Study and Evaluation of several Cache Replacement Policies on a Commercial MIPS Processor

Daniel Pinto Rivero

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Grado

June 15, 2016

Director/s and/or colaborators:

Daniel Ángel Cháver Martínez
Luis Piñuel Moreno
Enrique Sedano Algarabel

Autorización de difusión

Daniel Pinto Rivero

15 de Junio de 2016

El abajo firmante, matriculado en el Grado en Ingeniería de Computadores de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Study and Evaluation of several Cache Replacement Policies on a Commercial MIPS Processor", realizado durante el curso académico 2015-2016 bajo la dirección de Daniel Ángel Cháver Martínez y Luis Piñuel Moreno [y con la colaboración externa de dirección de Enrique Sedano Algarabel] en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Abstract

Computer performance is a hot topic today. With the technological and physical limitations in semiconductors to solve, a lot of effort is put from universities and industries trying to bring new architecture improvements to keep the Moore's law rolling.
In this project, we aim to study caches and memory hierarchy, one of the big topics in computer performance. For that, we have chosen MIPSfpga, an open hardware platform from Imagination Technologies, which allowed us to implement and test different cache replacement policies as a prove of concept and to show the possibilities of the platform.

Keywords

Moore's Law, Performance, Cache, Memory Hierarchy, Open Hardware, MIPSfpga.

Resumen

Actualmente, el rendimiento de los computadores es un tema candente. Existen importantes limitaciones físicas y tecnológicas en los semiconductores de hoy en día, por lo que se realiza un gran esfuerzo desde las universidades y la industria para garantizar la continuidad de la ley de Moore.
Este proyecto está centrado en el estudio de la cache y la jerarquía de memoria, uno de los grandes temas en la materia. Para ello, hemos escogido MIPSfpga, una plataforma hardware abierta de Imagination Technologies, lo que nos ha permitido implementar y testear diferentes políticas de reemplazamiento como prueba de concepto, demostrando, además, las bondades de la plataforma.

Palabras clave

Ley de Moore, Rendimiento, Cache, Jerarquía de memoria, Hardware abierto, MIPSfpga.

Contents

Index i

Acknowledgements iii

Dedication iv

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Miniaturization Problems . . . 2
  - 1.1.2 Thermal Wall . . . 2
  - 1.1.3 Memory Wall . . . 2
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 3

2 Memory Hierarchy 4
- 2.1 Cache Memory Organization . . . 5
- 2.2 Cache Placement Policy . . . 6
  - 2.2.1 Direct Mapping . . . 6
  - 2.2.2 Pure Associative . . . 6
  - 2.2.3 Set Associative . . . 7
- 2.3 Cache Replacement Policy . . . 7
  - 2.3.1 Least Recently Used (LRU) . . . 8
  - 2.3.2 Most Recently Used (MRU) . . . 9
  - 2.3.3 FIFO . . . 9
  - 2.3.4 Random . . . 9
  - 2.3.5 Not Recently Used (NRU) . . . 9

3 MIPSfpga 10
- 3.1 MIPS Core . . . 10
- 3.2 MIPSfpga System . . . 11
- 3.3 Memory Hierarchy . . . 13
- 3.4 Comparison with Other Open Softcore Alternatives . . . 13

4 Development of New Cache Replacement Policies for the MIPSfpga Core 15
- 4.1 The Cache Hierarchy in MIPSfpga . . . 15
- 4.2 The LRU Implementation of MIPSfpga . . . 16
  - 4.2.1 Allocation . . . 17
  - 4.2.2 Hit . . . 17

4.2.3 Invalidation . . . 17
4.3 Changes in the Cache Configuration . . . 17
4.4 Implementation of a NRU Policy . . . 19
4.5 Implementation of a FIFO Policy . . . 19

5 Implementation of Tests **20**
5.1 The MIPS Coprocessor Model . . . 20
5.2 Coprocessor 0 . . . 21
5.3 Core Configuration . . . 21
5.4 Performance Counters . . . 24
5.5 Implementation of Microbenchmarks . . . 24

6 Experiments **27**
6.1 FIFO Correctness . . . 27
6.2 Sequential Access Pattern . . . 28
6.3 Recency-friendly Access Pattern . . . 29
6.4 Mixed Access Patterns . . . 30
6.5 Array Enlargement Optimization Technique . . . 30
6.6 Array Merge Optimization Technique . . . 31
6.7 Other Tests . . . 31

7 Other Activities **32**

8 Results **33**

9 Discussion **36**
9.1 Cache Memory Study . . . 36
9.2 MIPSfpga in Academic Courses . . . 36

10 Future Work **38**
10.1 Cache Replacement Policies . . . 38
10.2 MIPSfpga Fundamentals Extension Labs . . . 38

Bibliography **40**

Acknowledgements

I would like to thank my directors, Daniel Cháver, Luis Piñuel and Enrique Sedano, for the opportunity to work in this project, and their invaluable help in its realization; and the people from Imagination Technologies, for the opportunity to be involved in MIPSfpga.

Dedication

To my friends and family, specially my mother, that respected my decisions, even when they seemed wrong, and always believed in me.

Chapter 1

Introduction

1.1 Background

Computer performance is one of the factors that push the limits of science and engineering. It is not of wonder that a lot of effort is put in this matter.

Since the invention of integrated circuits, many people started to see that performance reached by computer was going to experiment an unprecedented growth in the following years. A well known observation, made by Gordon E. Moore in 1965 [11], was that the number of transistors in a dense integrated circuit doubles each year, and predicted the continuity of that for at least a decade, what was later known as the Moore's law. Since then, a number of revisions have been made to the law. In 1975, he changed the prediction to double each two years for the close future. In 2015 Brian Kranich, CEO of Intel announced that their cadence was then closer to two and a half year than two, and Gordon Moore himself foresaw that the rate of progress would reach saturation by the next decade.

That is certainly true by now. Although many semiconductors companies have based their commercial objectives after Moore's law, some barriers have been found that compromise in a severe way that rate of growth.

Some of those technological and physical barriers are discussed below.

1.1.1 Miniaturization Problems

As the transistors have seen their size dramatically reduced, new problems have arisen, such as subthreshold conduction, caused by the low voltage required in order to maintain stability in such a small scale, interconnect capacitance, increment of defective circuits, gate-oxide and junction leakage caused by the arising electron tunneling quantum phenomenon.

1.1.2 Thermal Wall

The increment of the component density and frequency of operation also carries problems of heat dissipation, as all the semiconductors companies have experimented in the last years. That problem motivated many researchers to switch their research focus from frequency scale to multicore design.

1.1.3 Memory Wall

Although memory speed has seen a huge improvement throughout it's history, processors have evolved much faster than memory, so a memory wall [16] [10] had to be admitted. Today, it is obvious that memory access forces the processor to go, costing a lot of computational power. Solutions such as complex cache hierarchies, and memory access instructions reorganizations have been applied to soften the implications of that memory wall, hiding the access latencies, but some people still predict that it will not be enough [10] because of a growing gap between both components.

These barriers entails challenges to researches and the industry to keep doing research on this area. Some of them are avoidable with bright engineering, but others are natural limits, and a new technology will be needed in order to surpass them.

