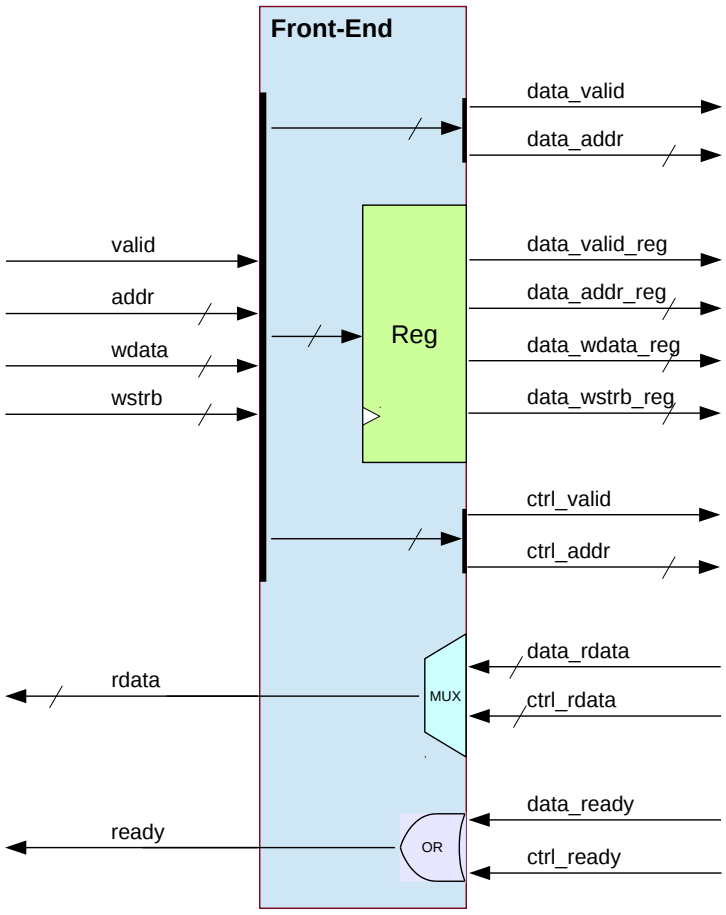

Figure 3.2: Front-End module diagram.

If the optional Cache-Control is implemented, this module also works as a memory-map decoder to select which unit is being accessed, the memory or the control unit.

This mapping is done using the Most Significant Bit (MSB) of the port "addr". When high (1), the Cache-Control module is accessed. This also required some additional logic to select which read data is sent to the master. This logic is a word-sized multiplexer (read-data) and an OR-gate (valid).

3.3 Cache-Memory

Cache-Memory is a module that contains the cache's main controller and memories. The available memories are the Tag memory, the Valid memory, the Data memory, the write-through Buffer, and, if applicable, the Replacement-Policy memory.

Its main controller accesses specific back-end modules using a handshake valid-ready approach. The ready-signal is always asserted excepts after valid is asserted, where it becomes 0 until the request's conclusion.

Depending on the choice of parameters, Table 3.1, the cache's implementation will either be direct-mapped or set-associative based on the number of ways given by N_WAYS.

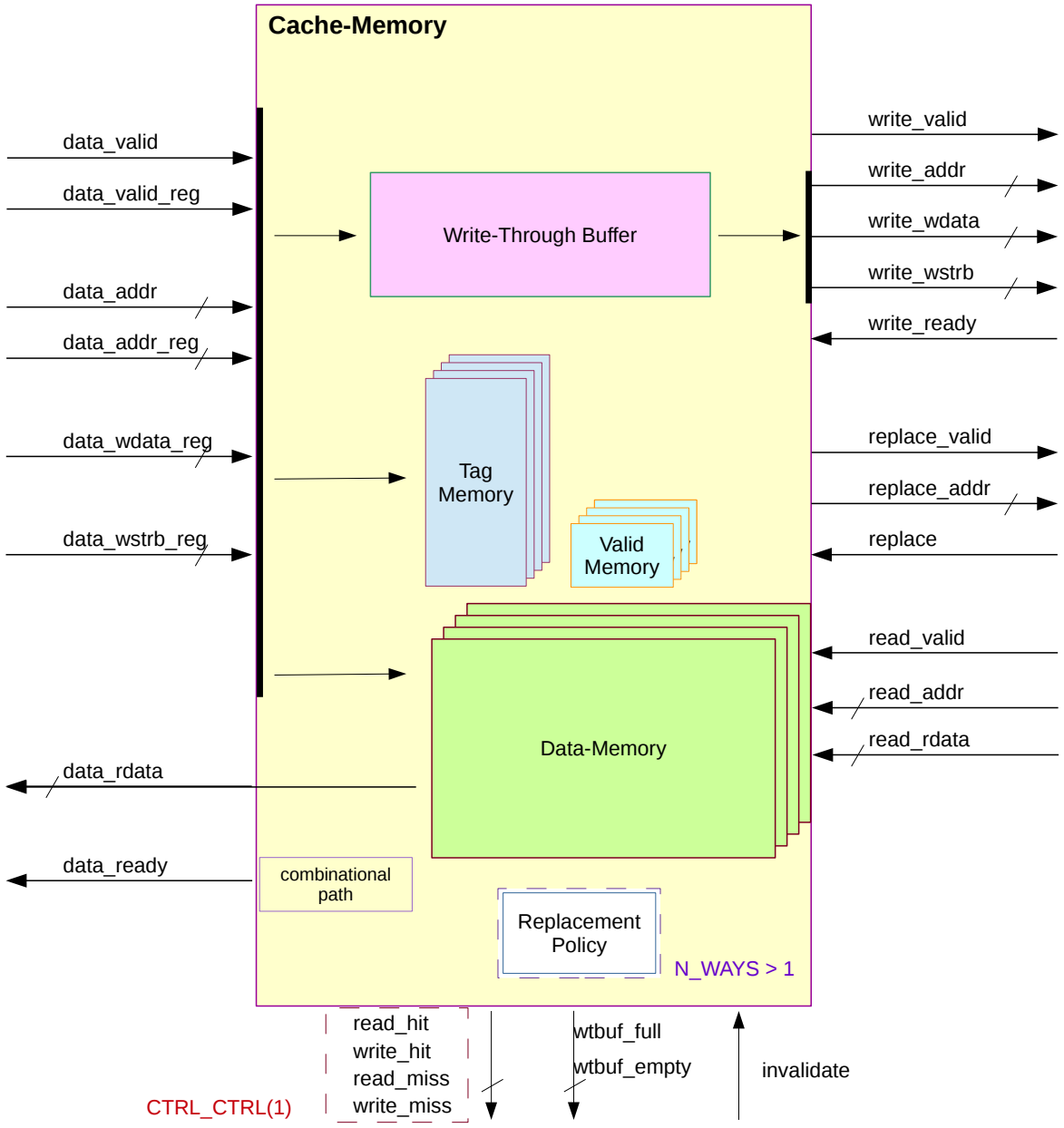

Figure 3.3: Cache-Memory module diagram.

Table 3.4 describes the Cache-Memory's ports. For simplicity, the Front-End signals' prefix, "data_" was removed.

Table 3.5 contains the additional ports used if the module Cache-Control is implemented.

Before the Cache-Memory's behavior and design description is presented, the implementation of each memory will be explained.

Figure 3.4 shows the Tag and Valid memories' implementation, while Figure 3.5 the shows the Data memory.

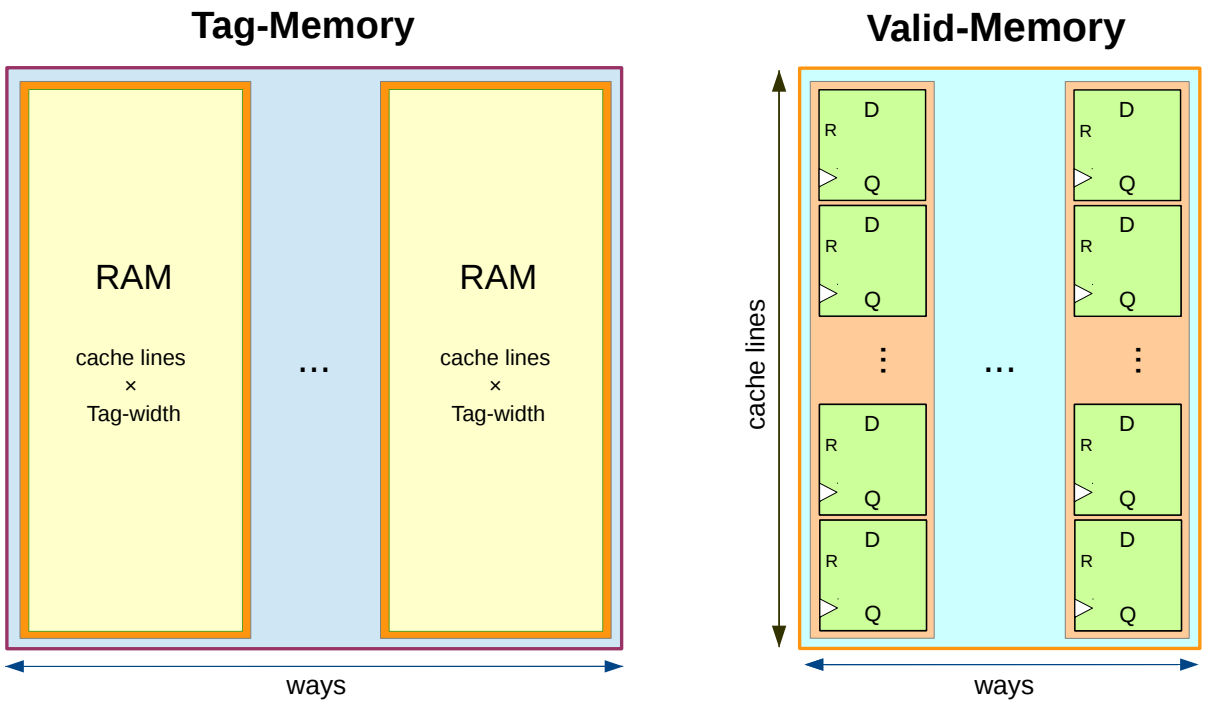

Figure 3.4: Tag and Valid-Memories.

The Tag memory is inferred using RAM. There is one Tag memory per cache way. Each of these has Tag-sized width, and depth equal to the total number of cache lines.

The Valid memory is composed of an array of 1-bit registers (register-file), one for each way. Each array's length equals the number of cache lines. This choice of implementation was a simple design choice to set its contents to 0 during either a system reset or a cache-invalidate.

The Tag memory has a 1 clock-cycle read latency (Random-Access Memory (RAM)), therefore the valid memory's output signal needs to be delayed, either by applying a 1-bit output stage-register or using the registered address "data_addr_reg". The latter was chosen because it requires less logic and has no impact on timing. Both Tag and Valid memories' outputs connect to comparators for producing hit/miss results required for memory accesses.

The Data memory is implemented by one RAM for each way and (word) offset. Each RAM has a width FE_DATA_W (cache word-size) and a depth of $2^{\text{LINE\_OFF\_W}}$ (number of cache lines). Since the write-strobe signal selects which bytes are stored, each RAM requires a write enable for each byte.

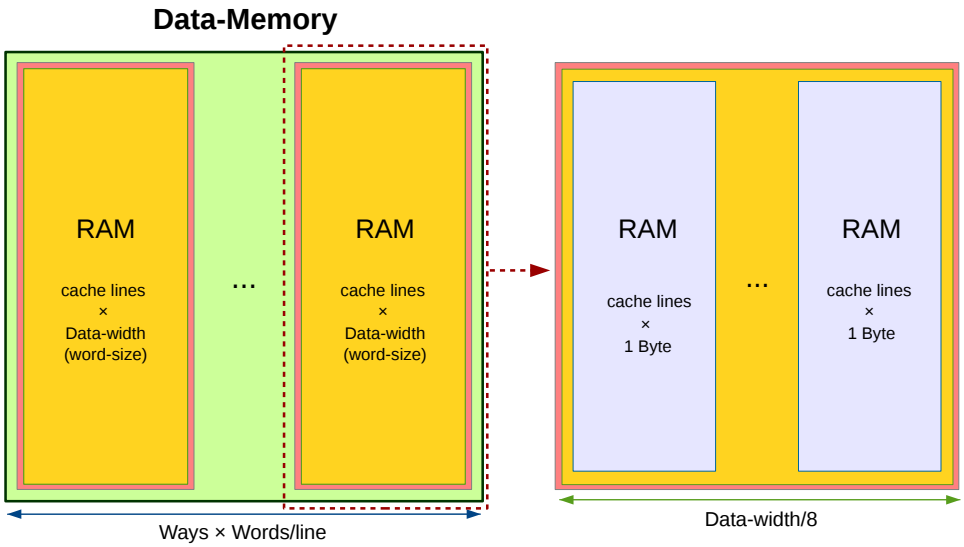

Figure 3.5: Data-Memory.

