Research and education platform based on a multicore pipelined RISC-V architecture implemented on FPGA.

Iván Ricardo Diaz Gamarra

Pontificia Universidad Javeriana
Facultad de Ingeniería - Maestría en Ingeniería Electrónica
Bogotá D.C., Colombia
2024

Research and education platform based on a multicore pipelined RISC-V architecture implemented on FPGA.

Iván Ricardo Diaz Gamarra

Degree work presented as a partial requirement to qualify for the title of:
Magister or master's in electronic engineering

Director:
Diego Méndez Chaves, Ph.D

Co-Director:
Eduardo Andrés Gerlein Reyes, Ph.D

Emphasis:
Embedded systems and IOT.

Pontificia Universidad Javeriana
Facultad de Ingeniería - Maestría en Ingeniería Electrónica
Bogotá D.C., Colombia
2024

(Dedication)

*T*o my parents, for their multiple efforts, for encouraging me to go beyond, and for their believe in me during this process. To Diana Marin, for her permanent support and for helping me go through each stage of this idea.

Acknowledgments

I would like to express my gratitude to the Pontificia Universidad Javeriana and its professors, for all the knowledge and guidance given to me, which were the starting point of most ideas in this project.

I wish also to thank god for all the inspiration, illumination and for letting me through the difficult times, to let this project be completed. To my parents and my cat for cheering me up when things did not worked as expected, and finally I would like to thank Diana Marin for lending me a hand when I needed it most and for giving me ideas when times were hard.

Abstract

Complex algorithms used by scientists require high-performing processors, which can be implemented in many ways using different architectures and components. This document showcases the implementation of a RISC-V processor with a variable pipeline architecture, enabling the processor to switch between pipelined mode and monocycle mode to leverage the strengths of each. The most important simulations are presented alongside the main features of the implementation.

Keywords: VHDL, pipeline architecture, monocycle core, multi core, cache memory, RISC-V, open-source.

Contents

Figure List

Table List

Chapter 1

Introduction

Our society nowadays relies on electronic devices to perform most task, from house chores such as vacuuming, doing laundry, and keeping our food from rotting to leisure activities like gaming, streaming, or chatting with friends and even most important tasks require electronic devices, like healthcare, finances, research, and transportation. As this has become one of the most important pillars of society, is important to wonder about what gives us such benefits, and the heart of most of the modern electronic devices that we rely on is the processor. [1] Computers, cell phones, home appliances, cars, planes and even in some cases entire houses depend on processors to do their duties, and therefore the way a processor is made, and its performance is vital to our day-to-day life. In this scope people always want the fastest processor available, even for devices that do not require too much power, but , there is a specific case in which the difference between a common processor and one built to be efficient and with a higher performance is evident, one easy example of such case is in research algorithms. Research algorithms are generated when a group of scientists craft a series of steps for a specific end, for instance to analyze data or to simulate a behavior, usually said algorithm uses several arithmetic operations and has multiples parts that requires a superior performance to be completed in a reasonable time, and therefore a good processor is usually the answer to the problem. [2]

But what makes one processor superior to another? The answer depends on what parameters are important to the user, for instance some users prefer processors that are efficient with its energy usage, as they prioritize a long battery life for their devices or lower prices on their electrical bill, while others prefer to have raw power, by having more instructions executed per second or multiple instructions being executed at once [3]. As previously stated, researchers use long and complex algorithms with multiple long instructions, such as multiplications of matrixes, thus they will always prefer a processor capable of finishing its algorithm as fast as possible.

However it is not only necessary to have fast processors, but also is important knowing how

to make them, therefore creating one with the required features to make it fast and document the process to allow others to follow the steps and learn from them is what motivated this project. Furthermore many students are interested in the topic and a documented process of design and implementation could be a good push in the right direction. On the same page, having a fully documented and open-source processor is also beneficial for those who want to edit it, to add more features or to get more performance by replacing existing parts of the final implementation by others with different benefits to achieve a processor in the right spot as needed.

To further understand what is needed from a processor a state of the art was made in chapter 2, where some other completed processors are analyzed, as well as the different techniques and possibilities used to get a processor with the desired performance. Before starting the design of the project it was necessary to define some characteristics such as the instruction set architecture (ISA), and as the specific purpose was completely open and the idea is to allow future users to modify it, the selected ISA was RISC-V, an open-source ISA known for its modularity and support from big companies such as Intel, AMD, and Google. Other characteristics that were defined using the available information were the possibility of making this a pipelined processor, the number of cores and the memory hierarchy. With the knowledge of which instructions would be executed and the other processor's key features the design was started using a top-down methodology as depicted in chapter 3. To ease the process and to get a better understanding of how the processor works, each part of it was encapsulated into blocks, with specific features, duties, and ports, and then each block was designed into other blocks with the same characteristics. After the design was completed the implementation of said design is shown in chapter 4. The implementation was made using VHDL as the main hardware description language, but describing the processor was not the only challenge as there was a need to test each block as its implementation was finished. To do so most test were automatized using CSV files and Model Sim to complete them thoroughly and with ease. Once the processor was complete the next part of the project was to find a way to input the desired program into it, and to do so an assembler program was made in C, that read an ASM file and convert it to the desired format for the processor's memory to store.

With all those pieces completed, the last part of the project was to make the relevant tests to check if the processor is working correctly, to do so each instruction was executed a few times, and even in some cases multiple combinations of instruction were evaluated to ensure that every case was handled. In chapter 5 those tests are shown, and each key characteristic of the implementation and ISA is shown functioning as expected.

The final product of this project is the methodology of implementation of the open-source dual-core processor with a variable pipeline architecture that uses the robust and versatile RISC-V ISA, as a simple tool and a first version that could be enhanced to tackle the challenges of running demanding research algorithms and also as an educational source to

learn the design and implementation process of a processor with these characteristics. Since it is implemented in a hardware description language, it allows edits as needed to achieve a particular required performance and also let a proper simulation to be done to verify that every instruction is working as expected. The author decided to name this processor ARK-I.

Chapter 2

State of the art

Achieving a higher speed on a processor depends on several features like its clock frequency, implementation and instruction set architecture. Increasing the clock frequency is an easy way to get a processor to go faster; however, depending on the implementation the core has a maximum frequency, determined by its technology and components, if for some reason a processor tries to run at a higher frequency than its designed maximum it would malfunction, either by miscalculating the result of certain instructions or even by skipping entire processes [4]. In the design and implementation process of a processor it is vital to set an operating frequency or in some cases a window of operating frequencies, these values are calculated using various techniques such as finding the times that a specific datapath takes to finish, starting from a stable input and finishing into a registered output, but it also depends on how the processor was implemented [5]

Therefore, the implementation is arguably the most important feature of a processor, and it comes hand to hand with the instruction set architecture (ISA) which describes the possible instructions that could be executed by the processor, this basically limits what resources must be present, and in how the software must be written, for instance, a processor that has a native multiplication instruction, and the corresponding hardware multiplicator will always outperform a processor without those capabilities [6].

As the ISA determines many aspects of the implementation of the processor, is a key aspect of how it will perform, and therefore is one of the first major decisions that must be made. One of the ways of thinking the ISA is based on the specific purpose the processor is made for, for instance if the code that will execute the processor uses several matrix operations, having dedicated instructions that perform said operations will generate a better performance in that area, but on the other hand, if that same processor runs a code that uses FFT as its main operation many times, it could end having a worse performance. An ISA that has complex instructions, sometimes dedicated to specific purposes, which do many steps in different areas of the core to complete a single goal is known as a Complex Instruction Set

Computer (CISC). Another option proposed for the idea of an ISA is by making it as simple and generic as possible, in this case the plan is to have mini-instructions that do one or two general tasks, but merging many of those instructions allows the programmer to write codes to execute complex algorithms. A core that has this kind of instruction set is known as a Reduced Instruction Computer (RISC) core, and its main advantage lies in its versatility, as no instruction is bound to a single application, but it usually requires longer codes than a CISC core. Is important to consider that in terms of speed both CISC and RISC cores are nowadays almost equal in performance, and both have their own cases when they overtake the other [7], [8] .

There are many ways of RISC ISAs but one of the most noteworthy in recent times is RISC-V, initially proposed by the Par Lab at Berkley in 2010, it is an open-source ISA released by the RISC-V foundation in 2015. Its status as an open-source ISA is one of the key aspects of its success, as it means that anyone can make implementations, contribute, and use the standard freely. The already mentioned foundation supports the different versions of the ISA and gives the necessary information about the standard and its limitations. To help this effort and to further develop the standard several major companies in the field are supporting the RISC-V foundation such as Intel, Microsoft, ZTE and IBM among others [9].