1.2 Motivation

As have been discussed, computer performance is a hot topic today. Researchers from universities and the industry work hard to bring solutions that procure a few more years of

technological progress in the meantime a new semiconductor technology arrives.

Due to that, research on techniques for improving performance of computer systems is an interesting field of study. Of particular interest for us, computer engineering students, are the architecture techniques, addressed to solve performance issues, mostly exploiting parallelism in algorithms and predictable memory access patterns.

This essay is focused in the study of cache memory techniques, in particular, the replacement policies.

1.3 Objectives

The main objective of the project is to study how cache memory is implemented in a commercial processor, being able, at the end of the project, of modify the original design to change the memory configuration, adding or removing functionality.

The selected processor was a recently released one, a MIPS microAptiv CPU from Imagination Technologies. That eventually brought the opportunity to collaborate with them in the academic courses they were developing, which leaded to new objectives, such as testing the existent and incoming materials, and possible contribution to create new materials.

Chapter 2

Memory Hierarchy

There is a well known principle in computation, named Principle of Locality or Locality of Reference [3], for which is assumed a predictable behaviour in memory access for most programs. This principle identify several different types of locality of reference. The most interesting for the subject of this project are: the Temporal Locality, which predicts that if a memory location is referenced, it is likely to be referenced again in the near future, and the Spatial Locality, for which if a memory location is referenced, nearby locations are likely to be referenced in the near future. This principle, added to the fact that memory becomes smaller, cheaper and faster day by day, and the gap between memory access time and processor speed enlarges, leads to the conclusion that the addition of small memory blocks distributed along the chip, is the most obvious thing to do, to solve the Memory Wall problem.

Cache memories are exactly that, but organized in a hierarchic fashion, from the CPU chip itself, filled with very small and fast, but expensive registers, to the surroundings of the chip, where bigger, slower and cheaper memories are usually located.

Although this information can be found in almost every computer architecture book, i.e. [5][4], a brief summary is given here.

The memory hierarchy is divided in several layers of memory, usually L1, L2 and L3 caches. The first layers are integrated in the CPU core, consisting on SRAM blocks not bigger than a few KB. The later are located in the chip surroundings, and are usually shared

between the CPU cores in multicore systems. These are certainly bigger than the previous, in the order of a few MB. Behind the last levels of cache, there is the main memory, reaching today between 8 and 16 GB of DRAM or SRAM for a home mid to high end computer. The last layer is the so called secondary and tertiary storage memory, virtually infinite.

This structure is usually transparent to the processor, so it just makes a memory reference and the cache controller search for the data in the first layer. If it is not there, then the access is forwarded to the next layer, and so on, until the data is located, copied to the previous layers and put at processors' disposal.

To take advantage of the Spatial Locality, discussed above, it is typical that not only the referenced location is copied to the previous layers, but a set of locations around the current reference, what is called the cache line.

Is also important to note that the cache memory not only has to store the data, but also the address of the data location in main memory, and usually some bits of metadata.

2.1 Cache Memory Organization

There are several decisions to make in order to design a proper cache memory hierarchy. The factors to consider are the Placement Policy, to decide where, in the cache memories, to place data from main memory; the Replacement Policy, to decide, in cases where there are more than one placement option, and all of them are occupied, which block to evict; the Extraction Policy, to select which memory locations to copy in cache memory; and the Write Policy, to decide how writes are handled in the hierarchy. The next sections elaborate the Replacement Policy, which is of special interest for the matter of this project, and the Placement Policy, needed in order to understand the application of the first. For the sake of simplicity, the following figures represent caches with a line size of 1. In the case of a line size bigger than 1, both the memory and cache are considered to be divided in blocks, instead of registers, so offset bits have to be added to cache tag to reference the actual memory location inside the block.

2.2 Cache Placement Policy

Cache memories are several times smaller than main memory. Due to that, there is never a one to one correspondence in memory locations between main and cache memories, and there exists the need to establish an algorithm to select where in the cache to allocate the data from main memory. The common approaches are:

2.2.1 Direct Mapping

A diagram is shown in figure 2.1. The lower bits of the data address in main memory are used to allocate the data in the cache memory, and the rest of the address is stored as a tag, so the memory controller is able to rebuild the full address. This usually results in a lot of data being evicted and re-cached continuously.

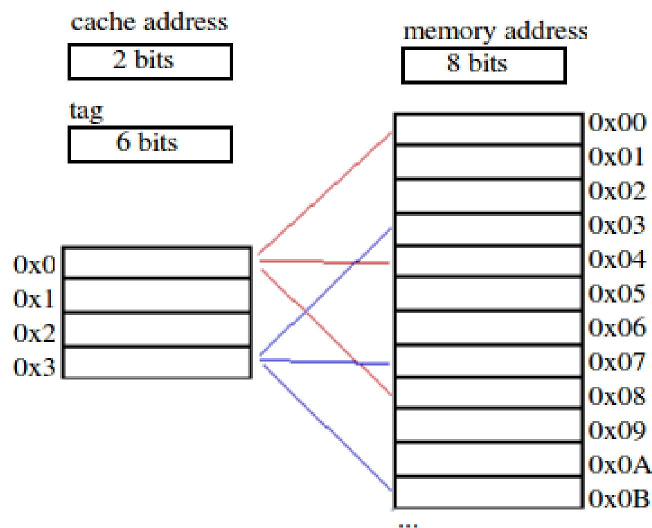

Figure 2.1: *Direct mapping cache diagram*

2.2.2 Pure Associative

As shown in figure 2.2, all the main memory address bits are stored as tag, and the data can be allocated in whichever position into the cache memory. This method reduces to the

maximum allowed by the cache size, the number of evictions, but has the problem of the huge number of parallel comparisons needed to know where data is located.

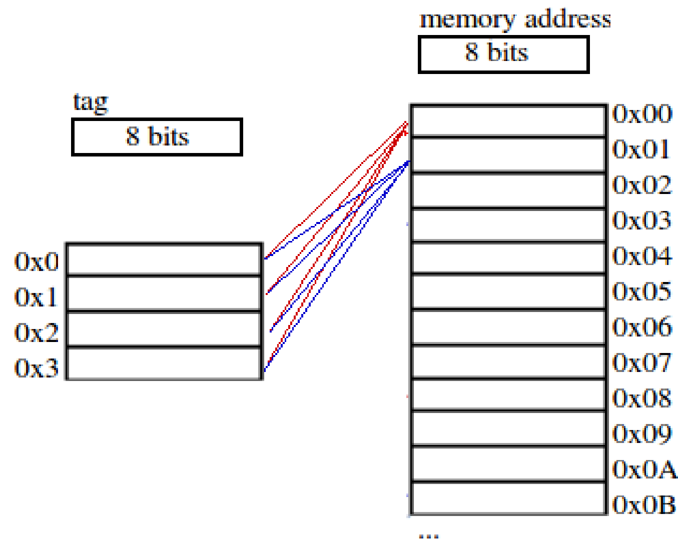

Figure 2.2: *Pure associative cache diagram*

2.2.3 Set Associative

This is a trade-off between the number of evictions and the number of comparisons needed. The memory is divided in sets of fixed size. These sets are mapped by Direct Placement, but inside the set, all the positions are Associative, so the data can be allocated in whichever location that is more convenient. Figure 2.3 illustrates this policy. Set Associative results to be a general case of the above organizations, being Direct Mapping a special Set Associative with a set size of one, and Pure Associative, the special case where the set size equals the memory size.

