Final Project book

Cache L0

Ori Steinberg
Mr. Udi Kra
Doc. Adam Teman

Rev. <1.0>
<Oct 2019>

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 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 work on the new Risc-V architecture, an open-source hardware architecture from Berkley University. Specific work on Pulpenix chip, which allowed us to implement and test different cache replacement policies as proof of concept and to show the possibilities of the platform.

Keywords

Cache, Memory Hierarchy, Replacement Policies, Risc-V, Pulpenix, performance.

Acknowledgments

I would like to thank my director Mr. Udi Kra and my academic advisor Doc. Adam Teman for the opportunity to work in this project. To Mr. Udi Kra for his invaluable help around the clock.

To Enics Labs in general and to SOC labs in particular for giving me the opportunity to implement this project on Pukpenix.

Dedication

To my parents and family that support me, believe in me and do whatever they can do to help all along even in the least pleasant times.

Table of Contents

List of Figures

1. Introduction

1.1 Scope

This document discusses a new level of Cache in low order which offers a possible solution for the memory bottleneck. It includes theories, implementation, and results.

1.2 Audience

This document intended for one that interested to build a cache or to research the field of memory access from the aspect of low-level implementation

1.3 Overview

This document presents nowadays status about memory issues including existing solutions. It will introduce the problem of computer speed in general and of memory speed in particular.
I will show my implementation and discuss it and about several issues that come in the way.

2. Background

2.1 The Cache

A CPU cache is hardware that use to reduce the average memory access time, the time to transfer data between the core and the main memory (instructions and data). A cache is a smaller, faster memory, then of the main memory, closer to a processor core, it success to be faster by stores copies of data of the main memory and then transfer them to the core in lower number of cycles (cause higher speed). Most CPUs have different independent caches for the data and instruction, also hierarchy of cache levels (L1, L2, etc.).

Historically L1 cache was one cache but later it divided into L1**d** (for data) and L1**i** (for instructions). Almost all current CPUs with caches have a split L1 cache. They also have L2 caches and, for larger processors, L3 caches as well.

The L2 cache is usually not split and acts as a common repository for the already split L1 caches. Every core of a multi-core processor has a dedicated L2 cache that isn't usually shared with other cores.

The L3 cache and higher-level caches shared between all cores.

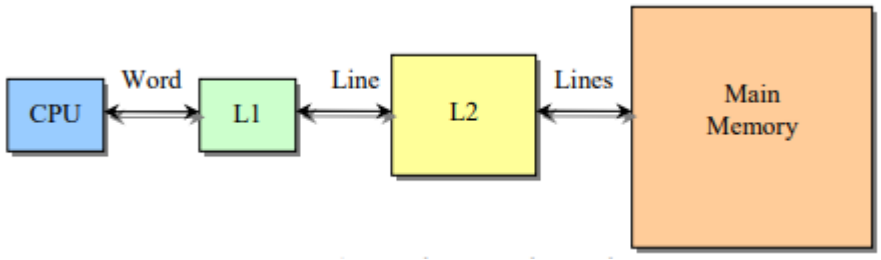


Figure 2-1: Hierarchy Memory structure

2.2 Between Instruction and Data

As we looking, we can immediately notice the difference between instruction memory (IM) and data memory (DM). First of all, the IM is read-only (most of the time) and the DM is both read and write. Furthermore, the nature of IM is to be called one by another sequentially as opposite to DM that in some sections usually call the same data not depended on their locations and in some sections (e.g. arrays) use sequential data.

3. Cache

3.1 Entries

Data transferred between memory and cache occur in blocks of fixed size, called "cache blocks". When a cache block copied from memory into the cache, a cache entry created. The cache entry will include the copied data as well as the requested memory address and other related flags (e.g. valid bit, dirty bit, etc.).

3.2 Operation

When the processor needs data from memory it will request it from the cache, the cache will look for a corresponding entry within itself (expanded in the next section). The cache checks for the address of the requested memory in any cache lines that might contain that address, If it finds that the desired data is in its tables, a cache hit has occurred and the processor immediately gets the data from the cache. However, if the required data isn't in its tables then a cache miss has occurred and the cache allocates a new entry copies data from main memory and sends it to the core.

3.3 Search for Line

It is very important to understand the process that occurs in the cache when the core requests data.
When the cache gets a request of an address from the core it will start with finding in which lines the address can be found, in our case, it is all the lines, then it compares every line's tag to the desired address. If there is a match, the cache checks the **valid bit** to know if the data is valid and can be used, if this isn't the case so the cache will request the data from the next memory level in the hierarchy. When the cache will have the valid data it will update itself and sends the data to the core.

3.4 Algorithms – Replacement Policies

When a cache miss occurred the cache needs to bring new data from memory, it not bringing one word but a **cache block**. Cache block it predefined number of words that we bringing in one memory access.

When there is free space in the cache, the new data will accommodate one of those blocs. If there is no free space the cache needs to decide which block to evict, this role played by the cache algorithm.
To reduce the cache miss, it is necessary to find an algorithm that will minimize the prediction error of the following memory access.
We will base our algorithms on the locality principle, i.e. when reading instructions, it is likely that the required data access will be following line by line, so we can simply bring to the cache blocks of data before the core even asks for it. But clearly, it is not that simple especially when we talk about the data memory.
In this project, I will analyze several algorithms in different test cases to find the optimal algorithm for this kind of cache.

4. Performance and Goals

The cache performance has become important in recent years were the gap between memory access speed to the processor's speed growth incredibly.

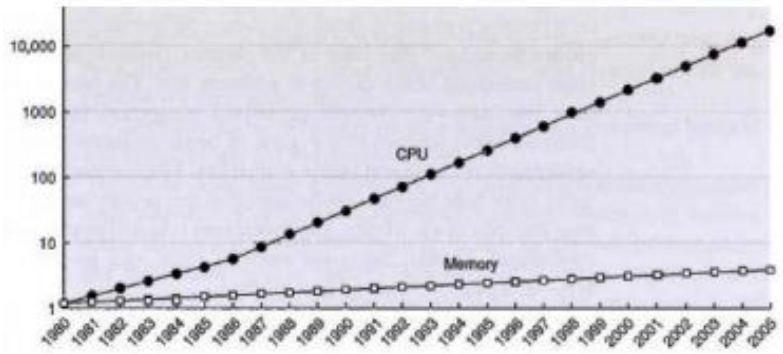


Figure 4-1: CPU vs. Memory performance

The cache created to reduce this gap. Thus knowing how well the cache is able to bridge the gap between the processor's speed and the memory's speed, becomes important, especially in high-memory access systems, e.g. matrices multiplication that we use a lot in Machine Learning and more. The cache hit/miss rate plays an important role in determining this performance. To improve the cache performance, reducing the miss rate becomes one of the necessary steps along with decreasing the access time to the cache.

Goal:
In our case we don't want to enhanced performance, we want to enable running memories at half rate clock at minimal impact on performance, while using low-cost silicon area and prevent memory access time performance bottleneck.