The RISC-V standard starts by setting two possible word sizes, 32 bits and 64 bits, and all instructions are available to both sizes; both word sizes are the most common possibilities for modern processors. This is part of one of the main features of RISC-V, its versatility. As another key characteristic of that adaptability the standard does not have a list of instructions set, but rather many modules with different groups of instructions. The idea of using modules allows the user to select the desired group of instructions that better fits their needs, letting them be in total control of what resources are present and, if necessary, also leaving a gap for improvement or expansion, all within the same regulations [10]. Some of the most known modules are the A, D, E, F, I, and M modules. Each one tackles a different problem and has the required instructions to do so. The A module contains atomic instructions, that are executed from start to finish without letting any kind of exception to interrupt its execution. The F and D modules contains the means to operate numbers with floating point, and their difference is that the first is used for numbers of standard longitude, while the second is used to operate decimal numbers with double precision. The M module encapsulates the multiplication and division instructions, in all their forms including signed and unsigned operations. While the E and I modules contains the basic instructions needed to run a code, such as arithmetic and logical operations, flow control, register operations and memory access [11]. There are many other modules, and the growing community submits their own modules using the standard-defined letters reserved exclusively for user-made instructions and modules. Furthermore, the standard does not limit how each instruction must be executed, allowing different implementations with different performance and characteristics, thus an implementation could have the M module but how it executes the multiplication

is up to the core's designer, in that order of ideas there are many control and status registers (CSR) described and ready for implementations with multiple access levels such as machine, user and supervisor, that limits the read and write permission to certain CSRs, but those registers also gives support to multi core implementations, exception management and even some operative system features, thus their implementation is done depending on the needs of the processor and the designer's considerations. [12].

Continuing with the key aspects of the processor performance, is important to consider how the processor executes instructions and if it is possible to execute multiple instructions at the same time, for the latter the solution is to simply use more cores. A core is the main part of a processor capable of fetching one instruction at a time, therefore having multiple cores allows the system to get as many instructions as cores at the same time, and then execute them independently, effectively increasing the speed of the processor. However as each core has its own private resources this also means that having more cores ends up using more chip area and therefore ends up generating expensive processors [13] . Even if the previously stated theory suggests that having more cores would always give a better performance in many cases that is not completely true, due to the fact that each core is usually independent in its resources and therefore the software must be written in an specific way to use both cores harmonically to get the best results when running a single code on a multi core processor, or alternatively is possible to execute many codes on a single processor, letting that each core handles one code or two [14].

How the processor executes each instruction comes down on how the core is implemented, some cores use lower frequency clocks to complete any instruction in just one cycle, others use faster clocks and complete the instruction in a variable number of clock cycles depending on the instruction difficulty and other factors and some others use what is called a pipeline architecture [15]. When a core uses a single clock cycle to compute an instruction it is called a single cycle core, or monocycle core, and as every instruction is completed as it arrives, it cannot interfere with the next one, furthermore, as the core needs to fetch data from memory from time to time, the delay associated with that operations must be taken into consideration when performing the maximum frequency calculation [5].

On the other hand, a pipelined core is a special architecture that creates multiple steps inside the core, each one in charge of a specific operation. When the instruction arrives at the core it must go from one stage to the next, and, theoretically, it will always perform the same sequency of steps, even if in some of them there is nothing to do for a specific instruction. This architecture allows for many instructions to be executed at the same time, because when the second instructions arrive, the first will be at the second stage, and one clock cycle after that both instructions will advance one stage forward into the pipelined system and a third instruction will arrive to begin its execution. Essentially is possible to execute at the same time as many instructions as stages, and after an initial delay, each cycle a new instruction will finish its execution. However, a pipelined architecture has two big problems,

the first one is when one instruction requires to read the register's address where a previous instruction will write its result, to solve this issue the core forces the instruction with the requirement to wait until the other is completed, this process takes as many cycles as stages, and during this time no new instructions are added to the pipeline architecture, and therefore is a complete waste of resources and time. To further accelerate this task is possible to use a data forwarding process, that evaluates the read address of all new instructions with the result address of the instructions that are being executed, and as soon as the instruction get the result that will be written, a new datapath enables the write to occur, effectively forwarding the data to shorten the wait time for the other instruction [15].

The second problem a pipelined architecture has is related to the branch instructions, as these depends on the result of a comparison between two values, and usually that result is not available in time before the instruction goes from the first stage to the second. To solve this problem the core could use the data forwarding unit to stall the branch decision until the comparison is completed, however there is another option for this problem and that is to try and predict what the result will be. Predicting the result is better than just waiting as it allows the core to start executing the next instructions, however in some cases the core will miss predict the result of the comparison and in such cases the advancements made on the instructions after the branch must be deleted, as the program will not execute them, but rather other instructions located at a different memory address [15].

Hence the way the core predicts the result of the comparison directly affects the throughput, lowering the number of instructions executed per cycle, as each fail means the deletion of some instructions to fix the error. The prediction process could just always assume that the core will take the leap and jump to a different memory address, but there are better ways to do this, for instance the core could remember the last outcome and try to use it in the next attempt. Likewise, is possible to remember the last two or even the last three results to decide whether to jump or not, another possible variable to keep track of is if the same branch is executed several times, as is possible to remember the last outcome and try and repeat it [16].

As a final note on this topic, is important to consider that a monocycle core does not have any of the problems associated with data dependency, or with branch prediction, as both are caused by the characteristics of the pipeline architecture, therefore if a program has a lot of branch instructions or if there are many instructions that uses the result from the previous one a monocycle core will have a better performance.

In either case, the process of reading or writing data to memory is a bottleneck [17] and to improve the performance of this operation an extra component is usually used, a cache memory. This memory is essentially a very fast memory but with very limited capacity, used to store multiple segments of the main memory. To quickly access them, saving a little bit of time in the average read or write operation. As a system that connects the core with the

memory, it could store instructions and data, and is possible for the core to write in it or to read from it, and the same is true for the main memory, to keep track of the information stored, the cache memory could use different techniques such as just copy blocks of data from memory using the address of the block to set an specific slice of the cache memory to store it, this method is easy and simple and the way it replaces the old information is when another address requires an used slice. As simple and easy as it is, this method known as direct mapping, has a big problem that comes when the processor requires information from multiple addresses successively and in case that two or more addresses are in a block that is mapped to the same slice then the data would need to be replaced as soon as possible, causing a long-term delay in the process [18].

Other possible option for a cache memory is to use a fully associative mapping, in which each slice of the memory has a tag, indicating which is the initial address of the slice, and since the number of words stored per slice is constant the final address is also known, and therefore is possible to locate any address into the cache contents. This method allows the memory blocks to be stored easily, without considering its specific address, however as the memory is not organized, the search time to know if the desired address is present into the cache is higher than the time used for the same operation in a direct mapped cache. The replacement method in this type of cache is usually by measuring the usage frequency of each slice and the one with the lower usage frequency is then replaced when new data arrives [19].

To summarize there are multiple factors that determines if a processor is fast or not, for instance if it has multiple cores then multiple programs can be executed at the same time, its ISA determines if it is going to be very efficient for specific tasks, but slower for others, or if in contrast is just good at doing general tasks. The resources used to build the core depend on what instructions are present, and more specifically the RISC-V was presented as a mature and developed standard with all necessary instructions for a complete core and an emphasis on versatility. To further speed up the processor the idea of using a pipelined system along with data forwarding and branch prediction was shown as a good option but if the code has many data dependencies and branches then a monocycle core could be faster. All these techniques could be combined with a cache memory to further optimize the core performance and achieve a higher execution speed.

After knowing some of the existing features that are used to get a better performance out of a processor, is important to also consider what offer other processor implementations. As this project aims to complete an open-source processor, using a hardware description language as the main implementation method, similar cores have already been implemented and are ready to be used. For starters one noteworthy implementation of a processor is the ORCA [20], an RV32IM processor made in VHDL and Verilog by Kammoh, it features a 4 to 5stage pipeline, a single core and can execute matrix operations, however its documentation is limited just to its interphase buses, interruptions and some registers. Then there is a

core called PicoRv32 [21] made by Yosys, this processor features a Verilog implementation of a variable ISA, allowing the user select from the RV32E, RV32I, RV32IC, RV32IM and Rv32IMC, and is optimized for size but not for raw performance, it only has a single core and some instructions like jumps can take up to 6 clock cycles to be completed. Another completed processor was made by Chen Chen and Xiaoyan Xiang [22], their processor was a 12 stage pipelined with out of order execution to get a massive performance out of a single core, the ISA they used was the RV64GCV, which allows compressed instructions, vector operations and the standard I, M, A, F and D extensions. None of the previous processors give any complete documentation about how it was designed or implemented, and only allow minor changes to its architecture, making them poor subjects for people that want to learn the specifics of a processor or that want to edit them for their own purposes like optimize certain algorithms.

There are other examples of soft processors made with a full documentation, and while their main purpose is not to execute complex algorithm and have a superior performance while doing it, is important to consider what characteristics have been introduced into these processors. For starters Abelardo V. [23] made a single-core processor, using the RV32I ISA that handles all the basic functions needed, and was made as an educational project, it also has interruption management and as all the other processors shown until now the possibility of using the GCC compiler. Ivan D. [24] also made a processor to be used as a drone controller, specifically aiming to execute PID algorithm. This processor featured a single core and no pipeline at all, but the documentation about the architecture is just enough to understand what is happening in each instruction. Finally, P. Jamieson [25] made a framework for undergraduates to get to know computer architectures by using a RV32I architecture in VHDL and has the option to implement one or two cores, but no pipeline at all. As a final note none of these processors report any usage of cache memory to speed up the memory operations.