2.3 Cache Replacement Policy

As described above, the Replacement Policy decides which cache location to evict when a new block of data has to be allocated. This policy is only meaningful in memories that use

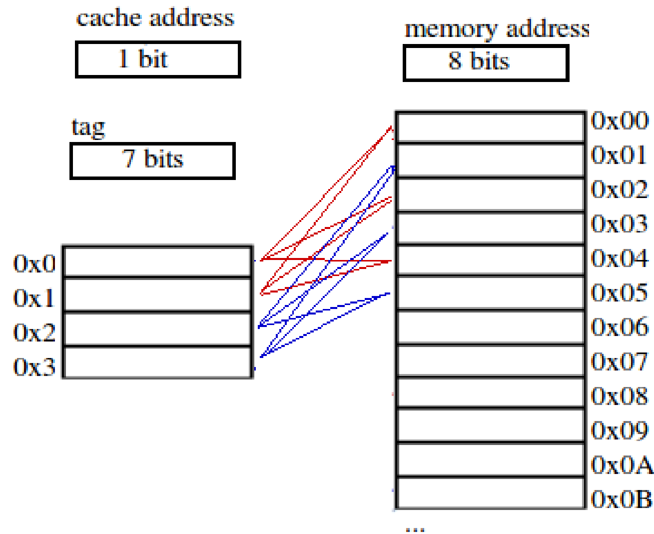

Figure 2.3: *set associative cache diagram*

associative approaches as a Placement Policy, because they need a way to select which data to evict in order to allocate the new data inside the associative set.

This policy has a huge impact in the performance of the memory hierarchy, but it is quite difficult to came to a good approach for all the possible situations, because different programs exhibit different access patterns, some of which can only be classified as random.

Because of that access pattern variety, there exist several different Replacement Policies. The most frequently used are described below. A more detailed description and some performance evaluations can be found in [1]

2.3.1 Least Recently Used (LRU)

This policy tries to take full advantage of Temporal Locality, evicting always the least referenced data location. When a new block enters the cache, the least recently used block is evicted, and a new least recently used block is selected. In case of a hit, the least recently used one has to be recalculated. A common way to implement this policy is with an ordered list of the set locations, in which one of the ends is the most recently referenced location,

and the other end is the least recently referenced location, which will be selected for eviction if needed. When a cache access occurs, the list have to be recomputed. It behaves very well, but its inherent complexity makes it difficult for it to be precisely implemented in a real system.

2.3.2 Most Recently Used (MRU)

In contrast with the previous, this strategy selects for eviction the most recently used data, which results in more hits than LRU in some kind of access patterns.

2.3.3 FIFO

This simple approach maintains a FIFO structure inside the associative set, in which the first allocated data is also the first evicted one.

2.3.4 Random

Always selects a random location to evict, which, surprisingly, performs pretty well in general, because of the stochastic nature of some access patterns.

2.3.5 Not Recently Used (NRU)

NRU is a simplification of LRU, in which the only information carried is if a cache location has been referenced in the near past. When an eviction takes place, one of the not recently used locations is selected. In the general case, more than one is selectable for eviction, so the evicted one is usually chosen randomly among them. A common implementation consists of one bit per sets' line. When a hit occurs, its corresponding bit is set. In case of eviction, a random line between the ones whose corresponding bit is not set is randomly selected for eviction, and the new block's bit is set. If all bits are set, they are all cleared, and so, marked as not recently used.

Chapter 3

MIPSfpga

MIPSfpga is an initiative from Imagination Technologies, with the aim to provide universities the tools to teach real world computer microarchitecture. It is a full package, includes an unobfuscated MIPS core, synthesis and simulation software, build tools and libraries, and a set of laboratories to get practice with the development process. A detailed description from [12] is provided below about the core and the labs.

The labs for the course are centred around the unobfuscated commercial MIPS soft-core processor provided as a set of Verilog hardware description language (HDL) files included in the MIPSfpga Getting Started package [7]. The installers for the programming and debugging tools (Codescape MIPS SDK Essentials and OpenOCD) are also included in that package.

3.1 MIPS Core

MIPSfpga is a version of the microAptiv UP core used in the popular Microchip PIC32MZ microcontroller and in the recent Samsung Artik-1, a low-power module tailored for the Internet of Things (IoT). It implements the MIPS32r3 instruction set architecture in a 5-stage pipeline [9]. The released core includes a memory management unit (MMU) with translation lookaside buffer (TLB), instruction and data caches, and several interfaces (such as EJTAG), as shown in Figure 3.1. The bus interface unit supports the AMBA 3 AHB-Lite protocol [2] [8]. Detailed specifications of the full core can be found in the datasheet

[9].

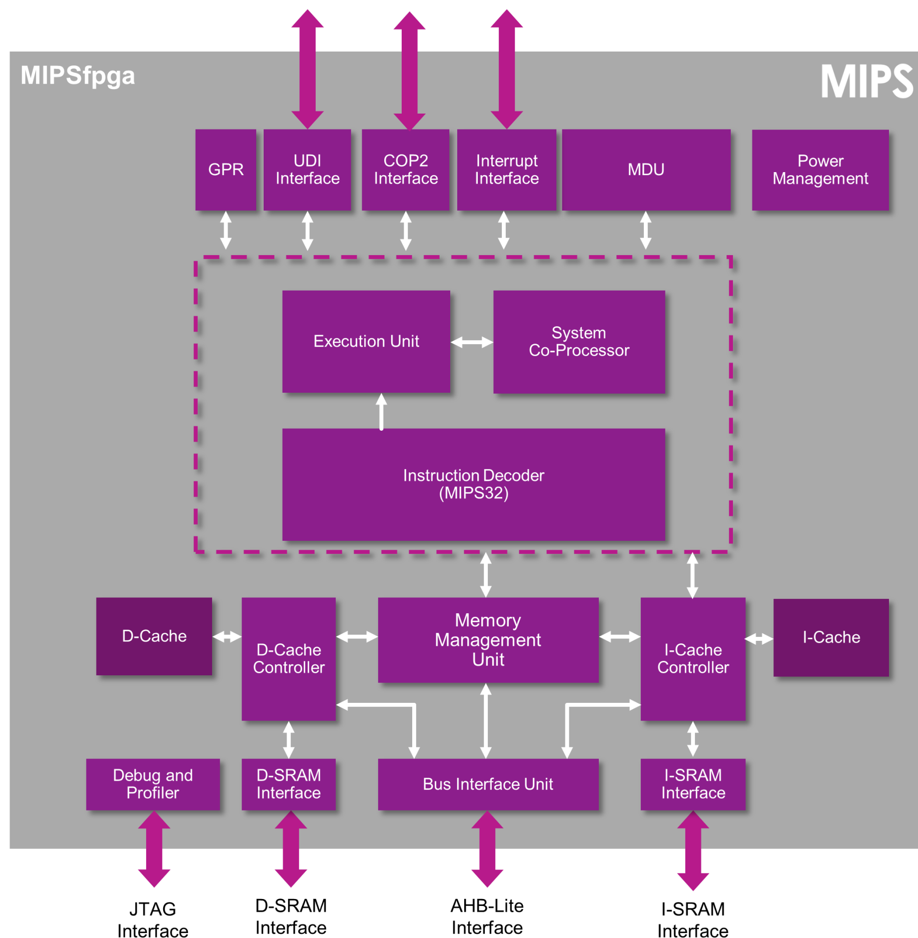

Figure 3.1: *MIPSfpga block diagram (from the MIPSfpga Getting Started Guide [7])*