Some synthesis tools can only infer single-enable RAMs [11, 12], therefore a RAM will be inferred for each byte.

The Write-Though Buffer is implemented using a synchronous FIFO [13]. It requires the data to be available on its output a clock-cycle after being read.

To address the words in the cache's memories, the input address signals are segmented as described in Figure 3.6.

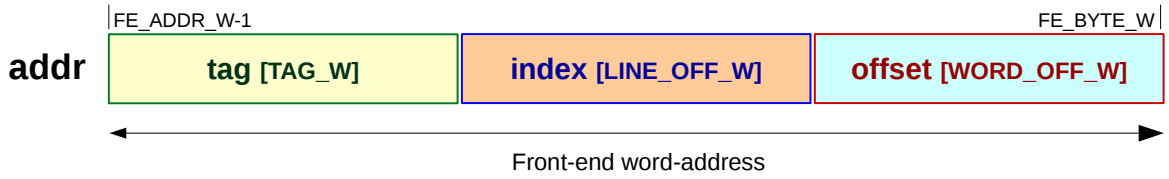

Figure 3.6: Address signal segmentation.

The address (data_addr) is only used for the initial addressing (indexing) of the main memories: Valid, Tag, and Data. On the next clock cycle, the registered address (data_addr_reg) will be checked to see if a "hit" occurred and identifying the word within the cache line.

The hit check uses the signal "way_hit". Each of its bits indicates a hit in the respective way. The hit is the result of a tag match.

If any bit of the "data_wstrb_reg" signal is enabled, it is a write-request, otherwise it is a read-request.

During a read-request, if a hit is produced, the respective word is already available in the Data-Memory's output, so the request can be acknowledged.

The Data memory allows input Data from both the Front and the Back-End. This selection is done using the signal "replace", which indicates if the replacement on a cache line is in action. While "replace" is not asserted, all accesses are from the Front-End. During a read-miss, the signal "replace" is asserted, which will start the Back-End Read-Channel controller, responsible for line replacement.

Both Tag and Valid memories are updated when the “replace_valid” signal is high, forcing a hit in the selected way. This allows the replacement process to act similarly to a regular write hit access, reducing the necessary logic. The replacement can only start if there are not currently write transfers to the main-memory.

The signals ”write_valid” and ”write_ready” constitute a handshaking pair for Cache-Memory to write to the Back-End Write-Channel. The former indicates the Write-Through Buffer is not empty, validating the transfer. The latter indicates that the Back-End Write-Channel is idle and thus enables reading the Write-Through Buffer.

The requirement that the replacement only starts after the write transfer is to avoid coherency issues, i.e. storing outdated data in the cache-line.

Write requests do not depend on the data being available in the cache, since it follows the write-not-allocate policy. Instead, it depends on the available space in the Write-Through Buffer, which stores the address, write-data, and write-strobe array.

During a write-hit, to avoid stalling, the Data memory uses the registered input signals to store the data, so the cache can receive a new request.

If a read follows a write-access, Read-After-Write (RAW) hazards can become an issue. The requested word may not be available at the memory output, since it was written just the cycle before. This word will only be available in the following clock-cycle, therefore the cache needs to stall.

Stalling on every read-request that follows a write-hit access can become costly performance-wise. Hence, to avoid this cost a simple technique has been employed: the cache stalls only if one wants to read from the same way and (word) offset that has been written before. This results in RAW only signaling when the same Data memory’s (byte-wide) RAMs are being accessed.

All the above conditions are implemented in the main controller circuit. The signals data_ready and hit are given combinatorially by Equation 3.1.

$$\begin{aligned} \text{data\_ready} &= (\text{write\_access AND !buffer\_full}) \text{ OR } (\text{read\_access AND hit AND !RAW}) \\ \text{hit} &= \text{OR(way\_hit) AND !replace} \end{aligned} \tag{3.1}$$

A stall occurs if ”data_ready” is 0 after the request, otherwise the request is acknowledged.

Because a new request can be issued in the same cycle as the previous is acknowledged, the registered request signals are used for the tag comparison, memory-writing and word-selection.

Since the line replacement controller uses the way_hit signal, the hit signal is 1 only when signal replace is de-asserted, as this signal already has a delay to compensate for the Data memory’s RAM 1 clock cycle read-latency.

Table 3.4: Cache-Memory ports.

Parameter	Width (bit)	Direction	Description
Front-End ports			
data_valid	1	input	Valid signal, requests the access to Cache-Memory.
data_valid_reg	1	input	Registered valid signal, required for memory writes.
data_addr	FE_ADDR_W -FE_BYTE_W	input	Address signal of the word. Indexes the cache lines.
data_addr_reg	FE_ADDR_W -FE_BYTE_W	input	Registered address signal. Required for memory writes, tag comparison, and Data-Memory's word-selection.
data_wdata_reg	FE_DATA_W	input	Registered write-data signal. Required for Data-Memory and Write-Through Buffer.
data_wstrb_reg	FE_NBYTES	input	Registered write-strobe signal. Write-data's byte-enable.
data_rdata	FE_DATA_W	output	Read-data signal. Requested Data-Memory's word.
data_ready	1	output	Ready signal. Acknowledges the conclusion of the request. Read-data is available (read-access).
Back-End: Write-Channel ports			
write_valid	1	input	Initiates write request to main memory. Write-Through Buffer's empty signal.
write_addr	FE_ADDR_W -FE_BYTE_W	output	Address of the write request, stored in Write-Through Buffer.
write_wdata	FE_DATA_W	output	Write-data, stored in Write-Through Buffer.
write_wstrb	FE_NBYTES	output	Write-strobe (byte-enables), stored in Write-Through Buffer.
write_ready	1	output	Reads Write-Through Buffer, data is available in its output next clock cycle.
Back-End: Read-Channel ports			
replace_valid	1	output	Requests replacement of the cache line. Asserted during a read-miss.
replace_addr	FE_ADDR_W -FE_BYTE_W -LINE2MEM_W	output	Memory block's base address to replacement a cache line.
replace	1	input	Signals that a line replacement is in progress.
read_valid	1	input	Validates the memory block's word. Only during line replacement.
read_addr	LINE2MEM_W	input	Addresses the current position in the cache line.
read_data	BE_DATA_W	input	Main memory's read-data, to be stored in the selected cache line, during the replacement.

Table 3.5: Cache-Memory ports: Cache-Control

Parameter	Width (bit)	Direction	Description
invalidate	1	input	Invalidates entire cache, reseting Valid-Memory, and if applicable, the replacement-policy's memory.
wtbuf_empty	1	output	Write-through buffer's empty signal, asserts when empty.
wtbuf_full	1	output	Write-through buffer's full signal, asserts when full.
write_hit	1	output	Write-hit, asserts when a write-request is available in the cache (hit).
write_miss	1	output	Write-miss, asserts when a write-request is not available in the cache (miss).
read_hit	1	output	Read-hit, asserts when the requested data is available in the cache (hit).
read_miss	1	output	Read-miss, asserts when the requested data is available in the cache (hit).

3.4 Replacement Policy

The line replacement policy in a k-way set-associative cache is implemented by the module shown in Figure 3.7. Different available replacement policies can be selected using the ”REP_POLICY” synthesis parameter. The module has three main components: the Policy Info Memory (PIM), the Policy Info Updater (PIU) datapath, and the Way Select Decoder (WSD). Table 3.6 explains the ports of the module.

The PIM stores information pertaining to the implemented policy. Note that replacement policies are dynamic and use data from the past, so memory is needed. The PIM has as many positions as the number of cache sets, addressed by the *index* part of the main memory address. The width of the PIM depends on the chosen policy. The PIM is implemented using a register-file so that during a system reset or cache invalidation, it can be set to default initial values.

When a cache hit is detected, the information stored in the PIM is updated based on the information previously stored for the respective set and the newly selected way. This function is performed by the PIU. When a cache miss is detected the information for the respective cache set is read from the PIM and analyzed by the WSD in order to choose the way where the cache line will be replaced.

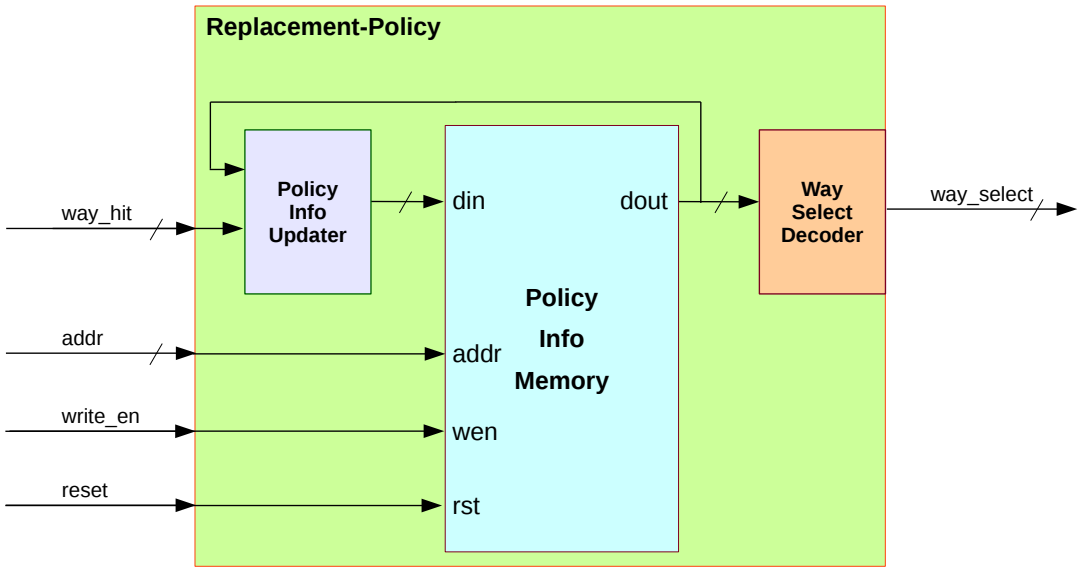

Figure 3.7: Replacement Policy module diagram.

The currently implemented policies are the Least-Recently-Used (LRU) and the Pseudo-Least-Recently-Used (tree and MRU-based). These are explained in the next subsections.

Table 3.6: Replacement policy ports.

Parameter	Width (bit)	Direction	Description
write_en	1	input	Enable signal to update the memory. CacheMemory's data_ready signal.
addr	LINE_OFF_W	input	Address of the respective cache line.
way_hit	N_WAYS	input	The signal that indicates in which way occurred a hit, one-hot format. Used in the encoder.
reset	1	input	Resets the replacement policy memory's data. Used during a reset of the system or a cache invalidate.
way_select	N_WAYS	output	The selected way decoded from the current replacement's key. Used to select a way during a line-replacement. In one-hot format.
way_select_bin	$\log_2$(N_WAYS)	output	The same as way_select but in the binary format. Uses a one-hot-to-binary encoder.

3.4.1 Least-Recently-Used

The Least-Recently-Used policy (LRU) needs to store, for each set, a word that has N_WAYS fields of log2(N_WAYS) bits each. Each field, named ”mru[i]”, represents how recently the way has been used by storing a number between 0 (least recently used) and N_WAYS-1 (most recently used), thus requiring $\log_2$(N_WAYS) bits. In total it requires N_WAYS$\log_2$(N_WAYS) bits per cache set.

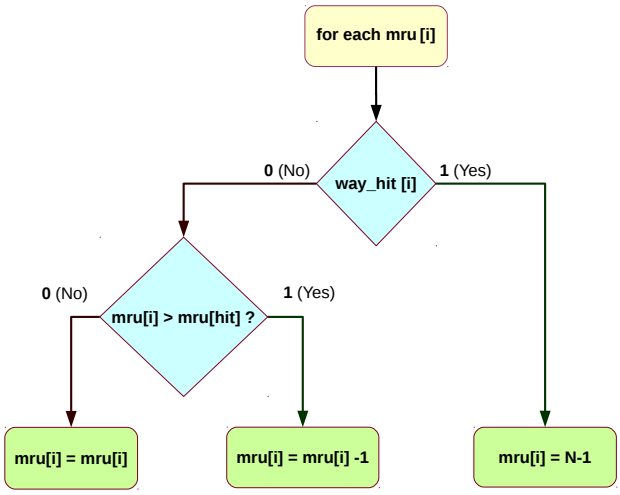

Figure 3.8: LRU Encoder datapath flowchart.