As seen in all the previously shown information in this chapter there are many processors already implemented with the RISC-V standard, but they have many differences in the number of cores, purpose, and even in how some of them are pipelined while others are not. However, there is a gap in the possible features a core could have to get the highest performance possible, as even if there are processors with multiple cores, with the RISC-V ISA to be as versatile as possible to perform any kind of algorithm, and with pipelined architecture with a broad range of stages or in some cases there are even single cycle or multi cycle processors that do not have any kind of issues with the execution of some instructions, there is not a fully documented processor capable of extracting the best of all the previous characteristics. That is what this document proposes: a multi core Rv32IM processor that can execute instructions in a pipelined system, but that given the necessity, could switch while running to become a monocycle processor to avoid any delay associated with data dependency and branch prediction, equipped with a cache memory to accelerate the memory

Table 2-1: RISC-V implementations.

Ref	**Name**	**HDL**	**RISC-V modules**	**Architecture Documentation**	**Purpose**	**Compiler**	**No. of pipeline stages**	**No. of cores**
[20]	Orca	VHDL	RV32IM	Limited to bus and some registers	Matrix Processor	GCC	4 or 5	1
[21]	PicoRV32	Verilog	RV32IMC	Limited to configuration	General purpose	GCC	1	1
[22]	-	No info	RV64GCV	Capabilities documented	General purpose	GCC	12	1
[23]	-	VHDL	RV32I	Architecture fully documented	Education	None	1	1
[24]	-	VHDL	RV32I	Architecture fully documented	Education	GCC	1	1
[25]	-	VHDL	RV32IM	Architecture fully documented	Education	None	1	1 or 2

transactions.

Chapter 3

System Overview

The proposed system consists of a dual-core processor with variable pipeline based on the RISC-V RV32IM architecture. Figure **3-1** shows the block diagram of the processor. The first important characteristic of the architecture is that each core has its own separate cache memory, which is connected to the main memory, so that each core can execute its own program, and both programs are completely independent.

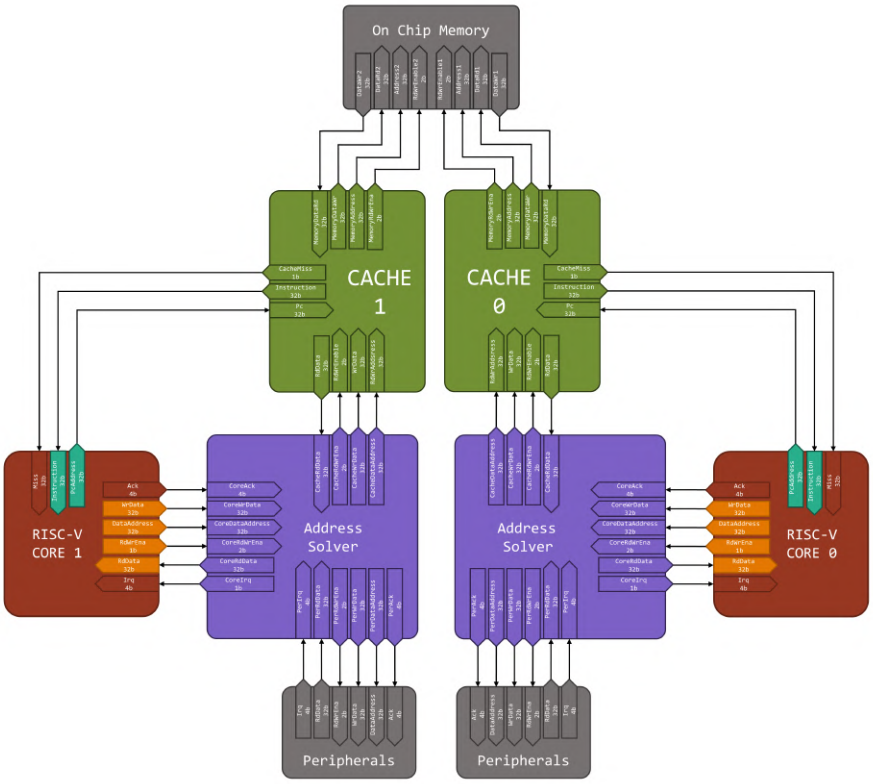


Figure 3-1: Multi core block diagram.

Any kind of memory could be used to store the program, for this specific case the selected memory was the dual-port RAM memory from the Quartus IP library, as it allows an easy to define contents and is uploaded to the core along with the rest of the implementation. As the memory has two separate read and write ports is possible to feed both cache memories at the same time, allowing a simple way to download or upload of data to and from the cache, as each core has its own communication bus to the memory. This cache configuration was selected to allow the processor to be symmetrical, which means that both cores are identical, as well as the cache memories and the connections from and to the cores. Each core also has an address solver, a block that is used to redirect information to the peripheric circuits and that solves which block is the one that the processor needs to read data from or write data to.

Figure **3-2** shows the block diagram of the proposed architecture of a single core. There are two principal areas: the one inside the grey dotted line, and the one outside the line. Outside that line, there are four blocks responsible for processes outside the pipeline structure, such as forwarding data from one stage to the next, predicting branch results, or generating the control word for the newly arrived instructions. Inside the grey line is the dynamic pipelined system, composed of five different stages, each with its own blocks and features.

Pipelining is the key feature that allows the execution of up to five instructions at a time, each at a different stage of execution, and therefore at a different stage of the pipeline system. However, it is not always possible to have all five instructions running smoothly, as sometimes one instruction needs data to be computed from the previous one, or sometimes the instruction requires changing the program flow by jumping in another direction, or sometimes even an external device requires a certain piece of code to be executed. All these possibilities cause interactions of the stages of the other core blocks that are outside the pipeline system, which are designed to provide reasonable solutions to these problems, either by trying to bring the necessary data one stage ahead in time, controlling the correct program flow when needed, or preparing the core to go to or return from a different subroutine successfully. Communication between the pipeline blocks and the blocks outside is vital for the correct execution of any program.

However, as the core uses a dynamic pipelining system, it is important to consider what makes it "dynamic". The idea is to have a core that can use the previously depicted pipelining features and architecture. However, there are some cases when the written code depends heavily on instructions that require data from the previous instruction, or that involve several jumps, and therefore do not fully utilize the primary features of the pipelined architecture. These instructions could have better performance if executed in a mono-cycle core.

Between each stage of the pipelined architecture, there are registers that allow information to be retained from one stage to the next. But the core has an option to bypass these registers and function as a mono-cycle core, rendering it a dynamic core. In this mode, the core does

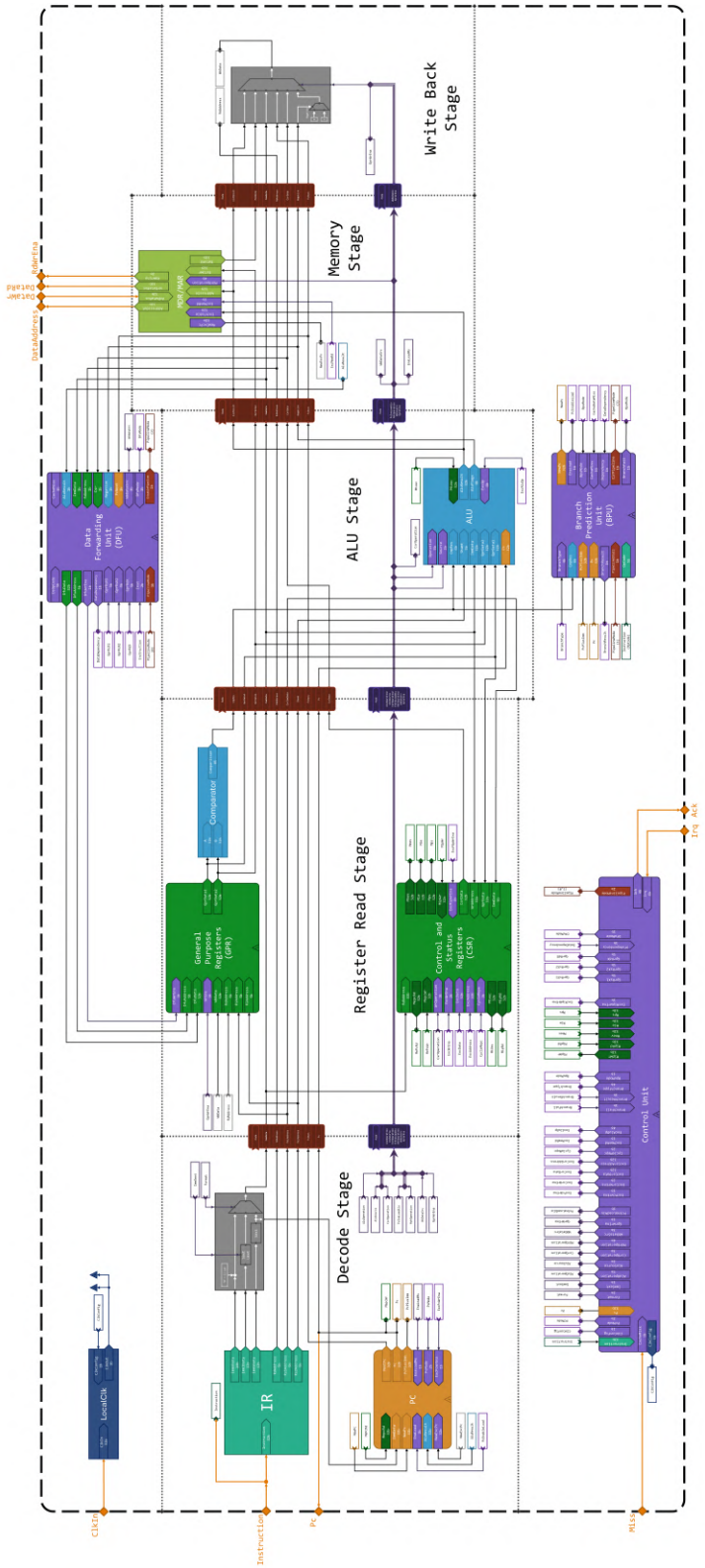


Figure 3-2: ARK-I single core block diagram.

not need to forward data, as there are no real stages, and there is no need to predict any jumps due to the fat that the result of branch instructions is delivered in the same clock cycle.