3.2 MIPSfpga System

The MIPSfpga system, shown in Figure 3.2, includes the MIPS core and peripherals that communicate with the core via the AHB-Lite Interface. The peripherals include memory, implemented as block RAM on the FPGA, and general-purpose I/O (GPIO) that interacts with the LEDs and switches on the FPGA boards. The system requires at a minimum a system clock (SIClkIn) and a low-asserted reset signal (SIResetN) to run. While not

fundamentally required, the EJTAG interface facilitates development and testing of the system by enabling users to easily download and debug programs on the MIPSfpga system. The physical memory map for the MIPSfpga system has two populated blocks, one starting at address 0x0 and the other starting at address 0x1fc00000, as shown in Figure 3.3.

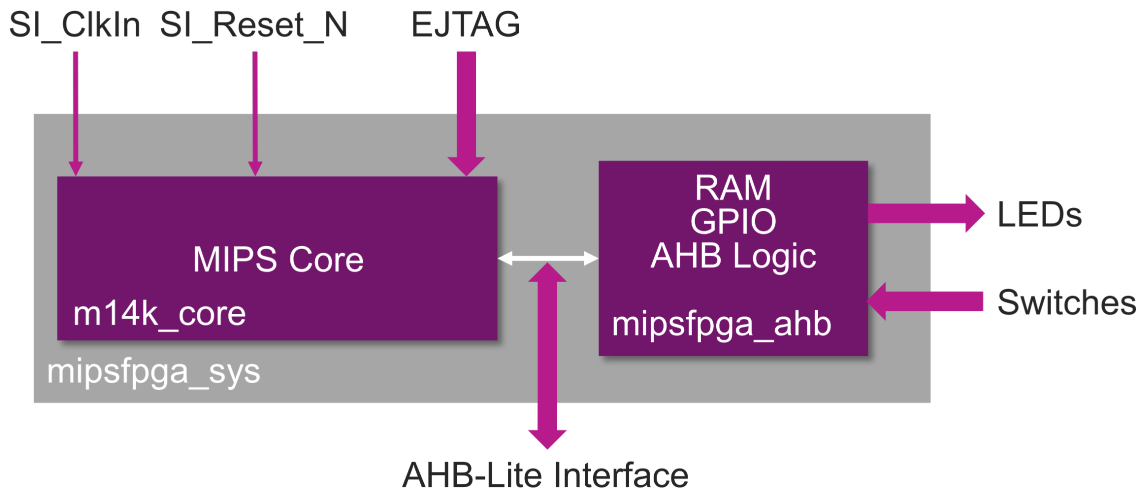

Figure 3.2: *MIPSfpga system (from the MIPSfpga Getting Started Guide [7])*

Upon reset, the MIPS processor begins fetching instructions at address 0x1fc00000. So, at a minimum, memory at that physical address must be populated. Typically, instructions starting at that address contain boot code that initializes the system and then jumps to the user code located in the lower memory block, labelled Code/Data RAM in Figure 3.3. However, as described in the MIPSfpga Getting Started Guide, simple programs that don't take advantage of system features such as caching can be placed directly at physical address 0x1fc00000 and run immediately upon reset. The MIPSfpga system has 128 KB of boot RAM (labelled Reset RAM in Figure 3.3) and 256 KB of program RAM (labelled Code/Data RAM) when targeted to either the Nexys4 DDR or DE2-115 FPGA boards. However, the system can run with smaller amounts of memory, for example, when targeting smaller FPGAs, as described in the MIPSfpga Fundamentals materials for the Basys3 and DE0 FPGA boards.

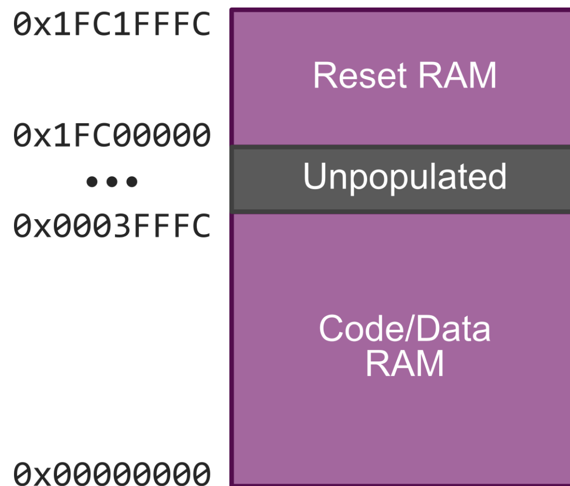

Figure 3.3: *MIPSfpga physical memory map (from the MIPSfpga Getting Started Guide [7])*

3.3 Memory Hierarchy

The Memory Hierarchy is composed of the Main Memory, divided in two blocks, a 128 KB boot RAM and a 256 KB program RAM and a L1 cache, also divided in two 4Kb RAM blocks, Instruction Cache and Data Cache, completely independent from each other. Both are configured as 2-way associative, with 2KB per way, and 16B per line. The Replacement Policy is LRU, and the Write Policy is configurable per Main Memory section, whose options are: WB, WT-NWA, WT-WA or not cacheable.

3.4 Comparison with Other Open Softcore Alternatives

Although there are several alternatives of soft-core processors that could be used in a project like this, all of them lack to satisfy one or some of the considered constraints. Open source is the main of those, because we need to study the core internals and make modifications on them, which let out Altera's Nios/NiosII, Xilinx's Microblaze and ARM's CortexM0 Design Start. We wanted to work with a real world processor, so it has to be used in real industrial applications. This discards the great majority of alternatives. In addition of all the above

mentioned, that are not used for the industry, we have to mention some well known open source soft core alternatives, such as RISC-V, developed by the University of California, and openRISC, developed by opencores.org. At this point, we can consider as competitive options the processors from the OpenSPARC and LEON families, developed by Oracle and Aeroflex Gaisler, respectively. But they have some disadvantages: they implement the SPARC RISC ISA, not as widely used in academia as MIPS, and they lack of good teaching materials.

MIPSfpga complies all the constraints. Although it is not technically open source, all the source is available unobfuscated, with the requirement to not put in silicon and contact Imagination Technologies before patenting a change. The MicroAptiv UP core is also used in the industry, it implements the MIPS ISA, and has a good set of teaching materials, from core synthesis, implementation and programing to integration in a System-On-Chip, capable of running linux.

Chapter 4

Development of New Cache Replacement Policies for the MIPSfpga Core

To implement new Replacement Policies, we had to study the actual HDL implementation. The comments from the HDL files and the HDL code itself was the main source of information, as well as the provided documentation [9]. One of the consulted document [13] was internal to Imagination Technologies, and thus, confidential.

