



**Design and Implementation of a Multi-Core Processor Using FPGA**

A dissertation submitted to Coventry University for the degree of Bachelor of Engineering in the faculty of engineering, environment and computing

By

**Mr Matteo Bovino**

SID: 8671055

Supervisor: **Dr Server Kasap**

Academic Year: 2020/21

**Abstract**

This dissertation discusses methods to increase throughput and performance in modern computing, through the study of an FPGA-based prototype of a dual-core processor developed in the Vivado software environment. In the document, various considerations around execution time, cost and complexity are laid out in order to justify the decisions made during the creation of the processor. The objective is to showcase the performance benefits arising from the utilization of the dual-core design, while also highlighting the challenges facing parallel computing as a whole. Given the complexity of the subject, not all paths by which modern CPUs improve throughput have been pursued; the focus was instead on the key topics that have paved the way for the rapid growth of computational power. Some of these include methods to boost single-core, memory and I/O performance. To develop the individual stages and consequently the whole processor, SystemVerilog source code has been written from scratch, with the only exception of the main memory IP core. The dissertation concludes with a summary of the findings and a comparison between single and dual-core performances under multiple test programs. As expected, the multi-core design has better characteristics, however, they lack linearity and stability.Duplicating the hardware resources does not in fact equate to double performance. The improvements are tightly dependent on the software and program, which can negatively overshadow most parallel hardware capabilities. The programs used for testing have been carefully selected in order to expose some of these critical scenarios, but also to demonstrate the overall benefits obtained from multi-core processing.

**Acknowledgements**

I would like to express gratitude towards my supervisor Dr Server Kasap, who has not only looked after every stage of the project and dissertation, but has also been supportive of my applications relative to engineering positions and further studies. His encouragement and sincerity have helped me to understand the challenges inherent to the field of electronics design, and how to navigate them. I’d also like to thank the development officer Nik Tsanov, for answering some of my technical queries throughout the duration of the project.

**Table of Contents**

[List of Figures v](#_Toc70478378)

[List of Tables vi](#_Toc70478379)

[List of Abbreviations vii](#_Toc70478380)

[**Chapter 1 - Introduction 1**](#_Toc70478381)

[1.1 Background Context 1](#_Toc70478382)

[1.2 Structure of the Report 1](#_Toc70478383)

[1.3 Aims and Objectives 2](#_Toc70478384)

[**Chapter 2 – Literature Review 3**](#_Toc70478385)

[2.1 CPU 4](#_Toc70478386)

[2.2 Memory 6](#_Toc70478387)

[2.3 ISA 8](#_Toc70478388)

[**Chapter 3 - Pipelined Processor 9**](#_Toc70478389)

[3.1 Fetch 9](#_Toc70478390)

[3.2 Decode 10](#_Toc70478391)

[3.3 Execute 13](#_Toc70478392)

[3.4 Memory Access 14](#_Toc70478393)

[3.5 Write Back 16](#_Toc70478394)

[3.6 Forwarding 16](#_Toc70478395)

[3.7 Predicting 17](#_Toc70478396)

[3.8 Elaborated Design 1 19](#_Toc70478397)

[**Chapter 4 - Dual-Core Design 20**](#_Toc70478398)

[4.1 Duplication Process 20](#_Toc70478399)

[4.2 Data/Instruction Cache 20](#_Toc70478400)

[4.3 Hazard Unit 21](#_Toc70478401)

[4.4 Main Memory 22](#_Toc70478402)

[4.5 Cache Coherency 23](#_Toc70478403)

[4.6 Elaborated Design 2 24](#_Toc70478404)

[**Chapter 5 – Practical Attainments 25**](#_Toc70478405)

[5.1 Fibonacci Sequence 25](#_Toc70478406)

[5.2 Additions and Multiplications 26](#_Toc70478407)

[5.3 Matrix Multiplication 27](#_Toc70478408)

[5.4 Critical Appraisal 29](#_Toc70478409)

[**Chapter 6 – Conclusion 30**](#_Toc70478410)

[**Chapter 7 – Future Work 30**](#_Toc70478411)

[**Chapter 8 – Student Reflections 31**](#_Toc70478412)

[List of References 32](#_Toc70478413)

[Appendix 34](#_Toc70478416)

[A.1 Project Management 34](#_Toc70478417)

[A.2 ISA 35](#_Toc70478418)

[A.3 Logbook and Meetings Material 36](#_Toc70478419)

# List of Figures

[Figure 2. 1 Dram capacity growth (Patterson & Hennessy 2013 : 25). 3](#_Toc70478678)

[Figure 2. 2 Memory-Processor performance gap (Patterson & Hennessy 2013:73). 6](#_Toc70478679)

[Figure 2. 3 Cache Hit Rates (Kumar & Singh 2016:4). 7](#_Toc70478680)

[Figure 3. 1 PC selection. 9](#_Toc70478681)

[Figure 3. 2 General five-stage MIPS processor (Mahmood & Omran 2014 : 39). 11](#_Toc70478682)

[Figure 3. 3 Register file code. 12](#_Toc70478683)

[Figure 3. 4 MULT/DIV logic. 13](#_Toc70478684)

[Figure 3. 5 Load scenario. 14](#_Toc70478685)

[Figure 3. 6 Store scenario. 15](#_Toc70478686)

[Figure 3. 7 Forwarding and dependencies (McQuain 2013). 17](#_Toc70478687)

[Figure 3. 8 BHT. 18](#_Toc70478688)

[Figure 3. 9 Faster acting hysteresis. 18](#_Toc70478689)

[Figure 3. 10 Elaborated design 1. 19](#_Toc70478690)

[Figure 4. 1 Two-way set associative cache. 21](#_Toc70479184)

[Figure 4. 2 BRAM IP core. 22](#_Toc70479185)

[Figure 4. 3 MESI protocol. 23](#_Toc70479186)

[Figure 4. 4 FSM for the cache controller. 23](#_Toc70479187)

[Figure 4. 5 Elaborated Design 2. 24](#_Toc70479188)

**f**

[Figure 5. 1 Fibonacci result. 26](#_Toc70479387)

[Figure 5. 2 Additions and Multiplications result using a single core. 27](#_Toc70479388)

[Figure 5. 3 Additions and Multiplications result using both cores. 27](#_Toc70479389)

[Figure 5. 4 Matrix schematic. 28](#_Toc70479390)

[Figure 5. 5 Matrix multiplications results using a single core. 28](#_Toc70479391)

[Figure 5. 6 Matrix multiplications results using both cores. 28](#_Toc70479392)

**f**

[Figure A. 1 Original Gantt chart. 34](#_Toc70478700)

[Figure A. 2 List of instructions, excluding *halt*. 35](#_Toc70478701)

# List of Tables

[Table 2. 1 Pipeline diagram. 5](#_Toc70479502)

**f**

[Table 3. 1 Memory alignment. 16](#_Toc70479504)

[Table 5. 1 Loop for the Fibonacci sequence. 25](#_Toc70479506)

# List of Abbreviations

* ADSR – Attack Decay Sustain Release
* ALU – Arithmetic Logic Unit
* AMAT – Average Memory Access Time
* AXI – Advanced eXtensible Interface
* BHT – Branch History Table
* BRAM – Block Random Access Memory
* CMOS – Complementary Metal Oxide Semiconductor
* CPI – Clocks Per Instruction
* DLP – Data Level Parallelism
* DRAM – Dynamic Random Access Memory
* FPGA – Field Programmable Gate Array
* FSM – Finite State Machine
* GUI – Graphical User Interface
* IC – Integrated Circuits
* ILP – Instruction Level Parallelism
* IP – Intellectual Property
* ISA – Instruction Set Architecture
* LRU – Least Recently Used
* MESI – Modified Exclusive Shared Invalid
* MIPS – Microprocessor without Interlocked Pipelined Stages
* MIMD – Multiple Instructions Multiple Data
* MMIO – Memory Mapped Input Output
* PC – Program Counter
* ROB – Re-Order Buffer
* RTL – Register Transfer Level
* SLT – Set if Less Than
* SoC – System on Chip
* UART – Universal Asynchronous Receiver-Transmitter
* VLSI – Very Large Scale Integration

# Chapter 1 - Introduction

## **1.1 Background Context**

Since the first modern computers from the 70s until today, improvements in both hardware and software solutions have been a constant. This continuous growth has led to the digital revolution of our times, in which all “smart” devices benefit in one way or the other of these improvements. In particular, the hardware resources have multiplied over the years and laid the foundation for such technologies to exist. The intelligent usage of these components alongside the creativity of computer architects has been fundamental for their high functionality. Processors due to their central role in computing have been the subject of multiple upgrades over the years, moving from uniprocessors all the way to warehouse scale computers. This fast-paced progression allowed for widespread improvements in computational power throughout most digital devices. Judging from the outside, it may seem that this growth has been unstoppable ever since it started, while in reality now, just as in the previous eras, the architects are faced with new challenges that require innovative solutions to maintain the overall trend. Two recent problems in this sector have been the exploitation of parallel hardware and power efficiency in both processors and general ICs. These complex problems have not been solved yet, but new methods have been discovered to mitigate their impact on performance. The objective is to implement some of these techniques in a multi-core processor to highlight their benefits and downsides and mention other viable alternatives that are currently being developed. The final design can run on the high-density Nexys 4 DDR board with interconnections to monitors and keyboards to display the correct functioning of the processor.

## **1.2 Structure of the Report**

In this document the guiding line will be performance, the key parameter that has governed the world of computer architecture since the start. In the literature review all the main and influential ideas in this sector will be covered in a schematic and sequential order, with logical connections to the design choices for the project. Graphs, tables and citations will be used to give quick and intuitive representations of the trends that define the evolution of these fascinating systems. To design any IC, whether it is a complex SoC or a single uniprocessor, many factors need to be considered, trade-offs are a constant in this field and the architect must utilize its resources in the best possible way. The crucial design choices have all been enlisted in the Aims and Objectives subsection below.

## **1.3 Aims and Objectives**

1. Dual-core MIPS processor:

To develop a 32-bit dual-core processor possessing two independent MIPS cores, each capable of executing over 50 instructions, including integer, fixed-point and memory operations.

1. Pipelined:

To pipeline the critical data path up to 5 stages, including fetch, decode, execute, memory and write back. The limited number of stages will reduce the branch misprediction penalty and provide for a simpler design.

1. Hazard unit:

To connect a hazard unit to both data paths in order to detect data dependencies and other hazards including the structural and control types. Stalling being the primary method to handle hazards in the processor.

1. Instruction and data level parallelism:

To use ILP and DLP to increase performance of both cores. Parallel hardware can be best exploited by resolving hazards and finding data and instructions free of dependencies.

1. Caches with LRU replacement policies:

To implement one data cache per core, practically reducing the number of accesses to main memory. Both are expected to be two-way set associative with LRU replacement policies capable of storing 64 bytes.

1. Shared memory with MESI coherency protocol:

To communicate through shared memory segments in main memory, and to preserve the coherency and consistency of the data in the caches using the MESI protocol.

Besides the practical objectives, there are numerous theoretical goals attached to the project itself. For instance, to increase the general understanding of computer architecture and digital designs and to expand the previously acquired knowledge. Also, to become a more competent user of HDL languages, in particular SystemVerilog.

# Chapter 2 – Literature Review

The opening spark that has led to the fast-paced evolution of computer hardware must be traced to the manufacturing of transistors. Transistors are the key components in ICs, they are principally needed to implement the logic functions and combinatorial circuits. As ICs become more complex, more transistors are needed in order to implement the new features, thus the move from dozens transistors per chip to billions and now trillions (Cerebras 2019). As stated by Moore’s law, this growth is so incremental that resources are expected to double at least every 24 months. The following graph expresses the capacity increase in DRAM over the years, which can be directly attributed to transistor count and build process.