The core in mono-cycle mode has one major downside: interruption management still requires some clock cycles to sort everything out, and during those cycles, no instructions are being executed. Also, as all the necessary tasks of the pipeline must be done in a single cycle the clock frequency should be lower in the core when it is in monocycle mode than when it is in pipelined mode.

3.1 Single Core design

The design of the single-core architecture was developed in compliance with the RISC-V RV32IM standard. This implies that the core can manage fundamental instructions, encompassing arithmetic and logic operations, transferring values between registers, comparing values, altering program flow control, and loading or storing values to/from memory. The selected standard also facilitates the use of multiplication and division operations, which are commonly encountered in academic and, more notably, research algorithms.

Figure **3-2** displays the block diagram of a single core. The signals entering and exiting the core are depicted in orange, those used for control purposes are shown in purple, and the clock signals are in dark blue. Any other data and address signals are presented in black. As observed in the block diagram, the core consists of five distinct pipeline stages: the Decode Stage (DS), Register Read Stage (RRS), ALU Stage (AS), Memory Stage (MS), and Write Back Stage (WBS). Each stage is encapsulated within its own dotted grey box and features its unique blocks, each serving a specific purpose and utilizing various signals from the control unit.

However, it is essential to note that this core is not designed as a conventional pipelined system. It offers the option to transition from pipelined operation mode to single-cycle operation mode through the utilization of a custom CSR named the “Machine Pipeline Status register” and the pipeline registers block. For a comprehensive understanding of each stage, the design and functionality of its respective blocks are explained in detail below.

3.1.1 Decode Stage

This stage is where the core receives the instruction from memory, extracts the essential information from it, and organizes it into the various datasets required. The program counter is also located in this stage.

Instruction Register (IR)

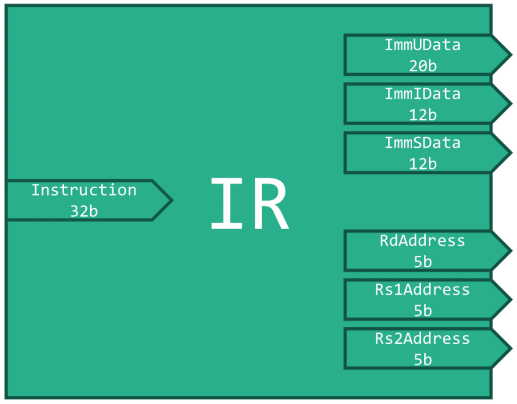


Figure 3-3: IR input / output block.

Although this block is referred to as a "register", it does not contain any synchronous elements. The name is used as a formality in computer architecture. A register is unnecessary in this context because the Program Counter (PC) will not change its value during the next clock cycle, and thus, the instruction being received for that clock cycle remains constant. Figure **3-3** shows the block's input and output signals.

This block extracts information from the instruction. However, as it is unaware of the instruction format, it obtains immediate data for all potential cases: U, I, and S formats. Additionally, this block is responsible for retrieving information about the "RS1", "RS2", and "RD" addresses. The first two represent the addresses of general-purpose registers that an instruction could read and utilize, while the last address pertains to the write address for the instruction. It is important to note that not all instructions use these fields for their intended purpose, and some even employ these fields as part of the immediate data received or as identifiers for the function operation code.

Program Counter (PC)

This block houses the register known as the program counter, and its primary function is to store the memory address where the current instruction is located. Typically, its value increases by one with each clock cycle. The PC is constantly calculating its next value, which is then transmitted to other blocks using the "NextPc" signal, Figure **3-4** shows the other PC's input and output signals. However, because most programs are non-linear, there are instances when the value stored within the PC needs to change. In such cases, it can engage in a calculation with an immediate value provided by the Immediate Selection Block to determine its next potential value. However, this value remains provisional until confirmed by the Branch Prediction Unit (BPU). This precaution is taken to prevent any issues with

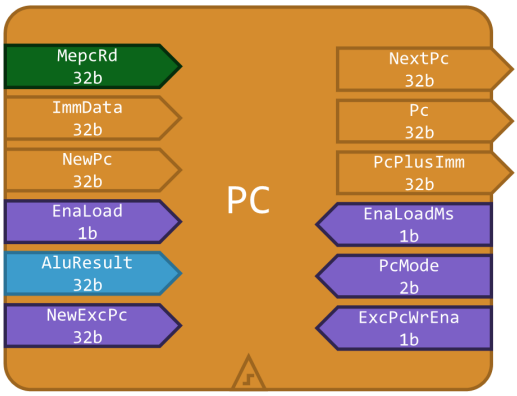


Figure 3-4: PC input / output block.

program flow due to the pipelining feature of the core. This block receives the control signals "ExcPcWrEna", which is used to write the value received from the MEPC when an interruption or exception requires it, "EnaLoad", used by the BPU to enable writing, "EnaLoadMs" to write data from the Memory Stage, and "PcMode", which controls the four different normal operation modes of this block.

Immediate Selection Block

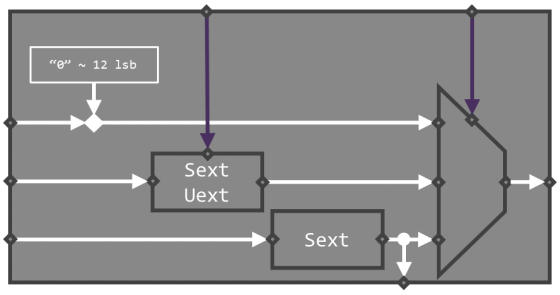


Figure 3-5: Immediate selection inner block.

This block is comprised of a multiplexer controlled by the "Format" signal. This multiplexer selects one of the three possible immediate fields used by the instructions. The first field takes the data from the U format port of the IR and appends twelve zeros to its least significant bits. This is a consistent action for all instructions in the U format, which modify their immediate data in the same way.

The I format port passes through a sign-extended or zero-extended block (Sext/Uext) that performs sign extension or zero-filling of the most significant bits to expand the data as signed or unsigned. The "Sext/Uext" block is controlled by the "ImmSext" signal from the control unit. The S format port always passes through a sext block, and its output is utilized

by both the multiplexer and the PC. The output from this block is the immediate data used in any given instruction, with its specific formatting already applied.Figure **3-5** shows the corresponding circuit for this beheavior.

3.1.2 Register Read Stage

This stage is primarily used to read information from the General-Purpose Registers (GPR) and Control and Status Registers (CSR). However, in this stage, there is also a dedicated comparator for the read GPR values.

General Purpose Registers

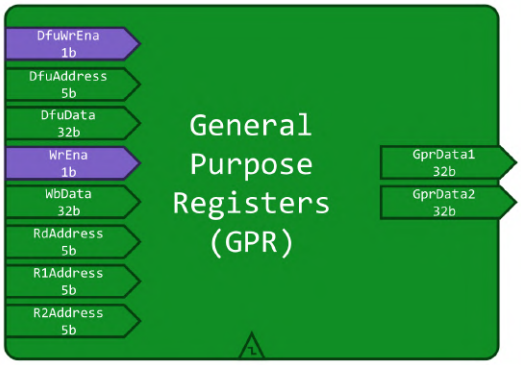


Figure 3-6: General Purpose Registers block.

Inside this block the main thirty-two registers of the processor are located; each register uses a 32-bit word. Some of these registers are special, such as the Zero register where it is impossible to write information, or the "SP" register used with other instructions to store and load data to and from memory. Even if certain registers have dedicated names, such as "Function argument", there are no restrictions on how they are used, and their names are merely suggestions for programmers who wish to utilize them. Figure **3-6** shows the GPR's input and output signals.

Reading from these registers is a simple operation. All that is needed is the address of the registers required by the instruction, either "R1Address", "R2Address", or both, and then the data stored in the respective register will appear on the outputs "GprData1" or "GprData2".

On the other hand, writing information to any register requires more steps. First, the "WriteEnable" or the "DfuWriteEnable" ports must be set to one. Additionally, a writing address and the data to be stored are required. The address can be provided via the "DfuAd-

dress" or the "RdAddress" ports, and similarly, the data can be received via the "DfuData" input or the "WriteBackData" input.

Although this block is part of the Register Read stage, the process of storing information is not within the scope of this stage. Instead, it belongs to the Write Back Stage, with an exception when the system does data forwarding, in which case the information would come from the Data forwarding unit (DFU).

