École Centrale Lyon

Report for the cva6 softcore contest

FPGA optimisation of the RISC-V processor core Cva6

Students :
Mengqian Zou
Ariel Rodriguez Alanis
Emma Delaroziere
Antoine Waegaert

Tuteurs:
Alberto Bosio
IanO'Connor
Marcello Traiola
Etienne Dupuis

Abstract

The open-source RISC-V ISA is gaining increasing attention both in industry and academia for its terseness and performance. In this context OpenHardware launched a new initiative, developing the RISC-V based open-source processor Ariane/CVA6. However this newly born application-class processor needs improvements. A critical factor in the performance of a CPU is the ability to process instructions quickly, making efficient execution of instructions paramount. Such optimization is the focus of this project. By analysing Ariane's architecture and its critical paths some key elements that could be improved were found. Afterwards, Ariane was simulated on QuestaSim and emulated on FPGA, providing data for a detailed analysis of its performance. Insights about cache system and development on branch prediction and Execution stage are given. Cache analysis stresses relation between cache size and miss rate, as well as resource usage. After finding a compromise between performance and resource cost, cache miss rates were decreased, improving benchmark score. This project also offered the opportunity for a more in-depth understanding of this processor and the applications of RISC-V ISA.

Acknowledgements

We extend our sincere gratitude to all the persons who contributed to this project.

We would like to express our thanks to Ian O'Connor head of INL department of l'École Centrale de Lyon for giving us this opportunity to participate in the contest.

We wish to acknowledge the help provided by our supervisor, Dr. Alberto Bosio, who guided us and gave us a technical support with software installation and set up of the remote server.

We would like to express our gratitude to PhD student Etienne Dupuis and Post Doc Marcello Traiola, who guided us throughout this project with their knowledge and technical support.

We wish to express our thanks to Thales research Thechnology France,the GDR SoC2 and the CNFM for offering this opportunity to learn and explore in RISC-V processor designs.

Contents

1 Introduction

For decades, computers have progressively pervaded society and have become a necessity in fields ranging from marketing to physics. Thus, improving processing power in computer is a major stake.

Microprocessors, or Central Processing Units (CPU) are the components executing elementary operations in computers. These operations are coded in binary in order to be used by the CPU. Instruction encoding may vary from one standard to another. Those standards are called Instruction Set Architecture (ISA). ISA is a key component in defining the extent of a computer's abilities and performance. However, most CPUs on the market use ISAs which are patented such as ARM (ARM ltd) and X86 (Intel). This is an issue for organizations aiming to design their own computer chips, as it implies less freedom of modification and further fees in order to be allowed to use them. Over-reliance on patented ISAs may consequently cripple the growth of independent circuit designers and overall innovation.
In this context, RISC-V, an open-source ISA is proposed. It has significant advantages : simplicity, being a "reduced" ISA in opposition to a complex one, its customizability, as it is possible to add several modules to the base ISA, its ability to handle a wide range of instruction formats and the general community that helps its development. This project deals with electronic design based on this ISA.

This research application project is also part of the competition launched by the French defense group Thales in association with GDR SoC2 and the CNFM. The goal of this competition is to make teams of French university students and engineering students compete in order to optimize the performances of the open-source RISC-V based processor **ARIANE** (see figure 8).
The Ariane processor project was originally developed by the Swiss Federal Institute of Technology in Zurich and then taken over by the OpenHW group, which freely publishes the documentation and source code of the system.

The following report tackles the optimization of the processor Ariane with architectural analysis and simulations as well as emulations on FPGA board.

2 Technical terms

- ISA : Instruction set Architecture ; The set of instructions available to the CPU.
- CISC : Complex instruction set computer.
- RISC-V : Reduced instruction set computer V; A category of ISA using few simple instruction as opposed to CISC.
- Ariane/CVA6 : A specific CPU designed with the RISCV ISA which can run Linux.
- FPGA : Field Programmable Gate Array; A physical board on which it is possible to emulate any computer design.
- HDL : Hardware description language; A programming language used to describe and simulate a physical design.

- Assembly : Language describing the most basic instructions executed by a CPU (e.g. : loading data from a given place in memory, adding numbers from given registers...) in a way easily understood by humans. It may vary from one ISA to another.

3 Goals and specifications

3.1 Goals

The main objective of this project is to improve the Ariane core. To do so, a few steps are needed :

- Analyze system architecture to find parts that can be modified for better performances.
- Edit the SystemVerilog program describing hardware behaviour.
- Run a simulation on QuestaSim in order to get first results.
- Emulate the CPU on a programmable board to test the design in real conditions.

3.2 Specifications

This project being about taking part in a national contest organized by Thales. The contest was launched on September 25th, and is evaluated by a small report to be send before april 23th. Specifications consequently depend on contest rules :

- Increase processing frequency: the number of executed operations for a given duration, it reflects processor's performance in terms of executing speed
- Increase the score of the CoreMark[1]
- Solution elegance: the impact on the initial structure should be minimised, some existing characteristics in CVA6 such as floating operation should be kept
- FPGA resources consumption restriction: the number of LUT[2] cannot increase by more than 50% compared to reference architecture

3.3 Deliverables and deadlines

In addition to this report turned in before April 13, there are deliverables specific to this contest :

- source code of the modified architecture,
- a short report with tests results and a description of what was modified in the code.

These deliverables have to be turned in by April 23.

[1]CoreMark : a standardized test bench for CPUs. Using common operations to reflect CPU's performance, it includes list processing, matrix manipulation, state machines...The increase of the score of CoreMark reflects performance improvement[3]

[2]LUT: Look-up table, used to replace a calculation by a simpler access in the result table. It corresponds to cells in the programmable board FPGA

4 Research work

In order to achieve the objectives mentioned in the previous section, the structure of the processor Ariane was first studied in detail, including each stage of the pipeline[3]; a simulation using software QuestaSim and Vivado was run to analyse connections between different processor units. These two softwares also allowed to study the time and resources consumption in each stage in order to decide the critical path, that is, the most time consuming part, with which we fixed the direction and approaches to optimize the processor: optimize branch prediction, optimize the out-of-order mechanism in the Execution Stage and reduce memory access time by optimising cache subsystem; finally the hardware design was implemented on a FPGA board.
The whole research process is presented in details in the following sections, including the introduction of research tools(both software and hardware), accomplished work as well as encountered difficulties.

4.1 Software design & hardware design

In this project, the research work mainly develops on the hardware design, hence it is necessary to distinguish software design and hardware design.
The major differences between the two types of designs can be characterized in the target architecture, input language and execution place. As shown in figure 1, the target architecture for software design is usually microprocessors and GPU, while for hardware design, it aims at FPGA and CPLD[4] architecture. Furthermore, software designs use ASM, C, C++ as input language and are executed in microprocessors; while for hardware designs, there exist special hardware languages such as VHDL and Verilog, and they are implemented in logic circuits like FPGA board.

[3]Pipeline: a technique used in processor design which divides the execution of instructions into different steps, aiming at executing instructions in series without stop, hence increasing the execution speed and efficiency

[4]CPLD: Complex programmable logic device, a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both. The main building block of the CPLD is a macrocell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations[10]

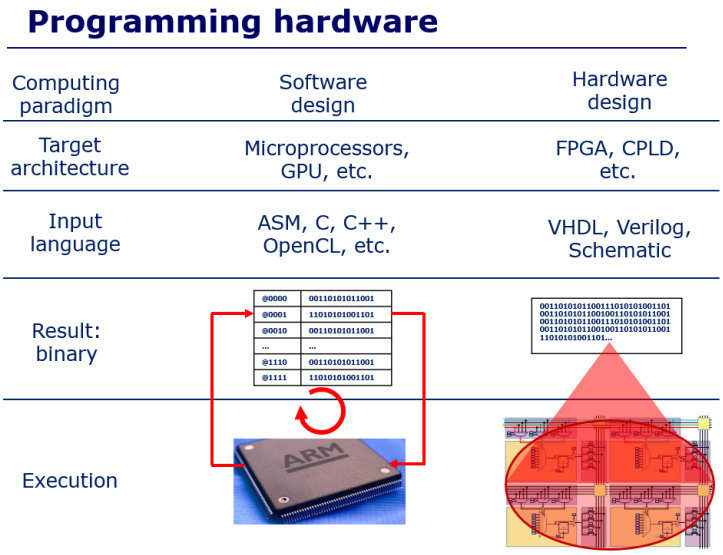


Figure 1: Major differences between software design and hardware design (*source : Centrale's courses*)

4.2 ASIC design & FPGA design

ASIC[5] design and FPGA design are two types of hardware designs, the comparison of their design flows are shown in figure 2. From this comparison, it can be noticed that the two design flows are quite similar to some extent, except that in FPGA design, the synthesis, verification, place-and-route and static-timing verification are integrated. This explains several advantages of FPGA design : programmable and fast, but also more expensive compared with ASIC design. Therefore, FPGA designs are usually used for hardware design test and verification before bulk production of processors, and that's also the reason why FPGA design was chosen over ASIC in this project. On the contrary, ASIC designs are fixed and the circuits can not be changed once implemented. They are usually used for mass production of mature and well-developed processor designs.

[5]ASIC stands for Application Specific Integrated Circuit. As the name implies, ASICs are application specific[11]

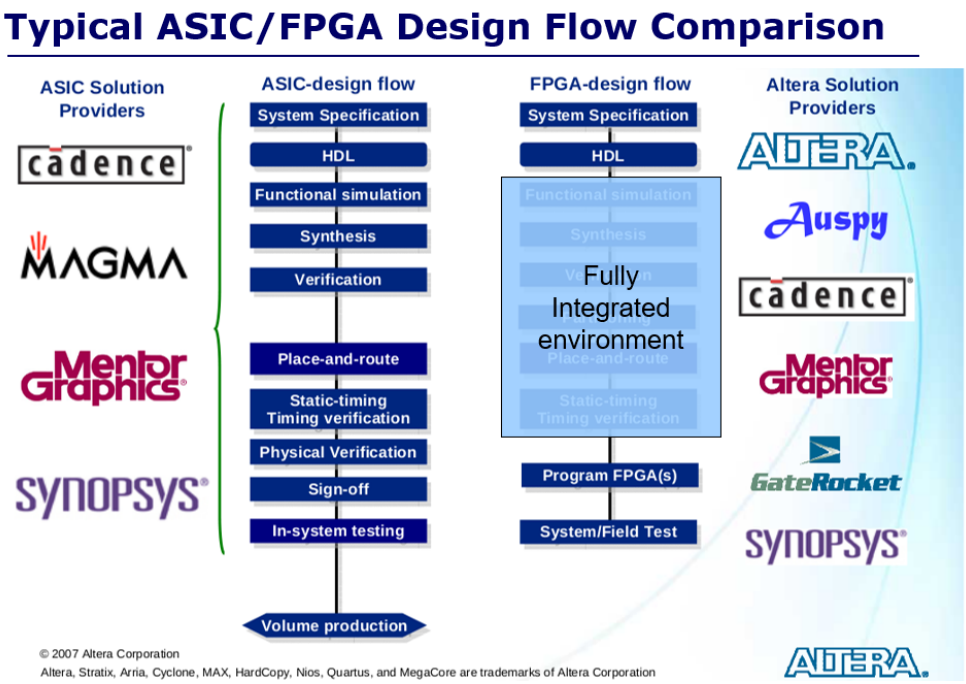


Figure 2: ASIC/FPGA design flow comparison (*source : Centrale's courses*)

4.3 Software

- QuestaSim
  QuestaSim is a software for simulation of hardware description languages such as VHDL, Verilog and SystemC. By using QuestaSim a function of hardware design can be tested more quickly, it also allows to generate different waves for different types of signals, which is very helpful in analysing the time consumption in each stage. Figure 3 gives an example of the interface of QuestaSim, where some pulses that represent the value of signals(0 or 1) are shown, the name of each signal is listed correspondingly in the left part of the interface.