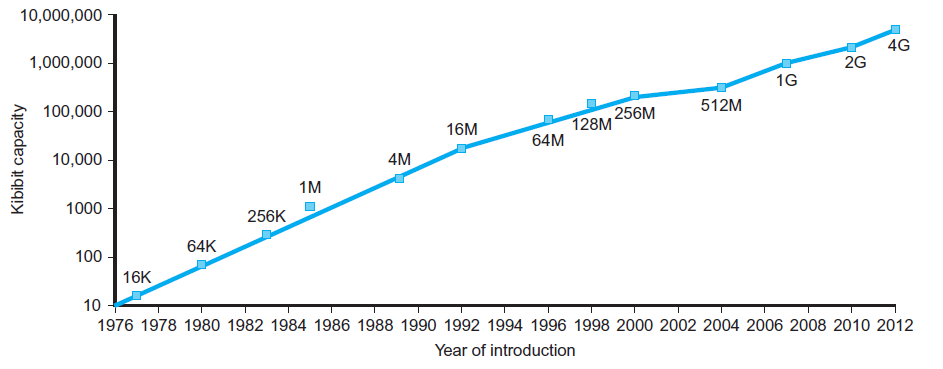


Figure 2. 1 Dram capacity growth (Patterson & Hennessy 2013 : 25). Although in the last few years the slope of the line has decreased, memory capacity is still doubling every two years.

With such a high rate of growth it would be natural to expect constantly increasing performance across every scope. Unfortunately, the boom of transistors per chip predicted by Moore’s law didn’t account for other related parameters which nowadays play a major role in computer efficiency. CMOS is the main technology used to build the logic circuits and while having many benefits such as high input impedance and simple gates, it has a crucial downside in its power consumption. It must be specified that this type of semiconductor has a great speed to power ratio, and that while it is held in a fixed state it consumes little power; however, a significant amount of energy is used when it changes state (0 to 1 and vice-versa). This energy, known as dynamic energy, can become an issue in VLSI systems where the number of transistors per chip is really high ([Kocanda](https://ieeexplore.ieee.org/author/37085448161) & [Kos](https://ieeexplore.ieee.org/author/37432091700) 2015). To avoid inefficiencies designers have gradually reduced the overall amount of voltage necessary to make these boards function, integrated circuit supply voltage (Vdd) is expected to reach 0.45V in 2030 (ITRS Report 2015:16).

Nevertheless, high power demands create huge problems from a battery life and heat dissipation perspective in all electronics, especially in embedded systems. Both issues must be addressed to avoid failures, therefore, new solutions are required to still increase performance, in light of the new power efficiency constraints known as the “Power Wall”. This state of being has made some question whether or not Moore’s law was still applicable in recent times, as stated in 2020 by Charles E. Leiserson, a pioneer in the world of VLSI and parallel computing “It's over. This year that became really clear”. While this dispute is out of the scope of the project, it shows the fundamental need for new ways to continue the overall trends, not just by relying on more resources. To analyse performance the following main parts of a computer will be covered:

1. CPU.
2. Memory.
3. ISA.

Improvements in these sectors would help to achieve better performances. Performance is a term familiar to all computer architects but there are multiple definitions available for it. However, for a computer to “perform”, ultimately, the most assertive value is time. In specific the amount of time required to complete a given set of tasks known as a benchmark. This value is often inversed, so that higher corresponds to better:

There are many factors that play a role in execution time, the main ones are instruction count, CPI, clock speed but also memory bandwidth, cache size and I/O datapaths. All these factors play a role in the overall performance and they can’t be ignored by simply focusing on CPU behaviour. On the other hand, it must be said that not all parts have equal relevance, as quantitatively expressed in Amdahl’s law.

## **2.1 CPU**

The central processing unit has the responsibility of completing instructions, conducting ALU operations and coordinating other components based on the instructions at hand. The time spent doing these operations is known as CPU time, which is a significant part of the overall execution time. This area of computing is so critical that it has its own law, known as Iron law**,** which can be summarized as:

The goal of having a smaller CPU time has led to many interventions on the parameters above. First of all, the number of instructions per program is directly connected to the ISA, compiler and algorithm; improvements over the years have been made to shrink down the code so that fewer instructions were required. The CPI while still dependent on the ISA is mainly a result of the processor design, in the following paragraph various techniques will be shown to better it. Clock cycle time can obviously be decreased with faster clocks, but it also depends on circuit design and transistor physics. What’s interesting to see is the trade-off between these parameters, less instructions introduce more complexity and hence a bigger CPI, while longer clocks cycle may reduce CPI to the detriment of clock speed.

Before diving into the parallel world, in the past, most CPU improvements happened while exploiting serial hardware, namely, pipelining and instruction level parallelism, which caused the birth of superscalars computers with CPI below 1. These techniques, while developed initially for uniprocessors with multiple functional units, remain present in modern day multi-core processors. As shown in the graphs below, the contribution of pipelining the processor and extending its functional units has been revolutionary.

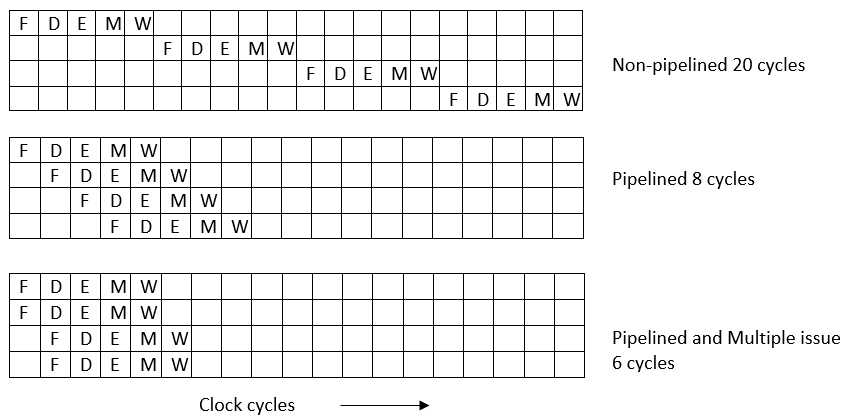


Table 2. 1 Pipeline diagram. Each letter indicates the initial for the name of the stage. Any vertical column represents a unique moment in time and corresponds to a single clock cycle.

This is clearly an ideal case scenario, with no hazards and perfect bypassing of the functional units, however it shows the speedup available in multiple-issue superscalars. The other and more recent drastic change occurred when parallel hardware started being used in the form of multiprocessors and multi-core processors. The idea of multiprocessing was present long before it’s commercial implementation, however due to its original cost and complexity, it was present only in high-end supercomputers (Ceruzzi 2003:30).

Instead, multi-core processors today are affordable options that can run parallel computing and multithreaded applications. These cores are processing units tightly dependent on each other, they share information through cache coherency protocols and always try to increase throughput and parallelism. The performance boost caused by adding an extra core to a uniprocessor is significant. For all the reasons mentioned above, the plan is to build a soft dual-core processor, with each core capable of ILP and pipelined to at least 5 stages; multithreading won’t be added due to its complexity and strict dependency to the OS.

## **2.2 Memory**

For parallel computing the memory system becomes a challenge. In order to benefit from all the advantages of parallel hardware, bottlenecks and failures must be avoided in the memory, otherwise simple stores and loads could cause a significant CPI loss. The distance in the performance gap between CPU and memory was already distinct in the last years of uniprocessors, but with the recent upgrades in processing units it has exacerbated:

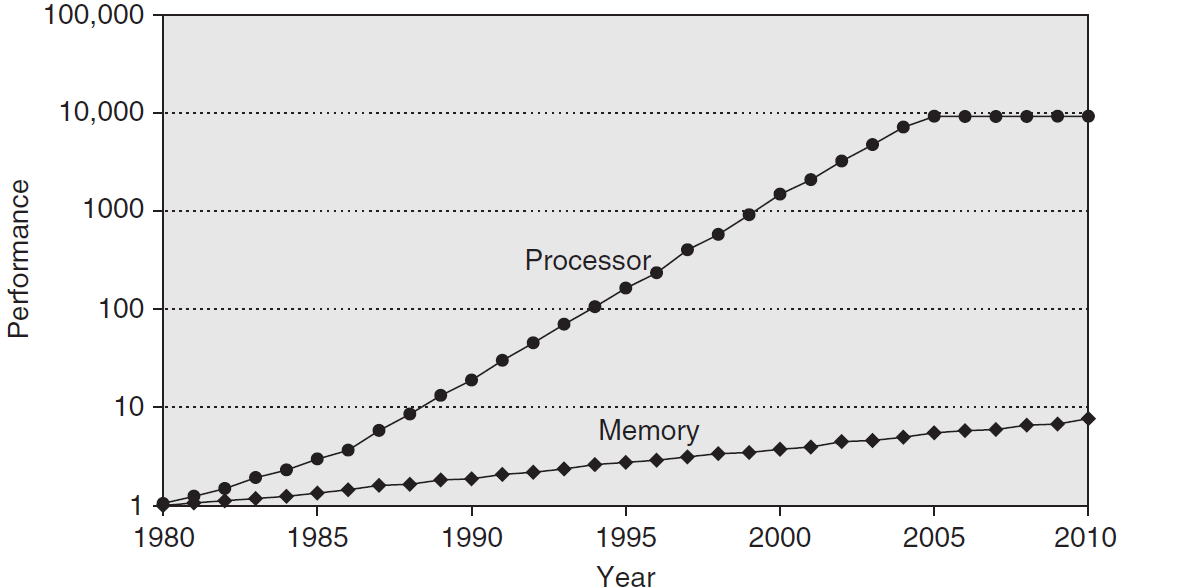


Figure 2. 2 Memory-Processor performance gap (Patterson & Hennessy 2013:73). While size for memory devices grew rapidly over time, their performance in terms of clocks needed per memory access is lagging behind.

Besides this gap, memory systems must also develop memory coherency protocols to allow individual cores to store and load data safely and consistently. Many ideas have been introduced, with two prevailing:

* Directory based, that uses a centralized location in which all the data is kept updated with information regarding its status (uncached, shared or exclusive/modified).
* Snooping based, in which cache controllers “snoop” on the common channels of communication with the intent of keeping their blocks of memory updated.

The issue of coherency has been a topic of academic discussions for years, with many new protocols developed, including MESI, Dragon and Firefly. Snooping based systems have been on a decline due to their high-power inefficiency; for a miss there are in fact power losses occurring in the locating, tag-checking and fetching of data that happen across all the other L1 caches. “The classical snoopy protocols are inefficient in the sense that they require all processors and related L1 caches to be active, as the snoop traffic might be received at any time” (Ahmed 2006:4). However, for a limited number of cores the power inefficiency is outweighed by the simplicity of the overall snooping mechanism. These caches are in fact much easier to design for fewer cores compared to the directory based alternative.

In modern memory hierarchy, multi-level caches are a must, their usage has allowed to hide the high latencies to main memory which were a big part of the performance loss, thus reducing the AMAT:

Nowadays, the common approach is to use level 1 and 2 caches uniquely for a core while sharing the last level cache L3 among all cores. This combination is not the standard, there are many variations in the memory hierarchy of modern processors, regarding cache sizes, replacement policies and associativity. The following graph shows the hit rates for different SPEC92 benchmarks for various replacements policies:

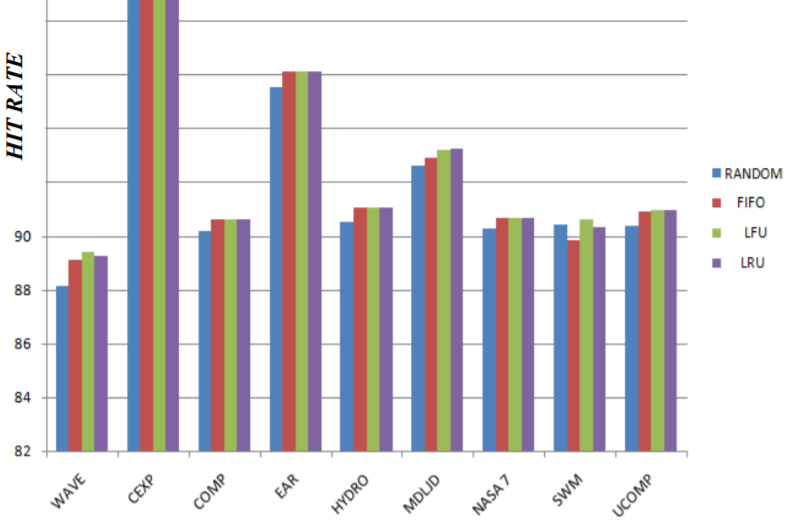


Figure 2. 3 Cache Hit Rates (Kumar & Singh 2016:4). Hit rates for various SPEC92 benchmarks given the same block size, coherency protocol and mapping.