5. Implementation

The cache is built **full associative, write back-through**, (thus concepts will be explained later in this chapter) with default size, that defines by parameters, of:

- Block size - 4 words (of 32 bits each).
- Number of blocks in cache - 8 ways.

Thus the total size of the cache is

$$Total\ size = 8 \cdot 4 \cdot 32 = 1Kb$$

To implement this cache unit I cooperate with SOC labs and I integrate the cache into the **Pulpenix** chip base on **RiscV** architecture.

5.1 Instruction Cache

The cache will contain three components – control, datapath and algorithm (Algo).

5.1.1 Control

The control component will manage the operation of datapath and Algo components, it will decide when to search, when to import new data from memory, prefetching and when activate the algorithm component to generate the next line to replace in our cache.

Control Operation

The control will get from the core as input:

- request – for instruction.
- W/R – type of the operation (instruction cache is read-only).
- Address – instruction address.

Then it will check with the datapath if it holds a line that matches the address, if so then the control will notify the core that the instruction is ready to read. If the datapath doesn't have the data, the control will mark to memory to import the instruction, datapath will use Algo to determine where to accommodate this line in its table. When the data arrive to datapath the control will notify the core that the data is ready and wait for the next request.
Prefetch – if a cache line used more than k times (k is a parameter, in default – half the number of words in cache line), the control bring the next cache line from memory.

5.1.2 Datapath

Datapath will hold the data in "cache lines" in a memory table, its main operation is to find out if the desired address is in the cache table or not, and to provide the data to the core.
For each cache line, the datapath will hold reference information:

- Tag - address of the line.
- Valid bit - to know if this line contains valid information.
- Dirty bit - to know if this line has been written by the core.

Operation

Datapath gets as input an address and control signals. When the control signal to search, datapath will compare each line's tag to the input address, if there is a match, it will check if the valid bit of the matched line is set, if so, datapath will signal that the data is ready for reading and send the data to the core.
If none of the lines matched, datapath will signal back to the control that there was a miss, in return, he should get the data from memory and locate it in a line which Algo determine. After the data is in the table, as before, datapath will signal that the data is ready

5.1.3 Algo

Algo is responsible to determine which line is the next line to replace. I will implement this component with several algorithms that will be discuss and investigate later in this project.

Operation

When a Miss occurred, the control will signal it to Algo, Algo will replay **immediately** which line to replace and only then start to calculate the next line to replace, to be ready for next time.

5.2 Data Cache

The main difference between Instruction cache and data cache is that data cache can write and not read-only. This change influences the control and DP components but not Algo.

5.2.1 Write Policy

In the case of a write, there are two main policies used by the industry:

- *Write-through*: write is done synchronously both to the cache and to the next stage memory.
- *Write-back*: initially, writing is done only to the cache. The write to next stage memory is postponed until the modified content is about to be replaced by another cache block.

In this project I implement the cache write policy slightly different, somewhere between **write-back** and **write-through**.
I will define a parameter of maximum dirty lines allowed, such that when there are more dirty lines than the parameter, the cache will start to write the blocks to the memory in background update(when the memory not in use). One can edit this parameter to go anywhere between write-back and write-through, i.e.:

- If max_dirty = 0 , the cache is write-through
- If max_dirty = #blocks , the cache is write-back

Note: This is not exactly write-through or write-back cause the cache trying to do background update.

Another addition looks two and three replace ahead and if those lines are dirty, the cache also try to write them to memory.

5.2.2 Control

The control is pretty much the same as the **Instruction cache**'s control but in addition, now it also manages memory write option.
Memory write occurs in two cases:

1. Datapath replacing dirty line.
2. There are more dirty lines than **max_dirty**.

Write to memory take at least one cycle so we try to write only in the second case when memory unused, if it happened and we in the first case, the cache first supply the data to the core and only then write-back the data to memory.
In this cache, we decide to enable write in the resolution of bytes, thus store-word, store-half-word, and store-byte is allowed.

5.2.3 DataPath

When control request data, DP search for the address in its table, if hit occurred, the control will notify the core that the data is ready for read/write. When a miss occurred and needs to replace an old line by

new one we must check first if there has been write to this line, that made by checking the line's dirty bits, we have bit per byte.
The decision of which line is still determined by Algo, DP check if the line's dirty bit is set if so, first he will write the line's data to memory and then insert the block into the cache. Now DP signal to the control that the data is ready and wait for the next request.

5.3 Cache line

The structure of a line is an important issue when working with caches. Cache line contain:

- Data - # WORDS_IN_BLOCK x 32b.
- Valid bit - Set if the data in the block is valid.
- Tag - The real address (in the main memory) of the block. Holds the higher bits not include offset.
- Dirty bits - Set if the data has been changed by the core, need to update main memory. Bit for every byte in the block.

Size of cache line in the specific implementation:

- The data - $\underbrace{4}_{\substack{number\\ of\ words}} \times \underbrace{32}_{an\ word} = 128b.$
- A valid bit - 1b.
- Tag - $\underbrace{32}_{32b\ address} - \underbrace{2}_{word\ offset} - \underbrace{2}_{byte\ offset} = 28b$
- A dirty bit - $\underbrace{4}_{\substack{number\\ of\ words}} \times \underbrace{4}_{\substack{bytes\ in\\ each\ word}} = 16b$

In total, 173b for each cache line.

5.4 Mapping principles

Our cache built on the principle of **full associative**.
A fully associative cache contains a single set with B ways, where B is the number of blocks. A memory address can map to any block in these ways. A fully associative cache is another name for a B-way set-associative cache with one set, it is as opposed to *direct map* (1-way associative) where there are B sets with 1-way each. In simple words that mean that any line from main memory can be mapped into any line in the cache, which is, in contrast, to *direct map* that any line from main memory can be mapped to only one specific line in the cache.

5.5 Replacement policies

As explained before, in chapter 3.4 - Replacement policies meant to decide which line is next to evict.
I started with simple replacement policies – FIFO. I implement it with a counter register, when a cache miss occurs we replace is line #<counter> and then increase counter by one.
As mention in chapter 5.2.3, in the case of data cache, before we clear the line we will check if there has been a write operation to this line by checking the **dirty** bits, and if a write operation has been done, we will write it to main memory first and only then replace the line.

6. Replacement Policies

Before I start to analyze different algorithms, we first need to define **the optimal algorithm** so we know how to measure the performance of different Algorithms.
I will also explain the **locality principle** that is the main principle that we are relying on when trying to improve performance.

6.1 The optimal algorithm

The optimal algorithm uses the cache in an optimal way, its policy is to replace the line that will not be used for the longest time.
Consequently, it must know all the future accesses but unfortunately this is impossible to implement because we don't know what lines will be used in the future. Only simulations on predefined memory patterns can be carried out. As a result, it only appears as a reference to measure the performance of other replacement algorithms.