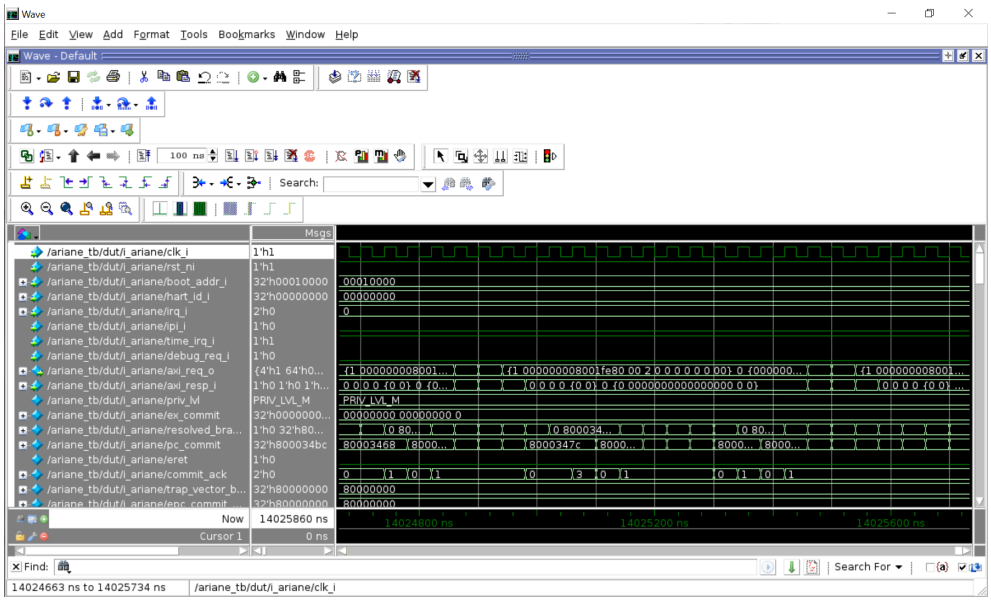


Figure 3: Interface of software QuestaSim

- Vivado
  Vivado is a software produced by Xilinx for synthesis and analysis of HDL designs. It helps verify the correctness of hardware design by realising almost the same process as the real hardware implementation. Figure 4 shows the device diagram provided by Vivado, it gives the block layout and connections according to the hardware design; when zooming in, we get Figure 5, where we can see circuit connections between different blocks

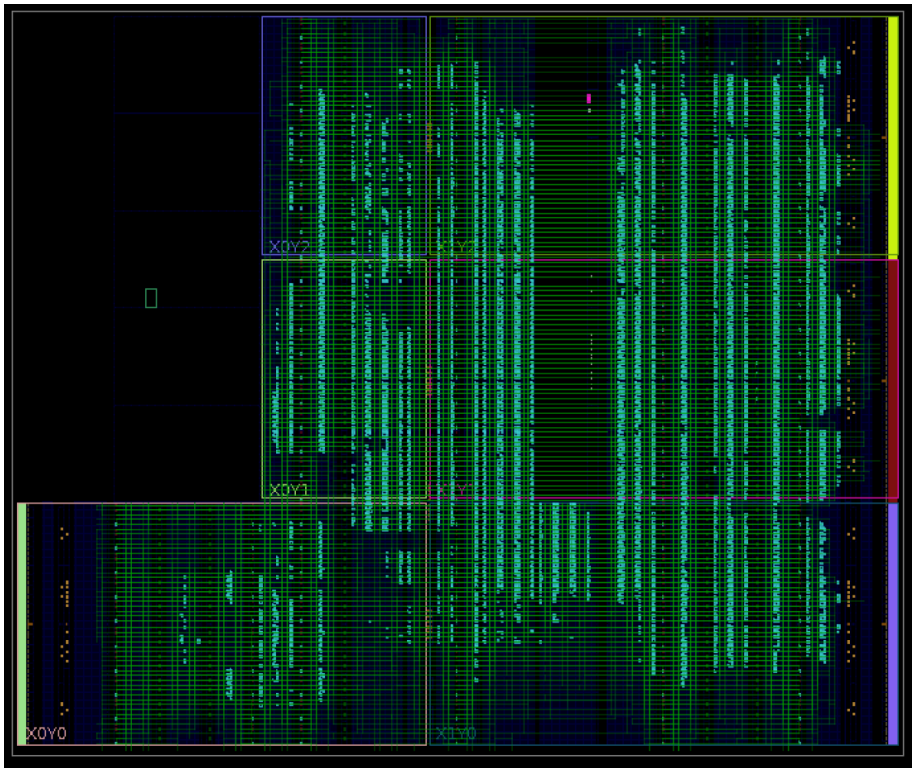


Figure 4: Device diagram in Vivado

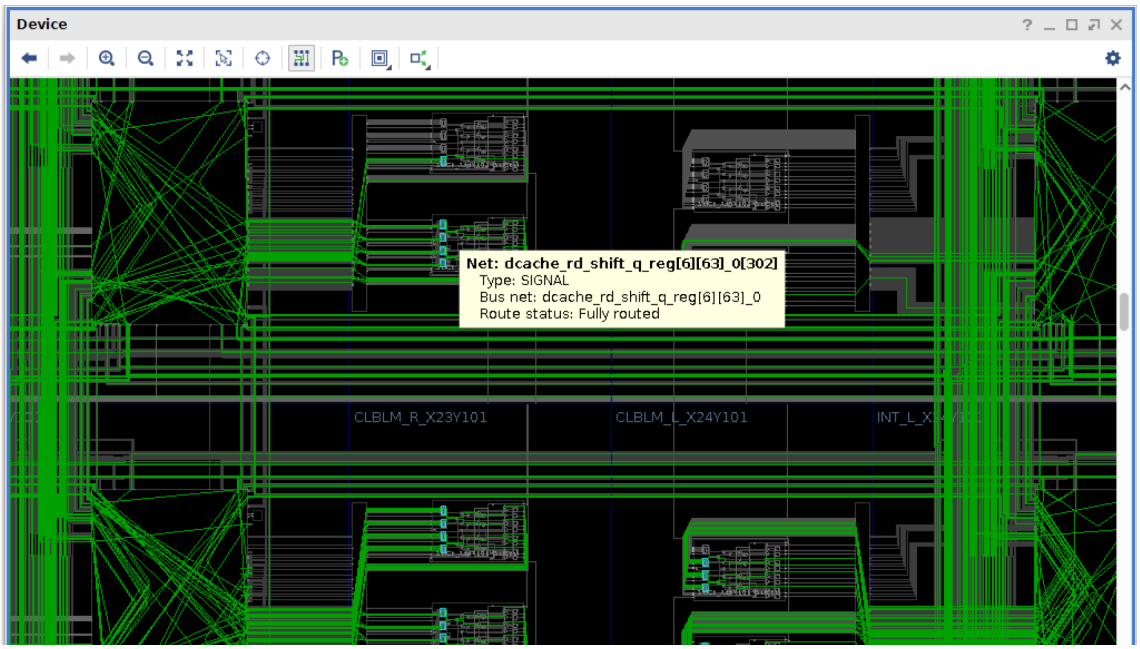


Figure 5: Zoom of device diagram of Vivado

Furthermore, Vivado can produce the RTL diagram which schematizes the hardware design's components, giving a direct way to explore the connections between different components and how the internal signals are related. Below is an example of an RTL diagram of the processor Ariane, where the green lines represent the wires connecting different parts. As shown in figure 6, the line marked in color blue transfers the value of the signal *ex_stage_i* between two components.

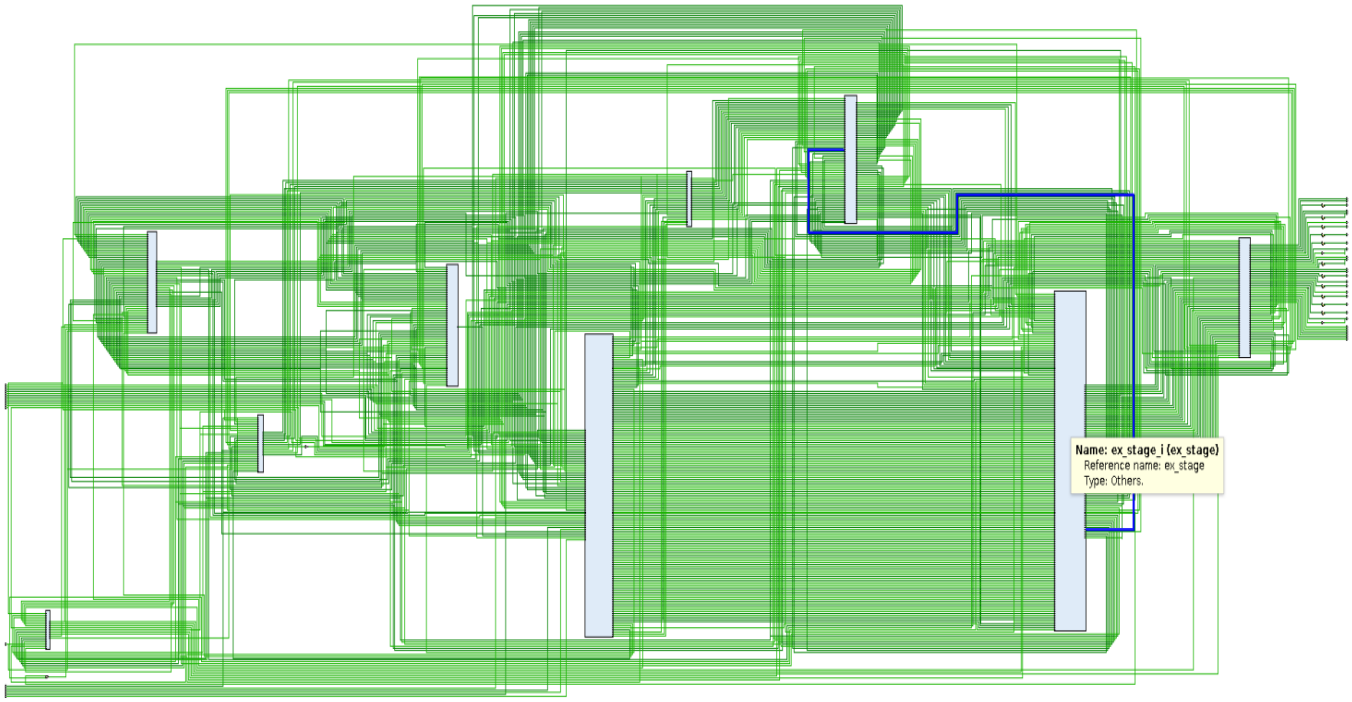


Figure 6: RTL diagram of processor Ariane in Vivado

Vivado also offers the timing reports and resources reports, in which we can find different indicators of the performance of the hardware design such as the number of used LUTs, which indicates the consumption of hardware resources.