It’s worth noting that even improvements of 1% can play a major role in memory performance, due to the high volume of memory accesses required in programs. While LRU replacement is the most efficient, it is also the most complex to implement especially for large caches. Trade-offs, as usual, are a part of computer design and they must be assessed on a case by case situation. A fundamental issue in the memory sector has been scaling, larger capacities are demanded every year while the overall size of the devices shrinks. The current situation can be accurately summarized by this statement:

Scaling of DRAM and NAND continues as innovations push out the perceived scaling wall, but fundamental economic-driven limitations are being approached. New kinds of memory -- emerging memories -- are being explored to supplement DRAM and NAND. These emerging memories begin to close the gaps in the memory hierarchy, but they do not challenge NAND cost or DRAM performance.

S. DeBoer (2018:4).

For the above-mentioned reasons and overall complexity to performance ratio, the processor will implement single-level caches with 2-way set associativity and LRU replacement policy. The caches are going to exploit the snoopy-based MESI protocol to maintain coherency and consistency.

## **2.3 ISA**

The motivations behind adopting the MIPS ISA are many and worth mentioning. This RISC type architecture has one of the most clear and comprehensible ISAs, and as shown in the latest editions it is also capable of handling more complex tasks revolving around multithreading and SIMD processing (MIPS 2010). The vast online material, covering every aspect of the MIPS architecture with in-depth analysis and practical examples, makes it a viable option for a self-directed project. The encoding is one of the key aspects of MIPS machines, its simplistic yet functional style is capable of supporting various types of processing, from embedded to parallel. In this design, the data contained in every instruction is easily retrieved thanks to the fixed positions provided by the encoding format. Having a rigid structure for the opcode and other fields significantly aids the decoding process. However, the instructions can synthesize only simple operations and thus the overall assembly language code is much more verbose compared to other architectures. This downside can be overlooked by taking into account the fact that with fixed-size instructions, pipelining can be done repetitively and safely. The subset of instructions that belongs to this ISA is extensive, especially considering the pseudo-instructions that contain multiple operations within them. In the design, the main priority were synthesisable instructions and thus only the fundamental instructions that map directly to machine code. The group of instructions specified in the project are included in the Appendix (A2). To indicate the end of the program, a special instruction named “*halt***”** has been developed; when *halt* is fetched, the processor completes its operations and returns to idle mode.

# Chapter 3 – Pipelined Processor

In light of the advantages given by pipelining, the processor has been divided in stages. Each of the five stages lasts a single clock cycle and connects through a series of registers to the next and prior stages. The following subsections describe the key functionalities that every stage covers, focusing primarily on operation rather than coding conventions. Unless otherwise mentioned, all the circuits within the stages are purely combinatorial, this will ensure that the one clock cycle timing is maintained.

## **3.1 Fetch**

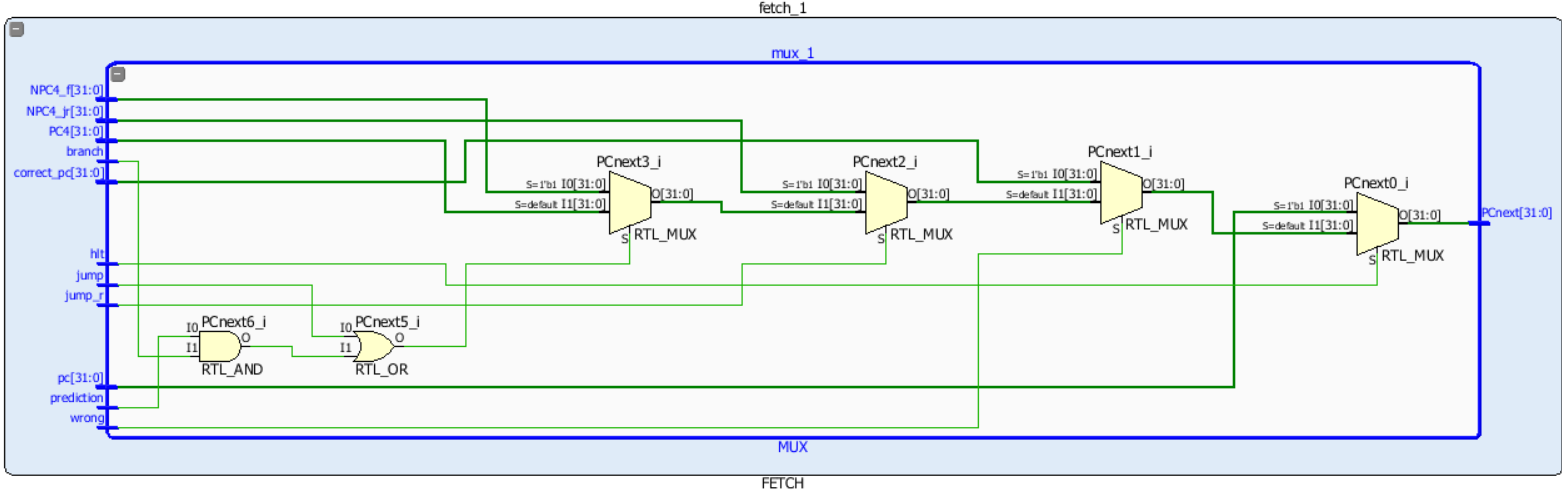
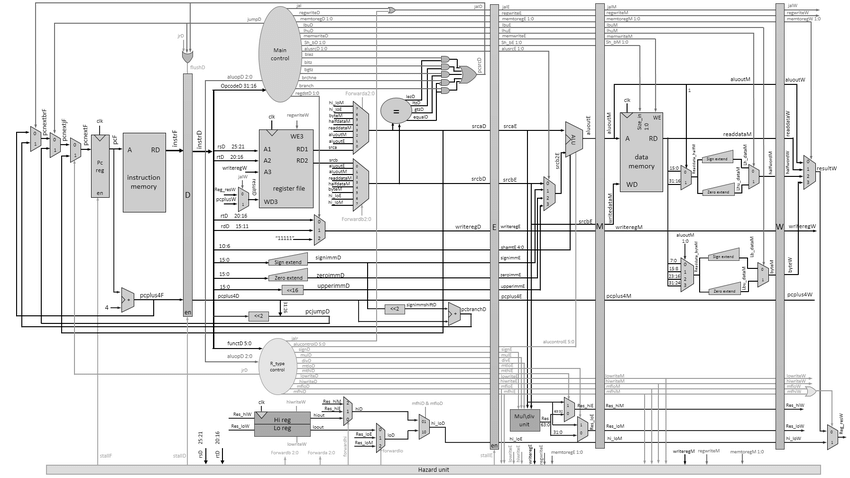
Fetching is the initial action for every pipeline, in this stage the instructions are retrieved from the program memory, based on the program counter that is updated every clock cycle. The program order can be altered at any time due to special instructions known as branches and jumps. In this case the fetch stage has to be able to substitute the instruction that was about to go in the pipeline with the correct one. To do so, a series of multiplexers is introduced ahead of the fetching mechanism, to specify the correct PC:

Figure 3. 1 PC selection. The chain of muxes ensures that only the correct PC is extracted.

The numerous *if-then-else*statements written in high-level languages to generate*for* loops, case statements etc. rely heavily on the execution of branches and jumps. Therefore, having reliable logic to implement them is critical. The fetch stage is tightly bounded to the prediction unit which will be covered at the end of the chapter; the information shared between the two units is fundamental for quick and rapid program execution. Once the instruction is retrieved, the decode stage can begin to deconstruct the operation.

## **3.2 Decode**

Decoding consists of breaking apart the instruction in order to obtain all the information encapsulated within it. Instructions, as mentioned above, are 32-bit wide and follow a specific encoding format that is the MIPS32 III standard. The instruction is here dived in its building blocks, and based on the information obtained, multiple control signals are sent to the other units. These signals contain valuable information in order to correctly resolve the instruction. As the pool of instructions and their complexity grows, so does the number of channels required to transmit the control signals. In multiple lane processors the control logic becomes even larger in size, since it will govern in real-time the issue logic responsible for routing the instructions. As we can see in the general diagram of a five-stage MIPS processor, the control signals that originate in the decode stage travel alongside the instruction down the pipeline.



CONTROL LINES

Figure 3. 2 General five-stage MIPS processor (Mahmood & Omran 2014 : 39). As highlighted in the red box, the registers across stages also intercept the control signals, which are then synchronised with the instruction as it travels down the pipeline.

This aspect proves to be extremely valuable in maintaining coherency between instruction and control logic especially for deeper pipelines. Besides the control logic, the decode stage contains the valuable register file, in which all the general-purpose registers are kept. This practically corresponds to a small memory unit with 32 elements, each being 32-bit wide. Given that many instructions require to read two operands simultaneously, the register file is dual ported. It receives two addresses and outputs the respective values all within one clock cycle. Being a register-register architecture, all the results of calculations must first be placed within registers so that they can be re-used later on during program execution. As explained in the next sections, when an instruction reaches the write back stage the results are ready to be committed to the register file. In this moment, the write operation begins, and the original destination address of the instruction is used to index between the 32 locations. Due to the fact that both the read and write operations must occur within a limited time frame, and that they may both require the same address, there is a strong need to order the sequence of accesses to the register file. With no priority, the data could be read without first being updated. To avoid this scenario and to meet the overall timing requirements, the register file has been inferred in the following manner:

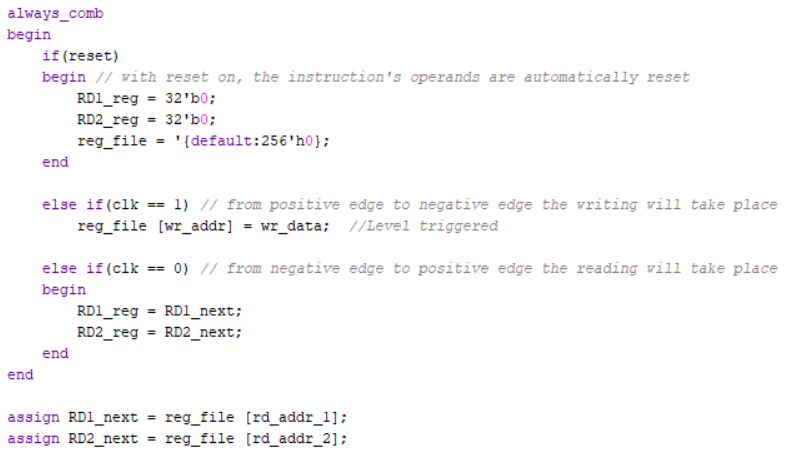


Figure 3. 3 Register file code. It is important to discern between edge and level triggering. The operations and hardware requirements associated for the two options are in fact much different.

By using both levels of the clock, in a purely combinatorial fashion, the correct access order is maintained while also resolving all operations within the clock cycle period. It is important to begin write operations during the positive phase so to prioritise them before any read task. The challenge with this approach is the high-level of uncertainty. When it comes to timing output changes, level triggering is much less precise compared to edge triggering. However, the design avoids any latch logic, which would see the output values directly connected to the inputs, without any sequential elements in between. Without latches and through strong sequential paths, the naturally chaotic level triggering can be sustained safely.