6.2 The locality principle

Since processors are not omniscient, a method predicting the next accessed data, needed to try to reach the performance of the optimal algorithm. To that end, replacement policies resort to the principle of locality.
Programs tend to reuse the data and the instructions that they have used recently (e.g. loops): this is the **temporal locality**.
Moreover, a program tends to use the data and instructions that are located in the vicinity of the used one, this is the **spatial locality**.
Studies show that on average a program spends 90% of its execution time in 10% of its code. As a result, we can reasonably predict the next data from the previously accessed ones. Replacement algorithms try to take advantage of the locality principles to be as near as possible to the optimal replacement policy.

6.3 Hardware implementation

The logic of an algorithm is a nice thing but implement the algorithm on hardware is a completely different thing.
FIFO implementation is pretty easy, however, LRU implementation is a lot more complicated, I'll discuss my implementation in the next

sections, here I want to introduce the common problem with LRU implementation.
LRU algorithm is complicated because we suppose to know all the time who is the Least Recently Used cache block and be aware to changes. Following every cache request is needed and this is a hard thing that expresses in a lot of hardware and computations, we will look on several common implementations:

1. Counters

   Use N counters of log(N) bits, in every cache request we update all the counters, some we increase by one and some stay the same.

   Disadvantage:

   - Power consuming – all counters shift every access
   - Computation – required two depended compares for address match and index match
   - Expensive in hardware and computation

2. Linked list

   Use N nodes such that the LRU place at the end of the list, in every cache request we need to go through all the nodes, find where is the used cache block and update his location.

   Disadvantage:

   - Power consuming – full list shift every access
   - Computation – required two depended compares for address match and index match

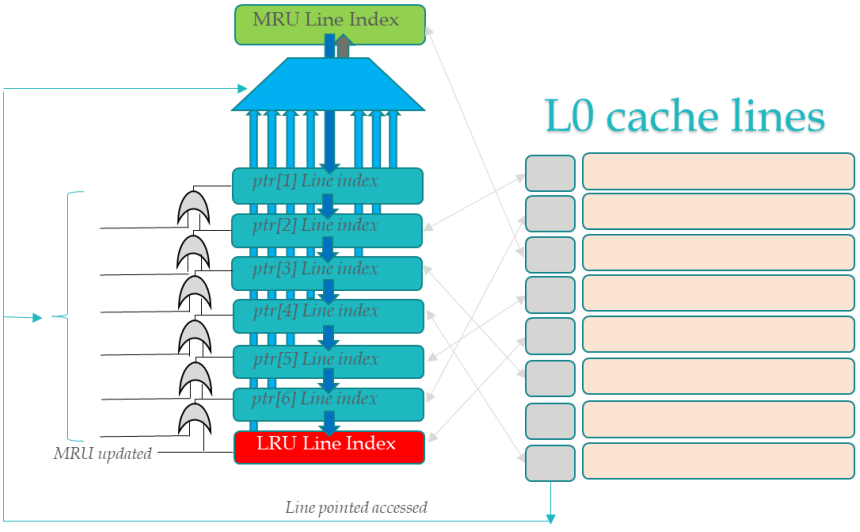


Figure 6-1: hardware implementation of Linked list

3. Clock

The pointer of the next block to replace go in rounds like a clock and mark blocks that have been used more than once.

Disadvantage:

- Implement approximate LRU
- Computation – all computation depended on one another

6.4 FIFO – First In, First Out

This algorithm is the simplest, it works like a queue, the first that goes in it is the first that goes out.
To implement this, we set a counter register that counts every cache miss while doing modulo of the cache's number of lines, the counter's number is the number of the next line to replace. For instance, when the first data will come, the algorithm accommodates the new data in the first line, then the next data in the second line and so on. When all lines are filled with data, the next line to be accommodated will be the first line again and so on.

6.5 LRU – Least Recently Used

The implement of LRU is complex then FIFO. In this algorithm, we need to keep track of the cache's use even when there were a hit because we want to know which of the cache lines is the least recently used (LRU) and choose this line to be the next to replace.
To do so I use the next technique:

I build a squares matrix of NxN where N is the number of cache lines.

Every time that cache line has been used Algo update the matrix as follows:

1. Fill the row of that line number with ones
2. Fill the columns of the line number with zeros

In all time there is at least one line of zeros, the line of zeros is the least recently used.

For example we have the next case:

The LRU line in each stage marked with gray and the operation in blue.

First, we have four misses, so we fill the buffer by order:

$$\begin{pmatrix}0&1&1&1\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\rightarrow\begin{pmatrix}0&0&1&1\\1&0&1&1\\0&0&0&0\\0&0&0&0\end{pmatrix}\rightarrow\begin{pmatrix}0&0&0&1\\1&0&0&1\\1&1&0&1\\0&0&0&0\end{pmatrix}\rightarrow\begin{pmatrix}0&0&0&0\\1&0&0&0\\1&1&0&0\\1&1&1&0\end{pmatrix}$$

Next, we use the next sequence, were there only hits:

$1 \rightarrow 0 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \rightarrow 2$

Look as the zero line changes to 'hold' the LRU line

$$\begin{pmatrix}0&0&0&0\\1&0&1&1\\1&0&0&0\\1&0&1&0\end{pmatrix}\rightarrow\begin{pmatrix}0&1&1&1\\0&0&1&1\\0&0&0&0\\0&0&1&0\end{pmatrix}\rightarrow\begin{pmatrix}0&1&0&1\\0&0&0&1\\1&1&0&1\\0&0&0&0\end{pmatrix}\rightarrow\begin{pmatrix}0&1&0&0\\0&0&0&0\\1&1&0&0\\1&1&1&0\end{pmatrix}$$

Here are two suggestions for hardware implementation of this replacement policy.
The first one based on static logic and the second one based on dynamic logic:

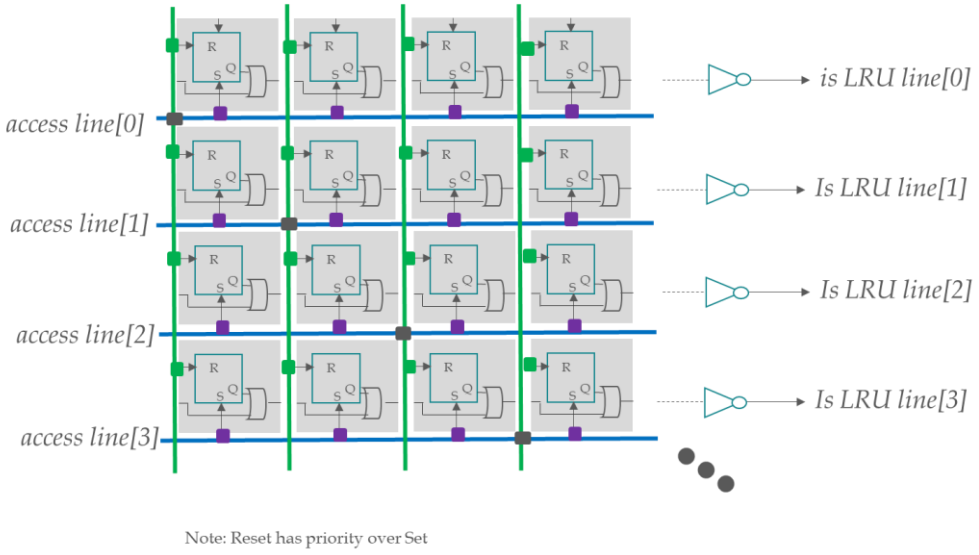


Figure 6-2: LRU square matrix physical implementation, static logic

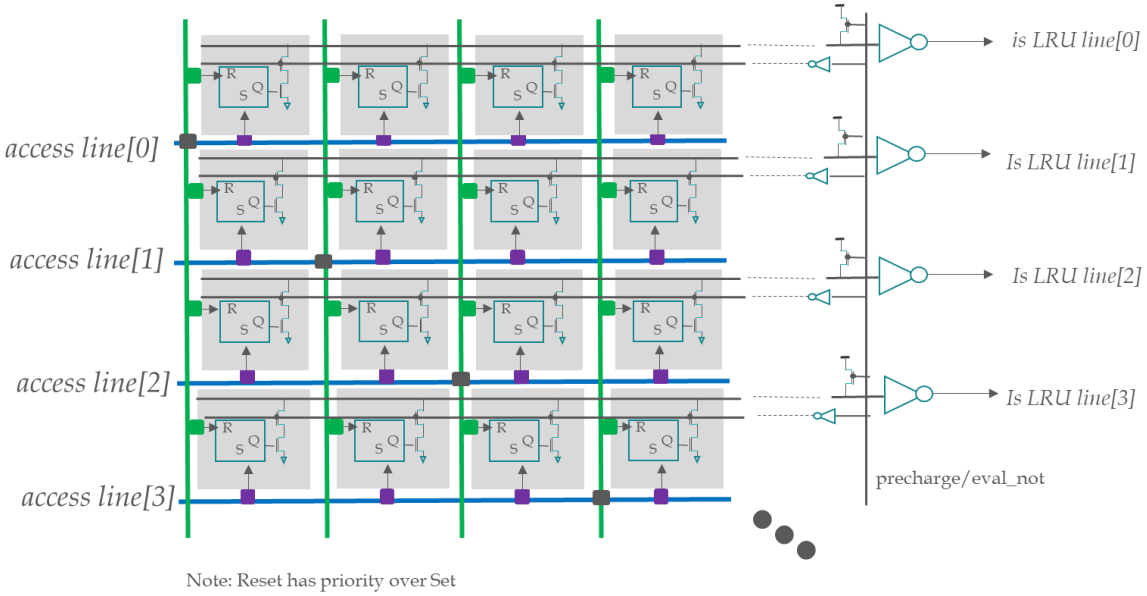


Figure 6-3: LRU square matrix physical implementation, custom dynamic logic

Another option for implementation is as follow:

Use matrix $(N-1) \times N$, in every use of line X update the matrix as follow:

- For each line Y in the matrix
  - If (line X == 0)
    - Shift left line Y by 1
  - Else If (line Y value < line X value)
    - Shift left line Y by 1
- Line X = 00..001

We look at the same example:

The LRU line in each stage marked with gray and the operation in blue.

First, as before we have four misses, so we fill the buffer by order:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Next, we use the next sequence (same as the previous example), were there only hits:

$1 \rightarrow 0 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \rightarrow 2$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We got the same results as previous, the marked line is LRU line.

7. Block diagram

In this part, I will present the Cache in a visual way for better understanding.

In figure 7-1 we can see the general location of the cache in a system and the connection between the process unit and the next level memory unit to the cache.

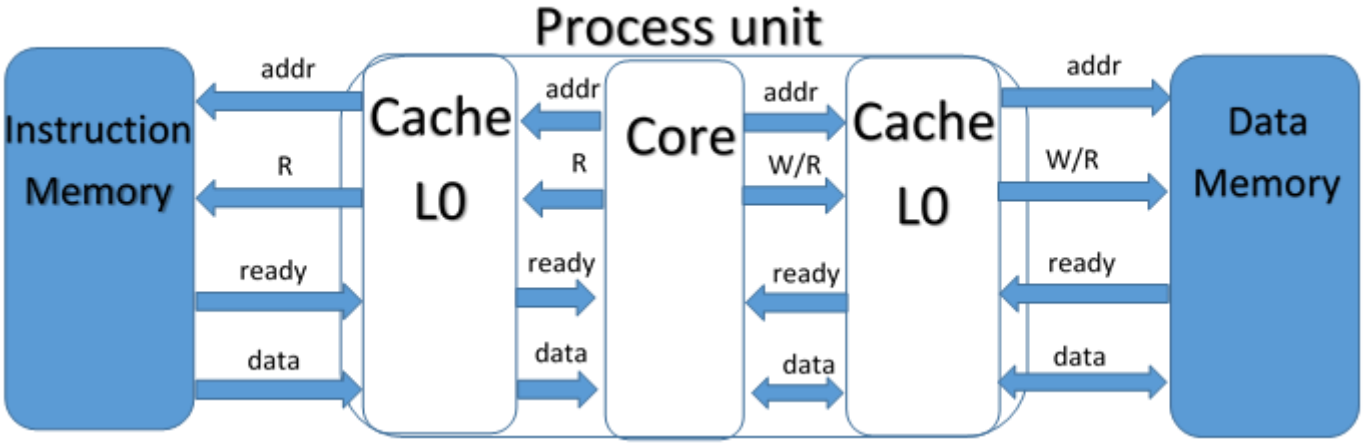
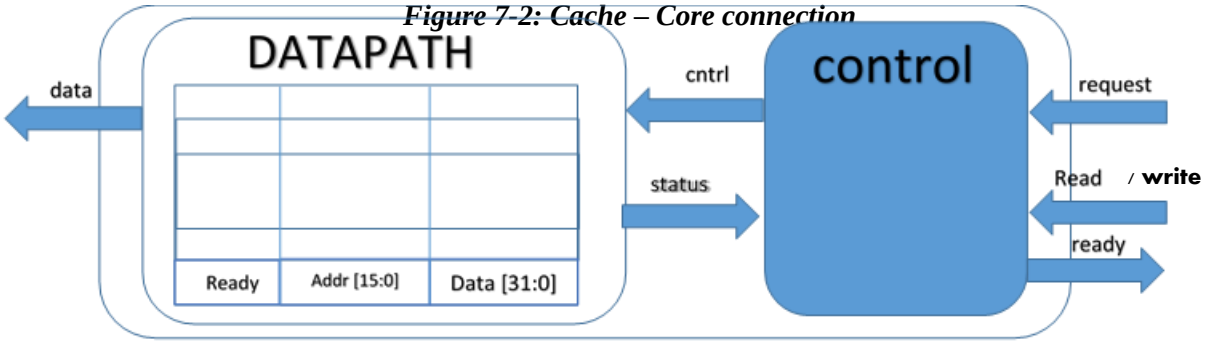