- Toolboxes
  Two toolboxes have been used : riscv-gnu-toolchain and riscv-openocd.
  riscv-gnu-toolchain is able to cross compiled code into a risc-v binary and riscv-openocd enables us to flash a piece of code on the FPGA card.

4.4 Hardware

- FPGA
  In this project, the ZYBO-Z7 development board, produced by Digilent, was used. It is a development board with 1 GB DDR3L with 32-bit bus at 1066 MHz, tightly integrated with a 667MHz dual-core Cortex-A9 processor[15]. As shown in figure 7, the part ① is the FPGA board, which contains electronic circuits and components as well as a set of multimedia and connectivity peripherals. When the board is connected to the computer, the bitstream generated by Vivado which represents the realisation of the hardware design can be transferred to the FPGA board, completing the hardware implementation.
  Thanks to the programmable characteristic of the FPGA board, the correctness and efficiency of hardware design can be tested and evaluated quickly and cheaply before large scale production, thus reducing production costs.

- JTAG-HS2 debug adapter
  In figure 7, part ② is the JTAG-HS2 debug adapter. It is an auxiliary equipment for hardware design test and debug.[6]

- Pmod USBUART
  The Pmod USBUART is indicated in the figure 7 part ③. Unlike PCs, FPGA boards cannot display information or receive instructions directly, that is why the Pmod USBUART is needed: instructions entered in the PC are sent to the FPGA board in order to control the implemented CPU, then after executing said instructions, the CPU sends information back to the computer, allowing it to be displayed on PC screen and achieving two-way communication.

[6]Debug: run the target program under controlled conditions that permit the programmer to track its operations in progress and monitor changes in computer resources[4]

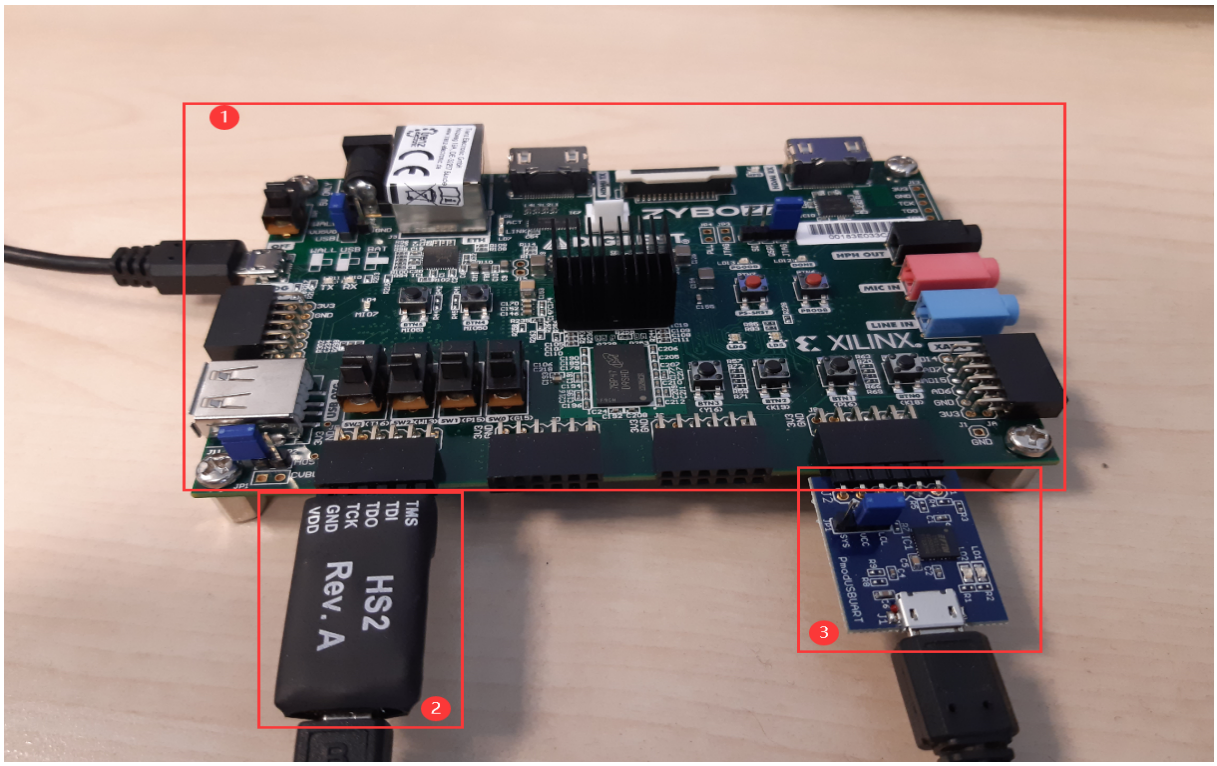


Figure 7: ZYBO-Z7 development board with JTAG-HS2 debug adapter and Pmod USBUART

4.5 Structure of processor Ariane

In this part, Ariane CPU will be introduced more thoroughly.

In micro-architecture, a pipeline is the element of a processor in which the execution of instructions is divided into several stages. By using the pipeline technique, the processor can contain several instructions, each at a different stage, thus increasing overall speed of execution [9]. Different ways of dividing the pipeline characterize different processors.

In our case, as shown in figure 8, Ariane is a processor with a 6-stage pipeline (from left to right):

1 - PC generation

2 - Instruction fetch

3 - Instruction decode

4 - Issue

5 - Execute

6 - Commit

Among the 6 steps, the PC generation and Instruction fetch form the Front-end, which has the function of deciding which instructions should be executed. The rest form the Back-end part, which deals with the execution of the instructions sent by the Front-end and the submission of results.

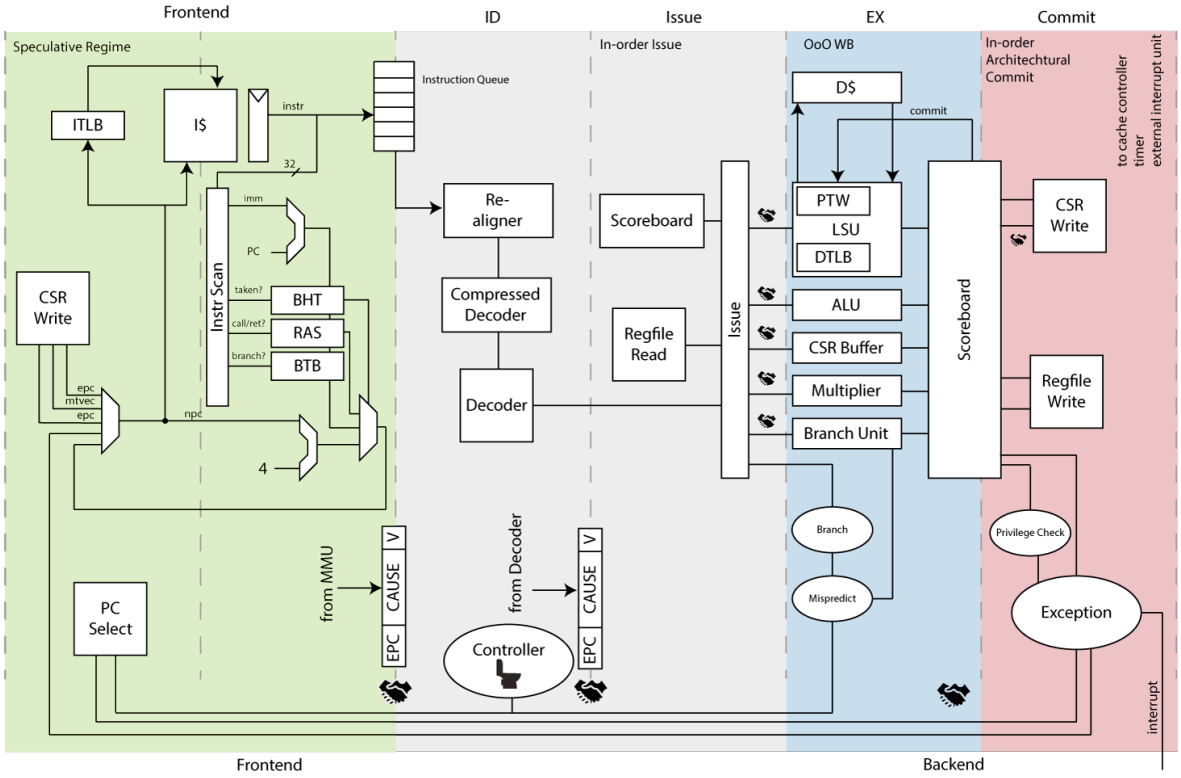


Figure 8: Ariane pipeline overview [14]

4.5.1 PC generation

PC or Program Counter is a register that allows to indicate the next instruction to take. As its name indicates the PC Generation Stage is dedicated to the selection of the next instruction. The choice is made according to the following hierarchy [12]:

1. Default assignment: increment of the PC
2. Branch Predict: BHT and BTB subsystems can predict the result of a branch
3. Control flow change request: occurs when a miss prediction happens.
4. Return from environment call :
5. Exception/Interrupt:
6. Pipeline flush
7. Debug

The branch Prediction is a system that aims to predict the result of branches, i.e. conditional instruction that can affect the value of the PC (refer to the annex 8.2 for more details). It is an essential part of a CPU, without which performance could drop by multiple factors of 10 in the worst cases! The system implemented in the CVA6 is a two-bits saturation counter. It works by increasing the counter when the prediction is successful and decreasing it otherwise. Branches are taken if the counter value is 2 or above and not taken otherwise.

In case of misprediction, the branch unit in the execution stage will communicate the result, which will force the system to execute the pipeline flush operation, removing any data from the pipeline, which is a waste of time that can negatively influence CPU performance.

4.5.2 Instruction Fetch

The Instruction fetch stage main function is to load instructions from the instruction memory, at the address pointed by the PC, toward the instruction queue. Since instructions may vary in size, it is important to know that they are no longer aligned once they enter the instruction queue. This is solved in the next stage of the pipeline.

The structure of the instruction queue is FIFO (First In First Out).

4.5.3 Instruction Decode

This processor uses 32 bites long instructions, yet some instructions can be stored in the memory as 16-bites compressed instruction. Because the processor stores all instructions in the memory as if it was 32 bites words, some instructions get unaligned.