4.1 The Cache Hierarchy in MIPSfpga

The MIPSfpga Microaptiv core has a simple cache hierarchy, as shown in figure 4.1. It is composed of only one cache layer, L1, integrated in the core. This layer is divided in two cache memories: Instruction and Data Cache, completely independent from each other. Both of them are 2-way associative 4KB DRAM blocks, with 16Bytes per line, LRU replacement policy and separated cache controllers. Both of them also have a Fill Buffer [13], that is a 2-line buffer in which the data is firstly stored before it is copied to the cache. When the Fill Buffer is empty, and a new block has to be written to cache, it is copied to the buffer's front line, which is read-accessible. If another block has to be written to cache, and the previous still remains in the buffer, the block is copied in the buffer's back line, not read-accessible. If more blocks needs to be copied in cache, some idle cycles are mandatory until there is available space in the Fill Buffer.

In a few cycles, the front line is moved to the cache, and the back line, to the front.

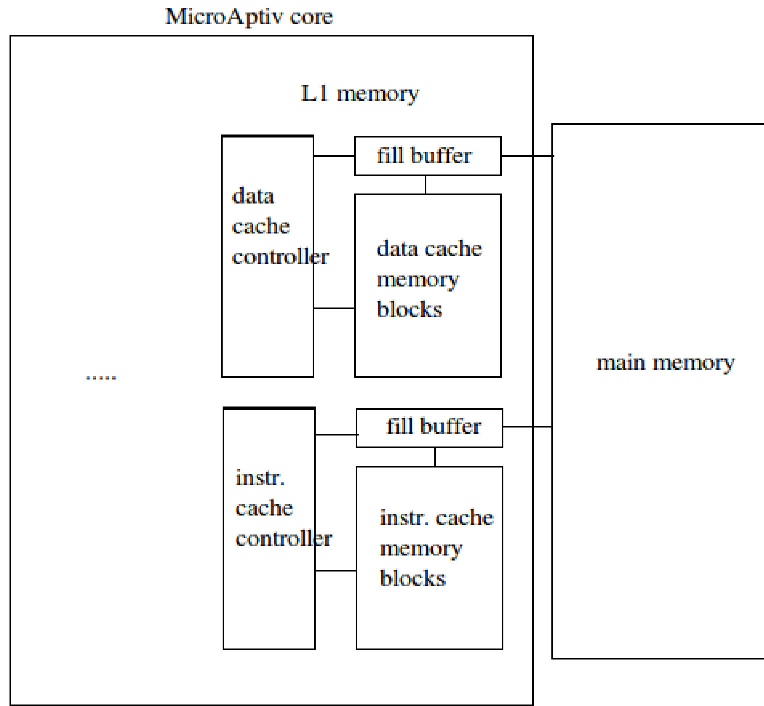

Figure 4.1: *microAptiv memory hierarchy diagram*

4.2 The LRU Implementation of MIPSfpga

LRU is implemented inside the cache controller, so the first thing to do was to understand the algorithm and isolate it in a different HDL module.

It is implemented in a way that there is no need to read the current LRU bits of a cache set, in order to establish the new ones after a new allocation.

They can do it because of a very particular algorithm and the usage of write masks in the cache memory.

The algorithm is described in the following sections.

Way	LRU[5:0] written on hit	LRU[5:0] written on invalidate	we[5:0] (write mask)
0	1x11xx	0x00xx	101100
1	x1x01x	x0x10x	010110
2	0xxx01	1xxx10	100011
3	x00xx0	x11xx1	011001

Figure 4.2: *LRU encode algorithm. The first column is the way that caused the hit, or the one to invalidate. The second and the third are the LRU bits to be written in the case of hit or eviction, respectively, and the forth is the write mask.*

4.2.1 Allocation

When a new data block is to be written in cache, the decode phase occurs (figure 4.3), to select, using the LRU bits of the set, where to allocate the new data. Some cycles later, the encode phase takes place, in which the LRU bits to be written and the write mask are computed based on the algorithm in the figure 4.2.

4.2.2 Hit

At hit, there is no decode phase; The LRU bits and the write mask are computed from the algorithm in figure 4.2.

4.2.3 Invalidation

In the case of an invalidation, as well as before, there is no decode phase; the LRU bits and the write mask are computed from the encode algorithm (figure 4.2).

4.3 Changes in the Cache Configuration

We had interest to work with 4-way associative caches, so the first change in the source code was to add new Cache Configurations, such as 1, 3 and 4-way associativity, and various cache sizes. In addition to the default 2KB per way, we implemented 4KB per way caches for all

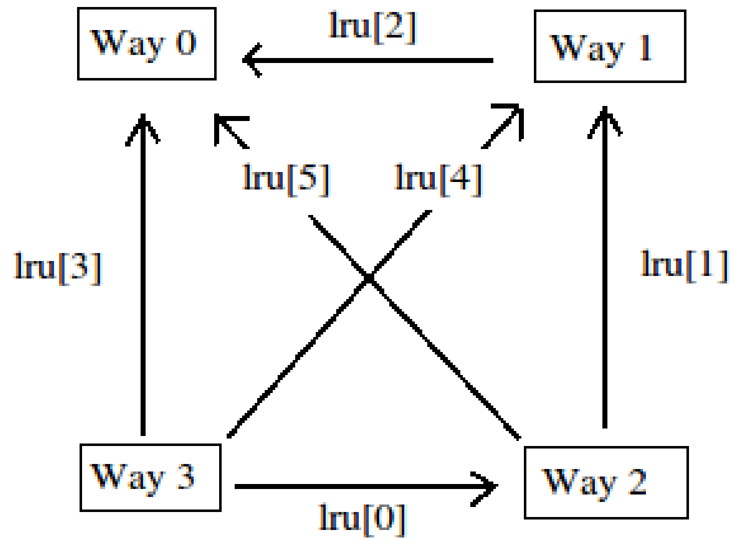

Figure 4.3: *LRU decode algorithm. The Way to be evicted is the one whose three arrows are pointing to. If a LRU bit is 1, the arrow is as shown, if it is 0, the arrow is inverted.*

the types of associativity.

The Core is actually pretty configurable. To change the cache configuration, the next lines in the m14k_config.vh have to be edited.

```
// Cache Associativity
//  1-4 way set associative

`define M14K_ICACHE_ASSOC 2
`define M14K_DCACHE_ASSOC 1

// Cache Way Size
//  Size/Way in KB
//  1,2,4,8,16 KB

`define M14K_ICACHE_WAYSIZE 2
`define M14K_DCACHE_WAYSIZE 2
```

```
`define M14K_IC_TAGRAM   tagram_2k2way_xilinx
`define M14K_DC_TAGRAM   tagram_2k2way_xilinx
`define M14K_IC_WSRAM    i_wsram_2k2way_xilinx
`define M14K_DC_WSRAM    d_wsram_2k2way_xilinx
```

```
`define M14K_IC_DATARAM dataram_2k2way_xilinx
`define M14K_DC_DATARAM dataram_2k2way_xilinx
```

In the last lines, the wrappers for the cache memories have to be specified. Those files have to be created to accommodate the new configuration.

