POLITECNICO
DI TORINO

Master's Degree in Electronic Engineering

Master's Thesis

Design of an OS compliant memory system for LEN5, a RISC-V Out of Order processor

Supervisor
Maurizio Martina

Candidate
Matteo Perotti

Academic year 2018-2019

“Homo sum, humani nihil
a me alienum puto.”
[Publio Terenzio Afro]
Nor the processors.

Abstract

The design of a complex memory system for an Out Of Order processor can be very intriguing, especially if it should be able to support an Operating System. There is not much documentation about how to design it and how to deal with the problems that Virtual Memory can bring with itself: the goal of this work is to provide a possible practical implementation of a memory system, along with thoughts and information about all the related issues and possible solutions.

Acknowledgements

I would like to thank Andrea Giannini and Maurizio Martina for their professional support. Moreover, I want to express my gratitude to Michele and Marco, who have shared with me this part of the journey.

Thank you Elisa, thank you “family”.

Contents

List of Tables

[1] Data Cache

[2] Write Back victim Buffer

List of Figures

[3]Instruction Cache

[4]Instruction Translation Lookaside Buffer

[5]Data Translation Lookaside Buffer

[6]Memory Management Unit

[7]Content Addressable Memory

Chapter 1

Introduction

1.1 Motivations

This Master Thesis work presents the design of a memory system for a RISC-V Out of Order processor, capable of supporting an Operating System.

Why an attempt to design a part of a new RISC-V core? There are many answers to this question, but it's worth to underline the main one: it's challenging, and as Florian Zaruba wrote in a presentation of his Ariane core, it's funny. [1]

A plethora of other reasons can be found pointing out that this work is mainly horizontal instead of vertical: a lot of thesis works focus on a limited aspect of an entire system, with a lot of efforts made to enhance it. This thesis is an exploration through the world of the processors and the memory systems, and a huge amount of time has been spent to understand the theory to translate it into an original design.

- The RISC-V (pronounced *risk-five* [2]) ISA[1] is gaining popularity over the years and the interest towards it is growing both for universities and industries. The interest is shared because this is an open ISA maintained by professional people and it was developed to be useful, focusing on simplicity, modularity, and elegance.

  Therefore, studying a complex architecture starting from the RISC-V specifications, manuals and literature is a great investment for the future, especially in this period.

- The world of the processors is vast, and, during the university courses, it hasn't been possible to study it in depth. In particular, the memory system has always been treated with strict and unlikely hypotheses to ease the discussion of the microarchitecture.

  A lot of difficulties arose during the reading of the literature regarding this theme, and this fact suggested that a lot of work was necessary to fill up these gaps.

- It's hard to find well-documented designs for such complex processors, and a lot of things are taken for granted. This work and its documentation will be made available

[1] Instruction Set Architecture

on GitHub, and the hope is to give to the whole academic world an easy to understand design.

- In the beginning, the project was enormous: a complete design of a multiple-issue Out of Order processor. Then, when its real complexity started to be clear, the main focus of the entire work switched to single parts of a single-issue processor.

  However, the main goal was unchanged: try to deal with a complex problem and design to learn from it and ourselves.

- During the various courses offered by the university, the main HDL[2] chosen for the projects was VHDL. In many foreign industries and especially for the verification process, System Verilog is often used instead.

  Trying to become familiar with this different HDL is an important step towards the job's world and can improve the comprehension of how modern hardware is described and verified.

The main difficulty of such a horizontal work is that also its goal was not so clear at a first glance; LEN5 (pronounced *lens*) is a general-purpose processor in its widest sense: it was not thought for a particular use, but only to study it. Therefore, a lot of choices taken to design it are mixtures of *best* choices to satisfy the majority of cases and compromises, due to the finite amount of time to complete the project.

LEN5 and its memory system are under development and aren't optimized yet, but the project is open and the basis now exists: everyone will be able to use it for educational and academic purposes.

1.2 Thesis organization

The thesis work is organized as follows:

- This is the *first introductory chapter*.
- The *second chapter* is focused on giving a background to the reader, to let him have enough knowledge to understand the next parts.
  First of all, a general discussion on the processors introduces the theory behind them, the related issues and their fundamental metrics. Subsequently, the same thing is done for the memory hierarchy, presenting a deeper analysis of today's Memory Systems.
- In the *third chapter*, the attention is brought to the RISC-V ISA. After a brief introduction to its history and motivations, the core set is presented with particular attention to the part related to the Memory System.
- The *fourth chapter* is about the LEN5 core. It is our open-source processor that will be available on GitHub. At the present state, it is composed of a front end, a back end and this Memory System.

[2]Hardware Description Language

- The *fifth and the sixth chapters* are the main body of the thesis work. They contain the description of the LEN5 Memory System, with its design and preliminary verification and synthesis results. During the discussion, some concepts will be repeated to highlight the motivations of the design choices. These chapters include parts of the documentation that will be available on GitHub.
- In the *seventh and last chapter* the results will be summarized, together with the main guidelines for possible future development.

In the electronic version of this document, all the words and concept that can result tough at first reading are explained in the glossary, together with the acronyms, and footnotes will aid the reader every time a new acronym is introduced.

Many parts of this thesis are more "verbose" than other similar works because the main purpose is to make this project and its documentation accessible also to students and people that have never tried to implement such a system.

Chapter 2

Background

2.1 Processors

It's hard to give a unique definition of what a processor is without being too generic because this term is often used interchangeably to indicate a general processor or a CPU[1]. Within this thesis work, the word *processor* will refer to a CPU.
A CPU is an electronic component of a computer system that is in charge of interpreting and executing instructions taken from a memory, elaborating data and controlling other parts of the system itself [3] [4] [5].

In the present, it's cumbersome to understand what a processor is starting the discussion with a hardware approach, because the main point is not the device itself, but the elaboration it's able to perform and why there is the need for performing it. People and industries need ways to store data and/or elaborate it in different ways. With a smartphone or a supercomputer, for controlling other systems or just for fun; the concept is the same, but how this is performed are different.
A possible way to deal with the complexity of such a goal is to divide the system into sub-blocks specialized in only a couple of works. Another great idea common in all the engineering fields is the **abstraction**. This concept is so pervasive in the life of everyone that it is often underrated and not clear.
The concept of abstraction makes sense when something should be used by someone (or something) else. If a thing is being used, there is always a goal: for example, an iPad is used to take notes, play games, listen to music. Which is the best way to allow someone to achieve his goals? Let him be focused on them. All the details about how the iPad works are irrelevant if the main goal is listening to music. Therefore, the user can use the iPad only knowing an incredibly small subset of all the characteristics of the device: if the finger press on this circle, a picture will be captured by the camera. Abstraction is the process of knowing only a small selected subset of characteristics of something, ignoring the others, to be more efficient in achieving the goal, or *what is important*, concentrating on it instead of on how the goal is achieved. This is valid also when hardware, software

[1] Central Processing Unit

or mixed design is performed: most of the physics is hidden to a hardware designer, and most of the hardware is hidden to a software designer.

A functional way of elaborating data and controlling systems is to be concentrated on the operations that should be performed instead of on the complexity of the functional devices, and the programming languages are the best example of it. A program is a set of instructions for a computer to perform some tasks, and a programming language is the set of rules on how to write those instructions that allow the computer to understand what the programmer intends.

The instructions readable by a human are translated into formats understandable by a particular processor and are encoded in predefined formats, which are series of bits. This set of instructions should be

- Stored somewhere
- Executed

The first need is fulfilled by the memory system (see chapter 2.2 for further details), the second by the processor or other processing units.

2.1.1 Basic concepts

The processor is a block that cyclically repeats a series of operations:

```
1) Fetch the correct instruction from the memory
2) Decode it
3) If needed, prepare the data useful for the execution
   of the instruction
4) Execute the related operation
5) If needed, save the results in the memory
```

This cycle allows the processor to be a flexible device because its functionality and possibilities depend on its internal structure and on the instructions it can execute.

The entire set of instructions the processor can understand and execute is abstracted into its ISA. The ISA is the first abstraction of the processor, from the hardware to the software level. It specifies the native data types, the instructions, the registers, the addressing modes, the architecture of the memory, the handling of exceptions/interrupts and, if present, the external I/O [5] (for further details on the ISA, refer to section 3).

Structure of a processor Each processor works on fixed packets of bits of different dimensions. Usually, a packet is generally referred to as word. In the following, the term word will be used to identify the natural packet of bits on which the processor can do some decoding or elaboration.

The memory system feeds the processor with instructions and data, but the processor itself needs a way to store these elements during the processing. The basic element used to save useful information is the *register*.

A register is a set of *flip-flops*, electronic primitives for the storing of a single bit.

Then, the CPU is also composed of blocks useful to elaborate the instructions and the data, along with connections among all these elements.

The aforementioned *Fetch-Decode-Execute* cycle can be basically viewed in this way:

- The address of the next instruction is stored in a register, usually called the PC[2]. This address is given to the memory. The memory returns the requested instruction to the processor, which stores it in another register.
- The instruction is decoded and particular signals are prepared to control the execution of the instruction itself.
- The operands, if required, are fetched or prepared too.
- The control signals drive the execution of the instruction, controlling the execution units.
- The result, if needed, is saved somewhere, in a register or back to the memory.

Naturally, the reality is more complex and complicated because the processor should also be a *good* processor, but this is the base.

Exceptions and interrupts The described cycle can be interrupted by particular events. These events can be synchronous with the execution and generated by the processor (exceptions), or asynchronous, coming from the external world. For further details, refer to [6].

2.1.2 What matters? The main fundamental metrics

It's not possible to define a *good processor*, because the concept of "good" varies a lot depending on the situation. However, three fundamental aspects usually characterize an electronic device of this type:

- Performance
- Energy and Power consumption
- Area

Performance

There are multiple ways to define and measure the performance of a computer, but in the following, it will be always used the *CPU time*. This is the time spent in computing for performing a particular task, without including the time spent waiting for data I/O or other tasks nonrelated to the measured one [6]. In formula, it can be expressed as:

$$t_{cpu} = N_{clk} \times t_{clk} = CPI_{avg} \times N_{instructions} \times t_{clk} \tag{2.1}$$

[2]Program Counter

where:

- t_{cpu} is the total CPU time.
- N_{clk} is the total number of clock cycles relative to a particular task.
- CPI_{avg} is the average CPI[3]. Each instruction of the processor can require a different amount of clock cycles to be executed. The average of the CPI of each instruction, weighted with the frequency of that instruction in the complete program, is the average CPI.
- $N_instructions$ is the total number of instructions executed to complete the task.
- t_{clk} is the clock period.

This formula is a reference when a choice finalized to performance improvement is made during the design. It is also helpful because its terms are easily measurable and can be used in practice [6].

Another fundamental equation that can aid the design especially during its first phase is Amdahl's law. It can be derived by the equation of the line. The law talks about the performance improvement achievable speeding up a part of a system: this improvement is limited by the fraction of time-dependent on that part. For example: if decoding an instruction takes 2 cycles and the execution takes 1000 cycles, it's worthless to waste resources to enhance the decoding stage to complete an instruction decoding in only 1 cycle. The decoding stage would be two times faster ($\frac{2cycles}{1cycle}$), but the overall speedup would be only $\frac{1002cycles}{1001cycle} \approx 1$ times faster. On the other side, a speedup of two times on the execution stages would have brought a total speedup of approximately two times. In formula:

$$Speedup_{overall} = \frac{ExecutionTime_{old}}{ExecutionTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \quad (2.2)$$

Where:

- $Speedup_{overall}$: the effective ratio between the old execution time and the new one
- $ExecutionTime_{old}$: the execution time without the improvement.
- $ExecutionTime_{new}$: the execution time after the improvement.
- $Fraction_{enhanced}$: the fraction of the computation time that can be converted to take advantage of the enhancement.
- $Speedup_{enhanced}$: the speedup the system would achieve if the fraction enhanced was 1.

[3]Cycles Per Instruction

[6]. In other words, the base concept is:

If there is a time interval that depends upon many parameters, the effective reduction in time of that interval achieved modifying one of that parameters is limited by the impact of that parameter on the total time interval.

It can seem a banality but this law is not only useful, but it is also severe.

Techniques to improve the performances There are different ways of improving the performance of a processor. The equation 2.1 states that there are three equally important parameters to be taken into account:

The clock period is the measure of how fast the processor works. If no power issues are present, it is determined by the critical path of the digital design. Usually, this metric seems to be the most determinant, but the reality is different: most of the time the clock period is linked to the average CPI. Adding a register can break the critical path, but it will likely increment the CPI too. Therefore, it's not wise to evaluate a design only on one of these parameters.

Reducing the Instruction Count If the processor is designed to support a lot of particular and complex instructions, the instruction count to complete a task can become low. For example: if there's the need for executing a FFT[4] on some data and a processor implements such instruction as a single one, the instruction count for this task will be only 1. This idea guided the design of the processors for many years and brought to what is called CISC[5] processor.

The main goal was to have the minor possible number of instructions to solve a particular task. The instructions were complex, and this complexity was absorbed by the hardware. A lot of silicon was used to implement execution units capable of dealing with such complex requests. The main effect was a reduction of the available area for the memory units, a higher clock cycle, and a very high average CPI.

For example, the pseudo-code operation

```
A = A * B
```

with both **A** and **B** stored in the memory, can be translated into only one multiply operation:

```
A MULT B
```

where **MULT** is a single processor operation.

During the execution of that instruction, the memory is accessed to fetch both the operand. Then they are multiplied and the result is stored back into the memory.

[4]Fast Fourier Transform

[5]Complex Instruction Set Computer

The main point is that usually, the more the instruction is complex, the less it is used. Wasting a lot of resources to maintain the support for instructions that were seldom used was worthless.

Reducing the CPI Another paradigm of designing a processor and an ISA is the RISC[6] one. This approach breaks the complex CISC instructions into simple and basic ones, which can be completed in one clock cycle. The execution units are simpler, and there is more space for registers. For these reasons, there are special instructions to talk with memory. The others perform operations on data previously loaded into the registers.

Using the previous example, the pseudo-code operation

```
A = A * B
```

with both **A** and **B** stored in the memory, can be translated into four basic operations:

```
LOAD A
LOAD B
A MULT B
STORE RESULT
```

This time, the **MULT** operation is only a multiplication between two operands already loaded into the processor. Other instructions are responsible for talking to the memory.

During the execution of that instruction, the memory is accessed to fetch both the operands. Then they are multiplied and the result is stored back into the memory.

With a RISC processor, more instructions are required to complete a program, but each instruction consumes fewer cycles.

Exploiting Instruction level parallelism To speed up the execution it is possible to execute instructions in parallel. Even if it is not obvious, not the whole instruction execution can be parallelized. The majority of instructions, indeed, depend upon the previous ones.

For example, one of the primitive elements of the programming is the conditional branch. On a higher level it can be seen as an *if statement*:

$$if\ (A) \rightarrow B \tag{2.3}$$

The instructions in the **B** block cannot be processed until **A** is verified.

[6]Reduced Instruction Set Computer

Pipelining The most important way of performing parallel works in digital processors is, without doubt, the pipelining. The name derives from the pipes: when a volume of water enters the pipe, it will exit it after an amount of time. This time can be called "latency". After this time, the water will go on exiting the pipe until it is not over. During this period, the pipe is full of water, and the flow at the exit point is continuous.

This concept is common in the industry with the name of "assembly line". Instead of building a car from the start to the end and only then begin the next one, industries began to have different people and machines continuously performing a smaller repetitive work on different cars. Mounting a glass on the first car, then on the next one, and so on. This allows having a lot of cars that are built at the same time. Each operator (or machine) works on a different car performing a different operation.

In the processors, this can be achieved dividing the unit into more different sub-blocks that are meant to perform a different operation each. For example, referring to the *Fetch-Decode-Execute* cycle, a first block can be responsible for fetching the instruction, a second block for decoding it, the third for its execution. When an instruction is fetched, it is saved into a register. The next clock cycle the instruction is decoded, while another instruction is fetched and so on. When the pipeline is full, the processor is processing three instructions, one per stage.

The pipelining can help in reducing the clock cycle and the CPI, but it's important to remember that is a form of parallelization. As already said, not all the instructions are parallelizable.

Enhance the common case Eventually, **Amdahl's Law** suggests that it's usually a good idea to focus the efforts on enhancing the most frequent case [6]. The common case is also usually simpler to be enhanced [7] and often is so more common than the infrequent cases that the overall performance benefit is much more positive than the hypothetical benefit from enhancement on the infrequent case.

Energy and Power

Initially, processor designers did not care about power consumption. There were relatively few transistors inside a chip, and the clock frequency was low. With the advent of the RISC architectures, the frequency started increasing and this brought to an increment of the power consumption with consequent issues with the power delivery and heat dissipation.

The power of a chip can be divided into three terms:

- Dynamic power: the power related to the energy dissipated to charge/discharge the resistive and capacitive lines of a circuit. This is the only "useful" consumed energy.
- Static power: the leakage power related to the non-idealities of the CMOS transistors. Even when they are supposed to be off, they let some charges pass through themselves.
- Short circuit power: the power related to the energy dissipated during the switching of a logic gate, inside the logic gate. It is due to the non-idealities of the dynamics of a logic gate.

In a first approximation, the third contribution can be neglected.
The Static Power is a leakage power, and depends on how many transistors are off, how

ideal they are, etc. Part of the device can be completely shut down to eliminate these non-ideality.
The dynamic power formula can be derived from the energy dissipated during a single commutation to charge the capacitance of a single transistor, from a logic 0 to a logic 1 or vice versa.

$$E_{tr} = \frac{1}{2} \times C_{tr} \times V^2 \tag{2.4}$$

If on average a percentage α (switching activity) of all the transistors switches during a clock period and the total capacitive load is C_{tot}, the average dynamic power of the chip becomes:

$$P_{dyn} = \frac{1}{2}\alpha \times t_{clk}^{-1} \times C_{tot} \times V^2 \tag{2.5}$$

$$P_{dyn} = \frac{1}{2}\alpha \times f \times C_{tot} \times V^2 \tag{2.6}$$

A frequency that increases can bring to greater power consumption because ideally the same work is done faster: more energy is requested in less time (for a fixed task, the requested energy remains the same). To address the problem of the power it is important to be careful at design time. There are a plethora of techniques used to mitigate this issue, but the "power wall" is the main cause that stopped the frequency increasing that was ruling the world of the processors until the first years of the XXI century. For a long time, the number of transistors inside a chip increased exponentially, and the frequency increased too: this was paid decreasing the voltage. Then, when this process reached its threshold, the power consumption became too high not to raise the concerns of the designers. Today, power consumption is one of the most important parameters to be taken into account [6].

Area

To understand why a large chip can be a problem it is useful to refresh how integrated circuits are made.
A cylinder of Silicon (*wafer*) is cut in slices. The largest possible number of circuits is "printed" on each slice, which has a fixed area. The slice is not perfect, and has a constant density of defects on its surface; hence, the number of defects is fixed for each slice. If a circuit is printed on a defect, it will not work. Therefore, if the single circuit is small, there will be less probability for one of them to have a defect inside it. On the contrary, a very big circuit will likely incorporate a defect. What is important for a designer is that the number of circuits that will not incur in defects grows roughly as the square of their area [6].

2.1.3 Advanced issues and techniques

There are a lot of techniques that allow the processor to better exploit the *Instruction Level Parallelism*, i.e. the parallelism between different instructions. In the following, the most interesting ones (from a LEN5 perspective) will be briefly summarized.

Dynamic Scheduling In an *in-order* processor, each *data hazard* or unpredictable event such as a *cache miss* stalls the processor, and no new instruction is fetched, issued, executed. With dynamic scheduling, the processor can move the instructions and handling them *out-of-order* to hide the stall latency whenever is possible.
With this technique, *control hazards* can still stall the pipeline.

Speculative processors Some processors deal with the control-hazards predicting the outcome of the next branch. Speculative processors perform also execution on the predicted branch. They need ways to restore the initial conditions if the branch was mispredicted.

Multiple issue processors If a processor is able to fetch, execute and commit one instruction per clock cycle, it's not possible to reach a final CPI lower than 1. For this reasons, *superscalar* processors are designed to perform these operations on more than one instruction (this operation can be done *in-order* or *out-of-order*). LEN5 is not *superscalar*, and therefore will have a *CPI* higher than 1.

2.2 The Memory System

The term *memory* indicates the part of a system that stores some information. Into the processor, the information is usually stored using registers. They are fast enough to sustain the internal clock frequency, but they are expensive in terms of area and power.

Indeed, a register is composed of flip-flops, and flip-flops occupy more area than a usual RAM[7] memory cell [10].

2.2.1 The interface between CPU and Memory

During the fetch stage, the processor needs to request the next instruction to the Memory System. The most simple request is composed of a *memory address* and a *valid request* signal. Usually, the minimum entity addressable by the processor is a single memory Byte. If the length of the address is **A** bits, the processor can address 2^A Bytes.
To simplify the following discussion, we will consider words long 4 Byte (32 bit).

2.2.2 The Memory Hierarchy

Usually, a processor needs a lot of storage space to have access to complex programs and data, and the internal registers are not sufficient. This information is stored in the Main Memory. Unluckily, a large memory is also much slower than the speed requested by the operating frequencies of the actual processors: this is caused by the increment of the capacitance of the memory with its increasing size. But, there is a loophole: data tend to be local both in time and in space. A processor working on some data will likely request again the same data after a brief interval of time (temporal locality), and it's also easy that it will use data stored in the proximity of the first ones (spatial locality). It's easy to see

[7] Random Access Memory

these localities in the following C example, where each element of a vector is incremented by a constant:

```
for(int i = 0; i < N_MAX; i++) {
  vector[i] += constant;
}
```

The elements of the array are stored in contiguous locations in memory and are accessed in order, one by one (*spatial locality*). On the other hand, *constant* is cyclically accessed during every sum (*temporal locality*).

These facts have brought designers to prefer a sparse and split memory system organized as a hierarchy: the *memory hierarchy*. Near the processor are present small and fast memories called **caches**, which are in communication with bigger and slower caches, that are connected to the Main Memory; as the memory size increases, the memory is physically located further away from the processor. When the processor needs data, a request is done to the lower-level caches, the ones near the CPU: if the data is not present there, the requested is forwarded to the higher-level caches (if present) and then to the Main Memory. When the data is found, it is brought to each of the memories of the lower levels and in the end to the processor. Then, the processor will find that data in the near and fast memory whenever it needs it. Different memory hierarchies can present different complexities, with a different number of levels. Usually, the first level cache is called **L1**, then the other levels, if present, are named **L2**, **L3** and so on. Then, the final memory (the biggest and the slowest) is the Main Memory. At the lowest edge of this hierarchy, before the L1 Caches, there are the registers of the processor: small and fast.

This is functional only because data manifest temporal locality. If it's likely that the same data will be preferred to the others, having them near the processor can increment the performances. To exploit spatial locality, data is moved into packets bigger than the dimension of a single word: more adjacent words are packed together to form a *line*, and the line is the minimal entity transferred to and fro the levels of the hierarchy.

The usual example done to make this concept more familiar is the producer-library-house-desk one. A person can read a lot of books during his lifetime, but neither always going to the library, nor keeping all these books at home is practical. Therefore, only a small subset of those books are kept at home, the ones that are more liked and read. While studying, only two or three of those books inherent to the subject are kept on the desk and can stay there even for weeks. When another book is needed it's easy to search for it into your home and then into the library. If the book is not present there, the producer is asked for it. The book is sent to the library, then it is borrowed and arrives in the house and then on the desk. And this is efficient only because books exhibit temporal locality.

Returning to the memories, the net effect of a memory hierarchy is to give the processor the illusion to have a memory large as the Main Memory and fast as the L1 Cache. The processor is not aware of the memory hierarchy, because its interface to the Memory System is unchanged. The processor puts a valid address on the address line, makes the request and waits for the data to be returned on the data line. The Memory Hierarchy is transparent to the processor. [19]

2.2.3 Von Neumann vs Harvard architectures

Sometimes, in literature, some terms evolve during the time. At the moment of their coining they mean something, but the meaning change because also the conditions change. In the beginning, for historical reasons, the architecture of a system was called *Von Neumann* if data and instructions were stored together in a unique memory, with a single path for both. On the contrary, if the paths were split with two distinct memories for instructions and data, the architecture was named *Harvard*. [19]

During the time, memory systems have evolved and memory hierarchies have modified the initial paradigm. In this thesis work, the term *Harvard architecture* is used with the following meaning, directly quoted from [7]:

"...the term Harvard architecture was coined to describe machines with distinct memories. Paying respect to history, this term is used today in a different sense to describe machines with a single main memory but with separate caches for instructions and data." [7]

2.2.4 Caches

A **cache** is a small and fast memory, usually located in the proximity of the processor. It is used to store the most frequently used data. Most of the time, caches are implemented using SRAM[8], which are faster then DRAM[9] even if they are bigger.

If the memory hierarchy is well designed, most of the time the processor will find the requested data accessing only the L1 Cache. When a request is done to a cache, there are two possible outcomes:

- **Hit**: the requested data was in the cache and is returned.
- **Miss**: the requested data was not in the cache, and it should be brought into the cache from the higher levels of the hierarchy.

Internal cache organization

When a particular word is requested by the processor but the request misses, the L1 Cache asks for the word to the higher levels, until the top of the memory hierarchy is reached; usually, the last block is the Main Memory. To exploit spatial locality, a block of adjacent words is transferred from the higher level to the lower one: when it reaches the cache, it is saved into a line.

In general, a cache can be logically viewed as a memory containing lines. The line is the minimal entity that is transferred to and from the cache. A block has the same size as a line, therefore the term "line" will be used indifferently to identify a block or a line. When a word request hits, a line is returned to the processor; then, the correct word can be extracted from it.

[8]Static Random Access Memory

[9]Dynamic Random Access Memory

The cache lines are grouped into sets. A cache can contain one or more sets and each set can contain one or more lines. Each line of the Main Memory is associated with a particular set of the lower level cache. Consider the following example:

```
Main Memory size: 1 GiB
Cache size: 8 KiB
Cache line size: 16 B per Line
Word size: 4 B per Word
Number of cache sets: 256 sets
```

The Main Memory keeps $\frac{1\ GiB}{16\ \frac{B}{Line}} = 64 \times 2^{26}$ Lines. The first 256 lines are associated with the first 256 Cache sets, line 0 to the set 0, line 1 to the set 1 and so on. The next 256 lines are again associated with the Cache sets in ascending order, and so on for all the lines. This means that if the cache is composed of **S** sets, the Main Memory line number **N** will be associated with the Cache set number **N** *mod* **S**. *Each line is associated with one precise set only.*

If the processor asks a word to the Memory System and a miss occurs, the line containing the correct word will be transferred from the Main Memory to the Cache and stored into its assigned set. It is clear that each set is assigned to many lines, but can't contain them all together. In this example, each set is assigned to $\frac{64\times2^{26}\ Lines}{256Sets} = 16Mi\frac{Lines}{Set}$. But physically, the Cache can contain only $\frac{8\ KiB}{16\ \frac{B}{Line}} = 512Lines$, and each set $\frac{512Lines}{256Sets} = 2\frac{Lines}{Set}$ at a time.

Therefore, if a Cache set is already full, a line should be evicted to make free space for the new requested line. The principle of spatial locality suggests that near lines in the Memory are likely to be used one after the other, therefore are placed in different sets to reduce the intra-set competition. This is why adjacent lines are assigned to different sets.

Each line saved in the Cache is uniquely labelled (*tagged*) with a tag which identifies it. This way it is possible to understand if the searched line is present in the set or not.

To summarize, the processor gives to the Memory an address, which is usually a Byte address and wants to receive a word. Under the following hypothesis

```
Address length = 32 bit
Bytes per word = 4 (addressed by the 2 bits: word offset)
Words per line = 16 (addressed by the 4 bits: line offset)
Number of Cache sets = 256 (addressed by 8 bits: cache index)
```

There are $32 - (2 + 4 + 8) = 32 - 14 = 18)$ bits in the address, the most significant ones, which are called tag. This is the part of the address which uniquely identifies each line. Indeed, when the set is addressed, it's not sure that the searched line is present. The tag portion of the address is associatively compared with the tags of each line to produce a hit or a miss.

N-Way Set Associative The organization of the Cache is named depending on the line capacity of the sets. If each set can contain N lines, the Cache is called *N-Way Set Associative* Cache. Each Main Memory line can be mapped to N lines of the Cache.

Direct mapped In the extreme case in which the sets contain one line only, the Cache is called *Direct Mapped*. With this organization, each Main Memory line is direct-mapped into a precise line of the Cache. When a line should be replaced in a set, the old line is evicted.

Fully Associative At the opposite, the Cache can be composed of only one set which contains the totality of the lines. In this case, no index is necessary, and the part of the address which is not a line offset or a word offset is a tag. Each Main Memory line can be mapped to any Cache line. When a replace occurs, there's the need for a way to select the old line to be evicted.

Cache performance

To understand the design choices of a cache, it is important to understand what is important. As for the processors, the fundamental metrics can vary depending on the situation, but for this work, we will focus especially on performance, area and power/energy. To understand these concepts, special terms will be used:

- Hit rate (**HR**): the probability for a request to hit
- Miss rate (**MR**): the probability for a request to miss
- Miss penalty (**MP**): the time (or the clock cycles, it will be clear from the context) spent starting when the miss occurs until the line is brought to the Cache
- Hit time (**HT**): the time spent to hit

The Memory Systems described is a *slave* of a processor. Therefore, the final metric to be taken into consideration is the execution time of the entire system, as it was hypothesized in 2.1.2.

The Memory System performance can be measured by the AMAT[10]. It measures the weighted average time for the processor to access data from the Cache after a request.

$$\text{AMAT} = HR\times(HT) + MR\times(HT+MP) = (1-MR)\times(HT) + MR\times(HT+MP) \quad (2.7)$$

And so:

$$\text{AMAT} = HT \ + \ MR \times MP \quad (2.8)$$

There are more summed terms and the impact of an improvement on one of these can be evaluated using the reasoning of **Amdahl's Law**.

This equation is fundamental because it can help in evaluating the performance of the processor, especially if it is an In Order processor and the stalls due to the memory system overwhelm the stalls due to I/O memory contention. [6]

[10] Average Memory Access Time

If the processor is Out of Order, the AMAT can be unlinked from the final execution time, because the impact of the stalls due to memory misses on the execution time can be mitigated by the Out of Order execution. The preferred method to evaluate design choices on the Memory Hierarchy for an Out of Order processor is usually the simulation. [6]

In this work, a full simulation was not possible. Therefore, in the first instance, minimizing the *AMAT* was taken as a design principle.

Cache design choices

Cache Size The more the cache is capacious, the more data will be able to hold inside.

If size increases:

- Miss rate is reduced

but

- The Cache is slower (the Hit Time can increase)
- More energy requested
- More area occupied

Associativity If a set can host more lines, there is less intra-set competition. For example, in a direct-mapped cache, all the lines of the set 0 are mapped in one position only. In a fully associative cache with the same size, the same lines are mapped on more positions. With higher associativity, the miss rate is decreased.

The main drawback of associativity is the increasing complexity: the TAG comparison inside a set is done in parallel, and its cost increases with the number of comparisons to be done. Also, the selection of the line which hit is more complex. This affects the occupied area and the energy consumption, but can also increase the delay end possibly

Increasing the associativity:

- Lower miss rate

but

- Higher hit time
- Higher area
- Higher power consumption
- Higher miss penalty (for a more complex replacement policy)

Replacement Policy The replacement policy dictates which line of a full set is evicted when a new line should be stored in the same set.

The best possible policy would be implemented knowing in advance which is the line that will be requested more later in time (Bélády's algorithm). Most of the time this is not feasible in practice, even if there are ways to try to predict it [13].

There are a lot of policies which can perform better than the others with particular workloads. The main ones are:

- LRU[11]: the next line to be evicted is the one which has been accessed less recently. This policy is easy to be implemented with 2 Way Set associative caches because only a bit is needed. When the associativity increases, usually a pure LRU is not used.
- LFU[12]: the line which has been accessed less is the one which will be evicted.
- PLRU[13]: approximation of the LRU, it can be implemented in more than one way. If the number of elements is a power of two, it can be implemented with a tree PLRU.
- NRU[14]: approximation of the LRU. When a line is accessed, an *access bit* is set. Until the access bit is set, that page cannot be evicted. When the *access bit* of the last page is being asserted, all the others are cleared.
- FIFO[15]: The older line of the set is the next victim.
- Random: the next line to be evicted is chosen randomly or pseudo-randomly.

A good algorithm reduces the miss rate, but a complex algorithm can increase the access time and/or the miss penalty.

Line Size Larger lines can promote a higher hit rate, because of the spatial locality of the data hold into each line. But with equal Cache size, larger lines mean fewer lines, and this means higher competition in the Cache. Additionally, a bigger line can increment the miss penalty and every other transfer, because more data should be transferred between memories.

If the line size increases:

- The hit rate can be incremented thanks to better exploitation of the spatial locality

but

- The hit rate can also be decremented because of higher competition of the lines
- Higher miss penalty
- Higher costs for line transfers

[11] Least Recently Used

[12] Least Frequently Used

[13] Pseudo Least Recently Used

[14] Not Recently Used

[15] First In First Out

Write policy The processor can also need for writing data to the memory. This operation can be done in several ways, but the first distinction is about how the cache behaves when it is written. If the processor writes data into a cache line and the cache updates the higher level of the memory, the Cache implements a Write Through policy. If, on the contrary, the Cache does not immediately update the higher level of the hierarchy, the Cache implements a Write Back policy.

For a WB[16] cache can contain updated lines that are not stored in the upper levels of the hierarchy, they cannot be simply evicted during a replace. Every time a line is written, a special state bit is asserted: the *dirty bit*. During a replacement, if the line to be evicted is marked as dirty, it should be written back to the higher level memory to avoid loss of data.

A WB cache can reduce the inter-level memory traffic, especially if the line size is high enough. Both the policies need buffers to reach high performances: if the cache remains busy and stalls when a line is written back to the upper level the processor, especially if In Order, can be critically slowed down [11].

Write Back Victim Buffer If the cache is WB, a WBB can speed up the back-up of the dirty lines which are evicted from the cache. The Victim Buffer introduces possible RAW[17] hazards because the value buffered inside it are updated versions of what is present in the *L2 Cache*. When a request misses the cache, there's the need for checking also the WBB and use that line for reading or writing, without asking for a stale copy of it from the *L2 Cache*.

Write miss policy Write-misses are not as easy as read-misses. If a read request does not find the correct line, nothing bad has happened. When a store is performed, data is modified, and if the line was not the right one there is a risk of data loss.

As summarized in [11] the policy adopted by the Cache when a miss occurs is defined by three semi-independent variables and primarily affects the latency of a memory operation:

- Write allocation: a line can be allocated or not when a write misses
- Fetch on write: the right block can be fetched from the higher level of the memory when a write misses
- Write invalidation: the line of the cache is marked invalid and the data is forwarded to the higher level

If the correct block is fetched from the higher levels, it's a nonsense to mark as invalid or not to allocate a line. Therefore, in literature, the combination of Fetch on Write and Write Allocate with no line invalidation is called **Write Allocate**.

With this policy, every time a write miss occurs, a line is allocated into the lower level Cache. Then, the correct block is brought into that line and the store is replayed.

[16] Write Back

[17] Read After Write

Other policies hypothesize a *No Fetch on Write* policy. A possibility is to allocate a line and write only the data into an empty line. Only that piece of the line is marked as valid. This policy is called **Write Validate**. Hypothesizing also a *No Write Allocation*, the line can be either invalidated (**Write Invalidate**) when a miss occurs or kept as it is (**Write Around**), directly writing the upper level of the Memory Hierarchy [11].

The **Write Allocate** strategy is especially useful if the Cache is WB. This is because when data is fetched can remain in the Cache and be written more times. On the contrary, the other policies which can be collected under the name of **No Write Allocate** can be wiser to be used with a Write Through Cache, not to double the memory traffic.

The conclusion of [11] is "*Write-validate and write-around always outperform fetch-on-write*" [11]. It would be interesting to understand if this is still valid, with the consideration of a possible increase in the complexity of a Write Validate policy.

Unified or split caches The processor, during the same clock cycle, can need for instruction and data. If the Cache is one and does not have two access ports, a Structural Hazard happens.

A possible way to deal with this hazard is to maintain two split caches, an i-Cache, and a d-Cache. Split caches force data to compete only with data and instruction with instructions. This means that the miss rate can be higher (but this is not a synonym of a lower AMAT).

If split caches with half the size of a unified cache, the first have:

- More bandwidth
- Fewer hazards
- Possibility to adopt different replacement policies and organizations

but

- A slightly higher miss rate

Multilevel cache The Memory hierarchy can be composed of a different number of cache levels. They can be on-chip or off-chip (the difference is in the energy requested and in the latency of the answers). There can be only one level or more.

Usually, L1 Cache is small to have a low hit time not to impact on the main clock period or the latency. But if the cache is small, the miss rate is high. The only way to keep the AMAT low is to reduce the miss penalty, as suggested in 2.8.

A possible way to do this is to interleave an additional cache level between the Main Memory and the L1 Cache: the L2 Cache. Even if the L1 Caches are split, the L2 Cache can be shared.

The L2 Cache is faster than the Main Memory because is smaller and this fact means that if a request misses the L1 cache but hits the L2 one, the miss penalty is reduced.

Inclusion policy If there are more cache levels, there are different ways to handle the inclusion policy.

- Inclusive: if data is present at a particular level, it is assured that it is also present in all the higher levels. This implies that if a higher level cache evicts one line, that line should be invalidated in the lower levels if present. This scheme is useful if the system is Multi Core or Many Core: if many processors share the same data, an inclusive cache policy can reduce coherency problems.