The way each mru[i] is updated is represented in Figure 3.8. Summarizing, when a way is accessed either by being hit or replaced, it becomes the most recently used and is assigned. The other ways with higher mru values than the accessed way get decremented. The ones with lower mru values are unchanged. The selected way for replacement is the one with the lowest ”mru” index. This can be achieved by NORing each index, as implemented in Equation 3.2.

$$\text{way\_select}\,[i] = \text{!OR}(\text{mru}[i])\,. \tag{3.2}$$

3.4.2 Pseudo-Least-Recently-Used: MRU-based

The Pseudo-Least-Recently-Used: Most-Recently-Used based (PLRUm) is simpler than the LRU replacement and needs to store, for each set, a word that has N_WAYS bits only. Each bit mru[i] represents how recently the way has been used, storing a 0 (least recently used) or 1 (most recently used), thus requiring $\log_2$(N_WAYS) bits.

The way each mru[i] is updated is represented in Figure 3.9. Summarizing, when a way is accessed either by being hit or replaced, the respective bit is assigned 1 meaning it has been recently used. When all ways have been recently used, the most recently assigned remains asserted and the others are reset. This is done by simply ORing the way_hit signal and the stored bits, or storing the way_hit signal if all have been recently used. To select a way for replacement, the not recently used way (mru[i]=0) with the lowest index is selected. This can be implemented by the following logic equation, Equation 3.3.

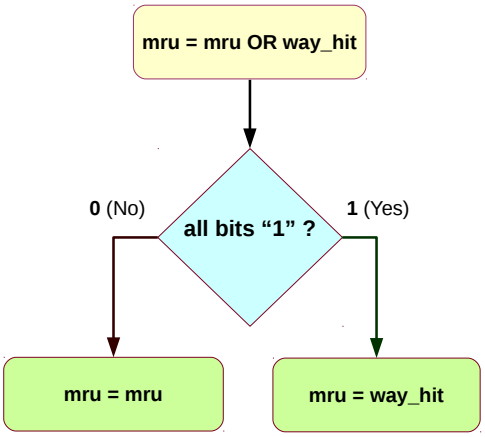

Figure 3.9: PLRUm Updater datapath flowchart.

$$\text{way\_select}\,[i] = !\text{mru}[i]\ \text{AND}\ (\text{AND}(\text{mru}[i\text{-}1{:}0]) \tag{3.3}$$

3.4.3 Pseudo-Least-Recently-Used: binary tree-based

The Pseudo-Least-Recently-Used: (binary) tree-based (PLRUt) needs to store, for each set, a binary tree with $\log_2$(N_WAYS) levels and N_WAYS leaves, each representing a cache way. Each level divides the space to find the way in two, creating a path from the root node to the chosen way, when traversed by the WSD. Each node is represented by a bit b[i] where 0 selects the lower half and 1 selects the upper half of the space. For a 8-way example, the binary tree is represented in Figure 3.10.

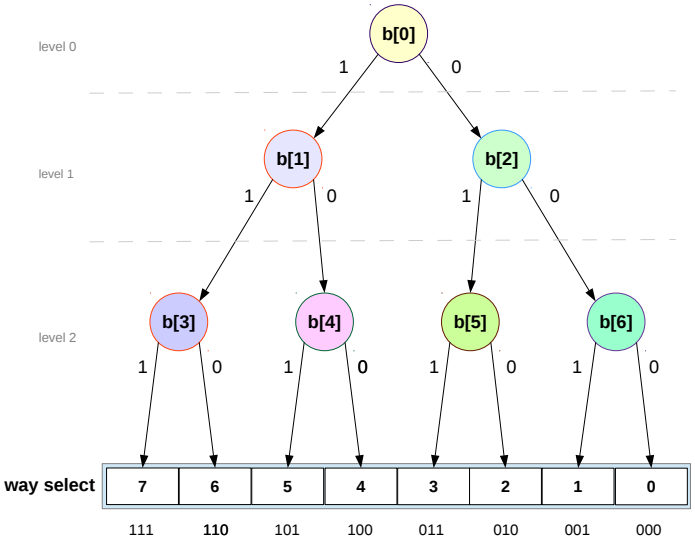

Figure 3.10: PLRU binary tree.

In order to update each node b[i], the first step is to get the slice way_hit[i] from the vector way_hit, relevant for computing b[i]. Figure 3.11 shows how to compute way_hit[i] for the first 3 notes, b[2:0]. After computing slice way_hit[i], the algorithm shown in Figure 3.12 is followed. The process is straightforward.

If the slice is not hit (all its bits are 0), then b[i] remains unchanged. Otherwise, b[i] is set to 0 if the hit happens in the upper part of the slice and to 1 if the hit happens in the lower part.

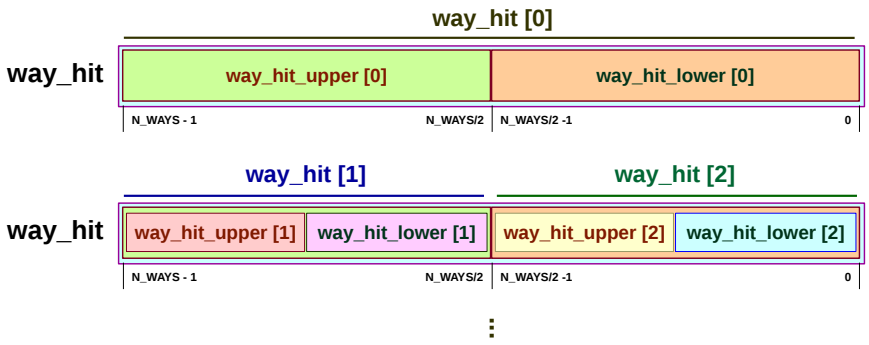

Figure 3.11: Computing way_hit slices.

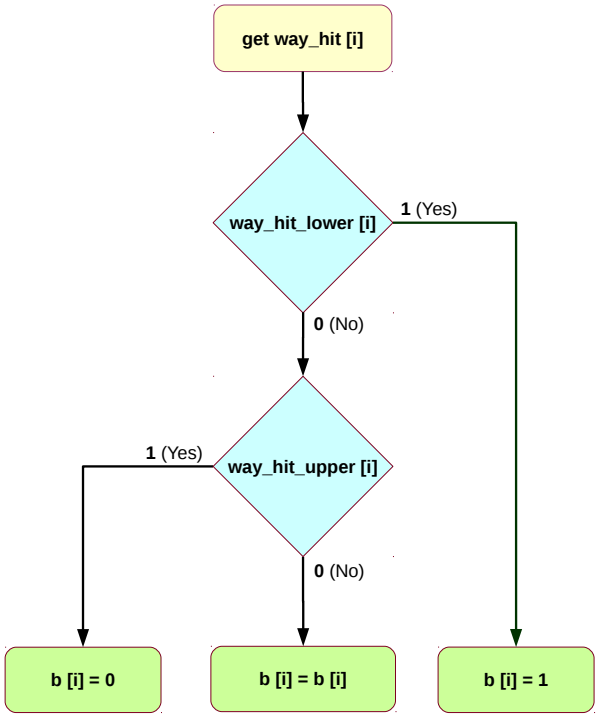

Figure 3.12: PLRU way updater.

To select the way for doing the replacement, the binary tree needs to be decoded. This can be done by iterating from the tree levels, from root to leaves, using the b[i] values to point to the next node until the leaf is reached. As explained before the leaf index is the chosen way.

3.5 Back-End

The Back-End module is the interface between the cache and the main memory. There are currently 2 available main memory interfaces: Native and AXI. The native interface follows a pipelined valid-ready protocol and is shown in Figure 3.13. The AXI interface implements the AXI4 protocol [1, 14] and can be seen in Figure 3.14.

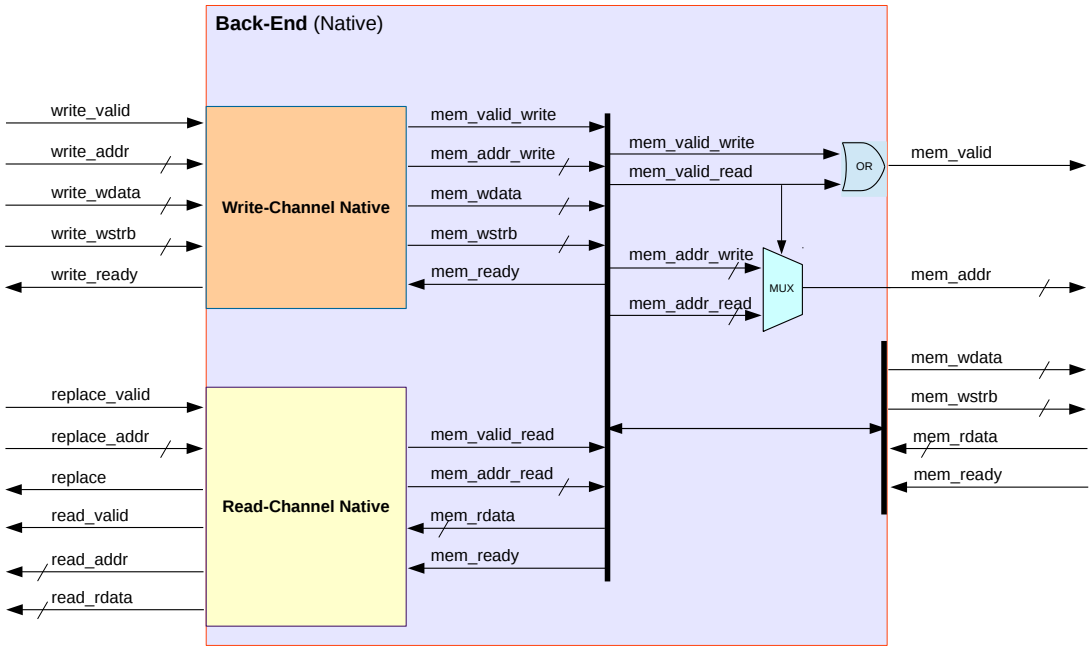

Figure 3.13: Back-End Native module diagram.

Although the AXI interface has independent write and read buses, the native interface only has a single bus available. In the native interface, the difference between a write and read access depends on the write-strobe signal (mem_wstrb) being active or not. This requires additional logic to select which controller accesses the main memory. The AXI interface transfers can be configured using the relevant synthesis parameters given in Table 3.1. There is no risk of conflict between the read and write channels: reading for line replacement can only occur after all pending writes are done.

The Back-End module has two controllers, the Write-Channel controller and the Read-Channel controller. The Write-Channel controller reads data from the Write-Through Buffer and writes data to the main memory while the buffer is not empty. The Read-Channel controller fetches lines from the main memory and writes them to the cache during a cache line replacement.

3.5.1 Write-Channel Controller

The Write-Channel controller ports for both the native and AXI4 interfaces are described in Table 3.7.

Table 3.7: Write-Channel ports.

Parameter	Width (bit)	Direction	Description
Cache Side Native Interface			
write_valid	1	output	Valid data in write-through buffer (not empty).
write_addr	FE_ADDR_W -FE_BYTE_W	input	Word-address of the write data. Buffer's ouput.
write_wdata	FE_DATA_W	input	Write-Data. Buffer's output
write_wstrb		input	Write-strobe. Buffer's output
write_ready		input	Ready, read write through buffer.
Memory Side Native Interface			
mem_valid	1	output	Validates the write transfer to the main-memory.
mem_addr	BE_ADDR_W	output	Address of the write transfer.
mem_wdata	BE_DATA_W	output	Write-Data. Data transfered to memory.
mem_wstrb	BE_NBYTES	output	Validates mem_wdata's Bytes.
mem_ready	1	input	Memory's ready, acknowledges the transfer.
Memory Side AXI Interface			
axi_awvalid	1	output	Validates Write-Address.
axi_awaddr	BE_ADDR_W	output	Write-Address, Byte addressable and word aligned.
axi_awready	1	input	Acknowledges the address request.
axi_wvalid	1	output	Validates the data transfer.
axi_wdata	BE_DATA_W	output	Write-Data.
axi_wstrb	BE_NBYTES	output	Validates Write-Data's Bytes.
axi_wready	1	input	Memory's acknowledges the transfer.
axi_bvalid	1	input	Write response valid, validates axi_bresp.
axi_bresp	2	input	Write response result, if the transfer was successful ("00" - OKAY) [1, p. A3-54].
axi_bready	1	output	Write response ready signal, awaits for the response result (handshake).
axi_awlen	8	output	Burst's length, 0 (single) [1, 14, p. A3-48,51]).
axi_awsize	3	output	Bytes per beat, $\log_2$(BE_NBYTES) [1, p. A3-49]
axi_awburst	2	output	Burst type, any option (single transfer) [1, p. A3-49]).
axi_awid	AXI_ID_W	output	AXI identification tag [1, p. A5-81].
axi_awlock	1	output	Atomic access. Normal access 0_b [1, p. A7-100].
axi_awcache	4	output	Type of memory. Default 0011_b [1, p. A4-68].
axi_awprot	3	output	All accesses are normal, 000_b [1, 14, p. A4-75].
axi_awqo	4	output	QoS identifier, unused, 0000_b [1, p. A8-102].

