DEGREE PROJECT

Development of a Light Weight L2-Cache Controller

Måns Arildsson

Space Engineering, master's level
2022

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Abstract

An L2 cache is a device that buffers data in fast memory closer to the Central Processing Unit (CPU) in order to deliver its contents with much lower latency than can otherwise be achieved by main memory. This provides a substantial performance increase in many systems as the memory interface is often a bottleneck. The goal of this thesis is to develop a simple L2 cache using VHDL for Cobham Gaisler's open source hardware library GRLIB which currently lacks such a core. The outcome of the thesis is the IP core L2C-Lite which will be released in Febuary of 2022 as an addition to GRLIB. L2C-Lite has been integrated into multiple systems and has provided major performance gains in applications running under linux as well as other benchmarks. In addition, some potential improvements have been identified to further increase the performance of the cache, as well as improve its usability in systems.

List of Figures

List of Tables

Acronyms

I^2C Inter-Integrated Circuit. 7, 9, 38

AHB Advanced High-performance Bus. 2, 7–9, 11, 17, 19, 32, 34–40, 44–46

AI Artificial Intelligence. 4, 39, 40

AMBA Advanced Microcontroller Bus Architecture. 6, 7, 29, 34

APB Advanced Peripheral Bus. 7–9, 38, 45

ASIC Application Specific Integrated Circuit. 4

ATF AMBA Test Framework. 34, 35

AXI Advanced eXtensible Bus. 7, 8, 11, 19, 32, 39, 40

BRAM Block Random Access Memory. 26

CAN Controller Area Network. 9

CISC Complex Instruction Set Computer. 9

CPU Central Processing Unit. 1–3, 41

DDR Double Data Rate. 38–40

EDA Electronic Design Automation. 4

FIFO First In First Out. 44

FPGA Field Programmable Gate Array. 4, 7, 34, 40

FPU Floating Point Unit. 9

GBM Generic Bus Master. 2, 3, 11, 19, 20, 32, 46

GPIO General Purpose I/O. 38

GPL GNU General Public License. 9, 12, 39, 40

GPP General Purpose Processing. 39, 45

GRLIB Gaisler Research Library. 1, 6–10, 16, 26, 35, 37, 38, 44

HDL Hardware Description Language. 4, 5

I/O Input/Output. 7, 17, 28–30, 32, 37–39

IC Integrated Circuit. 1, 4

IP Intellectual Property. 1, 4, 6, 8, 9, 11, 16, 19, 27, 34, 37, 38, 40

ISA Instruction Set Architecture. 8, 9, 37

JTAG Joint Test Action Group. 37, 38, 40

LRU Least Recently Used. 3, 12, 13

MIG Memory Interface Generator. 38, 39

MIPS Mega Instructions Per Second. 41

MSB Most Significant Bit. 28

NoC Network-On-Chip. 39, 40, 45

P&P Plug & Play. 7, 18, 29, 37

PCI Peripheral Component Interconnect. 9, 40

PLRU Pseudo Least Recently Used. 3, 13, 14, 26, 27, 44

R-M-W Read-Modify-Write. 26, 44

RAM Random Access Memory. 2–4, 16, 23, 35, 45

RISC Reduced Instruction Set Computer. 8, 9, 39

SoC System-on-Chip. 1, 2, 4, 6, 9, 12, 35, 37–42

SPI Serial Peripheral Interface. 9, 38, 40

UART Universal Asynchronous Receiver/Transmitter. 7, 9, 38, 40

Contents

1 Introduction

The computing industry has been one of, if not the most innovative industry in the world since the inception of the first computers in the late 1940s and early 1950s. As vacuume tubes were replaced by transistors and Integrated Circuit (IC)s, the transistor count on a typical chip has increased from 1500 to 50 billion, meaning a 33 million-fold increase[16]. Naturally, the increased transistor count has resulted in computer systems being much faster and able to perform tasks that previously were not possible. What has lagged behind the increase in processing power has been the speed of which it takes to access stored data[3]. The lag in memory access latency can starve the CPU which forces it to stop execution until the required data has been fetched. One of the ways this issue is countered is by introducing fast on-chip memory, where data can be buffered and stored for much faster access. This thesis report addresses the specification, design, implementation and verification of such a device, an L2 cache, and also briefly discusses potential improvements that can be done to the design in order to further increase the performance. The goal of the project is to create a simple, but modular IP core that can be used with already existing System-on-Chip (SoC)s to boost their performance. The cache is developed in collaboration with Cobham Gaisler, a company specializing in digital hardware for space applications and will be an addition to the companies open source IP library, GRLIB. The L2 cache developed during this thesis will be available in the upcoming February release of GRLIB under the name "L2C-Lite".

2 Background

2.1 Data Caching

As mentioned in Section 1, the growing disparity between memory access latency and processing speed requires hardware developers to find other solutions to keep a processor fed with data, see Figure 1.

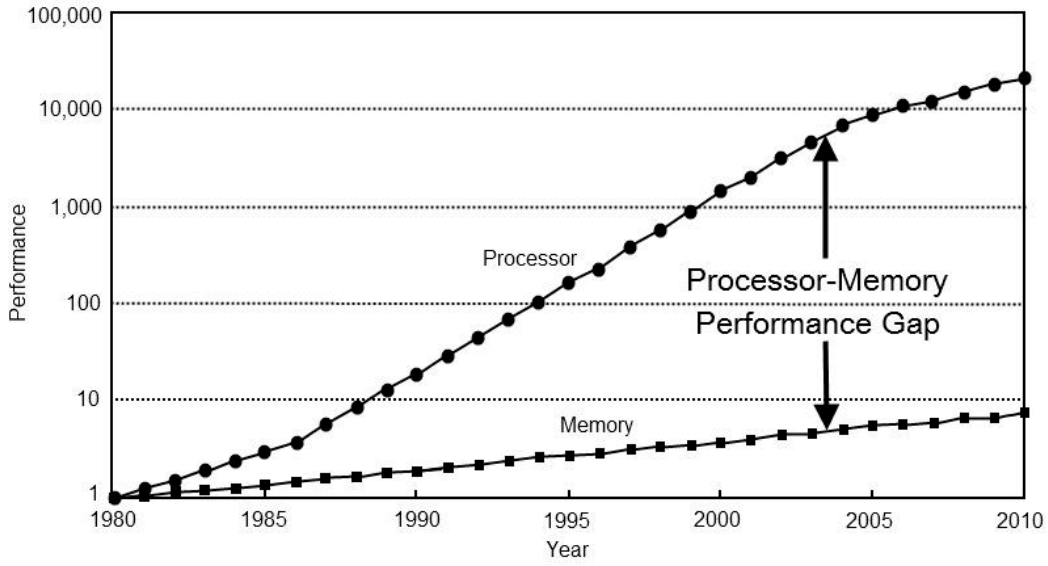

Figure 1: Relative performance gain in processing speed vs memory latency over time. [20] [10]

One way of solving the issue is using smaller, on-chip, memory buffers that make more frequently accessed data available much faster for the CPU. The idea was originally introduced in 1965 by british computer scientist Maurice Wilkes as "Slave memories" in the article "Slave Memories and Dynamic Storage Allocation" [22]. Since then, the concept has evolved and is now a central part of a processing systems architecture, in some cases using significant portions of the on-chip die area. To understand why caches are so effective in increasing the performance of a system, it can be helpful to compare the access latency of different types of memory. Table 1 shows the time required for different memory accesses in a high end consumer PC. It takes 68 times longer fetching data from main memory compared to the L1 cache even with fast and low latency RAM.

Table 1: Approximate access latency on AMD 5900X paired with 3800 MT/s, 15 CAS latency, DDR4, dual channel RAM. Benchmark data from AIDA64 Cache & Memory Benchmark.

Memory access	**Latency**
L1 cache	0.8 ns
L2 cache	2.3 ns
L3 cache	10.2 ns
Main memory	54.5 ns

The increased performance that can be achieved from adding caches to a system relies on the concept "principles of locality". The principles are divided into "spatial" and "temporal" locality and describe how memory typically is accessed[13][11]. Spatial locality states the tendency for programs to access consecutive memory addresses in sequence. As an example, an array allocation in C will return a pointer to the first address and then make sure that the following addresses, up until the end of the array, are free and available for the program to use. Iterating over such an array will thereby yield memory accesses that are adjacent to each other. A cache exploits this fact by fetching larger chunks of data, referred to as cache lines, while serving a specific access, in case the nearby data will be requested in the near future[13]. Temporal locality instead exploits the fact that data is typically reused during the programs execution and it can therefor be useful to keep already used data close and ready for a faster response[11].

Modern caching structures are generally designed to be hierarchical, see Figure 2. The L1 cache is small and usually built into the processor where it is divided into a data and an instruction cache. This cache is private to the core and contain the instructions that are currently being executed and the data that the core is working with. The next cache level, L2, is connected to the L1 cache with some implementation specific internal bus. It is larger and has worse latency than the first level, but is still significantly faster than going directly to main memory. Depending on the system, the L2 cache can be private to a single core, or shared between a cluster of cores, see Section 6.3 for an example of a shared L2 cache. Some systems have L3 and even L4 caches which builds on the hierarchy even further. [10]

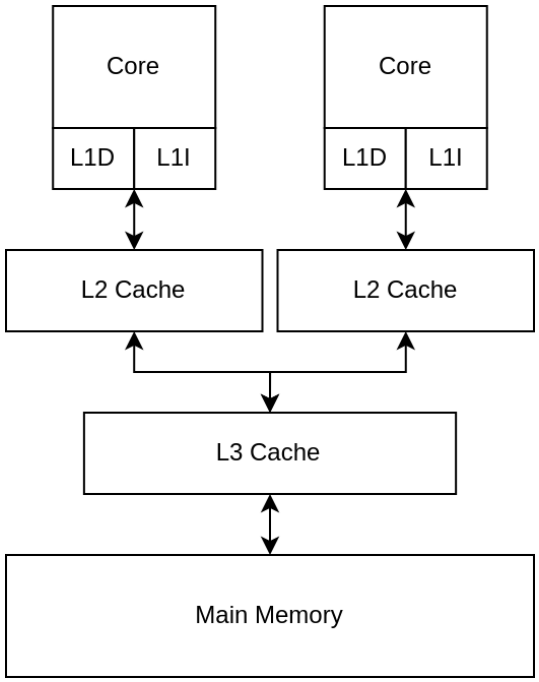

Figure 2: Cache heirachy of a typical, modern CPU.

When the requested data can be found in the cache, it is referred to as a cache hit and the data can be served as soon as the cache is ready. Data not found in the cache is referred to as a cache miss and the request is instead forwarded to either the next cache level or to the main memory.

In order to place and keep track of all cache lines inside the cache, a system that relates the lines to the address of the data is required. There are three ways of doing so, direct mapped, fully associative and way associative caches. A direct mapped cache maps one line to one location in the cache and is the simplest configuration. The fully associative cache allows any cache line to be stored in any part of the cache. Since this requires all locations to be searched to find the data, it is either expensive in logic and power, or dramatically increases the latency of the search. For these reasons, it is rarely used. Instead, a common middle ground is way associative caches, where one cache line can be mapped to multiple locations in the cache. In such a cache, increasing the way associativity will increase the locations where a cache line can be stored and thus, which locations needs to be searched. When a cache line are replaced in a system that maps to more than one location, a replacement strategy is required to select the data that is least likely to be used again in the near future. This is referred to as a "replacement policy" and vary from complicated schemes that record the usage history of the cache lines to simple counters that select which line is replaced randomly. How writing data to the cache is handled can also vary heavily between implementations. This is referred to as "write policy" and the two methods discussed in this report are compared in Section 4.1.

The state-of-the-art research is focused on more complicated replacement and write policies, and is more of an optimization effort in order to get the highest performance, for as little cost as possible. The complexity of these algorithems put them out of the scope for this thesis.

2.2 Cobham Gaisler

Cobham Gaisler was founded as "Gaisler Research" in 2001 by Jiri Gaisler. In 2008 the company was aquired by an American aerospace company Aeroflex[4] which later was aquired by Cobham in 2014[2] where the current name originates from. The company specializes in embedded computer systems designed for harsh environments (safety critical applications) and develops IP cores and custom ASICs. The company also develops software and toolchains to accompany their hardware products such as debug monitors and simulators.

2.3 HDL & FPGAs

A Field Programmable Gate Array (FPGA) is an Integrated Circuit (IC) that features reconfigurable hardware. The devices consist of logic blocks and interconnects that can be arranged in ways creating complex combinatory and sequential logic. This allows a user to implement anything from a full SoC to specialized hardware accelerators for AI, all on the same chip. FPGAs also have built in block-RAM that can be used in designs where larger storage is required. BRAM work as very fast (1 cycle response) RAM chips but have configurable address depth and data width. The flexible nature of the FPGAs make them an extremely useful tool in digital hardware development, where an iterative approach otherwise is prohibitively expensive and time consuming. Because FPGAs are designed to be flexible, they lose out on performance, both in power consumption and timing. Timing is of special concern, as complicated logic in combination with a high clock frequency might lead to wrong or unpredictable behavior. This is due to signal propagation delays through the logic and the interconnects.

Much like when writing software, hardware design is aided by the use of abstraction layers and instead of placing the logic by hand, the use of a Hardware Description Language (HDL) allows the designer to describe higher level functionality. The HDL code can be used by an EDA tool to produce a bitstream, which contains the programming information for an FPGA. The process of producing bitstreams from HDL code is referred to as synthezis and implementation. HDL code can also be simulated, which is where most of development generally takes place. This

allows the developer to have insight into all signals, and how they change during testing and is a powerful tool during debugging. The HDL language used in this project will be VHDL.

3 Target Systems

The cache development targets typical SoC's found in GRLIB. These systems range from simple single core designs with 32-bit interconnects to large multi-core SoC's with rich interfacing and wide 128-bit interconnects (see Figure 3). [6]

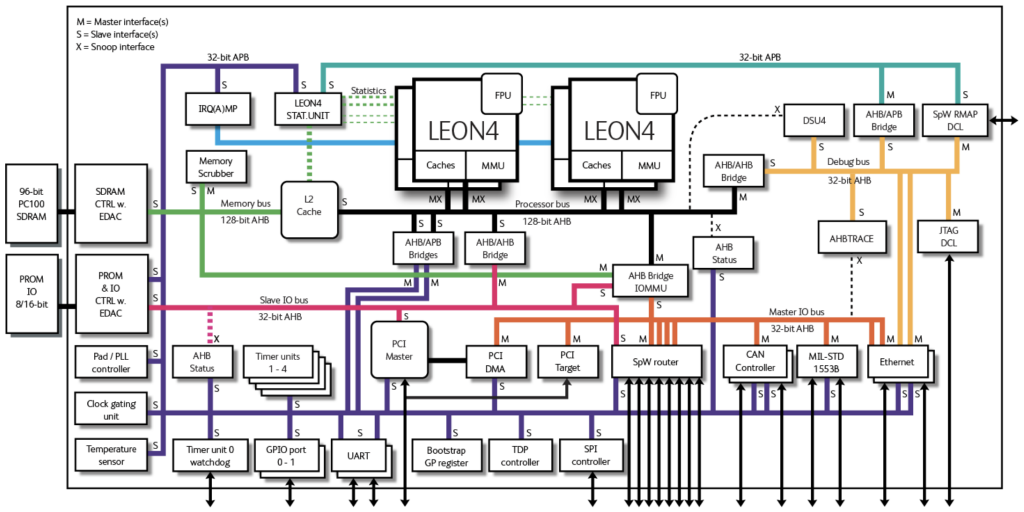

Figure 3: Cobham Gaisler GR740 Rad-hard Quad-core SoC. Source: `https://gaisler.com/index.php/products/components/gr740`

3.1 SoC Interconnects

IP cores developed by Cobham Gaisler generally use AMBA on-chip interconnects to allow for flexibility and reusability between projects. The bus standard is open source and is developed by ARM ltd. The AMBA interconnects are commonly used in SoC's and processing systems as the main interconnects between modules.[14][15] [1]

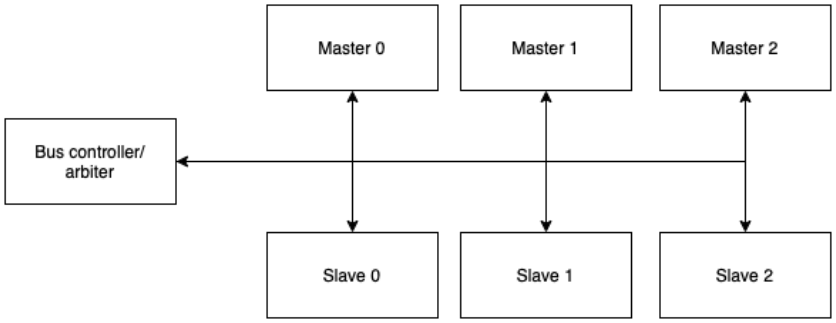

Figure 4: High level illustration of the bus concept.

[1] The functionality described in this section follow the specifications implemented in GRLIB. GRLIB implements AHB & APB specification 2.0 and AXI specification 3.0 and 4.0.