- Almost inclusive: if a data is requested by one level, that data is passed in *daisy-chain* through all the upper levels until the one in which missed. Therefore, the data is included in all the higher levels, but there are no constraints on the future inclusion: if an upper level evicts the line, no invalidation signals should be given to the lower levels. This is the most simple policy to be implemented.

- Exclusive: if a data is present in one cache level, it should not be present in the others. If a miss occurs at a lower level and the replacement is going to evict a line, that line is saved in L2 in place of the line sent to L1. This scheme can maximize the cached data.

Lockfree cache Normally when a miss occurs the cache stalls until the line is returned from the higher level. This behavior is tolerable if the processor would have been blocked anyway, as in a strict "In Order" fetch or execution. If the part of the processor which is in charge of making Memory requests is Out of Order, a stall frustrates the benefits derived from that type of execution.

For this reason, especially for the data, the cache is often *non blocking* (or *lockfree*). The first proposal of a valid *lockfree* cache can be found in [12], and metodically discussed again in [14].

The processor usually requests a word, but to resolve a miss an entire block is transferred to fill the Cache line. A possible way not to let the Cache stall is to decouple the requesting of the missed blocks to the higher level from the Cache itself. When a request misses, the information about the request can be saved somewhere, waiting for the upper-level memory to be ready for a transaction. In the meantime, the d-Cache is free to accept other requests from the Back End of the processor without stalling. In [12], the place in which are stored the requests which missed is called MSHR[18].

According to [14], there are three types of miss:

- Primary miss: if there are no entries in the MSHR associated with the missed block

- Secondary miss: if the request of the block is pending in the MSHR and there are enough resources to handle the miss (no stalls needed)

- Structural stall miss: if the request of the block is pending in the MSHR and there are not enough resources to handle the miss (stall needed)

[18] Miss Holding Status Register

The distinction is important because if a block request has already been taken into account, only information about the destination (and possibly other control bits) should be saved.

A classical implicitly addressed MSHR is a buffer with *N_MSHR* entries. Each entry supports one primary miss, and is composed of different fields:

- Missed block address
- A Valid bit
- A destination block composed of *N_WORDS* entries, each with:
  - A word Valid bit
  - The destination of that word
  - Control bits for that request

A primary miss needs an allocation of a new entry in the MSHR, which testimonies a pending block request to the upper level of the memory hierarchy. Then, if other requests miss for different words, the corresponding entry in the destination block is filled. Every time an entry is already occupied, a stall is mandatory.

Memory coherence Memory coherence is especially important in a multiprocessor system because in a uniprocessor environment it is the same as enforcing the program order of the memory operations.

Taking ispiration from [7], a memory system is coherent if:

- If a processor **P0** writes a value of X at the memory location $A0$ and then reads the same location, the obtained value is X if no processor has written to $A0$ between the write and the read.
- If a processor **P0** writes a value of X at the memory location $A0$ and no other writes are performed at that location by any processor, any read of that location made by any processor will return X if a sufficient amount of time has passed from the write.
- If a processor **P0** writes a value of X at the memory location $A0$ and then a processor writes a value of Y at the same location, all the processors will see the writes in the same order. Therefore, every processor, after a sufficient amount of time, will see Y after a reading of $A0$.

In the case of uniprocessor designs, the first and the third point should be taken into account and for RISC-V are discussed in 3.3.2.

Memory consistency This issue should be taken into account when the system has more than one core. A good tutorial on the memory consistency models can be found in [29].

2.3 Virtual Memory

The already described Memory Hierarchy lacks an important element, which was not fundamental to understand cache performance: the secondary storage. A program to be executed must be kept into the Main Memory, but not all the programs can physically be stored there for space reasons. Additionally, there's the need for a place to host the code and the data which should be permanently stored: a non-volatile memory; this memory, or secondary storage, is the higher level of the memory hierarchy. Today is common for Secondary Storage to be managed by the Operating System.

In simple terms, Virtual Memory is a static or dynamic encoding of the Address Space of the processor. Once it is encoded, it can be divided, selectively protected, resized and managed in different ways.

Why Virtual Memory Virtual Memory is important for two main reasons: the need for programs to exceed the Main Memory limitations and the need for executing different programs which share the same memory space in a safely and efficiently [7].

Historically, the size of the Main Memory was an important limit for the programmers, because they had to write and size the program to let it fit into the memory. When space was running out, particular chunks of the program were brought to the secondary storage. The programmer was in charge of dividing the program into chunks and marking the mutually exclusive ones, to let them fit the memory and be able to leave it without inferring on the functionality of the program.

With Virtual Memory, the Operating System manages all these aspects and rules the memory sharing between different processes. Today, indeed, it is used mainly to implement a safe sharing of memory among programs and virtual machines.

2.3.1 The modified Memory Hierarchy

With Virtual Memory, the secondary storage becomes an efficient part of the logic behind Memory Hierarchy. Especially with Demand Paging, the Main Memory is used as a cache for the secondary storage. Programs are initially stored in secondary storage, which is divided into Pages. There are differences in the terms used to identify situations common to the world of the caches and also in the implementation. For the terms, it follows an introduction to the main terms:

- Virtual address: if the Virtual Memory is on, the addresses handled by the processor are virtual, i.e. they do not correspond to the addresses in which the wanted data is placed. They address the page table to be translated into physical addresses.
- Physical address: the location in which the data is stored.
- Page table: a data structure that allows for the conversion from a virtual address to a physical one.

- PTE[19]: the PTE an entry of the page table. It contains the PPN[20] and other bits to define the status and the permissions relative to the pointed page.
- VPN[21]: the VPN is a piece of information useful to address the correct entry in the page table.
- PPN: the physical page address.
- Page: it is the minimum quantity of data exchanged between the Secondary Storage and the Main Memory.
- Page fault: if the Main Memory was a cache, the page fault would be the *miss*. If the requested page is not resident into the Main Memory, a page fault occurs.

The implementation is different from that of the caches because of the slowness of the transfers which involve the secondary storage. Most of the time, a page fault is handled in software. This means that the hardware should only recognize this event and trigger a page-fault exception to give the processor to the Operating System.

2.3.2 How does it work

The minimum part of the Memory which can be transferred is a Page. The Page table is the structure which holds the information to convert a virtual address to a physical address, and to be used should be resident in the Main Memory (also pieces of the page table can be brought back to the secondary storage, but if needed it can be moved to the Main Memory). It can be a simple table, in which each entry contains the physical location of a page. The address of the location of the main page table is saved into a register and is called *root page table PPN*. The processor gives to the Memory System a virtual address, which is composed of a VPN and a *page offset*. The VPN is an offset for the page table and, added to the *root page table PPN*, addresses a precise PTE. The physical address of the searched page is found in the PTE. To locate the data into the physical page is needed the *page offset*, hold into the virtual address. This is the only part of the virtual address that is also physical.

Different types of page table

The organization of the page table can be different and depends upon the dimension of the virtual address space. For a 64-bit system with 4 KiB pages, the 12 less significant bits of the virtual address identify the offset on the physical page. Even though, it's impossible to have one contiguous page table with $2^{64}/2^{12} = 2^{52}$ entries. There are many possible solutions to this problem, which are deeply elaborated in [22], but the RISC-V approach is to use a *Hierarchical Page Table*.

[19] Page table entry

[20] Physical page number

[21] Virtual page number

The VPN does not directly address the entry of the page table in which the PPN of the physical page is contained, but is composed of more parts. The first part is an offset on the root page table, but the found PTE contains the PPN of another level of the page table, and so on. At the first level, the PPN of the physical page is found.

This way it is possible to take into the main memory only parts of this page table, and there's no need for keeping them always ready.

In figure 2.1 and 2.2 is visually explained how the virtual address is translated into a physical one. In figure 2.3 is shown the modification for a 64-bit address and a page table with 3 levels of indirection.

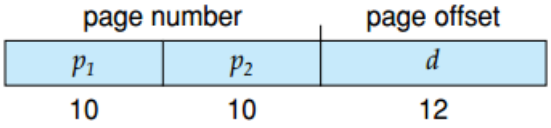

Figure 2.1. A 32-bit vaddr for a 2-level page table. (from [22])

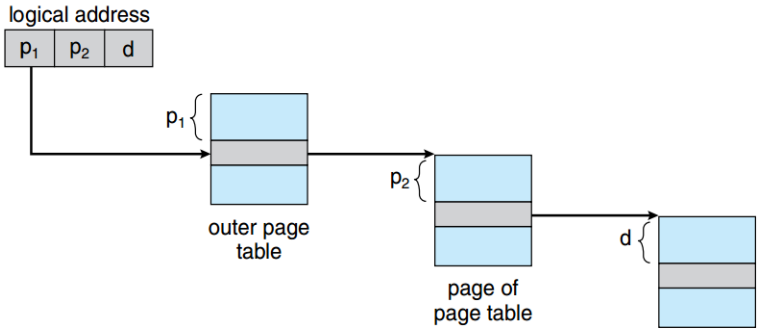

Figure 2.2. Address translation for a two-level 32-bit paging architecture. (from [22])

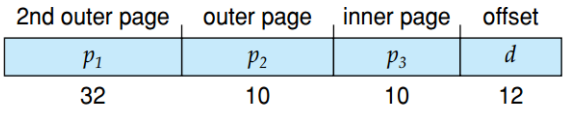

Figure 2.3. A 64-bit vaddr for a 3-level page table. (from [22])

RISC-V limits its virtual address space to only a portion of the physical one, therefore not all the 64 bits of the virtual address are used (see 3.3.3). This limits the page table size.

The Page Size An important parameter of a paging system is the size of the physical page. This choice influences the granularity of the minimum entity that is transferred to and from the Secondary Storage. RISC-V, for example, supports **4 KiB**, **2 MiB**, **1 GiB** pages [16] and this is an ISA choice. The main arguments to use larger or smaller pages are:

- Larger pages mean that one TLB[22] entry exploits a higher spatial locality because it can address more data. Large pages can highly reduce TLB miss rates [9]. Exaggerating, if the page is large as the Address Space, the miss rate of the TLB is 0, because every address is mapped by the only entry (this would be impossible and futile).
- The previous extreme situation points out the main issue: paging is useful to let programs fit the memory, ideally keeping there only their more used portions and the parts in execution. If the page is *too large*, and this depends on the application, the memory becomes *fragmented*. Because of the scarce granularity of the pages, entire chunks of the programs are brought to the main memory even if only small parts of them are useful. The memory is then full of useless data.
- Larger pages mean a higher "page-fault penalty" to bring a high quantity of data into the main memory.

Operating Systems and commercial architectures like x86-64 can support the same page sizes as RISC-V [9].

2.3.3 The TLB

Every time an address translation should be performed, one or more accesses to the memory should be done. This is not practical: with three levels of indirections, obtaining a data would require 4 memory accesses. For this reason, the page table is cached in special structures to allow for a faster translation, and this happens thanks to the locality of the requests. This cache of the page table is called TLB and holds all the information to directly address the physical page, starting from a virtual address.

The TLB can have its entries organiazed in a *fully associative*, *n-way set associative* or *direct mapped* way. Each set can be indexed and each entry is generally composed of:

- PPN: the physical page number of the requested physical page
- **TAG**: as for the caches, each entry is tagged. When a request is performed, the incoming tag is compared with the tags of the entries in the indexed set; then, a hit or a miss can happen.
- **V**alid bit: to mark if the TLB entry is either valid or invalid. An invalid entry is not considered for the tag comparison.
- **Access permission bits**: as read, write, execute bits. They set the access permission of the addressed page.

[22] Translation Lookaside Buffer

- **Dirty page bit**: a bit to indicate if the addressed page has been written since the last time it was clean.
- **Other fields**: there can be an ASID[23], a reference bit, an access bit, a global bit, etc.

2.3.4 Modern Hardware for Virtual Memory

If not otherwise specified, the material and the ideas of this section are taken from the [9]. The pictures are reworks of the ones found there.

The hardware necessary for supporting Virtual Memory should be balanced to maximize the benefit of this technique and the cost in Energy, Power, and Area. The processors of today work with many processors inside. Usually, for high performances, a structure like the one in figure 2.4 is used.

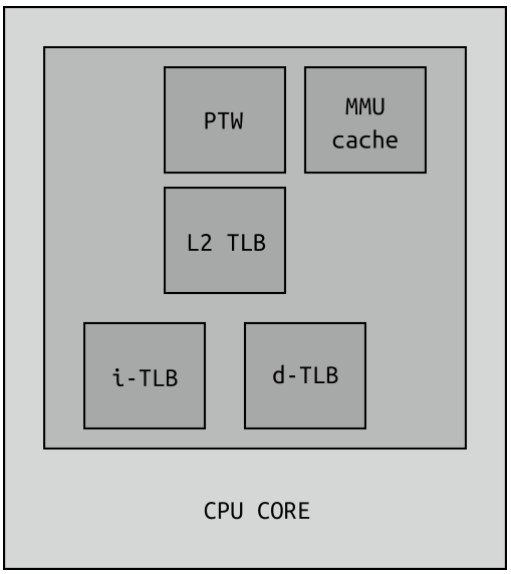

Figure 2.4. Address translation hardware in modern computers.

The TLB is usually organized on more levels, there can be specific hardware to handle its misses and the PTW[24] itself has its cache to speed up the access to the leaf page table entries. In the following, each aspect will be discussed more in detail.

L1 TLBs

As already introduced, it's frequent to have split TLBs not to incur in structural hazards for front end - back end contentions. A private i-TLB is used to translate instruction

[23] Address Space IDentifier

[24] Page Tagle Walker

addresses by the front end and a d-TLB is used for data addresses translation by the back end. Data and instructions exhibit different locality pattern and the organization and replacement policy of each TLB can be chosen accordingly. Additionally, if the processor implements an Out of Order back end, untranslated data addresses can also do not harm performances if other data requests proceed in the meantime: for this reason, the i-TLB is particularly important for performances. Intel's architectures support hyperthreading: more threads can be enabled within a single core. In its Skylake architecture, the i-TLB is divided into two fixed halves, one per allowed thread, to give each thread an equal amount of space.

As discussed in 2.3.2, an Operating System can use multiple sizes for its pages. This is also officially supported by the RISC-V ISA 3.3. Allowing multiple page sizes can be an issue and some trade-offs should be evaluated.

If the TLB is fully associative, no problem exists: it's enough to add to the entry the information on the page size. However, many L1 TLBs are located on the critical path of the corresponding Cache 2.3.6 and, additionally, they can consume up to 15% of the processor energy. This implies that they should be made as simple as possible, preferring low associativities.

The problem can be understood thinking about TLB addressing. The full virtual address is interpreted as follows:

- Tag
- Index
- Page offset

The page offset is not used to address the TLB, because it's already physical. The TLB gives back a physical page number, which is independent of the offset.

The Index bits are the less significant ones, once the *page offset* is removed. This is extremely important for TLB performance because this way the spatial locality is maximized, together with the hit rate: adjacent pages should go in different sets to decrease competition. The problem is that, obviously, for different page sizes the *page offset* has a different length. Thus, even the position of the Index part of the address is a function of the page size.

Since the page size cannot be known before the access, for L1 TLBs usually different TLBs are present for different page sizes. Each request is given in parallel to all the TLBs and can either hit (in only one of them) or miss; when an entry is brought to the TLB after a miss resolution, it is saved in the right structure. In 2.5 a rielaboration of the picture present in [9] is reported. For *Sv39* RISC-V processor, the Virtual Address is composed of 39 bits, of which the 12 less significant ones are *page offset* (see 3.3.3). The picture highlights which bits address which TLB, under the following hypothesis:

```
L1 TLB (1 GiB pages): 2-way set-associative with 8 sets.
L1 TLB (2 MiB pages): 2-way set-associative with 4 sets.
L1 TLB (4 KiB pages): 2-way set-associative with 2 sets.
```

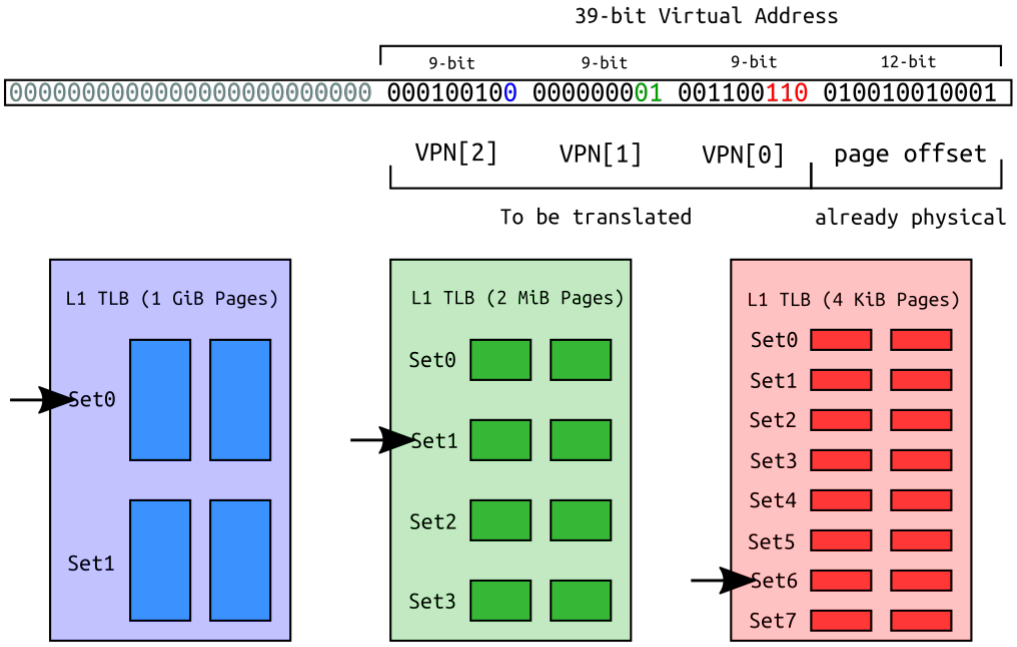

Figure 2.5. Sv39 RISC-V parallel lookup for multiple page size TLBs. Index bits for different TLBs are highlighted in different colours.

L2 TLBs

L2 TLBs, like L2 Caches, should present a high hit rate to avoid the L1 miss penalty to become too high. This means bigger structures, higher associativities, smarter replacement policies (like Pseudo LRU). They are usually shared between data and instructions.

As for the caches, the TLB hierarchy can present different inclusion policies:

- Strictly Inclusive: if an entry is present in L1, it should also be present in L2. It is useful for multi-core designs: if a processor modifies a particular entry, that entry should be invalidated also in all the privates TLBs of the other processors. A strictly inclusive policy can minimize these invalidation messages because if the entry is not present in the L2 TLB, it is not present in the L1 one.

- Mostly Inclusive: when a miss occurs in L1, the entry is filled in both L2 and L1. But there is no constraint on evicting the entry in L1 if it is evicted in L2. Each TLB can maintain its replacement policy.

- Exclusive: entries cannot be present in more than a TLB. Useful for design with less area/power budget, because no multiple entries are maintained.

Multiple page size support A higher access time is tolerated in L2 structures to obtain a high hit rate, therefore it is possible not to split the L2 TLB to support a parallel lookup for multiple page sizes. The requests can be serialized on the same structure. This approach is called *Hash-rehashing*: the base request is performed assuming one of the page sizes, usually the base one, and in case of miss another request is performed changing the page size.

To improve the performance of *Hash-rehashing*, it is possible to use predictors, perform parallel lookups and fire page table walks after the first miss, to partially overlap the miss penalty with the rehashing latency.

Another different way of addressing the problem is using skewed TLBs. More details on [9].

Page Table Walks

When an L2 TLB miss occurs, the information of the last level PTE (leaf page table entry) should be extracted from the page table. This process is called page table walk and can be performed in hardware or software.

Software page table walk The L2 TLB miss triggers "something" that transfers the control to the Operating System, which is in charge of completing the page walk in software.

Hardware page table walk To obtain high performances, a special structure is in charge of performing the page table walk and refilling the L2 TLB: this component is called PTW. The number of memory access it should do is variable and depends on the page table structure: for a three levels page table like the 64-bit RISC-V one, three accesses are needed, like shown in figure 2.6.

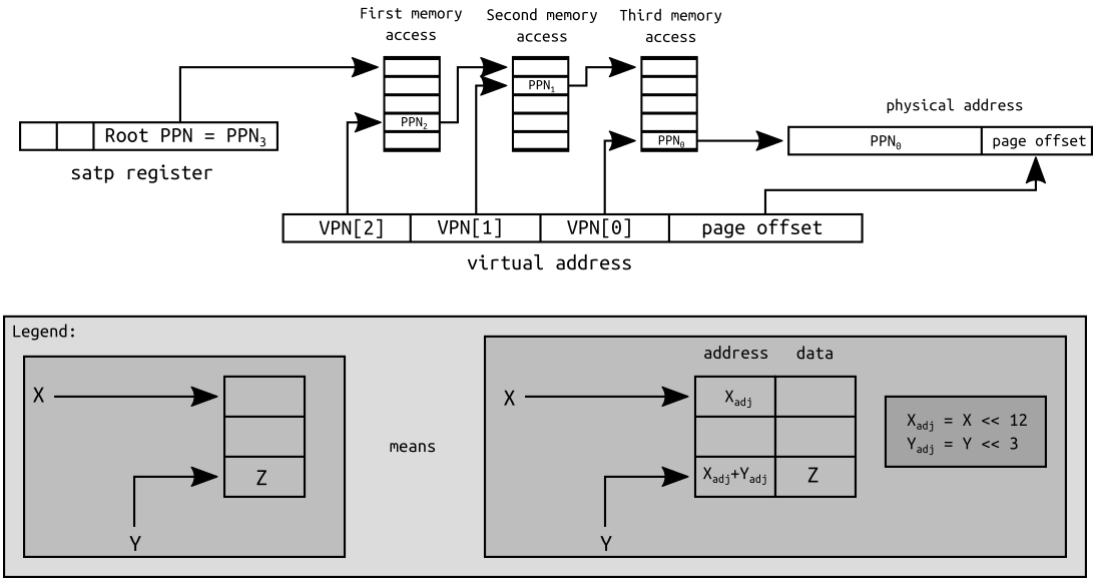

Figure 2.6. Sv39 RISC-V page table walk.

The PTW can be designed to support many outstanding misses. In this case, it is composed of a buffer that can hold many in-flight "requests". A special control unit can fire the memory requests and update the buffer when something is returned. Simpler implementations limit the PTW to handle only one miss at a time. In Out of Order processors, a page table walk can be overlapped with useful work.

The PTW performs memory requests, and even if they can be done to the L1 d-Cache (PTEs are considered as data [9]), architectures like the **AMD Opteron** ask directly to the L2 Cache, therefore the L1 d-Cache does not cache PTEs. This is only a design decision but it's not a requirement [25].

Software vs. Hardware The Hardware approach, even if it has a cost in area and power, is more performant. The main drawback is that it is less flexible because it is designed assuming a particular page table organization. In the case of RISC-V, the lack of flexibility is not a problem, because the page table organization is fixed and set by the ISA (the last sentence is an opinion of the author).

MMU Caches

The hardware page table walking can speed up the TLB miss handling but can require many memory accesses depending on the page table structure. A possible way to reduce this penalty is to use a dedicated cache to store the intermediate PPN to avoid the first memory references.

If the base page size is 4 KiB, each TLB entry maps 512 doublewords, which are much more then the number of doublewords in a cache line. If the page table is structured on 3 levels, the second one maps 512 *Mebipages*: 256Ki doublewords. A lot of requests will likely be done for pages belonging to a *Mebipage*. This fact is proved by the low variability of the most significant bits of a virtual address. If only the least significant bits of the VPN change, all the requests can be considered local to the same *Kibipage*, *Mebipage*, *Gibipage*. The TLB contains the leaf information of the PTE, the MMU cache keeps the information of the other intermediate PTEs.

Intel and **AMD** have tackled the problem of designing such a cache in different ways. A great paper compares the performances of these solutions, adding also other three possible structures [24].

Page table caches If the MMU cache is addressed by the physical address of a page table level. The L2 Cache and the *Page Walk Cache* of **AMD** behave like a *Page Table Cache*. Each entry is "tagged" by its PPN and VPN part (together, they form the physical address of a PTE). This is nothing different from a "physical" cache request. A MMU*cache* designed in this way is a dedicated cache for PTEs. This structure can exist both unified or split. In the first case, the information relative to PTEs of any level is stored in the same cache. On the contrary, a split design use *N-1* (*N* is the number of page table level) different caches, each keeping only PTEs from a particular page table level.

Referring to figure 2.6, it is possible to think the page table cache as the ones proposed in figure 2.7 and 2.8.

For example, if

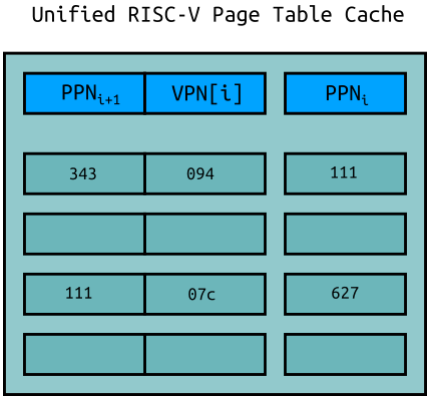

Figure 2.7. A possible Unified Page Table Cache, interpreted in Sv39 RISC-V key.

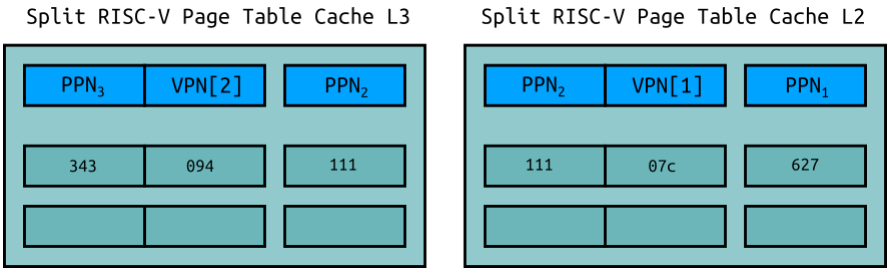

Figure 2.8. A possible Split Page Table Cache, interpreted in Sv39 RISC-V key.

```
Root PPN: 343
Virtual address: 094 07c 012
```

Then the first memory reference accesses $PPN_2 = 111$ and the second accesses $PPN_1 = 627$. This is used for the third and last memory reference to obtain PPN_0. This information is not saved in the MMU cache, because it will be already stored in the TLB.

The greatest disadvantage is this physical tagging: even though the cache is fast and no other data compete with the PTEs, the number of memory accesses hasn't dropped. A possible point in favor of this architecture is that the entries are independent of the root page table PPN because all the tags are physical (this is an opinion of the author of the thesis).

Translation cache These caches present entries tagged with parts of the virtual address. In practice, they have as tag a variable number of parts of the VPN.

This allows cutting some memory access if all the intermediate VPN parts are present. If more combinations of the VPN pieces are present, the one that corresponds more to the incoming virtual address is selected. For a possible Sv39 RISC-V implementation, refer to figures 2.9 and 2.10.

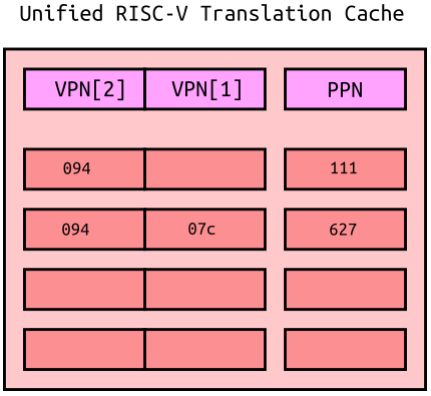

Figure 2.9. A possible Unified Translation Cache, interpreted in Sv39 RISC-V key.

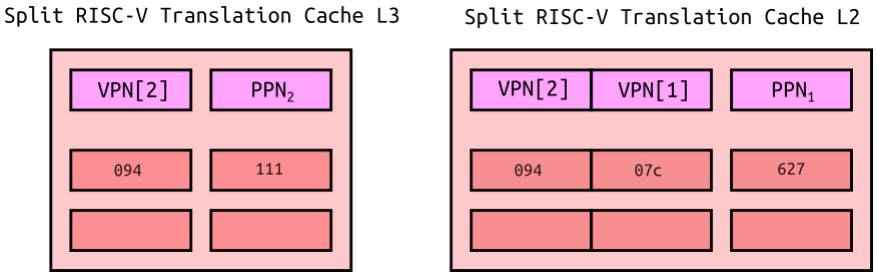

Figure 2.10. A possible Split Translation Cache, interpreted in Sv39 RISC-V key.

For example, if

```
Virtual address: 094 07c 012
```

Then in only one MMU cache reference $PPN_1 = 627$ is obtained. This is used for the second and last memory reference to obtain PPN_0. This information is not saved in the MMU cache, because it will be already stored in the TLB.

Since the addressing is completely virtual, something should be made to avoid address space invasions. If the root page table PPN changes, maybe the cache should be flushed if no other countermeasures are taken.

Translation-Path Cache (TPC) A TPC[25] is only a Translation Cache in which the redundant traces are merged to form the most complete one. When a virtual address is presented, a trace with differents lengths can hit. Depending on this length, a different PPN is returned together with the information on the type of the hit.

This structure can only be unified. The RISC-V flavoured translation path cache is presented in figure 2.11.

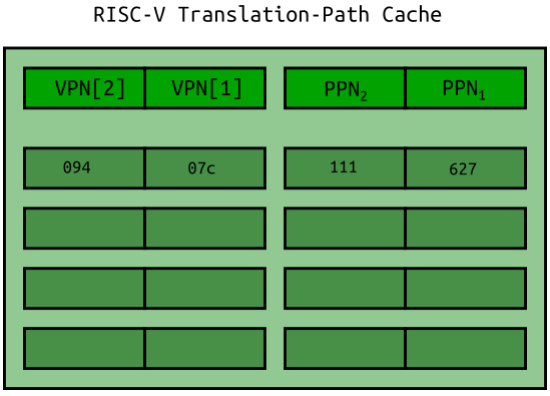

Figure 2.11. A possible Translation-Path Cache, interpreted in Sv39 RISC-V key.

For example, if

```
Virtual address: 094 07c 012
```

The hit is full, and $PPN_1 = 627$ is returned. The miss is resolved with only the last memory request. On the contrary, if

```
Virtual address: 094 119 012
```

[25]Translation Path Cache

The hit is partial, and $PPN_2 = 111$ is given back. In this case, two memory requests are needed.

Like for the Translation Caches, a flush could be necessary when the address space is changed.

2.3.5 Memory System, Speculation and Exceptions

If the processor performs speculative execution, it's not assured that the memory references will be committed. The more common situation is to speculate on the load instructions, because, if they have the permission to be executed, they are supposed to be harmless (otherwise, there are security issues 5.11). For sure, no exception relative to speculated instructions should be executed and resolved until the instruction is sure to commit.

Speculation has a cost in terms of energy because in real cases some speculations are wrong. The performance boost of the speculation is not granted and depends both on the correctness of the predictions and on how much is speculated. Usually, to avoid performance losses in programs with a high rate of TLB/cache misses, starting from a certain level of the hierarchy these events are served only if the instructions are no longer speculative. Misses that reach the DRAM have a so high penalty that are served only if the instruction is no more speculative [6].

2.3.6 Different TLB-Cache organizations

The integration within the same architecture of the Virtual Memory and the Memory Hierarchy is not trivial and needs attention.

Caches receive an address and return data, but there are two types of addresses. There is not a single strategy to be followed at design time, but some choices bring with them more problems than others.

A generic cache is addressed with an index and then the incoming TAG is compared with the others kept in the set. These two different parts can be virtual or physical, or a combination of them. If one of the parts is physical, it means that probably a TLB translation has occurred *before* the cache had been addressed.

Physically indexed, physically tagged This is the easiest case. The virtual address is first translated into a physical one, therefore the TLB access happens before the cache access. This access can happen in a different cycle or in the same. In the last case, the critical path could be exacerbated and the TLB would be limited in size. Access permissions are checked when the address is translated, thus the accessing mechanism to the memory hierarchy is not modified. The only drawback is that the TLB could be on the critical path.

A possible trick that allows reducing the penalty of the TLB on the critical path is discussed in 2.3.6.

Virtually indexed, virtually tagged The cache is accessed with the virtual address and the TLB is used only in case of a miss. This allows it to be large because it does not

impact the cache hit time; it is only part of the miss penalty which is weighted by the miss rate.

Unfortunately, there are some important drawbacks:

- The access and protection bits relative to the accessed page should be copied into the cache because the TLB is no more accessed if the request hits. This can be quite costly. [26]
- The problem of the homonyms in different Address Spaces forces the cache to be flushed on a context switch, or store an ASID near each TAG, or adopt other countermeasures (see 2.3.7).
- The problem of the synonyms in different Address Spaces or even in the same one (see 2.3.7) forces particular and complex designs or OS[26] support.

Virtually indexed, physically tagged This solution is clever because the physical tag avoids the problem of the homonyms 2.3.7. Also, while the cache is virtually indexed, its access time can be overlapped with the time the TLB uses to translate the tag.

The main problem is with synonyms. If the index is virtual, more copies of the same data can be present in different sets. For read-only caches is a problem of performance (the miss rate can increase), for r/w caches there is the possibility to have two copies of the same data, one updated and the other stale. If there's no dedicated hardware to solve this issue (for example, the AMD Opteron has it [6]), the OS should be careful and enforce the rule "same physical line in the same set". This can be done with the *page coloring*: the OS forces all the aliases to have virtual addresses with the bits that are used to index the cache all equal. This way, all the aliases are put in the same set because the "set" associativity is extended to the virtual memory.

There is a trick to avoid additional support: the virtual address is not completely virtual, because the less significant part, the *page offset*, is physical. If the index bits used to address the cache are chosen only among the *page offset*, the cache becomes de facto *physically indexed and physically tagged*. If the minimum page size is fixed, as for RISC-V [16], the cache parameters are constrained. The total number of indexable sets multiplied for the bytes in a line should be less than the bytes addressable on the page. In practice, the index portion of the address should be contained in the *page offset* portion.

A possible implementation for RISC-V is presented in figure 2.12.

Physically indexed, virtually tagged A strange combination, because the cache should be first indexed and then tagged. This structure puts the TLB on the critical path for the cache hit time for index translation and brings with it a lot of disadvantages due to the virtual tagging. For conventional designs, this solution is seldom used (even if not recent, [27] testimonies that, until that date, the only processor that adopted this organization was the Mips R6000).

[26] Operating System

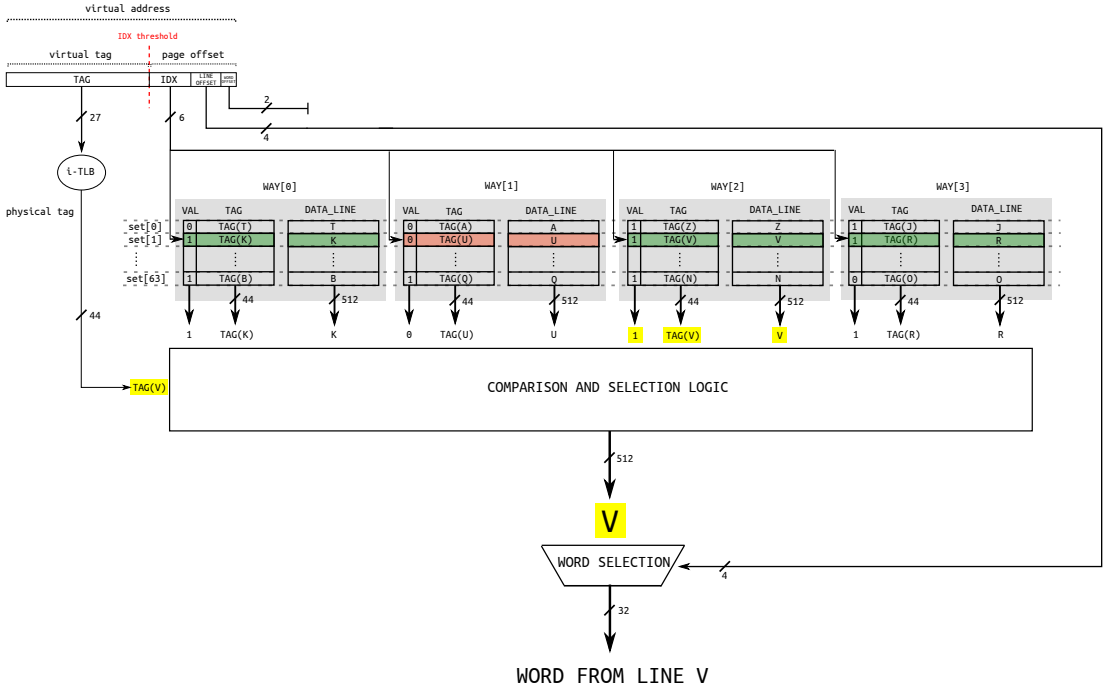

Figure 2.12. Possible VIPT organization with "physical indexing" for an Sv39 RISC-V Processor. In the example the entire physical memory is limited to be addressable with 56 bits, the cache has 64 sets, the line size is 64 bytes (512 bits) and the word size is 4 bytes (32 bits).

2.3.7 Homonyms and Synonyms

Homonyms

Two people (physical addresses) are homonyms if they share the same name (virtual address).

More formally, the problem can be viewed by two different perspectives:

- Different physical addresses are mapped with the same virtual address.
- Two or more virtual addresses are equal but refer to different physical addresses.

The first view is useful to understand why homonyms exist: one of the great advantages of the virtual memory is that the same address range can be used to map different portions of the memory.

The second view can help to understand the problem: there's the need for mapping correctly the address translation and the memory request.

The same virtual address can correspond to different physical addresses only if the Address Space is changed.

Homonyms in a memory:

Homonyms can lead to errors because if they are not correctly handled a wrong line can be accessed by the processor during a read or a write.