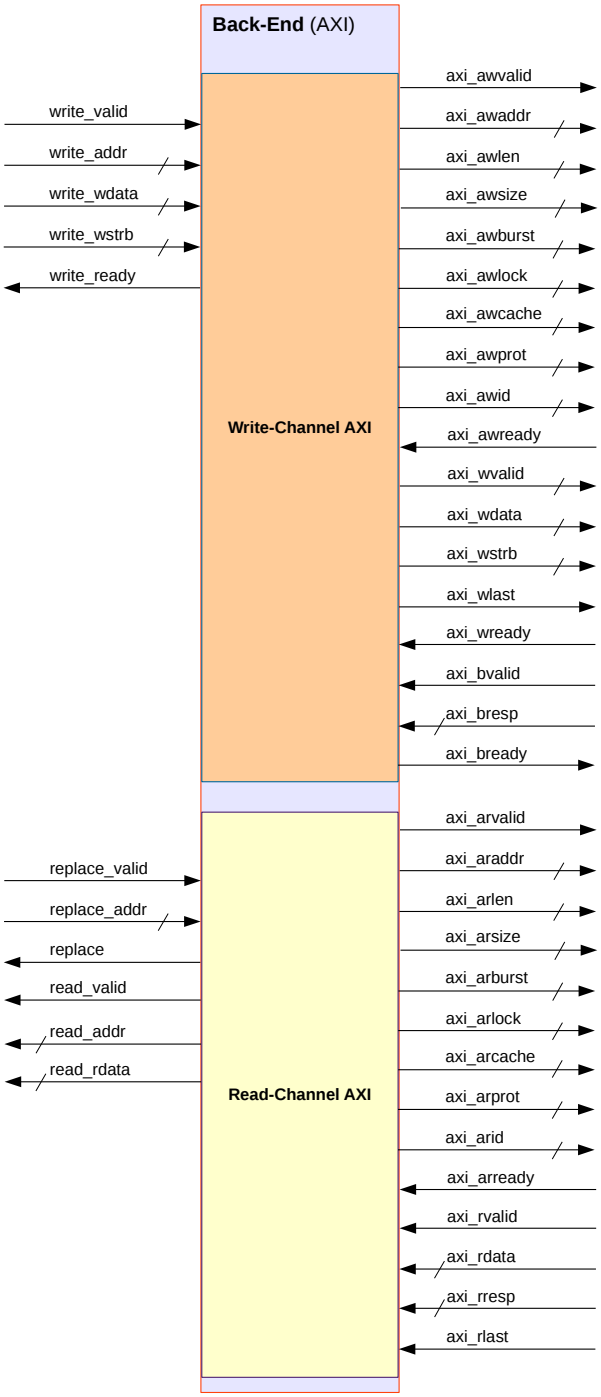

Figure 3.14: Back-End AXI module diagram.

The native interface's controller follows the control flow displayed in Figure 3.15. The controller stays in the initial state while waiting for the write-through buffer to have data. The write-through buffer uses a FIFO, and the FIFO starts the controller when it is not empty. When that happens, signal write_valid asserts, and the FIFO gets read.

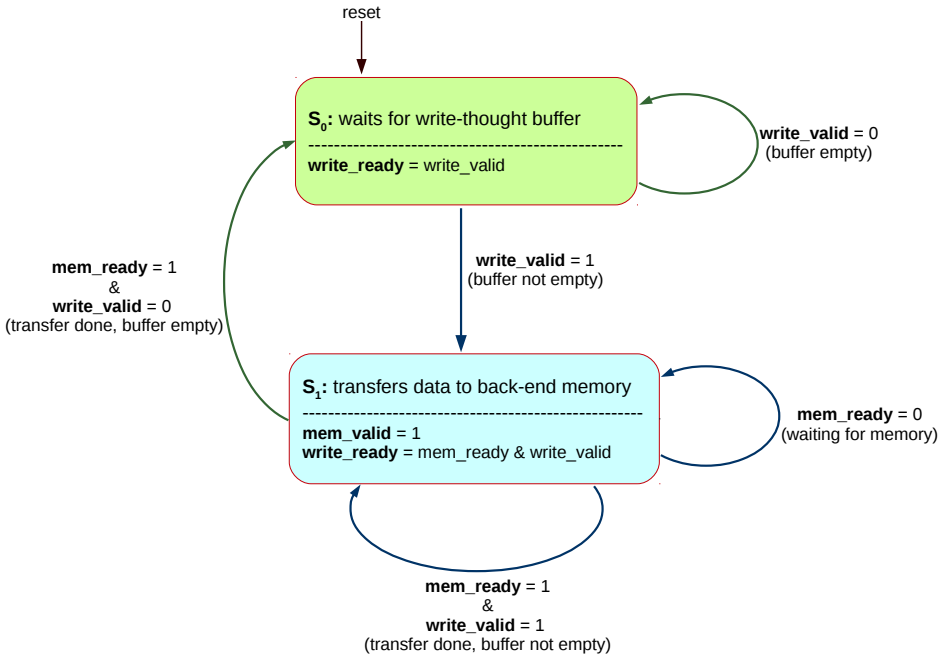

Figure 3.15: Back-End Write-channel Native Control-flow.

In the following clock cycle, the required data is available in FIFO's output and the transfer can occur. After each transfer, the FIFO is checked, and if it is not empty, it is read again so the data can be transferred in the following clock cycle. This keeps happening until there are no more available data in the Write Through Buffer, and the controller goes back to its initial state.

The write-through buffer can only be read after each transfer is completed (mem_ready received). Currently, there is no way to pipeline these transfers, which are limited to 1 word per every 2 clock cycles. While the controller is in the initial state, the memory's write-strobe signal is 0 to not disturb the Read-Channel controller.

The AXI-Interface (Figure 3.16) has similar behavior but follows the AXI4 protocol. The address valid-read handshake needs to happen before any data can be transferred. After the data is transferred, it is checked to see if it was successful through the response channel (B channel): if axi_bresp does not have the OKAY value (an AXI code), then the transfer was unsuccessful and the data is transferred again.

If the Back-End's data width (BE_DATA_W) is larger than the front-end's (FE_DATA_W), the data buses require alignment. The address signal becomes word-aligned, discarding the back-end's byte offset bits. These discarded bits are used to align both the write data and strobe (Figure 3.17).

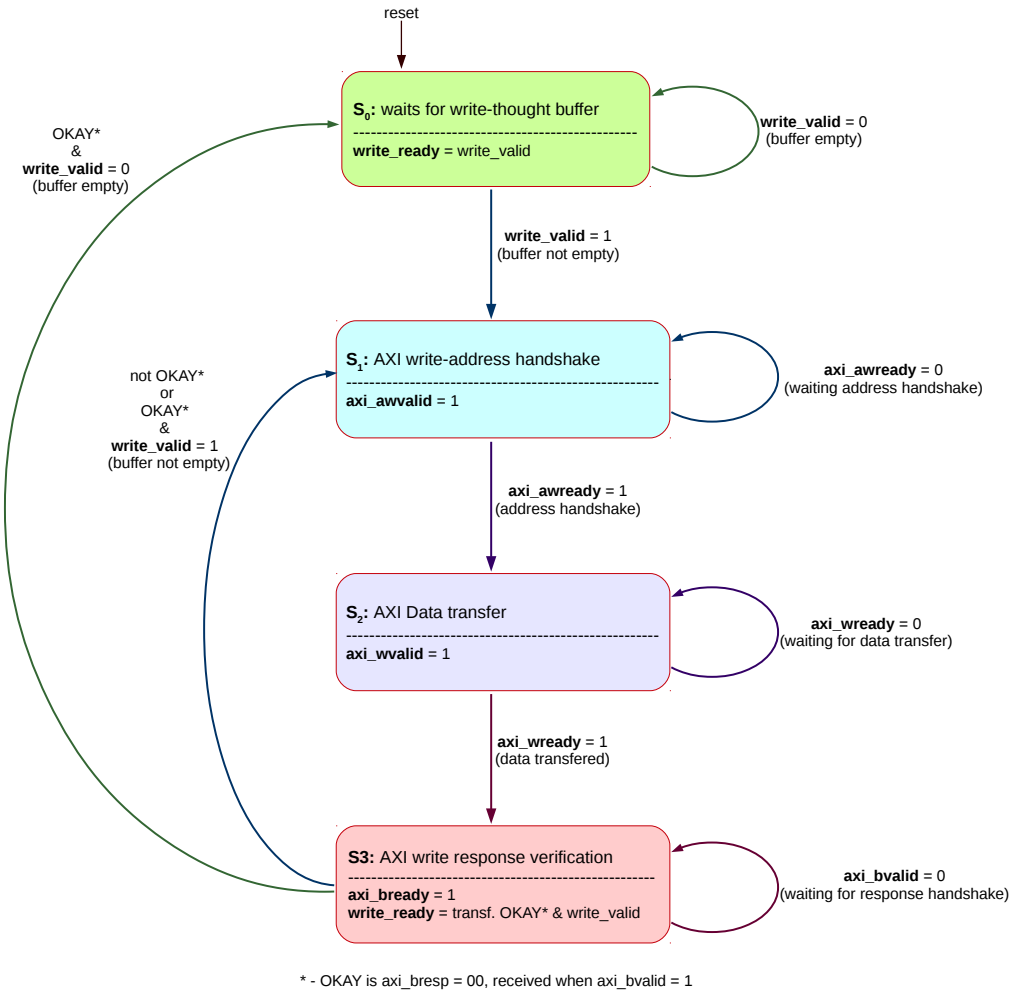

Figure 3.16: Back-End Write-channel AXI Control-flow.

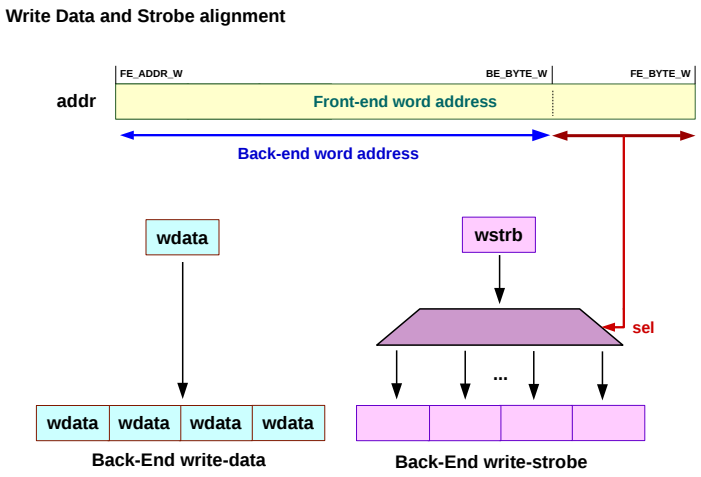

Figure 3.17: Back-End Write-channel alignment.

This results in Narrow transfers [1, p. A3-49], allowing the smaller words to be transferred to a larger bus. The Write-Channel data width is, therefore, limited to the cache's front-end word size. For example, in a 32-bit system, connected to a 256-bit wide memory, each transfer will be limited to 32-bit anyway.

3.5.2 Read-Channel Controller

The Read-Channel controller ports for both the native and AXI4 interfaces are described in Table 3.8.

Table 3.8: Read-Channel ports.