## **3.3 Execute**

The execution of all the major arithmetic operations occurs in this stage. The ALU covers additions, subtractions and more broadly any operation other than divisions and multiplications. These two calculations are reserved for the MULT/DIV unit, that is also included in the stage. The type of calculation is encoded by the control signals; from sign to immediate every aspect is specified to assist the ALU and MULT/DIV unit in their tasks. All the results from these calculations are associated with a specific general-purpose register, but the values are made instantly available thanks to the forward unit. Regardless of the size of the computation, the latency remains always one clock cycle. The instructions cannot refer to more complex calculus that is outside of the forementioned operations. Nevertheless, many calculations can be reduced to these operations, for instance branch instructions, which consist of comparisons, can be implemented through simple subtractions. The same example is valid for the SLT instruction, which is also translated to a subtraction followed by a special interpretation.

When creating these units, it is advised to guide the synthesis tool by simply detailing the desired functionality; hard-coding all the aspects of the operations rarely produces higher quality results compared to the synthesiser’s alternative. Thus, the SystemVerilog code expresses the functionalities of each opcode without dealing with any particular algorithm or scheme, like the Brent–Kung adder or the fast division scheme.

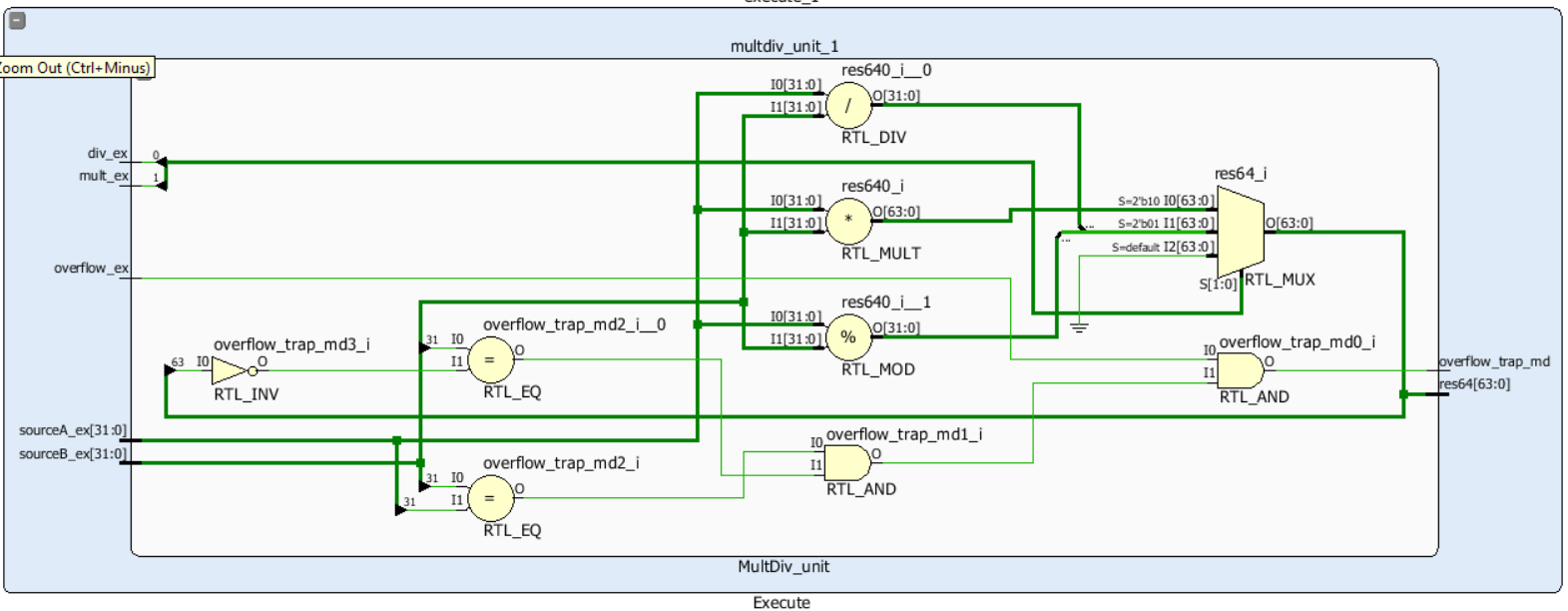
****

Figure 3. 4 MULT/DIV logic. As it can be seen from the schematics, general operation blocks have been extracted in the RTL design to complete the arithmetic tasks required by the processor.

For a group of special instructions, mostly dealing with memory and jumps, the execution stage is not needed and could be skipped. However, bypassing is feasible only with the aid of issue logic which would indicate the correct execution lane and route. Even for a few lanes, this added logic would require an extra stage, and thus eclipse any benefit obtained from the bypass. Therefore, the execution stage of the processor is respected even for these instructions.

## **3.4 Memory Access**

In the memory access stage, the instructions which require to work jointly with the memory system are finally resolved. The previous stage computes the memory address, and in case of a store, it also provides the data to be written. In the pipelined design, as per MIPS standard, the memory is byte-accessible, which means that a single byte within a block of memory can be accessed. This contributes to the efficient usage of the memory space, as it allows to pack the full width of the word. Instructions like Load Byte (LB) and Store Byte (SB) operate on this premise. Simultaneously, the processor is also capable of fetching an entire word if necessary. To dynamically make the decision between word, halfword and byte, within the static hardware environment, two scenarios are represented:

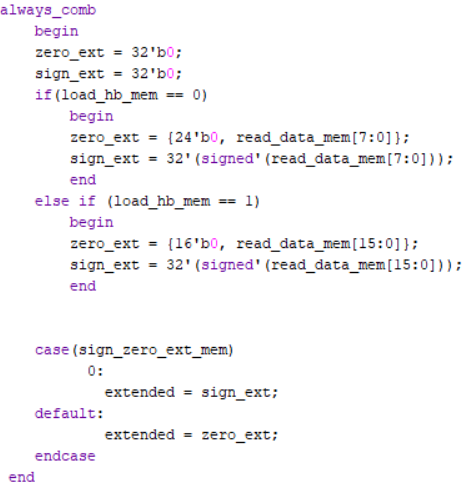
****

Figure 3. 5 Load scenario. The code snippet above is utilized to discern between loading a single byte or halfword from memory. The *read\_data\_mem* signal contains the word fetched from the address.

The *load\_hb\_mem* signal, created in the decode stage, is used as the selector for a mux which chooses between the byte and halfword options. The full 32-bit version of the data is automatically sent ahead so there is no need to add an extra bit to indicate this option. The trimmed data is both sign and data extended, with a final case statement selecting the particular version. In order to have easier readability of the data, the machine is set to be little endian, which means that the highest value of the data is stored in the highest memory location. This also facilitates the transfer of data from and to main memory.

The other scenario, which encapsulates all store operations is presented below. When storing data into the memory or caching system, the location is indicated by the offset, which consists of the lowest 4 bits of the memory address.

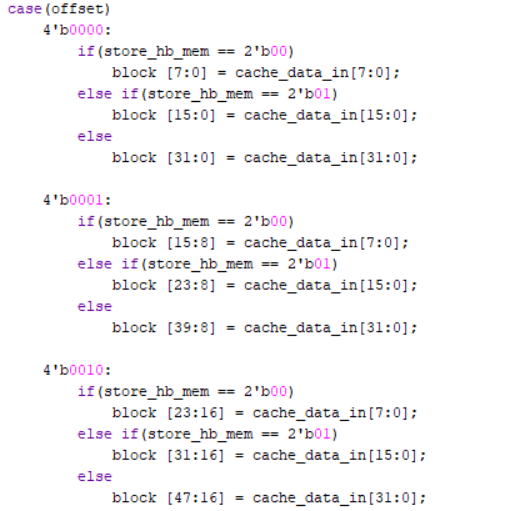
****

Figure 3. 6 Store scenario. The case statement is repeated for all the possible values of the offset in order to avoid inferring latches.

In this case, the *store\_hb\_mem* signal is used as the selector for the mux. Unlike the load control signal, it is formed by two bits since it also has to specify the 32-bit option. The case statement isolates the portion of the block/line to operate on, and the store signal indicates the specific byte within this segment. A key difference of this architecture with respect to the MIPS standard is the data structure alignment. Some architectures, for ease of design, have rigid word alignment policies, in which the 4 bytes constituting a word are fixed boundaries that cannot be moved. In these structures, the data is fetched or written only along these boundaries (Harris & Harris 2012:304). However, the memory system for this processor is much more flexible. As indicated above, the case statement allows to break the whole block of memory into multiple smaller bytes whose position is not dependent to the word itself. Thus creating a byte aligned memory system. This variability does not cause an overcomplication of the design; the data can now be simply fetched without having strict boundaries delimiting the start and end points. This approach, that allows for word-misaligned memory accesses, is feasible only for small designs which can tolerate the slower functionality caused by the extra logic.

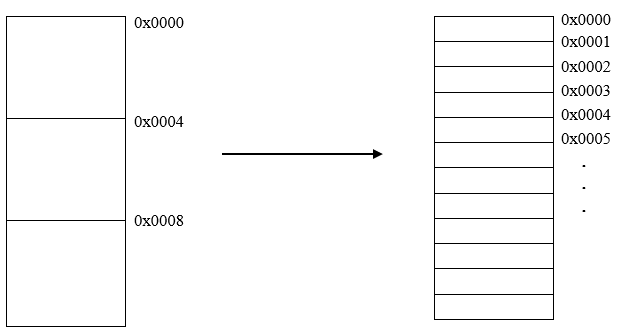


Table 3. 1 Memory alignment. The word alignment scheme (left) is substituted by the byte alignment scheme (right).

The goal of one cycle latency for the memory stage is often not sustainable. The memory units where the data is held can be scattered around the CPU and have different latencies. For close proximity caches the latency is inferior compared to main memory which could even be off-chip. Therefore, the time spent in this stage is dependent on the speed with which the address location can be accessed. It is often the case that this period of time exceeds the single clock cycle, in this scenario stalling the pipeline is required.

## **3.5 Write Back**

The sole objective of this stage is to update the register file with the correct information calculated within the pipeline. It is the last stage of the processor and decides what sort of information is ready to be committed to the register file. In many real-life processors, that implement out-of-order execution, this stage writes the results in the ROB buffer. This special structure, which works based on the Tomasulo algorithm for register renaming, allows the processor to fetch and complete instructions out of the normal order, while also committing or writing back in the correct sequence. For the simpler in-order design of the dual-core processor, the write back stage is directly connected to register file so the ROB is not needed (Joel Emer 2005).

## **3.6 Forwarding**

Forwarding consists in the quick transmission of data between stages with the purpose of avoiding lengthy stalls that would impact performance. The forwarding mechanism is rather straightforward, once it detects an anomaly or dependency, the system forwards the data to the desired stage as soon as it is available. In most cases this can resolve the vast majority of data hazards. However, there are occasions in which stalling cannot be avoided:

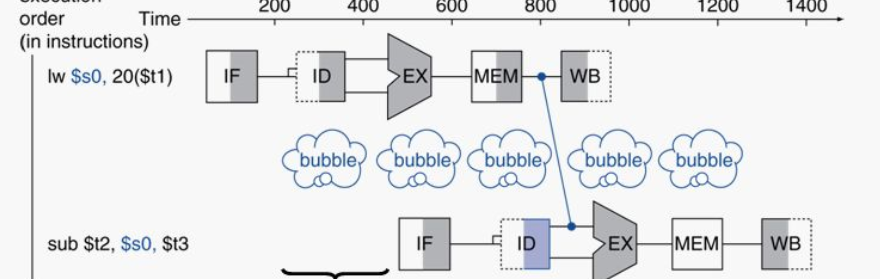


Figure 3. 7 Forwarding and dependencies (McQuain 2013). Forwarding backwards corresponds to sending information back in time, which is clearly impossible.