The standard includes three interfaces suitable for different applications; Advanced Peripheral Bus (APB), Advanced High-performance Bus (AHB) and Advanced eXtensible Bus (AXI). The APB bus is less feature rich and is appropriate for low bandwidth devices such as UART controllers, I^2C controllers and timer modules. The AHB and AXI interfaces are developed for high performance applications and the interfaces are typically used between high bandwidth devices such as processor cores, memory controllers and high speed I/O such as ethernet. The GRLIB implementation of the AMBA interconnects has been extended to include, AMBA Plug & Play (P&P), a register that allows masters and slaves to insert their configuration information. This register is accessable during run-time and allows the user to confirm that the correct configuration has been set[9].

GRLIB has only the AHB controller available thus making it the only option for the main bus connected to the core. The library does however have bridges allowing parts of the address space to be bridged to either AXI or APB[7].

3.1.1 Working Principles

All bus types are implemented by multiplexing (see Figure 5), thus avoiding tristate signals. This suits FPGAs well, as tristate behavior is typically synthesized by multiplexing either way.[9]

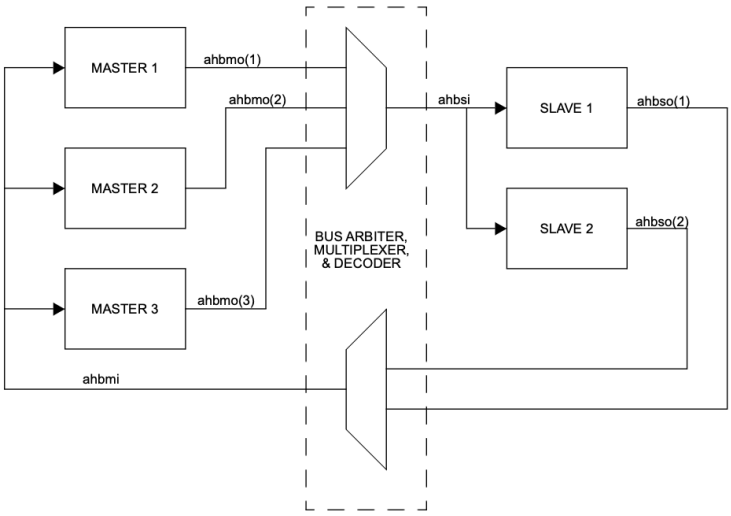

Figure 5: Multiplexed AHB bus illustrated. Source [9]

Each master has a common input record and its own dedicated output record. The records contain the signals required for the specific interface. The same structure is used for the slaves where they share a common input record and has their own dedicated output record. All the signals are routed to the bus controller which selects what master is allowed access to the bus.

The arbitration algorithm is not defined in the AMBA specificaton and is instead decided by the specific implementation. The AHB controller in GRLIB can use either fixed priority or a round robin approach. The fixed priority allows access depending on the master index and the round robin rotates which master has priority each time an access is made to the bus. If no master

is requesting access to the bus, it can be parked to the last bus owner or to a pre-configured master. For further details see implementation[8] and GRLIB manual[9].

3.1.2 Access types

Access types vary between the interfaces. The APB interface only allow simple single accesses. These accesses are as large as the data width of the bus, which can be configured between 8-32 bits. The AHB data bus width can be configured between 32-128 bits and the AXI data bus can be between 32-1024 bits (AXI4 specification). Both the AHB and AXI interfaces can create single and incremental accesses and each access can be from 8 bits to the bus width wide in steps of power of two's.[14][15]

3.1.3 Addressing

Each slave on the bus occupies a specific address range configured by generics. The master requests access to a specific address and the bus controller asserts the appropriate signals to the slave occupying the address range. The addressing vector is 32 bit, making the available address space 4 GiB (2^{32} byte).[14][15]

3.1.4 Split Support

Both AXI and AHB support some form of access split to allow for more efficient bus access. If a slave needs more cycles to generate the access response, the controller can give access to another master during the mean time. The split access is resumed when the slave is able to produce the requested data. This type of access behavior needs to be supported by both the slave and the controller.[14][15]

3.1.5 AMBA summary

A short summary of the main features of each AMBA bus type can be seen in Figure 2.

Table 2: Specification summary [14][15]

AMBA interface	**APB** (Spec 2.0)	**AHB** (Spec 2.0)	**AXI** (Spec 4.0)
Address width	32 [bits]	32 [bits]	32 [bits]
Data bus width	8/16/32 [bits]	32/64/128 [bits]	32/64/128/256/512/1024 [bits]
Access types	Single R/W	Single/Incremental R/W	Single/Incremental R/W
Access size	Data bus width	8 - Data bus width [bits]	8 - Data bus width [bits]
Split Support	No	Yes	Yes

3.2 Processor Cores

Two types of processing cores are available in GRLIB. The LEON cores are based on the SPARC v8 Instruction Set Architecture (ISA) and the NOEL-V cores are based on the relatively young instruction set, RISC-V. Both ISAs are open source, can be implemented free of charge and are RISC architectures. SPARC does however use big endian while RISC-V uses little endian, forcing GRLIB IP cores to support both endianesses if they are to be compatible with both processors. SPARC and RISC-V are both load-store architectures, meaning that the instructions never work directly on memory addresses. Instead they load the memory contents into an internal register using one instruction, manipulate the data and then stores the data back into memory with another instruction.[19][18]

3.2.1 RISC

A RISC ISA uses simple and shorter instructions to perform a processing task. The simpler instructions usually take less cycles to perform and can therefor compete, and in some cases, outperform the more complicated CISC instructions even though more instructions are required for the same operation. Having less complicated instructions can be favorable when implementing hardware, as the CISC systems require large amounts of dedicated logic to implement specific, not always useful, instructions that bloat the architecture. Chips based on RISC ISAs have for a long time dominated the mobile computing industry because of the often superior power consumption of the chips. It has recently gained traction in the consumer and enterprise markets as the performance gap between the traditionally dominating CISC chips and the RISC chips have become smaller. [17]

3.2.2 LEON

Within GRLIB there are multiple iterations of the LEON processor core. Each version iterates on the micro architecture, improving the pipeline, adding branch predictors and other features boosting performance substantially. The cores are configurable and can be synthesized with or without support for features such as FPUs, L1 caches and instruction trace buffers, although some features can only be found in the non-GPL version of GRLIB. All LEON cores are based on SPARC v8 ISA and are 32-bit architectures.[7]

3.2.3 NOEL-V

NOEL-V is Cobham Gaislers latest processor development. The core can be synthesized as one of 7 configurations, supporting different RISC-V ISA extentions. As with the LEON cores, the more advanced and high performance features of the core does not fall under the GPL license of GRLIB and can therefor not be used for free. NOEL-V high performance configurations have similar performance to the latest LEON (LEON5) cores.[7]

3.3 Debug Units

A debug unit is typically connected to the main AHB bus and can be interacted with using Cobham Gaisler debug monitor, GRMON. The debug unit allows for instruction and bus tracing, hardware breakpoints, stalling the processor, single step operation and internal register overview.[7]

3.4 I/O

GRLIB provides IP cores for all the common interfaces used within the space industry. This includes CAN, I^2C, PCI, UART, SPI, ethernet, space wire and more. It also provides modules for adding GPIO, timers, memory controllers, interrupt handlers, hardware watchdogs and other common functionality used in SoCs. The modules are usually connected to the system as a slave on either the AHB bus or the APB bus, making them accessable to the core via read and write accesses on the bus.[7]

4 Cache Specification & Design

The development process begins with creating a technical specification in collaboration with Cobham Gaisler, see Appendix A. The technical specification describes all functionality that the cache should be able to perform and to what degree it should be configurable to.

4.1 Specification

4.1.1 Design Method

To conform with the rest of GRLIB the L2 cache will be written in VHDL and use the two process design method. The two process method allows for writing well structured, more readable and easily maintainable code.

The method divides the sequential logic (registers) into one process and the combinational logic (asynchronous) into another process and uses a record type, r in Figure 6, to contain the current state.

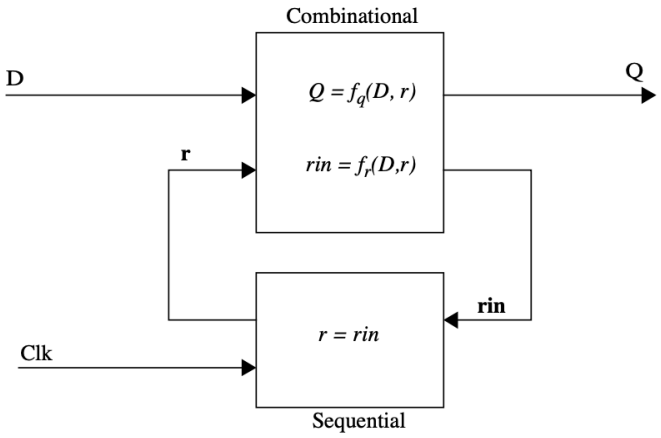

Figure 6: Generic two process circuit.

The record r is updated with r_{in} at rising edge of the clock cycle inside of the sequential process which triggers the combinational process. In the combinational process, r is temporarily stored into a variable, typically called v, where all state specific operations are stored. At the end of the process, outputs are assigned and r_{in} is assigned v, thus saving the new state. Another positive effect of using the r record type to save the state is the ease of adding new registers. Instead of declaring a new signal, adding it to the sensitivity list and driving the signal, it can simply be added to the record.

Below is an example of the two process method using a state machine to increment a counter, until it reaches 100, then changing the state where it is reset to zero in the coming cycle. Although a contrived example, one can imagine replacing the logic within the state machine to something more useful and complex.

```
-- TWO PROCESS METHOD EXAMPLE --
architecture rtl of example is
    type reg_type is record
        state_machine : state_machine_t;
        counter : integer range 0 to 100;
    end record;
    signal r, rin      : reg_type;
begin
    comb : process (r)
    begin
        v := r;
        ---- STATE MACHINE ----
        case r.state_machine is
            when STATE_1 =>
                v.counter := r.counter + 1;
                if v.counter = 100 then
                    v.state_machine := STATE_2;
                end if;
            when STATE_2 =>
                v.counter := 0;
                v.state_machine := STATE_1;
            when others =>
        end case;
        --------  OUTPUTS  --------
        external_output <= v.counter;
        rin <= v;
    end process;

    regs : process (clk, rstn)
    begin
        ---- ASYNC RESET ----
        if rstn = '0' then
            r.counter <= 0;
        elsif rising_edge(clk) then
            r <= rin;
        end if;
    end process;
end rtl;
```

4.1.2 Interfacing

To create a portable and reusable L2 cache it was determined that it should act as a bridge between the main AHB bus and a secondary, either AHB or AXI bus, containing the memory controller of the device that should be cached. A typical bridge has a slave interface on the frontend, accepting requests from masters on the bus and a master interface on the backend, generating the appropriate requests on the backend bus. The frontend of the cache controller can be hardcoded to work with AHB simplifying the development somewhat while the backend should be interchangeable between the interfaces. The backend flexibility will be achived using the Cobham Gaisler IP core Generic Bus Master (GBM).

4.1.3 Cache Configuration

Development effort should be put into making the cache as configurable and scalable as possible. This is to ensure that the cache can be used in both high and low end systems. The cache configuration is set to at least match the non-GPL L2 cache in size and flexibility. This meant 1-4 ways, each way between 1-512 KiB in size with line sizes either 32 or 64 bytes. This had proven to give Cobham Gaisler SoCs major performance gains previously and is a reasonable minimum requirement for the development.

The minimum specification would allow for a maximum cache size of 2 MiB, which is the size used in the latest rad-hard SoC developed by Cobham Gaisler, the GR740, see Figure 3.

4.1.4 Replacement Policy

Selecting the replacement policies that should be available for the cache will depend on other design choices. The considerations that need to be made are mostly related to overhead required for the algorithm and implementation complexity. A common replacement strategy is called Least Recently Used (LRU) [10]. As the name implies, the cache line that was least recently used is replaced, exploiting the temporal locality of memory accesses, see example of LRU on an 8-way associative cache in Figure 7. LRU has high performance in most programs, but can require large amounts of overhead, when scaling the caches associativity.

Figure 7: 8-way associative LRU scheme.

The LRU overhead for one index can be calculated with:

$$\text{overhead } [bits] = w * log_2(w), \tag{1}$$

where w is the way count in the set. This overhead is perhaps tolerable at lower way associativities but quickly become unreasonably large and this is the reason why LRU will not be used

in the cache.

Because of the growing overhead, other strategies have been developed to approximate the LRU behavior using less overhead, these are often referred to as Pseudo Least Recently Used (PLRU). While there are many different PLRU schemes one of the most common algorithms is tree based PLRU. The replacement policy can be shown to perform very similar and in some instances better than regular LRU while scaling significantly better[1]. Tree PLRU scales according to:

$$\text{overhead } [bits] = w - 1, \tag{2}$$

and a comparison between the two replacement policies can be seen in Table 3.

Table 3: Overhead comparison between Least Recently Used & Pseudo Least Recently Used

Way	**LRU** [bits]	**t-PLRU** [bits]
1	0	0
2	2	1
4	8	3
8	24	7
16	64	15

The t-PLRU works as a binary tree, where each level points you in the direction of the latest access. Each node in the tree is either 1 or 0, which represent a direction to traverse the tree. Following the nodes to the bottom of the tree gets you to the latest access, i.e. the most recently used block. Going the other direction, is what gets you to the "least recently used" cache line, that would be evicted. This cache line can not be guaranteed to be the least recently used, but it approximates the behavior well. An illustration of the tree being traversed can be seen in Figure 8 where the red arrows show the "PLRU" block and the green arrows show the most recently used block.

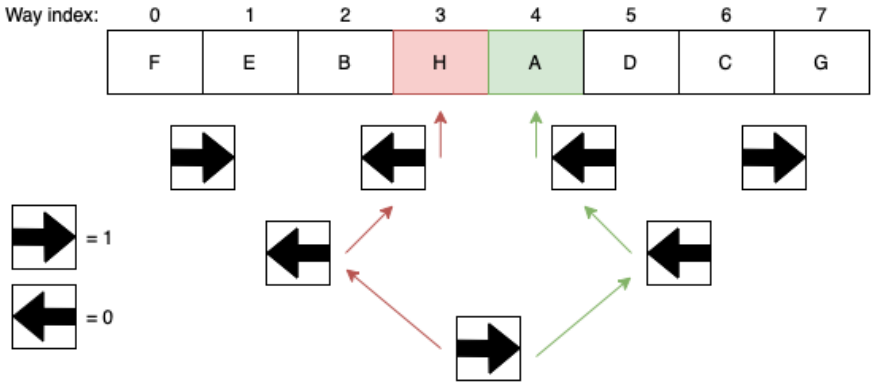

Figure 8: Illustration of the PLRU algorithm in 8-way associative cache.

When an access is made for data that is not present in the cache, first a cache line is selected for eviction(red block). This is done by traversing in the opposite direction of the arrows. Then a new cache line is fetched and replaces the evicted block. To update the access history, the

affected arrows (bits) are changed to point towards the new cache line. An example of this is shown in Figure 9.

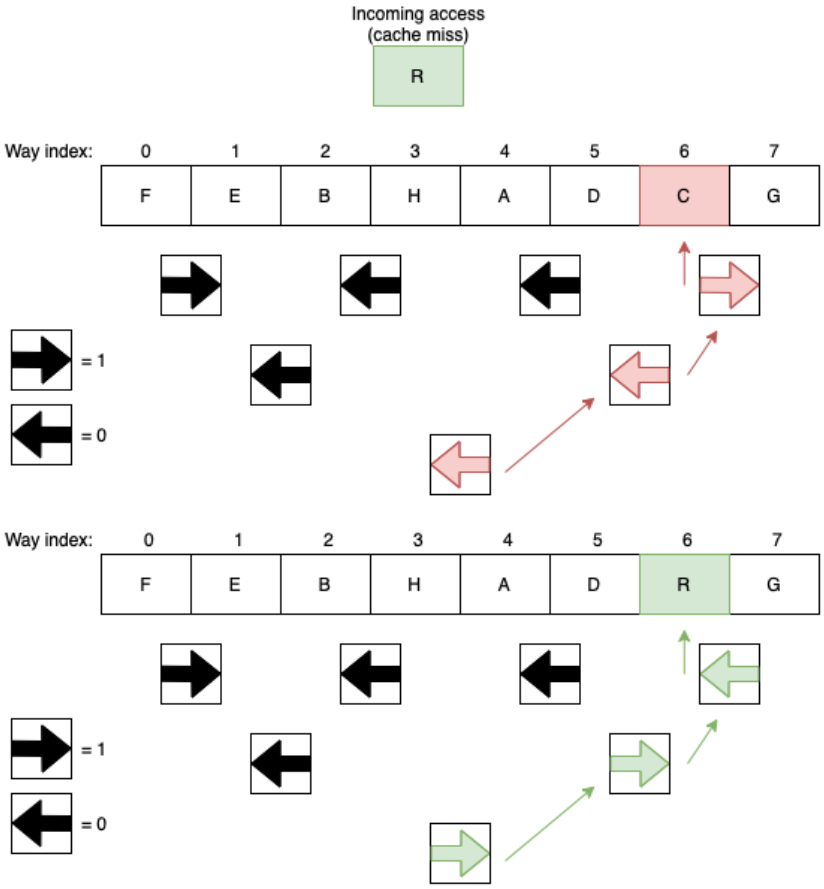

Figure 9: PLRU eviction sequence in 8-way associative cache.