Parameter	Width (bit)	Direction	Description
Cache Side Native Interface			
replace_valid	1	input	Requests a line replacement
replace_addr	FE_ADDR_W -LINE2MEM_W -BE_BYTE_W	output	Memory block's base address.
replace	1	output	Replacement in action.
read_valid	1	output	Validates the Back-End Memory's transfered data.
read_addr	LINE2MEM_W	output	Addresses the placement in the cache line.
read_data		input	Back-End Memory's transfered data.
Memory Side Native Interface			
mem_valid	1	output	Validates the read address.
mem_addr	BE_ADDR_W	output	Read address of the memory block's words.
mem_rdata	BE_DATA_W	input	Read data, used to update the cache line.
mem_ready	1	input	Ready signal, validates the received read data.
Memory Side AXI Interface			
axi_arvalid	1	output	Validates Read Address.
axi_araddr	BE_ADDR_W	output	AXI's Read Address, memory block's initial address.
axi_arready	1	input	Acknowledges address request.
axi_rvalid	1	input	Validates the read data sent by the memory.
axi_rdata	BE_DATA_W	output	Read data.
axi_rready	1	output	Cache acknowledges the transfer, asserted during line replacement.
axi_rresp	2	input	Read response signal. Verifies if transfer was legal ("00" - OKAY). [1, p. A3-54]
axi_arlen	8	output	Burst's length, depends on LINE2MEM_W [1, 14, p. A3-48].
axi_arsize	3	output	Bytes per beat, $\log_2$(BE_NBYTES) [1, p. A3-49].
axi_arburst	2	output	Burst type, incremental (01_b) [1, p. A3-49].
axi_arid	AXI_ID_W	output	AXI identification tag [1, p. A5-81].
axi_arlock	1	output	Atomic access. Normal access 0_b [1, p. A7-100].
axi_arcache	4	output	Identifies the type of memory. Default 0011_b [1, p. A4-68].
axi_arprot	3	output	Permission for illegal transfers, unused. All accesses are normal, 000_b [1, 14, p. A4-75].
axi_arqos	4	output	QoS identifier, unused, 0000_b [1, p. A8-102].

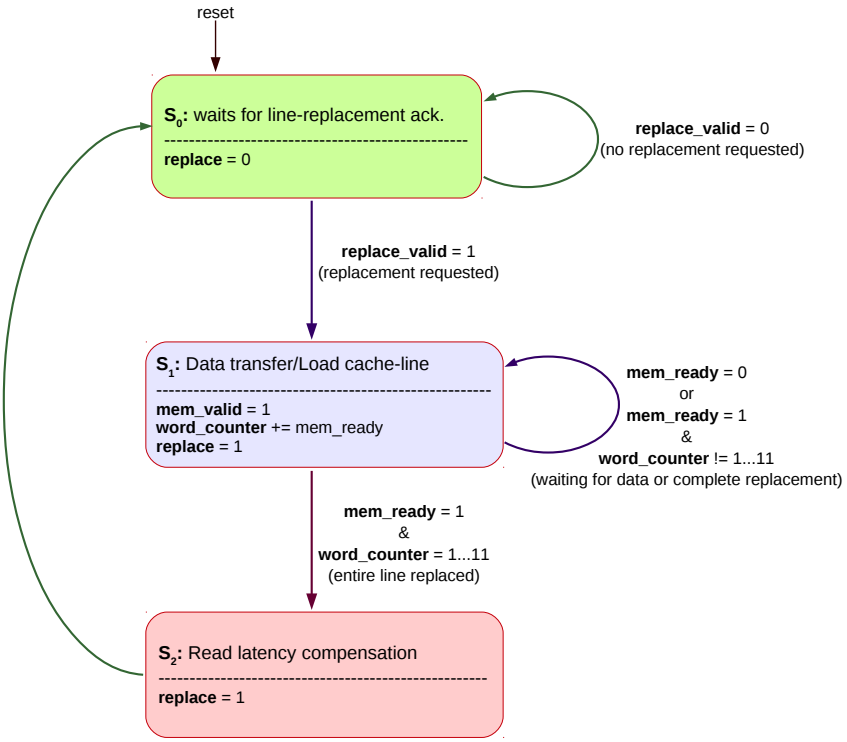

Figure 3.18: Back-End Read-channel Native Control-flow.

The native interface's controller follows the control flow displayed in Figure 3.18. The controller stays in the initial state S_0 while waiting for the request of a line replacement. When signal "replace" is asserted, the controller goes to state S_1 requests a word block from the memory and writes it to the cache line at 1 word per cycle after it arrives at the back-end. It requests the base address of the main memory's memory block and uses a word counter to count the received words. After the last word is received the controller goes to state S_2 for a single cycle to compensate for the Data memory RAM's read latency. Afterward, it goes back to its state S_0, de-asserting signal "replace".

If the back-end's data width (BE_DATA_W) is multiple the front-end's (FE_DATA_W), the number of words counted is proportionally shorter. If the back-end's data width is the same size as the entire cache line, the burst length is 1 and therefore the word counter is not used.

The AXI interface controller (Figure 3.19) has a similar behavior but uses AXI4 burst transfers. The AXI burst parameters are derived for synthesis, using the front-end and back-end data widths, and the cache line's offset width. Instead of using a word counter, the signal axi_rlast is used to know when the line has been fully replaced. During the burst, each beat (transfer) increments signal read_addr automatically.

Unlike the Write-Channel controller, the response signal, "axi_rresp", is sent during each beat (transfer) of the burst. This requires the use of a register which sets in the case at least one of the beats was unsuccessful. After the transfers, the verification of this register can be done at the same time as the read latency compensation.

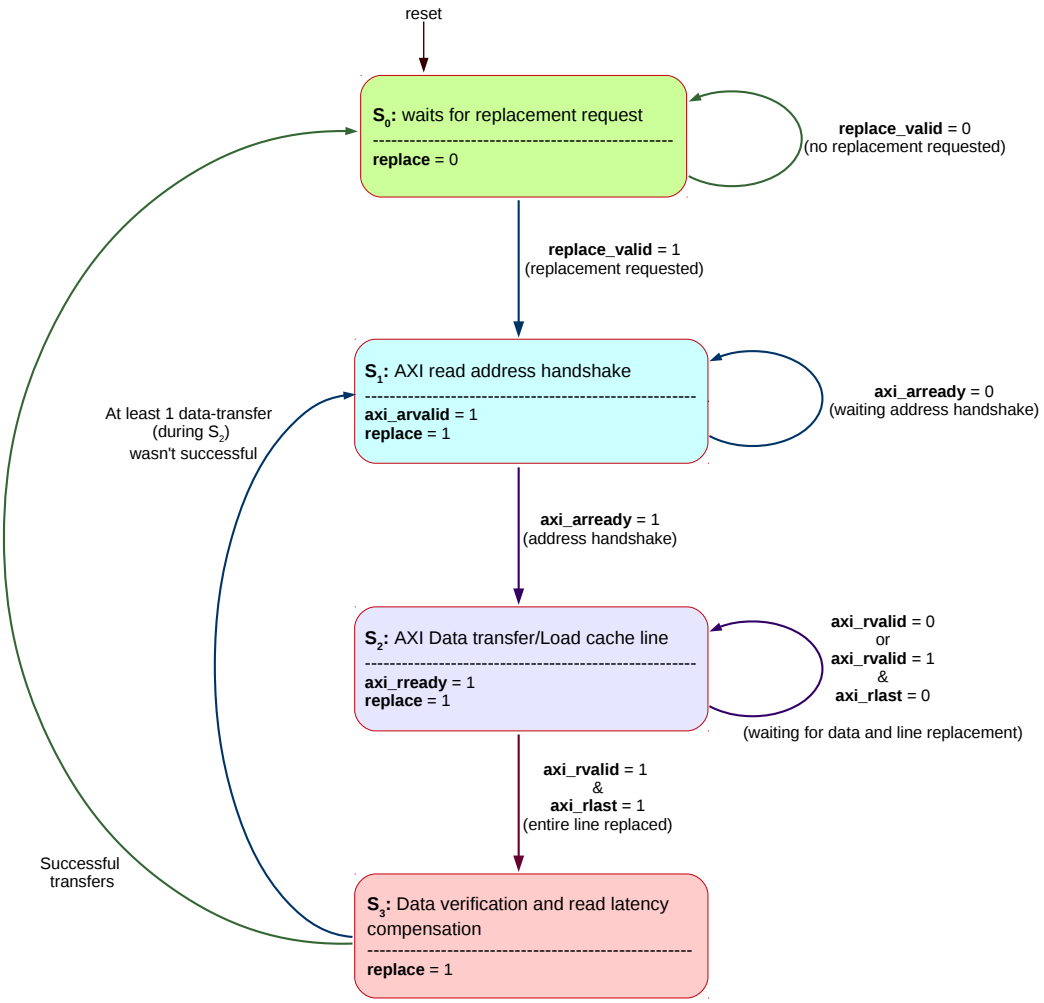

Figure 3.19: Back-End Read-channel AXI Control-flow.

3.6 Cache-Control

The Cache-Control module can optionally be implemented using the synthesis parameter ”CTRL_CACHE”. It is used to measure the cache performance, analyze the state of its write-through buffer, or invalidate its contents. Additionally, the parameter ”CTRL_CNT” implements counters for cache hits and misses, for both read and write accesses. The Cache-Control ports are described in Table 3.9.

Table 3.9: Cache-Control ports.

Parameter	Width (bit)	Direction	Description
reset	1	input	Sets counters to ”0” (if implemented).
valid	1	input	Valid signal, validates the requested access.
addr	CTRL_ADDR_W	input	Base address of the requested task.
rdata	FE_DATA_W	output	Read-data, returns the task’s result (buffer status or counters).
ready	1	output	Ready, acknowledges the request (always 1 clock-cycle after).
invalidate	1	output	Cache-invalidate. Invalidates Cache-Memory.
wtbuf_empty	1	input	Write-through buffer’s empty signal.
wtbuf_full	1	input	Write-through buffer’s full signal.
write_hit	1	input	Write-hit, increments respective counter (if implemented).
write_miss	1	input	Write-miss, increments respective counter (if implemented).
read_hit	1	input	Read-hit, increments respective counter (if implemented).
read_miss	1	input	Read-miss, increments respective counter (if implemented).

The Cache-Control functions are controlled by memory-mapped registers [9, p. 627]. The addresses of the software accessible can be found in the cache’s Verilog and C header files. In the current implementation, the registers are accessed from the cache’s front-end, as 32-bit integers when using the C driver.

The ports write_hit, write_miss, read_hit, and read_miss work as enables that cause the respective counters to increment. One exception is during a read-miss when the cache forces a write-hit in order to replace the cache line, and the write-hit counter is decremented. To reduce the number of registers, accessing the total number of hits (or misses) is in fact accessing the output of an adder that adds read and write hits (or misses). The counters can be reset by hardware (global system reset) or by software.

Chapter 4

System

This chapter contains a brief introduction to the system where IOb-Cache was implemented, and presents a multi-level cache implementation with multiple instances of IOb-Cache.

4.1 IOb-SoC

IOb-Cache has been integrated in IOb-SoC [4], an open-source synthesizable system developed by IObundle in Verilog. Its design can be seen in Figure 4.1.

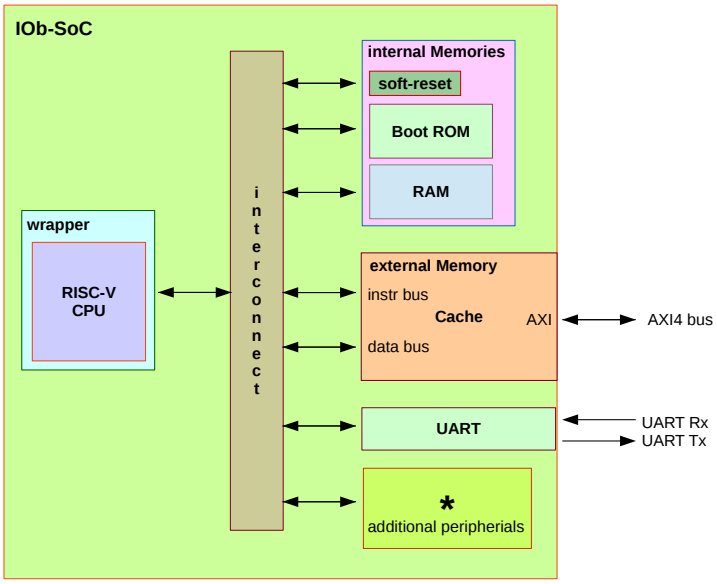

Figure 4.1: IOb-SoC module diagram.

The system is designed to allow the integration of multiple user peripherals, accessed through memory-mapping. Each individual peripheral device is given a specific address range to be accessed. The main peripherals required for a functional bare-metal system will be briefly described.

The interconnect is implemented with ”split” [15] units, which is the module responsible for connecting the processor (master) to the remaining peripherals (slaves). The connection is established through

memory-mapping, where the MSB or the MSB-1 bit of the address selects all peripherals, depending on whether a secondary memory is not present or present in the system, respectively.