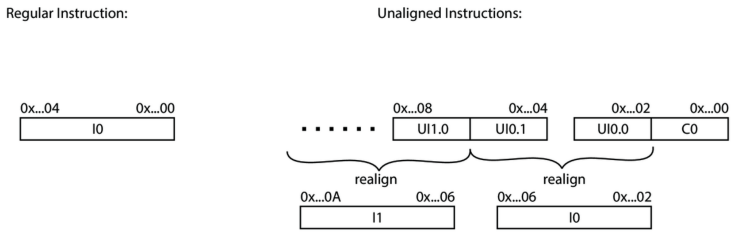


Figure 9: Realignment of instruction[12]

The instruction decode stage is subdivided in three main blocks :

1. Re-aligner : as stated instructions are unaligned in the queue ; this block takes instruction and re-align them for processing.
2. Compressed-decoder : The RISC-V ISA uses a particular form of instruction called compressed instruction, those instructions need to be decoded and translated into another to allow the decoder to process them.
3. Decoder : The decoder takes the instruction given and infer from it the data and control signal that needs to be manipulated in order to execute the instruction in the following stages.

4.5.4 Issue Stage

The Issue Stage receives the instructions decoded by the Instruction Decode stage, sends them to the corresponding functional units in the next stage and receives the results obtained, as shown in figure 10.

This stage communicates with different functional units independently giving them the corresponding operators. It follows the instructions sent and the states of each functional unit, which allows to execute the instructions without order: the operation can be

executed and submitted as soon as it is finished, without waiting for the instructions that are in front and that take more time, it is enough to submit the obtained results in the corresponding address registered in the Issue Stage. This design allows parallel execution for different units, thus increasing processing speed.

Thanks to the independence between each functional unit and the traceability of operations, it is possible to improve the performance by adding some functional units in parallel, which is taken into account in our solution proposals.

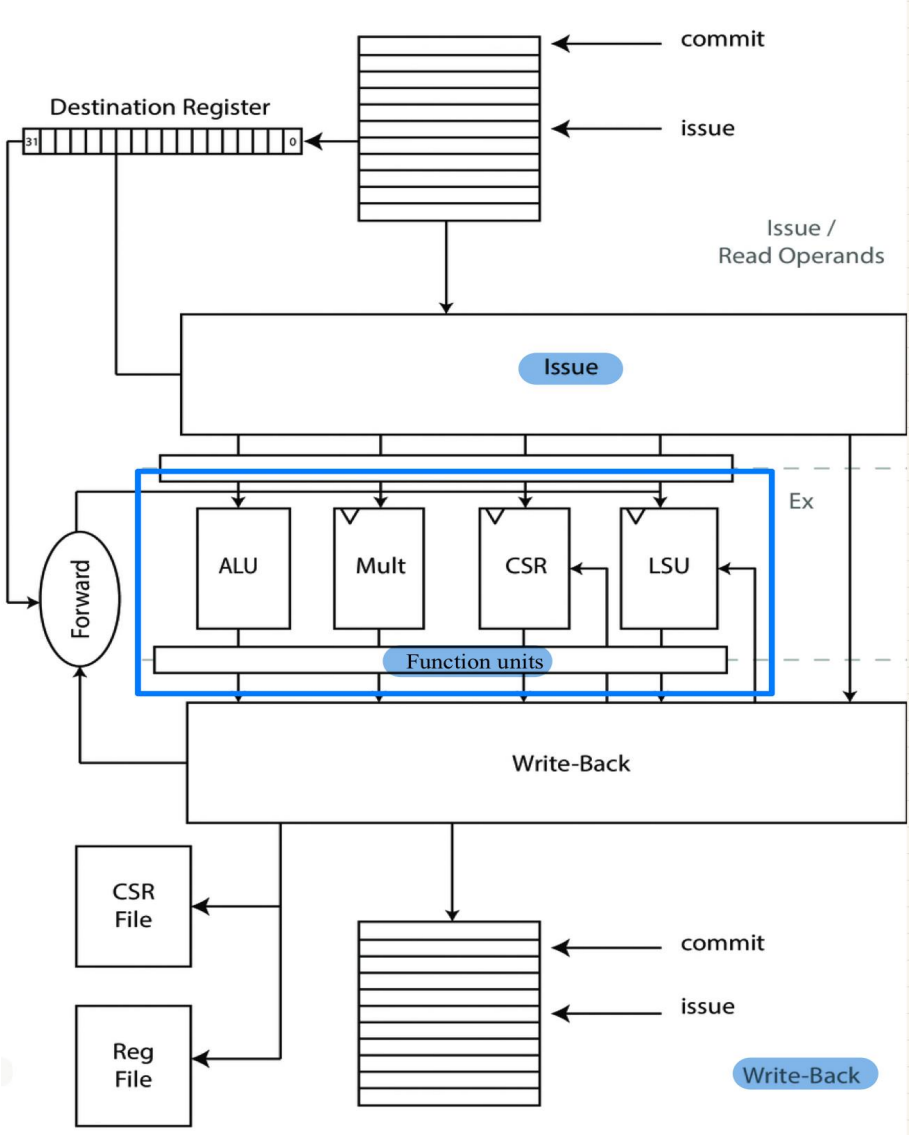


Figure 10: Issue Stage overview [12]

4.5.5 Execution Stage

The execute stage is the part that executes the instructions sent by the issue stage. This stage is composed of the following functional units[12]:

- an LSU (Load store unit) which allows to communicate with the data cache,
- an ALU (arithmetic-logic unit) which allows the simple arithmetical calculations,

- a CSR buffer (Control and Status Registers buffer) storing flags values used for conditional branching,
- a multiplier,
- a branch unit which allows to modify the execution of the instructions in the case of a bad prediction, this unit is where the critical path of the machine is located.

4.5.6 Commit Stage

The Commit Stage is the last step in the pipeline, which updates the global state of the architecture by changing the data in the CSR registers, committing the storage requests and passing the data to the universal registers. This stage also deals with all the exceptions [7] and the interruptions [8]. Indeed, the Commit Stage controls the processor's speed, if the stop signal is activated, the pipeline will be blocked, which will influence the execution speed.

A more detailed description of the subsystems is too vast for this report and it is recommended to refer to the github of the project [12].

4.6 Simulations in QuestaSim and Vivado

After a detailed study of the whole structure of the processor Ariane, we can now analyse the time and resources consumption of each stage by simulating the processor in QuestaSim and Vivado.

4.6.1 Timing analysis in QuestaSim

In order to measure timing in each stage, it is necessary to find out the essential signals in each stage, that is, the signals which represent the alternation of each stage. The signals chosen are as follows:

(1) Front-end: *i_frontend/fetch_entry_o/instruction*
This signal represents fetched instructions, hence, the alternation of this signal means finishing one instruction fetch. In comparing with the signal *clk_i*, the timing of the stage Front-end can be measured. Figure 11 shows an example of the simulation of this signal:

[7] Exception: particular conditions or exceptional conditions in the normal course of a part of a program [5]

[8] Interruption : a temporary suspension of the execution of a computer program by the microprocessor in order to execute a priority program [8]

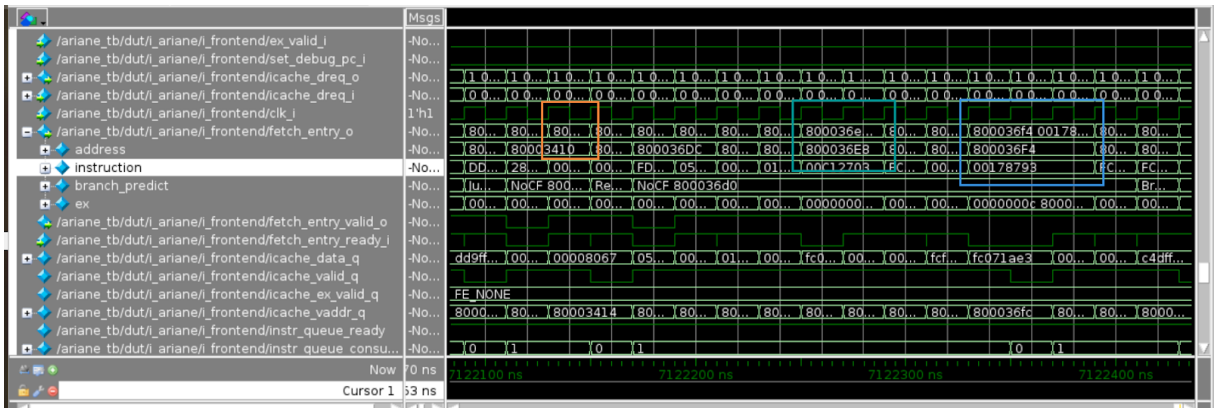


Figure 11: Simulation results of the Front End

It can be noticed that most signals take 1 cycle(one period's length of the signal *clk_i*), but there exist some signals taking 2 or even 3 cycles as it is remarked in the figure above.

2 - Instruction decode: *decoded_instruction*
Since the main function of this stage is to decode instructions, it is suitable to trace the changes of decoded instructions. This signal is actually a type of data structure which contains several signals associated with decoded instructions. As shown in the Figure 12, the signal *pc* was chosen for it stands for program counter, aiming at counting decoded instructions.
It's prone to choose the signal *instruction* which gives the abbreviation of decoded instructions as the representative signal. However, it is possible that the same instruction appears several times sequentially, thus the signal *instruction* will keep the same value for several cycles while actually it executes separate instructions.

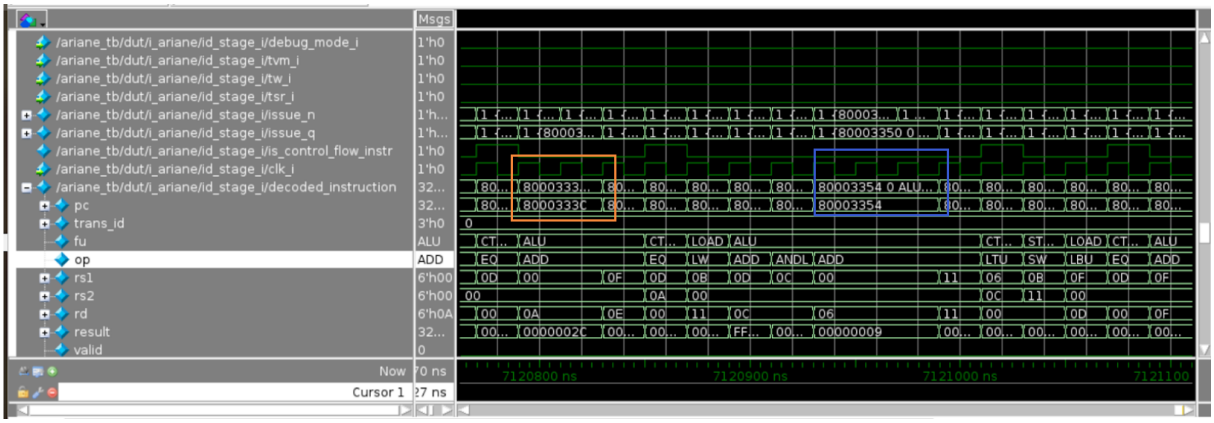


Figure 12: Simulation results of the Instruction Decode

(3) Issue: *fu_data_i/fu*
As it is introduced in the previous part, this stage is to dispatch different instructions into their respective function units for execution. In Ariane, there are 4 fixed-latency[9] (1 cycle) units, which are represented by number 1, 2, 3, 4 in the Issue Stage:

[9]It is important to distinguish between two criteria for speed measurement : latency and throughput.