After the new R block has been inserted into the cache, we can now move against the arrows and see that block H is up for eviction next.

To allow for better scalability, t-PLRU is therefor chosen as the main replacement policy for the cache. In most workloads PLRU schemes have good performance. There are however instances where a random replacement policy would outperform it. A random replacement policy evicts cache lines based on an incrementing counter, and would replace a filled block even if there are others that are empty. Since this is very simple to implement, and can be useful in some instances, it will also be an option in addition to t-PLRU.

4.1.5 Write Policy

There are many write policies to choose from, each with their own advantages and disadvantages, but two very common methods are called copy-back and write-through with allocation. Assuming copy-back policy, if a write access is made and the data is not found in the cache, the line is fetched and stored in the cache. The line is then modified according to the access and is

kept inside the cache, now out of sync with main memory. The memory stays out of sync until the modified line gets evicted, where it is written back to main memory. This makes it possible to aggregate multiple write operations before having to interact with the slow main memory. To keep track whether or not the cache is out of sync with the main memory, a single bit per cache line is stored, called the dirty bit. The asynchronous nature of copy-back makes it unsuitable to use if more than one L2 cache is used in the system, as coherency between the caches would have to be handled separately.

Write-through solves the coherency issue by simply writing the data directly into memory. This puts significantly more load on the backend memory interface and can also stall the cache pipeline and thereby impact the performance. In both methods allocation can be made at write accesses, meaning the cache line is fetched when a write access is made, the same behavior as with read accesses. This has shown to improve performance in almost all applications, once again exploiting the temporal locality of memory accesses. [12]

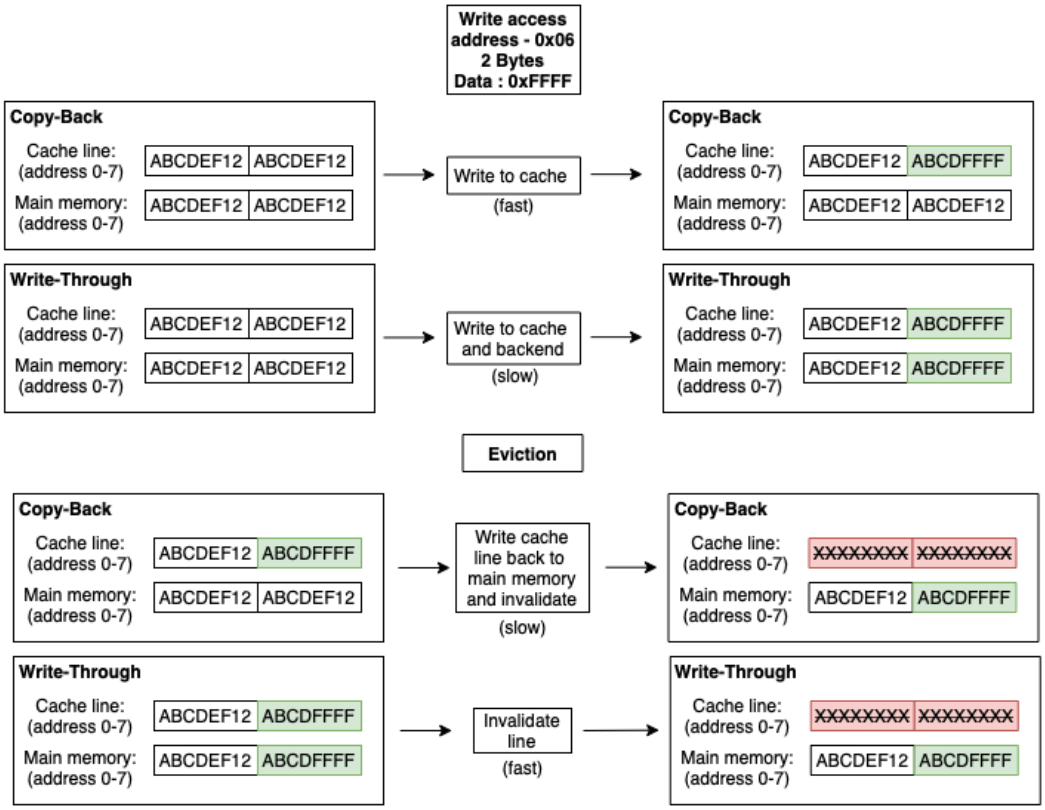

Figure 10: Behavior of copy-back and write-through write policies during write access hit & eviction of cache line.

4.1.6 Cachablility

In certain applications it can be useful to be able to exclude some parts of the cache address space from being cached. The cache would in those cases act only as a bridge between the

master making the access and the memory controller on the backend bus.

4.1.7 Endianess

Since the cache will have to support both the LEON and NOEL-V cores, the system will have to be bi-endian, i.e. support both little and big endian. This should be controlled via a VHDL generic.

4.1.8 Performance Counters

Some performance counters can be useful, to measure whether or not the current cache configuration is working well or needs to be iterated upon during testing. Therefore a hit, miss and access counter will be implemented. This can also be useful during implementation of the device, to see if accesses are handled correctly.

4.1.9 Specification Summary

In addition to the aforementioned specifications, it was also determined that it would be favorable if the cache utilized the GRLIB IP core "syncram" to create the RAM instances. The syncram core maps technology specific BRAM blocks to the required data width and address lengths specified by the user. This allows for efficient BRAM use, independent of the technology in which the core is implemented on to.

Table 4: Cache feature summary

Cache feature	**Configuration**
Cache Configurations	N-ways: 1 - 4 (preferably scalable to N-way) Way size: 1 - 512 KiB Set size: 32 / 64 byte
Interfacing	Front-end: AHB slave Back-end: AHB/AXI master
Replacement Policy	Pseudo-Random Pseudo-LRU
Write Policy	Copy-Back
Endianess	Little and Big endian support
AMBA Plug & Play	Yes
Performance Counters	Hit counter Miss counter Cache access counter

4.2 Design

To keep the development organized, the core is divided into multiple files each responsible for one or more functions within the cache. Two top modules are available depending on the backend interface that is used. The files are identical aside from what generic bus master is instantiated and used. The top file also instantiates the cache control module and the cache memory/logic module, see Figure 11.

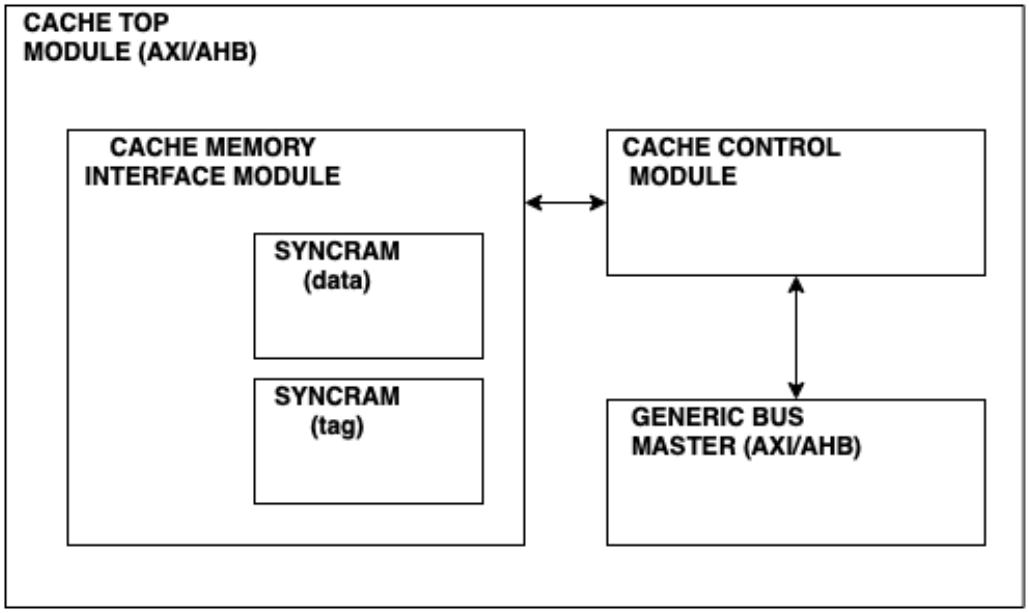

Figure 11: L2 cache file structure.

4.2.1 Frontend Interface

The cache is hardcoded to follow the AHB 2.0 protocol. The relevant interface I/O can be seen in Table 5. [2] The controller deciphers the incoming access and determines which state the cache should move to in order to handle the access. The access details such as address, transfer type, size and write data (if write access) are all buffered in registers to save the information before transitioning to the next state. This also has the added benefit of pipelining the logic, reducing logic delays in the implementation stage.

[2]Other inputs and outputs are described in the bus protocol, but are not used by the cache. They are deliberately left out to avoid confusion.

Table 5: Relevant AHB signals.

Signal Name	Type	Purpose
HSEL	Input, Vector	Slave select vector, if the bit representing the slaves index is asserted the slave is selected.
HADDR	Input, Vector	Access address.
HWRITE	Input, Signal	Mediates whether the access is read or write.
HTRANS	Input, Vector	Mediates the bus transfer status, idle, busy, nonseq and seq.
HSIZE	Input, Vector	Mediates the size of the access, 8, 16, 32, 64 and 128 bits.
HBURST	Input, Vector	Mediates what access, single or burst. Other access types exists, but are not handled by the cache.
HWDATA	Input, Vector	Data supplied during a write accesses.
HREADY	Output, Signal	Asserted when slave responds to access or when slave is ready for a new access.
HRESP	Output, Vector	Mediates slave response to an access, okay, error, retry or split.
HRDATA	Output, Vector	Data vector for responding to read accesses.
HCONFIG	Output, Custom type	Custom, static record type describing the slaves characteristics. Used by the bus controller to configure the address map and the P&P register.

An access is initiated when HSEL is asserted and HTRANS is set to either NONSEQ or SEQ. During the same cycle, HBURST, HWRITE and HSIZE are set to describe the type of access. If the cache is not able to respond to the access the following cycle it de-asserts HREADY to inform the bus controller that more time is required, this is referred to as inserting wait states. The bus controller then simply waits for HREADY to be asserted again together with the access response. This effectively locks the bus and no other accesses can be made during the wait states. See Figure 12 for a single read access of 4 bytes, where the slave inserts 1 wait state before responding with the data.

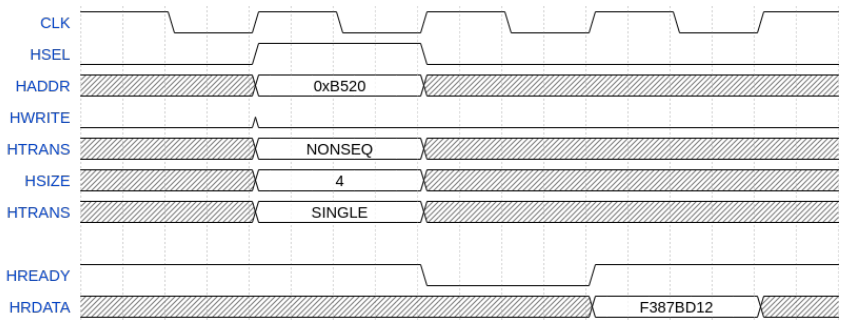

Figure 12: Example of single read transaction on 32-bit bus.

4.2.2 Backend Interface

The backend interface uses Cobham Gaisler IP core Generic Bus Master (GBM). The core defines a simple custom bus interface that is translated to either a AXI or AHB master interface, see Figure 13. This allows the user to develop for a single interface, while still being compatible with both protocols and future development to the generic bus master might also extend the caches compatibility.

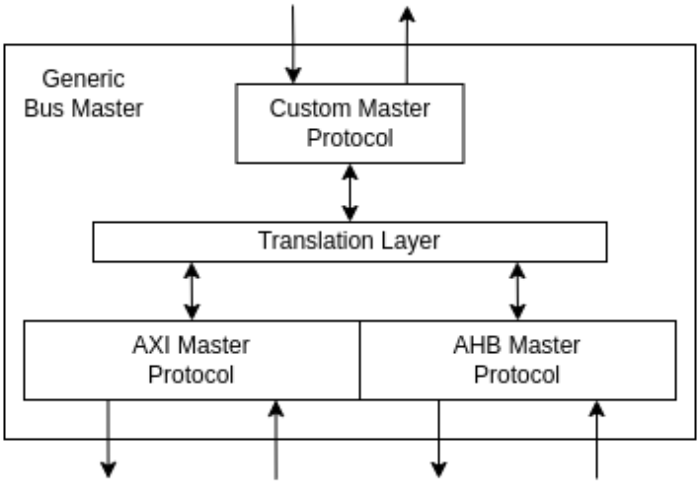

Figure 13: Generic bus master core overview.

GBM provides separate read and write interfaces, see Table 6. A transaction is initiated by asserting the request signal together with the address, transaction size and data, if its a write transaction. Once the GBM core is ready to start the transaction it de-asserts the request granted signal. If the read or write data is to large for the bus master interfaces, it is provided in smaller chunks until all data has been sent. The core uses the BMWR_FULL indicate when the next data chunk should be delivered during writes and BMRD_VALID when the read data on the bus is valid and should be sampled.

Table 6: GBM signals.

Signal Name	Type	Purpose
BMRD_ADDR	Input, Vector	Read access address.
BMRD_SIZE	Input, Vector	Read access size.
BMRD_REQ	Input, Signal	Read access initiation.
BMRD_REQ_G	Output, Signal	Bus master read access acknowledged.
BMRD_DATA	Output, Vector	Read access data.
BMRD_VALID	Output, Signal	Signal asserted when BMRD_DATA contains valid data.
BMRD_DONE	Output, Signal	Signal asserted the when access has finished.
BMWR_ADDR	Input, Vector	Write access address.
BMWR_SIZE	Input, Vector	Write access size
BMWR_REQ	Input, Signal	Write access initiation
BMWR_REQ_G	Output, Signal	Bus master write access acknowledged.
BMWR_DATA	Input, Vector	Write access data.
BMWR_FULL	Output, Signal	Signal asserted when the bus master is ready for more data.
BMRD_DONE	Output, Signal	Signal asserted when the access has finished.

Figure 14 shows an example of a 16 byte write transaction on a 8 byte wide GBM core. The first 8 bytes are delivered together with the address and size and the remaining 8 bytes are delivered the same cycle as BMWR_FULL is de-asserted. The last 8 bytes needs to remain on the bus until BMWR_DONE is asserted.

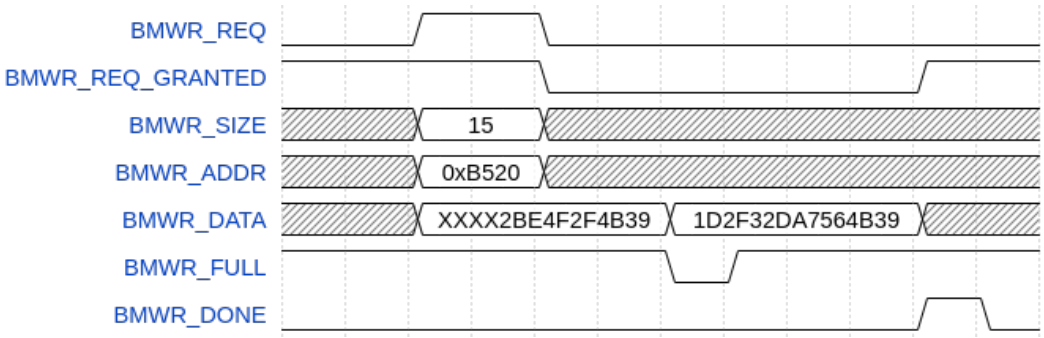

Figure 14: Example of GBM write transaction

4.2.3 Cache State

The cache functional flow is controlled by one main state machine shared between the control module and the cache memory interface. The state machine flow can be seen in Figure 15. The cache waits in IDLE_S until an access is made, all relevant input from the bus is buffered and

the cache moves to either READ_S or WRITE_S. The access address is compared to the tags to check for a cache hit. If there is no cache hit, the cache will move to the BACKEND_READ_S to fetch the cache line, assuming that the backend is not already busy with previous accesses. Once the data has been fetched and stored in the cache, it will move back to the READ/WRITE state where it will now register a cache hit. The cache fulfills the transfer and either moves back to IDLE_S or to the incremental access states which handles the burst accesses.