4.4 Implementation of a NRU Policy

Our first attempt in Replacement Policies was to implement the NRU Policy. For that, the algorithm was changed to use only as many bits as ways implemented. A value of 1 in one of those bits, represents that the corresponding line was not recently used. The decode phase consist in selecting the line for eviction, which was done choosing the lowest not recently used line. If all lines are recently used, i.e. all bits set to 1, all the NRU bits have to be cleared. The encode phase consists in setting to 1 the corresponding bit of the new allocated line, or the one that caused the hit.

This implementation has a mayor problem. We can not write in the decode phase, so there is no option to clear the NRU bits in that phase, and as the original LRU does not need to read the previous LRU bits in order to set the new ones, we can not read the NRU bits in encode, so we do not know if all bits are set in that phase. Due to that, there is no obvious way to implement the NRU bits reset.

Although is has not been implemented yet, we have a NRU implementation proposal for this scenario: As the decode phase is always followed by the encode phase some cycles later, intermediate registers in the cache controller could be used to temporarily store the NRU bits.

4.5 Implementation of a FIFO Policy

LRU behaves as FIFO when there are no hits, because the first inserted line is also the least recently used, so the change consists in setting the memory mask to all zeros for the case of a hit. The other encode cases are the same as LRU, and so are all the decode cases.

Chapter 5

Implementation of Tests

To prove the policies behaviour and the correctness of the changes, we need hardware support. The kind of support we need is access to CPU parameters, to ensure the hardware is configured as expected, and performance counters, capable of counting internal events such as data cache accesses and misses.

A description of the hardware support found in microAptiv is given below. Some of the following sections contains information from [15] and other are textual cites from that document.

Then, an overall description of the implemented microbenchmark structure is given.

5.1 The MIPS Coprocessor Model

The MIPS ISA provides up to four coprocessors. A coprocessor extends the functionality of the MIPS ISA, while sharing the instruction fetch and execution control logic of the CPU. Some coprocessors, such as the system coprocessor and the floating-point unit, are standard parts of the ISA and are specified as such in the architecture documents. Coprocessors are generally optional, with one exception: CP0, the system coprocessor, is required. CP0 is the ISA interface to the PRA and provides full control of the processor state and modes.

5.2 Coprocessor 0

CP0 provides an abstraction of the functions necessary to support an operating system: exception handling, memory management, scheduling, and control of critical resources. The interface to CP0 is through various instructions encoded with the COP0 opcode, including the ability to move data to, and from, the CP0 registers, as well as specific functions that modify CP0 state. The CP0 registers and the interaction with them make up much of the PRA.

To access these registers, the MIPS standard library offers some useful macros under mips/m32c0.h.

5.3 Core Configuration

Core configuration is accessible via Config0 to Config5 CP0 registers. In particular, we are interested in Config0 and Config1 registers, as they encode the memory coherency algorithm for each memory segment, and the cache attributes. Figure 5.1 shows the Config0 Register Format, and table 5.1 describes the relevant register fields.

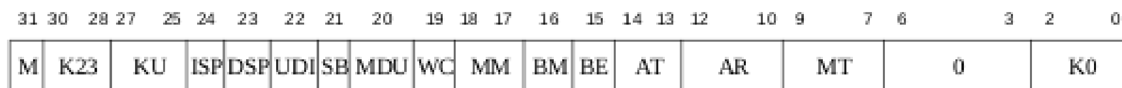

Figure 5.1: *Config0 Register Format (from the microAptiv User Manual [14])*

Figure 5.2 shows the Config1 Register Format, and table 5.4 describes the relevant register fields.

Figure 5.2: *Config1 Register Format (from the microAptiv User Manual [14])*

Name	Bit(s)	Description	Read/Write
K23	30:28	This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations. This field is valid in the FM-based MMU processors and is reserved in the TLB-based MMU processors. Refer to 5.2 for the field encoding.	FM: R/W TLB: R
KU	27:25	This field controls the cacheability of the kuseg and useg address segments in FM implementations. This field is valid in the FM-based MMU processors and is reserved in the TLB-based MMU processors. Refer to 5.2 for the field encoding.	FM: R/W TLB: R
K0	2:0	Kseg0 coherency algorithm. Refer to 5.2 for the field encoding.	R/W

Table 5.1: *Some Config0 Register Field Descriptions*

C(2:0) Value	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
3,4,5,7	Cacheable, noncoherent, write-back, write allocate
2, 7	Uncached

Table 5.2: *Cache Coherency Attributes*

Name	Bit(s)	Description	Read/Write
DS	15:13	This field contains the number of data cache sets per way. 0x0:64 0x1:128 0x2:256 0x3:512 0x4:1024 0x5 -0x7: Reserved	R
DL	12:10	This field contains the data cache line size. If a data cache is present, then it must contain a line of 16 bytes. 0x0:No D-Cache present 0x2:16 bytes 0x1, 0x2, 0x4 - 0x7:Reserved	R
DA	9:7	This field contains the type of set associativity for the data cache. 0x0:Direct mapped 0x1:2-way 0x2:3-way 0x3:4-way 0x4-0x7: Reserved	R

Table 5.3: *Some Config1 Register Field Descriptions*

5.4 Performance Counters

The MIPSfpga core does not implement traditional performance counters for the internal events. Instead, it has performance monitors, implemented inside the Coprocessor 0. They are composed of two configurable 32-bit counters, with a huge variety of events to count. The configuration and count value of the counters are accessible via the CP0 Performance Counter registers. Each counter has two register, the Configuration Register and the Count Register. The Configuration Register and Count Register Format can be seen in figure 5.3 and 5.4 respectively. The Event field in the Control Register is where the event to count is encoded; as there are many possible events, they are not listed here. For more information, refer to the MicroAptiv User Manual [14].

Figure 5.3: *Performance Counter Control Register Format (from the microAptiv User Manual [14])*

Figure 5.4: *Performance Counter Count Register Format (from the microAptiv User Manual [14])*

5.5 Implementation of Microbenchmarks

The tests were implemented based on the MIPSfpga Labs code, to initialize TLB, cache, and other processor parameters.

To display information, seven segment displays and switches were used, because there is no UART inside the core.

Name	Bit(s)	Description	Read/Write
DS	15:13	This field contains the number of data cache sets per way. 0x0:64 0x1:128 0x2:256 0x3:512 0x4:1024 0x5 -0x7: Reserved	R
DL	12:10	This field contains the data cache line size. If a data cache is present, then it must contain a line of 16 bytes. 0x0:No D-Cache present 0x2:16 bytes 0x1, 0x2, 0x4 - 0x7:Reserved	R
DA	9:7	This field contains the type of set associativity for the data cache. 0x0:Direct mapped 0x1:2-way 0x2:3-way 0x3:4-way 0x4-0x7: Reserved	R

Table 5.4: *Some Config1 Register Field Descriptions*

The general structure of the tests is as follows. First, the core is configured in high privileged mode through the assembly source code taken from MIPSfpga labs. Then, the main function is called in kernel mode.

This function reads some processor parameters and assigns each of them to a specific switch state. Then, a test function is called, in which the performance counters are configured, the test code is executed, and the counter values are read.

The test code is directly written in MIPS assembly, with the volatile modifier specified, to ensure that memory accesses are never optimized to registers. Also, the compiler optimization is disabled.

Is important to note that call several test functions in the same execution may lead to unexpected cache results, because the memory locations may actually have been referenced in the previous test. Due to that, only one test function is usually called in an specific execution.