If the indexing is virtual and the tagging physical, no problem exists. The only negative effect is that homonyms are mapped in the same set, so the competition can be increased for those sets. The accessed set is always the same because of the same virtual index, but the check on the tag is physical, so only the correct line is returned.

If the tag is virtual there can be errors. This is because the tag comparison can give false positives and return the wrong data even for readings.

Solutions:

- Use a SASOS[27]. In this case, no homonyms can exist.
- Use physical tags
- Keep an Address Space IDentifier near each line of the cache and compare it each time.
- Flush the cache (and update the upper hierarchy if needed) when the Address Space is changed

For a TLB, homonyms are common because the same address maps to different pages depending on the root page address. It is important to point out that for a TLB both the indexing and the tagging are always virtual. Therefore, the TLB should be flushed on context switches or should have ASID for each entry.

[27]Single Address Space Operating System

Synonyms

Two words (virtual addresses) are synonyms if they have the same meaning (physical address).

- Different virtual addresses refer to the same physical address.
- A physical address is mapped by more virtual addresses.

Synonyms can be present in different Address Spaces, but can coexist also in the same [28] [27].

Synonyms in a read-only memory:

they do not harm the correctness of the execution, but can slow it down and increase the energy consumption. This is because the physical address (and therefore, the line) is always the same for each synonym.

If the indexing is virtual and the tagging physical, the same line can be present in different sets. This is because different virtual addresses can look for a line in two different sets, even if the wanted line is the same. This is not good because the effective capacity of the cache can be reduced by a higher competition: the same line can occupy two or more places. In the worst case, it can occupy each set. If the cache is direct-mapped the address is completely virtual (there is no tag) and in the worst case, the cache is filled with the same line. If the cache is fully associative, no harm is done, because there is no index.

If the indexing is physical and the tagging virtual, the same line can occupy more than one position in the same set. This is because the indexing of the set in the cache is physical and is always the same, but the tag is different for each synonym virtual address. Therefore it can miss even if the wanted line is present in the set because the compared tags are virtual and can be different. This leads to a request to the memory hierarchy (which uses a full Physical Address) that will end up with the requested line. This line is put in the same set in a position that depends on the replacement policy. Thus, more equal lines can coexist in the same set. In the worst case, the entire set is filled with the same line. If the cache is direct-mapped, no tag exists, so the address is fully physical and there is no issue (each set is already filled with the same line... the only one of the set!). If the cache is fully associative, the problem is severe, because there is only one set.

If the entire address is virtual, the same line can be present everywhere in the cache, and, in the worst case, the entire cache is filled with the same line also for a set-associative cache.

Synonyms in a read-write memory:

they are a problem and lead to losses of data and errors. This is because the coherence cannot be maintained if multiple copies of the same data are present in a memory which can also modify them.

Synonyms can be present through different address spaces or in the same address space.

Solutions:

- Use a full physical address
- Hardware checks during cache access
- Flush the cache (and update the upper hierarchy if needed) when the Address Space is changed (useful only if no synonyms exist within the same address space)

- Aid of the operating system (like page coloring) to map synonyms in the same set and using the physical tag

For a TLB, synonyms are two different addresses that map to the same page. It is important to point out that for a TLB both the indexing and the tagging are always virtual. Therefore, the TLB can have multiple copies of the same page in multiple places if the operating system does not avoid it.

Chapter 3

The RISC-V ISA

What is an ISA? An ISA is a set of rules and specifications that abstract the processor to an assembly level. In practice, it specifies the functionality of the processors, and the minimum rules necessary to make it works in a certain way. Generally, it sets how the memory is accessed, how the memory behaves, the supported instructions and operations, the type and the size of the operands and data, how exceptions are handled.

There are a lot of different ISA, and the book [6] talks about their peculiarities in *Appendix A*.

History of RISC-V On the main site of RISC-V is present the history of this ISA, which is here summarized. It was born in 2010 in *UC Berkeley*, thanks to Prof. Krste Asanovic and graduate students Yunsup Lee and Andrew Waterman, as part of the *Par Lab* guided by Prof. David Patterson. After research, publications and achieved milestones, since 2015 is maintained by the *RISC-V Foundation*, a "*non-profit corporation controlled by its members*" which "*directs the future development and drives the adoption of the RISC-V ISA*" [2].

3.1 Peculiarities

The RISC-V ISA is characterized by the following qualities:

- **Open**: RISC-V is an open ISA, therefore it can be adopted without the need for paying anything to anyone, under the conditions of its license, which allows commercial use.
- **RISC**: it is inspired by the *RISC* principles, which are old and well experienced, and favor hardware simplicity.
- **Simple**: simple hardware usually means smaller chip area, easier design, and verification.
- **Modular**: unlike *incremental ISA* like the *80x86*, which forces newer implementation of the ISA to support past weaknesses, RISC-V is *Modular*. This means that it has

a solid and *frozen* base set, but all the extensions can be optionally implemented or not, depending on the needs [8].

- **Frozen**: after periods of discussions and studies, the main set and extensions become frozen. This means that they will never change and can be a certainty for all those who want to develop hardware and software that use it.

3.2 The core set

The main integer core set **RVxxI** is available in different formats, each of them supporting words with different sizes. Since LEN5 is a 64-bit system, the implemented set is the **RV64I**. For information about the supported instructions, refer to the official manual or to the "RISC-V Reader" ([8]) for a quick reference.

Core memory operations As it will be remarked in the next section, the RISC-V is a *load-store* architecture. It is compatible with aligned memory accesses and the support for the misaligned ones is optional. Loads and stores can address *Doublewords* (64-bit), *Words* (32-bit), *Halfwords* (16-bit) or *Bytes* (8-bit). The condition of alignment is different for each of them. For example, a byte address is always aligned in a byte-addressable memory system. A doubleword address is aligned only if its three less significant bits are zero.

3.2.1 Privilege Modes

RISC-V allows three different *privilege modes*:

- Machine mode
- Supervisor mode
- User mode

The current *privilege mode* is encoded in particular registers called CSR[1]s. A *privilege mode* defines which operations are allowed and which rules should be respected during their execution. If a not-allowed operation is executed, an exception is raised. [16]

Machine Mode The machine mode is the most privileged and the only mandatory one. Simple microcontrollers usually support only this mode. When operating in this condition, the processor has full access to the memory, the CSRs, to I/O, and can execute every supported instruction [8].

This is dangerous if the processor is used to run untrusted code. For this reason, RISC-V allows for two restricted operative modes.

[1]Control and Status Register

Supervisor Mode Normally, an OS is executed in *Supervisor Mode*, which is the most privileged between the two. It has not whole access to the registers and can't execute all the instructions, but can handle exception after an M-mode re-routing and has particular access privileges to the memory if Virtual Memory is used.

User Mode On the other hand, *User Mode* is the less privileged mode available. It is used to execute untrusted code because the possibilities of harm are almost nil.

3.3 Memory System specifications

All the information reported in *italic* paragraphs are literal quotes extracted from [15] and [16]. The information reported below is only part of the actual specification, selected to understand more deeply the LEN5 core. For further details, refer to the official manuals.

3.3.1 Load/Store

"*RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.*" [15]

The only way a Load-Store processor can access the memory is with the "Load" and "Store" instructions. All the other instructions can operate only on the internal registers.

Different types of ISAs present different internal storage organization and set different rules for the addressing of the operands. There are **stack architectures** and **accumulator archtectures**, in which at least one operand is implicitly addressed. Other architectures are based on general-purpose registers and the operands are always explicitly addressed. If it is possible to address an operand directly from the memory, the architecture is classified as **register-memory**, like Intel 80x86. Otherwise, if the only way to transfer data to and from the memory is the load and the store instructions and the operands are always kept in the registers, the architecture is a **load-store**, like ARM and RISC-V [6].

"*RV32I provides a 32-bit user address space that is byte-addressed and little-endian.*" [15] "*RV64I extends the address space to 64 bits. The execution environment will define what portions of the address space are legal to access.*" [15]

A 64-bit byte-addressed Address Space means that, if the execution environment allows it, there are 2^{64} available Bytes which can be addressed.

There are two possible ways to organize the bytes in a generic word: **Big Endian** and **Little Endian**. Assuming a little-endian convention, the less significant byte of a word is stored in the lower position. On the contrary, if the ISA follows the big-endian directive, the less significant byte is stored in the most significant position of the word.

In a RISC-V compliant address space, a word of 4 bytes stored at the address 0x0 will have its less significant byte stored at address 0x0 and its most significant byte at address

0x3.

"*Loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.*" [15]

3.3.2 Memory Model

Up to now, the reference manual specifies that the section is out of date. All this information should be read with care, and only the parts useful for the uniprocessor case will be taken into account. Look at the official specification for an updated document.

FENCE.I When many entities use a Memory System, some physical parts of it are shared and others are private. This is intuitive for a multi-processor system when a single memory can be shared between more processors but is true also when more parts of a single processors work independently. The *front end* and the *back end*, for example, can contact the memory system using two different paths: the i-Cache and the d-Cache respectively. If the program provides modifications of an instruction into the memory, the *back end* of the processor will perform a *store* into the d-Cache, because this is the only way to obtain a memory modification in a Load/Store architecture. When this modified instruction will be visible to the i-Cache? If no actions are taken, there are no guarantees that eventually the i-Cache will see the modified value. For example, if the instruction was already present in the i-Cache, that instruction becomes stale at the moment in which it is modified in the d-Cache. Therefore, that stale instruction should be removed from the i-Cache. A simple way to do this is with a generic i-Cache flush. But even if a flush is performed, there's the risk for coherency problems inside the same memory system for the same processor. The i-Cache refills from an upper level of the memory hierarchy. After a finite amount of split levels, there is a level shared with the data: for example, a shared *L2 Cache*. If that "meeting point" is not synchronized with the d-Cache, the i-Cache will go on fetching stale instructions. The only way to do this is to ensure that the modifications that are done in the d-Cache are backed up at least in that "meeting point".

"*The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart.*" [15]

At assembly-level, this is the instruction that triggers this synchronization. How this is performed depends on the implementation.

3.3.3 Virtual Memory

Virtual Memory is supported and regulated by the ISA itself. It can be enabled or not, and if enabled it rules the behavior of the Memory System. Up to now, there are three types of possible implementations: *SvX* indicates that the virtual address is composed of X bits:

- Sv32 (for 32-bit systems)
- Sv39 (for 64-bit systems)
- Sv48 (for 64-bit systems)

For RV64 systems, the virtual address is translated into a 56-bit physical address.

Important CSRs and CSR bits

Machine Status (mstatus) Register Even though complete information can be found on the reference manual, three important bits of this register will be directly cited:

- "**MPRV**: *The MPRV (Modify PRiVilege) bit modifies the privilege level at which loads and stores execute in all privilege modes. When MPRV=0, loads and stores behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and store memory addresses are translated and protected as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV.*" [16]
  An important point is that this bit **does not affect** the **instruction** address translation and protection, but only the *loads* and the *stores*.
- "**MXR**: *The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Figure 4.15) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. MXR is hardwired to 0 if S-mode is not supported.*" [16]
- "**SUM**: *The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 4.15) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it is in effect when MPRV=1 and MPP=S. SUM is hardwired to 0 if S-mode is not supported.*" [16]

Supervisor Address Translation and Protection (satp) Register For RV64 systems, the **satp** register has the following structure:

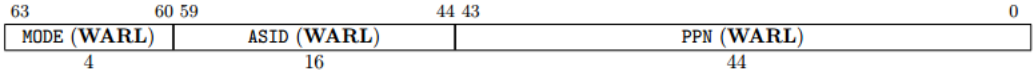

Figure 3.1. **satp** register for Sv39 and Sv48 MODEs

The MODE field hosts the actual implementation of the virtual memory, which can be:

- Bare (Virtual Memory is disabled)
- Sv39
- Sv48
- Other reserved values

The ASID field indicates the index of the actual Address Space. The PPN field holds the PPN of the root page table. The field is 44-bit wide because a physical address is a 56-bit address, and the last 12 bits represent the offset of the page, which is 0 for the *root address*.

Instructions

FENCE.VMA As for the instructions, also modifications relative to all the "in-memory memory-management data structures" [16] should be synchronized. Usually, they are performed with stores by the *back end* and focus on modifications of the page table.

"*The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to loads and stores in the instruction stream. Executing an SFENCE.VMA instruction guarantees that any stores in the instruction stream prior to the SFENCE.VMA are ordered before all implicit references subsequent to the SFENCE.VMA.*" [16]

Structures like TLBs, MMU caches, etc. can be implemented in systems which support the Virtual Memory. If a modification of a data shared among these structures happens, it should be synchronized. For example, if a *store* modifies a portion of the page table, a **FENCE.VMA** enforces the fact that all the structures which can share the modified information should be aware of it. The simplest way of dealing with such a fact is to flush all the TLBs and MMU caches and update the source from which these structures refill (e.g. the *L2 Cache*, if the PTW is fed from it).

Principles

From now on, *Sv39* will be taken as the reference, but the explained principles remain unchanged even for the other two possibilities. The page table is composed of three levels and is structured as a radix-tree. The virtual address, combined with the root page table address, is translated into a physical address. The page table structure has three levels. For RISC-V, each page table or page is composed of 4 KiB. To offset a page there's the need for 12 address bits, the less significant ones. For 64-bit systems, a generic PTE is a 64-bit doubleword: the last 3 bits of the address are reserved for the bytes inside the PTE. Therefore, each PTE inside a page can be identified using 9 bits only (bits from 11 to 3). The virtual address of 39-bit is contained in the less significant bits of the 64-bit processor address line. The bits from 63 to 39 should be equal to the bit number 38, or a page-fault exception is raised. This virtual address is composed of four different parts (3.2):

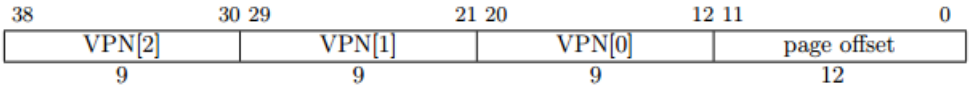

Figure 3.2. Sv39 Virtual Address

Each **VPN[i]** represents an offset for the page table level **i** and identifies a precise PTE stored inside it. The **page offset** should not be translated and it can be considered as "physical".

At the end of the translation, a physical address is returned and has the form shown in figure 3.3.

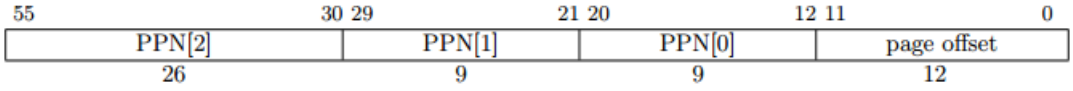

Figure 3.3. Sv39 Physical Address

Be aware: **PPN[i]** is not the translation of **VPN[i]**. RISC-V allows also for what in literature is called *Megapages* and *Gigapages*, and not only *Kilopages* [2].

If a virtual page is not a *Kibipage*, only part of the entire page table tree is traversed. There are 3 levels of page table: the translation of a *Gibipage* involves only the third level (L3). A *Mebipage* involves L3 and L2, a *Kibipage* L3, L2 and L1. If **L** levels of the page table are traversed, **PPN[i]** with **i** from **2** to **3-L** are converted. The others **PPN[j]** are not translated and correspond to the respective **VPN[j]**.

The PTE is reported in figure 3.4.

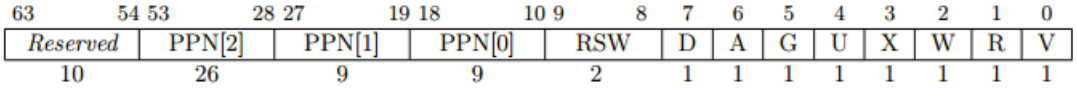

Figure 3.4. Sv39 page table entry

- The **PPN[i]** are the address of the next page table level, or, in case of a Leaf Page, addresses the physical page.
- The **RSW** field is ignored by the hardware (it is reserved for the OS).
- The **D**irty bit is the *Dirty Page Bit*. If asserted, the page has been written since the last time this bit was zero. There are two possible ways to deal with this bit (see 3.3.3).
- The **A**ccess bit indicates if the page has been read, written, or fetched since the last time the bit was zero.
- The **G**lobal bit indicates that this mapping is valid in every Address Space.

[2]Even though these names are well known in literature, throughout this thesis project they have been modified to *Kibipages Mebipages* and *Gibipages*, because only power of 2 are involved. This does not correspond to the official specification

- The **U**ser bit specifies if the page is a user page or not. A user page can be accessed only by the machine and the user mode, a not-user page only by machine and supervisor mode.
- The **R**ead, **W**rite, e**X**ecute bits are the access permission bits. For a leaf page, indicate whether the addressed page is readable, writable or executable.
- The **V**alid bit indicates if the *PTE* is valid. If not, a page-fault exception is raised.

Dirty and Access bits handling There are two possible ways to manage the **A** and **D** bits. One way requires more hardware support than the other:

- Maximal hardware support: the hardware implementation is in charge of modifying those bits, respecting atomicity and other conditions (see the manual for further details).
- Minimal hardware support: raise a page-fault exception if a page is written while its **D** bit is cleared or accessed when the **A** bit is cleared (then, the software will be in charge of modifying these bits).

For this project, the second approach was chosen.

Virtual Address Translation Algorithm The translation process is specified in the RISC-V reference manual. For convenience, it is also reported here (citing [16]):

1. "*Let a be satp.ppn* $\times PAGESIZE$*, and let* $i = LEVELS - 1$*. (For Sv32, PAGESIZE = 212 and LEVELS = 2.)*"
2. "*Let pte be the value of the PTE at address* $a + va.vpn[i] \times PTESIZE$*. (For Sv32, PTESIZE=4.) If accessing pte violates a PMA or PMP check, raise an access exception corresponding to the original access type.*"
3. "*If* $pte.v = 0$*, or if* $pte.r = 0$ *and* $pte.w = 1$*, stop and raise a page-fault exception corresponding to the original access type.*"
4. "*Otherwise, the PTE is valid. If* $pte.r = 1$ *or* $pte.x = 1$*, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let* $i = i - 1$*. If* $i < 0$*, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let* $a = pte.ppn \times PAGESIZE$ *and go to step 2.*"
5. "*A leaf PTE has been found. Determine if the requested memory access is allowed by the pte.r, pte.w, pte.x, and pte.u bits, given the current privilege mode and the value of the SUM and MXR fields of the mstatus register. If not, stop and raise a page-fault exception corresponding to the original access type.*"
6. "*If* $i > 0$ *and* $pte.ppn[i-1:0] \neq 0$*, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.*"
7. "*If* $pte.a = 0$*, or if the memory access is a store and* $pte.d = 0$*, either raise a page-fault exception corresponding to the original access type, or:*"

- "*Set pte.a to 1 and, if the memory access is a store, also set pte.d to 1.*"
- "*If this access violates a PMA or PMP check, raise an access exception corresponding to the original access type.*"
- "*This update and the loading of pte in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.*"

8. "*The translation is successful. The translated physical address is given as follows:*"

- "$pa.pgoff = va.pgoff$."
- "*If* $i > 0$*, then this is a superpage translation and* $pa.ppn[i-1:0] = va.vpn[i-1:0]$."
- "$pa.ppn[LEVELS-1:i] = pte.ppn[LEVELS-1:i]$."

[16]

Apart from the check on **PMA** and **PMP**, the translation process can only raise *PageFault* exceptions.

Chapter 4

The LEN5 core

LEN5 is an Out of Order RISC-V speculative processor under development.

4.1 Front end

The Front End is composed of a fetch unit and a branch predictor. Its function is to take the line in which the next instruction is present from the i-Cache, extract it, predict possible outcomes of a hypothetical branch and give it to the *Issue Queue*. The prediction gives information about the address of the following instruction. As soon as the real outcome of the prediction is known, the Front End should update its internal predictive structures and, possibly, correct the fetch address.
To know more about the LEN5 Front End, refer to the work of *Marco Andorno* [32].

4.2 Back end

The LEN5 Back End is discussed in the work of *Michele Caon* [33], but there are common important points that will be summarized here.

The instructions that should be executed are injected in the *issue queue*, which is a FIFO, and decoded strictly in-order. When an instruction is issued and decoded, it is sent to a *reservation station*. In parallel, an entry of the ROB[1] is reserved. This last structure is necessary to *commit* the instructions in-order, a fundamental requirement to guarantee *precise exceptions* and easy handling of the mispredictions.

4.2.1 The Load/Store Queue

All the memory instructions are issued to the LSQ[2]. In reality, there are two different queues to host the loads and the stores. The LSQ is a buffer layer between the memory

[1] ReOrder Buffer

[2] Load Store Queue

and the CDB[3]: it is in charge of keeping load and store instructions that are waiting for the generation of their address or the returning of the requested data, and updating the ROB after their execution using the CDB.
Initially, the instruction is inserted in the right queue, to wait for its Virtual Address. When it is generated, the d-TLB is queried for a translation if the *virtual memory* is on. Then, when the physical address is ready, a request to the d-Cache is made.

Memory Requests The memory must not be modified unless all the previous instructions have already committed because each instruction should be committed respecting the program order. This restriction is not limited to memory operations, because every instruction can raise an exception, and as long as there are instructions that have not committed, no permanent modification should occur.

Therefore, loads can be fired "out of order" and can be "speculative", while stores are always definitive if the corresponding request arrives at the memory.

4.2.2 Replay using a Wake-Up signal

Since each instruction is kept in the buffer until a memory response is returned, both loads and stores are put in a "sleeping state" as soon as the corresponding request is fired. In the case of a *hit*, the d-Cache will address the LSQ with an *index* after some clock cycles, writing the requested data (the precise latency depends on the number of pipeline stages in the cache). In the case of *miss*, the d-Cache will ask for the missed line to the upper levels of the Memory Hierarchy. As soon the line is returned, the d-Cache *wakes-up* in parallel all the instructions that missed for that line. When an instruction is *awake*, it can replay its request (and hopefully hit!).

For the d-TLB the mechanism is slightly different. Together with the *wake-up* signal, also the possible *exception* and the *PPN* are broadcasted in parallel (but they are used only by the *loads*, because *stores* can still raise dedicated exceptions). This forwarding happens because the instructions wanted precisely the entry that is brought from the upper level to the lower level of the memory hierarchy.
In the case of the d-Cache the data is not broadcasted, because the minimum object brought into the cache is a line, and different instructions can be interested in different doublewords belonging to the same line (the entire line is not forwarded because of the overhead of routing different parts of a vector to different entries of the queue).

Virtual Address checks Since the adopted Virtual Memory is *Sv39*, the Virtual Address is checked to respect its standard format 3.3.3. This operation is performed in the LSQ.

[3]Common Data Bus

4.2.3 Support for partial Load and Store

The core **RV64I** supports memory operations on *Doublewords*, *Words*, *Halfwords* and *Bytes* (3.2).
The LSQ can ask only for doubleword loads, because the requested information can be easily extracted from there. However, it can request stores of different "sizes", sending a doubleword with the correct information kept in its less significant bits. The d-Cache will write only the right byte(s), using the BE[4] control bits of the physical memory.

Unaligned memory operations and AMO[5] instructions are currently not supported.

[4]Byte Enable

[5]Atomic Memory Operation

Chapter 5

The LEN5 Memory System

In figure 5.1 is shown the LEN5 Memory System.

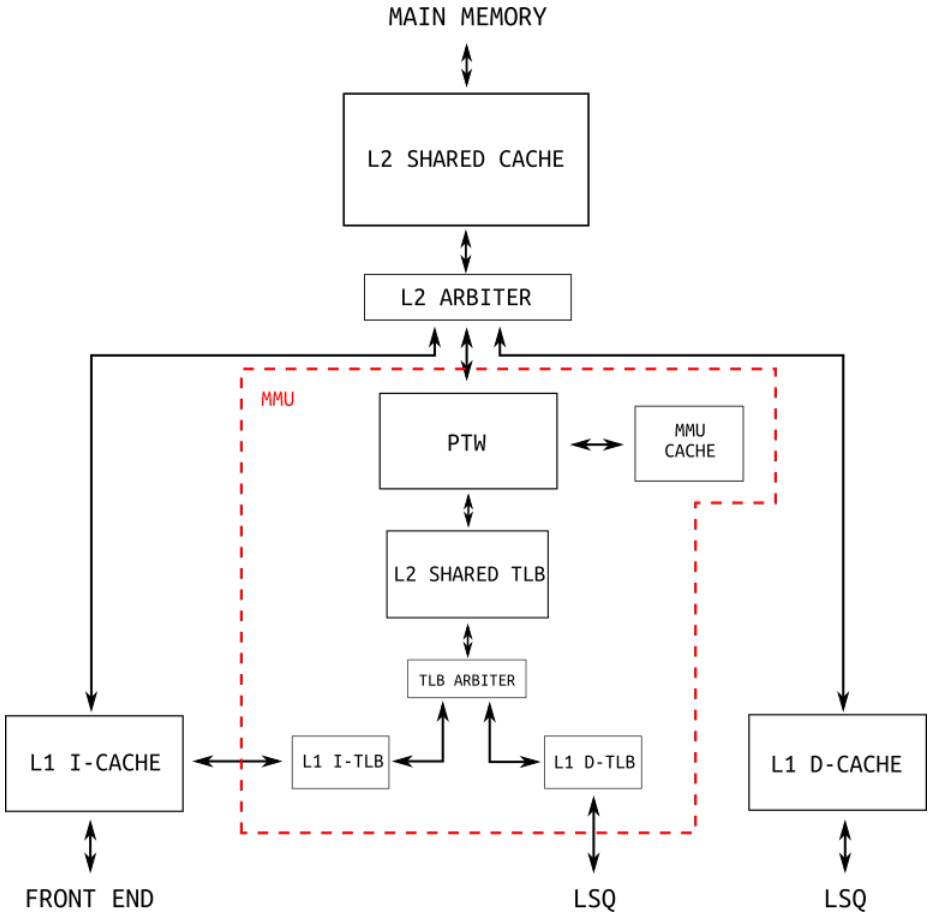

Figure 5.1. Memory System of LEN5.

Since **LEN5** is a general-purpose processor, no precise constraints were present at

design time. The rationale behind the design of this Memory System was to explore what is usually done today (2.3.4) trying to let it work.

The memory system is composed of two L1 caches (one for instructions and one for data) and a shared L2 cache. The system supports virtual memory, providing an MMU with two split L1 TLBs (one for instructions and one for data) and a shared L2 TLB. When an L2-TLB miss occurs, it makes a request to the PTW which interacts directly with the L2 shared cache for the page walk. To reduce the penalty of this walk, a translation path cache stores the paths of the radix tree based RISC-V page table. All the described blocks are, at least for now, on-chip.

The rationale under the double level of caches/TLBs is to allow the L1 elements to have a small hit time, not to impact the latency of the processor. Usually, this comes paying with a higher miss rate; therefore, the second level in the hierarchy can reduce the miss penalty, balancing the average memory access time.

Today, a double layer of caches/TLBs is quite common. The sizing of the caches is kept the more parametric as possible, but for the initial values of the parameters, it was taken a look at the ARM Cortex A53 [6]. All the caches implement a *write back* policy with *write allocate* if a write miss occurs (excluding the I-cache). The memories are described using SSRAM[1] blocks and the L1-TLBs are implemented using registers because they are small enough (like in Ariane. See the code on the GitHub page relative to [1]). Physically, the MMU cache has been temporarily described as a register file, but this can be far from the optimal case.

First hypothesis: the used SSRAM As already mentioned, the model of implementation for the used SSRAMs was the one used in Ariane. The interface/timing of that memory is real [34]. The memory is synchronous for both reading and writing and can be turned on or off using the "Chip select" control. It was hypothesized that a turned off memory does not consume energy derived from dynamic power consumption: indeed, it is turned-off every time it has nothing to do. The memory has separate input/output ports. In this design, the memory is clocked with the main clock only, and the "Output Enable" signal was not used.

Second hypothesis: 64 Byte Memory lines Another heavy hypothesis was done to design the entire system. All the memories can support 64 bytes I/O busses. All the line transfers are done in one clock cycle. This can be far from reality, but this simplification allowed to maintain the focus on the whole system.

PMA and PMP The PMA[2] and PMP[3] are not implemented in this processor. For more information, see the reference manual [16].

[1] Synchronous SRAM

[2] Physical Memory Attributes

[3] Physical Memory Protection

5.0.1 The Virtual Memory

Every time a Virtual Address is translated, the hardware should perform some checks to detect eventual exceptions. The translation algorithm is reported in 3.3.3.

Since neither PMA nor PMP is supported, the only exception that can be raised during an address translation is a **Page-Fault**. This simplifies the checks because the order in which are done is no more important: when a **Page Fault** is detected, it's not important to specify the "cause". The only thing to highlight is the **Page Fault** type:

- Instruction page fault
- Load page fault
- Store page fault

It is now explained how these checks are distributed through the Virtual Memory ecosystem, referring to 3.3.3. The main principles are that the more a check is performed, the more energy is consumed and that the checks need time to be done, therefore it's better to perform them where they cannot impact much on the latency of a block. These two ideas lead to keep the checks, if possible, in the higher levels of the hierarchy, towards the PTW.

For each point of the algorithm, it is specified where the check for the **Page Fault** occurs. The Page Fault **type** is always assigned by the front-end or the back-end as soon as they receive the answer from the corresponding TLB.

1. No checks needed.
2. No checks needed.
3. The check is done in the PTW, because it depends only upon information kept in the PTE: this means that if they are changed, a *fence.vma* would follow and the TLB structures would be flushed. Therefore, there's no need for performing this check in the lower levels.
4. The check is done in the PTW, because it depends only upon information kept in the PTE.
5. This point implies more checks that are done in a different part of the Virtual Memory hierarchy:
   - **X == 0** (the page is not executable): this check is done into the L2 TLB, because it generates a **Page Fault** only if the request is performed by the i-TLB.
   - **U == 0 && PRIV == U** (user mode tries to access a not-user page): this check is done in the L2 TLB (when the request was fired by the i-TLB) and in the L1 i-TLB, because it depends on the current privilege mode, which can change during the execution without a *fence.vma*.

- **U == 1 && PRIV == S** (supervisor mode tries to access a user page): this check is done in the L2 TLB (when the request was fired by the i-TLB) and in the L1 i-TLB, because it depends on the current privilege mode, which can change during the execution without a *fence.vma*. The **SUM** bit is ignored for the instructions (this rule is present only in the updated version of the manual: [17]).
- **U == 0 && LS_PRIV == U** (an effective user mode for Load/Store tries to access a not-user page): this check is done in the L2 TLB (when the request was fired by the d-TLB) and in the L1 d-TLB, because it depends on the current privilege mode, filtered by the **MPRV** bit, which can change during the execution without a *fence.vma*.
- **U == 1 && LS_PRIV == S && SUM == 0** (an effective supervisor mode for Load/Store tries to access a user page and the SUM bit is 1): this check is done in the L2 TLB (when the request was fired by the d-TLB) and in the L1 d-TLB, because it depends on the current privilege mode, filtered by the **MPRV** bit, and on a CSR bit, which can change during the execution without a *fence.vma*. If **SUM** is asserted, no exception is raised.
- **R == 0 && X = 1 && MXR == 0** (the page is executable but the MXR bit is cleared): this check is done in the L2 TLB (when the request was fired by the d-TLB) and in the L1 d-TLB, because it depends on the current privilege mode and on a CSR bit, which can change during the execution without a *fence.vma*. If **MXR** is asserted, no exception is raised. Since the check is performed outside the PTW, the real condition becomes **R == 0 && MXR == 0**, because a page neither readable nor executable cannot leave the PTW without any exception.

6. The check is performed in the PTW, because it depends only upon information kept in the PTE.

7. Since the software approach is used to handle the *D* bit and the *A* bit 3.3.3, two different checks are done:

   - **D** This check is performed in the L1 d-TLB, because it depends on the LSQ request type. The page can be loaded in the d-TLB for a *Load*, and then be accessed by a *Store*. Therefore, the check is performed on every back-end request.
   - **A** This check is performed in the PTW, because it depends only upon information kept in the PTE.

8. No checks needed.

5.0.2 The handshake protocol

When a block of the **LEN5** Memory System sends or receives a request or an answer from another block, a handshake protocol inspired by **AXI** is used.

Each request or answer channel is coupled with two other signals:

- Valid

- Ready

When a request is done by the block **A** to the block **B**, the **A** block control also a *Valid* signal, which indicates that the data on the request channel is valid and can be sampled or used by **B**. This last block controls the *Ready* signal, which is raised only if the **B** block is ready to sample or process the incoming request.

When an active edge occurs (in this design, a rising edge), the handshake is complete only if both *Valid* and *Ready* signals are sampled high. When this occurs, **A** can consider the request was made and change the data on the request line. For the answers, it's the same.

As specified for **AXI**, there is a constraint on the handshake signals to avoid deadlocks: given a channel, its *Ready* signal can depend on its *Valid*, but its *Valid* should be undependent from its *Ready* [20].

Figure 5.2 and 5.3 illustrate timings of possible transactions.

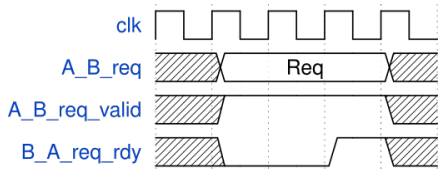

Figure 5.2. Basic handshake.

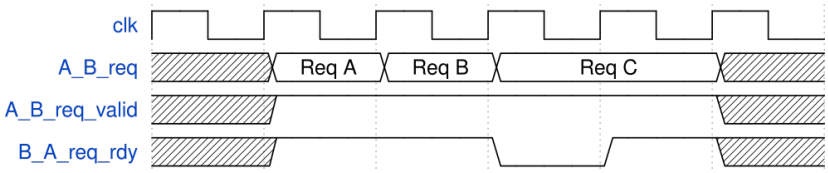

Figure 5.3. Continuous and stalled handshake.

Initially, many attempts were done to create Moore Control Units for the handling of the handshake between the blocks. The main problem with pipelined design with handshakes is that this kind of approach is not flexible and complicated. Additionally, the L1 blocks were designed to have the minimum latency possible. This has led to combinatorial controls, which can be considered as part of a Mealy Control Unit.

This is not the best for the inherent timing issues hidden in these designs. For this reason, when possible, the control of each pipeline stage or block is composed of a Moore Machine, which defines the present state and the controls that do not affect latency, and combinatorial/decoding logic to control the data flow (with the handshakes) and the operations to be done on the data. This way, if more registers or pipeline stages will be added for timing issues, it will be not too hard to adapt the design, especially if no dependencies exist between the data which fill the pipe.

5.0.3 SystemVerilog description

In the beginning, the size of the project was not clear. For this reason, a package called *memory_pkg.sv* was created to host all the possible parameters and type definitions. For the interfaces and inside the blocks, a lot of ad-hoc types were created to ease the visualization of the signals. This is convenient only if the editor allows for a fast search of the definition of the types, otherwisen it's a real hell.

Usually, the hardware is described behaviorally. There are more reasons for this choice:

- It's easier to describe the hardware this way.
- behavioral descriptions are usually more readable and self-describing than structural ones, and then, more maintainable.
- During the synthesis phase, the logic is optimized and the real final structure can be far different from the hypothesized one. Behavioral code abstracts this phase which is on the shoulders of the program that performs the synthesis.

Conventions When possible, the **lowRISC Verilog Coding Style Guide** was followed [23]. An exception is on the parts on the propagation of the **'X**: this part was not followed on purpose, because all the behavioral described logic has always default signals and no explicit *don't care* was used. This allows more readable testbench waveforms, thus can help the first part of the verification. When the entire system will be tested this way, all the behavioral blocks will be optimized with the *don't cares*.

For the parameters, the suffix *_LEN* is used to identify a length in bits, and *_SIZE* for the size in Bytes. When an *A* appears before *_LEN*, the parameter refers to the length in bit of the signal that addresses that quantity; for example, *L1_ICACHE_LINE_LEN* represents the i-Cache line length in bits and *L1_ICACHE_LINE_A_LEN* represents the length in bits of the part of the i-Cache address that point to a line.

All the signals written in the interface of the module should be tagged with their direction (*input* or *output*). This fact limits the inclusion of the handshake signals into an ad-hoc type for the requests and the answers. It's not possible to define a single type *lsq_dCache_req_t* that includes both the *Valid* and the *Ready*, because their directions are different. For this reason, each new type hosts the *Valid*, and the *Ready* is defined outside as a *logic* type.

For the more detailed diagrams (more details for the easy but important designs) two colors are used to roughly identify controls and data:

- Orange: DATA
- Light blue: CONTROL

5.1 L1 i-Cache

5.1.1 Requirements and constraints

Size and Associativity The i-Cache was designed to be theoretically *virtually indexed, physically tagged*, but with the index taken from the *page offset*. De facto, the addressing is completely physical, but the i-TLB access time can be overlapped with the cache critical path. This is a precise constraint: the minimum page size for virtual memory is 4 KiB, addressable using 12 bits. The addressing of the *page offset* is done using the least significant bits of the address, to favor spatial locality. These *page offset* bits are "physical" also in the virtual address because they are not translated. If these bits are used to access the cache while the TLB is translating the other part of the virtual address into a physical one, the TLB access time is masked by the access time of the cache. Therefore, the index part of the cache address must not exceed the 12th bit. If the contrary happens, a part of the index remains virtual. This is bad because, in this way, synonyms can be stored in different sets of the cache. Some techniques can eliminate the problem of the synonyms, like page coloring or other more complex HW controls, but they are not used for this project. Given that the minimum page size for RISC-V is **4 KiB**, the other parameters are constrained by the following equation:

$$\frac{Size}{Associativity} \leq 4Ki \tag{5.1}$$

Blocking cache The front end can fetch an entire instruction line but does not implement any form of prefetching. Thus, when a miss occurs, the front end should wait and cannot make other requests. This frees the cache from the hardware to keep track and handle outstanding misses.