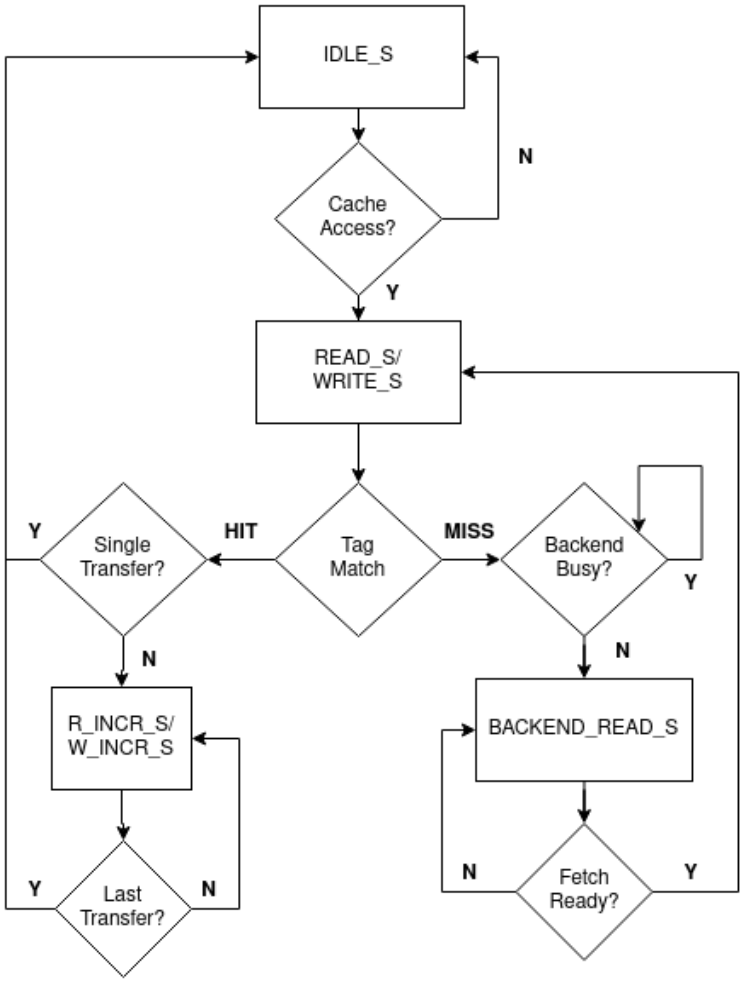

Figure 15: Control state machine flow chart. READ_S, WRITE_S, R_INCR_S and W_INCR_S are separate states, they are grouped in the image to compress the flow chart.

A secondary state machine is built to handle backend writes asynchronous to the main state. This allows for handling some requests while writing to the backend, thus not stalling the whole cache during line evictions. The secondary state machine waits in its own IDLE_S state for a flag from the eviction handling. Once asserted, it checks if the backend is busy and if not it enters the BACKEND_WRITE_S, where it stays until the eviction operation is finished, see Figure 16.

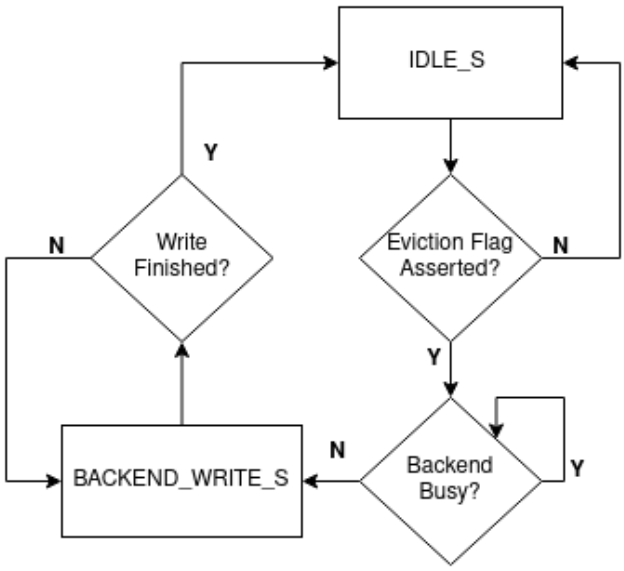

Figure 16: Backend write state machine flow chart.

4.2.4 Tag Matching Logic

To track which data is saved in the cache some identifier of the specific cache line has to be stored along with the data, this is usually referred to as a "tag". A tag is saved with each line and is usually derived from the address of the data in some way. The size of the tag in this implementation depends on the address range, the size of each cache way and the size of the cache lines. To calculate the overhead for one cache line see Equation 3:

$$\text{tag size} \;=\; \text{address range} - log_2\left(\frac{\text{way size}}{\text{line size}}\right) - log_2(\text{line size}). \tag{3}$$

To illustrate further, assume a 32-bit addressing range, 64 KiB way size and 32 byte line size. Since the way size is 64 KiB and each line is 32 byte, there will fit,

$$\frac{64 * 2^{10}}{32} = 2048, \tag{4}$$

unique cache lines. To individually go through the lines a total of $log_2(2048) = 11$ bits are required, these bits are called the "index". Then, in order to go through each cache line byte by byte, another $log_2(32) = 5$ bits are required, these bits are called the "offset". What is left of the address, 16 bits, is a unique identifier that is used to check if the address is cached, i.e. the tag, see Figure 17.

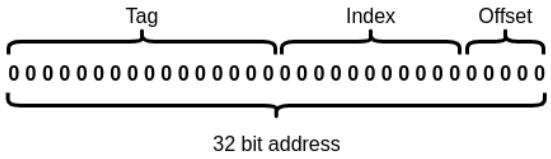

Figure 17: Address sectioned into tag, index and offset.

In the example above, also assuming 4 ways, the total cache size would be 256 KiB and would require 16 KiB of overhead for the tags. Since the index is used to go through all cache lines in memory, two addresses with identical index ranges occupy the same space in RAM. This is tackled by higher associativity where multiple cache lines with the same index but with varying tags can be stored.

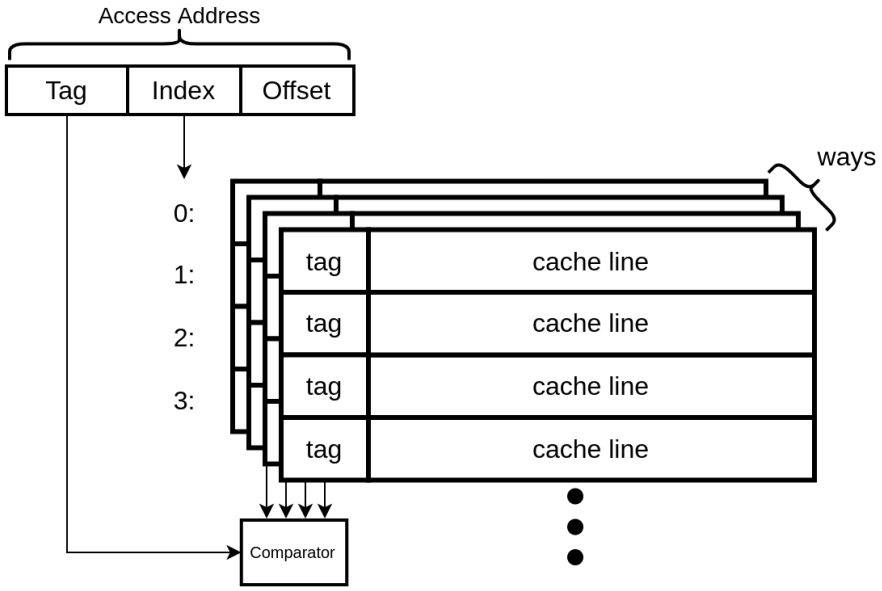

Figure 18: Illustration of multi-way cache.

Central to the caches function, is the tag matching logic required to check whether or not the data is in the cache. Tag matching can be implemented in different ways, depending on what is prioritized, energy efficiency and low logic overhead or latency. This design prioritizes low latency, as it will produce the highest performance gain for the system. The differentiation arise when deciding if the tag look up in all ways should be done in parallel or series. The parallel lookup enables all tag and data RAM during the same cycle where the tag data is compared to the access address and controls a multiplexer that selects between the cache data outputs, see example in Figure 19. The serial lookup would instead enable one tag and data RAM at a time, going through the cache one way at a time. This would consume many cycles as the associativity grows and is not an option for this implementation.

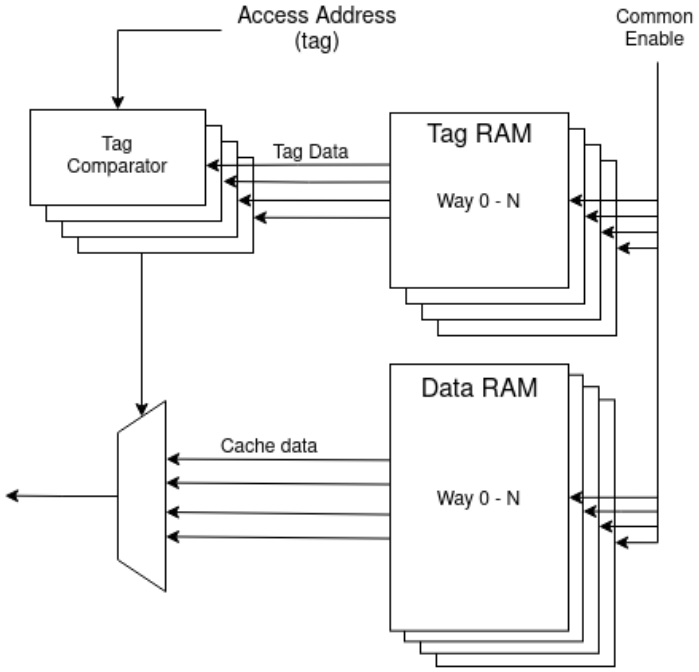

Figure 19: Parallel lookup, multi-way cache.

In addition to the tag for each cache line, two extra bits are stored. The first bit is used to show the validity of the cache line, if the bit is set to zero the tag will not produce a cache hit. This is used at boot when the cache is empty as well as when the cache is flushed. The second bit is used to identify cache lines that have been modified, assuming a copy-back write policy. This tells the cache that a line needs to be evicted before it is replaced, because it is not synchronized with main memory. As previously mentioned, this bit is referred to as the dirty bit.

4.2.5 Read Handling

Once the frontend has identified that the incoming access is a read access, it moves the state machine into the READ_S. In the read state tag lookup is performed to check for a hit in the cache. If a hit on the cache line is registered, the data is sent to a multiplexing stage which selects what part of the cache line needs to be delivered to the bus and the selection process is performed, with the help of the access size and OFFSET region of the access address, see Figure 20. [3] The latency of the read hit is 1 cycle.

[3] Unaligned access are not supported, example: 2 byte access to address 0x5.

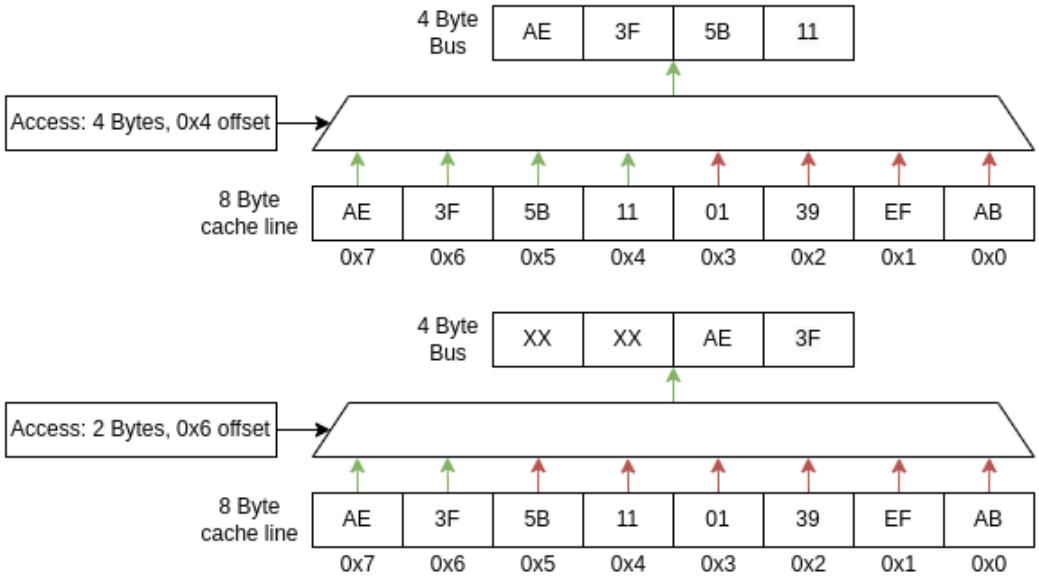

Figure 20: Data transfer between cache storage and the interface bus.

Furthermore, the burst type is checked to see whether or not the cache should move to the burst state or move back to idle, waiting for the next access. If the cache is moved to the burst state, it uses the size of the access and the fetched cache line to continue to deliver data each of the following cycles until the end of the cache line or when the bus terminates the access. If the end of the cache line is reached, the state is automatically moved back to READ_S, where a new tag lookup is performed.

When a tag lookup results in a miss, the replacement policy is used to select which way will be exchanged. Once the way has been selected the dirty bit is checked to see if the cache line needs to be evicted and written back to main memory. If the cache line is dirty, it is buffered to a register while the cache uses the backend to fetch the new line. Once the new line has been cached and the backend is no longer busy, the old line gets written back to memory. Writing back the old cache line is done asynchronously to the main state and allows the cache to accept new accesses during backend writes, see Figure 25.

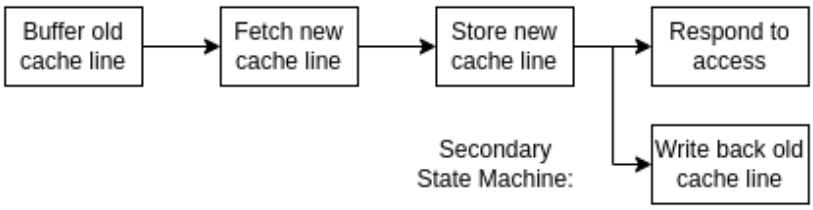

Figure 21: Functional flow when replacing dirty cache line.

4.2.6 Write Handling

In a very similar fashion to the read handling, the main state machine is moved into the WRITE_S and performs a tag lookup. When a hit on the cache line is registered, the whole line is buffered into a register. During the following cycle, the write data from the bus is written to the buffer in the correct position and the full line is then written back to the cache. This sequence is referred to as Read-Modify-Write (R-M-W) and is used because the BRAM does not accept single or multiple byte writes, the full data width needs to be written. Syncram with this feature does exists in GRLIB, but it was realized deep into the development, and is a potential future improvement as the R-M-W sequence adds one cycle of latency.

As with the read access, the cache checks whether or not it should move to the burst state or move back to idle. The burst accesses are written directly to the already buffered cache line, thus avoiding the extra read-out cycle and improving the latency. The cache line is written back to memory once the bus terminates the burst access or when the end of the line is reached, and a new tag lookup is needed in the WRITE_S state.

A write miss is handled in the same way as the read miss, see Section 4.2.5.

4.2.7 Replacement policy

Two replacement policies are implemented and can be selected via generics. The default policy is using a pseudo-random replacement strategy that evicts a random cache line based on a counter that increments each clock cycle. The counter is incremented from 0 to way - 1 and is sampled and buffered at time of eviction. The replacement policy has lower performance than other strategies in most applications, but has the advantage of requiring no overhead and being very simple to implement.

The second replacement policy is, as mentioned in the specification, the tree based Pseudo Least Recently Used. To implement the strategy, two functions are needed. One to update the tree at read and writes accesses, thus setting the most recently used cache line, and one to traverse the tree in the opposite direction, in order to select which cache line to evict. The main implementation issue is mapping each level in the tree two a one dimensional logic vector, see Figure 22. This has to be done as custom types are quite limited in VHDL and the functions need to automatically scale with increasing ways.

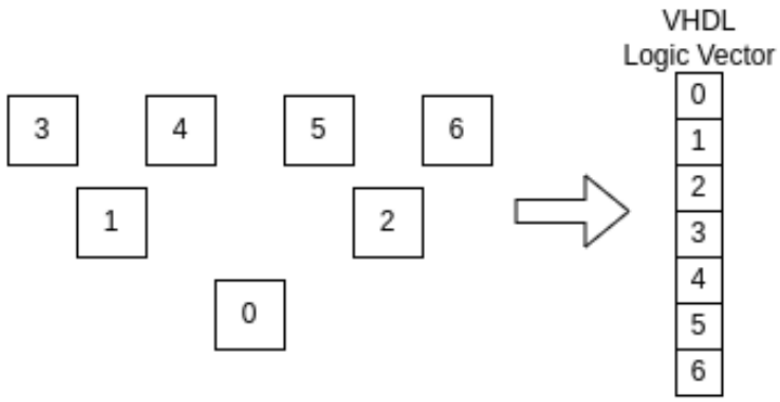

Figure 22: Tree PLRU bits mapped to VHDL vector.

To store all data related to the PLRU algorithm an array with the same length as the cache index is created. Each array element is a vector with the length of "way count" - 1 and will store the tree bits. Moving from a lower level in the tree to the next level, assuming the bits are organized as in Figure 22, can be done by using the following equation:

$$\text{New Index} = \text{Old Index} * 2 + 1 + \text{Direction} \tag{5}$$

To work, moving in the left direction is represented by a zero and moving in the right direction is represented by a one. The equation can be tested by simply following the arrows and calculating the index. The equation does however only work until the highest level is reached, and will not translate to the final way index. Instead, the final way index can be calculated by:

$$\text{Way Index} = 2 * (\text{Old Index} - (\lfloor \text{Way Count}/2 \rfloor - 1)) + \text{Direction} \tag{6}$$

The equations works because division in VHDL can not produce fractions and always truncates the value result[4]. The equations can easily be rearranged to allow traversing the tree in the opposite direction, thus making it possible to update the tree when a cache hit has been registered. An example of the equations and the algorithm in use can be seen in Figure 23.