The sequence of instructions above poses a dependency which even the forwarding logic can’t resolve. Therefore, a bubble, i.e. a stall signal, is inserted in the pipeline. There are various solutions to possibly avoid stalling completely, but the requirements in terms of cost and design time overshadow the benefits, particularly for simpler designs. One easy method, capable of limiting the number of stalls, will be shown in chapter 5. Nevertheless, the most important aspect is to create fail safe mechanisms in which at least the detection of hazards within the pipeline can be guaranteed.

## **3.7 Predicting**

The prediction unit has two mains goals:

* To successfully guess the outcome of branch instructions.
* To track the actual outcome of branches in the pipeline.

When a branch instruction is met in the fetch stage, the PC of the following instruction has to be chosen between two alternatives: the normal PC, i.e. PC + 4, or the target PC encoded in the branch instruction. The former option is correct only when the branch is not taken, while the latter is valid only for taken branches. To determine the status of a branch, the instruction has to at least reach the execution stage, where the comparison between operands is resolved. Obviously, stalling every time a branch is encountered in order to choose the following instruction is not tolerable. The prediction unit is located within the fetch stage itself, and it formulates an educated guess regarding the outcome of any branch that is fetched. It also has the duty of tracking down any branch that is resolved in the execution stage. This aspect is extremely important because, for mispredictions, the pipeline has to disregard the wrongly fetched instructions in an operation known as “flushing”, while also restoring the initial correct PC. To predict with accuracy, the processor adopts the Branch History Table (BHT) scheme, in which previous records of a given branch are held in order to guess its future behaviour. The table stores only two bits per branch, which function as saturating counters. Through the use of a simple hashing function, based on the current PC, any branch is matched to a specific entry of the table.

Diagram

Description automatically generated

Figure 3. 8 BHT. The hashing function forms the address based on five bits of the PC.

The reason behind the choice of using a BHT, with just one level prediction, is due to the hardware requirements and warm-up period associated with two level prediction schemes. Although the accuracy is higher for such schemes, the time needed to train all the tables and patterns would be too damaging for the short test programs used. A BHT on the other hand, has fairly reasonable prediction accuracy (above 80%) and it is also quite fast to train (UFMS 2008:9). To further speed up the training of the table, 2-bit saturating counters with taken and not-taken shortcuts have been developed, they are based on the phenomenon of fast hysteresis.

Diagram

Description automatically generated

Figure 3. 9 Faster acting hysteresis. Unlike normal counters, there is no link between weak states.

Whenever a branch executes, the counters are updated and the overall table is trained. The mechanism is coded as an FSM, in which every counter follows the same case statement. This combination of immediate prediction and delayed correction allows the processor to resolve branches instantly, but also to adjust the pipeline in case of mispredictions. The misprediction penalty, due to the relatively small pipeline and time latency per stage, is limited to only 2 clock cycles.

## **3.8 Elaborated Design 1**

The elaborated design shown below, contains the RTL schematic of the five-stage pipeline. The prediction unit is embedded within the fetch stage and the forward unit is present outside the stages as a purely combinatorial circuit.

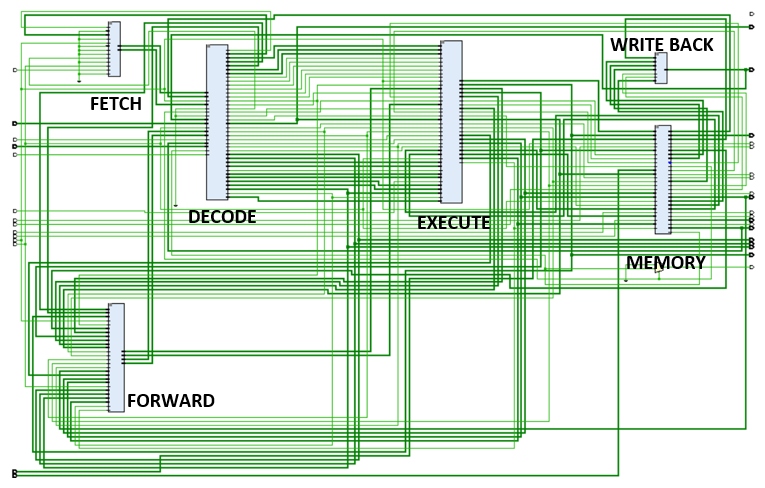
****

Figure 3. 10 Elaborated design 1. The size of the blocks depends on the number of I/O connections.

All the stages, including the forward unit, lie within the same level of the hierarchy, they receive the clock and reset signals from the higher module and pass it to the lower instances. The reset logic is purely asynchronous and takes precedence over the clock, it mostly causes an instantaneous clearing of the data present within the various stages, but in special scenarios, it sets particular values. For instance, in the case of the BHT, knowing that most branches are placed within loops, the reset value stores in every table, i.e. the weak taken state. This will potentially limit the initial number of mispredictions, since the first instance of any loop should be predicted correctly.

# Chapter 4 – Dual-Core Design

## **4.1 Duplication Process**

The standalone pipeline represents a huge improvement from the single cycle processors developed in the early years of computing. The uniprocessors, in fact, improved performance drastically simply by diving the instruction within the stages, which consequently allowed the usage of faster clocks bounded only to the slowest stage of the pipeline. To form the dual-core system, all the execution units, stages and connections of the pipelined processor have been duplicated, creating the exact same structure. To sustain the same circuitry and nomenclature for both pipelines, a transitionary step was needed. As the shared lower instances output their signals, the top modules must change the signals’ names to uniquely identify their origin. For instance, the execution unit is identical for both cores, and it is instantiated in two separate locations. But as the signals escape the lower module, their names are differentiated with the “\_1” or “\_2” suffix, which will distinguish the core origin up in the hierarchy. It is possible to easily duplicate the pipelined processor many more times. The issues that prevent this from happening are not related to the scaling of the hardware/resources, but rather, they are mostly tied to the program nature and memory system.

## **4.2 Data/Instruction Cache**

Caching is a fundamental aspect of modern computers. Regardless of the size of the CPU, caches can produce significant improvements in terms of memory access time. In this design, data caches have been introduced as an intermediary juncture between the pipeline and main memory. The caches are two-way set associative with an overall size of 64 bytes each. Although the size of the caches is rather minimal, they are sufficient for holding the blocks of data accessed regularly during program execution, without causing conflict or capacity misses. The associativity in fact, allows for smart usage of the space within each cache. Furthermore, the ways are replaced according to the LRU mechanism. This scheme is highly beneficial and easy to implement for two-way set associative caches. Two bits are added to the end of the tag indicating the status of the corresponding block of memory: binary 1, showing that the data has been recently accessed, and binary 0 the opposite. In such a manner, the data that has been accessed for last is guaranteed to be expelled whenever new data has to be introduced in the set (UIUC 2011).

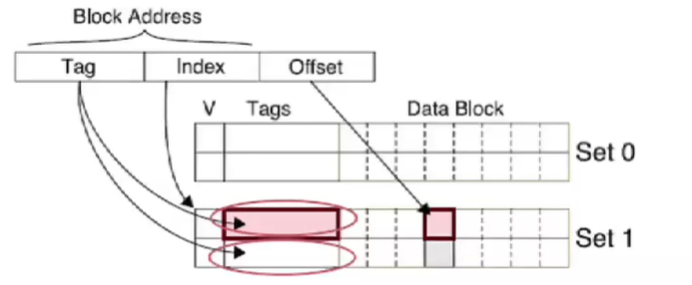


Figure 4. 1 Two-way set associative cache. There are two ways per set, each with an independent tag.

The tag uniquely identifies the data block and it is formed by the high order bits of the memory address, while the extra bits, before the tag, represent the MESI and LRU status. Once the index has specified the set, two tag checks will be conducted in order to determine if there is a hit or miss in the cache. Finally, the offset will isolate the given memory segment within the wider data block.

The first draft of the processor contained both data and instruction caches. However, during the development of the test programs, the lack of necessity for instruction caches became evident. These caches are utilised to store part of the program for very demanding jobs, which would otherwise require to constantly reach back to main memory. For the relatively small test programs used, instruction caches are not needed, and the information can be fully encapsulated in a ROM structure. This type of non-volatile memory is initialised before booting, cannot be written and is quickly available during program execution (Sloss et al. 2004:10). To infer the two ROMs in Vivado, the *ROM\_STYLE* attribute has been used alongside the memory files listing all the instructions of the given test program in a sequential order. (Xilinx 2020:175).

## **4.3 Hazard Unit**

The hazard detection is conducted in parallel to the program execution; as the data and instructions are fetched, the unit checks if any dependency is present with respect to the preceding operations. If necessary, the unit triggers the signals *stall\_1*and *stall\_2*, which are connected to the individual cores. The checking has to be done in a combinational fashion to ensure that stalling happens in a timely manner. The stall signal reaches all the stages within the core, in an asynchronous mode comparable to the reset signal, and it stops the transitions between them. The sudden stop is forced on all stages except on the one that originated the issue and the subsequent ones. Stalling is undoubtedly a necessary mechanism for all pipelines, but it is seen as a last resort option, and avoided if possible. The unit is instantiated as a high-level module, this allows it to control both cores at the same time and reduces the glue logic required to connect instances. Thus, stalling one core is independent from stopping the other.

## **4.4 Main Memory**

Main memory, in processors operating with large programs, is often bounded to IP cores having pre-defined structures. The RAM module could be inferred using elementary coding statements, however, as the number of cores and accesses increases, avoiding synchronization issues becomes more challenging. Therefore, it’s better to adopt pre-developed IP cores capable of guaranteeing performance and that greatly boost design time (Rajesvari et al. 2013). With the Block Memory Generator, it is also possible to defy vast memory modules in Vivado with many more features, including power and speed optimizations. For this system, through only the GUI, it was possible to create a small sized 0.5kB BRAM, organised in rows of 4 words, for a total of 32 rows.

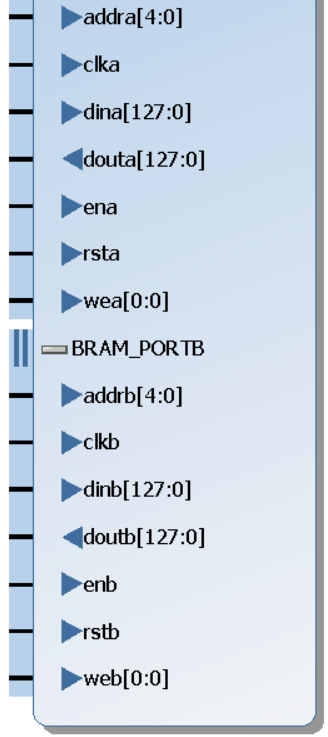
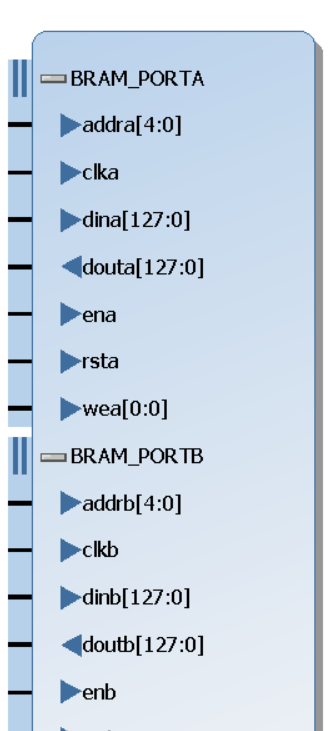


Figure 4. 2 BRAM IP core. The BRAM has been selected inside the Vivado IP core catalogue. The Write Enable signals are not used, thus the 0 to 0 bit range.

The practice of using IPs for complex and small designs is common in modern SoCs processors, since it shortens the overall production time. Like for the majority of IPs, the internal source code content of the BRAM is mostly encrypted and not modifiable at all. What can be edited is the instantiation template which connects the processor to the BRAM module. This file allows to incorporate the memory unit on the same hierarchical level of the cores and hazard detection unit.