Read Only memory The memory is a read-only memory for the front end. This simplifies the design a lot because no coherency problem exists.

5.1.2 Design Choices

Main parameters

Even though the values of these parameters are tunable, an initial choice was made following two principles:

- The size of the cache was set looking at today's architectures with a memory system similar to the one used by **LEN5**. As already said, the processor AMD Cortex A53 was taken as an example. Its i-Cache is a 16 to 64 KiB (configurable), 2-way, 64 bytes/block, WB cache.
- Tables in [6] show that the minimum miss rate for 16 KiB caches is achieved starting from 2-way associativity and 64 bytes per block. This combination also minimize the *AMAT*. The best choice seems to be a combination of a 16 KiB Cache, with 64 bytes/block and 2-way set associativity. To respect 5.1, 4-way associativity was chosen.

Therefore:

- Memory size: 16 KiB
- Organization: 4 Way Set Associative
- Line size: 64 bytes

These parameters will be used in the following as default values for the calculation of the other parameters.

- TAG length: 52 bit
- Index length: 6 bit
- Line offset length: 4 bit
- Word offset length: 2 bit

Link with the i-TLB

Each front end request is first routed to the i-TLB, which performs an address translation and gives back, respecting the handshake protocol, an answer, with a possible exception and the physical address.

5.1.3 Design

Interface

The interface of the i-Cache is presented in figure 5.4.

Even though *Valid* and *Ready* signals are shown in the picture, they are implied in each request/answer channel. Only in some special cases, the *Ready* signal does not exist because is implicitly set to **1**. For example, the front end is always ready for an i-Cache answer after having performed a request.

Front End The i-Cache can receive requests from the front end. A request is composed of a Virtual Address *vaddr*. This address is effectively *virtual* only if the Virtual Memory is on, otherwise it should be treated as physical.

After a variable number of clock cycles (if the request hits both in the cache and in the TLB, the access time is the minimum one) an answer is returned. It is composed of:

- The requested line
- Eventual exceptions (linked to the Virtual Memory)
- The virtual address linked to the request

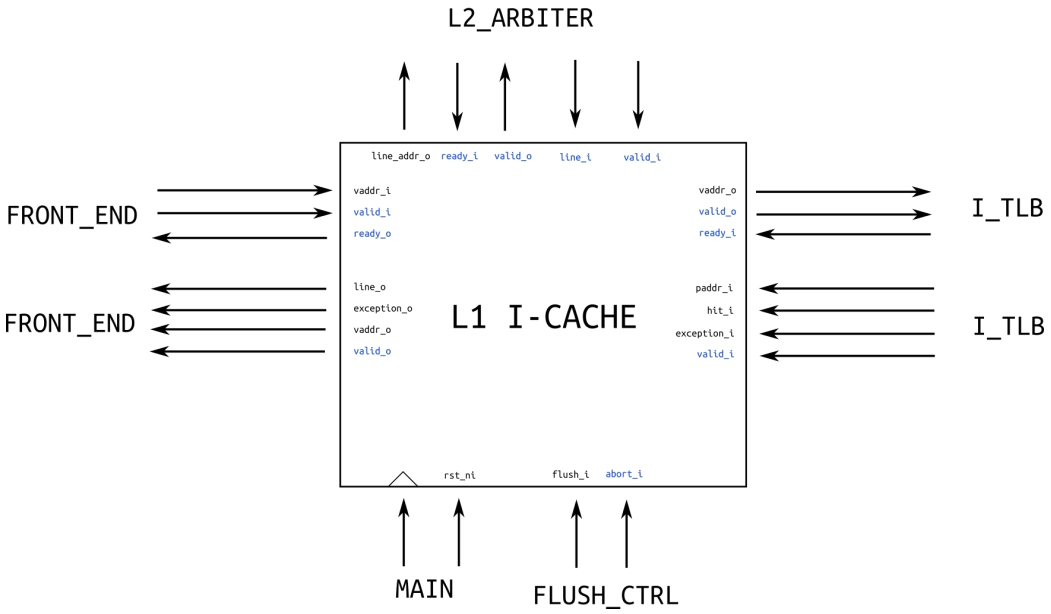

Figure 5.4. i-Cache interface.

TLB Each front end request is routed to the i-TLB. As soon as this block returns a valid answer (this can happen also in the same clock cycle!), the set, output by the cache, is checked with the physical tag from the TLB and a front end answer is possibly given. If the TLB raises an exception, a cache miss does not end up in a L2C[4] request.

L2 Cache When the TLB physical address is valid and a miss occurs in the i-Cache, a request is made to the L2C. This operation is not latency critical as the response to a hit, because the miss penalty is of several clock cycles, and if the process is controlled by Moore FSM the relative impact on the average time is hopefully negligible.

Implementation

The i-Cache DP[5] main schematic is depicted in figure 5.5.

The control is hidden to make the representation easier to be visualized.

The memory The *SSRAM BLOCKS ABSTRACTION* is self explicative. It is composed of N_Way memories that hold the lines (they are called *data* blocks) and N_Way memories that keep the tag associated with the corresponding line and the valid bit.

The registers The *vaddr register* keeps the virtual address to use it in case of *read replay* after a hypothetical miss because the memory cannot stall and a continuous request would be a senseless wasting of energy. The *tag register* is used to store the translated tag when an L2C line request is performed after a cache miss. This way, when the line is returned, no new TLB request is needed.

Flush control The flush counter addresses the memory when the flush is ongoing. This operation is requested at reset time and when a *FENCE.I* is executed 3.3.2. Each set of the memory should be cleared.

Replacement selection circuit When a new line is being brought into the cache and should be stored inside a set, the *Replacement selection circuit* is the module that indicates where to put it. Refer to 5.1.3 for further details.

Comparison and selection This block compares the incoming tag with the line tags and select the correct line, generating the cache *hit* signal.

The control For the control design, a raw ASM[6] chart was written to help the understanding of the different operations. It is reported in figure 5.6.

[4]Level 2 Cache

[5]Data Path

[6]Algorithmic State Machine

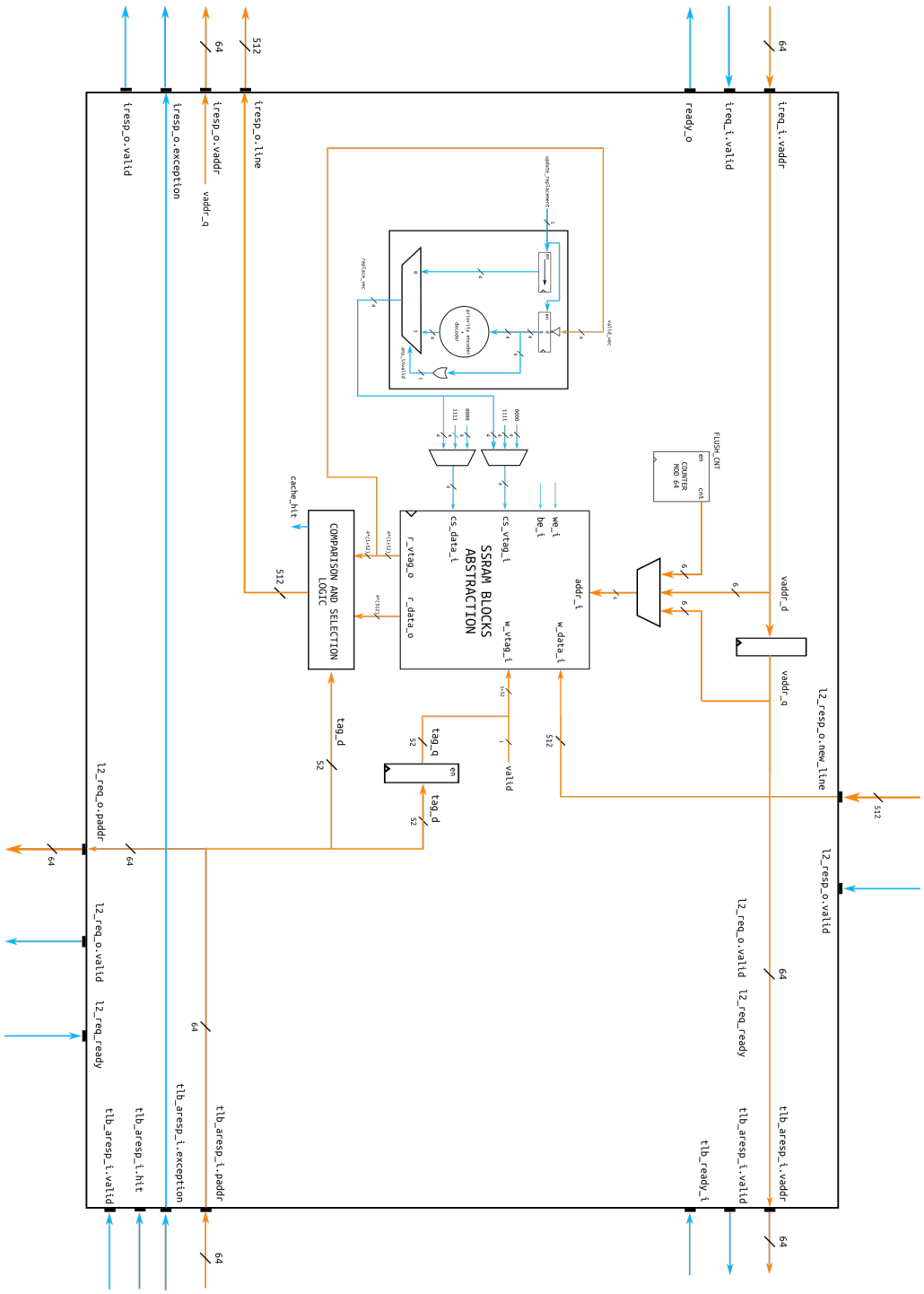

Figure 5.5. i-Cache interface.

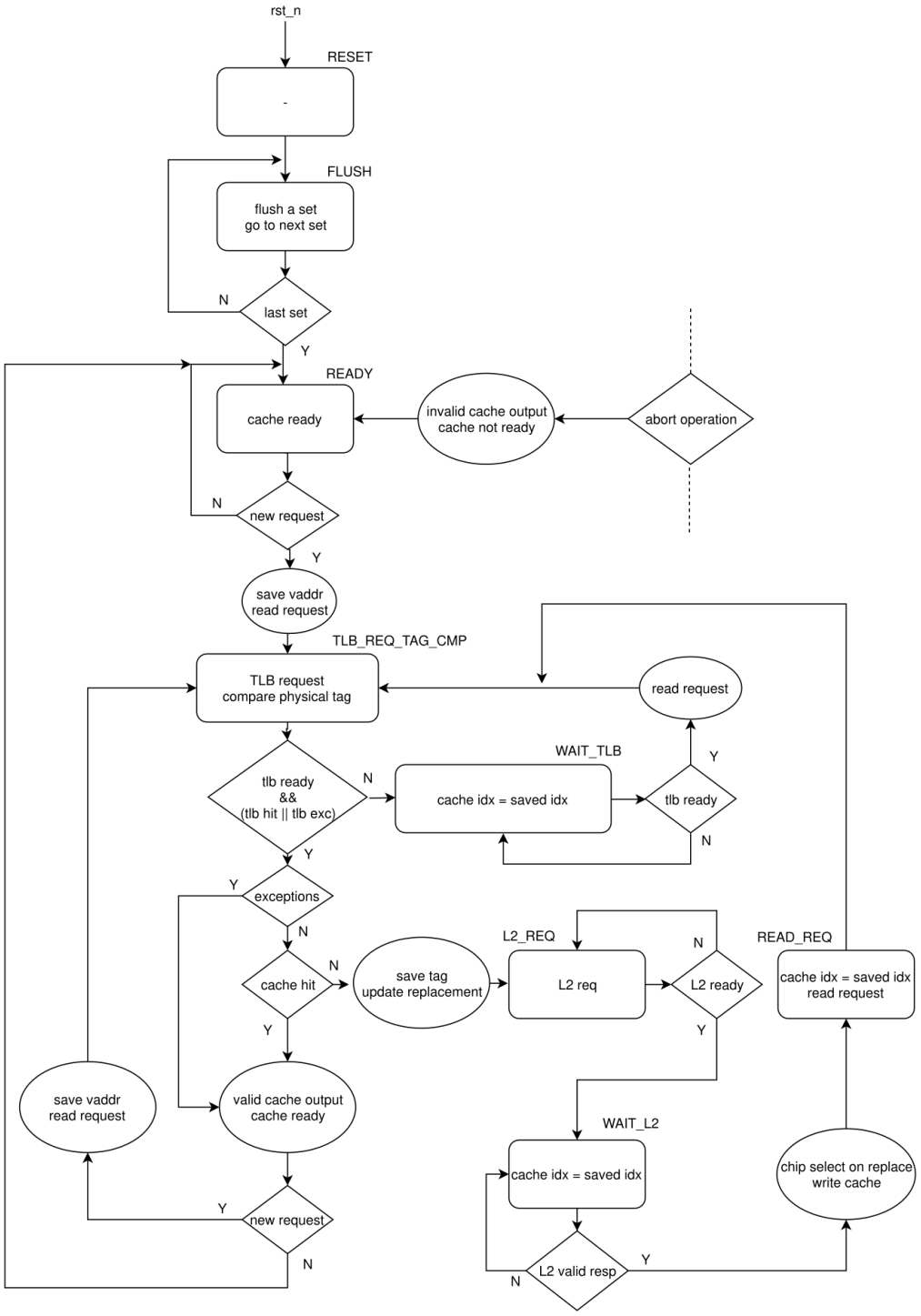

Figure 5.6. i-Cache CU raw ASM.

In the SystemVerilog description, the *Mealy FSM* was described with a *Moore FSM* (*icache_moore_cu.sv*) and a combinatorial module (*icache_ctrl_en.sv*). This way, it can be more clear where the control and status signals can impact the critical path.

Moore CU The main CU[7] sets the control lines, which depend only on the present state, and other signals to instruct the cache on what it should and can do in that cycle. For example: *l2c_req_valid_o* is the *Valid* signal for the L2C line request after a cache miss. It is asserted only in the *L2_REQ* state, therefore it is directly generated by the Moore CU. This is not the case of the physical memory control bits. The memory should be turned on and read-only when a valid handshake is done with the front end. But this fact depends on *icache_fend_req_rdy_o*: it is the *Ready* for the front end, and indicates that the i-Cache is ready to receive a new request. It strictly depends on other status signals, like the response of the TLB. If the previous request stalled because of a miss, the front end should wait too; however, this information is known only in the same cycle in which another request can be made. For this reason, the Moore CU encodes a conditional command for the memory, but it does not set the control bits to a precise value. They will be set by the *icache_ctrl_en.sv* block.

Combinatorial control This block is in charge of:

- enabling and setting the memory control bits;
- setting the handshake lines that depend on other signals. The handshake signals are combinatorial because they depend upon the response of the TLB and the cache TAG comparison. Initially, this choice was made to keep the latency as low as possible. Possibly and after a timing analysis, registers will be added to break eventual critical paths. Luckily, the handshake mechanism is latency independent;
- activating the replacement block. See 5.1.3;
- enabling the *vaddr* and the *tag* registers;

Replacement selection circuit The adopted *replacement policy* is a invalid pseudo-random one only if the set is full of valid lines. Otherwise, the invalid lines are filled first. Figure 5.1.3 shows the implementation of the algorithm. Since the cache cannot stall, the negated valid vector (invalid vector) is saved into a register when the TLB has hit but the cache has missed. This way, it will be available when the new line will be returned by L2C.

The invalid vector is first one-hotted (it's not important how this is done), to obtain a *one-hot vector* with the **1** in the position of the set line to be replaced. This information will be used only if at least one line of the set is invalid; otherwise, the pseudo-random policy will be used. To implement it, a one hot shift register with N_Way parallelism is used. This register has always a **1** that changes position every time a line is written into the cache. Since the indexed sets can be different, for each set the policy is nearly random.

[7]Control Unit

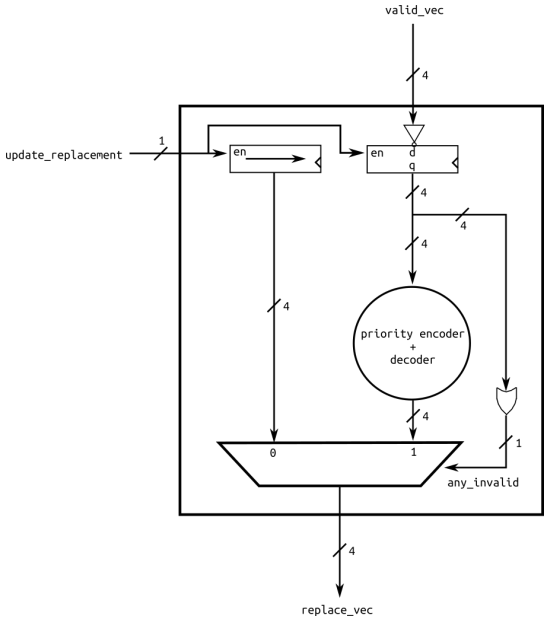

Figure 5.7. i-Cache interface.

5.2 L1 i-TLB

5.2.1 Requirements and constraints

Multiple page size support and Associativity

The system allows the following page sizes:

- 4 KiB
- 2 MiB
- 1 GiB

The choice on which size is supported impacts on the design of the i-TLB, as discussed in 2.3.4.

Usually, the i-TLB has low associativities, but a not fully associative organization makes the design complex if multiple page sizes are allowed.

Low latency, simple design

The i-TLB is on the path of the i-Cache address translation, and its hit time should be overlapped with the cache access time (an advantage of the *vipt* strategy with "physical" indexing). For this reason, it will not be too big. A study on the timing analysis will be useful to understand "where" the i-TLB should be put (see the "Synthesis" section).

Selective flush

The RISC-V TLB flush, which follows a *fence.vma* instruction can be total or selective. In the first case, all the entries are marked as invalid. In the second case, the flush can be selective on the ASID or the *Page*. This information is sent to the TLB using two other signals.

ASID, Global bit, User bit

Each entry of the TLB should keep an ASID, a *Global bit* and an *User bit*. The ASID is needed to avoid the homonyms problem, the global bit to define a page as Global (no ASID comparison on request, no flush for selective *ASID*) and the *User bit* should be checked for possible exceptions (see 5.0.1).

5.2.2 Design Choices

Main parameters

Since no precise timing constraints were available, it was decided to support all the page sizes. This decision brought to an "easy" fully-associative implementation, where the indexing problem does not exist. This choice has exacerbated the constraint on the size, which is initially of **10 entries**, like the TLB of the **ARM Cortex A53**.

5.2.3 Design

Interface

In figure 5.8 is present the interface of the i-TLB.

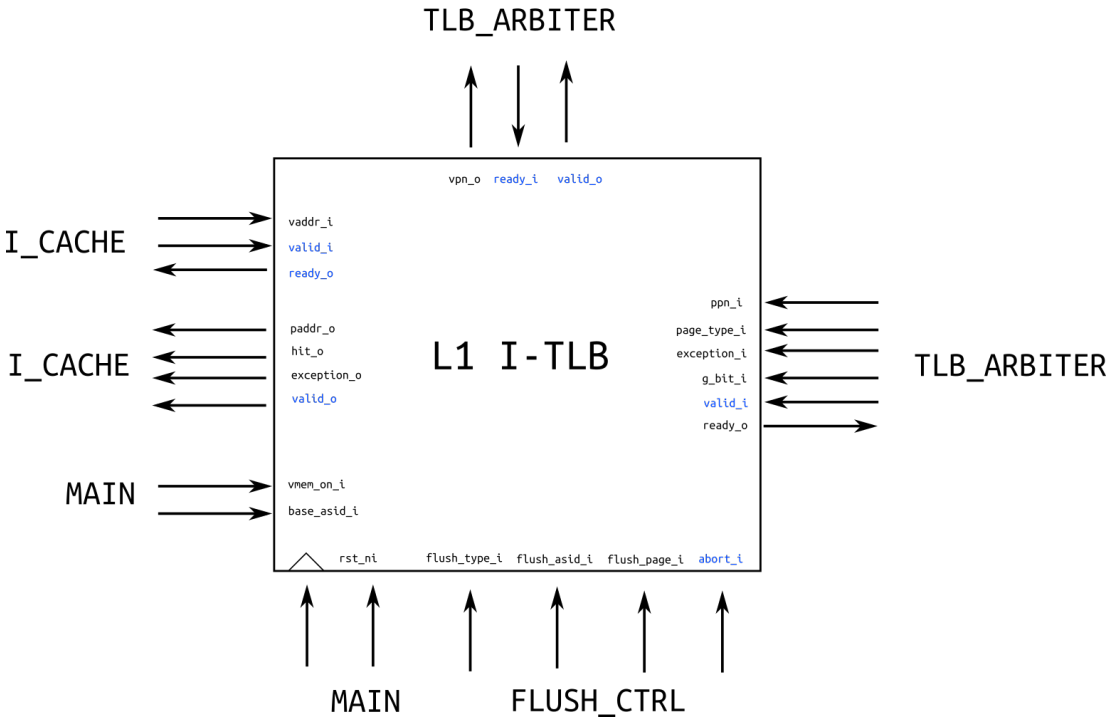

Figure 5.8. i-TLB interface.

Main The TLB receives information about the state of the Virtual Memory and the flush controls. Another important input signal is the current ASID, which comes from the *satp* register.

i-Cache The *Virtual Address* is translated into a *Physical* one, and the transactions happen on valid handshake events.

L2 TLB When a TLB miss occurs, the L2 TLB is queried with the VPN. This signal goes to the TLB Arbiter 5.5, which routes the request to the L2 TLB. After a variable

number of clock cycles, the corresponding full-size PPN is returned. Its parallelism is the one corresponding to a PPN of the minimum page size. The *exception* field of the answer contains the encoded exception encountered during the page request.

Implementation

The main diagram of the i-TLB is reported in figure 5.9. As it can be seen, the core part is the *i-TLB Abstraction* block. This is the set of registers that host all the information a TLB could store. Figures 5.10 and 5.11 show its composition.

The TLB is fully associative, therefore all the entries are available. The *compare and selection* select the entry that allowed the incoming request to hit, and finally the i-Cache answer is composed in the *paddr composition* block and with the aid of the output muxes.

The virtual memory The Virtual Memory can be activated or not when a request is made to the i-TLB. If it is off, all the incoming requests should be short-circuited to valid answers with no exceptions. Otherwise, the address is translated by the i-TLB.

The control As for the i-Cache, the control was firstly designed with a *Mealy ASM* chart (5.12). Then, the *FSM* was split in a *Moore CU* and combinatorial control. The CU controls the relationship with the L2 Cache. The other controls are combinatorial. If a flush request is given, it should be accompanied by an abort signal. This brings the CU to the Ready state.

The i-TLB waits, after the reset, in a ready state. When the i-Cache makes a valid request, the TLB performs a comparison and checks for a possible hit. If valid access is done, the replacement circuit is updated (5.2.3) and a valid answer is given to the i-Cache. Whenever a valid request misses, the i-TLB stalls and requests the correct data to the L2 TLB. After some cycles, the valid data is returned. All the information is saved into the correct entry of the i-TLB (pointed by the replacement vector from the replacement circuit) if no exception occurs. Also, the exception is saved into a particular register. Then, the i-Cache request is replayed, and the correct physical address with the corresponding exception is returned. In the end, the exception register is cleared.

The comparison The VPN portion of the incoming virtual address is compared with the "tag" of each entry. The size of this comparison is given by the page size of each entry. For example: if the entry hosts a *GibiPage*, only the first **9 bits** of the VPN and of the *TAG* are compared. This comparison is valid only if the *Valid* bit is asserted and if either the page is global or if the *base_asid* corresponds to the *ASID* of that entry.

paddr composition When the Virtual Memory is activated and the request hits, the physical address is composed knowing the page size of the entry. For example, the VPN of a *GibiPage* is only **9 bits** long: the rest of the *Virtual Address* is a page offset. Therefore, the physical address is composed taking the first **26 bits** of the PPN and the last **30 bits** of the vaddr. To refresh these information, see figures 3.2 and 3.3.

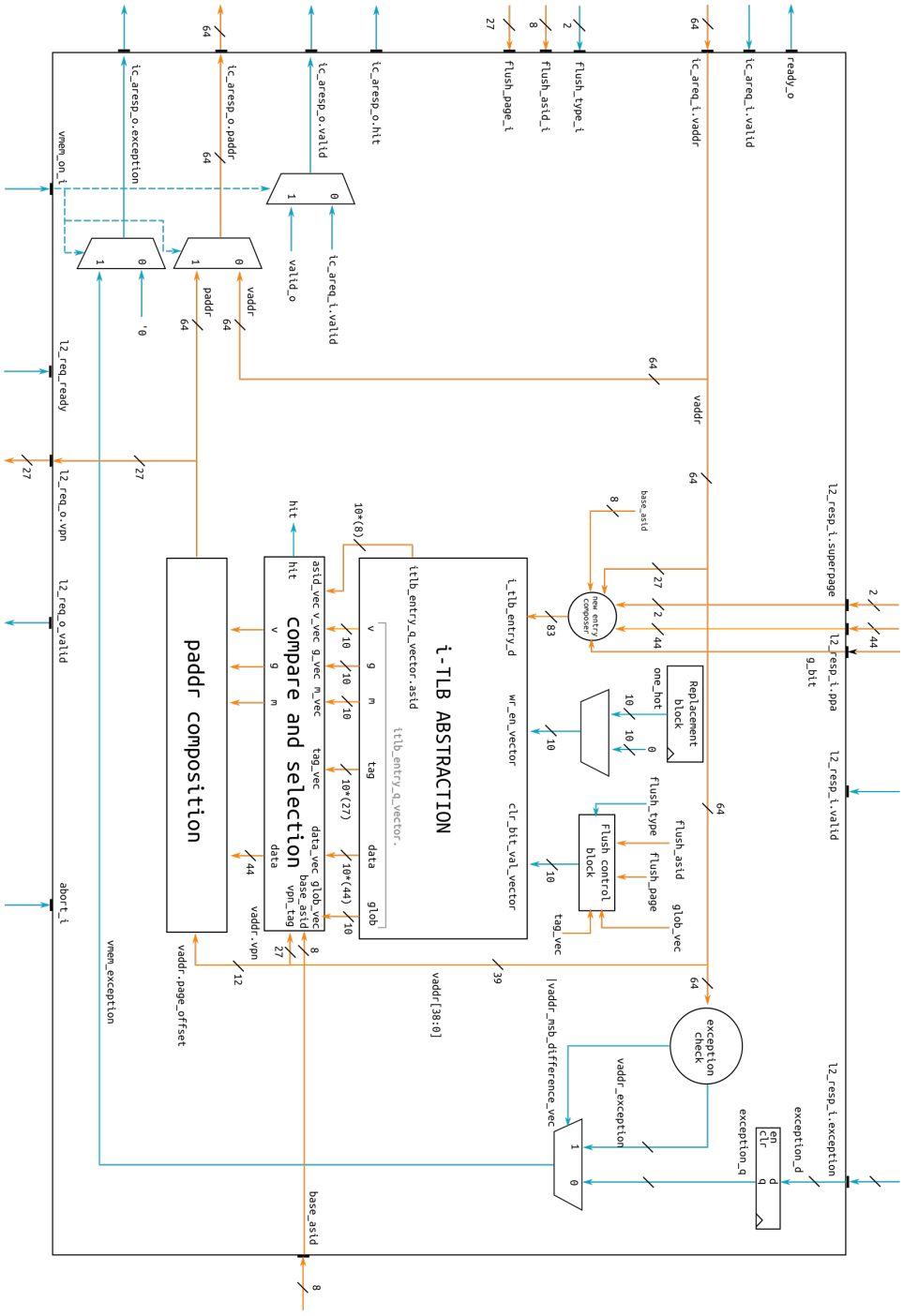

Figure 5.9. i-TLB main diagram.

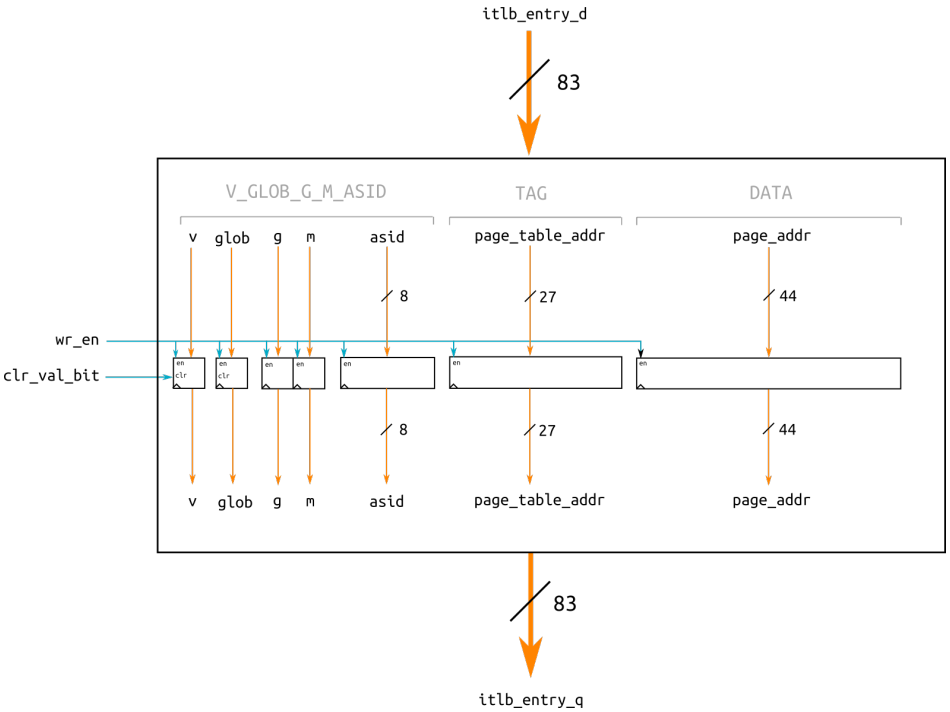

Figure 5.10. i-TLB register entry abstraction.

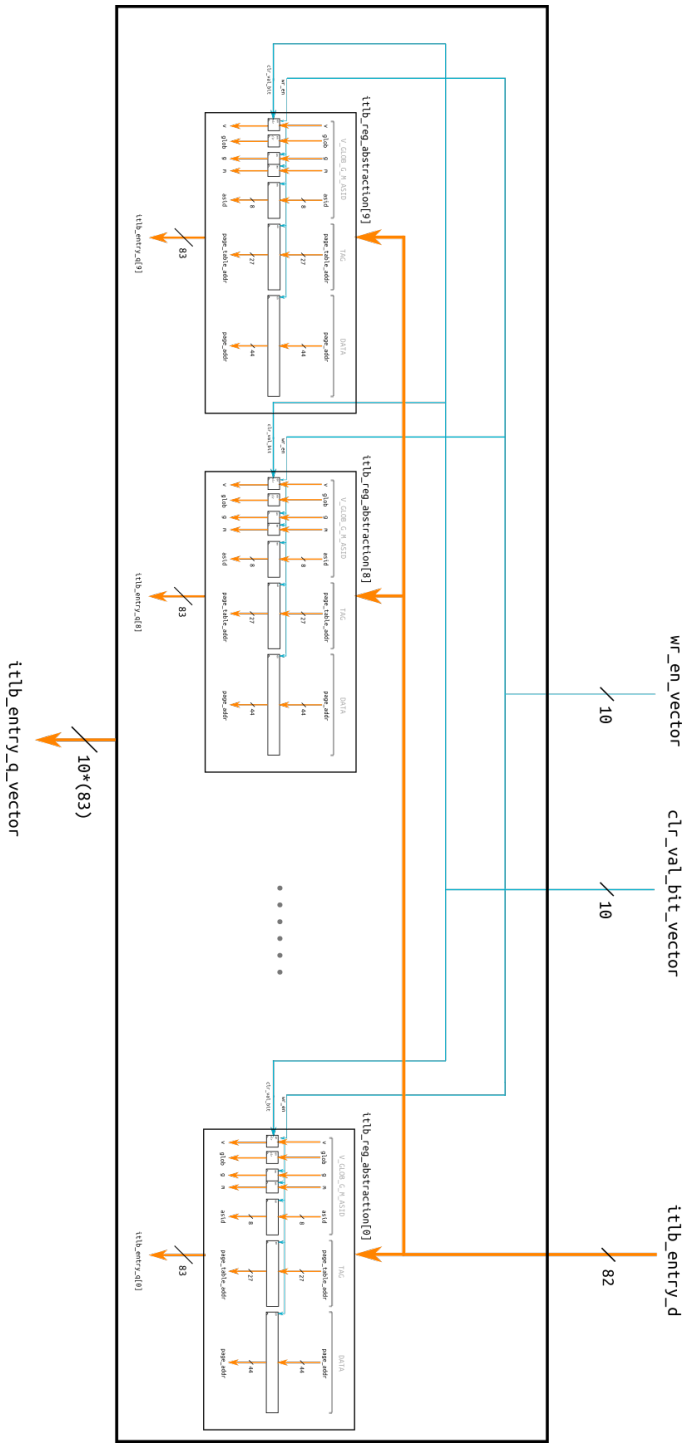

Figure 5.11. i-TLB register entries abstraction.

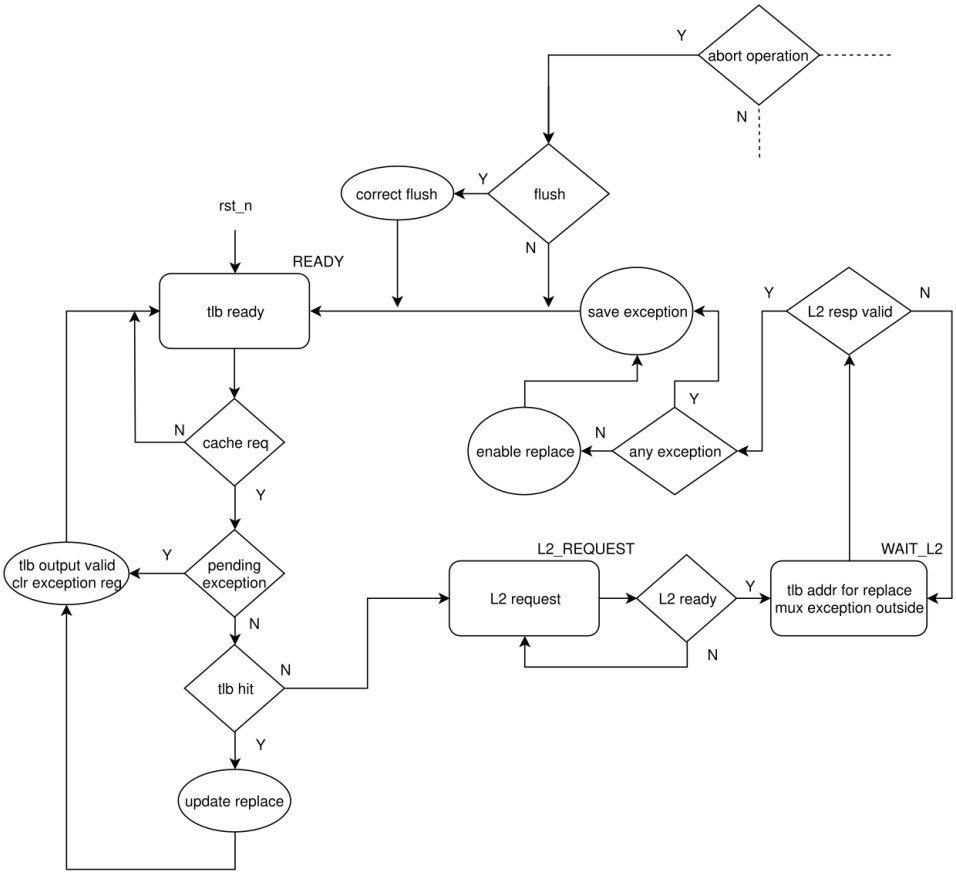

Figure 5.12. i-TLB raw ASM.

The exceptions The i-Cache request can raise an exception in more than one way when the Virtual Memory is on:

- The *Virtual Address* is not compliant with the Sv39 format, i.e. the bits from 63 to 39 are not equal to the bit 38.
- The combination of the **U**ser bit and the **SUM** bit should generate an exception.
- The PTW realizes that the page has the access bit cleared 3.3.3.
- The requested superpage is misaligned.
- The requested page is not executable.
- The requested page is on the secondary storage.

The first two exceptions are the only to be detected inside the i-TLB.

Replacement circuit Since the TLB can be selectively flushed, there is the possibility to have holes. For this reason, policies like **FIFO** cannot be implemented using only a one hot shift register.

The adopted policy is NRU every time all the entries are valid. Otherwise, the first invalid entry is primarily filled. The block takes in input all the signals used for the updating of its output: the replacement vector. It is the one-hot vector that identifies the next entry to be evicted.

NRU implementation A register with the same number of entries of the TLB, called *safe register*, is initially reset to zero. Every time a valid access to an entry of the TLB is performed, the corresponding bit in the *safe register* is asserted. Each flip flop of the safe register is singularly controlled by an enable. Normally, this enables is taken from the hit vector itself. Until an entry has its corresponding bit in the *safe register* asserted, it is safe and cannot be evicted when a replacement occurs. The replacement vector is formed one-hotting the negated output of the *safe register*. This way, the replacement vector always points to a not-safe entry.

When the last not-safe entry is becoming safe, all the other entries of the *safe register* are reset to zero.

It's impossible the *safe register* remains reset to zero when it should be used to define the replacement vector. This is because if there are invalid entries, they are used first. Every time an entry becomes valid, it is accessed one cycle later. If it is accessed, the replacement vector is updated; and once it is updated one time, it's impossible it returns in the reset state (unless a reset is fired from the system).