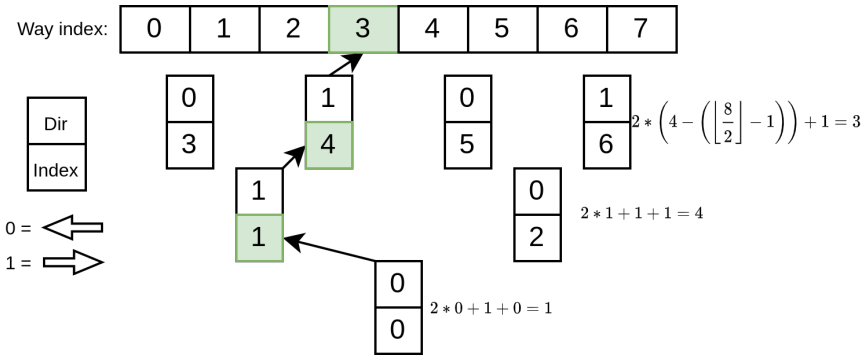

Figure 23: Example of PLRU algorithm in use.

4.2.8 Cacheability

Situations may arise where it is preferable or necessary for the IP core to not cache a certain address range. In these cases the cache would only act as a bridge between the frontend and backend buses and would immediately forward the incoming frontend access. The feature is implemented using a 16-bit mask, controlled via a generic, where each bit in the mask covers a 256 MiB range. Setting the bit to 1 enables caching in that range and a zero disables the caching. For example, the mask 0x00F1 creates the following cacheable area:

$$\text{0x00F1} \longrightarrow \text{0b0000000011110001} \tag{7}$$

[4]Hardware limitation, true fractions can not be synthesized.

Table 7: Cacheability map.

Address Range	Cacheablility
0x00000000 - 0x0FFFFFFF	CACHEABLE
0x10000000 - 0x3FFFFFFF	NOT CACHABLE
0x40000000 - 0x7FFFFFFF	CACHABLE
0x80000000 - 0xFFFFFFFF	NOT CACHABLE

The feature is implemented by typecasting the 4 MSB of the incoming address to an integer and using the integer to index into the mask vector. If the selected bit in the vector is a 1, the address is cacheable. To illustrate, assume the same mask as before, 0x00F1 and an incoming address of 0x6681FC. The top 4 bits in the address is 0x6 which converted to the base 10 integer value 6. Using 6 as the index in the mask vector will return a one which in turn tells the core to cache the access.

4.2.9 Endianess

The main implementation follows a big endian byte sequence. In order to integrate into little endian systems as well, a simple byte swapping function is applied at every read and write access in the cache, see Figure 24. The endianess of the cache is controlled via a generic.

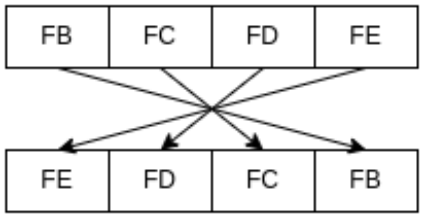

Figure 24: Endianess byte swap.

4.2.10 Cache Flush

Flushing the cache, i.e. evicting and invalidating all cache lines can be initated by writing 0x1 to the internal I/O register with offset 0x00. The main state machine enters the FLUSH_S and begins the flushing sequence. The sequence begins by resetting the way counter and index counter. These are used to cycle through each index and all ways to check for dirty cache lines that needs to be evicted. If a cache line is dirty, it is sent to the backend for eviction and the flushing is stalled until the backend write is finished. If the cache line has not been modified, the validity bit is simply set to zero and the correct counter is incremented. The processes is done until the last index is reached. See state diagram in Figure 25.

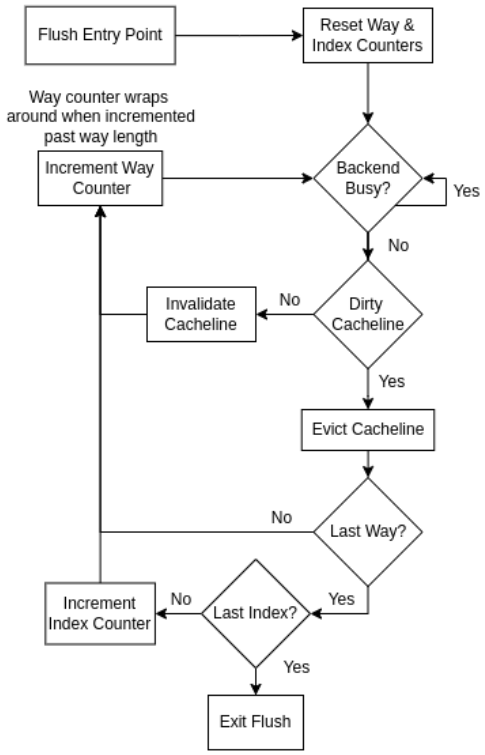

Figure 25: Functional flow of cache flushing feature.

4.2.11 Internal I/O registers

AMBA P&P allows a slave to add a dedicated I/O address range in addition to the memory space. The I/O address range is often used as a way to configure and control the core, for example, enable and disable, read the internal status or turn on a specific feature of the core. The cache implements 4 internal 32-bit registers that allow the users to read the cache configuration, cache performance data and initiate a cache flush. The register descriptions and offset can be seen in Table 8.

Table 8: Internal I/O registers

Offset	Register	R/W
0x00	Writing 0xFFFF to this address will initiate a flush of the cache.	W
0x04	Register contains a 32-bit counter that is incremented at each cache hit. The counter wraps around to zero when it overflows. Counter can be reset by writing 0x0 to the address.	R/W
0x08	Register contains a 32-bit counter that is incremented at each cache miss. The counter wraps around to zero when it overflows. Counter can be reset by writing 0x0 to the address.	R/W
0x0C	Register contains the current cache configuration. The register contains information on the replacement policy, the way count, line size and the way size. How the data should be extracted from the vector can be seen in Figure 26.	R

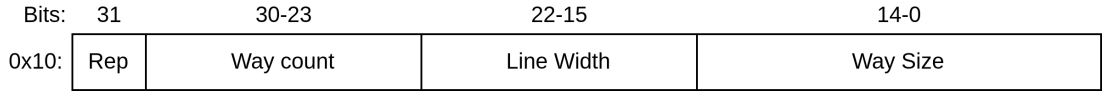

Figure 26: Cache configuration bit description.

The implementation of I/O-registers is completely stand alone from other memory accesses. The I/O can only be accessed with 32-bit read or writes as the registers are hard-coded for the length. The cache identifies an I/O access from the address range while in the idle state. From the full address, it extracts the register offset, and uses it to index into a large vector that contains all 5 registers and it is therefore possible to read and write to all 4 registers, although it should not be done.

If the flush enable register is written to, the main state machine moves into the flush state. Once the flush has finished, the register is reset to 0x0 and the state machine moves back into idle. During this period, the cache freezes the bus.

4.2.12 System Summary

What results from the design described in section 4.2 is a fully functioning L2 cache, compatible with most of the current and next generation SoCs from Cobham Gaisler. The full core can be seen summarized in Figure 27.

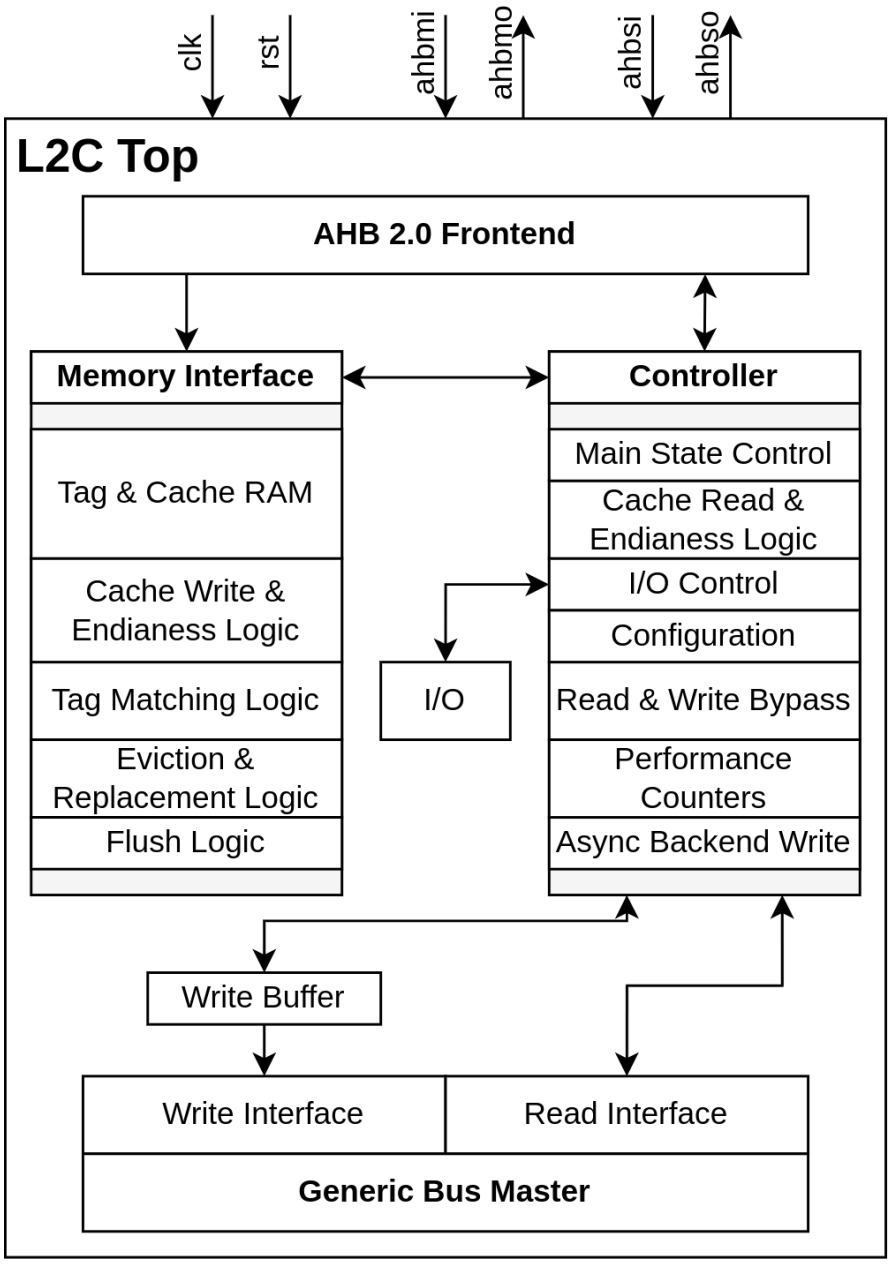

Figure 27: Full system summary.

The signals and generics going in and out of the cache are summarized in Table 9 and 10.

Table 9: Generics summary

Generic Name	**Purpose**
tech	Used to techmap the syncram.
hmindex	Backend master bus index.
hsindex	Frontend slave bus index.
ways	Associativity of cache.
waysize	Size of each cache way.
linesize	Cache line width.
repl	Replacement policy selection.
haddr	Base address of the cache.
hmask	Address range mask.
ioaddr	Base address of I/O registers.
cached	Determines cacheability of address range.
be_dw	Width of GBM bus width.

Table 10: Signals summary

Signal Name	**Type**	**Purpose**
rst	Input, Signal	Cache reset.
clk	Input, Signal	System clock.
ahbsi	Input, Custom Record	Frontend slave input.
ahbso	Output, Custom Record	Frontend slave output.
ahbmi/aximi	Input, Custom Record	Backend master input, either AXI or AHB.
ahbmo/aximo	Output, Custom Record	Backend master output, either AXI or AHB.

The latency of each access type can be seen in Table 11. It is important to consider that read and write misses are mostly dependent on the backend response latency. The response from the memory controller can be anywhere between 10-100 cycles depending on the hardware and makes up the majority of the latency.

Table 11: Latency summary

Access Type	**Latency (cycles per access)**
Single Read Hit	1
Single Write Hit	2
Burst Read Hit	1
Burst Write Hit	(1 + burst length)/burst length
Single Read Miss	4 + backend fetch latency
Single Write Miss	5 + backend fetch latency
Burst Read Miss	(4 + backend fetch latency + burst length)/burst length
Burst Write Miss	(5 + backend fetch latency + burst length)/burst length

5 Verification

Typically when developing hardware, the majority of the testing is done in simulation software such as ModelSim or Vivado[5]. Simulations has the advantage of giving complete insight into every signal within the core and how all modules interact with each other, which can not be done when running on real hardware such as an FPGA. Simulations also has the advantage of often being much faster than synthesizing the design at each design iteration. Synthesizing and implementing the 6-core SELENE design, for example, takes 6 hours on a high end machine, while simulating might vary from a few seconds to a few minutes. To simulate the design, it is important to develop an extensive test bench that covers all the behavior that the core can be exposed to. The test bench developed for this project is largely based on the proprietary AMBA Test Framework (ATF) developed by Cobham Gaisler.

5.1 ATF

ATF is a non-synthesizeable library that implements cores and functions that aid in testing AHB connected IP cores. The main two components of the library is the AMBA test slave and the AMBA test master.

5.1.1 AMBA Test Master

The master is connected to the AHB bus and can be instructed to create accesses to the slaves on the same bus. The access response can be registered and used to confirm the correct behavior of the slave. This allows the developer to incrementally extend what access types the slave is exposed to allowing for a structured development process. The accesses can be made in burst or single, back to back or with delay, to the whole 32-bit address range and target any slave on the bus.

5.1.2 AMBA Test Slave

The slave core acts as mass memory and allows a master to write to it and read from it via the AHB bus interface. Each slave has a debug port associated with it that allows the developer to change the response behavior of the slave as well as read and write content to the memory without creating AHB accesses. The debug port is typically used after a read or write access to confirm correct data transfer between master and slave on the AHB bus. The memory of the slave can also be preloaded with data from external SREC files which can be used to run software or have known data to compare against when reading from the memory.

[5] Vivado: `https://www.xilinx.com/products/design-tools/vivado.html`, ModelSim: `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html`

5.2 L2 Cache Test Bench

The test bench for the L2 cache is developed in parallel to the cache itself. As new features were added to the L2 cache, tests were developed to validate the features. The test system is developed around ATF and is set up to resemble a typical SoC in order to replicate a realistic use case.

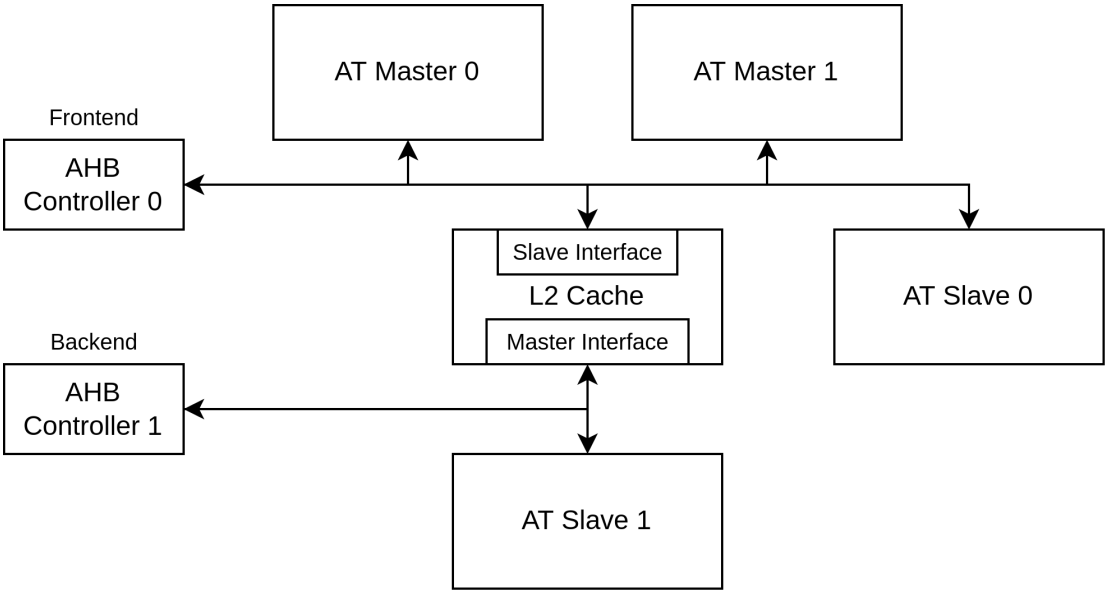

Figure 28: Testbench system overview.

The system consists of 2 AHB buses. The frontend controller connects the two AT masters together with the slave side of the L2 cache and an AT slave. The backend controller connects the L2 cache master interface together with the AT slave which simulates the main memory in a real SoC. Multiple masters are connected to the frontend to simulate a multi-core system, where each core acts as a master. The additional slave on the frontend bus simulates other devices being accessed on the same bus such as an ethernet contoller or a debug unit. The backend bus houses only the L2 cache and the memory, which is typically how the GRLIB SoC's are designed.