Chapter 6

Experiments

Several tests were implemented in order to prove the various cache parameters, as well as some to simulate several access patterns to check the policies behaviour and compare the performance. Other tests were implemented to prove the performance improvement obtained when some optimization techniques are applied. The tests are described below.

6.1 FIFO Correctness

To check if the FIFO implementation is actually behaving as FIFO, a simple test is executed. For this test, a 4-way associative cache is assumed. It consists in accessing four memory locations that map to the same set, so it is completely filled. Then the first entry is referenced again. An LRU cache should change the least recently used block to be the second entry. A new memory location that maps in the same set is referenced, so one entry have to be evicted. To check the evicted one, the first entry is again referenced. In the case a FIFO policy is implemented, the first entry should have been evicted, so this last reference should be a miss. In the LRU case, the second entry should have been the evicted one, so the last reference should be a hit. The code below implements this test.

```
asm volatile
(
    "   lui $t6, 0x8000;"       // t4 -> array[]
    "   addiu $t6, $t6, test_data4;"
    "   lui $t4, 0x8000;"       // t4 -> array[]
    "   addiu $t4, $t4, test_data4;"
    "   li $t2, 0;" // t2 -> i
```

```
        "    li $t3, 4;"
        "    lw $t5, 0($t4);"
        "    lw $t5, 2048($t4);"
        "    lw $t5, 4096($t4);"
        "    lw $t5, 6144($t4);"

        "    lw $t5, 0($t6);"     // Hit array[0]
        "    lw $t5, 8192($t4);"     // load array[2048] in lru set0
        "    addiu $t2, 256;"     // enough time to empty fill buffer
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        "    addiu $t2, 256;"
        // If fifo is implemented, set0[0] should be replaced
        // if lru is implemented, set0[1] should be replaced
        "    lw $t5, 0($t4);"     // array[0] to check lru from set0[0] to set0
[1]
 );
```

6.2 Sequential Access Pattern

This is a typical access pattern, usually seen in array iterations. It is represented by

$$(a_1 a_2 a_3 ... a_{k-1} a_k)^N$$

With $k <$ cache size. When $k >$ cache size, it is usually called a Thrashing Access Pattern [6]. The code is shown below.

```
    asm volatile
    (
        "    li $t1, 1024;"                // t1 -> k
        "    li $t6, 50;"              // t6 -> N
        "sec_n_loop:"
        "    lui $t4, 0x8000;"        // t4 -> array[]
        "    addiu $t4, $t4, test_data4;"
        "    li $t2, 0;"              // t2 -> i
        "    addi $t6, -1;"
        "sec_loop:"
        "    lw  $t5, 0($t4);"   // array[i]
        "    addi $t4, $t4, 1;" // i = i+1
        "    addi $t4, $t4, 1;" // i = i+1
        "    addi $t4, $t4, 1;" // i = i+1
        "    addi $t4, $t4, 1;" // i = i+1
```

```
        "    addi $t2, $t2, 1;"
        "    blt $t2, $t1, sec_loop;"
        "    nop;"
        "    bgtz $t6, sec_n_loop;"
        "    nop;"
    );
```

6.3 Recency-friendly Access Pattern

$$(a_1a_2a_3...a_{k-1}a_ka_{k-1}...a_3a_2a_1)^N$$

This access pattern represents a typical stack access pattern that repeats N times [6]. For any value of k, the access pattern benefits from LRU replacement. Any other replacement policy besides LRU can degrade the performance of these access patterns. This pattern has been implemented with the code shown below.

```
    asm volatile
    (
        "    li $t1, 2048;"              // t1 -> k
        "    li $t6, 5;"              // t6 -> N
        "rec_n_loop:"
        "    li $t2, 0;"              // t2 -> i
        "    lui $t4, 0x8000;"        // t4 -> array[]
        "    addiu $t4, $t4, test_data4;"
        "    addi $t6, -1;"
        "rec_up_loop:"
        "    lw  $t5, 0($t4);"   // array[i]
        "    addi $t4, $t4, 4;" // i = i+1
        "    addi $t2, $t2, 1;"
        "    blt $t2, $t1, rec_up_loop;"
        "    nop;"
        // reduce index so it does not access i=k again
        "    addi $t4, $t4, -4;" // i = i-1
        "    addi $t2, $t2, -1;"
        "rec_down_loop:"
        "    lw  $t5, 0($t4);"   // array[i]
        "    addi $t4, $t4, -4;" // i = i-1
        "    addi $t2, $t2, -1;"
        "    bgez $t2, rec_down_loop;"
        "    nop;"
        // correct index to i=0
        "    addi $t4, $t4, 4;" // i = i-1
        "    addi $t2, $t2, 1;"
        "    bgtz $t6, rec_n_loop;"
        "    nop;"
    );
```

6.4 Mixed Access Patterns

In order to test more general access patterns, we have implemented some mixed patterns as described in [6]. These are: a mixed pattern based on the sequential access described above

$$[(a_1 a_2 a_3 ... a_{k-1} a_k)^A P_\epsilon (b_1 b_2 b_3 ... b_{m-1} b_m)]^N$$

And a mixed pattern based on the recency-friendly access

$$[(a_1 a_2 a_3 ... a_{k-1} a_k a_{k-1} ... a_3 a_2 a_1)^A P_\epsilon (a_1 a_2 a_3 ... a_{k-1} a_k .. a_m)]^N$$

Where $k <$ cache size, $m >$ cache size and $0 < \epsilon < 1$. P_ϵ(<pattern>) means a probability of ϵ to execute a (<pattern>) access.

6.5 Array Enlargement Optimization Technique

Assuming a Direct mapped, 512B cache, with a line size of 4; an operation like

$$C = C + A[i] + B[i]$$

From $i = 0$ to $i = 511$, if A and B are defined to be 512B arrays, and B is just after A in memory, will result in $512 * 2$ memory accesses, being all of them fails. That is because when B[i] is accessed, it, along with the next 3 locations, are copied to cache; but, when A[i] is accessed, B[i] is replaced by it in cache, so, when B[i+1] is accessed, it causes another fail. This situation can be prevented by defining A to be a little longer, so A[i] and B[i] does not maps to the same cache line. When this technique is applied, the number of fails reduces to $1/linesize$. The following code illustrates this:

The original code:

```
uint8_t A[512];
uint8_t B[512];
```

Is changed to

```
uint8_t A[516]; // A[] size has been enlarged to be 4 more bytes long
uint8_t B[512];
```

6.6 Array Merge Optimization Technique

The situation described above can also be solved by defining A and B in such a way that B[i] is right after A[i] in memory, then A[i+1], B[i+1] and so on. In this case, the original code:

```
uint8_t A[512];
uint8_t B[512];
```

Is changed to

```
struct
{
  uint8_t A;
  uint8_t B;
} array[512];
```

Now, when A[i] is accessed, B[i], A[i+1] and B[i+1] are also cached, so there are no unnecessary evictions and the number of fails is reduced by a factor of 1/4, the same amount as with the Array Enlargement technique.

6.7 Other Tests

More tests have been implemented to prove different cache parameters. As they are not as relevant as the ones described above, they are not included in this section, but they can be referred in the appendix.