As a side note, it is important to clarify that if the DFU and the Write Back Stage are attempting to write information in the same cycle with the same address, only the value received from the DFU is stored. This is done this way because normally the data from the Write Back Stage would be stored, and then two cycles after that, the one from the DFU would replace it, meaning that said data is the newer one and the one that must be kept stored.

Comparator

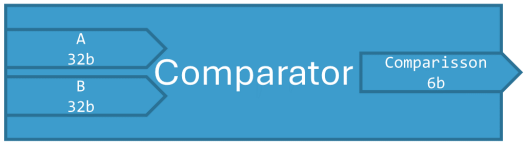


Figure 3-7: Comparator block.

This block exists to compare the data obtained from the General-Purpose Registers (GPR) stored in the RS1 and RS2 addresses. This comparison serves to facilitate certain ALU operations and provides additional time for tasks like division and multiplication. It also forwards the register comparison to this stage, rather than performing it in the ALU stage, which would result in a wasted clock cycle while waiting for the results to stabilize. This block generates a six-bit signal indicating whether the value of A is greater than B, equal to B, different to B, or less than B. In these six bits the signed and unsigned cases are differentiated. Figure **3-7** shows the comparator's input and output signals.

Control and Status Registers (CSR)

This block is also a register block; however, unlike the GPR, these registers cannot be used to store data. Each register serves a specific purpose and possesses dedicated write/read ports. Formally, the RISC-V standard includes 4096 addressable CSRs. However, many of these are designated for supervisor, hypervisor, or user modes. Since this core only implements machine mode and is designed to operate without an operating system or multi-thread control – each core runs its own program or independent threads of the same program –

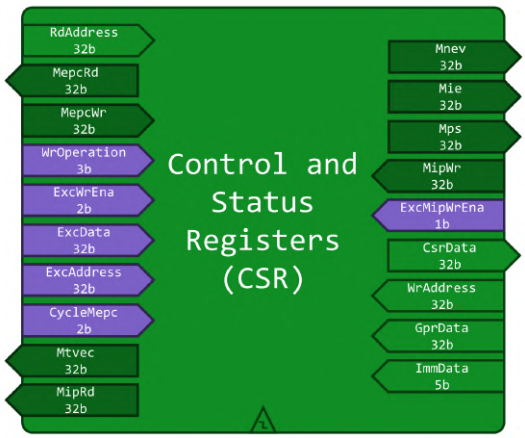


Figure 3-8: Control and Status Registers block.

only a select few CSRs have been implemented. In Appendix B, the implemented CSRs are presented along with concise descriptions and their corresponding addresses. Figure **3-8** shows the CSR's input and output signals.

In the event of attempting to access any CSR not listed, a write operation will have no effect, and a read operation will yield zeros. However, should a particular implementation require additional CSRs that were not considered during the design and initial implementation, expansion is possible.

Most of the CSRs shown in Appendix B are described in the RISC-V manuals; however, some are custom and therefore require an in-depth explanation. The first one is the MEPC register. Usually, it holds the value of the program counter prior to an interruption. However, since each core can sustain up to four levels of nested interruptions, it is important to save the previous PC value to continue its execution afterward. Therefore, this register is moved from its original position (0x0341) to an unused position, allowing it to be near the other MEPC registers. The CSR blocks have a port called "CycleMePc", which is used to move the values stored in MEPC0 to MEPC1 and the one in MEPC1 to MEPC2, and so on. This is done to have the last PC value needed in MEPC0 always, allowing the system to always read this register. When read, the "CycleMePc" port must give the signal to do the inverse operation, leaving the previous PC value in MEPC0 to keep the program running.

3.1.3 Alu Stage

This stage is home to the Arithmetic Logic Unit (ALU), which gives its name to the stage and is the most important stage of the core, as most instructions require the ALU in different ways. In this stage, you can also find the datapath used to write information to the CSR.

Arithmetic Logic Unit (ALU)

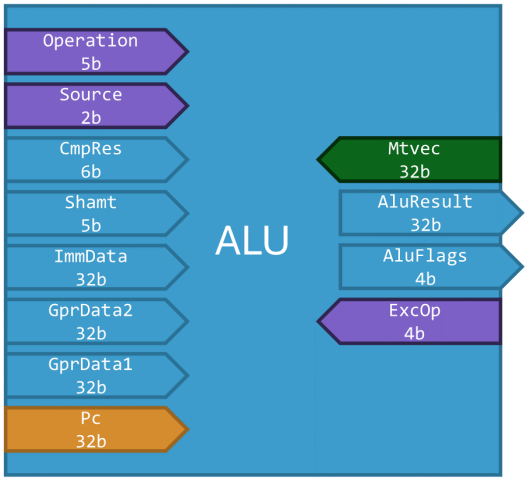


Figure 3-9: Arithmetic Logic Unit block.

The Arithmetic Logic Unit or ALU is the block where the core performs most operations. Figure **3-9** shows the ALU's input and output signals. It receives data from registers, both GPR and CSR, immediate data, the PC, and is also involved in processing interruptions. Due to its multifaceted utility, it is controlled by two signals: the operation signal, which designates the intended operation, including addition, subtraction, multiplication, division, remainder calculation in division, right and left shifts, as well as logical operations such as XOR, OR, and AND. The second control input is the source signal, which indicates the chosen input source for executing the designated operation. This block also features an independent mode employed to supersede its normal operation during exceptions. In this mode, the inputs from the operation and source signals are disregarded, and the operation required to complete the exception process is executed. This process invariably involves adding the value received from the Machine Trap Vector Base Address (MTVEC) to the data received from the exception operation port. This block has three different sub blocks, one used for addition and subtraction, the other for multiplication and the third one for division, however any of these operations must be completed in less than a clock cycle to comply with the timing requirements of the pipelined architecture. For the monocycle timing requirement, the division and multiplication are basically the ones that would decide the maximum clock frequency.

3.1.4 Memory Stage

This stage is used to store and load information from the memory; along with the decode stage this is the only other part of the core where there is a connection to the memory and peripherals.

Memory Data Register / Memory Address Register (MDR / MAR)

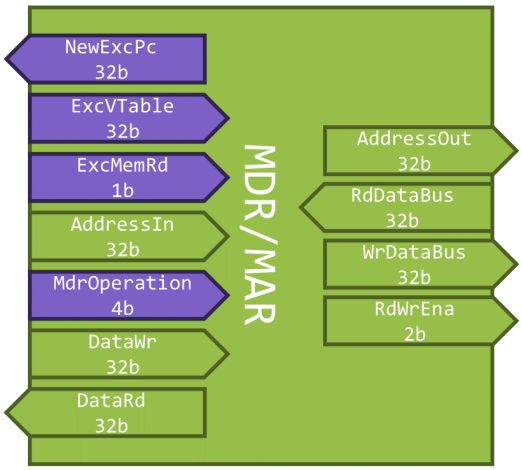


Figure 3-10: MDR / MAR block.

This block combines the functions of the memory data register and the memory address register. Despite both being referred to as registers, the current block does not contain any registers. This block can be configured to either read or write information to a specified address. Given that the bus is linked to the core's dedicated cache, there is a possibility that the requested data might not be immediately available. However, this scenario is managed directly by the control unit, and the MDR/MAR block does not take any action in this regard. Figure **3-10** shows this block's input and output signals.

Moreover, during reading or writing, the "MdrOperation" port provides information regarding the number of bits used from the given information. This signal employs its most significant bit (MSB) as an enable signal. The subsequent bit determines the state (read or write), and the last three bits indicate the number of bits utilized in the operation.

3.1.5 Write Back Stage

This is the last stage of the pipeline system, and is used, as its name implies, to write data to a previous block, the GPR. The CSRs are not written in this stage as they were written at the ALU stage previously.

Write Back Selection Block

This block collects data from various sources, including the CSR, the MDR/MAR, the ALU (both the result and negative flag), and even the Immediate data from the IR. Out of these five signal sources, this block selects one to transmit to the GPR for writing. To facilitate

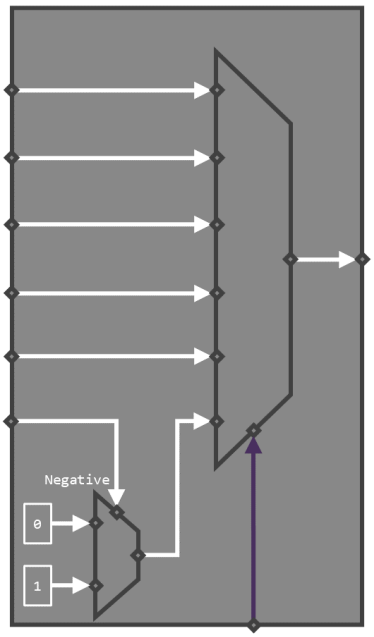


Figure 3-11: Write Back Selection Block.

this process, the control unit also transmits a signal to identify the required source for each instruction. Figure **3-11** shows this block's input and output signals.

In addition to managing signal selection, this block generates a value of zero or one based on the negative flag from the ALU. This value signifies whether one number is greater than another when subtracted. This result is utilized in certain instructions.

3.1.6 Other Blocks

There are certain blocks that are not part of any stage because they gather signals from various parts of the core to perform their specific tasks. These blocks are responsible for monitoring the instruction flow in the pipelined system and keeping track of when they need to retrieve data from the pipeline, ensuring that it is the exact data required.