The ATF library has been extended with features to allow for easier verification of the L2 cache. The extension mainly added a secondary storage unrelated to the L2 cache, where each write accesses is automatically duplicated to. The storage is instantiated as an array and can be written to like any other signal in VHDL. The secondary memory can be used during reads to confirm the data received on the bus. The debug port on the AT slaves are usually used for these purposes but could not be used as the L2 cache and the slave memory could be out of sync. One of the main drawbacks of the extension is that the full array worth of memory is allocated at the start of the simulation. If the L2 cache is set to cover an address space of 1 GiB, an additional 1 GiB RAM is required for the simulation, which most likely will crash the software. To work around this, much smaller address spaces are used in the test bench. Potential solutions for this issues, could be saving each write access in a dynamic linked list, allocating memory as the simulation moves on, but this was not implemented, as the development effort was deemed to high.

During the early development a handful of tests were made for testing specific features. Separate tests were run to check single write accesses, single read accesses, all burst types and other internal features such as replacement strategy. Deeper into the development a single, more exhaustive and automated test were made to verify the nominal behavior of the L2 cache. The test loops through a function that uses (pseudo)randomly generated numbers to select the characteristics of the upcoming access. This generates thousands of unique accesses in a random order testing most(all) edge cases. If a discrepancy between the data from the bus and the data in the secondary storage is registered, the address, the bus data, the expected data and the time of the event is written out to the simulation console, see Figure 29. The first version of the test bench supported only the 32-bit AHB bus width but was later extended to allow for 128-bit widths as well.

```
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c76 Expected: 0xe66a Read: 0xXXXX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c78 Expected: 0x1d88 Read: 0x88XX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c7a Expected: 0x41b9 Read: 0xXXXX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c7c Expected: 0x2faf Read: 0xafXX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c7e Expected: 0xfb6c Read: 0xXXXX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c80 Expected: 0x01cf Read: 0xcfXX
58555 ns : ERROR. Read data does not match expected, Address: 0x00001c82 Expected: 0xbb8b Read: 0xXXXX
```

Figure 29: Error transcript from simulation run in ModelSim.

6 Integration

Two SoC's were targeted for integration with the L2 cache. The systems vary in complexity, where one is a single core LEON3 design with less I/O and peripherals and the second is a large 6-core NOEL-V design with large bus widths and considerable I/O. Since the SoC's implement both RISC-V and SPARC ISA, it also gives the opportunity to verify the caches endianess handling. Appendix B shows the IP core manual which describes the features of the L2 cache and how to instantiate one in a design. To interact with the SoC's during integration a debug monitor called GRMON3 is used.

6.1 GRMON3

GRMON is a debug monitor developed by Cobham Gaisler which highly integrates with GRLIB IP cores. The debug monitor connects to systems using debug IP cores and acts as a bridge between a PC and the systems main AHB bus. Multiple debug interfaces are available, such as JTAG, UART and ethernet all with varying transfer speeds. Typically, in higher end SoC's such as SELENE (see Section 6.3), the JTAG connection is used initially to configure the Ethernet controller, which is then used as the main debug interface because of its superior transfer speeds.

GRMON is used for all interactions with the system, this includes reading P&P information from the SoC, upload software, running software, observing the internal CPU's state, tracing the AHB bus, instruction tracing and disassembly, manual AHB accesses and more. During integration, reading the P&P registers and manually creating AHB accesses is especially useful, as it can verify changes made to the SoC and test the newly integrated core.

A typical debugging session during the L2 cache development used many of the tools provided by GRMON. The most frequently used features were break points (of all kinds), AHB bus tracing, instruction tracing, memory read and write and memory dumping. AHB bus tracing allows the user to get insight into what accesses are being made and how the L2 cache responds to each access. The trace buffer can be varied in size and is controlled via generics pre-synthesis. What accesses that are traced can be filtered by addressing range, read or write access and by what master. The AHB trace buffers all data related to the access such as read/write address, data, transfer type, transfer size, master index, burst type and slave response. Figure 30 shows a series of single 64-bit write accesses, made by masters 1-5, which in this case are processor cores.

```
grmon3> at
      TIME    ADDRESS  D[127:96] D[95:64] D[63:32] D[31:0]  TYPE  TRANS SIZE BURST MST LOCK RESP HIRQ
    512563887 00022ed8  00000000 00022f20 00000000 00022f20 write   2    3     0    2   0    0   0000
    512563887 00020ed8  00000000 00020f20 00000000 00020f20 write   2    3     0    3   0    0   0000
    512563887 0001eed8  00000000 0001ef20 00000000 0001ef20 write   2    3     0    4   0    0   0000
    512563887 0001ced8  00000000 0001cf20 00000000 0001cf20 write   2    3     0    5   0    0   0000
    512563887 00024ed8  00000000 00024f20 00000000 00024f20 write   2    3     0    1   0    0   0000
    512563887 00022ed8  00000000 00022f20 00000000 00022f20 write   2    3     0    2   0    0   0000
    512563887 00020ed8  00000000 00020f20 00000000 00020f20 write   2    3     0    3   0    0   0000
    512563887 0001eed8  00000000 0001ef20 00000000 0001ef20 write   2    3     0    4   0    0   0000
    512563887 0001ced8  00000000 0001cf20 00000000 0001cf20 write   2    3     0    5   0    0   0000
    512563887 00024ed8  00000000 00024f20 00000000 00024f20 write   2    3     0    1   0    0   0000
```

Figure 30: Example of AHB bus tracing using GRMON.

The break points allow for the user to stop execution when the program counter hits a certain address or when an AHB access to a certain address is made.

6.2 LEON3 Artix-7 SoC

Arty Artix-7 T100 is an FPGA development board developed by Digilent using the Xilinx Artix-7 FPGAs [5]. The board is equipped with an ethernet jack, a USB-UART Bridge via a microUSB, PMOD connectors, switches, buttons and LEDs. The board also features 16 Mb SPI flash and 256 Mb of DDR3 SDRAM that can be used as its main storage, see Figure 31.

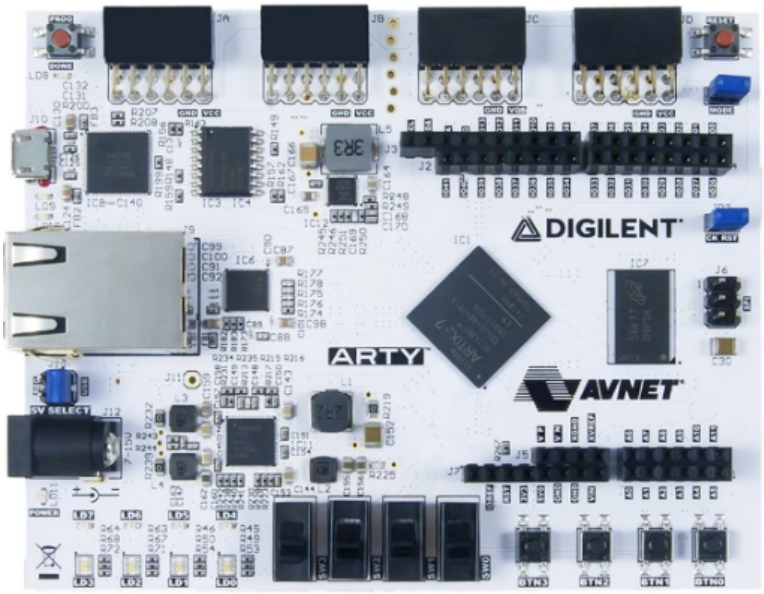

Figure 31: Arty Artix-7 T100 development board. Source: https://digilent.com/shop/arty-a7-artix-7-fpga-development-board/

In GRLIB, an SoC specifically designed and mapped for the Arty board has been made for public use. The design is built around a LEON3 core and a 32-bit wide AHB bus. The design uses the DDR3 memory as mass storage and the SPI flash as a locked down bitstream storage for automatic reconfiguration at power toggle. To use the DDR memory an AHB Memory Interface Generator (MIG) is required as an layer between the bus and the physical DDR module. The MIG is an IP core that can be generated by the Vivado synthesis tool and is specific to the memory type and the bus used. Connected to the AHB bus is also a debug support unit, JTAG debug interface and an AHB to APB bridge. Behind the APB bridge, I/O and peripherals such as I^2C, UART, GPIO, timers and interrupt controllers are connected.

To integrate the L2 cache into the SoC, a new backend AHB controller is added. The DDR MIG is moved to the backend bus and the L2 cache is connected as a bridge between the two buses. The L2 cache slave side is set up to cover the same address range as the MIG did previously and will thereby intercept all accesses that are meant for the main memory· The full SoC with an integrated L2 cache can be seen in Figure 32.

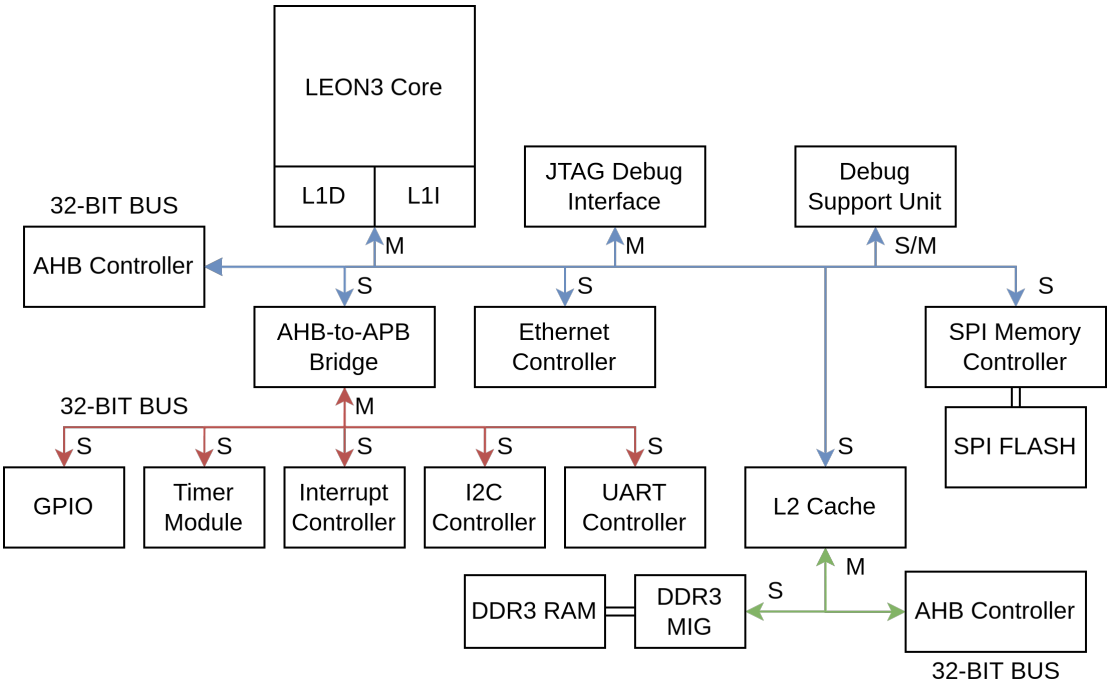

Figure 32: LEON3 SoC overview.

Initial hardware testing is performed on the LEON3 SoC, as the system is simpler to use and debug. Since the SoC is smaller, it also requires less time to synthesize and implement the design. Test software such as CoreMark and Dhrystone were compiled for the SPARC architecture and run on the system to confirm nominal functionality.

6.3 SELENE 6-core NOEL-V SoC

The H2020 SELENE project is a European collaboration between industry and academia developing an open source, multi-purpose, safety critical computing platform. The Cobham Gaisler contribution to the platform is a multi-core RISC-V based SoC, featuring a rich set of high speed I/O such as gigabit Ethernet, multiple Space Wire and CAN FD controllers. The SoC design is built around 3 subsystems, the General Purpose Processing (GPP) subsystem, the memory subsystem and the I/O subsystem. The GPP element instantiates 6 NOEL-V cores connected to a common AHB bus together with either an L2 cache or an AHB-to-AXI bridge, which in turn is connected to the memory subsystem. Since the L2 cache developed by Cobham Gaisler is not released as GPL, the public version of the SoC features the bridge instead, severely impacting performance. The memory subsystem connects the backend of the bridge or L2 cache to a 128-bit AXI bus, referred as the Network-On-Chip (NoC). Also connected to the NoC is a DDR4 MIG (has recently been extended to two MIGs) as well as two dedicated AI hardware accelerators. The I/O subsystem bridges the GPP AHB bus to a secondary AHB bus that is connected to the high speed I/O previously mentioned. See Figure 33 for the full SoC. The system can be set up to run bare metal(no operating system) or boot an embedded linux distrobution (SBI), although Linux without an L2 cache runs very slow.

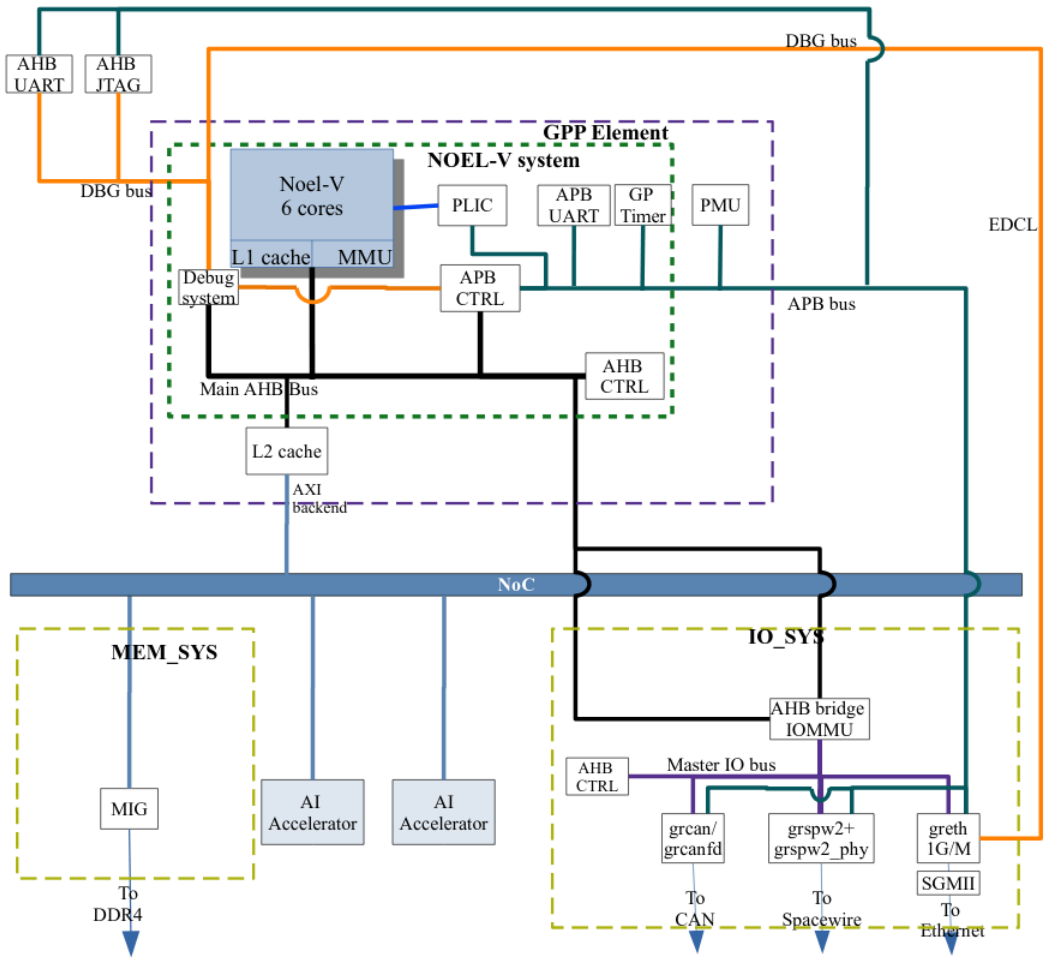

Figure 33: SELENE SoC overview. Source: Internal SELENE git.

Because of the large system, the SoC is implemented onto the high end VCU118 development board, featuring a Virtex UltraScale+ FPGA[23]. In addition to the FPGA, the board has 8 GB DDR4 memory, 1GB SPI flash, PCI express, ethernet, UART, JTAG, etc.

Integrating the GPL L2 cache into the SoC can be done by simply changing out the AHB-to-AXI bridge with the L2 cache IP core. The L2 cache is configured with the same addressing range and master/slave index as the bridge and can now cache the DDR4 memory. To exclude caching the AI accelerator address range, that are also connected to the NoC, the cacheablity mask is configured accordingly. To confirm the nominal functionality of the SoC, Linux is booted from the cached memory, and benchmarks are run to identify the performance increase.

7 Benchmarks & Performance

7.1 Dhrystone

Dhrystone is a synthetic CPU benchmark that was developed by Reinhold P. Weicker. The benchmark has been around since 1984 and aims to test integer performance of processing systems but has the reputation of having some shortcomings. The more serious issues with the benchmark includes small program size (fits in many modern L1 caches), code compilers can easily optimize for it and it tests only a few mathematical and basic operations. One advantage of the dhrystone benchmark is the simplicity of how it reports the score. The benchmark reports DMIPS (Dhrystone-MIPS) which is essentially a measure on how many loops of the benchmarks can be run during 1 second. It is also common practice to divide this number by the CPU frequency to get DMIPS/MHz to also take into account the clock speed of the processor[21].