1. ALU
2. Branch unit
3. CSR
4. Multiplier/Divider

Therefore, the signal *fu_data_i/fu* represents the alternation of issue destination of instructions. As it can be seen in figure 13, sending instructions to function units CSR and Multiplier (number 3 and 4) takes 2 cycles, the others take only 1 cycle.

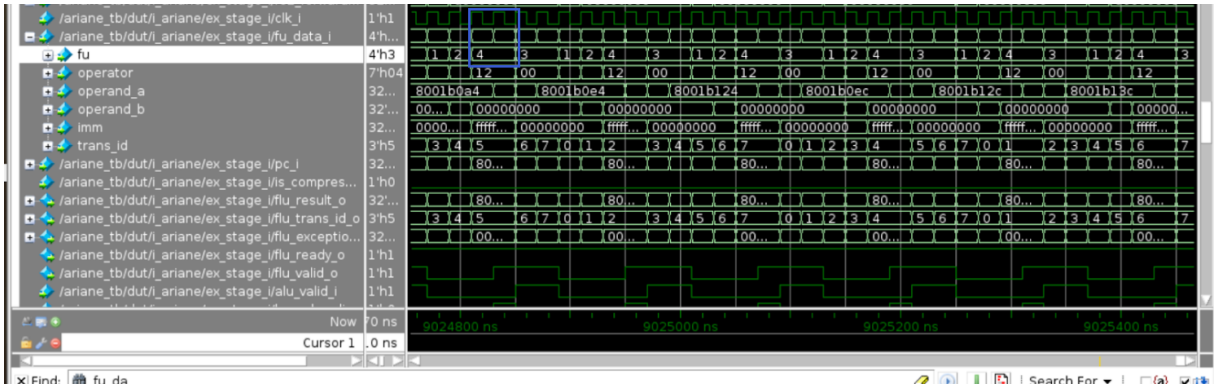


Figure 13: Simulation results of the Issue Stage

4 - Execute: *xx_result*
In this stage, instructions are executed in the 4 function units mentioned above, hence, the signal of the execution results can represent reliably the alternation of each execution process. As mentioned in the Issue Stage, the latency of those function units are fixed to 1 cycle, which is also verified in the simulation results: in figure 14, some signals are highlighted, and most of the time they all take 1 cycle. But sometimes those signals also remain unchanged for quite a few cycles. However, they do not correspond to the execution process: according to the signal *xx_data*, which represents the data of executed instructions, those function units (represented by *NONExxx*) don't execute instructions during these periods.

Latency is the amount of time it takes for an instruction to cross a given part of the circuit, while throughput is the number of instructions executed by this part per time unit.

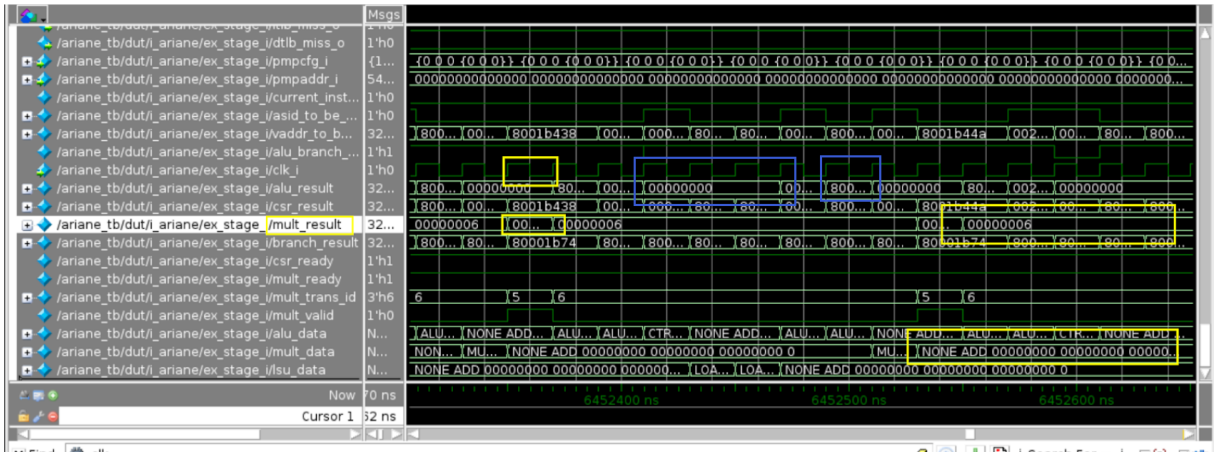


Figure 14: Simulation results of the Execute Stage

5 - Commit: *wdata_o*
The chosen signal stands for the data meant to be written in the CSR register. As shown in figure 15, the write-back process is slow, sometimes lasting 4 cycles. This can happen when the results need to be written back to the main memory, which is a time-consuming process.

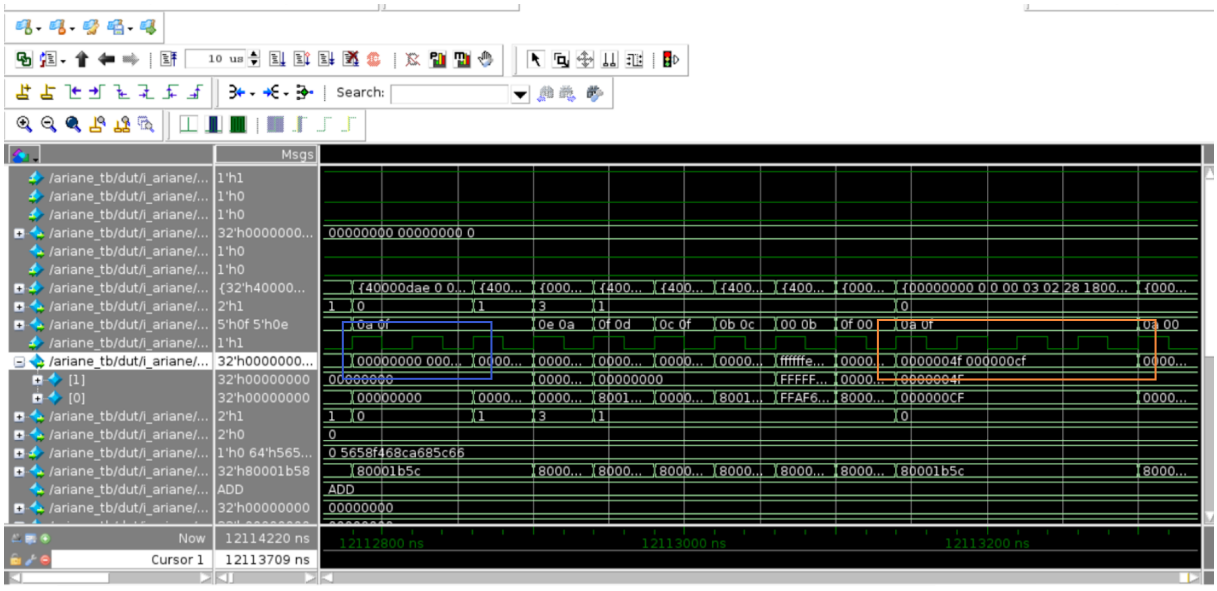


Figure 15: Simulation results of the Commit Stage

4.6.2 Resources analysis in Vivado

After the timing analysis by simulating in QuestaSim, the resources analysis can now be realised with Vivado, generally with its synthesis report and implementation report.
As mentioned before, in this contest there is a main constraint with the number of occupied LUTs, therefore, this indicator is an object of focus in the resource reports. Figure 16 gives an example of an implementation report. It can be seen that the number of LUTs is indicated clearly, here it is 20033, accounting for 37.66% of the total LUTs.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	20033	0	53200	37.66
LUT as Logic	20033	0	53200	37.66
LUT as Memory	0	0	17400	0.00
Slice Registers	10614	0	106400	9.98
Register as Flip Flop	10614	0	106400	9.98
Register as Latch	0	0	106400	0.00
F7 Muxes	2334	0	26600	8.77
F8 Muxes	230	0	13300	1.73

Figure 16: Screen shot of the implementation report in Vivado

4.7 Hardware implementation in the FPGA board

As explained in the part 4.4, a FPGA board has been used in order to emulate the processor on hardware.

As shown in the figure 17, the processor circuit needs first to be flashed on the FPGA board. To do so, Vivado is used in order to create a "bitstream". Vivado takes the SystemVerilog file, calculates all the logic gates needed to emulate the circuit and encodes it into a bitstream. This bitstream contains the circuit of the processor and can be flashed onto the FPGA board.

After flashing the processor on the FPGA board, a piece of code (here the coremark program) has to be given to the processor to be executed. The coremark program is written in C which is not understandable for the circuit, so it needs to be compiled into machine language which can be understood by our processor. Since the CPU works with RISC-V ISA, the coremark has to be compiled into RISC-V assembly code using a toolchain named RISC-V GNU. This piece of code can then be flashed into card memory with the HS2 port in order to by executed.

The output data can be read on the serial port.

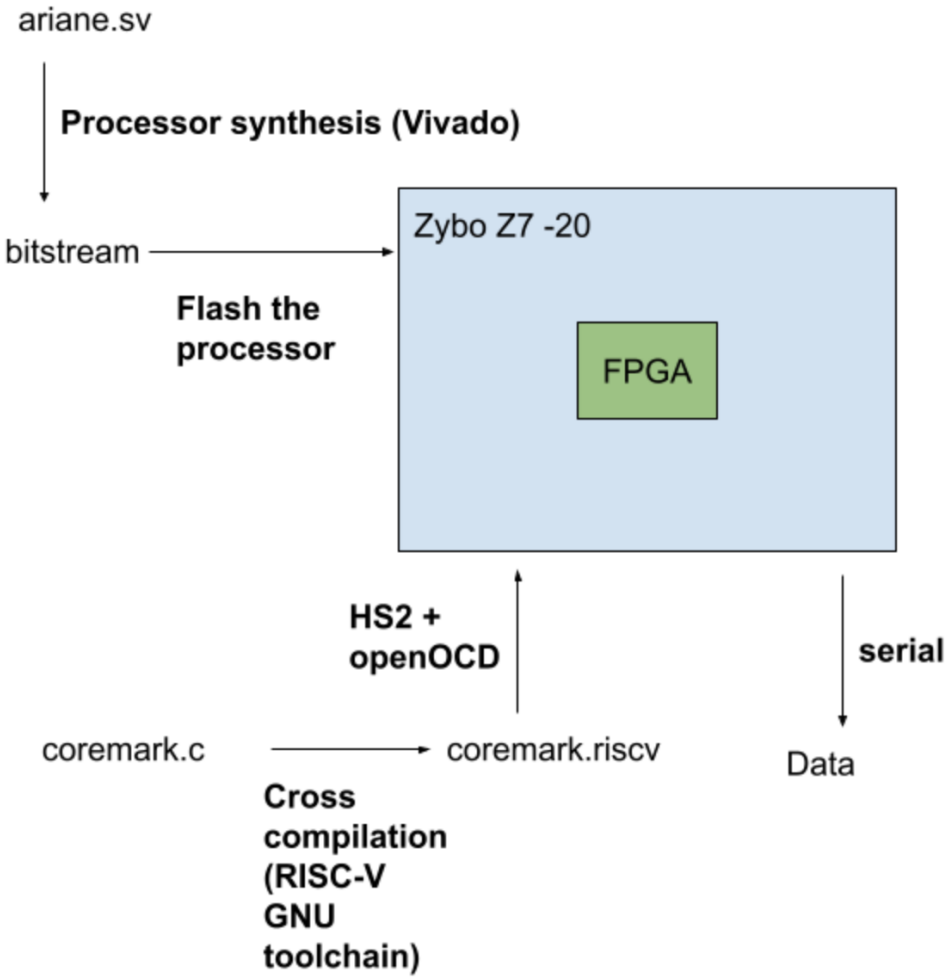


Figure 17: Workflow of the FPGA emulation

4.8 Optimization propositions

Through the analysis of signals with the help of QuestaSim and Vivado, the duration of different stages was measured, and it can be noted that some stages take several cycles to alternate. Thus, a natural research direction is to focus on these processes and try to improve their performance. In the following sections three possible optimization propositions will be presented: cache optimization, branch prediction optimization and Execution Stage optimization.