Figure 7-1: CPU – Memory connection

The interaction between the cache and the different units affects the interaction between the units of the cache itself.
In figure 7-2 we can see the interaction between the core and the cache and the effects of this interaction

Figure 7-2: Cache – Core connection



Also the interaction and effects between cache and Memory

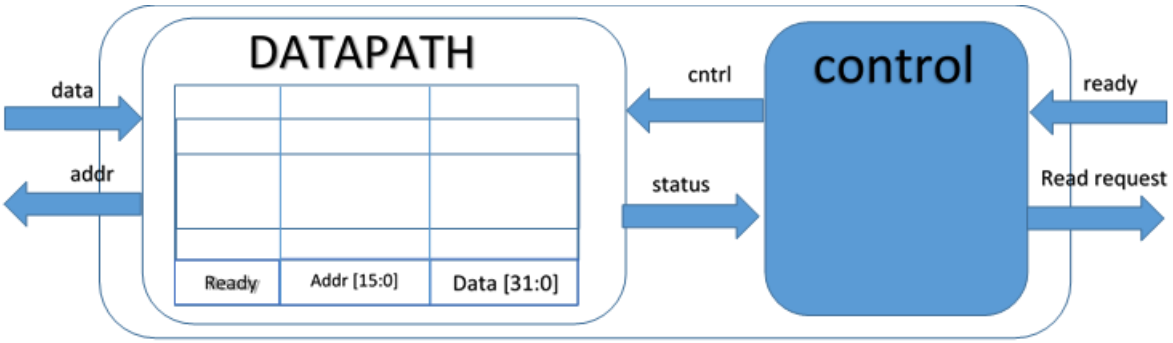


Figure 7-3: Cache – Memory connection

Finally, this is the general structure of our cache, here is all I mentioned above, in previous sections.
Start from right, we see the **Datapath** unit includes data cells and indication bits i.e. tag (line address), valid bit and dirty bit for each line. Below are the comparison and select operation.
On the left is the **Control** units that control the whole system and its interaction to Datapath and **Algo** unit that responsible for determine the next line to replace.

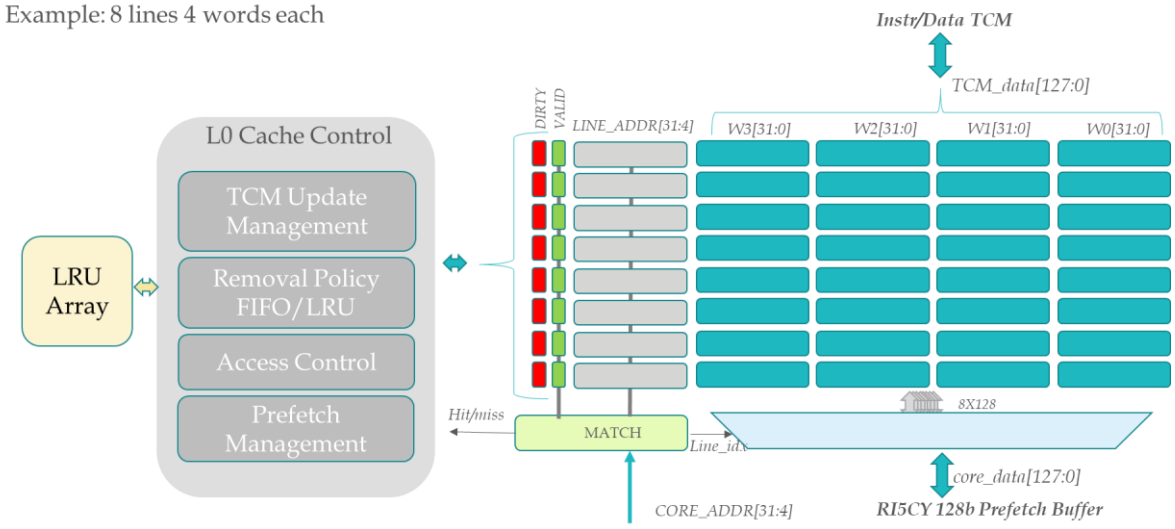


Figure 7-4: Cache structure

The cache will implement on the Pulpenix chip, here is the chip structure and the implementation of the cache inside it.
We will concentrate on the Core Region (in the red circle)

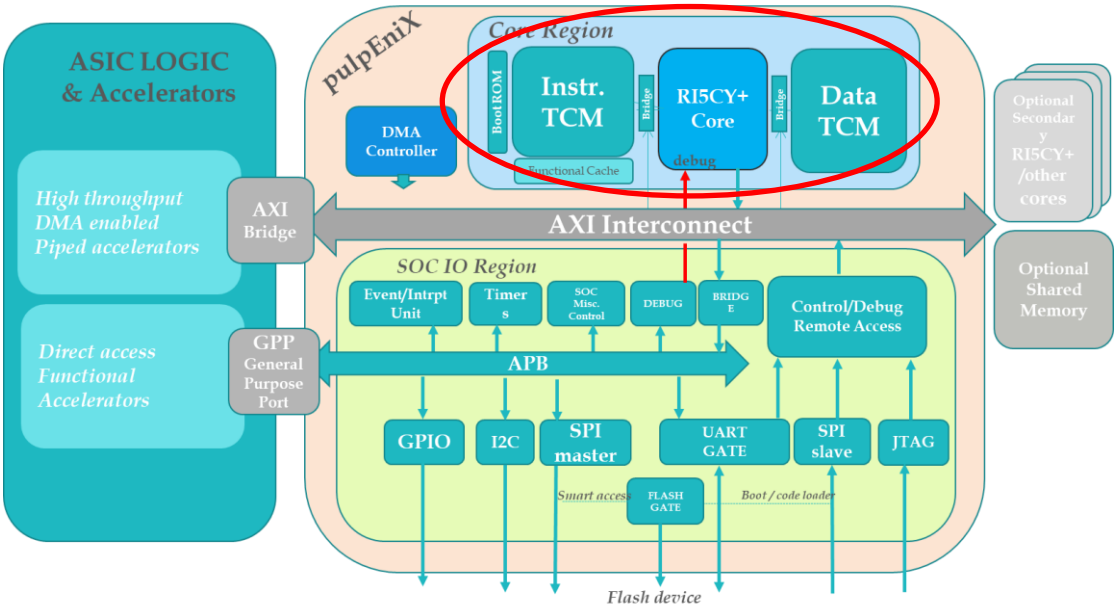


Figure 7-5: Pulpenix structure

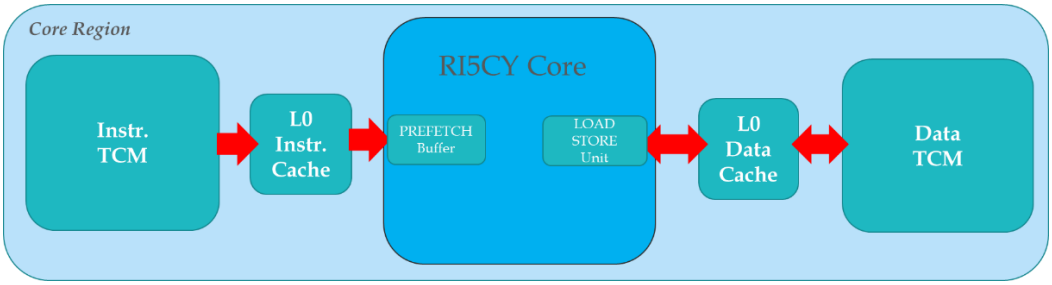


Figure 7-6: Cache implementation on pulpenix

8. Verification

To verify our design we need to do so carefully with a lot of attention to the details so we will break it down to pieces:
The verification will split into two main parts:
Part I – logic check:

- First, testing the modules as standalone.
- Second, combination of DataPath and Algo modules together.
- Finally, combine the whole three modules and check them together as cache.

Note: Those tests were with arbitrary data.

Part II – performance check

- We will start by check the cache as one unit with typical code from core mark check
- The next stage is to check the cache as a unit in a complete system.
- Synthesis – check that the cache is synthesizable and have a reasonable timing.

To check the performance I will run the same code on the optimal algorithm and compare its results to our cache results.

Note: I will start by check the instruction cache and only afterward I will check the data cache.

8.1 Part I – logic check

8.1.1 Standalone modules

8.1.1.1 Datapath

I start by insert an address to the module and signal it to start search. As expected to module return **miss**, then I signal him to import new data in this stage I insert to the module data (128b) and which line to replace. The module inserts the new data to its table and updates relevant bits – valid, dirty and tag. Then the module updates the output and signals beck by set **data_ready** bit.
I try to search again the same address but this time **DataPath** gets a **hit** and signal beck and **data_ready** immediately.
In the next step, I insert several entries with different - addresses, offset, data and line_to_replace, everything works as expected.

8.1.1.2 Algo

To test the Algo module I insert enable signal and toggle it, as expected output return immediately and after it, the module starts to calculate the next output. Obviously, I also check that the calculation it right and it returns the right line to replace every time.
I have done this check for each of the algorithms.

8.1.1.3 Control

I insert the module request signal and let him start work, when the module passes between states and signals me back I responded to it with signals that imitate the DataPath and Algo modules.
The test was to check that it moves to the right state and response with the right signals according to the input signals.
The test passed and the module works as it should.

8.1.2 Combine Datapath and Algo

In this test, I inserted address to DP and supply the control signals to both DP and Algo and let them do their work, Algo chooses which line to replace and DP accommodate them in its table. I insert words until the memory (the table) filled up, then I continue to add a couple of words to see what happens when there is overflow and they should override on the existing memory according to the replacement policy (FIFO or LRU). From time to time I try to read some addresses that are already in the memory and check that the data I got as a response matched to the respective data I inserted earlier.

8.1.3 Combine whole three

In this test I gave the module the environment and the connectivity, I signaled to the control **request**, then I passed his signals to DP and Algo modules and return their signals back to the Control or each other.
The whole three functioned as they should.

8.2 Part II – Performance Check

8.2.1 Cache Verification

To check the whole system as motioned earlier it little bit complicated. First, we checked that the cache works well as a standalone unit now I test him against core and memory with a typical program's code that we bring from the coremark test.
We build the next test bench:
The core sends his signals to the cache:

- en_i - request
- addr_i - address
- wdata_i - data to write
- we_i - R/W
- be_i - which byte to write

And expecting to get back:

- rdata_o - the data for the request
- rvalid_o - the data is ready

The signal that the core sends, is address from the coremark code and signals that it generates for the specific instruction, e.g. for instruction memory it was all the time as follows:

- en_i - 1 when it wants the next instruction
- addr_i - address
- wdata_i - 0
- we_i - 0
- be_i - 0

To check the cache in a more complete way we create a memory unit that will supply the data to the cache. we initial the memory with data from the core mark file.
Now the cache can interact with the memory and core without my intervention.

To make the test bench more robust we need to turn it self-checking so in the TB read address and data from the same core mark file, the address is the address that the core sends to the cache. Every time that **data_ready** is set, the TB compares the data from the cache to the data from the file, if they differ, he wrote it to the error file, the TB read next line of address and data and so on.

To understand the TB better I illustrate it in the next figure:

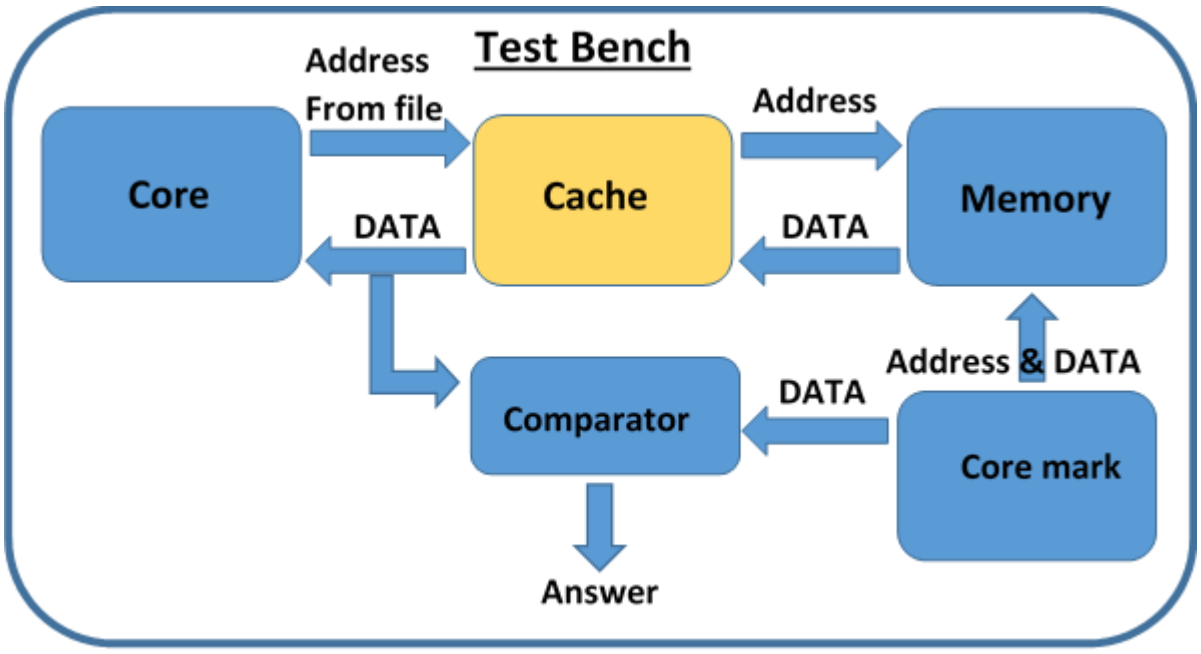


Figure 8-1: Cache test bench

8.2.1.1 Results and Improves

From analyzing the results, we can see that logically all three operates as they should.
In performance, we see that it takes 11 cycles from **request** until **data_found**.