Especially important to the measurement of the L2 cache performance is the small program size. The LEON3 core features a 8 kB L1 cache, divided equally between instructions and data. This is not enough to fit all dhrystone code or data, which is $\sim$ 60 kB and $\sim$ 16 kB respectivly. It is therefor reasonable to expect some performance increase by the addition of an L2 cache. The NOEL-V core has a 4 times larger L1 cache, 32 kB, also split equally between an instruction and data cache. This is still likely not enough, but should make the difference much less noticeable.

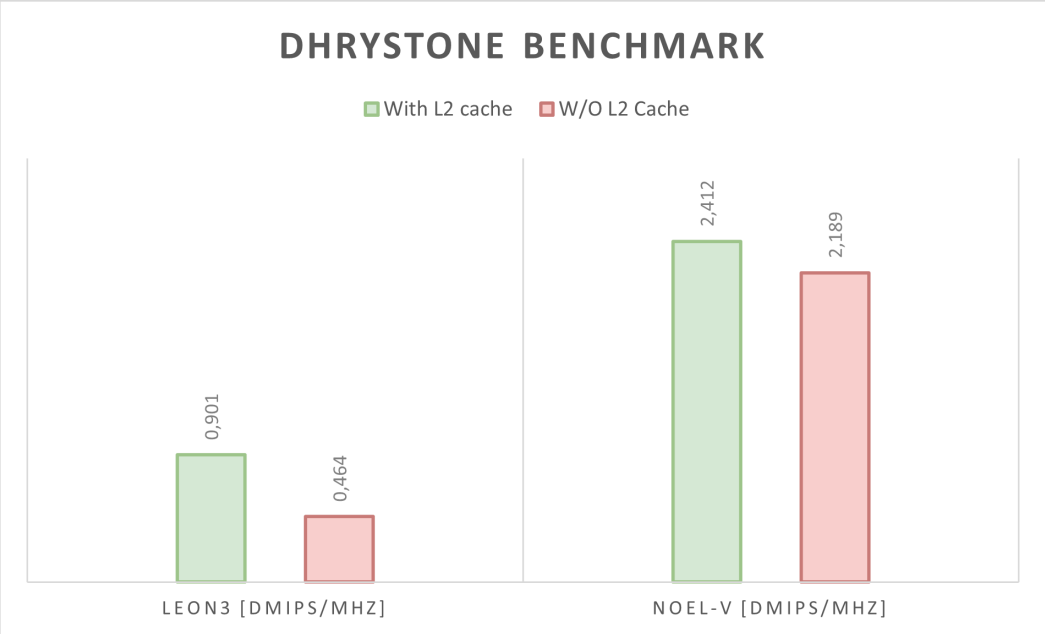

Figure 34: DMIPS/MHz with and w/o L2 cache. LEON3 Artix-7 SoC and NOEL-V VCU118.

As seen in Figure 34, the performance increase seen by adding a cache to the LEON3 system is substantial. An almost 2x increase in performance shows that the system was bottlenecked by the memory interface. Adding an L2 cache to the NOEL-V shows a much smaller increase in performance of 10%. The smaller impact on the NOEL-V system is expected and could be the result of a still to small L1 cache. Some of the gain could perhaps also be accounted for by

the write policy of the L2 cache. Since the L1 cache always uses a write-through policy, to keep coherency between cores, without an L2 cache this would mean writing directly to slow main memory. Instead the L1 cache writes to the L2 cache with much lower latency, which in turn is able to aggregate many writes before evicting back to main memory, thus allowing for much better performance.

7.2 Linux

Linux operating systems comes in many shapes and sizes. The operating systems themselves are generally not used as a benchmark, but can be used to host other, often more exhaustive benchmarks such as the SPEC2006 benchmark suite. Additionally normal operations used within the shell can be benchmarked to measure responsiveness and overall fluidity of the system. Using the SELENE platform discussed in Section 6.3 it is possible to boot a Cobham Gaisler developed linux distribution based on SBI. A shell script has been developed that measures the time it takes to move files around in the file system. The script measures the time it takes to perform the operation and allows the user to change the sizes of the files. Running the script with and w/o an L2 cache yields the result seen in Figure 35.

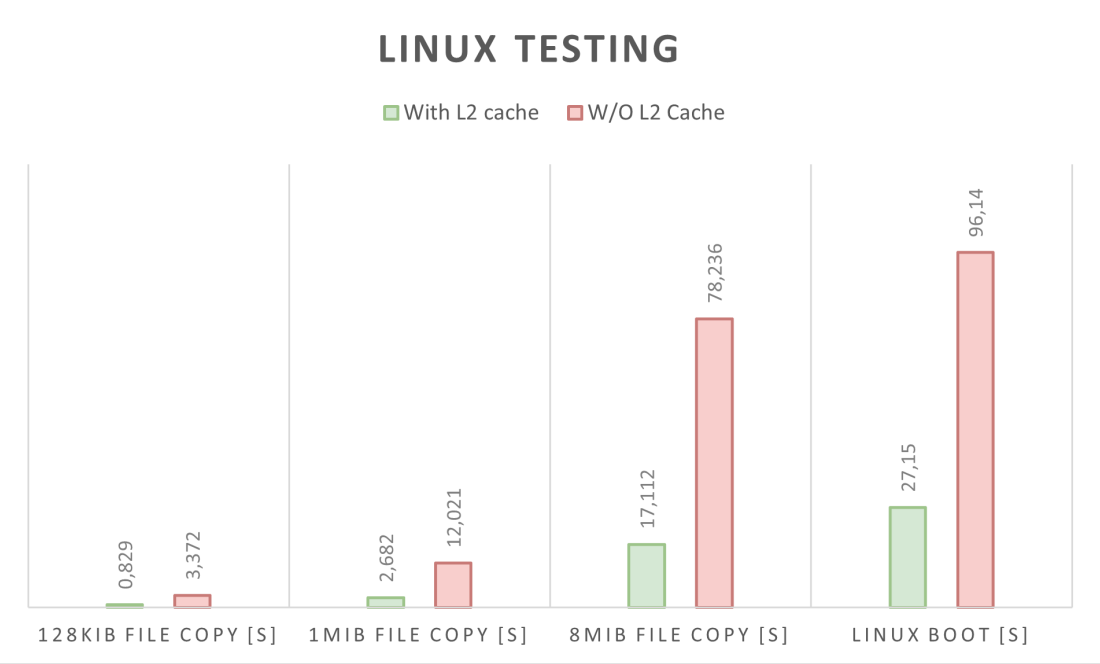

Figure 35: Linux file system benchmark with and w/o L2 cache. H2020 SELENE 6-core SoC.

The performance increase from using an L2 cache in the file system test shows a very large increase in performance. Analyzing what exactly is causing the very large performance increase can be difficult, without going into details on how the file system and linux implements the "cp" command. Further testing using smaller ways, as well as varying the replacement policy of the L2 cache showed very similar results to the ones seen in Figure 35. This suggests that the L2 cache fits all data and code used to move the files and the benefit comes from the much lower latency that is achieved from interacting with the L2 cache instead of main memory. The boot

process is also significantly faster taking only 27 seconds to reach the login shell compared to 96 seconds without an L2 cache.

8 Future Work

8.1 Latency Improvements

The latency of the response from the L2 cache can be improved in a few ways. The most notable improvement would most likely be generated from adding split support to the slave. Split support would allow the L2 cache to inform the bus controller that it is waiting for data from the backend and it can not respond immediately. Instead of locking the bus while waiting for the response, other AHB requests can be fulfilled in the mean time. Splitting the accesses would be implemented only when the L2 cache is making backend reads, as writes are already asynchronous to the frontend, and does not lock up the bus.

The current implementation of handling write accesses also leaves some room for latency improvements. Since the syncram used in the L2 cache does not have byte write functionality a R-M-W sequence is required every time a cache line is modified. Implementations of syncram with byte write does exist within GRLIB and making the change could likely be done without to much development effort. The improvment would remove one cycle at each single write access.

Conceivably, a pipepline could also be constructed around the write accesses, simply acknowledging each request immediately, saving them in a FIFO buffer and writing to the L2 cache asynchronously. The only time the L2 cache would need to either lock the bus or insert split response when handling write accesses would be when the buffer is full. This approach would improve both latency and implementation timing but would also require a major architectural redesign of the L2 cache.

8.2 Timing Optimizations

Improving timing within the core and thus allowing it to be implemented in higher frequency systems can be done by pipelining "logic heavy" segments of the L2 cache. From implementations done in Vivado, some critical paths have been identified and could be the focus of such optimizations. Pipelining the design does often come with a cost to the response latency which needs to be taken into consideration, as it will hurt the performance.

Meeting timing requirements when configuring the cache with 64 byte cache lines or larger has shown to be an issue. This is likely a result of the complex multiplexing scheme required to insert the data from the bus into the cache and vice versa. In order to improve the timing in this regard, it could be reasonable to insert 1 cycle of delay between the tag lookup and the data output to the bus.

Using PLRU can also results in lower timing performance in some configurations. This could be the result of the complex logic required to implement the binary tree traversing in combination with the tag lookup, as they depend on each other. To improve timing in this instance the same solution as previously mentioned could be applied, inserting a 1 cycle delay between the tag lookup and binary tree traversing.

It should be mentioned that what causes the critical path can be hard to exactly identify. The Vivado synthesizing and implementation tool reports what signal paths are of concern, but because of the complicated nature of the implementation optimizations, it is not always clear what is the culprit.

8.3 Coherency

Coherency between multiple L2 caches might be of interest in some systems. The SELENE-platform was originally planned to have clusters of CPUs (multiple GPP-elements) each connected to the NoC through their own L2 cache. This could not be done since the cache currently uses a copy-back write policy and the main memory will be out of sync. If one CPU cluster has an address cached and modified, a secondary cluster accessing the same address from the main memory will receive wrong data.

Solving the coherency issue could be done in multiple ways, but a common implementation is the use of bus snooping. This requires each cache, or perhaps a centralized unit, to monitor the accesses made to all caches and confirm that the requested data is not cached in any other cache. If the data is present in another cache, it is ordered to evict the line before the access is finished, thus keeping coherency. This is the solution used for both LEON and NOEL-V L1 caches, to keep coherency between multiple cores.

8.4 Cache Diagnostics Port

Adding a diagnostics port to the cache might be useful to give software developers more insight into the caches behavior and its interaction with the code. The debug interface could be implemented either as a secondary AHB or APB slave. The interface could allow access to individual ways and cache lines as well as all configuration and performance registers. Disconnecting the debug interface from the main access bus could be useful for more complicated systems that might have a separate debug bus, like the SELENE platform, see Figure 33. If the main bus would hang or stop functioning reliably, diagnostics could still be recovered from the core through the secondary interface.

8.5 Flush Improvments

One improvement to the flushing functionality built into the cache would be to not lock the bus while the flush is ongoing. Instead the frontend could, after the flush is initiated, set the slave as busy and let accesses to other slaves through.

Allowing the user to flush the cache without invalidating all cache lines might also be of use in some circumstances. This too, could quite easily be implemented using very similar logic to what already exists. Flushing without invalidating would simply bring back coherency between the cache and main memory by evicting the data and clearing the dirty bit, while keeping the valid bit set.

8.6 Cache Invalidation

Adding a way to invalidate the entire cache or specific lines without flushing it back to memory might be of use during startup and initialization of the system. If the cache RAM contains random data during boot, it could be tricked into thinking that the data is valid. This can be dealt with by software implementations currently, but invalidating the full cache during initialization would simplify the process. Invalidating the full cache could likely be implemented somewhat effortlessly, as the method of doing so would be the same as with flushing without evictions.

8.7 Frontend Layer

As the design currently stands, the frontend interface is hardcoded to work with the AHB 2.0 slave interface. To increase compatibility and perhaps "future proof" the core, the frontend interface could be replaced with a generic cache interface and have a separate layer which translates the real frontend to the generic cache frontend. In such an implementation, the translation layer can be exchanged to match future bus interfaces and thus not requiring any change to the caches core functionality. The functional principle would resemble that of the GBM, but for a slave interface instead.

8.8 SPEC2006 benchmark

In order to measure the difference between cache configurations, much more thorough and exhaustive benchmarks are needed. The SPEC benchmark suite is generally used for this purpose because it uses a series of test programs derived from real applications. SPEC benchmarks also tend to work with large data sets, putting a larger load on the memory interface which is not only relevant but necessary when performance testing an L2 cache.

Effort was put in to building and running SPECINT2006 (SPEC integer benchmarks) on the SELENE platform but was unfortunately unsuccessful. If the benchmarks was ported to the platform, it would allow for measuring performance difference between the replacement policies, way associativity, way size and cache line sizes more effectively.

References

[1] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite". In: New York, NY, USA: Association for Computing Machinery, 2004. ISBN: 1581138709. DOI: 10.1145/986537.986601. URL: https://doi.org/10.1145/986537.986601.

[2] Businesswire. *Aeroflex To Be Acquired by Cobham plc For Approximately $1.46 Billion*. https://www.businesswire.com/news/home/20140519006808/en/Aeroflex-To-Be-Acquired-by-Cobham-plc-For-Approximately-1.46-Billion. Accessed: 2022-01-27.

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

[4] Design and Reuse. *Aeroflex Incorporated Announces the Purchase of Gaisler Research AB*. https://www.design-reuse.com/news/18670/gaisler-research.html. Accessed: 2022-01-27.

[5] Digilent. *Arty A7 Reference Manual*. https://digilent.com/reference/programmable-logic/arty-a7/reference-manual?redirect=1. Accessed: 2022-02-10.

[6] Cobham Gaisler. *Cobham Gaisler website*. https://www.gaisler.com/. Accessed: 2022-02-10.

[7] Cobham Gaisler. *GRLIB IP Core User's Manual*. https://www.gaisler.com/products/grlib/grip.pdf. Accessed: 2022-01-15.

[8] Cobham Gaisler. *GRLIB IP Library*. https://www.gaisler.com/products/grlib/grlib-gpl-2021.2-b4267.tar.gz. Accessed: 2022-01-15.

[9] Cobham Gaisler. *GRLIB IP Library User's Manual*. https://www.gaisler.com/products/grlib/grlib.pdf. Accessed: 2022-01-15.

[10] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[11] In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Harris. Morgan Kaufmann, 2022. ISBN: 978-0-12-820064-3.

[12] Daranee Hormdee, J.D. Garside, and Steve Furber. "An asynchronous copy-back cache architecture". In: *Microprocessors and Microsystems* 27 (2003). DOI: 10.1016/S0141-9331(03)00101-7.

[13] Bruce Jacob, Spencer W. Ng, and David T. Wang. "Memory Systems - Cache, DRAM, Disk". In: ed. by Bruce Jacob, Spencer W. Ng, and David T. Wang. San Francisco: Morgan Kaufmann, 2008, pp. 57–77. ISBN: 978-0-12-379751-3.

[14] ARM Limited. *AMBA™ Specification (Rev 2.0)*. https://documentation-service.arm.com/static/5f916403f86e16515cdc3d71?token=. Accessed: 2022-01-06.

[15] ARM Limited. *AXI4 Protocol Specification*. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification. Accessed: 2022-01-06.

[16] Hannah Ritchie Max Roser. *Transistor count 1970 - 2020*. https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png. Accessed: 2022-01-25.

[17] David A. Patterson and David R. Ditzel. "The Case for the Reduced Instruction Set Computer". In: *SIGARCH Comput. Archit. News* (1980). DOI: 10.1145/641914.641917. URL: https://doi.org/10.1145/641914.641917.

[18] Inc SPARC International. *The RISC-V Instruction Set Manual*. https://riscv.org/technical/specifications/. Accessed: 2021-12-27.

[19] Inc SPARC International. *The SPARC Architecture Manual*. https://www.gaisler.com/doc/sparcv8.pdf. Accessed: 2021-12-27.

[20] Bill Tran. *A Brief History of Cache*. https://trantriducs.medium.com/a-brief-history-of-cache-5e51826f4873. Accessed: 2022-02-10.

[21] Alan R. Weiss. *Dhrystone Benchmark, History, Analysis, "Scores" and Recommendations*. https://johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf. Accessed: 2022-02-10.

[22] M. V. Wilkes. "Slave Memories and Dynamic Storage Allocation". In: *IEEE Transactions on Electronic Computers* EC-14 (1965). DOI: 10.1109/PGEC.1965.264263.

[23] Xilinx. *VCU118 Evaluation Board*. https://www.xilinx.com/support/documentation/boards_and_kits/vcu118/ug1224-vcu118-eval-bd.pdf. Accessed: 2022-02-10.

Appendix A Technical Specification Open Source L2 Cache

Technical Specification
Open Source L2 Cache

Version 1.2

Måns Arildsson

February 13, 2022

Version History

Version	Date	Modifications	Modified by
1.0	2021-09-05	Document created	M. Arildsson
1.1	2021-09-10	Major part of feature set	M. Arildsson
1.2	2021-09-15	Clean-up	M. Arildsson

Contents

1 Project description

The project aims to develop a lightweight L2 cache for the open source VHDL library GRLIB. Open source hardware has in recent years gained traction and projects such as H2020 SELENE are looking to develop fully open source safety-critical computing platforms. Because the processor cores in GRLIB currently lacks an open L2 cache, considerable performance gains can be achived, even with a somewhat simple cache design. The cache IP core will act as a bridge between an internal AHB bus typically connecting CPU cores and an external AHB/AXI bus connected to slower main memory.