Chapter 7

Other Activities

This project has certainly involved many activities in addition to those described in previous sections.

The collaboration with Imagination Technologies' MIPSfpga team began with the beta test of the course materials, for which several feedback was reported to the team. In addition, a linux installation guide for the tools was made and published in the Imagination university programme forum. We have recently made a document based on my linux tutorial to include in the next MIPSfpga version.

I also had the opportunity to participate as speaker in the Madrid MIPSfpga workshop, celebrated in the Complutense University of Madrid. There I introduced the Advanced course, with a demonstration of the final system running on fpga. That workshop also involved presentations from Imagination Technologies, Xilinx and Altera, as well as multiple Hands-On experiences.

Finally, I should mention that Imagination Technologies has granted me with a scholarship for my collaboration in the MIPSfpga project along the 15-16 academic year.

Chapter 8

Results

As shown in the tables 8.1, 8.2, 8.3 and 8.4, LRU and FIFO have the same results in most of the executed tests.

Sequential access pattern (table 8.1) misses 1 in 4 references in the 2 KB direct mapped cache, because of the cache size. The data array is exactly twice the size of the cache, so all the cached data is evicted before it is referenced again. In the other cases, as the cache memory is at least twice the first case, the only misses are the mandatory ones. The following references find the data in the cache.

In Recency-friendly Test (table 8.2), the same as the previous applies, but now with a different data array size.

In Mixed patterns (tables 8.3 and 8.4), LRU and FIFO exposes the same behaviour as well, and again, is because of the relative size of the data respect to the cache. Recency-friendly has a better Reads/Misses ratio, because it references the data twice than sequential pattern. As this data is completely cached in the first read (it fits in cache), the second reads are all hits.

The Array Enlargement and Array Merge optimization techniques were also tested. For that, only a 2K direct mapped cache was used, with the results in the table 8.5. In both cases, the misses have been reduced to just the mandatory ones.

Cache	Reads	Misses	Reads/Misses
2k1w_lru	51200	12802	4
2k2w_lru	51200	258	198.4
2k3w_lru	51200	256	200
2k4w_lru	51200	256	200
2k1w_fifo	51200	12802	4
2k2w_fifo	51200	258	198.4
2k3w_fifo	51200	256	200
2k4w_fifo	51200	256	200

Table 8.1: *Sequential Access Pattern Test Results. The patterns is described in section 6.2. For this test, $k = 1024$ and $N = 50$*

Cache	Reads	Misses	Reads/Misses
2k1w_lru	20480	3970	5.2
2k2w_lru	20480	2818	7.3
2k3w_lru	20480	1666	12.3
2k4w_lru	20480	514	39.8
2k1w_fifo	20480	3970	5.2
2k2w_fifo	20480	2818	7.3
2k3w_fifo	20480	2690	7.6
2k4w_fifo	20480	514	39.8

Table 8.2: *Recency-friendly Access Pattern Test Results. The pattern is described in section 6.3. For this test, $k = 2048$ and $N = 5$*

Cache	Reads/Misses
2k1w_lru	8
2k2w_lru	7.8
2k3w_lru	7.6
2k4w_lru	7.8
2k1w_fifo	8
2k2w_fifo	8.1
2k3w_fifo	7.8
2k4w_fifo	7.6

Table 8.3: *Sequential-based Mixed Access Pattern Test Results. The patterns is described in section 6.4. For this test, $k = 512$, $A = 5$, $m = 8192$, $\epsilon = 0.3$ and $N = 100$*

Cache	Reads/Misses
2k1w_lru	12.6
2k2w_lru	13.1
2k3w_lru	12.9
2k4w_lru	12.5
2k1w_fifo	13.1
2k2w_fifo	13.1
2k3w_fifo	12.6
2k4w_fifo	13.1

Table 8.4: *Recency-based Mixed Access Pattern Test Results. The patterns is described in section 6.4. For this test,* $k = 512$*,* $A = 5$*,* $m = 8192$*,* $\epsilon = 0.3$ *and* $N = 100$

Cache	Reads	Misses	Reads/Misses
No opt	1024	1026	1
Enlargement	1024	145	7.1
Merge	1024	258	4

Table 8.5: *Array Enlargement and Array Merge Test Results. These tests are described in sections 6.5 and 6.6.*

Chapter 9

Discussion

9.1 Cache Memory Study

FIFO replacement policy does not provide better results than LRU in the general case, just as expected, and though the tests made in the project were quite simple and controlled, in comparison to those found out there, we know that FIFO would not behave as good as LRU in real world programs. Also, There were some odds results in the tests, e.g. Array Enlargement Tests results in less than 256 misses, which is hard to believe, because they are mandatory misses; and other results not shown in the tables. These results may be caused by uncontrolled accesses in the configuration code. However, this implementation serves as proof of concept for the project's objective. The core's cache has been documented and modified in a way to allow significant changes in the replacement policy.

Also, the necessary wrappers to change the cache size and associativity have been successfully implemented and tested. Finally, several microbenchmarks for proving different situations have been developed and tested.

9.2 MIPSfpga in Academic Courses

MIPSfpga results to be a great promise as teaching material. It has been tested throughout this project, with very favourable results. Although we do not forget the value of other good alternatives, such as OpenRisc, OpenSparc and LEON, they implement microarchitectures

that happens to be less widely used in academia than MIPS, and most of them have no industrial use. Other universities are using this material in their courses, and UCM has been using it during the 15-16 academic course in one computer architecture course.

Chapter 10

Future Work

10.1 Cache Replacement Policies

We have a NRU implementation proposal, yet to be implemented. Also, is interesting the possibility to test the different cache configurations and replacement policies in the SOC described in the Advanced tutorials, to try them with real-world programs, running in a real-world operating system.

10.2 MIPSfpga Fundamentals Extension Labs

All the documentation and code developed in this project has the potential to be included or used in future extension labs. There is a real interest in cache based laboratories, and new architecture specific laboratories are been developed to be include in the MIPSfpga version 2.0.

At least during the next three months, I will keep collaborating with this new part of the project, documenting and developing architecture based laboratories.

Bibliography

[1] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272. ACM, 2004.

[2] ARM. *AMBA 3 AHB-Lite Protocol Specification*. 2006.

[3] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[4] Sarah Harris and David Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2013.

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

[6] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[7] Imagination Technologies Ltd. Mipsfpga.

[8] Imagination Technologies Ltd. *MIPS32 microAptiv UP Processor Core AHB-Lite Interface*. 2013.

[9] Imagination Technologies Ltd. *MIPS32 microAptiv UP Processor Core Family Datasheet*. 2013.

[10] Philip Machanick. Approaches to addressing the memory wall. *School of IT and Electrical Engineering, University of Queensland*, 2002.

[11] Gordon E. Moore. Cramming more components onto integrated circuits.

[12] Enrique Sedano Daniel Chaver Martinez Sarah L. Harris, Robert Owen. Mipsfpga: Hands-on learning on a commercial soft-core, 2016.

[13] Imagination Technologies. *Data Cache Control Module Definition (Confidential)*. 2001.

[14] Imagination Technologies. *MIPS32 microAptiv UP Processor Core Family Software User's Manual*. 2014.

[15] Imagination Technologies. *MIPS Architecture For Programmers Volume III*. 2015.

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