Control Unit

This block is where instructions are decoded and processed into a control word, composed of various individual signals targeted towards specific blocks within the core. The control unit is designed as a hybrid between a finite-state machine (FSM) and a microprogrammed control unit. Figure **3-12** shows the control unit's input and output signals, grouped by their functionality and destination within the core. The finite-state machine oversees particular cases such as interruptions, exceptions, or issues like bubbles and stalls when

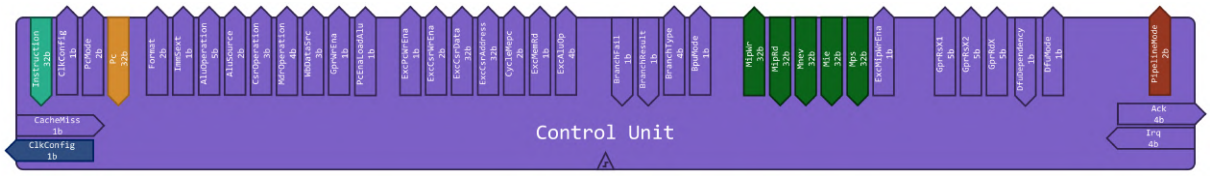


Figure 3-12: Control Unit block.

needed, whether due to cache misses, data dependencies, or branch mispredictions and the JAL and JALR instructions in pipelined mode. The microprogrammed control unit manages all other instructions in the RV32IM. To do this, it reads the operation code (OpCode) field of the instruction and other specific fields like "Funct7" and "Funct3". With this information, it fetches the desired codes for each block from its memory and consolidates them into what is called a control word.

The micro-program memory also generates two additional signals. The first one provides information on whether the received instruction was a branch and, if so, which comparison must be made and confirmed to commit to the new memory address. The other signal provides information on which register address fields are being used. This information is vital for the DFU, which needs to know if any of the current instruction's input registers are being calculated by previous instructions to attempt to forward the required data.

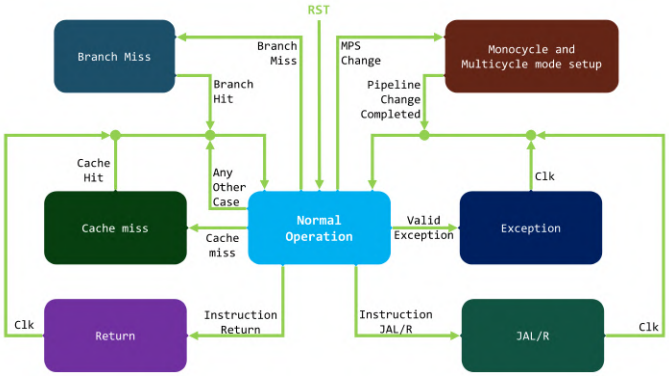


Figure 3-13: Control Unit's Finite State Machine.

The finite state machine shown in Figure **3-13** displays its general structure, where each square, except for "Normal Operation" and "Cache Miss" represent a series of many states used to complete the task enunciated in the group square. For most received instructions, the active state would be "Normal Operation". However, the instructions, JAL (Jump to an address relative to PC), JALR (Jump to an address relative to a GPR), and all types of return, are managed by specific sequences of states because they require reading and writing data from CSRs and checking if all previous instructions were completed.

The finite state machine also manages cases where there is a branch misprediction or a

cache miss, and it executes the necessary stalls according to what is needed. Along with its functions, it monitors the transition from a monocycle core to a pipelined core while it is running. Each of these cases is described in detail in Appendix C along with its specific figure describing each set of states and its transitions.

Exceptions and interruptions

An exception or interruption is an event that disrupts the standard program flow due to an unexpected situation. Each core of the current processor can oversee up to four exceptions or interruptions. An exception is caused by the program itself, and there could be multiple reasons for an exception. However, the only one that is implemented pertains to an unknown instruction, which is triggered when the microprogram fails to recognize the received instruction. An interruption occurs when something outside the processor sets one of the IRQ signals to a high value, indicating that its routine needs to be executed.

Both interruptions and exceptions have a memory address where the instruction for jumping to their respective program routines is stored. These addresses are consecutively located in memory from a base address stored in the MTVEC CSR, starting with the error subroutine address, then the zeroth peripheral then the first peripheral then the second and then the third. Inside the finite state machine each one of these addresses is also associated to the cause of interruption, which is also used to calculate the priority of each kind of exception. Is important to remember that the Machine Interrupt Enable (MIE) register enables or disables interruptions by having a one or a zero in its eleventh bit respectively.

The process used to handle an interruption is the same as the one used for an exception, and in most cases the RISC-V standard and this document uses both words as synonyms. Both are controlled by the control's finite-state machine, as seen in Figure **3-13**, where there is a segment dedicated to handling exceptions and returning from them. In that segment the core evaluates if any other interruption with a higher level of priority is being executed, disables the PC, and allows all the current instructions to be fully executed, and after that it reads the memory address defined by MTVAL register offsetting it accordingly to the received exception, loads the value in that memory address to the program counter and issues the ACK signal if needed. During this process the MePc0 register stores the next PC value that will be executed after the interruption.

To return from an exception the program must execute the return instruction, as this core will not differentiate between U, M and S mode all three instructions are practically the same. To enter the return states the core will verify that an exception is being executed and then it will stop the PC and let all other instructions to be executed, then it will restore the PC using the data stored in MePc0.

To enable nested interruptions the least significant bit of the MNEV register must be set

to one, and up to four nesting levels are allowed. That same register stores the data of the previous level of priority as well as how many levels of nesting are being executed. The finite state machine also contemplates those scenarios for nesting exceptions and does all the necessary steps within the MNEV and manage the multiple Machine Error Program Counter registers accordingly.

Data Forwarding Unit (DFU)

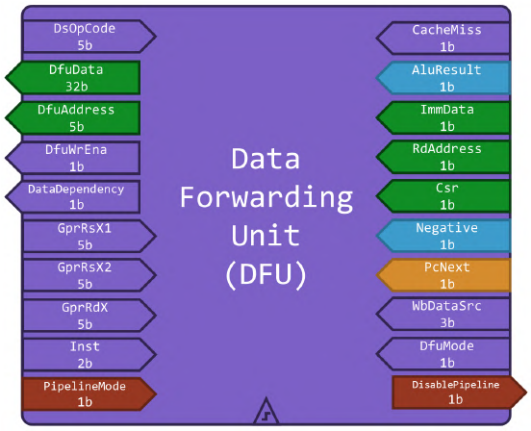


Figure 3-14: Data Forwarding Unit block.

This block oversees the namespaces utilized by other blocks, essentially retaining the read and write register addresses. If a new instruction exhibits data dependency on a previous one, this block detects such a dependency and checks whether it's feasible to forward the required data from the Memory Stage. However, when data exists in any other core's stages like the ALU or Register Read, its only option is to stall the core until the instruction that causes the dependency reaches the Memory Stage. The stall is lifted upon the completion of data forwarding or when the data reaches the Write Back Stage. Figure **3-14** shows this block's input and output signals.

It's important to comprehend why data forwarding can only occur from the Memory Stage and not from other stages. This design choice was made because most instructions employ the ALU for operations, and there are scarcely any instructions with values ready before the conclusion of the ALU Stage. Since data forwarding is a pipeline-specific feature, the required information is stored in the registers between stages. Hence, at the beginning of the Memory Stage, the desired information is ready for forwarding. As for instructions using the memory block, their data stabilizes solely after the Memory Stage, positioning them in the stage where information is written to the GPR.

Branch Prediction Unit (BPU)

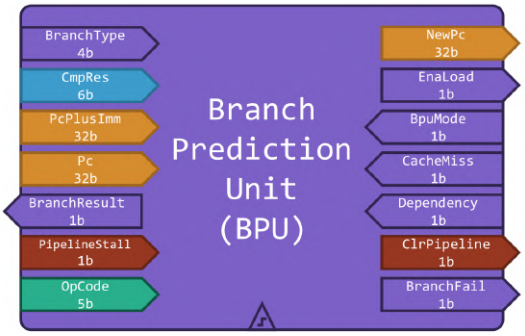


Figure 3-15: Branch Prediction Unit block.

This block is employed when the core is operating in pipelined mode. Its function, as the name suggests, is to predict a PC value when a branch instruction is encountered and to do so it gets information from the ports shown in Figure **3-15**. These instructions are used to transition from one memory address to another and they rely on the outcome of specific operations. While the operation is being carried out, it's feasible to attempt forwarding the execution of certain instructions. But, since the result of the operation is unknown, the core remains uncertain about whether to execute the next instruction or the one located at the branched memory address. To resolve this problem, the BPU assumes the operation's result based on prior outcomes. This logic is illustrated in Figure **3-16**, which portrays the finite state machine dedicated to prediction.