## **4.5 Cache Coherency**

To maintain the data within the caches coherent, the MESI protocol is used in both cores. The four states shown below are developed using an FSM, which is part of the cache controller.

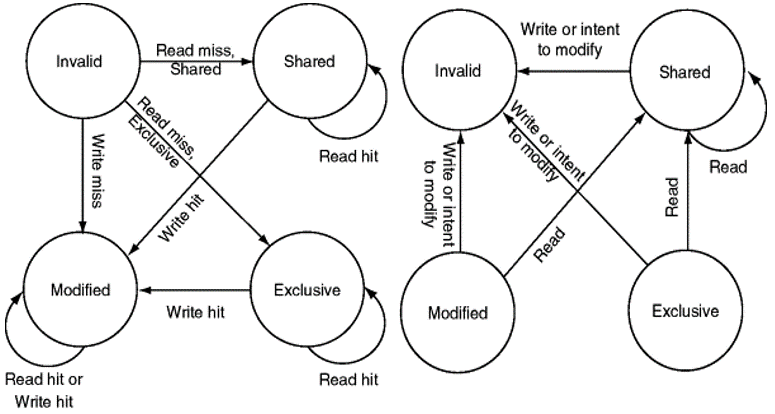
****

Figure 4. 3 MESI protocol. The photo includes both periods of operation.

The MESI bits are attached to the front of the tag and are initialized in the invalid state. The snooping mechanism, although power inefficient, is rather straight forward and cost-effective for fewer cores. The Write and Read Intent signals are broadcasted in the common bus that connects the cores. These signals are needed to inform the other cache that the data they possess might have to change state. Within the cache controller itself, a bigger FSM is used to govern all memory accesses:

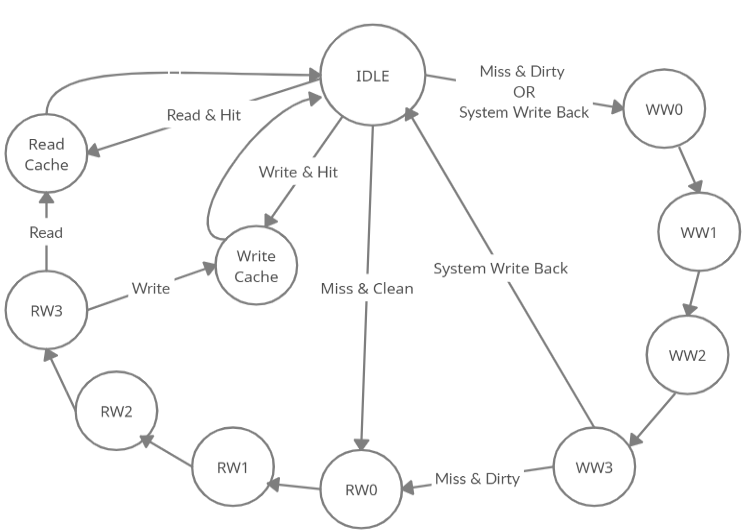
****

Figure 4. 4 FSM for the cache controller.

The FSM is modelled following the diagram above. The hit/miss, read/write and the dirty/clean signals all originate from the cache controller unit. The System Write Back, on the other hand, is an external signal triggered by the other cache; It is crucial for the operation of the dual-core processor, as it forces caches to update the main memory when modified data has to be shared. Also, whenever dirty data is evicted, the cache is required to write it back. This is in line with the Write-Back policy, that highly reduces the overall accesses to memory compared to the Write-Through scheme. As shown in the state diagram, during the writing and reading of the main memory, many cycles are spent. For a simple cache miss involving a dirty block, the total latency before the operation is completed is 10 cycles. There are multiple ways to reduce this time frame; shortcuts between idle and other states are effective ways to do so and are utilised in practice by the FSMs.

## **4.6 Elaborated Design 2**

The elaborated design shown below, contains the entire RTL schematic for the dual-core system.

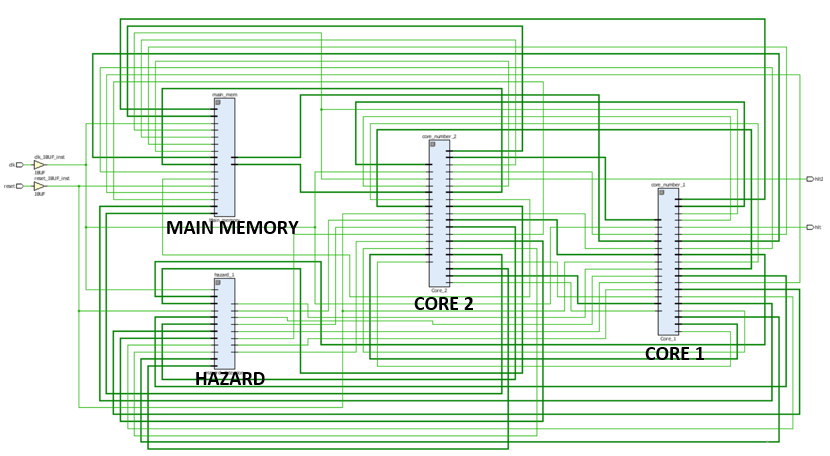
****

Figure 4. 5 Elaborated Design 2.

As mentioned previously, the hazard detection unit and main memory reside on the same hierarchical level of the cores. The entire dual-core design consists of the modules included in the figure. If implemented on the FPGA, the system would receive the clock and reset inputs directly from the board network, and it could output the calculated results to other units within the FPGA environment; thus creating a full functioning computer.

# Chapter 5 – Practical Attainments

Starting from the simplest program, all the results comparisons between single core and dual-core processor are included in this chapter. The simulations indicate execution time and values stored within the cache/memory storage. To facilitate the comparison, the test programs have not been altered even when moving from the single core to dual-core system. The signals included in the screenshots have been selected to represent the progression of the program without cluttering excessively the screen. Repeating signals have been diversified using the suffix “\_2” to distinguish the core origin.

## **5.1 Fibonacci Sequence**

This test program consists of calculating the first 40 numbers of the Fibonacci Sequence. The assembly program is developed using a recursive loop that iterates the operations present in the calculus. To do so, the program relies heavily on branches and it rarely accesses main memory, since the values are stored in the register file. The fact that the results are constantly needed and are the base for further calculations, makes sharing the workload between cores rather complex. If every result is the starting point for the next calculation, the program can’t be easily divided across cores, unless high-level threading is in place. The data traffic across caches even in this scenario would be extremely high. The total number of operations to calculate the final value is 40: 38 additions and 2 value initializations. This translates to 16 instructions in assembly, with the core loop of 9 instructions included below. Although the parallel solutions are of little help, simple code rescheduling proved to be highly beneficial. By inserting add instruction A between the loads and add instruction B, a taxing stall cycle arising from a data dependency is entirely eliminated from the loop.

|  |  |  |
| --- | --- | --- |
| **Assembly**  Loop  A  B | **Description** | **Machine** |
| addi $t3, $t0, -8 | Put t0 – 8 in t3 | 210BFFF8 |
| addi $t4, $t0, -4 | Put t0 – 8 in t4 | 210CFFFC |
| lw $t1, 0($t3) | Load from memory t3, and put into t1 | 8D690000 |
| lw $t2, 0($t4) | Load from memory t4, and put into t2 | 8D8A0000 |
| addi $t0, $t0, 4 | Increase t0 by 4 | 21080004 |
| add $t5, $t1, $t2 | Put t1 + t2 into t5 | 012A6820 |
| sw $t5, -4($t0) | Store t5 in memory t0 - 4 | AD0DFFFC |
| slt $t8, $t9, $t0 | If (t9 < t0) set t8 to 1 else 0 | 0328C02A |
| beq $zero, $t8, loop | If t8 is zero branch to loop | 1018FFF7 |
| hlt | End of program | F0000000 |

Table 5. 1 Loop for the Fibonacci sequence. Add instruction A is inserted there in order to distance in time the load and add instruction B.

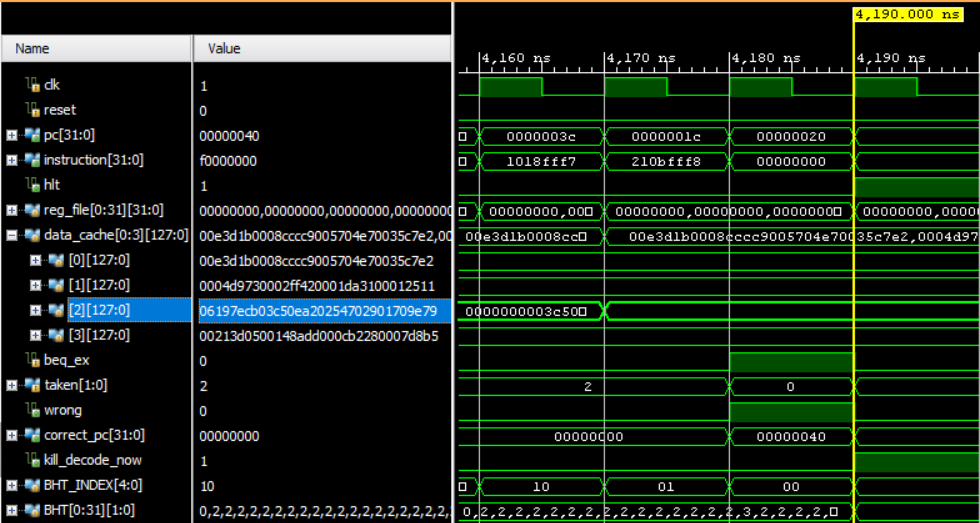


Figure 5. 1 Fibonacci result. The final value of is the expected result of the calculations.

The result is placed within the cache after 4170 ns, two cycles before the halt signal is triggered. The timing is a consequence of the clock speed and number of instructions. To calculate the total CPI for the program, it’s necessary to first break apart the loop, in a process known as “loop unrolling”. Once this is done, the instructions outside the loop can be added to get the final total number, with which the CPI is obtained:

As expected, the CPI didn’t reach a value smaller than 1. While attempting to break the program among the two cores, the non-parallel characteristics of the program became evident. The overhead arising from the cache fetching and loading mechanism, actually caused the opposite effect, making the overall execution time slightly longer. This indicates that for serial programs, that have limited or no parallelism at all, dividing the operations in threads is laborious and often counterproductive.

## **5.2 Additions and Multiplications**

In this other test program, parallel calculations are conducted in order to exploit the hardware resources of the processor and have a comparison with the single core design. The test is formed by a single factorial and one addition sequence. The results of both operations are then added in one memory location showing the final result. The sequence consists of the sum of all the numbers between and . In order to maximise throughput, the starting factorial number and addition sequence need to be matched exactly. In this case the factorial of 7 and the additions both occupy the individual cores in an even manner.

d

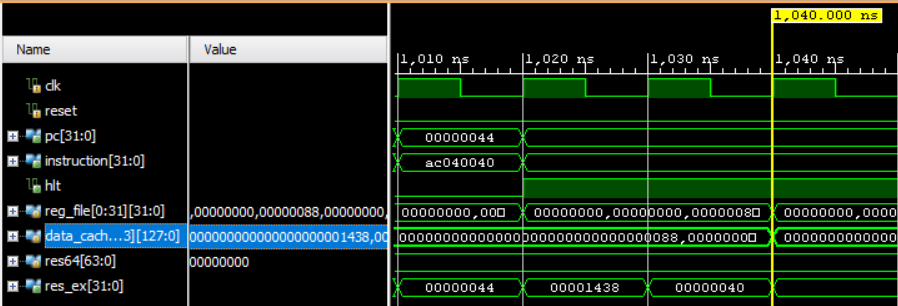


Figure 5. 2 Additions and Multiplications result using a single core.