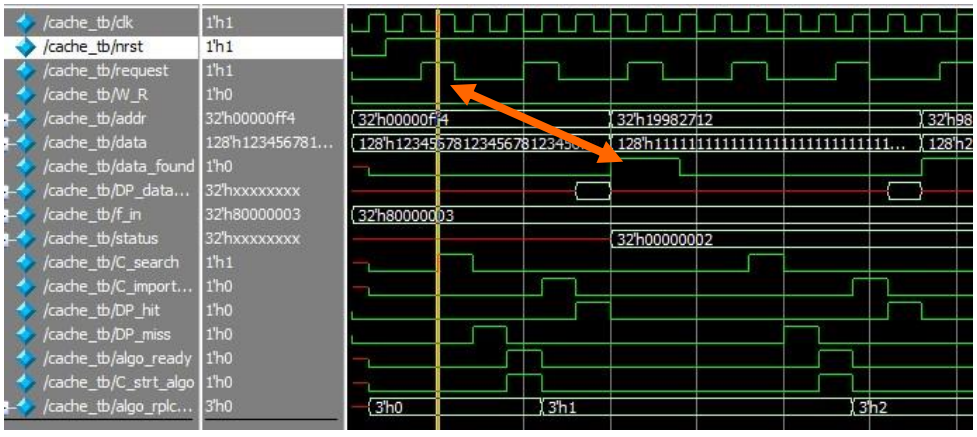


Figure 8-2: cycles from request to data ready I

This is too slow because it is the same speed as cache L2, to achieve the minimum clocks it is not a simple task so for decrease cycles I try a few changes:
First sketch - I switch the clock of the control to be opposite to Algo and DP then I got only 8 cycles.

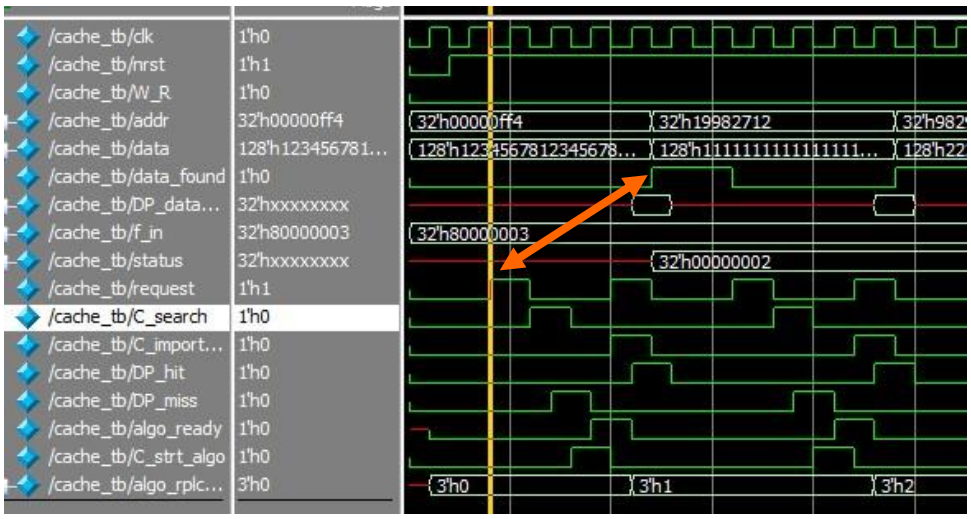


Figure 8-3: cycles from request to data ready II

Second sketch - the idea of verse clock is a little problematic so I change control in a different way, instead of divided the signals to Algo and DP to separate clocks I tried to combine them together to operate in the same clock in the limits of system's logic, e.g. signals **miss** (to Algo) and **data_import** (to DP) combine to the same cycle. I manage to decrease the cycles to only five.

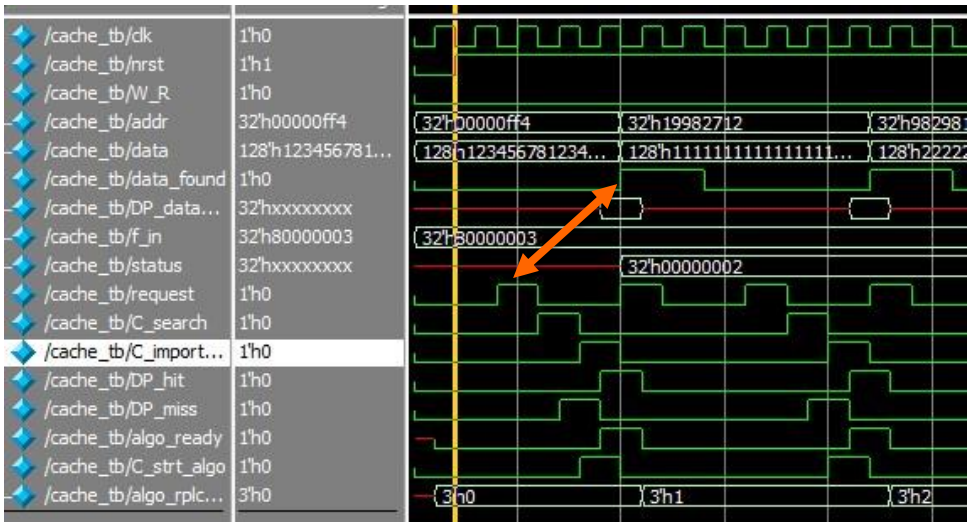


Figure 8-4: cycles from request to data ready III

Third sketch - this is still slow so I have done the next changes:

- Improve the control response time by making it Mealy FMS
- Improve DP to be more a-synchronic
- Make Algo respond immediately and make its calculation offline

I succeed to improve the speed and got the next results:

- 1 cycle per request for Hit
- 2 cycles per request for Miss

As shown in figure 8-5

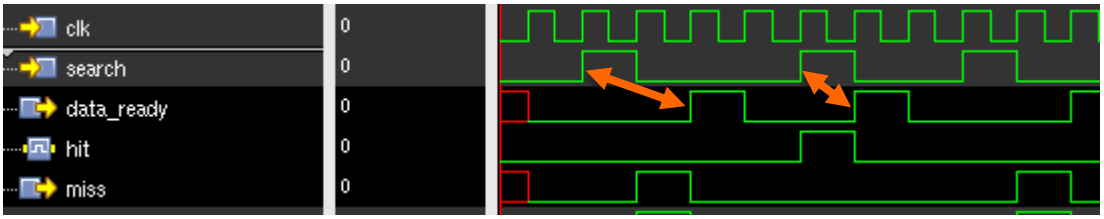


Figure 8-5: cycles from request to data ready IV

Till now it sounds great, but what is this mean to us?