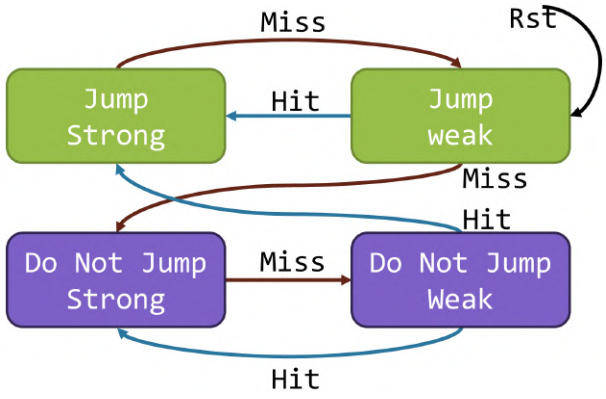


Figure 3-16: Branch prediction Unit's finite state machine.

In the finite state machine, a miss implies that the BPU predicted something, but the result was different from what was predicted, and a hit implies that the prediction was correct. Each action, Jump and Do not jump, has two different states associated: weak and strong. The idea is that if the core encounters the same result many times consecutively, it continues predicting the same result until it's proven wrong twice in a row. The first miss will bring it

to the same result but in the weak state, while the second miss will take it to the opposite result in the strong state. The logic behind transitioning to the strong state in the opposite direction is to prevent the system from oscillating between opposite results repeatedly.

Local Clk

This block was designed to allow a future expansion of this core, allowing the control unit to change the clock frequency from a lower value, associated to the monocycle configuration, to a higher value, associated to the pipelined configuration. However, in its current state it just reroutes the "ClkIn" port to the "ClkOut" despite the values received in the "ClkConfig" port.

In theory the desired clock frequencies should be selected in such a way that one is a multiple of the other to minimize possible problems when making the change.

Pipeline Registers

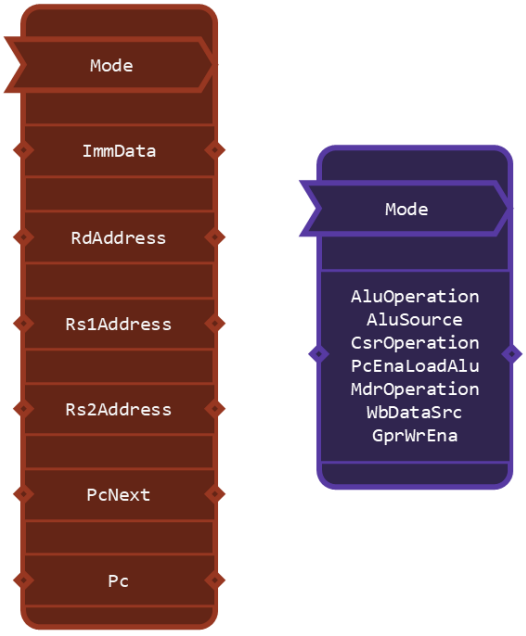


Figure 3-17: Pipeline data and control word registers.

These blocks are located between stages and are used to store the values of signals generated by the preceding stage. They come in two distinct colors: the red ones are used to store data or addresses, while the purple ones are reserved for storing the control signals needed in the next stage Figure **3-17** shows an example of both blocks' input and output signals. All the registers found in these blocks work on the rising edge of the clock signal. Each of them has

a port called "Mode" that receives the pipeline mode, that signal carries the information of whether the core operates with the pipeline stages, and the registers must be active, or in case that the core is running in monocycle mode the registers are disabled and all signals just go directly to the output. The pipeline mode signal also has information from the BPU and DFU and allows those registers to clear its contents or stop any data from being stored, according to those blocks information.

3.2 Cache Memory

As previously stated, the ARK-I processor consist in two identical cores, described in section 3.1. Each core has an independent data bus and an instruction bus that connects to the cache memory, making the cores and cache a Harvard architecture.

Each core has its own fully associative cache memory, and both caches function identically. However, the idea is that each one accesses different memory addresses as each core executes a different code. If both cores require the same address, the caches cannot determine if the fetched information from memory is the most recent, as they assume that only themselves have the most recent version of the data.

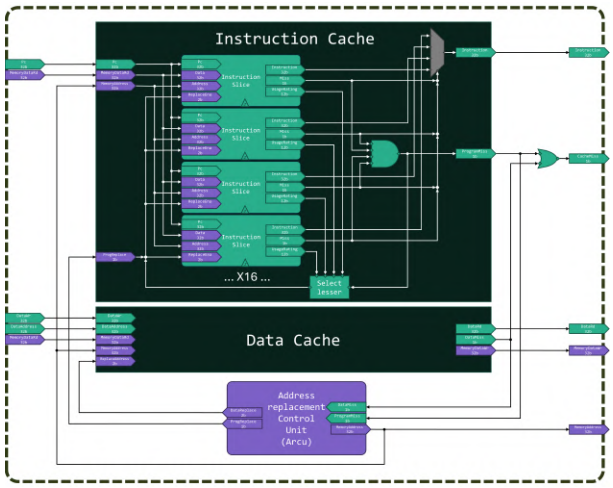


Figure 3-18: Cache block diagrams.

The cache memory has two different segments, one dedicated to store instructions, known as instruction cache, and other to store data, known as data cache. Both segments, shown in **3-18** are identical in most characteristics, as both have sixteen slices inside them and the means to decode which one of those segments is the one that the core wants to store data in, as well as the means to select which slice should be replaced next. Each slice consists of sixteen words of 32 bits, an address register, and an eight-bit counter. The address register stores the address associated with that slice and said address will always be the starting

address of the block, and therefore must be a multiple of sixteen. This is a 32-bit register, and it also features a validation bit, that shows whether the address and data stored in the slice are valid or not. Each time the core is starting all addresses in the cache are invalid, until something from memory is written to them. The counter is configured to go up by twelve each clock cycle that any data from that specific slice is used, and each clock cycle that no data from the slice is used its value will go down by one. This counter is used to measure how often the data from the slice is being used and its value is the principal deciding factor when it comes to replacing any slice.

To replace any slice, the replace signal must be active, which freezes all counters in all the slices, and then the "select lesser block" will evaluate the value in each counter for each slice, that value is called its rating and then it will select the minimum rating and will point to that slice as the one that will be replaced. To avoid more than one slice having the same rating each slice has a unique id that is concatenated as the least significant bits of the rating, making it impossible to have two slices with the same rating.

The data cache and instruction cache are almost identical, as previously stated, however their only difference is that the data cache is designed to be read and written, while the instruction cache can only be read. The reason behind the two different cache segments is that no matter what happens on the core under no circumstances the slices where the instructions are stored will be replaced, resulting in instructions that are being used to be replaced and lost, which in turn could cause a bottleneck at the cache memory.

Finally, the cache has a block called the Address Replacement Control Unit (ARCU) that checks if there is a miss in any of the two caches and using that information it issues the order to begin the replacement of one of the slices in the necessary cache segment. If both segments set the miss signal to one at the same time, then the ARCU will give priority to replace the instructions and then the data. The data replacement process contemplates that in the event of replacing a slice without an invalid address said data must be written to memory before writing the new information.

3.3 Address solver

As the data bus between the core and the cache has a 32-bit address component, this same bus was decided to be used to reference the needed peripheral modules, its input and output signals are shown in **3-19**. This block was made to prevent them interacting with the cache and fill its contents with non-essential information, and to simplify the design. The address solver is a block that controls the communication between the core, cache, and peripherals and to do so, it evaluates the address issued by the core and if its value is over 4,294,901,760 (0xFFFF0000) then address solver redirects the write information to the peripherals and the read information to the core. For values under the previously stated limit this block just does

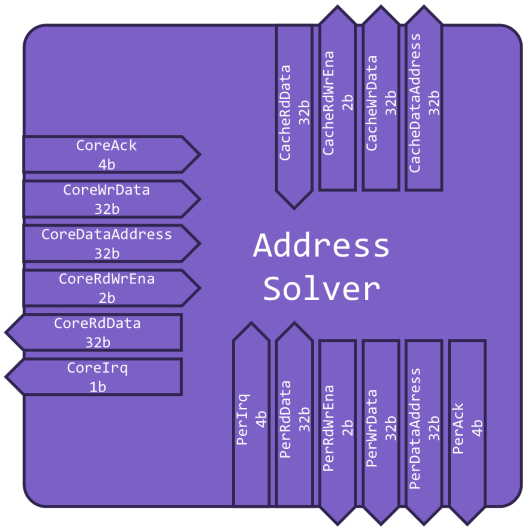


Figure 3-19: Address Solver block.

the same but communicates the cache and core. This block does not communicate the cache and peripherals as its intention is not to do direct memory access but rather, as previously stated, to ease the connection between the core and cache and core and peripherals.

3.3.1 Peripheric Modules

Each core can have as many peripheric modules as needed; however, they are limited by two factors: first there are only four Interruption Request (IRQ) and Acknowledge (ACK) signals per core, meaning that in some cases many peripherals will need to share those signals. On the other hand, as the peripherals are mapped to memory addresses over 0xFFFF0000 there are effectively 65,535 possible addresses, which is probably more than enough for the required peripherals.

The IRQ and ACK signals are present to ease the communication between the core and peripherals, the first one is used to let the core know that a peripheral needs certain subroutine to be executed, and that signal could be set to one in just one clock cycle and the core will receive it and execute it. The ACK signal is used when the core starts doing the corresponding subroutine and is sent for one clock cycle. Finally, as the core can read or write information from and to the peripherals is vital that each assigned address has a register to store the information that will be read or written.