This system is controlled by a RISC-V processor. A CPU wrapper converts the CPU interface signals to the Native interface used internally throughout the system for interconnecting the different modules. The wrapper has the necessary combinational and sequential logic for implementing this interface. Currently, a simple 2-stage machine (PicoRV32 [16, 17]), or a more complex super-scalar multi-issue processor (SSRV [5, 18]) are supported.

For communications between the system and the host, a UART module (IOb-UART [19]) is integrated. It uses the Universal Asynchronous Receiver / Transmitter protocol (UART) for transmitting and receiving serial data.

An SRAM memory and a Boot Read-Only Memory (ROM) memory are integrated into a module called Internal Memory, which also contains a soft reset mechanism for transitioning from the bootloader to the main program and vice-versa.

The Boot ROM memory contains a program that runs the boot sequence and runs when the system is reset. This program loads the firmware into the main memory while receiving it from the host using the UART. After the firmware is loaded, the system is soft-reset, which causes the memory map to be restructured. After soft reset the system restarts but, instead of running the bootloader program, it starts fetching instructions from the main memory where the firmware had been previously loaded, specifically at address 0. When the soft reset register is accessed it toggles a control register which is used to alter the system's memory-map. After running the firmware, this register is retoggled and a new soft reset is issued, causing the system to reboot and run the bootloader again to load the firmware, which thus can be changed without recompiling the hardware.

External Memory module allows access to an external and larger DRAM memory (DDR3 or DDR4), and is where the IOb-Cache modules are placed. External Memory module connects the system to an external DDR memory soft controller provided by the FPGA vendor and using the AXI4 interface. This explains why AXI4 interfaces have been implemented for the cache back-end.

The IOb-SoC system has been implemented in an XCKU040-1FBVA676 FPGA [20], which is part of the Xilinx's Ultrascale FPGA family. A diagram of the FPGA system and board is shown in Figure 4.2.

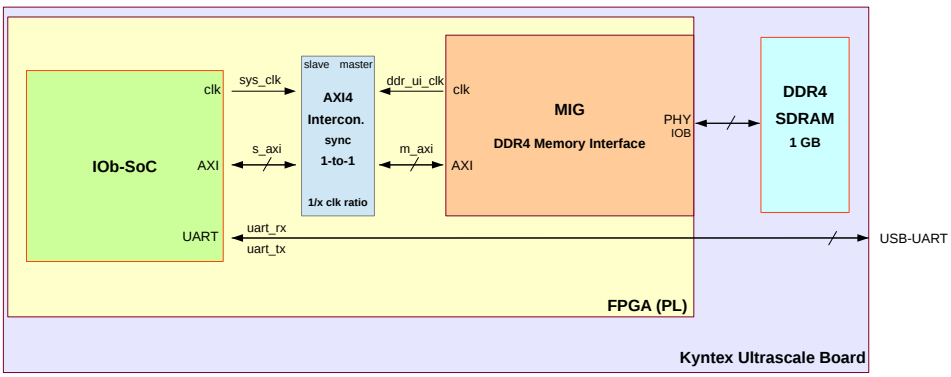

Figure 4.2: IOb-SoC FPGA implementation module diagram.

This board features a 250MHz system clock oscillator and a 1 GB DDR4 Synchronous Dynamic Random-Access Memory (SDRAM) memory module. The memory controller Physical Interface (PHY) is implemented using Xilinx's Memory Interface Generator (MIG) software which generates memory controller IP cores [21] from user parameters. The controller is implemented with the widely adopted AXI4 interface which connects to the cache's back-end.

The memory controller clock has 1/4 of the frequency at which the memory PHY operates. The range of available frequencies for the DDR4 is between 625 and 933 MHz. This limits the range of clock frequencies for the memory controller and system if connected using the same clock, to be between 156.25 and 233.25 MHZ. In the current design, the controller is running at 200 MHz, which means the DDR4 runs at 800 MHz.

The DDR4 module has a bandwidth of x32@1600Mbps, so it would be able to exchange 32-bit words with the system at a frequency of 1600 MHz, but this frequency is too high for an FPGA to achieve. Because the controller is only running at 200 MHz, the AXI4 data bus width must be increased proportionally to the desired memory bandwidth. With a width of 256 bits, the full memory bandwidth can be put to use. The external memory and cache back-end bus widths need to be configured for 256 bits for this to happen.

The system may not be set to operate at the controller's frequency range, so connecting it to the controller requires an asynchronous 1-to-1 AXI-Interconnect IP is also supplied by Xilinx. However, the Xilinx asynchronous interconnect IP introduces a high latency, so it is best to choose an integer clock ratio and use the synchronous version.

4.2 IOb-Cache: Multi-Level configuration

IOb-Cache modules can be connected to each other to form multi-level cache systems. A two-level cache system, composed of an L1-Instruction cache, an L1-Data cache, both connected to a larger L2-cache is represented in Figure 4.3. The two L1 caches access different types of data, one accesses instructions, and the other accesses data. The L2 cache merges the accesses of the instruction and data caches and thus may contain both instructions and data.

The back-end of the L1 instruction and data caches use the Native Interface and are connected to a 2-to-1 interconnect called ”merge” [15]. The merge unit connects several masters to a slave interface using a fixed and sequential priority encoder. A master remains connected to a slave until the request is acknowledged. The back-end of the merge block is connected to the front-end of the L2 cache which also uses the Native interface. The L2 back-end uses the AXI4 interface and is connected to the memory controller.

The Cache-Control optional module can only be implemented in the L1-Data cache since it is the only cache directly accessed by the processor, and the instruction L1 cache does not need one. To access the L2-cache, either for a cache invalidation or checking of the status of the write-through buffer, the CTRL_IO pins are used instead. The CRL_IO interface supports multi-cache systems, so accessing the Cache-Control module for status verification, shows the status of the downstream connected caches.

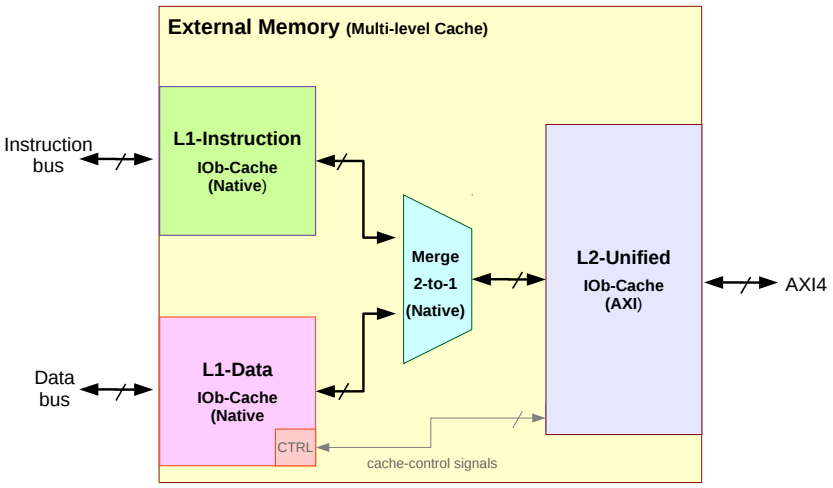

Figure 4.3: External Memory: two-level cache system implementation.

This is necessary during the IOb-SoC booting procedure, to check if the entire firmware has already been written to the main memory before restarting the system to run it.

Chapter 5

Results

This chapter presents results on IOb-Cache's performance and FPGA implementation. A comparison between IOb-Cache and other open-source caches is also presented.

5.1 Performance

The Dhrystone [22] benchmark is a useful tool for measuring the performance of processors, using the Dhrystones/s score. The frequency-independent Dhrystone score is called Dhrystone Mega Instructions per Second (MIPS)/MHz or simply DMIPS. Here the Dhrystone benchmark is used to indirectly evaluate the cache as the performance of the system translates the performance of the cache if the processor remains the same. To do that, the Clocks-per-Instruction (CPI) measurement is taken while running the benchmark, as it provides a more direct indication of the cache performance compared to the DMIPS figure.

To efficiently test the cache, a pipelined processor is required, with a performance close to 1 CPI when using an ideal RAM memory. This way it is possible to analyze the average delay of the cache during memory accesses, Equation 2.1.

The Dhrystone benchmark has one shortcoming for testing the various policies, it is a small program that can be fitted entirely inside an instruction cache of common size. This shortcoming becomes an advantage for testing the pipeline operation, since it after full it behaves like a RAM, in a system connected to a larger SDRAM. Being a RAM, the correct pipeline operation happens with consecutive loads and stores which should take 1 cycle per instruction. A correct cache design allows for 1-cycle loads and stores whereas a poorer design will need 2 cycles for load/store instructions.

The tests are run in IOb-SoC [4] (Section 4.1), using the SSRV [5, 18] multi-issue superscalar RISC-V processor. Despite being multi-issue, the processor was limited to 1 instruction per clock cycle in the tests, which is a simple setup but allows testing the cache. Connected to the IOb-SoC's internal memory (RAM only and no cache), it achieved CPI=1.02, running for 40445 clock cycles. The cache is implemented following the structure represented in Section 4.2: an L1-Instruction and L1-Data caches connected to an L2-Unified cache. In the subsections below simulation and FPGA results are presented.

5.1.1 Simulation

The simulation results are displayed in Table 5.1, with the cache is connected to an AXI4 RAM.

The minimum possible size for 2-level configuration is 48 Bytes, 16 Bytes for each of the 3 caches. This is the worst possible scenario performance-wise. If the L1s do not have the requested word, neither does the L2. The large delay in between instructions is caused by the high miss rate, causing accesses to the main memory, as well as traffic congestion between the L1s and L2 accesses.

Using 2 KB caches, one can see there is no performance difference between the replacement policies in a 2-way set-associative cache. The way selected is the one that was not the most recently used in all cases. It also shows the difference in performance between the set-associative and directly mapped cache. Using a set-associative in the L2-Unified cache represents the largest improvement in performance (up to 0.315 CPI). If the three caches only use direct mapping, the performance drops by 25.8%.

Using 4 KB caches highlights the differences in performance of the different replacement policies. The PLRUm policy displays the highest performance in all three caches, while the LRU policy gives the worst performance. The reduced size of the L1-Instruction (1 KB), and the firmware's instruction loops constitutes an environment where replacing the least recently used is not effective, due to low time locality. The PLRU policies lack memory compared to the LRU and are worse at identifying the most recently used line. However, this ends up not being a handicap as there is no time locality to exploit. The L2-Unified is more likely to see a performance improvement with PLRU policies [10, 23]. This results from the fact L2 is accessing different memory blocks (instructions and data) with inherently low time locality.

Using 16 KB and 32 KB caches, the size is large enough to fit the program. There is no change in performance between the different replacement policies. Despite the program being 25 KB in size and the L1-Instruction caches 4 KB and 8 KB, respectively, the program is not required to fit entirely in these memories. As the program is executed, the only misses that occur are the initial compulsory misses, followed by capacity misses that replace the previous non-looping instructions. As the caches are big enough to store all recently looping code, conflict misses becoming inexistent.

In all the previous examples, the choice of the size for the L1 data cache has little significance since the Dhrystone benchmark is an instruction-intensive program.

Table 5.1: Simulation Dhrystone SSRV (IOb-SoC) 32-bit. 100 runs using gcc -O1 optimization. Parameters: number of ways (repl. policy), lines, words per line.