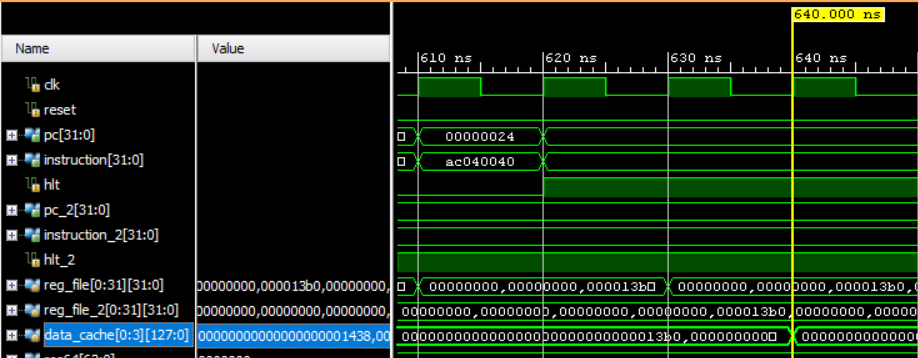


Figure 5. 3 Additions and Multiplications result using both cores.

The same result of is reached in both processors, the clear difference is the timing. The dual-core system is 500 ns faster compared to the single core version, which finishes in almost twice the time. What’s worth noticing is that the CPI of the dual-core design is below 1, like in any other superscalar processor.

The fact that both calculations were independent from each other, made the memory interactions straightforward.This simple format of work-sharing is rarely found in real-life processing, where programs usually have semi-dependent threads of work that communicate constantly with each other. This program, while it does occupy both cores simultaneously, it does not test the cache coherency and data sharing mechanism to the full extent.

## **5.3 Matrix Multiplication**

In this program the parallel capabilities of the processor have been tested to a greater extent, both cores in fact, have to resolve more than 60 lines of assembly each. The program itself consists of matrix multiplications of two distinct sets of matrices. The results of both multiplications are then added in one memory location which is showcased at the end of the program execution. The original matrices are the following:

Diagram

Description automatically generated

Figure 5. 4 Matrix schematic. Only the values of the final matrix are in hex format.

The goal was to complete the entire program with a CPI significantly lower than 1, this would represent a great improvement compared to the CPI of single core processors.

f

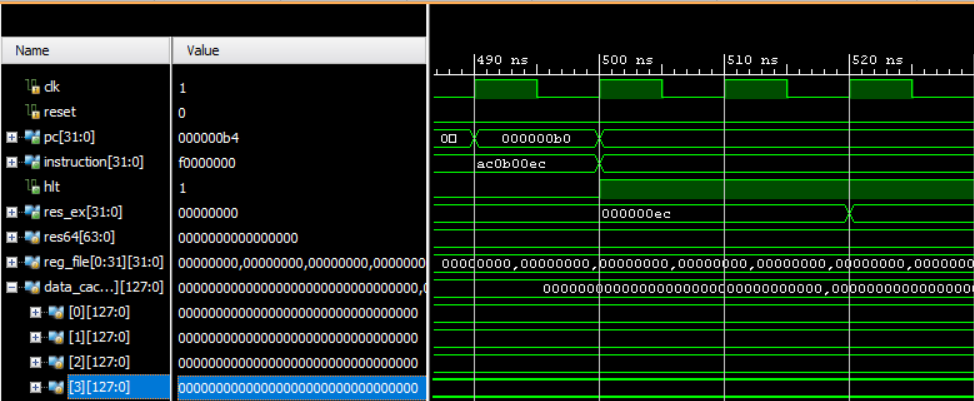
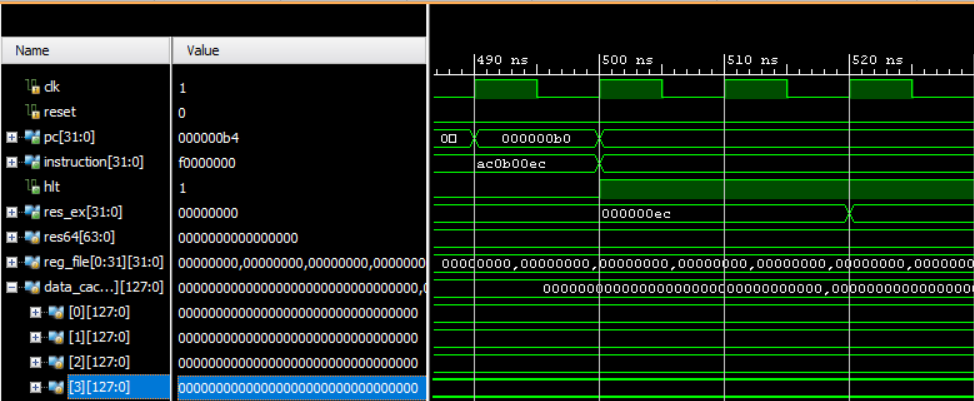
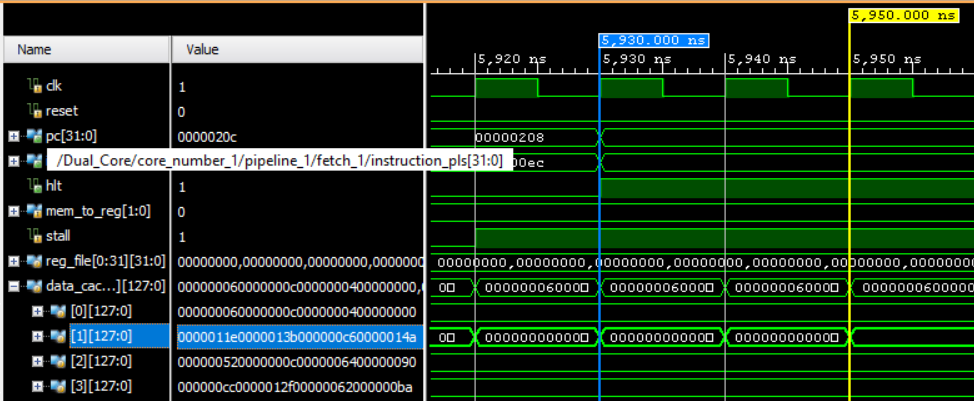


Figure 5. 5 Matrix multiplications results using a single core.

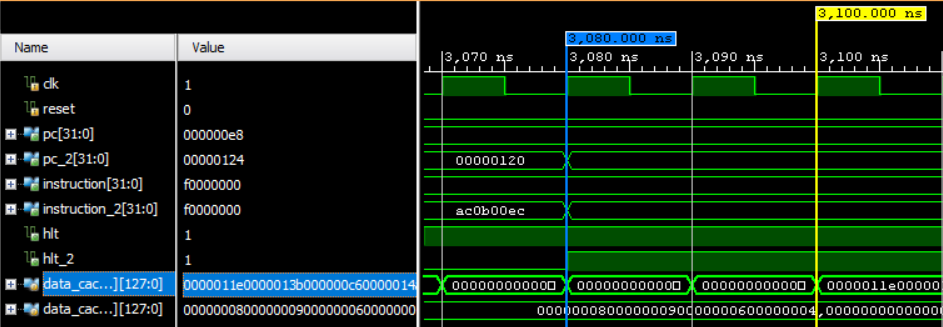


Figure 5. 6 Matrix multiplications results using both cores.

Although for both processors the end of the program is represented by the halt signal, the actual data cache is updated two cycles later, therefore this latency is used to obtain the CPI:

The overall program completes in 3100 ns, which indicates an improvement of 2850 ns from the single core design. This is due to the parallel resources of both the hardware and software, which together allow for a faster program execution. The ultimate goal remains a full symbiosis between parallel program and hardware capabilities, with CPIs tending towards zero. Throughout the program execution the cores retrieve similar types of instructions. This is expectable since both operations revolve around matrix multiplications. However, maintaining a constant utilisation of both cores at all times is rather challenging. In the last screenshot it can be seen how, as the last core terminates its execution, the previous one had already halted long before. This is indeed one of the obstacles limiting perfect parallelisation in multi-core processors.

## **5.4 Critical Appraisal**

Experimenting with a limited number of cores and relatively simple programs, highlights both the benefits and challenges of multi-core computing. The increase in throughput is undoubtedly present, regardless of the software/hardware interface. Faster execution times are expected, but their extent is clearly dependent on more factors than core number itself. As caching solutions become more elaborated, the possibility to exploit parallel hardware more efficiently increases highly. Nevertheless, for purely serial programs the only performance boosts are given by known uniprocessor techniques, like ILP and code rescheduling. The pursuit of parallelisation, in these cases, can require extreme efforts just for minimal gains. Even in the opposite scenario, the overall benefits do not depend only on the number of cores used, since the overhead and memory units often play a role in the performance outcome. The advantages, while limited, are prominent enough to have started a push towards the generation of ad-hoc multi-threaded applications, that can take maximum advantage of the hardware capabilities. The programmers, no longer relying on faster clocks or transistors count, are forced to embrace the only new method to speed execution times (Bellairs 2019).

# Chapter 6 – Conclusion

The design of a dual-core MIPS processor has been proposed, showing the intermediary steps and outcomes. The key results include:

* Development of a five-stage MIPS processor containing all the relative elements.
* Complete core development including caches and hazard detection unit.
* Dual-core design based on the duplication process.
* Simulation results highlighting advantages and limitations of modern parallelism.

The utilization of a memory IP core within the dual-core design is another fundamental aspect of the project. Re-using IP cores is an extremely common practice that is behind the fast development of many new SoCs. All the remaining sections were coded from scratch in SystemVerilog, which is a tolerable approach only for small systems. The caching mechanism, with his associativity, replacement policy and coherency protocol, was the most laborious section of the processor.

# Chapter 7 – Future Work

As mentioned throughout the project, the processor can be strongly improved in order to be more versatile and adaptable. The first clear follow-up would be the synthesis of the code and further implementation in the Nexys 4 DDR board environment. This task is straightforward but requires a significant amount of time, especially considering that not every section of the code is synthesisable at the moment. Once the processor is ready to be implemented, the bare-metal system can be adorned with a simple OS or even just I/O interface programs. To do so, the SDK tools included within Vivado can be used. A practical solution would be the connection of the processor to the LEDs present within the board; based on the lights sequence the status of the pipelines or the final results could be shared. For larger results, not encodable using the LEDs, a VGA cable could be used to transfer the cache or memory blocks containing the results. The VGA port is already present within the target board, and the I2C protocol used to direct the flow of data can be implemented through an IP core. The quality of 640 pixels by 480 pixels is sufficient to render the data clearly in any monitor supporting VGA. After these adjustments have been made, many other modules can be instantiated alongside the dual-core processor, effectively forming an SoC. Extra IP cores revolving around I/O interface, like UART and ADSR can provide further sound/connectivity functionalities. The structure that connects all the entities can also be found in already established IP cores based on the AXI bus interconnection protocols, or it can be modelled based on the MMIO access methodology. Either options would highly benefit on-chip communications and transform the system from a simple CPU to a more well-rounded system.

# Chapter 8 – Student Reflections

Completing this project has benefitted me in a variety of ways, both from a technical and personal standpoint. Starting with the technical side, I have greatly expanded my previous knowledge of computer architecture, digital design, and prototyping. I now have a much better understanding of the challenges involved in electronics design and what to expect when embarking on a new project. The vast subjects covered throughout the semesters have also launched my interest towards new fields, like ASIC and I/O devices. I plan and hope to explore these areas in the upcoming months and to incorporate them in the project itself. From a personal perspective, conducting the project in the midst of all the other assignments, duties and the current environment, was rather challenging. The difficulties associated with not having instantaneous solutions to my problems or fellow collogues to compare myself with, have forced me the grow and to become much more self-sufficient. From scheduling to accountability, I have improved on many fronts, and now I am more independent with my work. Looking back, reviewing all the phases of the project, there are undoubtedly decisions I would change. These mistakes, born out of inexperience, consumed a lot of time and ultimately prevented me from completing all the initial objectives, as shown in the Gannt chart and logbooks (Appendix A1-A3). In fact, while the simulation results are sufficient given the current situation, my goal still remains to complete the implementation on the board itself, with the aid of another student or during the summer. The achievements are nevertheless important, and considering the effort required to overcome the challenges, I am satisfied of the final results. Also, by working independently, I now have a much better appreciation of the advantages that are intrinsic to teamwork. For both major and minor projects, having members to confront yourself with, is much more important than what I previously thought.