The updating of the replacement vector is automatic and happens when a valid access is done, and not when a replacement is performed. When a replacement is performed, a valid access will be done the cycle later.

5.2.4 Further improvements

The awareness of why some decisions should be made in a certain way has arrived during the whole thesis work. In the beginning, it was not clear how to decide where to put the block and why. This block was designed together with the i-Cache, and initially, they were highly coupled with two main control units. However, this kind of design is not flexible.

A better-designed i-TLB - i-Cache system would be more flexible and allow an easy pipelining. The next version of the i-TLB will be accessed in parallel with the cache indexing, and not after it.

5.3 L1 d-Cache

5.3.1 Requirements and constraints

Number of sets The back-end is firstly designed to make separate requests to the d-TLB and the d-Cache, thus the address translation and the memory request are not performed in parallel and happen during different clock cycles. This means that the memory request is performed using the full physical address. This choice frees the parameters of the d-Cache, because the number of its sets, given the number of bytes in a line, is no more limited by the minimum virtual page size.

Non Blocking Since the back-end of the processor is Out of Order, the d-Cache should be non-blocking (or *lockfree*) to avoid stalls whenever a request misses. This choice is beneficial for the miss penalty because when a miss occurs the cache is free to serve other requests. A lock-free cache occupies more area and consumes more energy when it is working, but it can help in reducing the execution time and eventually the total energy consumption.

Coherence The d-Cache is the only part of the memory hierarchy which is directly written by the processor. When a write is performed, the coherence between the various memories is put at risk. The d-Cache needs a method to enforce the coherence, following the RISC-V directives.

High throughput An Out of Order processor is capable of having a lot of requests ready to be processed. To decrement the average CPI is important to be capable of maintaining a high throughput, also in the case of multiple hits. Since the critical path is not known "a-priori", a flexible pipelined architecture can be a good choice.

5.3.2 Design Choices

Main parameters

The cache is fully parameterized to support benchmarks, but at design time a choice for the default values of the parameters was done. Looking at the tables showing the AMAT in the function of the cache size, the organization and the line size, data show accordance

with the choices adopted in the design of commercial architectures like the ARM Cortex A53. Thus, the values are similar.

- Memory size: 32 KiB
- Organization: 2 Way Set Associative
- Line size: 64 bytes

These parameters will be used in the following as default values for the calculation of the other parameters.

- TAG length: 50 bit
- Index length: 8 bit
- Line offset length: 3 bit
- Word offset length: 3 bit

The physical memory block used is an SSRAM memory with one clock latency "address to data", controlled by three main signals:

- CS[8]
- WE[9]
- BE

Into the cache are instantiated N_Way couples of these blocks. Each couple is composed of a TVD[10] memory for the tags, the valid bits and the dirty bits, and the other DATA memory for the lines. Each memory block has N_Set rows. The TVD memory has TAG length columns for the tag and 2 columns for the valid and the dirty bit.

Nonblocking behavior

The LSQ is a buffer with more than one entry and can make requests to the cache during each cycle. The cache should not stall when a miss occurs. Therefore, when a request misses, the information about the miss is saved into the MSHR. Until the requested line is brought back to the d-Cache, the MSHR keeps the information of the primary misses and ignores secondary misses. When the line is returned, a wake-up signal is given to the LSQ to wake up all the instructions that were sleeping, to let them replay their requests.

[8] Chip Select

[9] Write Enable

[10] Tag Valid Dirty

The Miss Holding and Status Register

The MSHR hosts all the addresses of the requests that have missed. A new entry is allocated only if a primary miss occurs, i.e. a request misses and the MSHR does not contain its information. At the same time, the MSHR tries to make requests to the upper level. When a request is performed, the related entry is put in a waiting state, to indicate that no other requests should be performed for that line address. When the line is returned from the upper level, the corresponding MSHR entry is cleared.

Write coherence and miss policy

To reduce the memory traffic and to face new challenges, the d-Cache implements a Write Back policy to handle writes. To speed up the back writing to the upper level of the hierarchy without stalling the LSQ interface, a WBB is provided.

When a write miss occurs, the missed line is requested to the upper level and brought into the cache following a "Write Allocate" strategy. This makes sense especially because the cache is write-back: the line that is brought to the first level can be re-written other times without generating memory traffic between the levels.

The Write Back victim Buffer

The WBB hosts all the lines evicted from the cache and tries to write them to the upper level. When a request is successfully performed, the entry is put in a waiting state. If the write hits in the upper level, an acknowledgment is returned to the WBB buffer and the corresponding entry is cleared. If the write misses in the upper level, a wake up is given to the WBB when the line becomes available in the upper level, and the entry is woken up.

The WBB is a convenience for the lock-free behavior, but it introduces some critical issues. First of all, it generates coherence problems if the RAW constraints aren't enforced. If a line with address LA is kept into the buffer and the LSQ requests that line, that line should not be requested to the upper level, because this would create a RAW hazard. That line should be forwarded from the WBB instead, possibly to the cache.

The forwarding from the WBB to the cache can replace a dirty line, thus a line switch can occur. The dirty line is written into the same entry of the WBB which is being freed during the same cycle.

The WBB to d-Cache forwarding introduces another issue: if a *write request* is successfully performed by the WBB to the L2C, there is a time interval between this request and the answer. If the write hits, an acknowledge will be returned to the L2C and the corresponding entry in the WBB will be cleared. The problem arises in the following case: if a line is written into the buffer and the buffer performs an L2C request, the line is put in a waiting state. If this line is then forwarded into the cache, modified, and then replaced, written to the buffer and another request is done to L2C, this line will be in the write buffer in the same condition as the previous one. Therefore, when the L2C answer will arrive, a loss of data can occur. If the answer is a "wake-up" (i.e., "please, repeat the write request, because previously your request missed, but now the data is ready"), no problem: the line will perform the request another time. If the answer is a "store hit" (i.e. "Your write request hit, bye!"), this store hit was relative to the previous line write back. If the actual line is cleared from the write buffer, data is lost. This issue is very rare, but it can

be completely avoided. The problem manifests only if the first request hits the L2C and the second request misses it. Because if the second request hits, no data is lost.

This can happen if the L2C accepts requests which can evict L2C cache lines between the first buffer request and the second one. There are two possible solutions to this problem:

- Forbid the WBB to d-Cache forwarding if an entry of the WBB is waiting.
- Assign to the entries a tag at the moment in which the WBB request happens. This way the entry is marked with a tag and marked as "waiting". When the acknowledge takes place, the tags are compared instead of the line addresses.

The WBB of this design is implemented with the tags.

Data miss policy

The cache follows a "Write Allocate" policy, therefore when a general request misses, the correct line is requested to the upper level. This line will be written into the correct set into the d-Cache. If the set is full, an old-line should be evicted to make space for the new one. If the old line is dirty, it should be written to the WBB not to lose the changes previously made on that line.

There are two possible ways to deal with this eviction:

- Early eviction
- Delayed eviction

In the first case, as soon as a miss is detected, a line of the set is evicted to free a cache entry for the new line. In the second case, the eviction is done only when the correct line is returned by the higher levels.

To decide which policy to use, a personal analysis of the possible pros and cons was done.

Early eviction:

- - If the WBB can forward a line to the d-Cache when a WBB hit occurs, there's the need for saving the set replacement index along with the buffer entry because the replacement index in the replace block should be updated when the line is evicted. This is because if an entry of the set is freed but the replacement index is kept the same, if a WBB hit occurs, the WBB line is written back to the same place from where it was evicted. Therefore, the line coming from L2 will find no place to be stored. If this is the situation, another check one the dirty and valid bit is mandatory even when the upper level returns the line.
- + When L2 returns, there is the need for a cycle penalty only if the victim buffer has only one comparison port for the line address. If D1 is processing a request and L2 returns, D1 is using the comparison port of the buffer, but the L2 answer needs it too to know the replace index (structural hazard). In the case of LEN5: 1 cycle penalty (the request goes from d0 to d1).
- - Hypothetically more traffic between the buffer and the cache. If the line is evicted in advance, there is the possibility to use it other times before L2 is ready to accept the write-back request. An early eviction puts a hole into the cache.

- - If a speculative load misses, it can evict lines from the cache, occupy an entry in the WBB and fire requests to L2. What is returned from L2 should be written back to L1 even if the speculation was wrong.
- - If the number of entries of the store buffer is greater than the number of elements per set in the cache, the cache should stall also if all the elements of a set have been evicted. This is because, in case of another hypothetical miss, no element in the set can be evicted.
- + When d1 is stalled and L2 returns an answer related to a load, it's sufficient to access the WBB to know the replacement index.
- - The main limit to the number of outstanding misses supported by the processor is the number of entries in the WBB, because each miss can correspond to a new WBB allocated entry.

Delayed eviction:

- + No need for saving the replacement index into the WBB for each entry.
- - Even if the L2C answer of the actual request will replace a not-dirty line, an entry of the WBB is "occupied" anyway at request time. This is because that clean line could be dirtied between the LSQ request and the L2C answer.
- - Danger of deadlock if more requests than WBB free entries are "speculatively" performed. For example: if the buffer has 1 free entry and two misses are handled by the MSHR, the two requested lines may return before the buffer is freed. If both returns, the second will stall, together with the L2C. But if the L2C is stalled, it can't receive other WBB requests. The system is in deadlock. The solution is to allow MSHR requests to L2C only if the already pending ones are less than the free WBB entries. This way, the buffer behavior of the MSHR is decoupled from the requests that are fired to L2C. The MSHR can store miss requests and can decide when to fire them.
- - When L2C returns, there is always the need for a cycle penalty for the reading of the line to be replaced, because it can be dirty.
- + The line that will be replaced is kept into the cache for more time: this means less traffic between the cache and WBB.
- + If MSHR is flushed for a ROB commit flush (exception, branch misprediction...), it is easy to deal with older wrong requests even if the upper hierarchy was not flushed: when L2C returns, there is no hit in the MSHR, thus no update is performed. If L2C returns something which is just being added in the MSHR anyway, it is handled correctly (only the d-Cache can write, no other modifications have happened).
- + What is inside the WBB must be always written into the L2C. If a speculative load misses, it is put in the MSHR. While the MSHR is waiting for L2C response, no line is evicted from the Cache. If the speculation was wrong, the MSHR is cleared and the WBB is not used. Therefore, there is more time for the processor to correct its speculations without wasting energy.

- - When d1 is stalled and L2C returns an answer related to a load, to complete the replace there is the need for using the d1 stage to know if the line to be replaced is dirty. Therefore, other registers are needed for the previous instruction replay.
- + If a convenient L2C read/write request policy is implemented to avoid deadlocks as exposed in the third point, the main limit for the supported outstanding misses is the number of entries in the MSHR. This is good because the MSHR is smaller and less consuming than the WBB.

After the evaluation of these pros and cons, the delayed eviction was chosen. The only constraint is to avoid possible deadlocks (L2 on flight requests limited by the number of buffer entries. When the limit is reached, a new request is possible only after an L2 write back occurs).

Line replacement policy

When a request misses in the d-Cache, it is forwarded to the upper level of the memory using the MSHR. Then the returned line will be written into the cache in the correct set. The position in the set is determined by the Replacement Vector. This vector is generated by the replacement block (or replacement unit) and can be updated using different policies. The one adopted for the d-Cache is the FIFO one: each set has a corresponding one hot shift register, which is updated every time the cache is updated. This way, the older element of the set will be the first to be evicted.

Updating L2

When the program wants to modify an instruction stored into the memory it can only do it using store operations, which write only the d-Cache. Since the cache is write-back, the dirty lines are kept inside the cache until a new line evicts them. How these “modified” instructions are made visible to the i-Cache? RISC-V provides a special *fence.i* instruction to let all the modifications performed in the d-Cache be made visible to the instruction part of the memory hierarchy.

The *fence.i* instruction force an effective ordering between all the memory references made before it and the ones made after it.

A special block is in charge of synchronizing the L2C with the d-Cache. It receives an updating request and forces the d-Cache to write all the dirty lines to the upper level.

Internal vs. External replay

When an instruction replay is needed, e.g. after a stall or after a WBB <-> d-Cache line switching, it is possible to replay it internally or externally.

If the instruction is saved internally in d1 into some replay registers, the replay is faster, especially for a store that hits the WB Victim Buffer. Also, with only one wake-up port, the wake-up can be done from d0 (faster).

With the sleep-wakeup implementation of the LSQ, it is possible also to only wake-up the instruction immediately when the stall begins, or when there is a WB Victim Buffer hit. This allows, together with the priority mechanism, to resolve the "stall" and safely

replay the instruction without additional control or structures inside d1. This implies that when L2 returns should also wait to pass through d1 to wake-up the LSQ.

The mechanism is now implemented with “replay registers”, but if they are seldom used it will be possible to remove them and use the “wake-up”.

5.3.3 Design

Interface

In figure 5.13 is reported the interface of the d-Cache. The signals are not completely shown not to lose the focus on the actors in action.

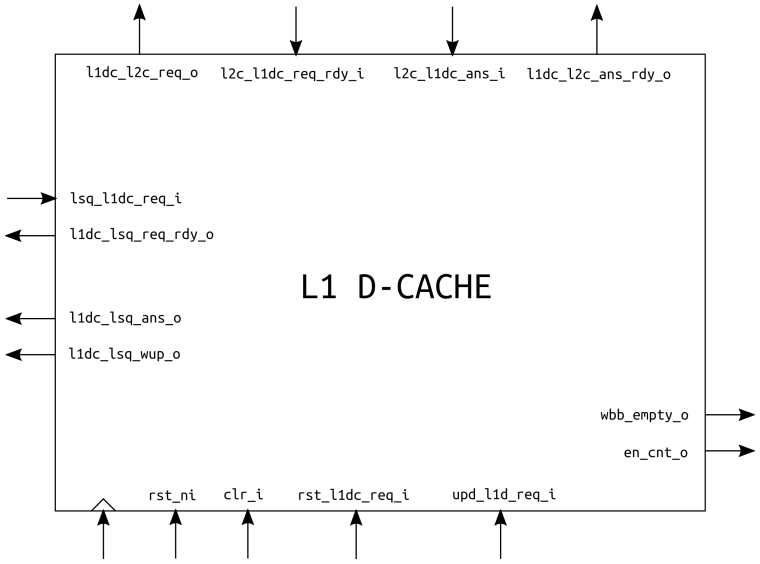

Figure 5.13. d-Cache interface.

A full description of the signal is present in the comments near the SystemVerilog code.

Implementation

The d-Cache is complex and should be flexible. It can receive requests from:

- LSQ: it makes load/store requests
- L2C: returns the requested lines and acknowledge successful writes back and wake up
- *Update L2 Block*: it forces the d-Cache to write all the dirty lines to the upper level
- *Reset Block*: it resets the cache, marking each line of each set as invalid
- the d-Cache itself: almost only a load hit is immediately resolved without performing back requests. An effective store writing or a WBB to cache forwarding, or a replacement requires that the cache is capable of making conditional requests to itself.

To better understand the cache structure and needs, a high-level diagram is reported in figure 5.14.

The simpler structure of the d-Cache is a two-level block. The first level is called **d0**, the second one **d1**.

d0

This is the part of the cache in which the requests are collected. A scheduler and an arbiter work together to schedule the next block to serve, give it a ready if there's the need, and inform the instruction encoder. This block receives the various requests and the information on the scheduled block (i.e. the *winner*). This way, it can encode an instruction for the d0 block and another instruction to inject in the registers between d0 and d1. The next cycle that instruction will percolate into d1. This way, the cache can work "in pipeline".

This request-decoding and instruction-encoding are both beneficial for the development of the block and the flexibility of the structure. If other pipeline levels should be added to break an eventual critical path, the control is not difficult to be adapted.

Initially, the design was faced using a complete division between CU and DP. Using a single CU for all the levels is impractical or even impossible. The amount of possible cases brings to a state explosion with too many states. This design would have been hard to implement and even harder to maintain.

Instead, each level has its combinatorial control. Some levels can also have a Moore CU for the operations that can be handled easily in this way, like the stalls.

When an instruction is encoded, the cache controller and the data selector are informed. The first block gives the control to the cache to activate only the correct blocks and the correct bytes with the right operation (read/write). The second block routes the data associated with the request to the cache and the d0-d1 register.

The stage d0 serves a request only if it is ready to serve it. When the cache is stalled, for example, it does not see any block but the L2C, to avoid deadlocks.

Scheduler The d0 stage of the cache system can be used by five different actors, listed in descending order respect to their priority:

- **Reset block**: it resets the cache. It has the highest priority.
- **d1 stage of the cache**: it back-requests a write during the second part of a store, it is used in the case of WBB forwarding to the cache and for immediate write or clear requests. If no reset is occurring, it has the highest priority.
- **L2C**: it returns the requested lines, acknowledges the WBB when a write-back succeeds and wakes up the WBB when an upper-level miss is resolved. Generic L2C returns are fundamental because they can un-stall the buffer when it is stalled.
- **Update L2 block**: used for L2C synchronization. When this process is ongoing, d1 requests behave normal (they clean the dirty lines when the lines are written into the WBB).

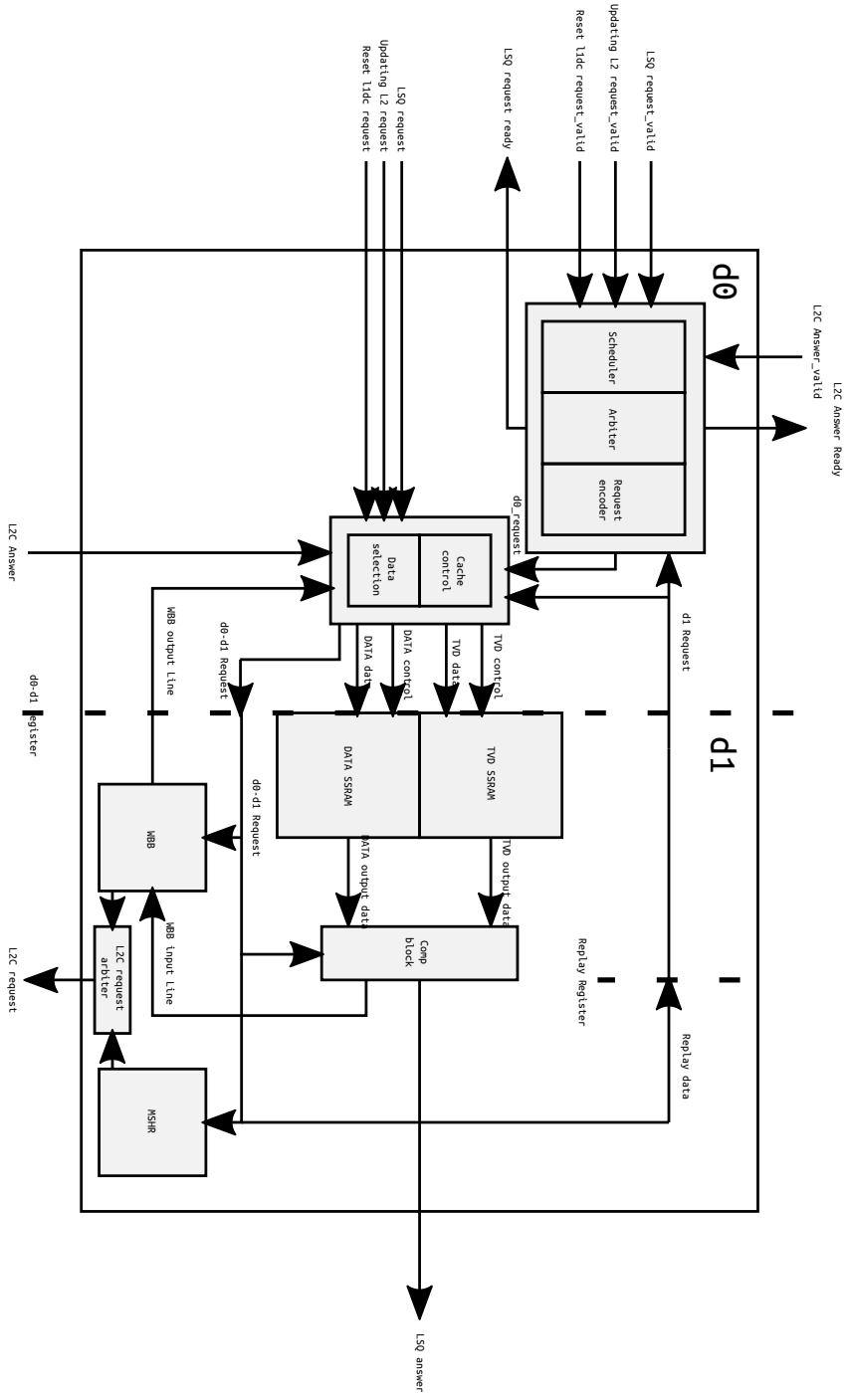

Figure 5.14. d-Cache high level schematic.

- **LSQ**: it makes load/store requests. This block can perform requests only if no other blocks are trying to use the d-Cache.

The scheduler is responsible to understand the block to serve next. In case of ties, the next block is the one with the higher priority.

If synchronization between L2C and the d-Cache is ongoing and L2C is trying to answer to acknowledge the WBB, a deadlock can occur if the Update L2 block has a higher than the L2C.

The scheduler understands and encodes the actor in action:

- Nobody (no requests)
- LSQ
- UpdateL2
- L2C
- D1
- ResetBlock

Arbiter The arbiter is responsible for giving the acknowledge signal (a ready) to the right actor. The d1 stage does not need an acknowledge because it has the highest priority, nor the update L2 block because it is controlled directly by d1. The Arbiter knows if the d1 stage is ready or stalled, enable d0 if it is active because a request can be fulfilled. This signal is also used to update the output registers and enable the memory.

d0 Request Encoder The request encoder understands and encodes the request type to control the d0 stage in the same clock and the d1 stage in the following (if d1 is ready to sample new requests).

The block creates 2 output signals:

- d0_req_type_o, i.e. the request type to instruct d0 stage
- d1_next_req_type_o, i.e. the request type to instruct d1 stage the "next" cycle

The first signal is decoded again in the d0 stage by the "Data Sel Block" and by the "Cache Controller" to understand which data and commands inject in the memory and the output d0-d1 register.

The second signal is directly injected into the output registers and will be decoded by the d1 stage to perform different actions.

Different d0 - d1 behavior are characterized by different couples of these two signals.

d0 possible requests:

- **Idle** Nothing to be done, no fulfillable requests.
- **ReadLsq** Cache read (source: LSQ).

- **ReadD1** Cache read (source: d1).
- **WriteCleanLineD1** Cache clean-line write (source: d1).
- **WriteDirtyLineD1** Cache dirty-line write (source: d1).
- **WriteStoreD1** Cache DW, W, HW, B write (source: d1).
- **ReadL2** Cache read (source: L2).
- **FwdL2StAddr** Inject the store addr into the d0 -> d1 regs (source: L2).
- **ReadUpdate** Cache read (source: update-L2 block).
- **CleanDirty** Clean the dirty bit addressed by d1.
- **ResetLines** Invalid the set (source: rst_block).

d1 possible requests:

- **Idle** Nothing to be done.
- **Load** Check for hits and possibily answer to the LSQ (Cache, WBB, MSHR).
- **Store** Check for hits (Cache, WBB, MSHR) and possibily make a back-write request to d0.
- **L2WBBWakeUp** Wake up the WBB entry.
- **L2StHit** Acknowledge the WBB for a successful L2C line-write.
- **ReplaceReq** Check dirty bit of the line to be replaced, back-write the line to d0.
- **UpdateL2** Give the one-hotted dirty vector to d0, check the WBB, back-clean request to d0.

Cache Controller The cache controller enables only the correct bytes that should be read (usually all, because the wanted output is the whole line) or written (it depends on the size of the store if the write is linked to a store). It also activates the cache and controls the reading and the writing processes.

d0-d1 register

The d0-d1 register is enabled only if there is something useful to be propagated. If d1 is ready but nothing useful can be injected in it, the register is simply cleared.

d1

The d1 level receives and executes an instruction from the d0-d1 register. In d1 the output of the memories is checked to understand if the response is a hit or a miss, and the same operation is done on the MSHR and the WBB.

MSHR and WBB

To better follow the working principles of these two blocks, their interfaces and diagrams are reported in figures 5.15, 5.16, 5.17, 5.18.

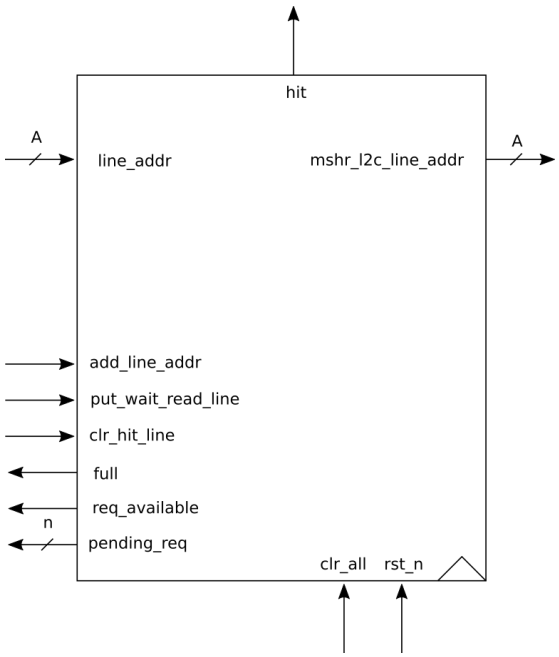

Figure 5.15. MSHR interface.

When the buffer is non-empty and a miss occurs, the entries of the buffer have to be associatively checked to avoid RAW memory hazards, and a forwarding mechanism is mandatory if there is no other way to read and update directly the WBB.

A line L can be:

- in the cache and not in the write buffer
- in the write buffer and not in the cache

When a Load or a Store instruction is made and the Cache is read, the Cache, the WBB and the MSHR are searched in parallel for a hit, because the requested line can be stored in one of the first two places or not. In this case, it is in an upper level of the hierarchy, but if the MSHR already hosts its related address, the *miss* is pending (secondary miss) and nothing should be done but waiting.
If the line is kept in the write buffer, it is forwarded to d0 with a back-write request and the instruction is replayed using the *replay registers*. If the WBB entry that hit was trying to make an L2C request, the request is masked by the d1 control logic, because the line is being moved to the d-Cache during this cycle.

When L2C returns a WBB answer, it can be a “Wake-Up” or a “LineReplacement”. In the first case, the WBB line which hits in the buffer is woken up. In the second case, the

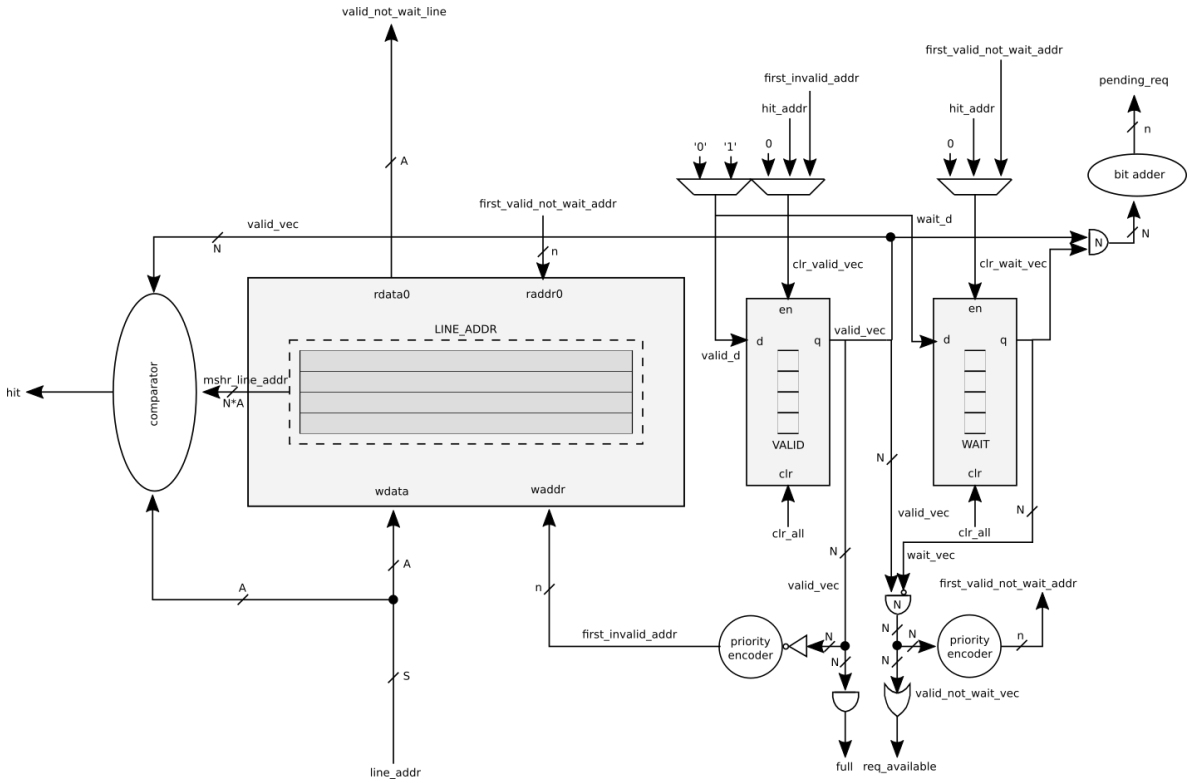

Figure 5.16. MSHR main schematic.

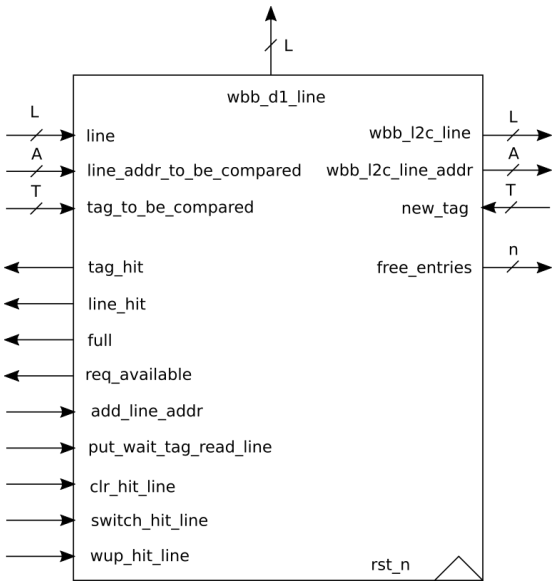

Figure 5.17. Write Back Victim Buffer interface.

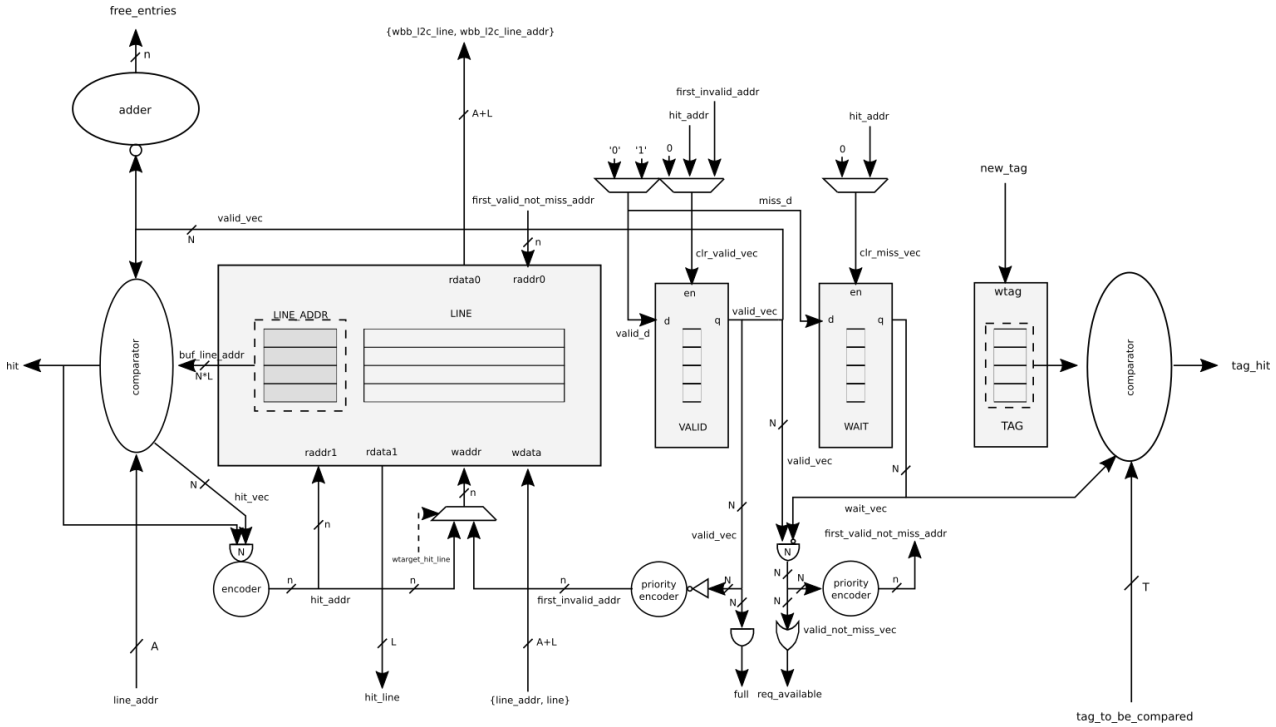

Figure 5.18. Write Back Victim Buffer main schematic.

line which hits in the buffer is cleared. It can happen also that no line hits in the buffer because it has been previously forwarded to the cache. In this case, nothing happens.

Requests to L2 When an L1 primary miss occurs, a new entry of the MSHR is allocated to save the address of the missing line. Then, if the buffer has enough free entries to allow for a possible line replacement, the L2 cache is queried to bring down the hierarchy that line. When L2C returns it and in case of a full set in the cache, a line should be evicted from the cache to free up space. If it is dirty, an entry of the WB Victim Buffer should be allocated to write back that dirty line. The buffer is always capable of saving that line because otherwise the MSHR request would not have been fired. If the MSHR is full, there's the need for waiting for L2C responses to free up space in the MSHR or the WBB. Therefore, when L1 is stalled, L2 should be able to fulfill requests and the d-Cache should be able to elaborate L2C responses. The policy of avoiding MSHR requests to L2C if the WBB would not be able to store all the hypothetically evicted lines is due to possible risks of deadlocks. They can occur if L2 is stalled because is trying to answer to L1 which is stalled because is waiting for the buffer to write to L2. This could happen if the MSHR fires more requests than free lines in the buffer.

This deadlock can be avoided in more ways. One method is to use a counter to keep track of the in-flight requests. Only when a write-back has success or when an L2 answer does not evict a dirty line the counter is decremented. When the counter is full, no more miss can be handled by L1. Another way is to use a block that takes in input the number of MSHR valid and waiting entries and the number of free buffer lines. The MSHR can be filled, but the number of requests to L2 is limited.

In the case of this project, we can exploit the single port of the L2 memory. Since only one type of request can be fired to L2, the MSHR and the WBB compete with each other and an arbiter should decide who will talk next. The key idea is to normally give priority to the read (MSHR) in case of ties, but prevent the MSHR to make other requests if the number of waiting MSHR-requests (which are the in-fligh ones) is equal to the number of the free buffer entries.

Operation flow

These are the possible flows of different incoming requests. The operation in d0, if exists, will be executed only if the transaction is completed (d1 stage should be ready!).

If the next instruction for d1 is "Idle", it means that no information should be transferred through the output register, and so no update will be requested even if the d1 stage is ready. If the actual d0 instruction is "Idle" the memory is not active. It is also possible that one of them is Idle and the other not Idle and vice versa.

- Load:

  d0_req_type_o = ReadLsq
  d1_next_req_type_o = Load

  The cache is read using as address the index of the incoming paddr[11]. This operation

[11] physical address

is requesting a set of lines, together with their tags, valid and dirty bits. Each line has its tag, valid bit and dirty bit.

In the next cycle, the cache produces its output: a tag vector, a valid bit vector, a dirty bit vector, and the line vector. The tags of the set are compared to the tag portion of the sampled paddr of the load request. In parallel, the portion of the paddr which identifies the line is compared to the content of the MSHR and of the WBB. Different endings are expected.

- Cache hit, WBB miss, MSHR miss: the more likely case (hopefully!). If the LSQ is ready, the data is returned to it, explicitly addressed. The d1 stage is ready for a new request.
- Cache miss, WBB miss, MSHR miss: the worst case. This is a full miss, and an entry of the MSHR should be allocated. If the MSHR is full, or if there are too many in-flight L2C requests (more than the dimension of the WBB) the d1 stage is stalled. When d1 is stalled, only L2C answers can reach the d1 stage. When the stall is resolved, the load is fully replayed.
- Cache miss, WBB miss, MSHR hit: this is a not-primary miss. It means that the line request is ongoing, so d1 is ready for another request and nothing is output to the LSQ.
- Cache miss, WBB hit, MSHR miss: the line is present in the WBB (it is rarely possible that this line is or will be present also in the higher level of the memory hierarchy, but this is not a problem). A d0 request is done to write immediately that line into the cache. If the next evicted line is dirty, the two lines are switched: the WBB line is saved in the same position of the line which is being written into the cache. In the same cycle, d1 activates its replay registers if they are present.

No other combinations are possible. A line is either in the cache, or in the MSHR (i.e. the line is stored in a higher level of the memory hierarchy), or in the WBB.

- Store:

  d0_req_type_o = ReadLsq
  d1_next_req_type_o = Store

  The cache needs to be read to check if the requested line is present. If this is the case, the cache is written in the second cycle with a d1 -> d0 request. It's important to point out that the store can be allowed to write Bytes, Halfwords, Words or Doublewords.