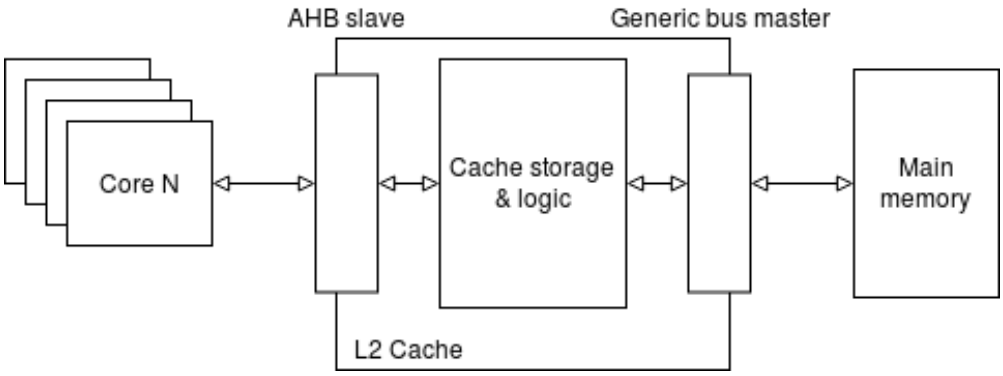

Figure 1: Cache SoC interaction.

1.1 Design principles

During design special care will be taken to keep design complexity down, while still providing valuable features required by modern processing systems. The IP core will be developed in VHDL using the two process method.

1.2 System considerations and constraints

To ensure wide compatibility, the back-end interface will us the "Generic Bus Master" core. This allows the cache to be integrated into systems that uses either AHB or AXI without any major modifications. The front-end slave interface will use AHB as this is the typical protocol in the type of systems that the core will be attached to.

2 Cache Design

Cache feature	Configuration
Cache Configurations	N-ways: 1 - 4 (Should be scalable to N-way) Way size: 1 - 512 KiB Set size: 32 / 64 B
Interfacing	Front-end: AHB slave Back-end: AHB/AXI master
Replacement Policy	Pseudo-Random Pseudo-LRU
Write Policy	Copy-Back
Endianess	Little or Big endian
AMBA Plug & Play	TBD
Debugging	Hit & Miss counter Cache access counter

Table 1: Cache feature summary

2.1 Interfacing

The L2 cache will interface with the CPU and L1 cache using an AHB slave interface. For increased portability, the cache shall use the "Generic Bus Master" IP core as its interface with the main memory. This allows for flexible integration into systems using either the AHB or AXI bus on the main memory interface.

2.2 Cache configuration

The cache will feature configurable line & way sizes as well as a direct-mapped or multi-way associative cache. The cache lines will be configurable as either 32 or 64 Bytes with a way size between 1 - 512 KiB (steps power of 2). The cache will have configurable associativity from 1 to 4 ways(at a minimum) using either pseudo-random or pseudo-LRU replacement policies. Configurations will be determined by VHDL generics.

Configurable parameter	Value	Generic
Line size	32 / 64 Bytes	TBD
Way size	1 - 512 KiB	TBD
N-way associativity	1 - 4 ways	TBD

2.3 Replacement Policy

Naturally the replacement policy is dependent on the cache configuration. A direct-mapped cache configuration uses no special policy, when a cache miss is registered the corresponding line will be evicted and replaced.

In multi-way associative configurations it should be possible to configure with either pseudo-random or pseudo-LRU replacement policies. The pseudo-random policy will simply replace the way depending on a wrap-around counter that is incremented by the clock. Pseudo-LRU or pLRU is a lightweight option to LRU that features a similar performance but with less overhead, particularly for higher way-associative caches. pLRU uses $log_2(N-1)$ bits of overhead per cache line compared to LRU's $log_2(N!)$. An 8-way associative cache requires

Configuration	Replacement Policy	Generic
Direct-mapped	-	TBD
N-way associative	Pseudo-Random Pseudo- Least Recently Used	TBD

2.4 Write Policy

The cache will be using copy-back write policy.
Using the copy-back policy will write data back to the cache during write hits. Only during line eviction does the cache write back the data to main memory and can therefore aggregate many writes into one, off loading the main memory interface. To keep track of which line that has been modified a bit referred to as the "dirty" bit is introduced.

2.5 Cachability

Cachability will be determined by the cache field in the AMBA plug & play information. This will allow the user to set up to four address spaces for the cache and select whether or not they are cacheable.

2.6 Cache flush

Flushing and invalidating the cache will be controlled by writing to the cache control register (see 3). Invalidating the cache clears all "valid" bits in the tag memory. Each line is invalidated regardless if it is out of sync with main memory. Flushing the cache will write the dirty cache lines back to main memory before invalidating the line.

2.7 Endianess

The system shall be compatible with both big and little endianess. Selecting which configuration is done through VHDL generics.

Endianess	Generic
Little Endian	TBD
Big Endian	TBD

2.8 Storage

The cache shall utilize RAM as both tag and data storage. The RAM will be implemented using the SYNCRAM two port core from GRLIB as the it allows for optimizations dependent on the targeted platform.

2.9 Debugging features

The cache will feature an optional debugging unit that can be enabled by VHDL generics. Data from the unit can be extracted by reading debug registers.

The debugging unit will implement the following features:

- Cache utilization
- Hit and Miss rates

3 Cache register layout

This section describes the registers within the cache and their respective functionalities. Access to the registers is done with 32-bit read or write accesses to the I/O bank address + register offset.

Offset 0x00 : Flush (W)
Offset 0x04 : Hit Counter (R/W)
Offset 0x08 : Miss Counter (R/W)
Offset 0x0C : Eviction Counter(R/W)
Offset 0x10 : Cache Configuration(R)

Appendix B L2C-Lite - Level 2 Cache controller

83 L2C-Lite - Level 2 Cache controller

83.1 Overview

L2C-L implements a level-2 cache for processors with an AHB interface. The cache uses the GRLIB core "Generic Bus Master" to act as a bridge from AHB-AHB or AHB-AXI. The cache supports both big & little endian, making it suitable for NOEL-V and LEON designs. The frontend and backend support bus widths between 32 - 128 bits.

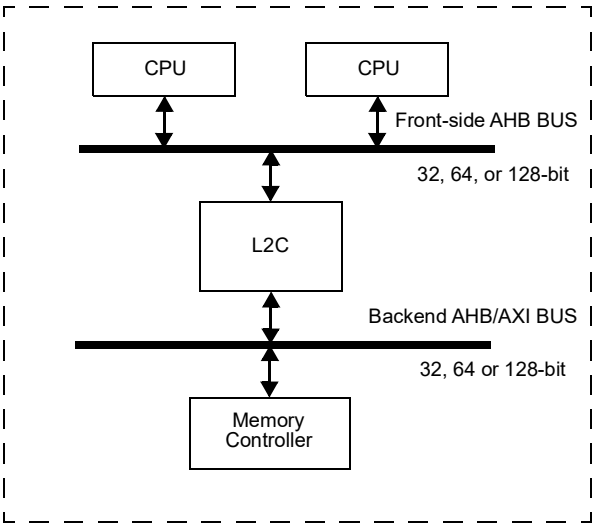

Figure 233. Block diagram

83.2 Configuration

L2C-L can be configured as a direct-mapped or a multi-way associative cache. The cache supports 2 - N way associativity, assuming that the pseudo random replacement policy is selected. Using pLRU replacement policy requires the way associativity to be a power of two, i.e 2, 4, 8, 16 ... 2^N. Cache line size can be configured between 16-256 bytes and the size of each way can be configured between 1-N KiB assuming powers of 2.

83.2.1 Replacement policy

The IP core implements pLRU and pRandom replacement policies. pLRU is a replacement method trying to mimic LRU, but using less memory overhead. The policy has been shown to perform very close to real LRU but requires N-1 bits/index. The overhead benefits become more pronounced at higher way associative caches. The pseudo random replacement selects the cache line to evict based on a counter that is incremented each clock cycle.

83.2.2 Write policy

The cache uses a copy-back write policy. A write access will fetch the associated cache line from memory(unless a cache hit), write the new data to the cache line and set the valid and dirty bits in the tag data. When the cache line is evicted it is written back to memory.

83.2.3 Cachability

The core uses the VHDL generic, “cached”, to determine which address range is cachable. Each bit in the 16-bit value defines the cachability of a 256 Mbyte address block on the AMBA AHB bus. A value of 16#00F3# will thus define cachable areas in 0 - 0x20000000 and 0x40000000 - 0x80000000

83.2.4 AHB address mapping

The AHB slave interface occupies two AHB address ranges. The first AHB memory bar is used for the cache data range and is set up using VHDL generics “haddr” and “hmask”. The second range is used for accessing I/O registers and its address is set up using VHDL generic “ioaddr”. The size of the I/O area is statically set to 64 kB.

83.3 Operation

83.3.1 Read

The read access will start with a tag lookup. If the tag is found and valid, the cache will deliver the requested data the following clock cycle. If the tag is not found, the replacement policy is used to select which line is going to be replaced. The valid and dirty bits are checked to determine if eviction of the cache line is required. If the cache line needs to be evicted, it is buffered and written back to main memory once the new line has been fetched. During the fetch of the new line the cache will insert wait states until the data is ready to be delivered. Writing back evicted cache lines is done asynchronously when the backend is free and allows the cache to handle other accesses in the meantime.For a non-cacheable read access, the cache controller will issue single read accesses to the backend to fetch data from memory. This is done regardless if the frontend access is a single or burst access and may therefor affect the performance.

83.3.2 Write

A write access will start with a tag lookup. Writes to the cache are handled with a read-modify-write sequence, where the cache line is initially read from memory, then modified and written back to memory the following cycle. because of the read-modify-write sequence, one wait state is inserted every single write hit. For write burst accesses the line is read once and written back when the burst is terminated or when the end of the cache line is reached, triggering a new tag lookup. A write hit will also result in the dirty bit being set, thus indicating that the cache line will need to be evicted before it is replaced. A write miss is handled in the same way as a read miss and allocates a new cache line.

83.3.3 Cache flushing

The cache can be flushed by writing to the flush register. One flush mode exists, flushing the entire cache with write-back. During this operation, every cache line is invalidated and dirty lines are evicted.

83.3.4 AHB slave interface

The slave interface is the core’s connection to the CPU and the level 1 cache. The core can accept 8-bit(byte), 16-bit(half word), 32-bit(word), 64-bit, and 128-bit single and burst accesses. INCR and WRAP accesses are *not* supported.

83.3.5 AXI/AHB Master interface

The master interface is the core’s connection to the memory controller. It uses the Generic Bus Master IP core and supports AXI and AHB. The width of the backend is controlled with VHDL generic “be_dw “and can be configured as either 32, 64 or 128.

83.3.6 Endianness

The core is compatible with big and little-endian systems.

83.3.7 Performance counters

Two performance counters are available, access and miss counters. The counters are 32-bit and will wrap around if overflown. The counters can be read and reset by reading and writing to their respective addresses, see register overview below. A hit counter can also be calculated by subtracting the miss counter from the access counter.

83.4 Registers

The core is configured via registers mapped into the AHB memory address space. Only 32-bit single-accesses to the registers are supported.

*Table 1677.*L2C-Lite: AHB registers

AHB address offset	Register
0x00	Flush enable register
0x04	Access counter
0x08	Miss counter
0x0C	Cache configuration

83.4.1 Flush enable Register

*Table 1678.*0x00 - L2C-Lite - Flush enable register

31	0
RESERVED	Flush enable
	0
	w

0 Write 1 to initiate cache flush

31:1 RESERVED

83.4.2 Access Counter Register

*Table 1679.*0x04 - L2C-Lite - Access counter register

31	0
Access counter	
0	
r/w	

31 : 0 Access counter. Write 0 to clear counter.

83.4.3 Miss Counter Register

*Table 1680.*0x08 - L2C-Lite - Miss counter register

31	0
Miss counter	
0	
r/w	

31 : 0 Miss counter. Write 0 to clear counter.

83.4.4 Status Register

*Table 1681.*0x0C - L2C-Lite - Cache configuration

31 30	29 28	27 20	19 16	15 14	13 0
REPL	RESERVED	WAYS	LINE-SIZE	RESERVED	WAY-SIZE
*	*	*	*	*	*
r	r	r	r	r	r

31: 30 Replacement policy (REPL), 0 = Pseudo random, 1 = Pseudo LRU

29: 28 RESERVED

27: 20 Multi-way configuration (WAYS) - (Associativity - 1)

19: 16 LINE-SIZE -

0 : 16 B

1 : 32 B

2 : 64 B

3 : 128 B

4 : 256 B

15: 14 RESERVED

13: 0 WAY-SIZE - Size in KiB

83.5 Vendor and device identifiers

The core has vendor identifier 0x01 (Cobham Gaisler) and device identifier 0x0D0. For description of vendor and device identifier see GRLIB IP Library User's Manual.

83.6 Implementation

83.6.1 RAM usage

The L2C-Lite uses dual-port RAM to implement both cache tags and data memory. The tags are implemented using the SYNCRAM_2P core, with the width and depth depending on the cache size configuration. The data memory is implemented using the SYNCRAM_2P which means that it can limit the line size and way size depending on which technology is used. See SYNCRAM_2P for technology specific limitations.

83.6.2 Endianness

The core changes endianness behaviour depending on the settings in the GRLIB configuration package (see GRLIB User's Manual).

*Table 1682.*Configuration options

Generic name	Function	Allowed range	Default
tech	The memory technology used for the internal Syncram.	0 - NTECH	0
hmindex	Master index	0 - NAHBMST-1	0
hsindex	Slave index.	0 - NAHBSLV-1	0
ways	Number of cache ways	1, 2, 3 ... N	2
waysize	Size of ways (KiB)	1,2,4,8,16,32,64, 128 ...	64
linesize	MASK field of the AHB BAR.	0 - 16#FFF#	0

*Table 1682.*Configuration options

Generic name	Function	Allowed range	Default
repl	Replacement policy: 0 = pseudo-random, 1 = pseudo-LRU	0-1	0
haddr	ADDR field of the AHB BAR	0 - 16#FFF#	0
hmask	MASK field of the AHB BAR.	0 - 16#FFF#	0
ioaddr	ADDR field of the AHB I/O BAR	0 - 16#FFF#	16#000#
cached	Cacheable memory ranges	x“0000” - x“FFFF”	x“FFFF”
be_dw	Backend bus width	32, 64, 128	32

83.7 Signal descriptions

Table 1683 shows the interface signals of the core (VHDL ports).

*Table 1683.*Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
AHBMI	*	Input	AHB master input signals	-
AHBMO	*	Output	AHB master output signals	-
AXIMI	*	Input	AXI master input signals	-
AXIMO	*	Output	AXI master output signals	-

*) see GRLIB IP Library User’s Manual.

83.8 Library dependencies

Table 1684 shows the libraries used when instantiating the core (VHDL libraries).

*Table 1684.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	L2CACHE	Component	Component declaration

83.9 Instantiation

This example shows how the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.tech.all;
library gaisler;
use gaisler.l2c_lite.all;

entity l2c_lite_ex is
  port (
    clk : in std_ulogic;
    rst : in std_ulogic
  );
end;
.
.
signal ahbsi  : ahb_slv_in_type;
signal ahbso  : ahb_slv_out_vector := (others => ahbs_none);
signal ahbmi  : ahb_mst_in_type;
signal ahbmo  : ahb_mst_out_vector := (others => ahbm_none);
signal ahbsi2 : ahb_slv_in_type;
signal ahbso2 : ahb_slv_out_vector := (others => ahbs_none);
signal ahbmi2 : ahb_mst_in_type;
signal ahbmo2 : ahb_mst_out_vector := (others => ahbm_none);
signal aximi : ahb_somi_type;
signal aximo : ahb_mosi_type;


architecture rtl of l2c_lite_ex is

begin

(AHB backend instantiation)
...

  l2c_lite0 : l2c_lite_ahb
  generic map(tech => 0, hmindex => 1, hsindex => 1, ways => 1, waysize => 64,
              linesize => 32, repl => 0, haddr => 16#400#, hmask => 16#C00#,
              ioaddr => 16#FF4#, cached => 16#00F3#, be_dw => 32)
  port map(rst => rst, clk => clk, ahbsi => ahbsi, ahbso => ahbso(1),
           ahbmi => ahbmi2, ahbmo => ahbmo2(1), ahbsov => ahbso2);

...

(AXI backend instantiation)
...

 l2c_lite0 : l2c_lite_axi
 generic map(tech => 0, hmindex => 1, hsindex => 1, ways => 1, waysize => 64,
              linesize => 32, repl => 0, haddr => 16#400#, hmask => 16#C00#,
              ioaddr => 16#FF4#, cached => 16#00F3#, be_dw => 32)
  port map(rst => rst, clk => clk, ahbsi => ahbsi, ahbso => ahbso(1),
           aximi => aximi, aximo => aximo);

...

end;
```