L1-Instr	L1-Data	L2-Unified	clock cycles	CPI
48 B - Minimum size				
1, 2, 2	1, 2, 2	1, 2, 2	319580	8.066
2 KB				
2 (LRU), 8, 8	2 (LRU), 8, 8	4 (PLRUm), 8, 8	162147	4.092
2 (PLRUm), 8, 8	2 (PLRUm), 8, 8	4 (PLRUm), 8, 8	162147	4.092
2 (PLRUt), 8, 8	2 (PLRUt), 8, 8	4 (PLRUm), 8, 8	162147	4.092
1, 16, 8	1, 16, 8	4 (PLRUm), 8, 8	174620	4.407
1, 16, 8	1, 16, 8	1, 16, 16	204016	5.149
4 KB				
4 (LRU), 8, 8	4 (LRU), 8, 8	8 (LRU), 8, 8	95331	2.406
4 (LRU), 8, 8	4 (LRU), 8, 8	8 (PLRUm), 8, 8	87031	2.196
4 (LRU), 8, 8	4 (LRU), 8, 8	8 (PLRUt), 8, 8	90417	2.282
4 (PLRUm), 8, 8	4 (LRU), 8, 8	8 (PLRUm), 8, 8	79310	2.001
4 (PLRUt), 8, 8	4 (LRU), 8, 8	8 (PLRUm), 8, 8	84854	2.141
4 (PLRUm), 8, 8	4 (PLRUm), 8, 8	8 (PLRUm), 8, 8	79310	2.001
4 (PLRUm), 8, 8	4 (PLRUt), 8, 8	8 (PLRUm), 8, 8	79310	2.001
1, 64, 4	1, 64, 4	1, 64, 8	107668	2.717
8 KB				
2, 16, 16	2, 16, 16	4 (LRU), 16, 16	50758	1.281
2, 16, 16	2, 16, 16	4 (PLRUm), 16, 16	50751	1.281
2, 16, 16	2, 16, 16	4 (PLRUt), 16, 16	50758	1.281
1, 32, 16	1, 32, 16	1, 64, 16	77306	1.951
1, 64, 8	1, 64, 8	1,128, 8	71543	1.805
16 KB				
4, 16, 16	4, 16, 16	8, 16, 16	41837	1.055
4, 32, 8	4, 32, 8	8, 32, 8	41762	1.055
2, 64, 8	2, 64, 8	4, 64, 8	41886	1.057
1, 64,16	1, 64, 16	1, 128, 16	56848	1.434
1, 128, 8	1, 128, 8	1, 128, 16	54986	1.387
32 KB				
8, 16, 16	8, 16, 16	16, 16, 16	41837	1.055
2, 128, 8	2, 128, 8	4, 128, 8	41762	1.054
1, 256, 8	1, 256, 8	1, 256, 16	41811	1.055
2, 64, 16	2, 64, 16	4, 64, 16	41837	1.055

5.1.2 FPGA

The FPGA system is implemented in the ultrascale FPGA, as described in Section 4.1. It uses a 50 MHz clock, 1/4 of the Memory Interface's frequency, so it requires the implementation of the synchronous AXI-Interconnect with a 1:4 clock ratio. The results are presented in Table 5.2.

Table 5.2: FPGA emulation of Dhrystone SSRV (IOb-SoC) 32-bit at 50 MHz. 100 runs using gcc -O1 optimization. Parameters: number of ways (rep. policy), lines, words per line.

L1-Instr	L1-Data	L2-Unified	clock cycles	CPI
48 B - Minimum size				
1, 2, 2	1, 2, 2	1, 2, 2	513926	12.971
2 KB				
2 (PLRUt), 8, 8	2 (PLRUt), 8, 8	4 (PLRUm), 8, 8	185163	4.673
8 KB				
2 (PLRUt), 16, 16	2 (PLRUt), 16, 16	4 (PLRUm), 16, 16	51298	1.294
32 KB				
2 (PLRUt), 64, 16	2 (PLRUt), 64, 16	4 (PLRUm), 64, 16	42397	1.070

In simulation, the main memory is modeled using an AXI4 RAM description. In the FPGA the main memory is implemented with a DDR4 external module, with a longer access time. The increased SDRAM's access time compared to simulation results in an increase of 60.8% (4.905 CPI) for the 48-Byte cache. Since it is the smallest cache, its average access time is closest to the SDRAM's. With the 2 KB cache, the access time only increases 14.2% (0.581 CPI). With 8 KB the cache is large enough to fit most of the program, and the FPGA times compared to simulation are only 1.01% higher (0.013 CPI).

5.2 Synthesis

In this section, the cache's synthesis results are analyzed. First the resource utilization for each individual module is checked, followed by the resources consumed by the entire cache.

The synthesis tool used is Xilinx's Vivado Design Suite 2017 [24], with the FPGA part "xcku040-fbva676-1-c" (AES KU040 Ultrascale). The resources in analyses are: Look-Up Table (LUT); Look-Up-Table Random-Access-Memory (LUTRAM); Flip-Flop (FF); and Block Random-Access-Memory (BRAM). For the Block-RAM resources, there are 2 variants: RAMB18, RAMB36 [25, p. 7].

Despite being able to change the cache's word size with the parameter FE_DATA_W, it was left to 32-bit, since the cache was only tested in 32-bit systems.

The cache submodules are synthesized using a 100 MHz clock for the resources presented in the next subsections. The entire cache is synthesized at 100 and 250 MHz clock frequency and respective resources presented.

5.2.1 Front-End

The Front-End has two possible configurations, with or without the Cache-Control's implementation. The results are represented in Table 5.3. The number of registers to store the input signals: 32 bits for write-data, 28 bits for the address (word-addressable) and 1 for valid signal is the same with or without the Cache-Control module. With the Cache-Control module, the area is essentially consumed by the multiplexers implemented to select the read-data.

Table 5.3: Front-End resources.

CTRL_CACHE	LUT	FF
0	1	67
1	19	67

5.2.2 Back-End

For Back-End module, both the Read-Channel and Write-Channel modules are analyzed individually. The Back-End is synthesized for data widths of 32 and 256 bits. Table 5.4 shows the Write-Channel's resources for both Native and AXI interfaces. The 256-bit back-end adds multiplexers to place the 32-bit words and 4-bit write-strobe in the back-end's bus. The AXI bus type adds negligible resources compared to the Native type.

Table 5.4: Back-End: Write-Channel's resources.

FE_DATA_W	BE_DATA_W	LUT	FF
Native-Interface			
32	32	4	1
32	256	35	1
AXI-Interface			
32	32	6	2
32	256	40	2

Table 5.4 represents Read-Channel's resources for both Native and AXI interfaces. The amount of logic required depends strongly on the width ratio between the main memory word and the cache line. The Back-End Native-Interface requires an additional 15 LUTs to select which controller accesses the main memory since both share the same bus.

5.2.3 Resources: Cache-Control

Cache-Control has 2 available implementations, with counters or without them. In a bare Cache-Control implementation, the 3 LUTs and 3 FFs are the memory-mapped registers, used to select the functions: invalidate, write-through buffer empty and full. If implemented, each counter requires a 32-bit register. The large increase in LUTs results from the arithmetic adders. Each counter requires

Table 5.5: Back-End Read-Channel's resources.

FE_DATA_W	BE_DATA_W	Cache line Width	LUT	FF
Native-Interface				
32	32	64	4	3
32	32	128	5	4
32	32	256	7	5
32	64	64	4	2
32	64	128	4	3
32	64	256	5	4
32	256	256	4	2
32	256	512	4	3
32	256	1024	5	4
AXI-Interface				
32	32	64	7	4
32	32	128	8	5
32	32	256	9	6
32	256	256	5	2
32	256	512	7	4
32	256	1024	8	5

incrementers, and 2 additional adders are required for the number of cache-hits and cache-misses. With the counters, 7 Cache-Control functions are available, requiring additional memory-mapped registers. The functions are: retrieving the number of cache hits / misses / read-hits / read-misses / write-hits / write-misses, and resetting the counters.

Table 5.6: Cache-Control resources.

CTRL_CACHE	CTRL_CNT	LUT	FF
1	0	3	3
1	1	266	163

5.2.4 Replacement Policy

The Replacement Policy module is analyzed before the Cache-Memory module since the former is implemented in the latter. The results of the analysis are available in Table 5.7.

The test is divided into 2 sections: single cache line and multiple cache lines. It is not possible to synthesize the entire cache with a single cache line, so this is only valid for the analysis of this module.

The single cache line results show how many LUTs are required to implement the Policy Info Updater and Way Select Decoder. The number of FFs represents the number of bits the Policy Info Memory module needs to store for each set.

Table 5.7: Replacement Policy resources.

R.Policy	Ways	Lines	LUT	FF
Single cache line				
LRU	2	1	3	2
PLRUm	2	1	3	2
PLRUt	2	1	3	1
LRU	4	1	20	8
PLRUm	4	1	8	4
PLRUt	4	1	6	3
LRU	8	1	81	24
PLRUm	8	1	22	8
PLRUt	8	1	13	7
Multiple cache lines				
LRU	2	16	26	32
PLRUm	2	16	26	32
PLRUt	2	16	28	16
LRU	4	16	70	128
PLRUm	4	16	42	64
PLRUt	4	16	33	48
LRU	8	16	196	384
PLRUm	8	16	77	128
PLRUt	8	16	56	112
LRU	2	128	222	256
PLRUm	2	128	222	256
PLRUt	2	128	212	128
LRU	4	128	445	1024
PLRUm	4	128	297	512
PLRUt	4	128	259	384
LRU	8	128	1076	3072
PLRUm	8	128	452	1024
PLRUt	8	128	403	896

The multiple cache lines results show the current actual amount of resources required to implement each replacement policy. The current implementation of the Policy Info Module (PIM) is register-based, so it requires additional logic (LUTs) to address each set. The number of LUTs is proportional to the total number of bits in the PIM.

Since the LRU requires N_WAYS $\times$ $\log_2$(N_WAYS) bits per set, initially its size grows fast with the number of ways. In an 8-way set-associative cache with 128 lines, the LRU requires more than twice the

amount of LUTs and at least thrice the amount of FFs compared with the PLRU policies. In a 2-way set-associative cache, the replacement policies had the same performance but the PLRUt's PIM requires half the number of FFs compared to the other two.

5.2.5 Cache-Memory

Cache-Memory is the module that contains the majority of the cache's resources. It contains all the RAM memories and, if configured, the Replacement Policy module too. The synthesis results are available in Table 5.8.

As mentioned in Section 3.3, Data-Memory infers a RAMB18 blocks for each byte in the cache line. Since the RAMB18 block is 18-bit wide, roughly half of it is wasted. This issue can be solved by describing a RAM with multiple byte-enables as in the Vivado Synthesis Guide [25] but unfortunately the description is not portable for other FPGAs such as Intel's.

In the configurations with 128 lines or lower, the cache is implemented with LUTRAMs plus output registers. With 128 lines or more, RAMB18 is used. RAMB36 blocks are never inferred because these have a 36-bit width. The Write-Through Buffer, which is 64-bit wide, is implemented with LUTRAMs plus output registers if its depth is 32 or lower, or is implemented with RAMB36 if the depth is higher than 32. Note that RAMB36 blocks can be configured for 64-bit width and RAMB18 blocks can not.

In general, looking at the results in Table 5.8, the memory resources increase with both the width and depth of the cache memory, although they increase by steps that have to do with the capacity of the FPGA RAMB18 and RAMB36 memory resources. Increasing the number of ways, increases everything, memory, and logic. The logic increases significantly to combine multiple ways and implement the Replacement Policy module.

5.2.6 IOb-Cache

In this section, the entire cache module is analyzed. Table 5.9 displays the synthesis and timing results of IOb-Cache using the Native interface for 2 different clock frequencies: 100 and 250 MHz. The results for IOb-Cache with AXI Back-End are similar and differ only in 15 LUTs and 2 FFs.

The implementation differs for the 2 clock frequencies. The used memory is enough for BRAMs to be inferred for both the Tag and Data memories. For 100 MHz, the critical-path is from Tag memory output to a Data memory write enable signal. This path is caused by the signal way_hit, which results from the tag comparison and respective validation, and needs to be connected to write enable bits on write-hit access. However, for 250 MHz the synthesis tool deliberately decides to implement the Tag-Memory with LUTRAMs, with a stage register at the output, to be able to meet the timing constraint.

Table 5.8: Cache-Memory resources.

Ways	R.Policy	Lines	Words/line	LUT	LUTRAM	FF	RAMB36	RAMB18
1 KB								
1		16	16	496	534	599	1	0
1		64	4	292	128	239	1	1
1		128	2	300	0	174	1	9
2 KB								
1		32	16	548	533	614	1	0
1		128	4	341	0	175	1	17
2	PLRUt	16	16	950	1068	1167	1	0
4 KB								
1		32	32	995	1044	1126	1	0
1		128	8	405	0	176	1	33
2	PLRUt	128	4	819	0	433	1	34
8 KB								
1		128	16	551	0	177	1	65
2	LRU	128	8	1037	0	562	1	66
2	PLRUm	128	8	1037	0	562	1	66
2	PLRUt	128	8	1003	0	434	1	66
16 KB								
1		128	32	957	0	178	1	129
1		512	8	933	0	560	1	33
4	LRU	128	8	2055	0	1590	1	132
4	PLRUm	128	8	1913	0	1078	1	132
4	PLRUt	128	8	1877	0	950	1	132
4	PLRUt	64	16	2282	0	1845	1	68
32 KB								
1		128	64	1760	0	179	1	257
1		1024	8	1616	0	1072	1	33
8	LRU	128	8	3935	0	4158	1	264
8	PLRUm	128	8	3341	0	2110	1	264
8	PLRUt	128	8	3293	0	1982	1	264

Table 5.9: IOb-Cache (Native) resource and timing analysis.