  The operations to handle line misses are the same as for load requests.

  If a line is forwarded from the WB Victim Buffer to the cache, the store can be either replayed waking up the instruction in the LSQ or directly done without checking for tag hit, using the replay registers if are present. The second strategy allows for better throughput because the store will take only one cycle (direct write).

- L2 returns a line:

  d0_req_type_o = ReadL2
  d1_next_req_type_o = ReplaceReq

When L2 returns a line, the LSQ instructions are instantly woken up (or are woken up in d1, it depends on the replay registers policy). In d0 the cache is read to understand if the line to be replaced is dirty.

In the next cycle, D1 checks the line to be replaced, and if dirty saves it into the buffer. N.B. in this stage a stall cannot occur. This is because L2 in-flight requests are kept under control by the limit counter, and a new request is done only if there will be enough space into the WB Victim Buffer. In the same cycle, a D1 Write request is done.

- L2 returns an acknowledge for the buf store:

  d0_req_type_o = FwdL2StAddr
  d1_next_req_type_o = L2StHit

  When L2 returns an acknowledge for the WB Victim Buffer, d0 should only forward the address of the successful store. The instruction for d1 will percolate through the register and the buffer will be informed during the next cycle.

- L2 returns a wake up for a WBB entry:

  d0_req_type_o = FwdL2StAddr
  d1_next_req_type_o = L2WBBWakeUp

  When L2 returns a wake up for the WB Victim Buffer, d0 should only forward the address of the WBB entry to be woken up. The instruction for d1 will percolate through the register and the buffer will be informed during the next cycle.

- D1 load request:

  d0_req_type_o = ReadD1
  d1_next_req_type_o = Load

  This is a replay request for a normal load operation. The only difference from an LSQ load is that the source of the request information is taken from d1.

- D1 store request:

  d0_req_type_o = ReadD1
  d1_next_req_type_o = Store

  This is a replay request for a normal store operation. The only difference from an LSQ store is that the source of the request information is taken from d1.

- D1 write clean line request:

  d0_req_type_o = WriteCleanLineD1
  d1_next_req_type_o = Idle

  This is instantaneous writing of the line into the cache, with information taken from d1. The line should be marked as clean. It is needed when a line has to be replaced, after L2C returns a line.

- D1 write dirty line request:

  d0_req_type_o = WriteDirtynLineD1
  d1_next_req_type_o = Idle

This is instantaneous writing of the line into the cache, with information taken from d1. The line should be marked as dirty. It is needed when a line has to be replaced after a WBB to cache forwarding.

- D1 write Store request:

  d0_req_type_o = WriteStoreD1
  d1_next_req_type_o = Idle

  This is an instantaneous write of a doubleword into the cache, with information taken from d1. It is needed when during the second stage of a store.

- Update L2:

  d0_req_type_o = ReadUpdate
  d1_next_req_type_o = UpdateL2

  When an L2 update request is fired, the cache is queried by the update L2 block. In d0, the cache is read using the address of the update block. In the next cycle, in d1 the dirty bits are checked to understand if, in the current set, dirty lines are present. If this is the case, a direct "CleanDirty" request is sent to d0 (highest priority!), and the line is written into the buffer if there is at least one free entry. Stalls are possible here and can be resolved thanks to the priority model of the d0 requests (L2 returns have higher priority wrt the UpdateL2).

- D1 clean request:

  d0_req_type_o = CleanDirty
  d1_next_req_type_o = Idle

  During the updating of L2C (synchronization process), dirty lines should be written to the WBB and then marked as clean.

- Reset the cache:

  d0_req_type_o = ResetLines
  d1_next_req_type_o = Idle

  The cache is being reset. Each line is invalidated.

Replacement block

The replacement block, already introduced in 5.3.2, is depicted in figure 5.19.

Reset block

In the beginning, the caches should be reset. L1 D-Cache is reset via the dcache_rst_block. When rst_n resets the machine, the counter starts counting and sends to d0 a valid reset signal. It has the highest priority. Until the cache is reset, it is not ready for any other block.

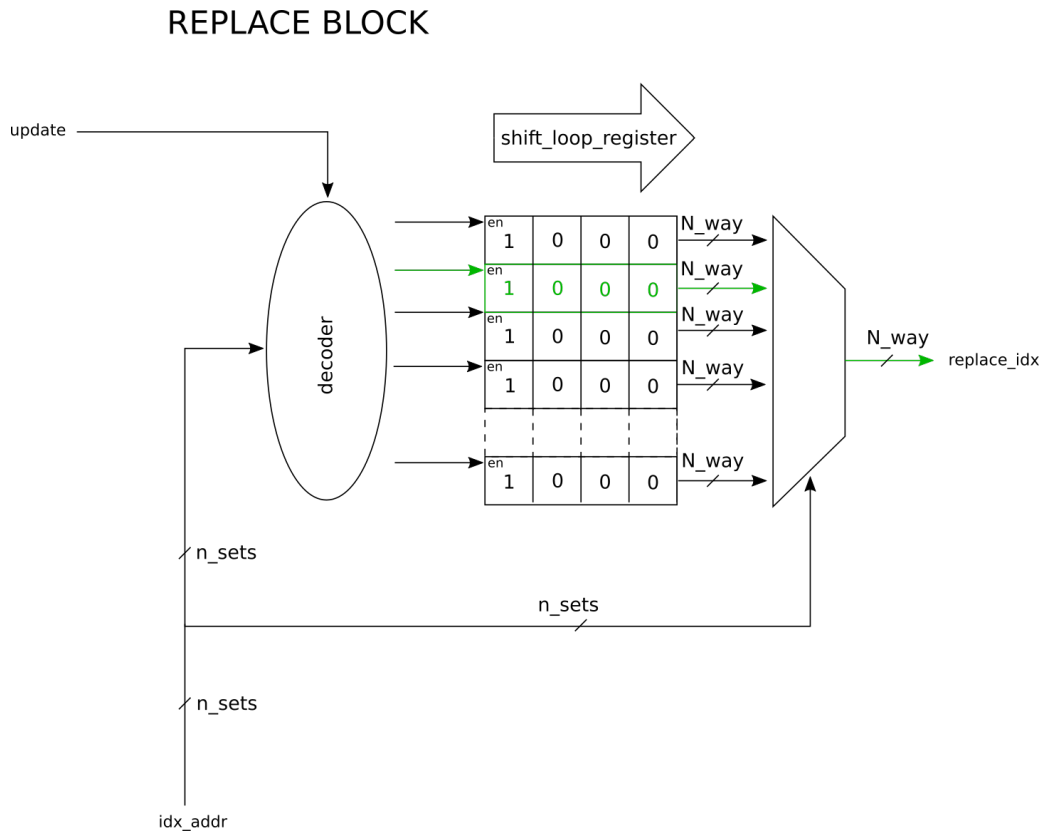

Figure 5.19. d-Cache FIFO replacement block..

Updating L2 Block

The L2 updating process is controlled by "updateL2_block", with a counter, a CU and the cache controller.

The L2 updating should be an unstoppable process when it begins but can be paused by higher priority requests, like L2 answers. These answers will free the WB buffer or the MSHR, and wake-up instructions in the load/store queue. In reality, when an L2 updating is ongoing, no L2 replacement line answers are to be taken into account, because the instruction "fence.i" is executed at commit time only, and the following instructions could be incorrect (an MSHR flush can be done). In an optimized version of the cache, the L2 level can acknowledge the WB buffer store directly without passing through d0. This way an L2 updating process can be unstoppable: when executing, d0_l2c_ans_rdy is always set to '1' (i.e. all valid answers end to valid transactions) and all the interactions between d0 and L2 are filtered because are not important.

A counter indexes the cache starting from the first index. The dirty bits are one-hotted, and if there is at least one dirty line AND the store buffer has at least one free entry, a signal is given to d0 (highest priority request) to update the dirty bit and read another time the line. If the one-hotted dirty vector is equal to the original dirty vector AND the write buffer has at least one free entry, OR if the dirty line is zero, the counter is incremented and the next line is read. When the TC is reached AND a zero is detected in the dirty vector, a "done" is returned to the system. If d1 is stalled, no request is done until a new buffer entry is made available.

d1 samples every request (if it is ready) and acts consequently. If a request "Update L2" is sampled, it is checked to see if at least one line of the set is dirty. If this is the case, one line is copied to the buffer and marked no more dirty. The command to mark the line no-more-dirty is given in parallel with the buffer copy, therefore every single update is atomic and cannot be split by any L2 request.

The memory can't stall, so each cycle has to be re-read. This is suboptimal, especially for large dirty sets. Moreover, when a dirty bit is cleared, the successive read suffers a cycle latency penalty. The lines with the dirty bits can be saved and then the dirty bits can be cleared altogether when all the lines have been copied to the buffer. In our design the sets are small, so maybe it is better to keep all simpler and re-read each cycle the memory.

5.3.4 Further improvements

The d-Cache can be on the critical path. This information can be known only after a full synthesis of the processor, or at least after synthesis of the memory system and the other parts of the core. For this reason, maybe other levels of pipeline will be requested and added.

The pipelining of the d-Cache seems easy and can be done without harm. The requests made by the LSQ are all fully independent. If many signals in a stage should be combined to make a decision, they can be delayed and accumulated via the pipe registers. If more pipeline stages are added, the controls and the data can percolate through the pipeline. If the d0 back-access is maintained only in one of these stages, the modifications to the design can be limited. If some stages need to stall, their ready signals can be brought back to the previous pipeline levels without passing through other registers. Each stage can

sample the next request only if it is valid and if the stage itself is ready. Otherwise, they can stall if not ready, or be cleared if the next request is not valid.

If the ready should pass through registers, skid buffers may be necessary.

The main problem is with the request right after the physical cache, which cannot be stalled. That request can be either replayed when the stall is resolved, or its request can be replaced with a wake-up signal for the LSQ to replay the instruction without too much harm.

5.4 L1 d-TLB

5.4.1 Requirements and constraints

Nonblocking behavior

The d-TLB should be non-blocking to maximize the performance of the Out of Order back end. When the LSQ makes an address translation request, it can hit or miss. In the first case, the translated address is returned. In the second case, an entry of the MSHR is allocated if the miss is primary. The MSHR stalls only if the MSHR is full.

Wake Up signal When the L2 TLB returns the information about the request that missed, a *wake up* signal (tagged with the corresponding VPN) is given to the LSQ, to wake up all the sleeping entries associated to that miss and let them replay. Also, the exception and the PPN are forwarded. The first is given to all the LSQ, the second only to the *Load Queue*, because the Store instructions can still raise an exception if that page is not writable.

Particular checks and exceptions

In addition to what has been said for the i-TLB, the d-TLB should also check:

- The Dirty Bit: if a store requests for a clean page, an exception is raised to let the software marks the page as dirty.
- The Write Bit: if a store wants to write a read-only page, an access exception is raised.
- The User Bit, the SUM bit and the effective Privilege Mode.
- The eXecute bit and the MXR bit.

For a full reference, see 5.0.1.

5.4.2 Design Choices

To ease the separate design of the LSQ and the Memory System, the d-TLB and the d-Cache requests happen in different moments and they are fully decoupled.

Replacement policy

The chosen replacement policy is a PLRU implemented with a binary tree. For this reason, the number of entries should be a power of two.

Main parameters

The d-TLB is fully associative with 8 entries realized as a set of registers, to allow an efficient PLRU replacement policy. In case of too high miss rates, it's possible to vary this parameter. This choice is very similar to the one for the i-TLB; indeed, the motivations are similar. The only difference is that the d-TLB is fully decoupled from its reference cache. Therefore, maybe also 16 entries can be good.

5.4.3 Design

Interface

Implementation

For the first implementation, a scheme similar to the d-Cache was used (fig. 5.21). However, this block is designed to answer within the same clock cycle when the request hits. As for the d-Cache, it will be easy to add additional levels of pipeline if needed.

Arbiter/Scheduler There are three types of request the d-TLB can fulfill during the clock cycle, ordered in descending priority order:

- Flush Request
- L2C Answer
- LSQ Request

In practice, the d-TLB serves the LSQ if it hasn't others to do. The scheduler and the arbiter receive the *Valid*s of the incoming requests, encode a simple control, which is the name of the *winner* among the incoming "requests", and possibly gives a ready to the LSQ.

Control and Data Selection The control and the data selection are performed depending on the control encoded by the *scheduler*, which can be:

- dtlb_Flush
- dtlb_L2Ans
- dtlb_LSQReq

The flush can be selective, as for the i-TLB, and it's performed in the same clock cycle in which the request arrives.

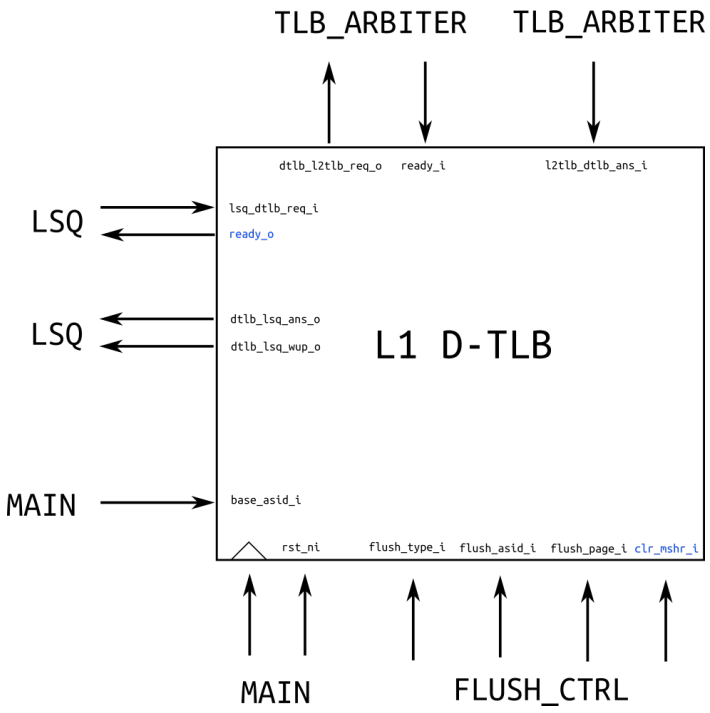

Figure 5.20. Interface of the d-TLB.

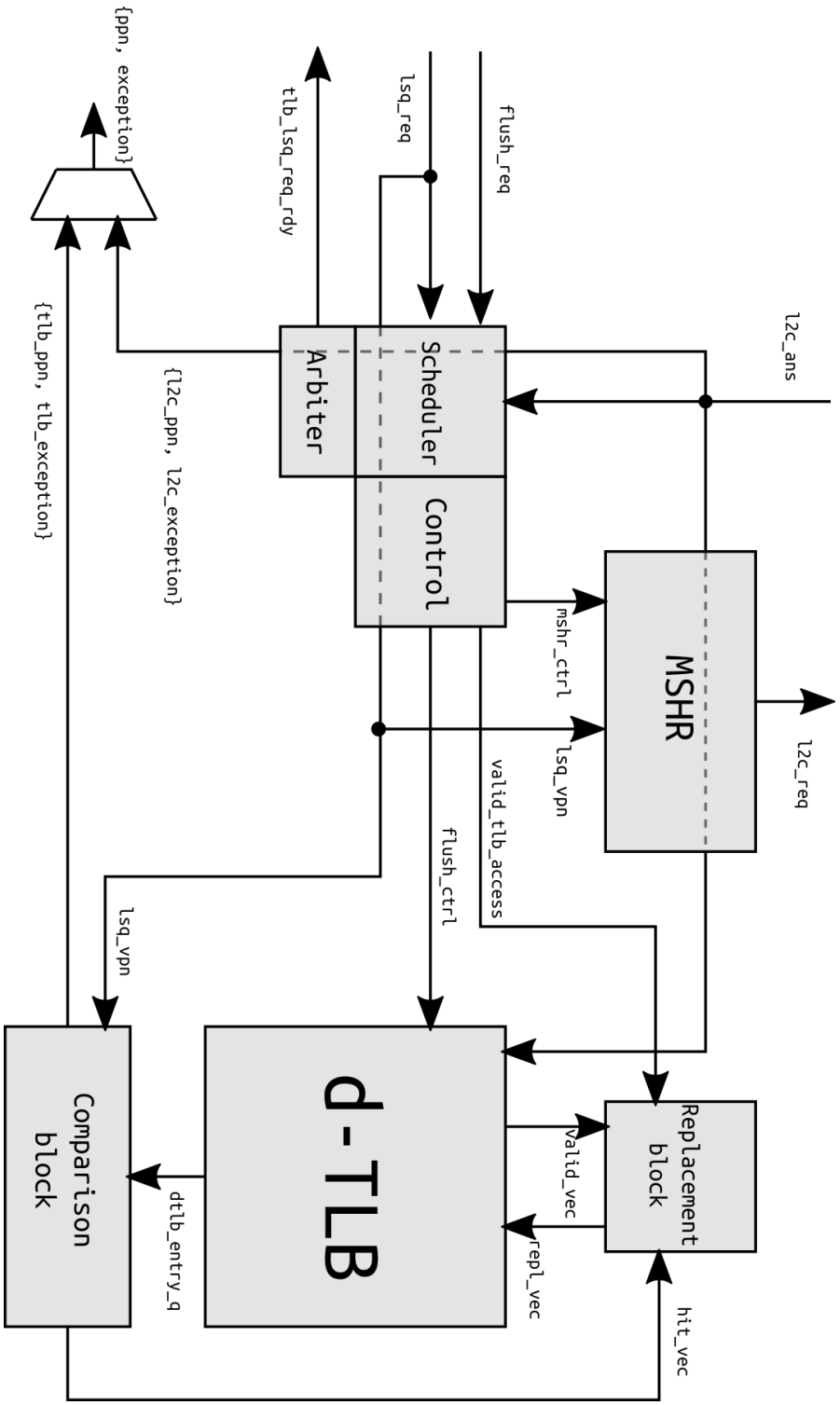

Figure 5.21. Main diagram of the d-TLB.

When the L2 TLB is returning data and the *Virtual Address* hits in the MSHR, both the PPN and the eventual exception are routed to the LSQ. Then, the MSHR entries that hit are removed and the new d-TLB entry is saved in the place pointed by the replacement vector.

MSHR A picture of the MSHR is shown in figure 5.22. The control is not reported, but is composed of signals to enable the writing of a new VPN in a free entry, the removing of the entries that hit in the comparison block (used when L2 TLB returns) and the assertion of the *Waiting Bit* for the entry that triggered an L2 TLB request.

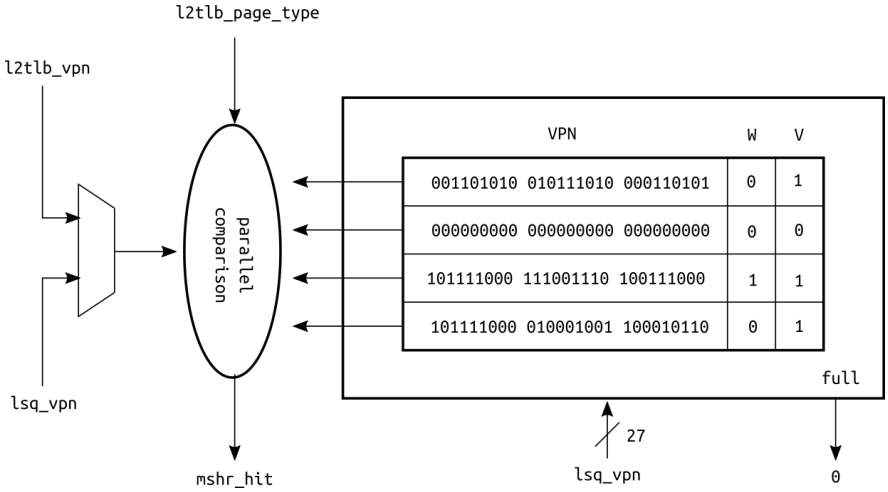

Figure 5.22. MSHR of the d-TLB.

It is composed of **N_MSHR** entries, and each of them hosts a VPN, a *Waiting Bit* and a *Valid Bit*. The *Waiting Bit* is asserted when the request to the L2 TLB for that entry is made.

The MSHR has one comparison port which is shared by the L2 TLB answer and the LSQ request. Only one of them at a time is routed into the MSHR. When the MSHR is full, no other requests can be performed until an entry is freed.

If a *superpage* is requested by at least two instructions that differ in their least significant part of the VPN, more entries of the MSHR will be reserved. This brings to a necessary loss of performance because the MSHR risks to be filled in advance by secondary misses: but it's not possible to know in advance if the requested page is a *superpage* or not. When L2C returns, it's necessary to remove all the entries related to that hypothetical *superpage*. For this reason, the VPN returned by L2C is compared with the VPNs inside the MSHR, but the *superpage* type determines which part of the address is compared for each entry. All the entries associated with the same *superpage* are removed in parallel.

If L2 returns something which is not in the MSHR, no entry is removed. This way, if more requests were done for the same *superpage*, nothing happens.

When a *fence.vma* is performed, all the MSHRs in the whole hierarchy (and the PTW too!), should be flushed completely and no pending requests/spurious answers should percolate. The L2Cache should also selectively clean all the PTW pending requests to avoid

spurious returns in the TLB.

Comparison Block The comparison block is very similar to the one of the i-TLB. Only the checks are different 5.0.1.

Replacement Policy The adopted replacement policy is a tree PLRU, with 8 entries. If not all the entries are valid, the invalid entries are filled first. Otherwise, use PLRU.

An example of the tree PLRU that shows part of the implementation is depicted in figure 5.23. The one-hot vector on the bottom is the *replacement vector*. every time an access on an entry is performed, the tree is gone back. Every flip-flop on the path should be forced to make the corresponding decoder not to point on that path. Conceptually and referring to the figure, after an access to the entry **X**, the path starting from **X** and ending to the top of the tree should be completely black.

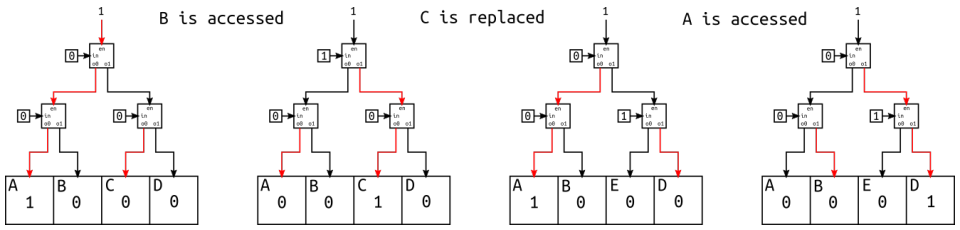

Figure 5.23. Tree PLRU algorithm with 4 entries. Creation of the replacement vector.

There are two distinct parts of the implementation: the creation of the replacement vector and its updating.

- **Creation of the replacement vector**: a binary tree of 1 input bit decoders with enable, with one decoder of one level which has its two output bits connected with one decoder-enable of the lower level each. Each decoder is fed by one D-FF. Initially, all the FF are reset to zero.

- **Updating**: when a d-TLB access is done, the hit vector is given to the replacement block. This hit vector defines the enables and the inputs of the FFs in the last level of the trees. Each couple of bits of the hit vector is combined with its FF: if at least one of the couple is 1'b1, the FF is updated. Each input is the LSB bit of each couple. This is because if the two bits are zero or one, no update is done. If they are different, then the decoder related to the FF has to point to the opposite direction of the enabled bit. For the other levels, it's the same, but in place of the hit vector, it is used the vector of the FF enable of the lower level.

If a d-TLB access raises an exception, the access is not considered valid and the replacement block is not updated.

5.5 TLB Arbiter

The TLB Arbiter is very similar to the L2C Arbiter (5.9). The only difference is that the two L1 TLBs are connected to the L2 TLB. In case of a tie in the request part, the priority is inverted. This way, each block wins a tie before losing the next, and so on. A quite fair policy.

5.6 L2 TLB

5.6.1 Requirements and constraints

High hit rate

The L1 TLBs are small and fast, to allow a high hit time and paying it in terms of miss rate. The L2 TLB should have a high hit rate to make up for it: this implies a large size and high associativity.

Nonblocking behavior

Like the other TLBs, it should not stall if a miss occurs. The misses in this block are handled in-order because the PTW can support only one miss at a time.

Particular checks and exceptions

Refer to 5.0.1 for a full reference.

Design Choices

Size and Associativity

Taking example by the *ARM Cortex A53*, the initial values of the design were set to:

- 512 entries
- 4-way set associativity

The physical structure of the TLB is relized with an SSRAM, like for the d-Cache.

Multiple page size support

As discussed in the previous chapter (2.3.4), the L2 TLB can support multiple page sizes paying with cycle times. The adopted strategy is *hash-rehashing*. During the first request, the L2 TLB is probed using the index relative to the minimum possible page size. If it misses, other requests are done with the other page sizes, until at least one hits or the last misses (refer to 2.5 to understand this concept. In the figure, the probing is done in parallel on more TLBs, with *hash-rehashing* is done sequentially on the same).

Replacement Policy

The replacement policy implemented is a PLRU. The TLB implements this clever policy to keep the miss rate as low as possible. The effectiveness of this strategy is worsened by the selective flushes, which create holes.

Design

Interface

The L2 TLB can communicate with the TLB Arbiter and the PTW. A schematic of its interface is reported in figure 5.24.

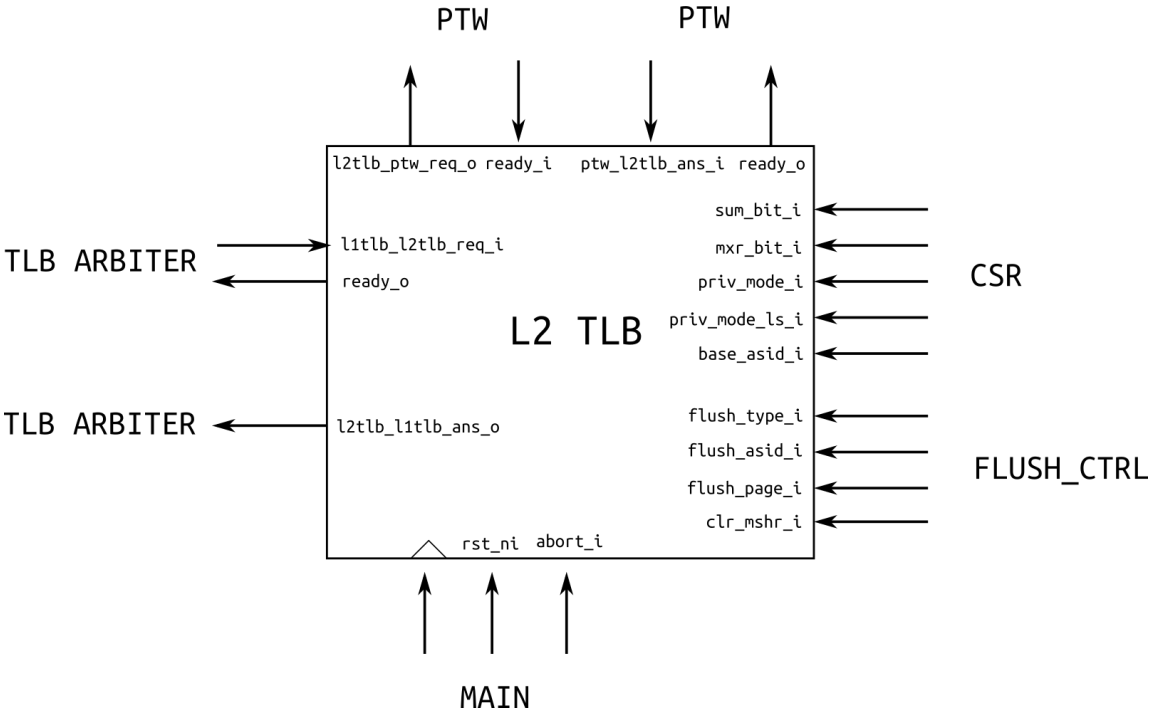

Figure 5.24. L2 TLB MSHR.

The *Abort* signal can interrupt all the actions performed by the TLB in the current cycle. The *Clear MSHR* signal can clear all the MSHR entries. This combination of signals brings the MSHR to a ready condition and deletes all the pending miss requests.

The Flush control is the same as for the other TLBs.

Implementation

In figure 5.25 is reported the high level diagram of the L2 TLB, and in figure 5.26 more details are presented. The design principle is the same as for the d-Cache, to ease a future piplining.

In this first design, the main structure is divided only in two regions: **t0** and **t1**. In the first part, all the incoming requests are collected by the *Scheduler*. This unit, with the aid of the *Arbiter* and the *Register Enable*, conditionally gives the *ready* signal to the *winner* block (the block that will use **t0** within this cycle). An instruction for both the regions is encoded in **t0**, and one of them is used, the other is sent to the *t0_t1_register* (to be used in the next clock by **t1**).

The L2 TLB cannot stall, because **t1** requests on **t0** have the highest priority, and the lower level TLBs are always ready for an answer. If the MSHR is full, no additional requests can be processed.

Flush Unit The L2 TLB flush is controlled by the **Flush Unit**. For a complete flush, is addresses the memory and clears all the valid bits of a set. Then, the next set is flushed and so on until all the sets have been cleared (only write cycles) If the flush is specific for an *ASID* or a *page*, the memory is firstly read and the information on the flush type is given to t1. Then, a comparison is performed in t1 to select only the correct entries to be cleared within the set. This information is sent to the memory controller in t0. The memory is then written before the flush address is incremented (Read cycle + Write cycle). The flush address is incremented each time a successful write operation is performed during a flush. The memory controller performs write requests if the flush is not selective, read requests in the other case. every time a selective flush instruction percolates through the registers to t1, a write back-request is done to t0 (t1 has the highest priority).

t0 Four types of requests/answers can arrive to the L2-TLB:

1. Requests from L1-TLBs
2. Answer from the PTW
3. Request from the flush Unit
4. t1 requests for the alternative superpage indexing and the selective flush

The winner request is decided by the *scheduler*, which gives this information to each other block.

Memory Controller The memory can be controlled in different ways:

1. Read a set. It can be requested by an L1 TLB request, a t1 read *superpage* request, a selective flush read, a PTW answer. Therefore, from L1 TLB, t1, flush unit, PTW.
2. Flush a whole set. It can be requested by the flush unit to perform a total flush.

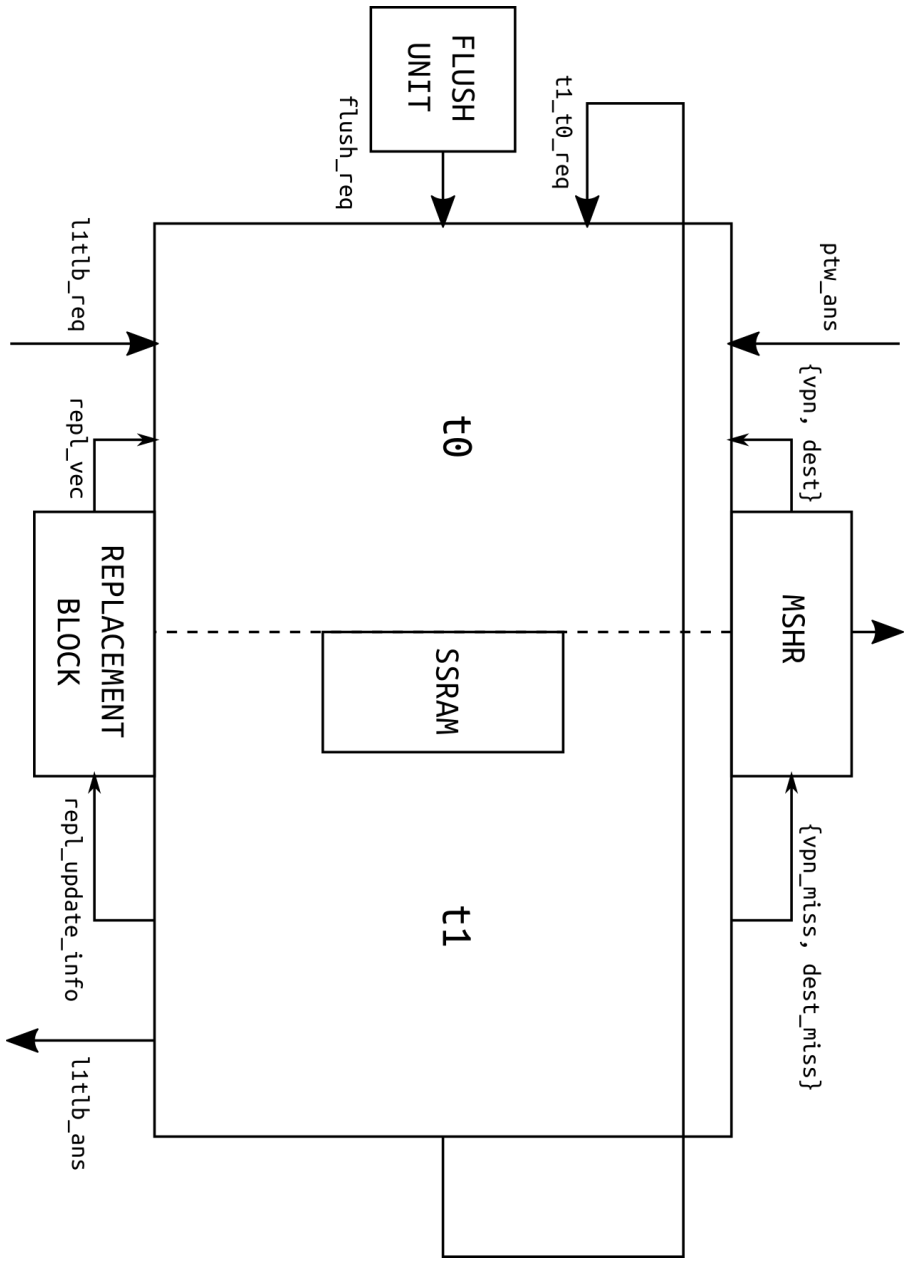

Figure 5.25. L2 TLB main general schematic.

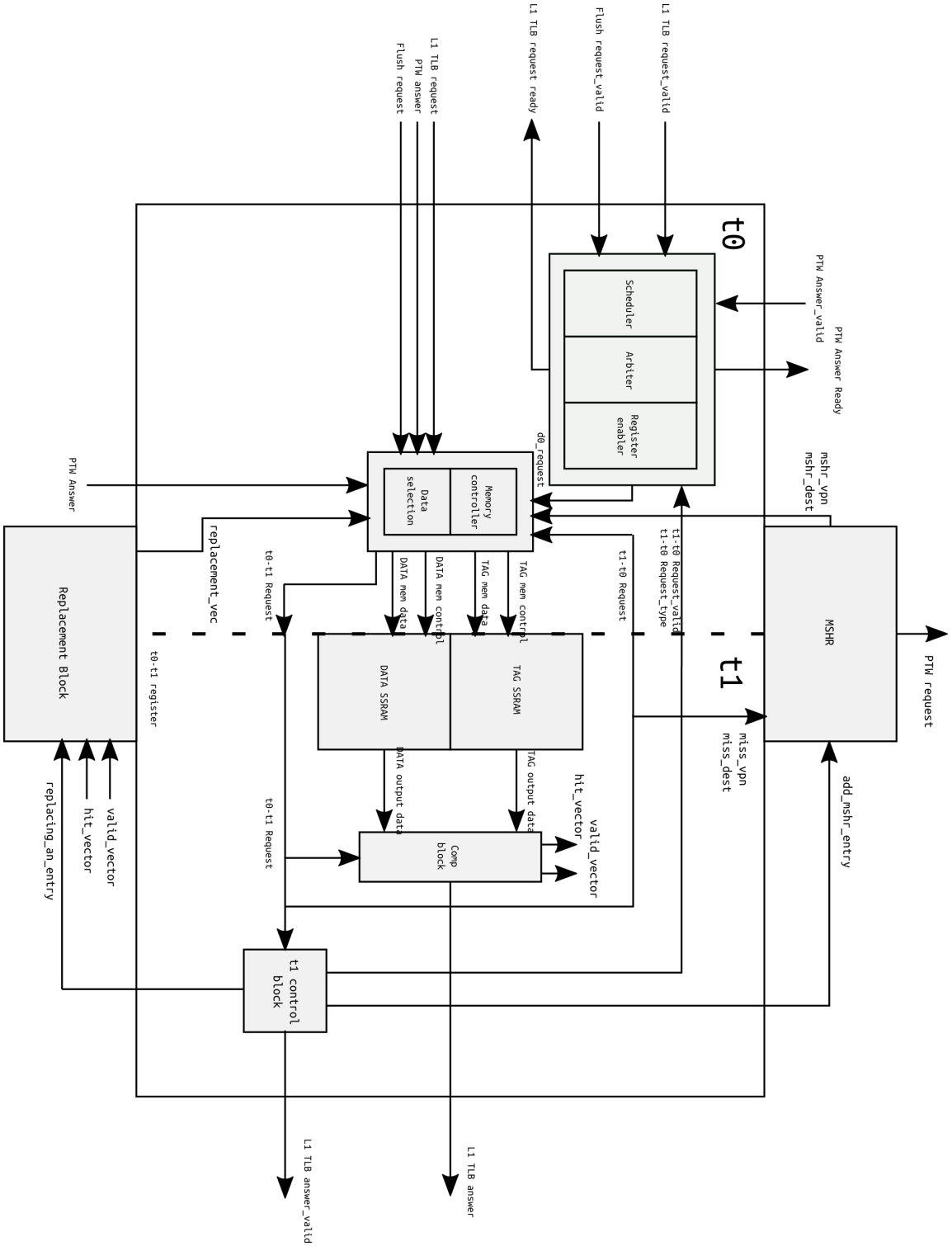

Figure 5.26. L2 TLB more detailed general schematic.