1) Cache optimization
At first, a statistical analysis of the CoreMark program was made to check what kind of operations could be worth improving.
Before running CPU simulation, the program which will be executed by the CPU has to be compiled. A trans-compiler converts C/C++ code into machine language (hexadecimal [10]) and assembly language and saves it in a .D file. This file describes how instructions are stored in memory. As seen in figure 18, code is divided in

[10]CPUs actually executes binary code, however machine language is often written in hexadecimal to make it shorter and easier to read for humans, since binary equivalent of these instructions would require 32 characters each.

paragraphs corresponding to subprograms. Each subprogram has a tag written at the beginning between brackets, so that it can be referred to in other subprograms. Every line of a subprogram matches with an instruction. The leftmost column indicates the part of the memory where it is stored (memory address). The second column to the left is the instruction coded in hexadecimal. The third column is the corresponding instruction in assembly and the last column shows instruction parameters.

```
39 80000068:       1690006f                j       800009d0 <__no_irq_handler>
40 8000006c:       1650006f                j       800009d0 <__no_irq_handler>
41 80000070:       1610006f                j       800009d0 <__no_irq_handler>
42 80000074:       15d0006f                j       800009d0 <__no_irq_handler>
43 80000078:       1590006f                j       800009d0 <__no_irq_handler>
44 8000007c:       2690006f                j       80000ae4 <verification_irq_handler>
45
46 Disassembly of section .init:
47
48 80000080 <_start>:
49 80000080:       0001b197                auipc   gp,0x1b
50 80000084:       fec18193                addi    gp,gp,-20 # 8001b06c <__malloc_top_pad>
51 80000088:       00040117                auipc   sp,0x40
52 8000008c:       f7810113                addi    sp,sp,-136 # 80040000 <__heap_end>
53 80000090:       00000517                auipc   a0,0x0
54 80000094:       f7050513                addi    a0,a0,-144 # 80000000 <__vector_start>
55 80000098:       00156513                ori     a0,a0,1
56 8000009c:       30551073                csrw    mtvec,a0
57 800000a0:       fdc18513                addi    a0,gp,-36 # 8001b048 <g_stdio_uart_init_done>
58 800000a4:       0001b617                auipc   a2,0x1b
59 800000a8:       7c860613                addi    a2,a2,1992 # 8001b86c <__bss_end>
60 800000ac:       40a60633                sub     a2,a2,a0
61 800000b0:       00000593                li      a1,0
62 800000b4:       7bc050ef                jal     ra,80005870 <memset>
63 800000b8:       00005517                auipc   a0,0x5
64 800000bc:       6c850513                addi    a0,a0,1736 # 80005780 <__libc_fini_array>
65 800000c0:       67c050ef                jal     ra,8000573c <atexit>
66 800000c4:       718050ef                jal     ra,800057dc <__libc_init_array>
67 800000c8:       00000513                li      a0,0
68 800000cc:       00000593                li      a1,0
69 800000d0:       00000613                li      a2,0
70 800000d4:       00c000ef                jal     ra,800000e0 <main>
71 800000d8:       6780506f                j       80005750 <exit>
72
73 800000dc <_fini>:
74 800000dc:       00008067                ret
```

Figure 18: Code taken from coremark.D

Through a statistical analysis of the code, it is possible to obtain an approximate overview of the most commonly used instruction types during CoreMark execution. To do so, instructions were categorized into six main instruction types [1] : instructions without effects, unconditional jumps[11], conditional jumps, arithmetical and logical operations, memory access and debugging instructions. A Python code was made to count instructions per type in a given subprogram. From the execution results, it can be concluded that the overwhelming majority of instructions in almost every subprogram are of the memory access type. Thus, improving memory access speed can be considered as a solution to improve processor speed.
Furthermore, memory interface is among the critical paths[13], particularly the tag-comparison operation in the cache(refer to annex for more details about cache). Therefore, the study and optimization of the cache is a research focus.

2) Branch Prediction optimization

As mentioned previously, the Branch Prediction algorithm in use for the CVA6 is a two-bits saturation counter, and this algorithm is a one level predictor, which is a quite primitive design.

[11] Jump : action of exiting linear execution of a program to execute instead a subprogram stored in another part of the memory

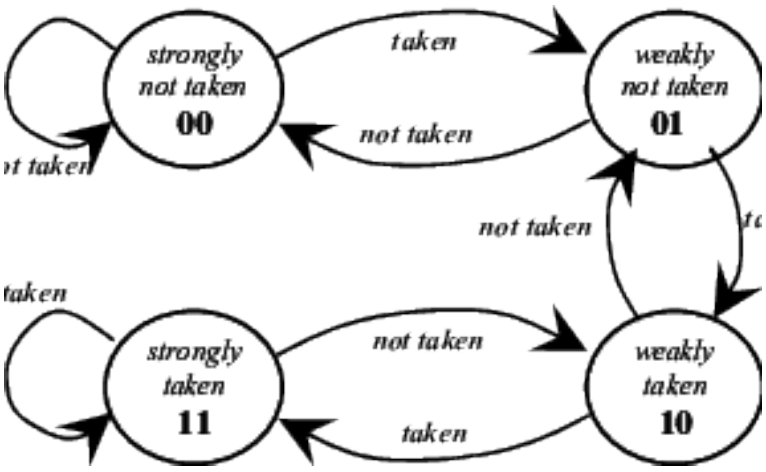


Figure 19: Code taken from coremark.D

On this aspect multiple algorithms exist, with varying performance, thus the branch prediction is also a potential research point. One option would be to switch to a two-level predictor. Such a predictor could function by using multiple two-bit saturation counter each associated with a particular history of branch, and thus would be extremely efficient at recognising loops and predicting their behaviour correctly. The state of the art models in terms of predictor uses machine learning to generate efficient predictor.

3) Execution Stage optimization
As mentioned in the section 4.5.4, all the function units in the Execution Stage work in parallel, therefore it is also possible to add some function units to optimize the performance.

The realizability and profitability of these three propositions were then evaluated.
For the Branch Prediction, it is not decoupled with other units, therefore a slight modification demands a lot of modifications in other function units with different signals. Besides, the Branch Prediction mechanism is quite complicated, with time limited, it can be difficult to optimize.
As for the Execution Stage, though function units work in parallel, according to simulation results, they are all fixed to 1 cycle, which may not have great impact on the critical path. Furthermore, adding function units also demands to modify a number of signals and block interfaces, therefore the profitability is to be doubted
Finally for cache optimization, it is more about parametric modification, and as mentioned above, the cache has a great impact on memory access speed and it is on the critical path, so optimizing the cache seems realizable and profitable

In conclusion, among the three propositions, more attention will be paid to cache optimization.

4.9 Encountered difficulties

During the research process, some difficulties were encountered, either in software or in hardware

4.9.1 Software issues

As mentioned in section 4.3 used softwares are Vivado, QuestaSim, riscv-gnu and riscv-openocd. Vivado, QuestaSim and riscv-gnu can be run on both windows and linux. However riscv-openocd has to be run on linux. This caused operating system issues and was solved with virtual machines such as virtual box. Nevertheless, system errors and unfamiliarity with Linux system hindered us for quite a long time during the research process
Another issue was the size of Vivado and QuestaSim, those two softwares were too large for some of us to be installed on our personal computers. The use of a remotely accessible server with all the tool-chains installed solved this issue.

4.9.2 Simulation issues

During some simulations, some internal signals were monitored with QuestaSim. One of those simulations was about the multiplication subsystem, more precisely about detecting when the processor was ready for another calculation with the $mult\_ready\_o$ signal. However this signal is constantly defined as high in the source code, causing troubleshooting of non-existent bugs.

4.9.3 Hardware issues

Another major problem is during the hardware implementation. Through the instructions in github of Thales, there exists 2 platforms: one using BRAM as main memory and another one using DDR conected to Zynq PS as main memory. However, during the implementation, it was found out that the DDR platform was not supported and there existed packet errors, which derived probably from the original designs. Finally, changing to the BRAM platform solved this problem.

5 Results and future research direction

As presented in the previous section, improving the performance of the cache can be a good research target. After a systematic study on the architecture of the cache, we enlarged the cache size and optimized performance with success. However this optimization is quite basic and limited. There exists other optimization methods such as adding multilevel caches, implementing more efficient algorithms in Branch Prediction, which can be future research directions.
The results obtained and discussions on them in details are presented in the following sections.