Ways	R.Policy	Lines	Words/line	LUT	LUTRAM	FF	RAMB36	RAMB18	WNS
100 MHz (10 ns)									
4 KB									
1		128	8	431	0	249	1	33	4.047
4	PLRUm	16	16	1727	1068	2407	1	0	3.212
1		128	8	413	0	251	1	33	
8 KB									
2	PLRUt	128	8	1025	0	509	1	66	2.977
16 KB									
4	PLRUm	128	8	1940	0	1154	1	132	2.158
32 KB									
4	PLRUm	256	8	2961	0	2187	1	132	1.199
1		1024	8	1638	0	1145	1	33	4.003
250 MHz (4 ns)									
4 KB									
1		128	8	510	40	269	1	32	0.398
4	PLRUm	16	16	1730	1068	2407	1	0	0.024
8 KB									
2	PLRUt	128	8	1084	80	549	1	64	0.228
16 KB									
4	PLRUm	128	8	1974	160	1234	1	128	0.103
32 KB									
1		1024	8	1714	272	1162	1	32	0.523
4	PLRUm	256	8	2981	304	2289	1	128	-0.120

5.3 Open-Source Caches

In this chapter, the IOb-Cache is compared with the configurable PoC.cache design included in the PoC-Library [8] library of open-source cores. PoC.cache is the most competitive open-source cache one was able to find, so the other caches mentioned in Chapter 2 are not evaluated here as clearly they cannot compete with IOb-Cache or PoC.cache. The comparison between the two caches is available in Table 5.10 below.

In addition to the information in Table 5.10, the following remarks are needed. The data-width of the back-end of PoC.cache is fixed to the cache line's size, and therefore not configurable to be smaller such as in IOb-Cache. The PoC.cache tag and valid memories are implemented with distributed LUTRAM and registers, respectively, to combinatorially check for a hit and achieve 1 read per clock cycle. Lastly, despite using the Write-Though policy, PoC.cache does not have a buffer and accesses the main memory for write transfers, which is comparatively slower.

Based on the information in Table 5.10, the following conclusions can be drawn. There are 2 points where PoC.cache is better than IOb-cache: (1) the implementation of the cache invalidate function and (2) the implementation of a fully associative cache. PoC.cache is able to invalidate individual lines whereas IOb-Cache can only invalidate the entire cache. PoC.cache can be configured as a fully-associative (single set) cache and IOb-Cache needs at least 2 sets. However, besides its theoretical interest fully-associative caches are seldom used in practice.

In the remaining features, IOb-Cache is better than PoC.cache: configurable back-end size with AXI4 interface as an option; write-through buffer and independent controller for fast, most of the time 1-cycle writing (PoC.cache only supports 1-cycle reads); more replacement policies to choose from; a modular design that allows changing both front and back-ends without affecting the cache's core functionality.

Both PoC.cache and IOb-cache have the same issue of implementing the Tag-Memory and Policy Info Module using registers, and thus consuming more silicon area than necessary. However, because IOb-Cache is designed to work with the 1-cycle read latency of RAM, it can easily be upgraded to replace these memories with RAMs while PoC.cache needs more drastic design changes.

Table 5.10: Comparison between PoC.cache and IOb-Cache.

	PoC.cache	**IOb-Cache**
HDL	VHDL	Verilog
Configurability		
num. of ways	Yes	Yes
num. of lines	Yes	Yes
num. of words/line	Yes	Yes
back mem.'s width	Yes	Yes
Mapping		
Direct	Yes	Yes
Set-Associative	Yes	Yes
Full-Associative	Yes	No
Policies		
Write	write-through not alloc.	write-through not alloc.
W.Through Buffer	No	Yes
Replacement	LRU	LRU, PLRUm, PLRUt
Back-End Connectivity		
Native	Yes	Yes
AXI	No	AXI4
Implementation		
Main-control	FSM	Data-path
Data-Memory	BRAM	BRAM
Tag-Memory	LUTRAM	BRAM
Valid-Memory	Register	Register
Rep-Pol. Mem	Register	Register
Invalidate	Yes	Yes
Inval. source	input port	input port + aux.module
Performance (best case scenario)		
clk/read (hit)	1	1
clk/write	2	1
Ready assertion	same cycle as valid req.	cycle after valid req.
Read-Data availability	cycle after ready	same cycle as ready
New Valid req.	cycle after ready	same cycle as ready

Chapter 6

Conclusions

In this thesis IOb-Cache, a high-performance configurable open-source cache is developed. IOb-Cache is being used in dozens of projects. It is currently integrated into the IOb-SoC Github repository, which has 16 stars and is being used in 38 projects (forks). In the Github cloud community, it is currently the only Verilog cache found by its search tool, with this level of configurability, that supports pipelined CPU architectures, and the popular AXI4 bus interface.

The cache is composed of 3 modules: Front-End, Cache-Memory, and Back-End. The Front-End interfaces the processor with the cache. The Cache-Memory contains the memories and the cache's main controller. The main controller is implemented by a streamlined datapath that evaluates every necessary condition for read and write accesses. The Back-End implements Native and AXI interfaces, allowing flexibility in connecting to 3rd party memory controllers (likely using an AXI interface), and other cache levels (likely using the Native interface). The The native interface follows a pipelined valid-ready protocol, while the AXI4 interface is a full master bus implementation. Each interface has a specific controller for write and read accesses. The Back-End's Write-Channel module is responsible for the write accesses: it reads data from the write-through buffer and writes them to the main memory. The Back-End's Read-Channel module fetches lines from the main memory and writes them to the cache during a cache line replacement. An optional module called Cache-Control can be selected. This module implements cache performance meters, and analyses write-through buffer states and cache invalidates. These functions are controlled by the CPU using memory-mapped registers. When the Cache-Control module is present, the Front-End module acts as a splitter between accesses to Cache-Memory or the Cache-Control modules, also through memory-mapping.

In the remainder of this chapter the main achievements of this work and the main directions for future work are outlined.

6.1 Achievements

In this work, several important achievements deserve to be highlighted. The the following list highlights them.

- The cache supports pipelined memory loads and stores, honoring 1 request per clock cycle. The available configurations revolve around the cache's dimensions, replacement policies, and back-end memory interface data-width. Given the 1-cycle latency present in RAMs, a request can be served while processing the next, which results in a throughput of 1 request per clock-cycle and latency of 2 clock cycles for hit addresses.
- The cache has a modular design that allows keeping its core functionality independent from its interfaces intact, implemented by the Front-End and Back-End modules.
- The cache is able to pass high frequency (250 MHz) timing requirements for a Xilinx Ultrascale FPGA, in a large number of configurations, including the 32 KB direct-mapped or the 8KB 4-way set-associative configurations.
- If large enough, the results show that its performance is close to that of having a fast on-chip RAM connected to the CPU. Using the multi-issue superscalar SSRV CPU, which has CPI=1.02 when connected to a RAM, the cache achieved CPI=1.07.

6.2 Future Work

IOb-Cache can still be further improved beyond the development time allocated to it. The main improvements are listed below in decreasing priority:

- Increase resource efficiency by reducing the amount of logic and memory used. Both the Valid-Memory and PIM modules would be more efficiently implemented with RAM; the Valid Memory could be merged with the Tag-Memory adding only 1 bit to its width. This would eliminate many registers and logic but the ability to invalidate the cache in a single cycle would be lost, as each line would have to be accessed for invalidation using a hardware or software scheme. Implementing the back-end controllers with logic datapaths instead of FSMs is not expected to further reduce the resources used but it would definitely improve the code's readability.
- Implementation of the Write-Back Write-Allocate policy. Currently, only the write-Through policy is supported, which limits the write-bus capacity to the cache's word width and uses more bandwidth of the memory controller, blocking other devices they may need to access the external memory. Using the write-back policy it is possible to optimize this bandwidth and the general performance for some applications. Ideally, these two policies should be configurable.
- Improve the Cache-Control module. The current Cache-Control module is very crude, especially the invalidate function. It should be possible to invalidate a single selected cache line, which would require an address decoder. Alternatively, merging the Valid memory with the Tag memory as proposed before solves this problem. Moreover, if the write-back policy is implemented, it will need a line flush function, which is automatically guaranteed by the ability to individually select lines for invalidation.

- Cache Coherency. The current cache is weak for multi-processor systems since it lacks a coherency controller. A dedicated module and interface to implement a cache coherency algorithm would significantly expand the usability of IOb-Cache.

Bibliography

[1] *AMBA AXI and ACE Protocol Specification*. ARM, 2020. URL https://developer.arm.com/documentation/ihi0022/h/?lang=en. Accessed on December 2020.

[2] IEEE 1364-2005 – IEEE Standard for Verilog Hardware Description Language, November 2005. URL https://standards.ieee.org/standard/1364-2005.html.

[3] IOb-Cache, December 2020. URL https://github.com/IObundle/iob-cache. Accessed on December 2020.

[4] IOb-SoC, December 2020. URL https://github.com/IObundle/iob-soc. Accessed on December 2020.

[5] risclite. Superscalar-riscv-cpu, 2018. URL https://github.com/risclite/SuperScalar-RISCV-CPU. Accessed on October 2020.

[6] airin711: Verilog caches, April 2016. URL https://github.com/airin711/Verilog-caches. Accessed on December 2020.

[7] prasadp4009: 2-way-Set-Associative-Cache-Controller, March 2016. URL https://github.com/prasadp4009/2-way-Set-Associative-Cache-Controller. Accessed on December 2020.

[8] Poc - Pile-of-Cores, . URL https://github.com/VLSI-EDA/PoC. Accessed on December 2020.

[9] A. O. Guilherme Arros, José Monteiro. *Arquitectura de Computadores – dos Sistemas Digitais aos Microprocessadores*. IST Press, 2nd edition, July 2009. ISBN 978-972-8469-54-2.

[10] G. Damien. Study of Different Cache Line Replacement Algorithms in Embedded Systems. Master's thesis, KHT - Royal Institute of Technology in Stockholm, March 2007. URL https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf. Accessed on October 2020.

[11] Recommended hdl coding styles, quartus ii 9.1 handbook, volume 1, November 2009. URL http://www.gstitt.ece.ufl.edu/courses/spring10/eel4712/lectures/vhdl/qts_qii51007.pdf. Accessed on October 2020.

[12] Intel Quartus Prime Standard Edition User Guide: Design Recommendations, . URL https://www.intel.com/content/www/us/en/programmable/documentation/ntt1529445293791.html#mwh1409959579917. Accessed on October 2020.

[13] IOb-memories, December 2020. URL https://github.com/IObundle/iob-mem. Accessed on December 2020.

[14] *UG1037: Vivado Design Suite - AXI Reference Guide*. Xilinx, July 2017. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. Accessed on December 2020.

[15] IOb-interconnect, December 2020. URL https://github.com/IObundle/iob-interconnect. Accessed on December 2020.

[16] C. Wolf and et. al. PicoRV32 - A Size-Optimized RISC-V CPU, July 2020. URL https://github.com/cliffordwolf/picorv32. Accessed on September 2020.

[17] IOb-PicoRV32, December 2020. URL https://github.com/IObundle/iob-picorv32. Accessed on December 2020.

[18] IOb-SSRV, September 2020. URL https://github.com/IObundle/iob-ssrv. Accessed on October 2020.

[19] IOb-UART, December 2020. URL https://github.com/IObundle/iob-uart. Accessed on December 2020.

[20] *Kintex UltraScale KU040 Development Board*. Avnet, Inc., 2015. URL https://www.avnet.com/opasdata/d120001/medias/docus/13/aes-AES-KU040-DB-G-User-Guide.pdf.

[21] *PG150 - UltraScale Architecture-Based FPGAs Memory IP*. Xillinx, June 2020. URL https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf.

[22] A. R. Weiss. Dhrystone benchmark: History, analysis, "scores" and recommendations. November 2002. URL https://johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf.

[23] M. M. HusseinR. Al-Zoubi, Aleksandar Milenkovic. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. pages 267–272, January 2004. doi: 10.1145/986537.986601. URL https://www.researchgate.net/publication/220996061_Performance_evaluation_of_cache_replacement_policies_for_the_SPEC_CPU2000_benchmark_suite. Accessed on December 2020.

[24] Xilinx Vivado Design Suite - HLx Editions, 2020. URL https://www.xilinx.com/products/design-tools/vivado.html.

[25] *UG573 - UltraScale Architecture Memory Resources*. Xillinx, August 2020. URL https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.