3. Selective flush with a mask. It can be requested by t1 to perform a selective flush.

4. Write an entry of the set. It can be requested by t1 after a PTW answer.

Data Selector The possible requests **t0** can send to **t1** are:

1. t0_t1_Idle

2. t0_t1_KibiRead

3. t0_t1_MebiRead

4. t0_t1_GibiRead

5. t0_t1_FlushASID

6. t0_t1_FlushPage

7. t0_t1_PTWAns

The possible requests **t1** can send to **t0** are:

1. t1_t0_MebiRead

2. t1_t0_GibiRead

3. t1_t0_ReplaceLine

4. t1_t0_FlushMasked

When an L1 TLB request is accepted, a *KibiPage* read is performed (it means that the index bits to address the memory are taken as the page was a KibiPage), and an instruction *t0_t1_KibiRead* is given to **t1** for the next cycle. This information is used for the tag comparison and to trigger an eventual *rehashing* with a **t1 -> t0** back request. If this is the case, the TLB is rehashed two more times with a *t0_t1_MebiRead* and a *t0_t1_GibiRead*.
If a total Flush request is accepted, all the Valid bits in the tag memory are cleared, one set per clock cycle; this involves only write operations and no instructions to **t1**. If a selective Flush request is accepted, the memory is read set by set, and each time either a *t0_t1_FlushASID* or a *t0_t1_FlushPage* is sent to **t1**. One cycle later, a vector containing a **1** in correspondence to all the entries that should be flushed in the set together with a back write request (t1_t0_FlushMasked) are sent to **t0** from **t1**.
When the PTW answers, the memory is indexed and read using the correct information about the page size, and a *t0_t1_PTWAns* instruction is injected in the **t0 -> t1 register**. The memory is read because if some lines in the set are invalid, they should be filled first. In the next cycle, a back-write request (t1_t0_ReplaceLine) replaces the line.

TLB Entries Each TLB entry contains:

1. VPN
2. PPN
3. Read access bit
4. Write access bit
5. eXecute access bit
6. Dirty page bit
7. User page bit
8. MebiPage
9. GibiPage
10. Global
11. ASID
12. Valid bit

The Valid Bit is the less significant one, to easily flush the L2-TLB.

t0 -> t1 Registers For power saving reasons, the t0 output registers are enabled only if there is a valid t0 -> t1 request, and are cleared if there isn't.

t1 In **t1**, the output of the memory is checked and compared to generate the *hit vector*, the *valid vector* and the *flush vector*. In addition, the back request **t1** -> **t0** is composed.

Exceptions The L2 TLB performs a lot of checks for the exceptions, and if one is raised, the L2 TLB is not updated (the **t0 -> t1** back request is masked).

L2 TLB Comparison Block The comparison can be done for hit checking or for flushing. When a page flush is requested, the page is checked depending on its page size.

For the hit checking, each request has three possibilities. This is because the L2 TLB is first probed for a *KibiPage*, then for a *MebiPage*, then for a GibiPage. For each page size, the requested set can be different. When comparing the result with the tag, it's not necessary to check the page size of the entry during the first probing. Indeed, each page in the TLB can be present in one size only, because if it is modified, it is flushed. When probing, the page checking starts from the *KibiPage*, the most restrictive comparison. If a *Mebi/Gibi* page matches the entire tag during this first probing, no harm is done if the L1 answer page size is produced using the *Mebi/Gibi* bits of the L2 TLB entry, which reflect its real size. This is because that page would have hit later during the second/third probing because if it is present in the TLB, it can be present in one place only. On the contrary, the size should be checked on the successive readings. A lot of *KibiPages* can be

present in the TLB even if they refer to the same *Mebi/GibiPage*. If the requested one is present it should hit, otherwise, it should miss, but the other *kibipages* should miss for sure! The same argument can be presented for *MebiPages* in GibiPages and for *KibiPages* in *MebiPages*.

The comparison ends if a hit occurs or if a GibiRead misses. In the first case, the answer to the L1 levels is made and sent (with the hypothetical exceptions). In the second case, the MSHR is updated.

MSHR The MSHR is depicted in figure 5.27.

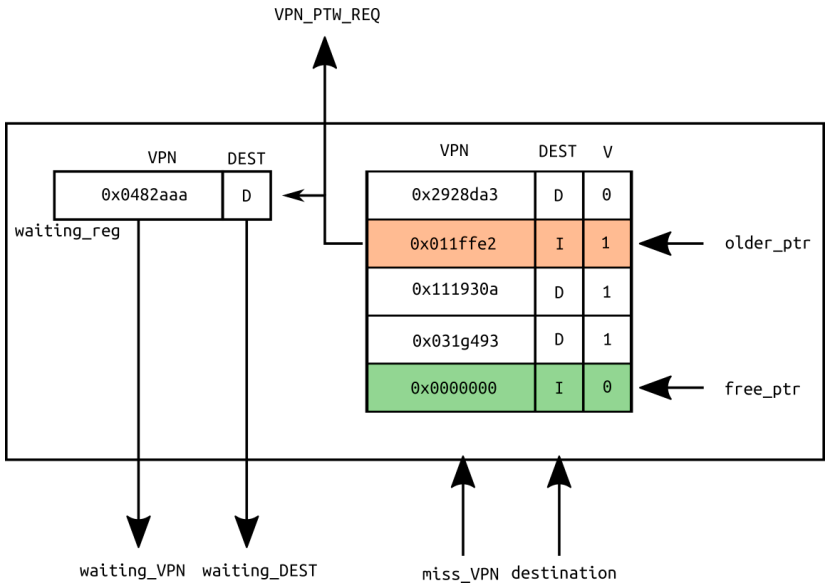

Figure 5.27. L2 TLB MSHR.

Two main reasons simplify the MSHR block:

- The PTW supports only one request at a time (no multiple pages walks allowed).
- If an L2 TLB request misses, the miss is usually primary. The only way to have a secondary miss is to have both the L1 TLBs requesting the same page, which should be at least executable. If this hypothesis will be contradicted, the handling of the secondary misses will be introduced.

Thanks to the first observation, the MSHR of the L2 TLB can be a blocking FIFO for the requests to the PTW. Also, if the second observation is true, there's no need to implement a parallel comparison of the "tags" of the MSHR entries. In any case, if a secondary miss occurs, the page table walks is shorter thanks to the MMUC[12] cache.

[12]Memory Management Unit Cache

When a valid PTW request is made, the corresponding MSHR entry is moved to a special *waiting register*. This register stores the information related to the next PTW answer, and let other MSHR entries be ready to fire a new request as soon as the PTW becomes ready. When the PTW answers, no comparison is needed, because each answer corresponds exactly to what is kept into the waiting register. If flushes/aborts occur, no PTW answer is returned and the entire MSHR is cleared.

MSHR Status and Control

- add_entry_i: add a new couple *VPN - Destination* in the entry pointed by the *free_ptr*. Assert the corresponding *Valid* bit.
- rm_entry_i: clear entry in the *waiting register*.
- let_entry_waiting_i: save in the *waiting register* the entry pointed by the *older_ptr*.
- mshr_full_o: all the entries are valid. No other misses can be handled.

PTW answers When the PTW returns, a read is performed. This is because it is necessary to access the valid vector of the desired set. This set is selected depending on the *superpage* information associated with the PTW answer, and also the TAG is influenced. Then, a cycle later, a back-write request for the replacement is fired from t1 to t0 if there are no pending or fresh exceptions. If the PPN from the PTW keeps an exception with it, t0 does not perform the read and t1 does not back-request anything. If no exceptions come from the PTW but one is raised when the destination field is compared to the access bits, no t1 back-request is done.

Replacement Block The actual implementation for the L2 TLB, in which there is more than one set, is slightly different from the d-TLB one. The base structure is replicated for each set, and when an access is performed, only the accessed set is updated. An access can be both a valid access to an entry (the access vector is the hit vector) and a replacement of an entry (the access vector is the replacement vector). This is because the replacement unit should be able to update the replacement vector of a particular set every time an element of that set is accessed (even if it is replaced!).

5.6.2 Further improvements

It would be interesting to evaluate the impact of having all the *Valid* bits outside the memory in a register. This would allow for less cycle penalty when a replacement occurs and when a flush is performed (and this is true also for the Caches).

5.7 PTW

5.7.1 Requirements and constraints

To be compliant with the RISC-V Specifications

The RISC-V privileged architecture manual sets the precise rules to be followed when an address translation request is made 3.3.3. The PTW should extract the correct PPN from the page table checking for possible exceptions (see 5.0.1).

5.7.2 Design Choices

Clean interface

The Page Table Walk is a slow process by definition. Without the aid of a MMU cache, it requests more memory accesses. Moreover, the memory to which the requests are made is shared also with the i-Cache and the d-Cache. Therefore, some cycles of additional latency are hopefully negligible. For this reason, the interface is clean and each output **Valid** and its relative data are registered.

5.7.3 No support for more outstanding misses

Modern PTW supports more misses at a time, keeping all the information of different requests in different entries of a buffer. For simplicity, this first implementation can serve only one miss at a time.

5.7.4 Design

Interface

The PTW can receive a request from the L2 TLB. The effective transaction happens when the request is valid and the PTW is ready to sample it. Once it happens, the PTW probes the MMU Cache to catch possible hits. The L2 TLB miss penalty will be reduced. For each Page Table level to be accessed, a new request to the L2 Cache is performed. The L2C Arbiter is in charge of routing this request to the L2C as soon as possible: this can happen if the PTW request is the only valid, or if the PTW is the *winner* of the hypothetical tie.

In figure 5.28 is reported the main interface of the block.

The **satp** register 3.3.3 sends the actual *Page Table Root PPN* to the PTW: it is used to address the main Page Table.

The PTW can be flushed, to block all the elaborations and reset it to a ready state.

Implementation

The main schematic of the PTW is reported in figure 5.29.

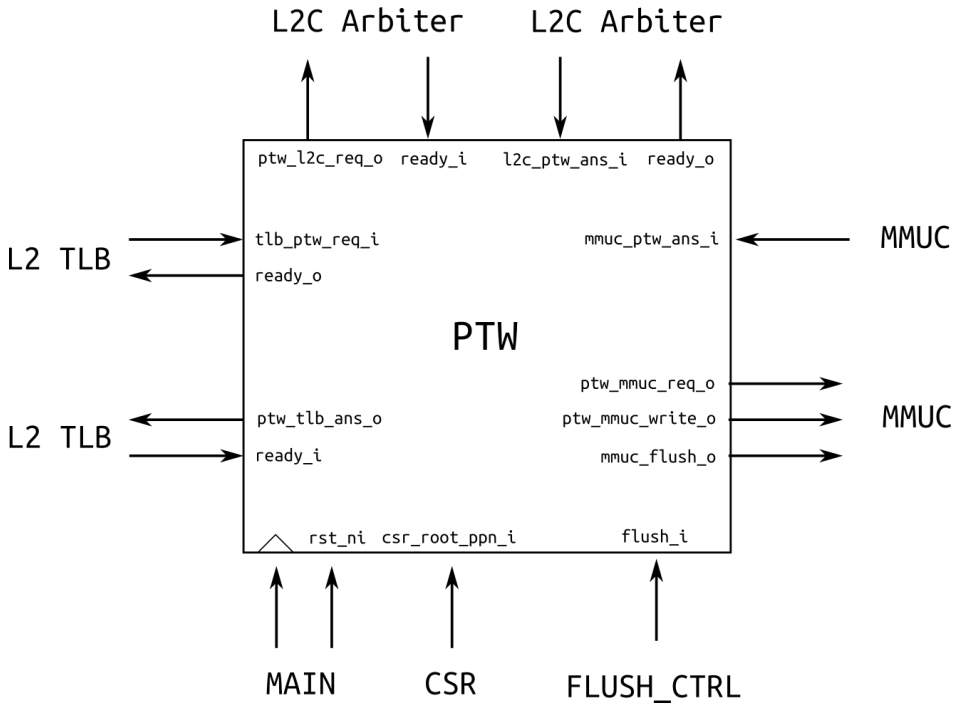

Figure 5.28. The interface of the Page Table Walker.

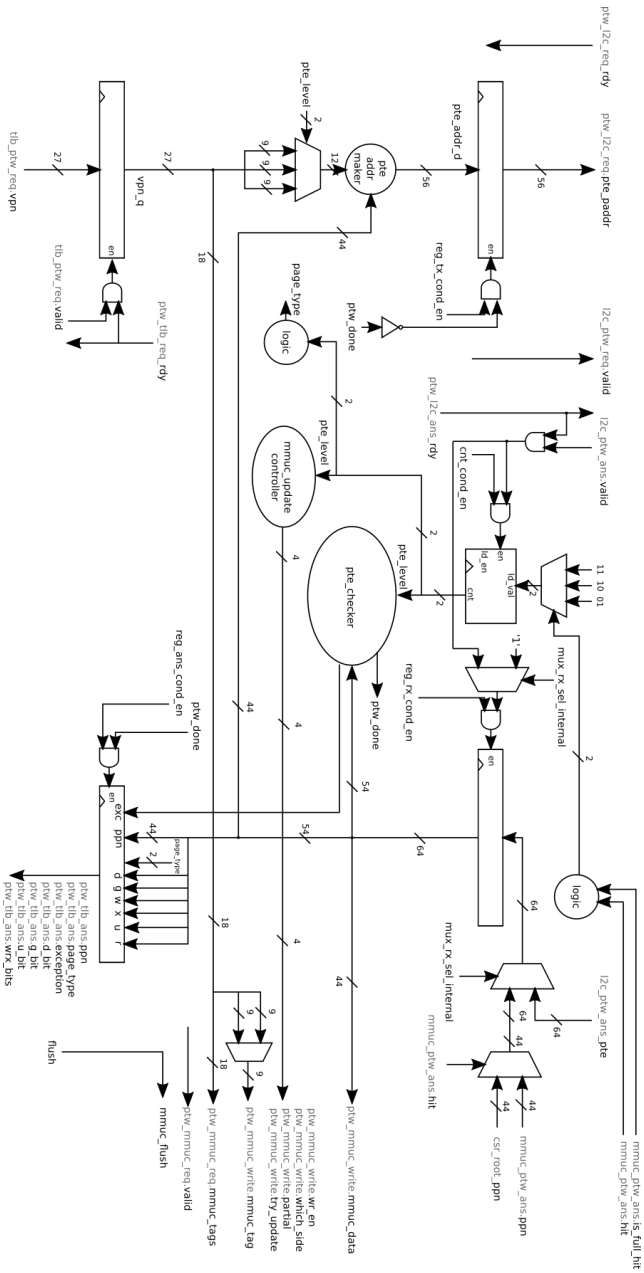

Figure 5.29. The main schematic of the Page Table Walker.

Without the MMUC The PTW repeats the same operations cyclically on different PPN. To understand its working principles, let's pretend that no MMUC is used. To complete the translation process, the PTW should traverse three levels of the Page Table and a total of four PPN will pass through it:

- The **Page Table Root PPN**, to address the third level of the Page Table (a *GibiPage*). It is combined with **VPN[2]**.
- The **The PPN of the third level of the Page Table**, to address the second level of the Page Table (a *MebiPage*). It is combined with **VPN[1]**.
- The **The PPN of the second level of the Page Table**, to address the first and last level of the Page Table (a *KibiPage*). It is combined with **VPN[0]**.
- The **The PPN of the first level of the Page Table**, to address the physical page. It's not combined, because it's what the L2 TLB requested.

When a new request is accepted, a first physical address is formed combining the **Page Table Root PPN** with the **VPN[2]** of the request. This is used to address the first level of the page table: a request to L2C is made and, after some cycles, the answer is returned. The answer is a full PTE, which contains a PPN and the relative control/access bits 3.4.

The PTW performs on the PTE all the checks it is supposed to do (5.0.1), and this operation can bring to a formation of a new physical address or to an interruption of the process. In the first case, the PTE was not a leaf page table entry, and another request to L2C is performed. In the second case, either an exception should be raised or a leaf page table entry has been found. However, a valid answer is given to the L2 TLB.

With the MMUC The presence of the MMUC does not add important differences. In the beginning, it is probed to have, if possible, a PPN of an advanced level of the Page Table. Since the MMUC does not store the final PPNs, only two levels can be bypassed. At least one L2C request is always mandatory. The MMUC cache response can be:

- Miss: no level is bypassed. The PTE register samples the **Root PPN** and the main "sequencer" records the actual request level (**Root**).
- Partial Hit: one level is bypassed. The PTE register samples the **PPN** of the third level of the Page Table (pointing to the second one), and the main "sequencer" records the actual request level (**L3**).
- Full Hit: two levels are bypassed. The PTE register samples the **PPN** of the second level of the Page Table (pointing to the first one), and the main "sequencer" records the actual request level (**L2**).

Then, an L2C request is performed. When the answer is returned, the sequencer is "decremented" by one to keep track of the actual Page Table level following this algorithm:

```
1) L2 Cache request
2) L2 Cache answer
3) (Leaf PTE or Exception) ? 4) : 1)
4) L2 TLB return
```

The control The control unit is a *Moore FSM* and is represented in figure 5.30 and 5.31.

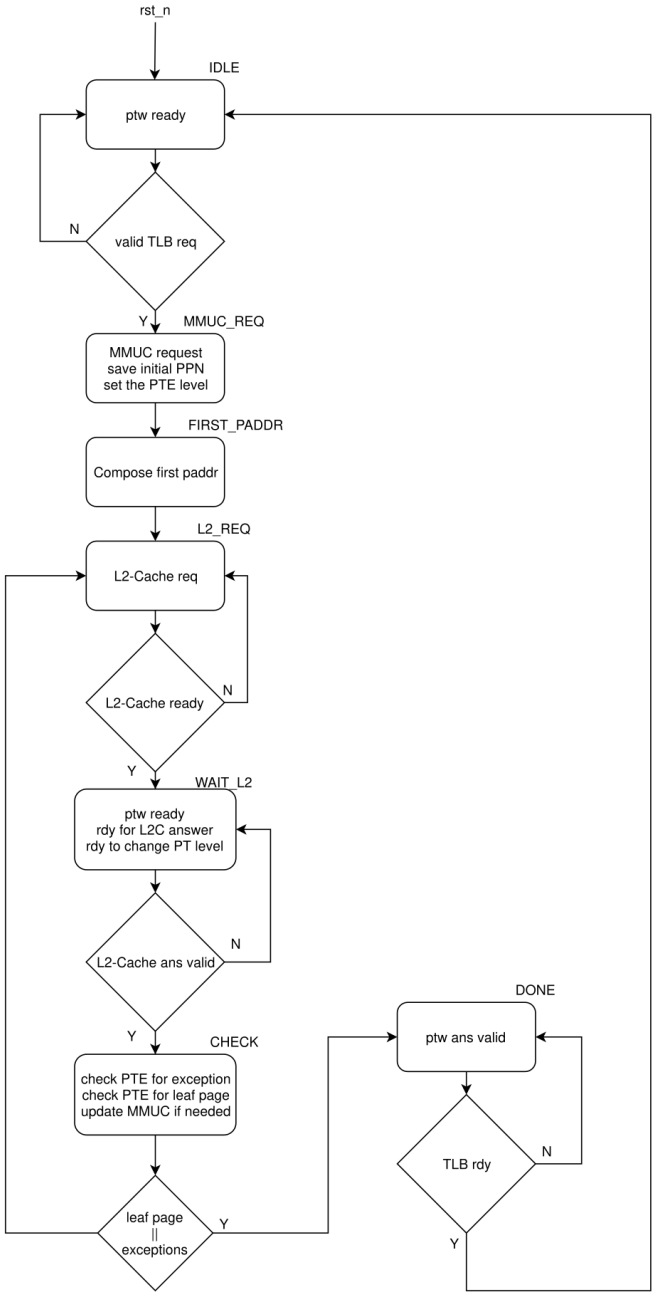

Figure 5.30. PTW ASM chart.

Inside the block, some circuits, like the sequencer, are activated in a combinatorial way with the incoming **Valid** of the L2C answer. The *Moore FSM* enables the final controls

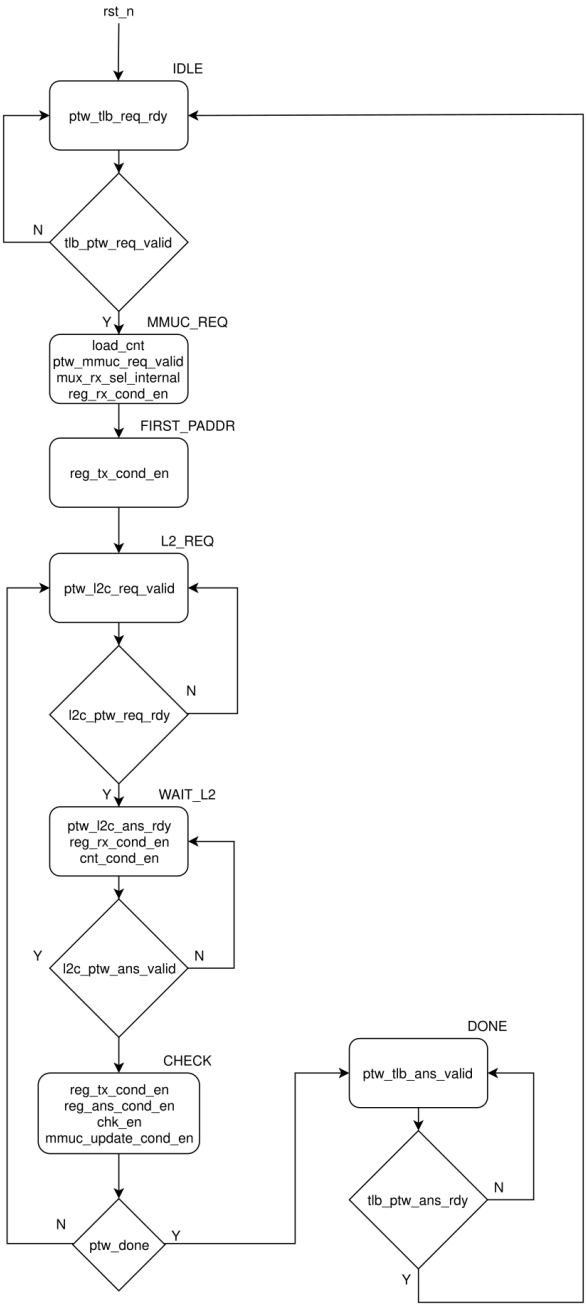

Figure 5.31. PTW ctrl ASM chart.

only in precise states, masking or unmasking them.

The PTE checker The PTE checker performs the checks on the PTE only in the correct states (when a new "PTE" or the Root PPN are sampled). For example, no check should be done when the sequencer is set to "Root" because the incoming data does not represent a real PTE.

MMUC Update Controller A special controller is reserved to handle the MMUC updating process. The MMUC can store only two couple VPN-PPN: one relative to a *MebiPage* and one relative to a *KibiPage*. This is because the information that will be put in the TLB it's not kept also in the MMUC; in case of a *GibiPage*, the PPN found in the third level of the page table after the first request is already sent to the TLB. Since it's not possible to know in advance the page size, the MMUC is updated sequentially.

- If the sequencer is set to **L3** and the algorithm is going on, the page is not a *GibiPage*. Therefore, the first part of the trace ($VPN[2] - PPN_2$) is saved in the MMUC.
- If the sequencer is set to **L2** and the algorithm is going on, the page is not a *MebiPage*. Therefore, the second part of the trace ($VPN[1] - PPN_1$) is saved in the MMUC.

Actual implementation updates the MMUC even if the PTW walk is interrupted for an exception. Given the actual implementation of the MMUC, it's mandatory to flush it when the **Page Table Root PPN** is modified (see 5.8.4 for further details).

5.8 MMU Cache

5.8.1 Requirements and constraints

The MMUC should be compliant with the RISC-V specifications about the structure of the page table.

5.8.2 Design choices

Size and Associativity The MMUC, for its first implementation, is composed of registers and has 10 entries and present a fully associative organization.

Flush The flush can't be selective and clear all the valid bits of the MMUC.

5.8.3 Design

Interface

The interface of the MMUC is shown in figure 5.32. It is in communication with the PTW, which can perform read and write requests, to shorten the page table walk and updating the MMUC. The answer, given within the same clock cycle of the request, is composed of a PPN and a **hit** signal, coupled with **is_full_hit** to qualify the size of the hit. No valid is used for this first design because the PTW knows when the request should be sampled. If the latency is changed, a handshake signal will be added.

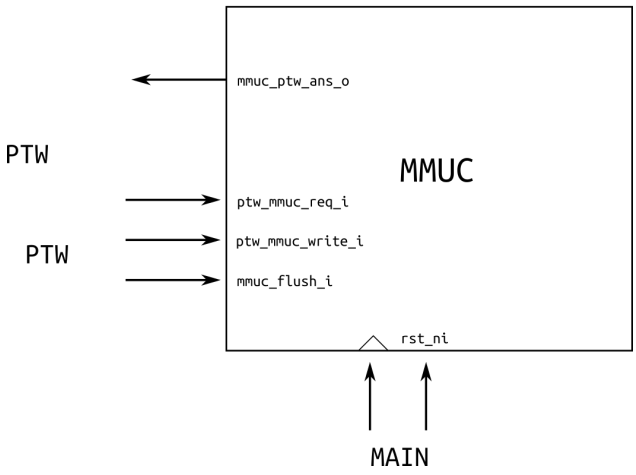

Figure 5.32. MMUC interface.

Implementation

The main schematic is presented in figure 5.33. When the PTW writes a new couple *tag-data* ($VPN[i]$-PPN_i), it uses the write port of a register file, addressed with an already decoded address.

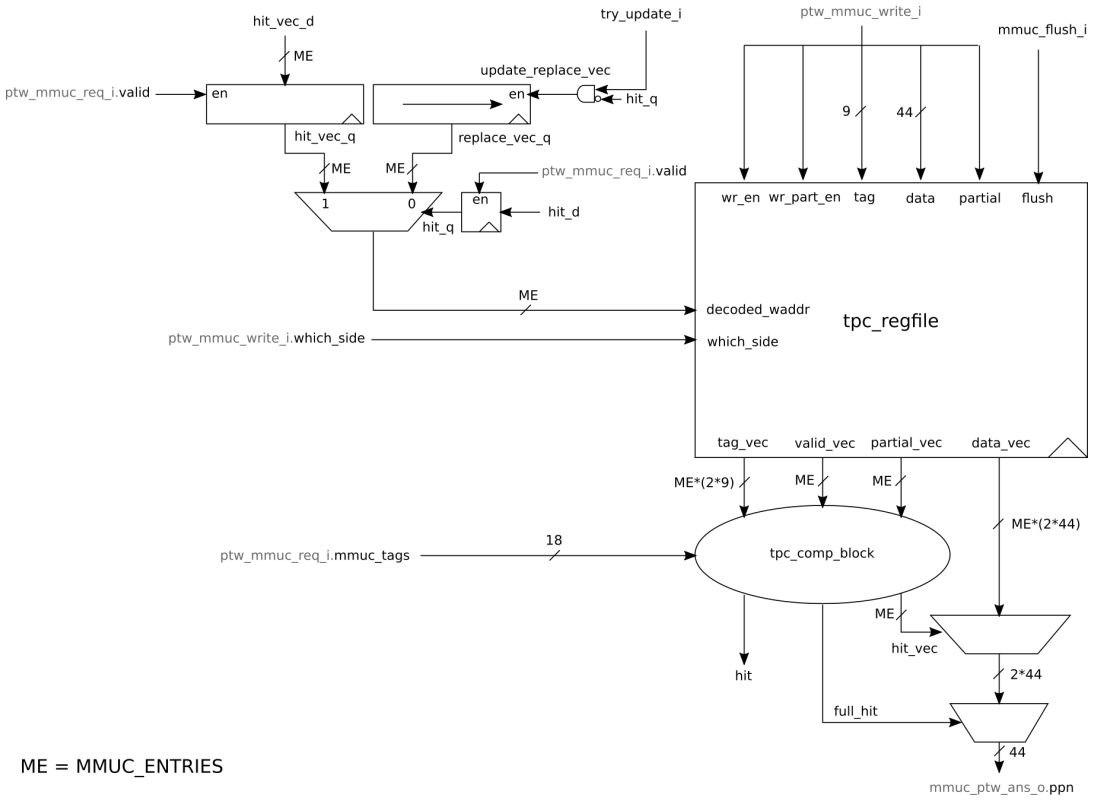

Figure 5.33. Main schematic of the MMU Cache, realized as a Translation Path Cache.

The physical organization of the core cache is depicted in figure 5.34. It is possible to select the part of the tag-data to be written (using *which_side*), and all the register vectors are available in parallel.

Comparison The incoming tag (**{VPN[2], VPN[1]}**) is composed of two different tags (**VPN[2]** and **VPN[1]**). These two sub-tags are compared with the respective of each MMUC entry. If only *VPN[2]* hits, the hit is valid if the entry is valid, and the hit is considered *partial*: in this case, only the first page table level will be bypassed. The value of the **PARTIAL** bit is not important.

If both **VPN[2]** and *VPN[1]* hit, the hit is considered valid only if the entry is valid and the **PARTIAL** bit is cleared. This hit is *full*, and two levels of the page table are bypassed. If the **PARTIAL** bit is asserted, the hit is *partial*.
If the hit is *full*, *PPN_0* is returned; if it is *partial*, *PPN_1* is returned.
Obviously, in the case in which only *VPN[1]* hits, this is **NOT** considered as a hit.

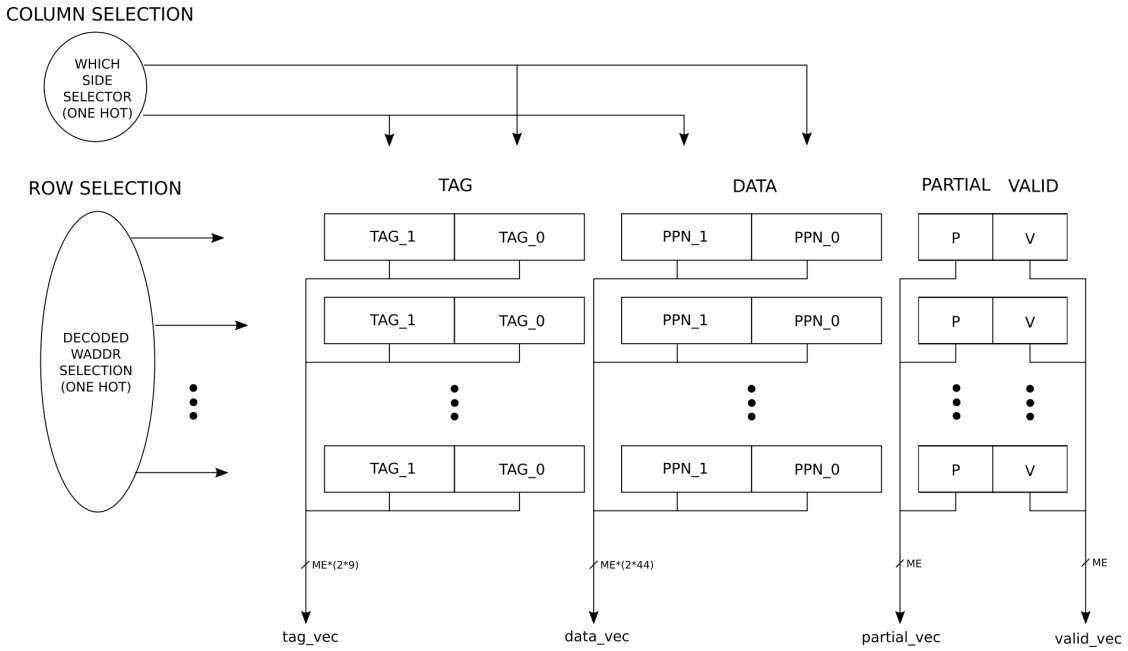

Figure 5.34. Translation Path Cache physical organization. The actual one is a register file, but this is quite unrealistic. Maybe, a successive step will replace it with a CAM.

The Updating Process The updating process can involve an already stored entry or a new one. The second case is the one handled in FIFO order, using the shift register. The first happens when a trace is partially hit, and the PTW, in the end, extracts the address of a *KibiPage* from the Page Table. In this case, the second part of the entry is substituted and the partial bit is cleared.

The address is selected by a mux driven by the sampled **hit** signal relative to the current request of the PTW.

5.8.4 Further Improvements

Use a CAM To date, the MMUC is composed of registers. This can be quite expensive and not unrealistic for certain numbers of entries. Maybe a **CAM** could fit better.

Add ASIDs Up to now, the MMUC should be flushed when the Address Space is changed. This brings to poor performances. It would be a good idea to implement here the policies adopted with the TLBs:

- Add an ASID to each MMUC entry and use it at comparison time.
- Implement also an ASID specific/Page-specific flush. [13].

5.9 L2C Arbiter

This block has a double function:

- Arbiter the incoming requests to the L2C.
- Route the L2C answer to the correct block.

Looking at figure 5.35, it is possible to note that the requests interact with the following blocks:

- **Request Scheduler**: select the block with the highest priority
- **Request Arbiter**: route the ready signal to the correct block
- **Request data selection block**: select the correct data for L2C in case of ties

Request side In case of ties the request priority is handled with a simple and almost-fair policy. Two of three requests are handled together, and the resulting request is handled with the last one. Each couple of requests is treated using a Round Robin policy: in case of ties, the priority is inverted respect to the last successful transaction. Three blocks compete to have L2C attention:

[13] It will no more possible to use a FIFO replacement implemented with a one-hot shift register because the selective flush can create holes in the structure.

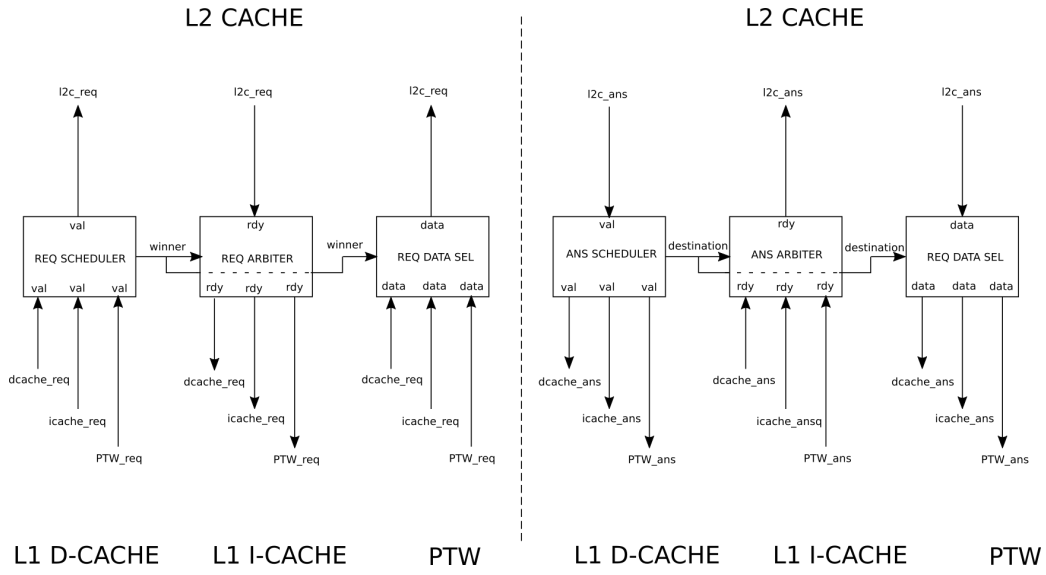

Figure 5.35. L2 Cache Arbiter. It routes request channels from L1 i-Cache, L1 d-Cache, PTW to the L2 Cache and the answer channel from the L2 Cache to the correct block.

- i-Cache
- d-Cache
- PTW

L2 TLB misses are (hopefully) less probable than L1 cache misses, but they can affect a lot of requests (a TLB entry is linked to a lot of different addresses because it refers to an entire 4 KiB page). Therefore, a little advantage is given to the PTW over the caches. This implies that the D-Cache and the I-Cache directly compete with each other, and the winner competes with the PTW.

The scheduler takes all the valid signals from the Caches and the PTW and chooses which will talk next, following the exposed above strategy. It also sends the valid bit to the L2C. When at least a winner is chosen, the arbiter routes the ready signal to the right block. Then, the data selection block selects the data to be passed to the L2C.

Answer side When L2C answers, it answers a to a precise receiver. The answer is tagged, and this tag redirects the valid signal for the answer transaction. This redirection is performed by an arbiter. The data is simply connected to all the blocks The valid signal is a one-hot signal, redirected by the source using the answer tag; the ready signal is routed on consequently.

5.10 L2 Cache

Due to lack of time, this block was only described behaviorally to allow a testbench 6.1.3.

5.10.1 Differences with the d-Cache

Even though the L2C was not designed, some notes can be shared:

- In case of a miss, the d-Cache should wait for an L2 answer. When it comes, a wake-up signal is forwarded in parallel to the LSQ. This is because the LSQ needs different words and not the entire line. If the entire line was forwarded and the single doublewords dispatched, a lot of resources would go wasted, especially for large lines. For the L2C it is different: the entire line can be forwarded to the d-Cache.
- L2C does not need a mechanism for a memory update (useful only in a multiprocessor system).
- L2 cache should speak to more than one block, so the answers are fired all from d1. When DRAM returns a line, the line is injected into the memory and only then the lower-level memory "is woken up". The d-Cache answers only to the LSQ, and the wake-up signal path is completely split from the normal answer one. This is because the LSQ wake-up, in the d-Cache, is given from d0 and the answer is given from d1. The lines which are brought back to L2 can be directly forwarded to the lower level, but this would imply possible structural hazards and stalls because lines can be returned from both d0 and d1 of L2.

In L1 there is no need for a parallel multi wake-up system because each line corresponds to only one entry. Only WBB entries could be woken-up to repeat the request because the lines returned from the DRAM to L2C always pass through all the L2C (it's like they reach the d-Cache the from the d1 stage of the L2C). Therefore, the MSHR requests are always satisfied during a return. On the contrary, WBB requests involve an L2C store.