5.1 Performance overview in original design

In the first place, the original design of the processor Ariane was successfully simulated and emulated, allowing to get the following performance results.
For the implementation of FPGA, BRAM platform was used. CoreMark results are shown in figure 20, in which the most important indicator is the final score. The original design got a score of 111.766144 with an execution time 0.026842 s.

```
# [UART]: 2K performance run parameters for coremark.
# [UART]: CoreMark Size    : 666
# [UART]: Total ticks      : 1342088
# [UART]: Total time (secs): 0.026842
# [UART]: Iterations/Sec   : 111.766144
# [UART]: Iterations       : 3
# [UART]: Compiler version : GCC10.2.0
# [UART]: Compiler flags   :
# [UART]: Memory location  : BRAM
# [UART]: seedcrc          : 0xe9f5
# [UART]: [0]crclist       : 0xe714
# [UART]: [0]crcmatrix     : 0x1fd7
# [UART]: [0]crcstate      : 0x8e3a
# [UART]: [0]crcfinal      : 0x2e87
# [UART]: Correct operation validated. See README.md for run and reporting rules.
# [UART]: CoreMark 1.0 : 111.766144 / GCC10.2.0     / BRAM
```

Figure 20: Performance overview in original design

In terms of resource usage, according to the implementation report, the number of LUTs is 26670 as shown in the Figure 21.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	26670	0	53200	50.13
LUT as Logic	26592	0	53200	49.98
LUT as Memory	78	0	17400	0.45
LUT as Distributed RAM	52	0		
LUT as Shift Register	26	0		
Slice Registers	22399	0	106400	21.05
Register as Flip Flop	22399	0	106400	21.05
Register as Latch	0	0	106400	0.00
F7 Muxes	1366	0	26600	5.14
F8 Muxes	36	0	13300	0.27

Figure 21: Resource usage in original design

5.2 Optimization of Ariane's Cache

The time of data fetch during the execution is highly associated with the cache miss(refer to the annex 8.3 for more details), which will cause the processor to halt. Therefore, simulation was firstly used to analyse cache miss rates in the original design. In Ariane, there are 2 caches : D$ for data cache and I$ for instruction cache. Figure 22 shows the simulation result of the block i_perf_counters which gives statics of important performance indicators. According to the SystemVerilog code, here the signal [2819] gives the number of I$ miss, and [2820] gives that of D$ miss. As observed in the original design, I$ miss equals to 750 and D$ miss equals to 150.

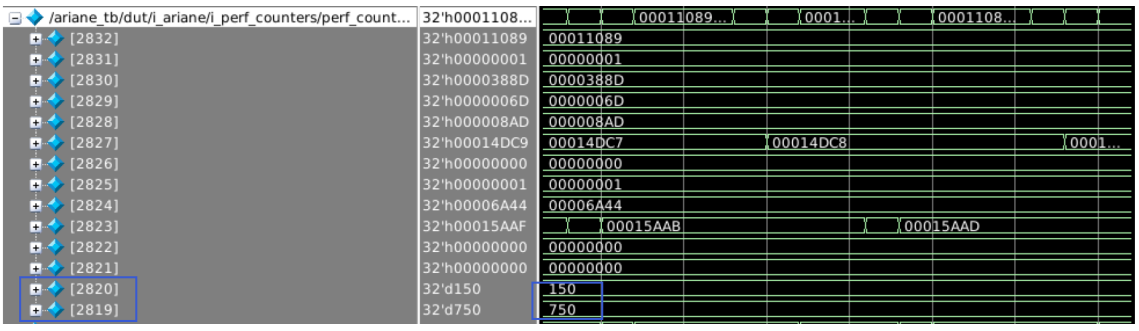


Figure 22: Simulation results of performance counters in original design

To reduce cache miss, there exist several methods such as enlarging cache size, adding multilevel caches and increasing associativity[7], cache size enlargement was chosen as it is the most simple way.
In the SystemVerilog code of the original design the following cache parameters were found :

I$ INDEX WIDTH	12
I$ TAG WIDTH	PLEN - I$ INDEX WIDTH =32-12=20
I$ CACHELINE WIDTH	128
I$ ASSOCIATIVITY	4
D$ INDEX WIDTH	12
D$ TAG WIDTH	PLEN - I$ INDEX WIDTH =32-12=20
D$ CACHELINE WIDTH	128
D$ ASSOCIATIVITY	8

Figure 23: Cache size parameters in original design

Figure 24 shows general cache organization, according to which line width can be modified to change cache size. The obtained result is shown in figure 25

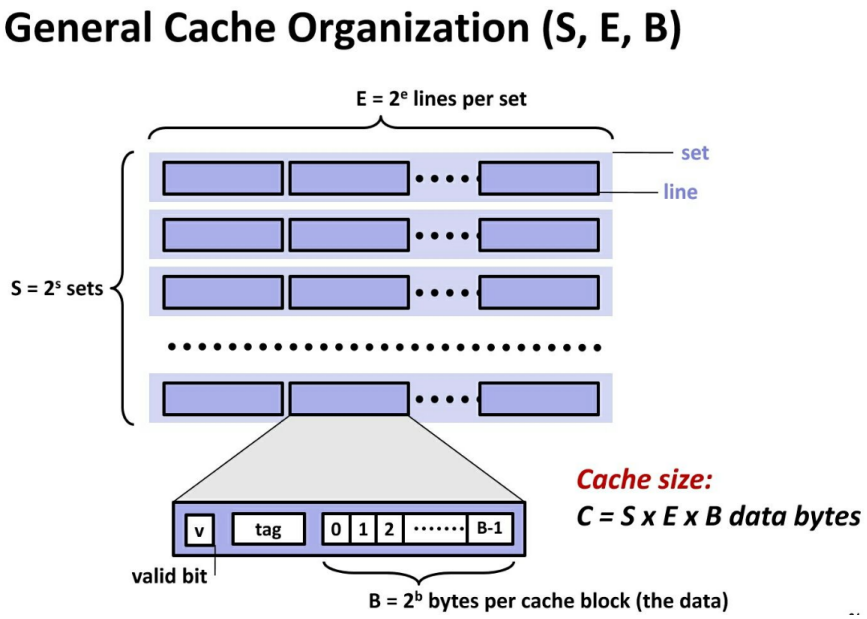


Figure 24: Architecture of the cache

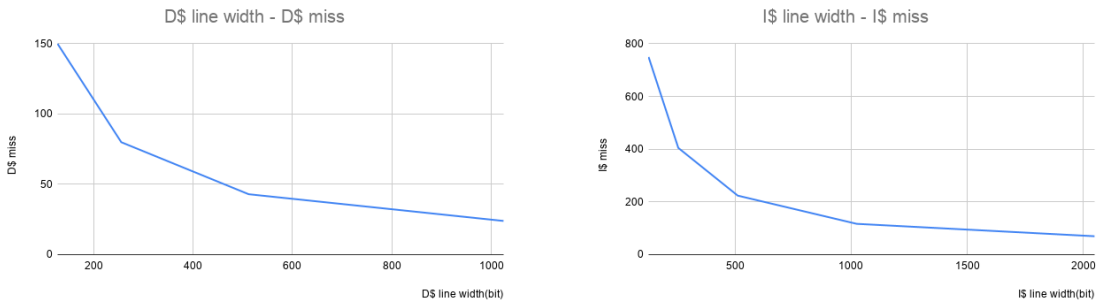


Figure 25: Relation between line width and cache miss

It can be seen clearly from the figure 25 that the cache miss decreases with the increase of the line width for both the D$ and the I$. However, concerning the score, a saturation

was observed, as shown in Figure 26: when the line width increases to 2048, the score decreases. This may due to the fact that a large cache makes it slower to find required data. To some extent, the cache behaves like the main memory.

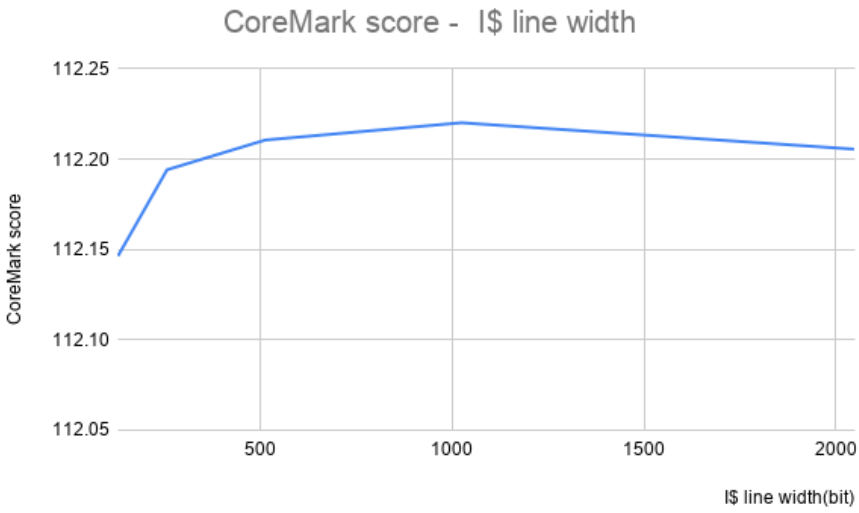


Figure 26: CoreMark score - I$ line width

As for the number of LUT, after the emulation in the FPGA board with different parameters, giving figure 27, in which the number of LUT increases with the increase of cache line width. This result is quite reasonable as enlarging cache will use more logic cells in the hardware.

Figure 27: Relation between the cache miss and the number of used LUTs

Through the experiments mentioned above, it can be observed that enlarging cache line width to enlarge cache size does reduce cache miss. However, it is a performance-resource trade-off, and there exists a saturation of the CoreMark score. Finally, through different combinations of I$ and D$ parameters, the best configuration was found : $I\$linewidth = 512, D\$linewidth = 256$, and the corresponding results are as follows:

```
# [UART]: 2K performance run parameters for coremark.
# [UART]: CoreMark Size    : 666
# [UART]: Total ticks      : 1337085
# [UART]: Total time (secs): 0.026742
# [UART]: Iterations/Sec   : 112.184341
# [UART]: Iterations       : 3
# [UART]: Compiler version : GCC10.2.0
# [UART]: Compiler flags   :
# [UART]: Memory location  : BRAM
# [UART]: seedcrc          : 0xe9f5
# [UART]: [0]crclist       : 0xe714
# [UART]: [0]crcmatrix     : 0x1fd7
# [UART]: [0]crcstate      : 0x8e3a
# [UART]: [0]crcfinal      : 0x2e87
# [UART]: Correct operation validated. See README.md for run and reporting rules.
# [UART]: CoreMark 1.0 : 112.184341 / GCC10.2.0     / BRAM
```

Figure 28: Performance overview after modification of the cache

Site Type	Used	Fixed	Available	Util%
Slice LUTs	29121	0	53200	54.74
LUT as Logic	29043	0	53200	54.59
LUT as Memory	78	0	17400	0.45
LUT as Distributed RAM	52	0		
LUT as Shift Register	26	0		
Slice Registers	23007	0	106400	21.62
Register as Flip Flop	23007	0	106400	21.62
Register as Latch	0	0	106400	0.00
F7 Muxes	1525	0	26600	5.73
F8 Muxes	68	0	13300	0.51

Figure 29: Number of LUTs after modification of the cache

From figure 28 it can be seen that the score increases to 112.184341 with a shorter execution time 0.026742 s. Figure 29 shows that the number of used LUTs is 29121. The size increase is less than 50% of the original size (26670), so the constraints of the contest are also respected.

5.3 Future research directions

Though the modification of cache size has optimized the processor's performance, the improvement is still quite limited. In terms of solution elegance, it is also rather primitive. For future research, more optimization options can be attempted.
The Branch Prediction is a potential optimizable point, particularly with the maturity of various algorithms. As for the memory access performance, there also exists room for improvement, for example frequently used operations like Load and Store operations, address translation and some in-order designs[13]. Some existing comments in the SystemVerilog code also suggest that parallel issue stages could be added, though the increase of resource consumption and the difficulty of modifying instruction dispatching without causing errors are not easily estimated.

6 Conclusion

For this project, the processor Ariane had to be improved while maintaining existing functionalities and minimizing resource consumption. The CPU, described with SystemVerilog, was simulated on QuestaSim and emulated with a FPGA board to estimate its performances.