# List of References

Cerebras (2019) *Wafer Scale Engine: Why we need big chips for Deep Learning*  [online] available from <https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/> [25/04/2021]

David A. Patterson, John L. Hennessy (2013) *Computer Organization and Design: The Hardware/Software Interface 5th edition* USA: Morgan Kaufmann

[P. Kocanda](https://ieeexplore.ieee.org/author/37085448161), [A. Kos](https://ieeexplore.ieee.org/author/37432091700) (2015) *Static and dynamic energy losses vs. temperature in different CMOS technologies* [online] available from <https://ieeexplore.ieee.org/document/7208560> [25/04/2021]

ITRS - International Technology Roadmap for Semiconductors 2.0 (2015) *System Integration* [online] available from <<https://www.semiconductors.org/wp-content/uploads/2018/06/1_2015-ITRS-20_System-Integration.pdf>> [25/04/2021]

Charles E. Leiserson (2020) *We’re not prepared for the end of Moore’s Law* [interview by D. Rotman] in MIT technology review (2020) [online] available from <https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/> [25/04/2021]

Paul E. Ceruzzi (2003). [*A History of Modern Computing*](https://archive.org/details/historyofmodernc00ceru_0) *2nd edition.* USA MIT Press. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [978-0-262-53203-7](https://en.wikipedia.org/wiki/Special:BookSources/978-0-262-53203-7)

Rana E. Ahmed. (2006) *Energy-Aware Cache Coherence Protocol for Chip-Multiprocessors* [online] available from <<https://ieeexplore.ieee.org/document/4054836>> [25/04/2021]

S. Kumar, P. K. Singh (2016) *An overview of modern cache memory and performance analysis of replacement policies* [online] available from <https://ieeexplore.ieee.org/document/7569243> [08/11/2020]

S. DeBoer. (2018) *Memory Technology: The Core to Enable Future Computing Systems* [online] available from <https://ieeexplore.ieee.org/document/8510707> [25/04/2021]

MIPS (2010) *MIP64 Architecture* [online] available from <https://www.mips.com/products/architectures/mips64/> [21/04/2021]

Hadeel S. Mahmood, S. Omran (2014) *VHDL Prototyping Of A 5-Stages Pipelined Risc Processor For Educational Purposes* [online] available from *<*<https://www.researchgate.net/publication/327968316_VHDL_PROTOTYPING_OF_A_5-STAGES_PIPELINED_RISC_PROCESSOR_FOR_EDUCATIONAL_PURPOSES>> [25/04/2021]

David M. Harris, Sarah L. Harris (2012) *Digital Design and Computer Architecture 2nd edition* USA: Morgan Kaufmann

Joel Emer (2005) *Advanced Superscalar Microprocessors* [online] available from <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-823-computer-system-architecture-fall-2005/lecture-notes/l14\_superscalar.pdf> [26/04/2021]

William D. McQuain (2013) *Computer Organization II* [online] available from <https://slidetodoc.com/hazards-pipeline-1-situations-that-prevent-starting-the/> [25/04/2021]

UFMS (2008) *Branch Prediction* [online] available from <<http://www.facom.ufms.br/~ricardo/Courses/AdvTopCompSys-2008/Lectures/Branch-Prediction-lec.pdf>> [25/04/2021]

UIUC (2011) *Set-associative caches* [online] available from <https://courses.engr.illinois.edu/cs232/sp2011/section/disc8sol.pdf> [25/04/2021]

Andrew N. Sloss et al. (2004) *ARM System Developer's Guide Designing and Optimizing System Software* USA: Morgan Kaufmann

Xilinx (2020) *Vivado Design Suite User Guide Synthesis* [online] available from <<https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf>> [27/04/2021]

R. Rajesvari et al. (2013) *IP core based architecture of telecommand System-on-Chip (SoC) for spacecraft applications* [online] available from <https://ieeexplore.ieee.org/document/6497979> [25/04/2021]

# R. Bellairs (2019) *How to Take Advantage of Multithreaded Programming and Parallel Programming in C/C++* [online] available from <<https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp#why>>

# Appendix

## **A.1 Project Management**

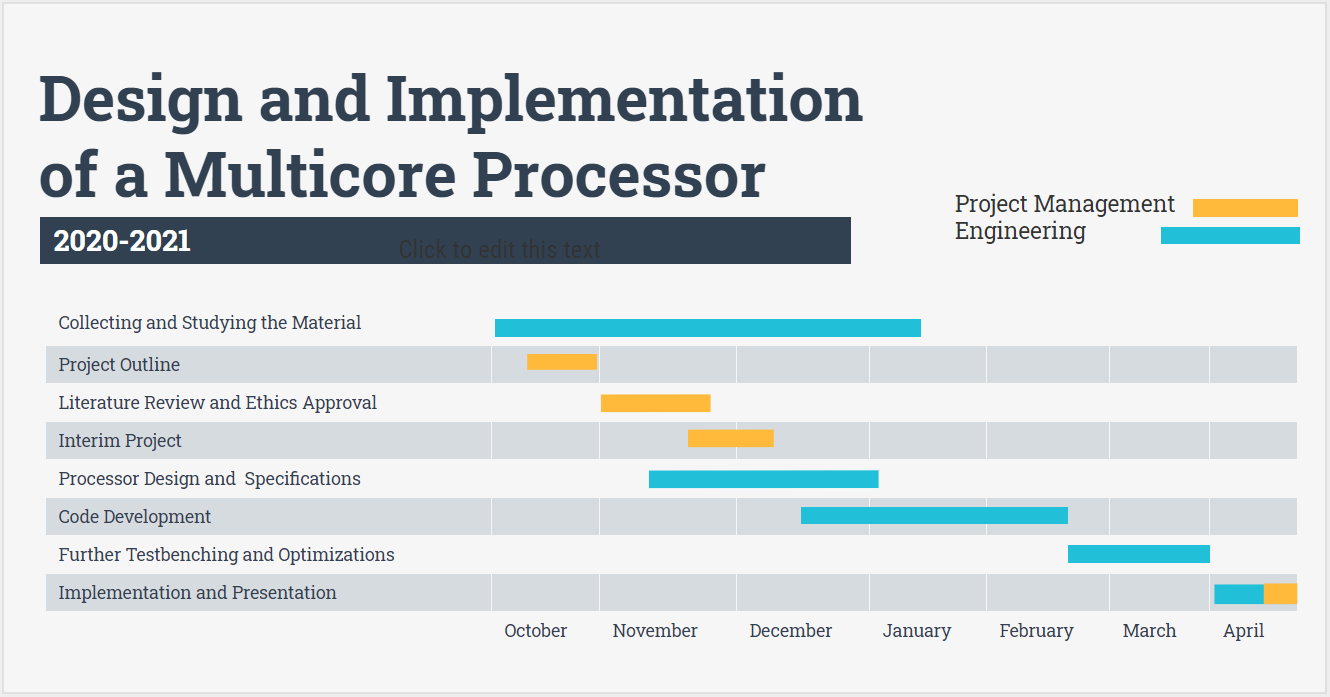
****

Figure A. 1 Original Gantt chart.

As shown in the Gannt chart, the intentions were to research the subjects at hand throughout the entirety of the first semester, including the break. The code development started and overlapped with the end of the design phase; this is due to the fact that some specifications were altered in accordance with code constraints. A functioning code was present before the end of February, however, the issues encountered during synthesis couldn’t be resolved throughout the remaining months. After unsuccessful attempts to implement the design on the board environment, the focus shifted towards the simulation results. The dissertation and presentation phases were consequently anticipated to the end of March, and lasted until the end of April.

## **A.2 ISA**

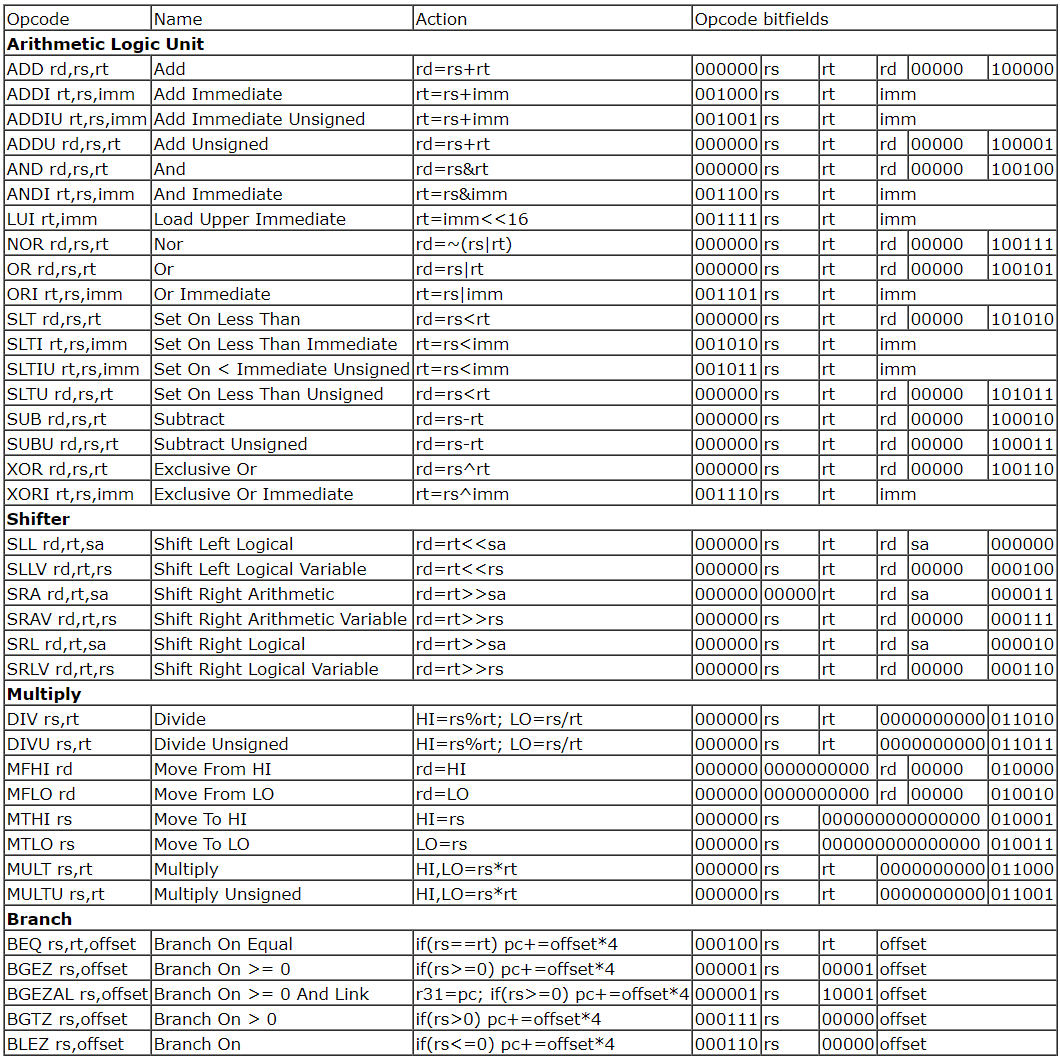


Figure A. 2 List of instructions, excluding *halt*.

## **A.3 Logbook and Meetings Material**

The logbook entries alongside all the meetings records are included in the following repository:

https://github.com/Matteo-Bovino/Logbook-and-Meetings-Material