Thus, all the returned lines should pass through d1 and then be forwarded to the lower level (the return line of L2 is unique).

5.11 Security issues

In 2018 and 2019 two fundamental papers regarding architectural vulnerabilities were published [18] [21]. *Meltdown* and *Spectre* are two different types of attacks that can cause leaks of information in particular systems. For this thesis work, the attention will be put on the first one, because the second attack is deeply linked with the concept of *speculation*, and is very difficult to avoid without hurting performance.

A complete analysis of these types of attacks is out of the scope of this work, but in the following, a brief introduction will be given to allow a more aware design.

Both attacks are based on the possibility to exploit invisible microarchitectural state changes to seize secret information. In literature, there are a lot of examples of techniques that can be used and the aforementioned two main articles highlight the main ones.

Meltdown Meltdown exploits the Out of Order execution of the processor and the possibility to load data from the memory even if the processor has not the necessary privilege to do so. The loophole is that the possible exceptions raised by the memory protection mechanism are handled only when the instruction that caused it commits. In the meantime, if the corresponding page has been accessed and the data returned, the processor can use that data for other load operations, because they can be executed without waiting for the commit of the preceding instruction.

If a secret data is accessed, and that data is used as an address to load another data in the cache, it is possible to perform a side-channel timing attack like "*Flush+Reload [63], which detects whether a specific memory location is cached, to make this microarchitectural state visible*" [18].

The main strategy adopted in **LEN5** is not to allow the injection of inaccessible data in the execution pipeline if an exception is raised. Indeed, if the d-TLB returns an exception to the LSQ, the LSQ does not save anything but the exception in that entry. This way it's not possible to accidentally forward physical addresses to other load instructions different from the first. In any case, neither a cache request or forwarding is performed if an LSQ entry is marked with an exception.

Chapter 6

Verification and Synthesis

6.1 Verification

6.1.1 Testbench organization

The testbench is executed starting from the top-level of the hierarchy and going down because the DRAM should execute simple instructions like loading/storing in a blocking fashion. This is easy to be emulated. Then, the hierarchy can be unrolled and each block can be tested singularly. Once the DRAM is working, the L2 Cache can be emulated too. Then the arbiter, the PTW, and the TLB hierarchy.

The L2-Cache can be emulated in a fully behavioral way. The point is that a request to the L2-Cache can be one among the following:

- Line read
- Line write
- Doubleword read

The answer to these request can be:

- A line
- A WBB wake-up request
- An acknowledge to the WBB
- A doubleword

The only hidden operation to be emulated is a line read. When a line is read, a doubleword can be extracted, and the acknowledge/wake-up requests can be generated without additional support. It's not important to emulate also the writing of data to the memory, because no data should be returned to the memory hierarchy except for an acknowledge. If the reading process is correct, the modification of the content of the memory can't introduce errors.

Conceptually, from an L2 Arbiter point of view, the L2 Cache is a block that can be either ready or stalled. When it's ready, it can accept requests and in parallel producing answers after a variable amount of time, because the request can hit or miss.

If no deadlocks occur, the entire system can be tested using random delays to produce the answers. This will not emulate the real timing but will work for behavior-related verification purposes.

6.1.2 Memory emulation

The memory is emulated with a SystemVerilog class and a file. The file contains a fixed number of lines, and each line is composed of two fields:

- Hexadecimal address
- Binary DoubleWord (64 bit)

This allows having a mapping of distant portions of the memory without the need for instantiating also the parts which are not needed. The simulation of a memory, mapped with a 64-bit address, would require 2^{64} entries and this is not reasonable (even if each entry occupied 1 bit, 16 Ei bit would be required). On the other hand, it is important to simulate different addresses to see if the system works correctly, especially because addresses are often managed and converted, and different portions have different meanings in each different block (e.g. tag, index, line offset, etc). For these reasons the file contains only selected addresses and each request should contain an address mapped in the file.

The memory object is meant to be instantiated as a static variable during the simulation, to represent a physical memory. When the memory object is created, the path of the memory file should be passed as the only argument to the class object constructor. The class makes available only one method, which is a task. It reads the memory file during the same cycle in which is called and updates the object property "line", which is the line read from the file.

6.1.3 L2 Cache Emulator

The L2 Cache Emulator can provide the following functionality:

- An interface compliant with the Valid/Ready handshake protocol
- Out of order completion of the requests
- Variable latency for each completion

Internally, the emulator is composed of a buffer with BUF_LEN entries. Each entry can keep an incoming request. A request is saved into the first free entry when the request is valid and the buffer is not stalled and if there is at least a free entry. The request can also replace an occupied entry which is going to be freed within the same cycle. The buffer is stalled if a stalling_event occurs or if an answer to the L2 Arbiter is valid but the L2 Arbiter is not ready. The stalling events happen randomly; if the system can work with unpredictable stalls, it can work also with deterministic stalls (this is true if the system is

deadlock-free and no stall can be forever). When a request is saved into a buffer entry, a latency and a tag are randomly assigned to it. The tag is either a "HIT" or a "MISS", and its latency can be either H_DELAY or M_DELAY, to emulate a hit or a miss. M_DELAY > H_DELAY, and H_DELAY = BUF_LEN - 1. In this way, the delay represents how many pipeline stages divide the request from its processing. Each cycle, if the buffer is not stalled, all the latencies of the valid entries are decremented by 1. When the latency of an entry reaches '0', the corresponding entry will be processed during the next cycle if the memory is not stalled. In the processing cycle, the request is fulfilled, and in the same cycle, the valid answer is put on the answer channel. The answer will wait until the L2 Arbiter will be ready to accept the answer. If more latencies related to valid requests are '0' within the same cycle, the net effect is a stall for the decrementing and the incoming requests. When one of the requests of the tie can be processed, the stall is resolved.

The processing is different for each request. If a line should be read, a line read memory request is done to the memory object and the correct line is returned. If a Doubleword should be read, it is extracted from the line after a line read memory request. If the request is a write, the answer is a "hit" if the request hit and a "wake-up" if the request missed.

6.1.4 L2 Arbiter Testbench

The arbiter was tested with single not-overlapped requests, with single overlapped requests, with multiple equal overlapped requests, and with multiple different overlapped requests. Each block among D-Cache, I-Cache and PTW is emulated with a Moore FSM which can begin a request/answer process. Each FSM can be triggered with a specific signal. The order in which these signals are asserted determines the overlap of the requests. The content of each request is controlled in separate blocks and can be modulated to obtain single or multiple requests.

6.1.5 i-Cache Testbench

The i-Cache was tested with the Virtual Memory off. In this way, the I-Cache was independent of the i-TLB. First of all, a single request was performed. Then, multiple requests, with at least five of them requesting lines of the same set, which has a capacity of four lines.

6.1.6 d-Cache Testbench

A special script was developed to aid during the testbench of the cache. This script was meant to generate three files:

- memory.txt
- paddr.txt
- info.txt

The first file is the already mentioned memory file containing the couples "doubleword address - doubleword". It is used by the L2 Cache emulator to extract and return the line

to the L2 Arbiter when a request is ready to be processed. All the addresses are written to the "paddr.txt" file. This is used as a source for the memory requests during the cache testing. The last file contains the addresses coupled with the number of the set in which they are supposed to be put.

Initially, the name of the script was "gen.py", but during the first running it generated a lot of strange results in the "info.txt" file: the problem was the using of the division operator (/) instead of the floor division one (//). For this reason, the name of the script is now "gene.py".

The d-Cache testbench was not aimed to be complete, because it would have requested a deeper emulation of the system from both the sides of the hierarchy: the memory emulator and the request side (the LSQ). The testbench was written to understand if all the signals weren't floating and if minimal support to the operations was assured. For this reason, a series of requests was performed and the Cache content, at the end of the simulation, was checked to see if the correct lines were placed in the correct positions (set and physical memory) and if the replacement vector and the requests/answers were coherent. Also, the flush unit was tested.

The testbench read multiple times the "paddr.txt" file, feeding the d-Cache with continuous read requests, then write requests. It was checked that during the reading cycle all the requests missed, and, during the writing one, all the requests hit.

A lot of imprecisions and issues were found and corrected this way.

6.1.7 Virtual Memory Testbench

The part related to Virtual Memory was tested altogether. This choice is linked to the fact that it's easy to follow the path of a request through the structures.

A SystemVerilog testbench was prepared to host and test:

- **The d-TLB**
- **The L2 TLB**
- **The PTW**
- **The MMU Cache**

Then, also the **i-TLB** and the **TLB Arbiter** were tested too.

To test the Virtual Memory system, another Python script was developed to write into the memory file a three-level page table and the data. The script generates a list of virtual addresses to be converted by the hardware and a root page table PPN. These elements are read by the **d-TLB** which initially misses, like the **L2 TLB** and the **MMU Cache**.

The first environment was tested using 10 Virtual Addresses, read for two times. All the PTE in the page table were created not to raise exceptions. The behavior was the expected one (after a few corrections).

6.2 Synthesis

The presence of memories within the design requires special care to avoid problems during the synthesis. Usually, there are special "Memory Compilers" that can synthesize memories, because normal compilers infer only registers.

For this reason, the design was synthesized removing all the memories from the description, leaving floating signals at the interfaces. Obviously, this estimation of *area* and *critical path* is only partial. Luckily, it's possible to estimate what is missing with **CACTI**. Citing what is written on the main page of its GitHub repository, **CACTI** "*is an analytical tool that takes a set of cache/memory parameters as input and calculates its access time, power, cycle time, and area.*" [30]. Its effectiveness and reliability are proved also in [6].

6.2.1 Architecture part

First of all, the physical memories of the d-Cache, i-Cache and L2 TLB were removed from the hardware description. All the signals to/from the memories were brought to the interfaces and left floating, not to let the synthesizer simplify them.

A partial synthesis was done for each block, together with a full one for the whole described Memory System. The results can be credible because the logic before the inputs and after the outputs, in both the front end and the back end, is very limited. This consideration, however, suggests looking at these results as a best-case scenario, or as an upper limit.

The synthesis was performed using Synopsys Design Compiler with the low leakage library **UMC 65 nm** in the "typical conditions". Multiple instances are treated using the *uniquify* method, and a clock is defined with an uncertainty of *0.07 ns*.
The capacitive loads at the outputs were set to buffers with *drive strength index* of 4 called **BUFM4R** ([31]). To let the compiler does its best, the requested clock period to be reached was set to 0 ns. The synthesis was performed using the *compile* command.

Area

All the blocks and the complete design were synthesized. The *area reports* shown the data reported in tables 6.1 and 6.2.

	C Area (μm^2)	**NC Area** (μm^2)	**Tot Cell Area** (μm^2)
d-Cache	33138	34927	68065
i-TLB	7371	9500	16871
MMUC	5154	11140	16295
d-TLB	6770	8438	15209
L2 TLB	6845	3109	9955
i-Cache	7108	1319	8427
PTW	1479	1623	3103
TLB Arbiter	190	19	210
L2C Arbiter	431	24	455

Table 6.1. Partial synthesis results. "NC": NonCombinational. "C": Combinational.

	C Area (μm^2)	NC Area (μm^2)	Tot Cell Area (μm^2)
Total from partials	68486	70099	138585
Complete design	49222	51454	100676

Table 6.2. Partial and total synthesis results. "NC": NonCombinational. "C": Combinational.

As shown, the complete synthesis can optimize the area.

d-Cache The synthesis of the block was performed with the following parameters:

1. MSHR entries: 4
2. WBB entries: 4
3. Line size: 64 bytes (512 bits)

The d-Cache is the largest block in the design. Its area is mainly dependent on the line size, which influences the WBB and the **d0 -> d1** registers. In the table 6.3 is reported its main structures.

d-Cache	Main contributions	Bits
WBB	N_entries * (Line + Addr)	2300
FIFO replacement	N_set * N_way	1024
d0 ->d1 regs	Line + Addr + Data+ others	651
MSHR	M_entries * (Addr)	240
Replay Register	Addr + Data + others	135

Table 6.3. Main structures of the d-Cache block

The primary cause of area occupation seems to be related to the WBB because it has four entries end each one contains a line and a line address. The line address depends on the cache parameters, but it is around 40 bits. The line itself is 512 bit. Both the MSHR and the WBB have parallel comparators on the line addresses, thus they influence also the area linked to the combinatorial logic.

To show the impact of the number of entries of the WBB and of the MSHR on the area, different syntheses were performed varying these parameters. The results are shown in 6.4.

As already hypothesized, the WBB number of entries can impact on the area of the block. However, the area could be not the first reason to reduce its number of entries: the energy cost for the comparisons is paid every time a data reach **d1**. Therefore, it is likely that the power saving obtained reducing the WBB can be not negligible.

The replay registers don't seem to occupy a lot of area. If in timing and power terms are not problematic, they can be kept. The area overhead of the associated control is hypothesized negligible.

d-Cache	N_MSHR	N_WBB	C A (μm^2)	NC A (μm^2)	Tot A (μm^2)
Synth_0	4	4	33138	34927	68065
Synth_1	4	2	26073	23783	49857
Synth_2	4	1	21906	19654	41560
Synth_3	2	4	29107	37186	66293
Synth_5	2	2	24413	23565	47978
Synth_6	2	1	20959	18162	39121
Synth_7	1	1	20119	17924	38043

Table 6.4. How the d-Cache area varies in function of the entries of the WBB and the MSHR.

	Cache bits	Other bits
i-TLB	840	15
MMUC	1080	21
d-TLB	696	65

Table 6.5. Size in bits of the architectural blocks with storing function, realized using registers.

i-TLB, MMUC, d-TLB The size of these structures is determined by their storing function. In table 6.5 the sizes (in bits) of the non-combinational parts of these blocks are reported.

The MMUC stores more information than the TLBs (and this is proved by the non-combinational area occupation in 6.1), but it should do less parallel operations because the TLBs are fully associative with at least 8 entries.

L2 TLB From the "elaboration" log is possible to know how many bits of sequential logic are instantiated: 626, which is a value similar to the one of the d-TLB. Maybe there can be a problem with the L2 TLB replacement block because one-third of the flip flops are simplified by the compiler. A dedicated testbench will be necessary to understand why. In this hypothesis, the real estimated non-combinational area of the L2 TLB would be almost 70% bigger than now. Since this simplification affects also the combinational area, another synthesis will be performed after the correction of the eventual errors.

i-Cache The i-Cache has a lot of area dedicated to the logic because it is a 4 way set associative caches (tag comparison) and the blocks are of 512 bits (output mux).

PTW The PTW works on doublewords, and has only a few registers with a simple control unit.

Arbiters They are quite small and negligible in terms of area.

Timing

The partial timing results are reported in the table 6.6, the final ones in table 6.7. The synthesis was performed adding a *clock uncertainty* of 0.07 ns. In the timing reports generated by *Design Compiler*, this result is added to the **Data Arrival Time**. For coherence with the synthesis of the Back End ([33]), all the timing results are the **Data Arrival Times**, therefore the *clock uncertainty* is neglected.

	Slack [ns]	**Start Point**	**End Point**
L2 TLB	1.32	t0_t1_req_q.vpn[2][2]	L2tlb_repl_unit/q0_reg[117]
d-Cache	1.14	d0-d1_req_q.req_type[2]	dcache_wdata_o_[56]
MMUC	1.00	tpc/tag_vec_o[3][1][5]	mmuc_ptw_ans_o.ppn[43]
i-TLB	0.88	entry_vec_q[3].vpn[0][8]	ic_aresp_o.paddr.tag[1]
d-TLB	0.75	dtlb_entry_q[3].vpn[1][6]	dtlb_lsq_ans_o.ppn[11]
i-Cache	0.54	mem_out_i.tag_vec[2][18]	iresp_o.line[149]
PTW	0.40	ptw_cu_i/state_q_reg[0]	pte_or_ppn_q.ppn[14]
L2C Arb	0.31	cache_ptw_priority_reg_q	l2arb_l2c_req_o.paddr.idx[0]
TLB Arb	0.25	tie_winner_reg[1]	l1tlb_l2tlb_req_o.vpn[1][5]

Table 6.6. Partial timing results for the architectural part of the Memory System. The reported "Slack" is the one of the *timing_report*, without *0.07 ns* of clock uncertainty.

	Slack [ns]	**Start Point**	**End Point**
Design (NO mem)	1.53	ic/vaddr_q.tag[18]	ic_fend_ans_o.line[0]

Table 6.7. Final timing results for the architectural part of the Memory System. The reported "Slack" is the one of the *timing_report*, without *0.07 ns* of clock uncertainty.

6.2.2 Memory part

As said, the memory-related information about the area and the timing were only estimated using **CACTI**. This tool can be configured setting the memory parameters using a configuration file, and the "technology size" was set to 65 nm. As it was hypothesized to have single blocks of *SSRAM* with parallel available output, the memories with associativity N_Way greater than 1 were ideally decomposed in blocks of N_Way direct-mapped caches with tag. The area estimation is likely to be this minimum area multiplied N_Way. In this way, the hope was that the estimated parameters were relative to one block only without set comparison logic. These results were also checked estimating the total area of memories with the correct size and associativity.

For example, for:

```
Cache size: 32 KiB
Associativity: 2
```

CACTI was configured both for these real parameters and for:

```
Cache size: 16 KiB
Associativity: 1
```

The results of the first estimation (data array + tag array) were then compared with double the results of the last estimation and they were very similar. It's not clear if a comparison logic is taken into account, but this result can be considered as an upper limit because in the architectural part of the Memory System the comparison logic is already present. They are shown in 6.8.

Area

	Direct Mapped Area (μm^2)	**Associativity**	**Tot Area (μm^2)**
d-Cache	365000	2	730000
i-Cache	210000	4	840000
L2 TLB	150000	4	600000

Table 6.8. Area of the memories of the LEN5 memory system, estimated with CACTI.

The estimated areas show the how many resources can be occupied by the memory blocks. The whole architectural part consumes about five times less area. Therefore, all the considerations about the WBB should be weighted, because the impact of its hypothetical area reduction can be very limited. However, the main goal of such an action can be the reduction of the power consumption or performance improvement.

It is worth to highlight the area occupation of the memory relative to the i-Cache: it is half the size of the d-Cache, but the "area efficiency (Memory cell area/Total area)" is around the 20% for the first and 50% for the second one.

Timing

For what concerns the timing, only the results of the direct-mapped caches are taken into considerations. This first estimation of cycle time is calculated as all the blocks of an associative memory are accessed in parallel, because the comparison logic was already counted in the architectural design. A table showing the results is reported in 6.9. The Access Time is the time passed from the request to the results, the Cycle Time is the minimum time between two consecutive requests.

	Access Time [ns]	**Cycle Time [ns]**	**Max ([ns]**
d-Cache	0.607979	0.630455	0.630455
i-Cache	0.459273	0.528493	0.528493
L2 TLB	0.535191	0.454388	0.535191

Table 6.9. Cycle and Access time of the Direct Mapped blocks of the LEN5 Memory System.

The worst time is the Cycle Time of the d-Cache: almost 630.5 *ps*.

6.2.3 Overall Design

Area

The result obtained for the area is easy to be combined: the two partial results are added together (6.10).

	Arch. Part [mm^2]	**Mem. Part** [mm^2]	**Tot Mem. System** [mm^2]
Area	0.1	2.17	2.27

Table 6.10. Final estimated area of the Memory System.

The architectural part accounts only for the 4.4% of the total Area. Maybe, all its future optimizations will target the timing and the power.

Timing

The overall architectural critical path is relative to the address translation of the **TAG** part of the Virtual Address, for the i-Cache comparison. It starts from the register in which the Virtual Address is saved (*vaddr reg*, in the i-Cache). Then, it is "filtered" by the bit of the virtual memory status, and it is directly compared with the i-TLB **VPN**s, used as tags. The memory is *Virtualy indexed, Physically tagged*, and the memory is accessed in parallel with the i-TLB. Therefore, the *access time* of the memory is overlapped with the delay of accessing the i-TLB.
In a future design, the i-TLB and the i-Cache will be decoupled and accessed together, to reduce the possibility of having the i-TLB on the critical path. For this reason, the *vaddr register* will be moved to break the path of the i-TLB: this would be useful because now it's likely the i-TLB to be the limiting block.
The partial result of the critical path of the i-TLB shows that from one of its VPN entry

to the answer to the i-Cache, the delay is 0.88 *ns*. The *vaddr* just enters in the i-TLB, is filtered, and then goes directly to the comparison logic, combined with the *VPN* entries of the i-TLB. Therefore, even if the complete synthesis can present a different logic organization, it's likely that the critical path of the i-TLB is less than the time spent by *vaddr* in the i-TLB. Thus, it's probable that the access time of the i-TLB is greater than 0.88 *ns*. Since the memory access time is less than this value, the i-TLB is on the critical path. This also means that the access time of the i-Cache is hidden by the i-TLB access time.

Therefore, the critical path can become the one relative to the L2 TLB. The *t0_t1_reg_q.vpn* signal is compared with the *TAG* of the L2 TLB, to create the *hit vector* to feed also the *Replacement Block* (the end of the partial critical path).
The critical path of the L2 TLB is 1.32 *ns*. At the active edge of the clock, both the memory and the register are accessed. These two accesses are done in parallel; therefore, when the *t0_t1_reg_q.vpn* signal reaches the first logic gate after a *clock to output* time, part of the memory access has already happened. The *timing report* shows that the time related to the register access is 0.12 *ns*. This time can be subtracted from the partial slack of the L2 TLB. Then, the memory access time can be added. The same reasoning can be done for the d-Cache, but the worst case is obtained by considering the L2 TLB. The results on the total critical path are presented in table 6.11.

	Time [ns]
L2 TLB Critical Path	1.32
Memory Access Time	0.54
Overlap	0.12
Estimated Slack	1.74

Table 6.11. Estimation of the total critical path of the Memory System.

Chapter 7

Final remarks

7.1 Results

The Memory System for LEN5, an open-source RISC-V Out of Order processor under development, has been presented and discussed. It is designed to support an OS and be compatible with the other parts of LEN5: its front end [32] and back end [33]. It is parameterized to support studies on its optimization and give valid answers to different needings. It's modular and flexible thanks to the handshake protocol used among the various blocks, and this fact makes the life easier to who will try to modify it. Groups of blocks were tested simulating probable conditions and the entire system synthesized. Even though the timings and area results are only preliminary, they are a good point to start for the next steps of the project.

Under the hypotheses mentioned in the previous chapter, the final *area* and *timing* results are presented in table 7.1.

	Area $[mm^2]$	**Max Frequency [MHz]**
LEN5 Memory System	2.27	574

Table 7.1. Final area and timing estimation for the LEN5 memory system.

7.2 Next Steps

Some possible improvements have already been discussed in the section relative to each block. In the following, it is presented a possible roadmap with the most important steps to be taken in the immediate future.

Merge the testbenches and massively simulate the system

The *testbenches* used to simulate the different parts of the design (Physical Memory and Virtual Memory) can be grouped and the entire system must be simulated without using

only the waveforms. The first method is useful during the first phase of the verification process, but now it's time to use a standard method to be used every time a modification is performed, not to introduce new bugs.

Use real memories

The hypothesis of implementing all the memories using a simple SSRAM has allowed being focused on the entire system, but now that the backbone is ready, studies on real memories are necessary. Maybe it is more convenient to bring some parts of the tag comparison "inside" the memories or directly use CAMs. It would be ideal to develop a modular system in which a possible parameter is the type of memory itself, to understand the real benefits and drawbacks of each solution.

Another important hypothesis to be evaluated is the bus width on which all the lines are moved. Likely, working with busses whose size is smaller than the cache lines could be necessary; especially when the data should be transferred off-chip, maybe to the main memory. The AXI protocol, which supports bursts, seems to be attractive for this purpose.

Add pipeline levels to the Caches

The estimations on the timing suggest that another register could fit well after each memory. This would allow breaking the path of the memory output, which will be likely critical. This modification is not devastating, especially for the d-Cache and the L2 TLB, because all the controls can percolate through the registers. The "Wake Up" system can be also used for handling the cache stalls: since the cache cannot stall, every time a request is blocked immediately after the memory block, it can be replayed when the stall is resolved. But the "sleep-wakeup" design allows also transforming that request in a "wake up" answer and let the other levels replay it without additional control. If this is the case, some MSHR could need some little modifications.

Valid and Dirty bits outside the Caches

The overhead introduced for resetting and *cleaning* the cache, or when an upper block should be synchronized with the lower, can be large. For this reason, extracting some bits (like the "valid" and the "dirty") from the memory and storing them in registers can be a good idea. This would simplify the *Reset Block* and the *Updating L2 Block*, for example, and all the design would become more readable and maintainable; also, it would be, hopefully, more performant and efficient.

Implement the L2 Cache

During this first phase, the L2 Cache and the Main Memory were only emulated. The Memory System cannot be complete without real support for them.

Other cache improvements

The parameters of the caches can be initially taken from real Out of Order processors, instead of using the ARM Cortex A53, which is not Out of Order. This would let the project have a more realistic flavor and hopefully a greater overall efficiency or at least performance level, even if a deep study on the domain of application is essential to obtain a real enhancement.

If the bus size is reduced, a lot of techniques can be studied and applied to leverage this fact (for further information, read [6]).

Merge the three parts and test real situations

Merging the Memory System with the Processor would let the verification process easier. With the memory system alone it's difficult to think and test everything to understand if all the real conditions have been verified.

Power Analysis

Having a real processor and a testbench with a probable load allows for useful power analyses. This analysis is very important because a fast and small processor is more useful if not burned. And in general, the energy considerations influence considerably the design.

Improve the code

The code was written without using *don't cares* and all the SystemVerilog "X" were avoided when possible, to have more readable waveforms during the verification step. This choice prevents the synthesizer from performing important optimizations. As soon as a powerful and modular way of testing is used and the design works, it can be optimized for the synthesis.

Chapter 8

Appendices

Glossary

Address Space A logical range of addresses that are available to a program. 24, 38, 48, 50

Back End The part of the processor in which the execution and the completion of the instruction take place. 22

block A block is a group of words. Its length is fixed. It is the minimum quantity that can be stored into a cache. It is common to call the block as "line" because they should have the same length. To be more precise, a block in contained into a cache line. In this work, the two words have the same meaning, and the word "line" is preferred. 15

commit The operation that makes definitive the outcome of an executed instruction. 37

computer system An electronic system capable of receiving some input, elaborating information, and output the results. 5

critical path The path between two registers on which the signals accumulate the higher delay from the first to the second register. 9

Demand Paging The path between two registers on which the signals accumulate the higher delay from the first to the second register. 24

homonym Two physical addresses that are mapped by the same virtual address are homonyms. 38

index The index is the portion of the cache address that selects a particular set. 37

Index length The length of the index part of a cache address expressed in bits. The index addresses a single set of the cache. 64, 80

In Order Given a flow of instructions, an operation or a set of actions are defined as "In Order" if they are performed on an instruction only if they have been already performed on the previous ones. A processor is defined "In Order" if all its operations are "In Order". 17, 20

Leaf Page A page of a page table that contains the physical address of the requested page. 50

leaf page table entry A page table entry pointing to a physical page. 31, 117

LEN5 LEN5 (pronounced *lens*) is the name of our core processor, that will be open and available on GitHub. LEN5 is a polysemic word and even an acronym. The 5 is a link with the RISC-V ISA, on which the core is based. *Lens* can mean *perspective* and in Italian has the same meaning of the word *objective*: it means *goal*. For what concern the author of this thesis, the most important part of the name is enclosed into the **E**: and it wouldn't be fair to hide it: his real and final objective. 2, 3, 12, 13, 44, 45, 53, 141

line A cache is divided into one or more sets, and each set is divided into lines. The line is the minimum quantity transferred to and fro the cache. The word "line" is used also to indicate the content of the line itself, which is more correctly a "block". 15

Line offset length The length of the line offset part of a cache address expressed in bits. The line offset addresses a single word of a line. The term "word" is generic and does not refer only to a 32-bit word. 64, 80

Main Memory The memory between the caches and the secondary storage in a memory hierarchy. It is the fundamental part of the memory system if there is no memory hierarchy at all. Usually, it is a DRAM. 13, 14, 15

Many Core A device that is controlled by many processors of different type. 22

MMU cache A cache used by the PTW to store the mappings between the first level of the page table and the intermediate ones. It is useful when a radix-tree page table is used. 48

Multi Core A device that is controlled by many processors of the same type. 22

N_Set The number of sets of a cache. 80

N_Way The number of lines into a cache set. 66, 69, 80, 138, 149

Out of Order Given a flow of instructions, an operation or a set of actions is defined as "Out of Order" if it can be performed on an instruction even if it was/were not performed on the previous ones. A processor is defined "Out of Order" if at least one part of its operations are "Out of Order". 18, 22, 53, 141, 143

Page The path between two registers on which the signals accumulate the higher delay from the first to the second register. 24, 25

page table walk The page table walk is the process to handle a TLB miss when occurs at the top of the TLB hierarchy. The page table is accessed until a leaf PTE is found and the searched information retrieved. It can be performed in software by the OS or in hardware with a PTW. 31

page table entry An entry of the page table. 25, 148, 151

page fault An event that happens if the requested physical page is not in the Main Memory because is in the second storage. 25

page table A logical structure of the virtual memory. It contains the mapping between a virtual address and the corresponding physical address. 24, 25, 48

physical page number The address of the beginning of a page. 25, 151

physical address The real Main Memory data address. 24

pseudo-code A logical representation of a program, which does not correspond to any real language. 9, 10

Replacement Vector A "one hot" vector with N_Way bits. The asserted bit indicates the position in the set in which the new line will be written. 84

set A cache set is a group of cache lines that share the same address index for that precise cache. When a cache is addressed, it is first indexed. The indexing process returns a set of the cache, in which are present one or more lines. Then, the tag portion of the requested address is compared to the tag portion of each line of the set to understand if the requested line is present or not in the cache. 16

speculative execution A technique used by the processor to reduce the execution time. Some instructions are executed even if it's not sure they should be, and this decision is based on a prediction. Later in time, that prediction can be confirmed as right or wrong. For example, in the case of a branch prediction, the processor can speculate on the outcome of a branch. If it is right, the penalty of a possible stall due to a control hazard is avoided. Otherwise, the initial condition should be restored and the mispredicted branch is paid in terms of energy and timing performance. 37

Structural Hazard Each resource has a limit on the number of contemporary access it can allow. A structural hazard is a condition in which a physical resource, like a memory, is accessed by more devices than its limit would allow. 21

synonym Two virtual addresses are synonyms if they refer to the same physical address. 38

tag The TAG is the portion of the address that uniquely identifies a line. The TAG bits are the most significant ones. The TAG part of the address is saved along with the line into a Cache because when the Cache is addressed, the TAG portion of the incoming address is compared to all the TAGs of the set in which the line can be present. If the comparison is positive, a line hit occurs, otherwise the request misses. 16, 17

TAG A TAG is a set of bits that uniquely identifies an object. In cache terminology, it identifies a line. 37

TAG length The length of the tag part of a cache address expressed in bit. 64, 80

untrusted code A code that is not trusted by the system. By the system perspective, the untrusted code to be dangerous. 44, 45

virtual page number It contains the information to address the page table entry. 25, 152

virtual address A logical address. It contains information to address the page table entry and the offset for the data into the final physical page. 24, 25, 73

word The natural packet of bits on which the processor can work. 6, 16

Word offset length The length of the word offset of a cache address expressed in bits. The word offset addresses a single byte of a word. The term "word" is generic and does not refer only to a 32-bit word. 64, 80

Write Back A cache implements a Write Back policy if when one of its lines is modified, a dirty bit for that line is asserted. Dirty bits identify modified lines that have not been backed up to the upper level of the Memory Hierarchy. The line is written back to the higher level only when it is evicted from the cache, to avoid data losses. 20

Write Through A cache is Write Through if, after one of its lines has been written, writes that line also to the higher level of the Memory Hierarchy. 20

AMAT Average Memory Access Time

AMO Atomic Memory Operation

ASID Address Space IDentifier

ASM Algorithmic State Machine

BE Byte Enable

CAM Content Addressable Memory

CDB Common Data Bus

CISC Complex Instruction Set Computer

CPI Cycles Per Instruction

CPU Central Processing Unit

CS Chip Select

CSR Control and Status Register

CU Control Unit

d-Cache Data Cache

d-TLB Data Translation Lookaside Buffer

DP Data Path

DRAM Dynamic Random Access Memory

FFT Fast Fourier Transform

FIFO First In First Out

i-Cache Instruction Cache

i-TLB Instruction Translation Lookaside Buffer

ISA Instruction Set Architecture

HDL Hardware Description Language

L2C Level 2 Cache

LFU Least Frequently Used

LRU Least Recently Used

LSQ Load Store Queue

MMU Memory Management Unit

MMUC Memory Management Unit Cache

MSHR Miss Holding Status Register

NRU Not Recently Used

OS Operating System

paddr physical address

PC Program Counter

PLRU Pseudo Least Recently Used

PMA Physical Memory Attributes

PMP Physical Memory Protection

PPN Physical page number

PTE Page table entry

PTW Page Tagle Walker

RAM Random Access Memory

RAW Read After Write

RISC Reduced Instruction Set Computer

ROB ReOrder Buffer

SASOS Single Address Space Operating System

SRAM Static Random Access Memory

SSRAM Synchronous SRAM

TLB Translation Lookaside Buffer

TPC Translation Path Cache

TVD Tag Valid Dirty

VPN Virtual page number

WE Write Enable

WB Write Back

WBB Write Back victim Buffer

Bibliography

[1] Zaruba, Florian. *Ariane: An open-source 64-bit RISC-V ApplicationClass Processor and latest Improvements*, 9 May 2018. https://content.riscv.org/wp-content/uploads/2018/05/14.15-14.40-FlorianZaruba_riscv_workshop-1.pdf

[2] "About the RISC-V ISA", *RISC-V*, RISC-V Foundation, https://riscv.org/

[3] "cpu", textitMerriam-Webster, Merriam-Webster, https://www.merriam-webster.com/dictionary/cpu

[4] "CPU", *Cambridge Dictionary*, Cambridge University Press, https://dictionary.cambridge.org/it/dizionario/inglese/cpu

[5] Conte, Gianni, et al. *Architettura dei Calcolatori*. CittàStudi Edizioni, 2015.

[6] Hennessy, John L., and David A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann, an imprint of Elsevier, 2019.

[7] Patterson, David A., and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, an imprint of Elsevier, 2018.

[8] Patterson, David A., and Andrew Waterman. *The RISC-V reader: an open architecture atlas*, Strawberry Canyon LLC, 2017.

[9] Bhattacharjee, Abhishek. "*Advanced Concepts on Address Translation*", *Computer Architecture, Appendix L*, Morgan Kaufmann, an imprint of Elsevier, 2018.

[10] Harris, David Money, and Sarah L. Harris. *Digital Design and Computer Architecture (Second Edition)*, Morgan Kaufmann, an imprint of Elsevier, 2013.

[11] Jouppi, Norman P. "*Cache Write Policies and Performance*", WRL Research Reports 91/12, Western Research Laboratory, 1991.

[12] Kroft, David. "*Lockup-free instruction fetch/prefetch cache organization*", 1981.

[13] Jain, Akanksha, and Calvin Lin, "*Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement*", 2016.

[14] Farkas, Keith I., and Norman P. Jouppi, "*Complexity/Performance Tradeoffs with Non-Blocking Loads*", 1994.

[15] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

[16] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

[17] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, June 2019

[18] Moritz Lipp et al., "*Meltdown: Reading Kernel Memory from User Space*", 2018.

[19] Conte, Gianni et al., *Architettura dei Calcolatori*, CittàStudi Edizioni, 2015.
[20] *AMBA® AXI and ACE Protocol Specification - AXI3, AXI4, AXI5, ACE and ACE5*, page A3-44, ARM, 2017
[21] Kocher, Paul et al., "*Spectre Attacks: Exploiting Speculative Execution*", 2019.
[22] Silberschatz, Abraham, Peter Bael Galvin, Greg Gagne. *Operating System Concepts, ninth edition*, Wiley, 2013.
[23] *lowRISC Verilog Coding Style Guide*, https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md
[24] Barr, Thomas W., Alan L. Cox, Scott Rixner. "*Translation Caching: Skip, Don't Walk (the Page Table)*", Rice University, Houston, TX, 2010.
[25] Bhargava, Ravi, Benjamin Serebrin, Francesco Spadini, Srilatha Manne. "*Accelerating Two-Dimensional Page Walks for Virtualized Systems*". In ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pages 26–35, New York, NY, USA, 2008. ACM.
[26] Jacob, Bruce, Spencer Ng, David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, an imprint of Elsevier, 2006.
[27] Cekleov, Michel and Michel Dubois. *Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors*, IEEE Micro, vol. 17, pp. 64–71, Sept. 1997
[28] Qiu, Xiaogang, and Michel Dubois. "*The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches*", IEEE Trans. Comput., vol. 57, pp. 1585–1599, Dec. 2008.
[29] Adve, Sarita V., and Kourosh Charachorloo. "*Shared Memory Consistency Models: A Tutorial*", WRL Research Report 95/7, Western Research Laboratory, 1995.
[30] *Cacti Github Repository*, https://github.com/HewlettPackard/cacti
[31] *UMK65LSCLLMVBBR_B UMC 65nm Low-K Multi-Voltage Low Leakage RVT Tapless Standard Cell Library Databook, Ver. B03*, UMC, 2014/01/16.
[32] Marco Andorno. *Design of the frontend for LEN5, a RISC-V Out-of-Order processor*. 2019.
[33] Michele Caon. *Design of the execution pipeline for LEN5, a RISC-V Out-of-Order processor*. 2019
[34] *Applications Note: Understanding Static RAM Operation*, IBM, 1997.