After analyzing the behaviour of the reference architecture, it has been decided to focus on cache structure in order to improve CPU performance. This modification successfully increased the CoreMark score. Although, in the end, modifications made on the processor are rudimentary, this research project has been an opportunity to learn about CPU architecture in depth, especially the inner workings of pipelined processors, and about tools used to design electronic circuits.

Further modifications could still be done in several aspects and could be research subjects : better branch prediction algorithms could be chosen, memory access could be improved even further and some stages could be multiplied and run in parallel.

7 Bibliography

References

[1] andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. May 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf.

[2] "Cache Placement Policies". en. In: *Wikipedia* (Mar. 2021).

[3] "Coremark". en. In: *Wikipedia* (Sept. 2020).

[4] "Debugger". en. In: *Wikipedia* (Mar. 2021).

[5] "Exception". fr. In: *Wikipédia* (Jan. 2019).

[6] *Figure A. A Three-Input Lookup Table (3-LUT) FPGA. A Programmable...* en. https://www.researchgate.net/figure/Figure-A-A-three-input-lookup-table-3-LUT-FPGA-A-programmable-interconnect-wires-the_fig1_2955257.

[7] John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Sixth edition. Cambridge, MA: Morgan Kaufmann Publishers, 2019. ISBN: 978-0-12-811905-1.

[8] "Interruption". fr. In: *Wikipédia* (Mar. 2020).

[9] "Pipeline (architecture des processeurs)". fr. In: *Wikipédia* (Oct. 2020).

[10] "Programmable Logic Device". en. In: *Wikipedia* (Mar. 2021).

[11] Rohit Singh. *FPGA vs ASIC: Differences between Them and Which One to Use?* en. https://numato.com/blog/differences-between-fpga-and-asics/.

[12] *ThalesGroup/Cva6-Softcore-Contest*. Thales Group. Jan. 2021. URL: https://github.com/ThalesGroup/cva6-softcore-contest/.

[13] Florian Zaruba. "Ariane: An Open-Source 64-Bit RISC-V Application- Class Processor and Latest Improvements". en. In: (), p. 25.

[14] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2019.2926114. arXiv: 1904.05442.

[15] *Zybo Z7 - Digilent Reference*. https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start.

8 Technical annex

8.1 What is an FPGA

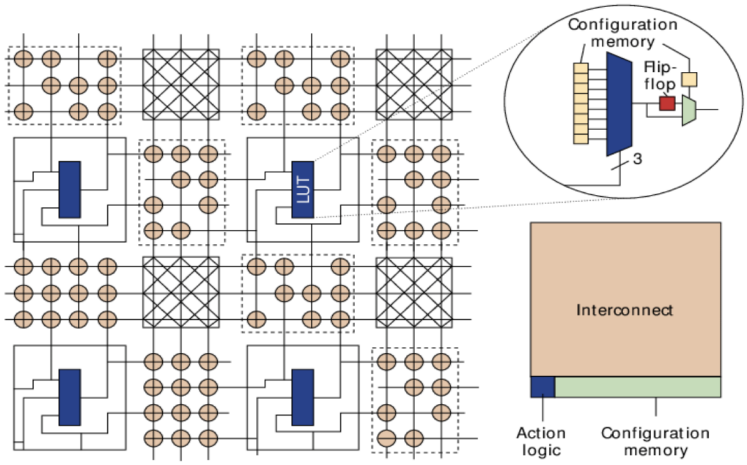


Figure 30: Schematic of an FPGA [6]

FPGA stands for Field-Programmable Gate Array, it is an large matrix of components named LUT (Look Up Table, sort of programmable logic gates) that can be wired together in order to emulate the behavior of a circuit.

FPGA are used for circuits emulation, it can give accurate results on the circuit behavior without having to manufacture the circuit on silicon.

8.2 What is a branch

Branches take place in basic programming structures like :

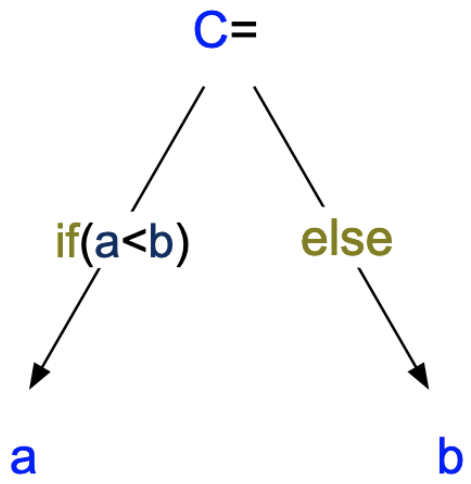
- if statements
- for and while loops

Example: This is a fairly easy C program to understand using an if statement:

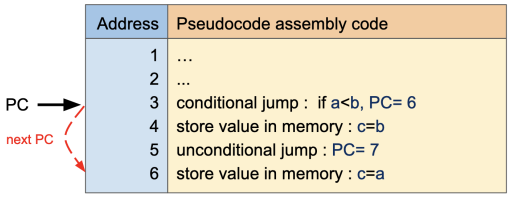
```
int a=2;
int b=3;
int c;

if(a<b){c=a;}
else {c=b;}
```

Once run, $c = 2$ is obtained.

This program can be broken down like in figure 31a. There is the if branch and the else branch.



(a) Branches of the program



(b) Assembly code illustration of the program

Figure 31: Branches illustrations

Branches can be spotted in the assembly code with jumps, instruction that changes the value of the Program Counter, like in the figure 31b.

8.3 What is a cache memory

1) Introduction to cache memory

During the execution in the processor, instructions and data are needed, however, they are usually stored in the main memory, which is very large and far away from CPU, thus fetching data can be very time consuming. To resolve this problem, cache mechanism is proposed. Thanks to the widely applied locality principle [12], it is possible to restore some data considered to be used soon in a temporary memory, that is, the cache to reduce the data fetch time.
Cache memory is the nearest memory to the processor, information in the cache memory is provided by the main memory as illustrated in figure 33. In addition, as shown in the example in the figure 32, memory slows down and get larger as we move away from the processor.

[12]Locality principle: the principle which gives information about the data that will be needed soon. It contains temporal locality which tells that a word is likely to be needed soon and spatial locality which tells that the other data in a certain block will be needed soon

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD

Figure 32: The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server [7]

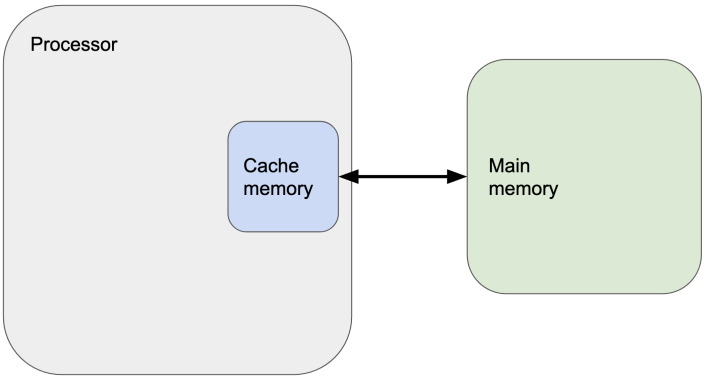


Figure 33: Caption

It is possible that the data needed is not in the cache, which is call a "cache miss", if the data is found, it's called a "cache hit"

2) Cache placement policies

The cache placement policy determines how data from the main memory can be placed on the cache memory. Their is three common placement policy :

- Direct mapped
- Fully associative
- Set associative

Figure 34 illustrates those policies.

In our project we focused on the set associative cache placement policy because it is the most commonly used and the native policy of Ariane.

To summarize, in our case cache is divided into sets and data from the main memory can only be mapped to only one specific set (Figure 35)

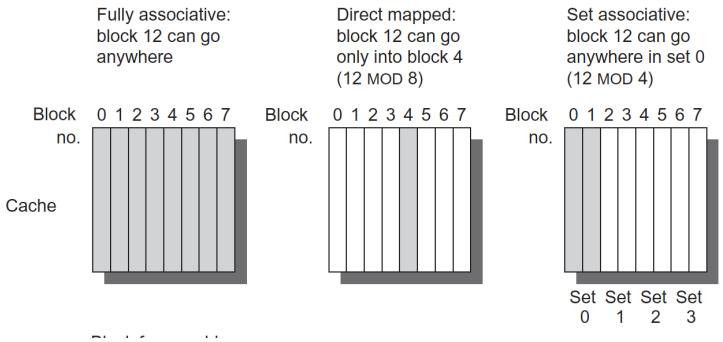


Figure 34: **This example cache has eight block frames and memory has 32 blocks.** The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called two-way set associative. Assume that there is nothing in then cache and that the block address in question identifies lower-level block 12. [7]

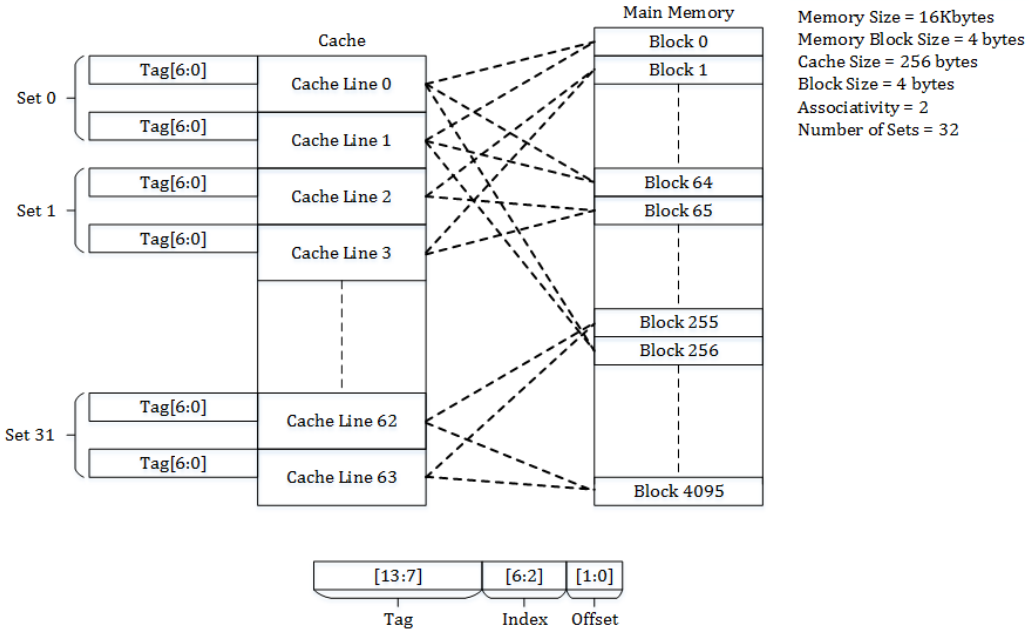


Figure 35: Another example of a set associative cache, Set 0 can only be mapped to block 0, 64, 128.. of the memory, Set 1 can only be mapped to block 1, 65, 129.. and so on [2]