With no statistic of the occurrence of hit and miss, this tells us nothing.
We check the statistic and got **~84%** of hit rate, this means that 84% of the time, request for data will be answered in one cycle (!) this sounds great but more on that in the result section.

In figure 8-6 you can see a sample segment of hit and miss in the coremark program.

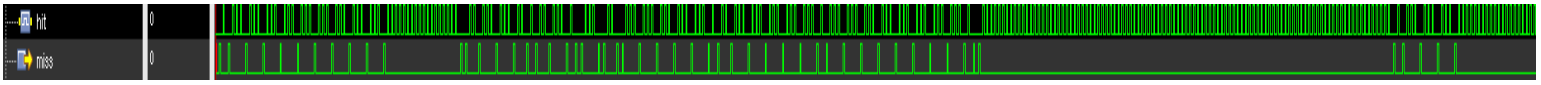


Figure 8-6: sample segment of coremark program

8.2.2 System Verification

This probably the most interest test we can do to the cache.

In this test, I planted the cache into a real chip, the chip is the **Pulpenix** from SOC Labs, based on Risc-V architecture.

I integrated the cache into the core and run several programs on the system to see it work properly. When everything will works well, for

the final test, I will run coremark program to see the cache performance, then compare the origin coremark to the new coremark, here we want to see minimal impact on performance and lightly worse when using memory at half-rate.

There are two places that we can integrate the cache:

1. Out of the core – between the core and the TCM
2. In the core:
    a. Instruction cache – replace buffer_L0 in prefetch_buffer_L0
    b. Data cache – between the core and the LSU

In the project I integrate the cache out of the core, I have done this in three stages:

1. Only instruction cache
2. Only data cache
3. Integration of the whole system – both instruction and data cache

8.2.2.1 Results

First to implementation of the cache in the chip demand some adaptation according to the cache's memory protocol. After the arrangement I have run the tests on the chip, the program completes the run without errors, in figures 8-7 and 8-8 we can see some of the results.

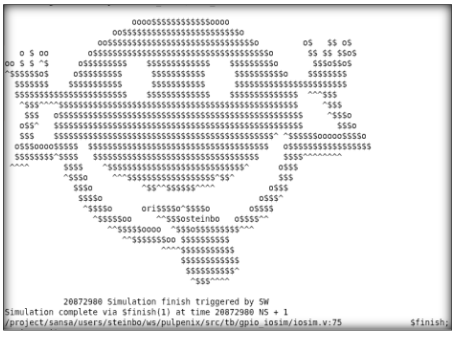


Figure 8-7: helloworld program's output

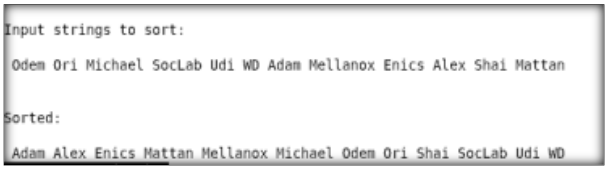


Figure 8-8: bubblesort program's output

The system works properly, now we check the performance on coremark code.
At full-rate we manage to done no harm to the performance what so ever, in half-rate we manage to do minimal impact as show in the next table:

Table 8-2: CoreMark results

	Instruction & data mem FR	**Instruction mem HR**	**Data mem HR**	**Instruction & data mem HR**
No cache at all	2.7CM			2.07CM
Instruction cache only	2.7CM	**2.63CM**	2.43CM	
data cache only	2.7CM	2.26CM	**2.57CM**	
Instruction & data cache	2.7CM			**2.51CM**

We can see the improvement when using the cache.

Synthesis – the cache pass the synthesis and got timing about ~2.2-2.4ns.

9. Summary and Results

In part one we implement the three modules (control, datapath and Algo) and check them on logic tests in three stages:

- Separately.
- Algo and DP together.
- And finally all three together.

In the second part, we check the whole system and check also the performance of the component.
We succeed to achieve the performance of:

- 1 cycle per request for Hit.
- 1-2 cycles per request for Miss (depend if write to cache is also needed).

At the beginning of the book I mentioned how we can measure the cache performance, to do so we measure the performance of the optimal algorithm, in addition to our two algorithms, in order to compare its performance to the performances of our algorithms.
The results are shown in the next table:

Table 9-1: Algorithms performance

Algorithm / Words per line	Instruction			Data		
	2 words	4 words	8 words	2 words	4 words	8 words
FIFO	78.4%	87.3%	92.2%	72.5%	72.7%	69.4%
LRU	78.3%	87.3%	92.3%	73.8%	74.8%	74.5%
Optimal	85.6%	91%	93.9%	82%	84.4%	81.3%

We can see that our algorithms did not reach maximum performance, as expected, but also not so bad.
In the case of instruction memory we able to get good results but in the case of data memory, we must check other algorithms.

Finally, to understand the improvement of the cache we need to calculate the AMAT:
AMAT = Average Memory Access Time.
Furthermore, to easier the understanding, we will assume the next memory level's AMAT is 4.

$$AMAT = Hit\ time + miss\ rate \cdot penalty$$

$$AMAT = Hit\ time + miss\ rate \cdot (cache\ L0\ miss\ penalty + cache\ L1\ AMAT)$$

$$AMAT_{Instruction} = 1 + 0.08 \cdot (1 + 4) = 1.4$$

$$AMAT_{Data} = 1 + 0.25 \cdot (1 + 4) = 2.25$$

We have achieved great improvement compare to AMAT of 4 without using the Cache.

10. Future Research

There are many possible ways to continue this research, in my opinion, the main ways for future research should include several issues:

10.1 Cache

- Evaluate more testbenches beyond coremark
- Implement the cache into the Pulpenix core.
- Implement on specialize systems, e.g. system for matrix multiplication.
- Test different size configurations.
- Co-operate with other researches and extensions

10.2 Replacement Policies

- Enhancing prefetch prediction for instruction and data caches
- Synthesis second implementation of LRU
- Explore more algorithms beyond FIFO and LRU

11. Glossary

Table 11-1 defines the acronyms used in this document.

Table 11-1: Acronyms

Term	Definition
TB	Test bench
DP	Data Path
FIFO	First in, first out
LRU	Least recently use
TCM	Tightly coupled memory
AMAT	Average memory access time

12. Bibliography

Computer memory Architecture:

1. *Memory hierarchy* - *Link*
2. *The structure and work of the cache* - *Link*
3. *Expended on memory hierarchy and algorithms* - *Link*

Cache's replacement policy and algorithms:

4. *Algorithms* - *Link*
5. *Expended on Algorithms (thesis)* - *Link*
6. *Replacement policies* - *Link*

General understanding about Cache:

7. *General explanation for the purpose of cache* - *Link*
8. *Wikipedia – Cache* - *Link*
9. *YouTube series on cache* - *Link*