Chapter 4

ARK-I Implementation

The ARK-I Processor implementation was executed on Quartus prime 17.1 and its intended hardware for a future implementation is an Altera's Cyclone V FPGA. From the many possible boards containing that chip, the one that was specifically selected was the one found on the DE1-SoC board. The first step of the implementation was to choose a hardware description language and VHDL was chosen due to the author's familiarity with the language. To validate the processor, each module designed in the previous chapter underwent simulation using Model Sim. That is a program made by Altera which facilitates the use of specific statements to create test vectors and prove that, at least logically, the system operates as expected.

Given the numerous stages, each comprising multiple internal blocks, and the need to test various scenarios, specific block-oriented test protocols were developed. These protocols involved reading input values assigned to each signal from a CSV document, in which the desired output is also written for each set of outputs. Then Model Sim simulates the circuit and stores the result values to another CSV. These values were then compared to precalculated desired outputs to determine if the entity functioned correctly with the provided inputs. To further streamline this process, a C code was created. This code uses the entity information as input and with that creates the necessary VHDL test protocol.

This entire process simplifies the procedure of testing multiple inputs, as it merely involves creating a set of test vectors of the required length, calculating the expected results from them, running the C code to generate the VHDL test protocol file, and then executing the simulation. The simulation results are written to a CSV file, making it easy to detect any potential errors.

The C code used to generate each test protocol as well as the CSV files containing the test vectors, are both available in the GitHub repository. The first one in a directory named Auto Test Generator, and the other one in the CSV directory within the processor's Quartus project files.

4.1 Single Core implementation and simulation

The processor implementation begins with the RISC-V core. As each core is composed of several stages and within each stage are many blocks. The goal was to implement each one of them as they are used in the instructions. This means starting from the Decode Stage and ending in the Write Back Stage. As each block was being implemented a dedicated test bench was performed to verify its functionality and solve any possible problem right away.

4.1.1 Decode Stage

This stage is composed by the Instruction Register, Program Counter and Immediate Selection Block. Other than those blocks there is a pipeline register to connect and spread the necessary data to the Register Read Stage. Those pipeline registers are made in such way that its content could be erased or if needed, disabled to keep the current data in them.

Instruction Register (IR)

As this block is not a register per se, its implementation was rather simple, just by creating the output signals with the desired lengths and setting the necessary bit positions to those outputs. This is one of the few blocks that were not tested as its function is to separate certain bits from the received instruction, and therefore there is no logic to test.

Program Counter (PC)

The program counter was implemented as a 32-bit register. Its standard operation mode makes it increase its value by one every clock cycle, however there are many other possible cases in which the next value is not as easily calculated. To handle all possible cases, the block has three multiplexers that select the value that will be assigned to the PC register the next cycle. Here, one of the most important remarks is that the PC increases its value by one due to the cache memory and main memory both having 32-bit words inside them. Contrary to other implementations where these memories have 8-bit words and the PC increases its value by four.

Figure **4-1** shows the result of the PC testbench, it began by testing the PC mode 0, that disables the program counter and as seen in the 260 ns time the PC signal did not change. Afterwards the Pc mode 1 was tested, it corresponds to the normal operation mode in which the PC must increment its value by one, that change is seen in 300 ns. At that same moment the mode 2 is tested, it writes the data from the input “AluResult” to the PC; in this case that input signal has its test value set on “0x1C6B14”. That same value is shown into the

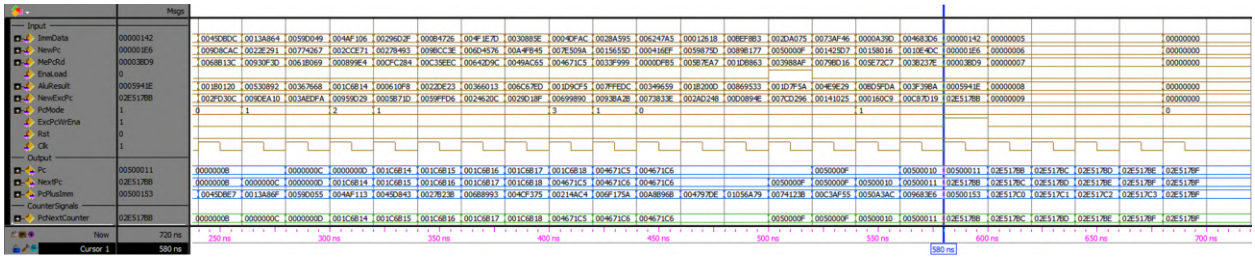


Figure 4-1: Program Counter testbench

corresponding output the next clock cycle, at 320 ns. The last mode is tested at 400 ns, and it is used to write the value from the input "MePc" to the PC, setting the output value to "0x4671C5". Similar situations are tested for the ports "EnaLoad" and "ExcPcWrEna", used to replace the actual value of the program counter with another provided from the inputs "NewPc" and "NewExcPc" respectively.

Immediate Selection Block

The immediate select block is implemented using a multiplexer that selects one of three signals, the first one are the twentieth most significant bits of the instruction with its twelve least significant bits set to zero. The second are the twelve most significant bits of the instruction. To fill the missing twenty most significant bits the system selects by using the "ImmSext" signal, whether to replicate the most significant bit to create a signed extended version of the data read, or to fill those twenty bits with zeros to create an unsigned version of the data red. The third possible signal takes bits thirty-one to twenty-five and bits eleven to seven, and joins them as a twelve-bit piece of data. Then, it sets them as bits twenty to zero of the 32-bit word. The other bits are filled by replicating the most significant bit of the instruction, to end up with the desired number in sign extended format. This last signal is also always sent to the program counter. With all three signals ready, the "Format" signal is used to select one of them as the "ImmData" output.

4.1.2 Register Read Stage

This stage houses the General-Purpose Registers, a comparator, and the Control and status registers. The pipeline register in this stage can be cleared or disable depending on the core's current state and operation.

General Purpose Registers

The General-Purpose Registers are a collection of thirty-one registers. Each one of these are 32-bits long and can be read and written as necessary. In this block the register with address zero is a hardwired zero value and therefore cannot be written. Other than the registers the block features a series of multiplexers used to write data to the desired registers. A multiplexer is used to index the written data from the Write Back Stage to the desired register according to the address received from said stage and the other written data collected from the DFU, using the address from that block to know where to write it.

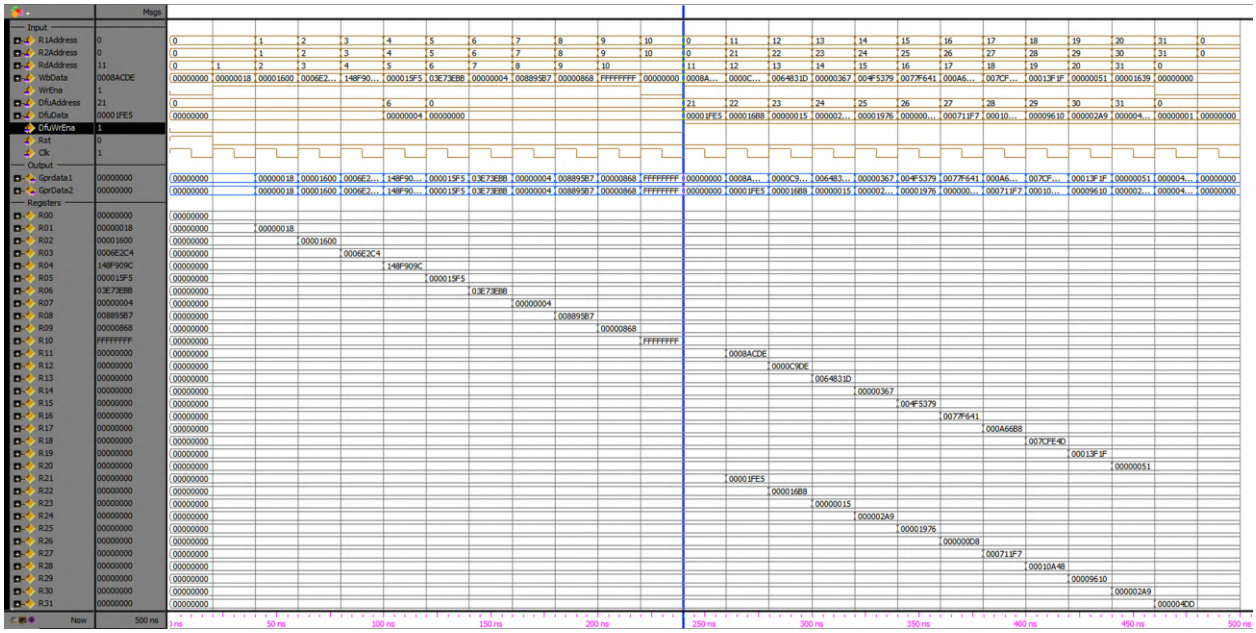


Figure 4-2: General Purpose Registers testbench

Figure **4-2** shows the results for the General-Purpose Registers testbench. In it from 20 ns to 220 ns, the block receives information to write every register from address zero to ten with different data. As seen in the grey signals, all registers store the value given, except for R00, which as previously stated cannot be written. In 240 ns the "write" signal is set to one, but this time the block also receives the "WriteEnable" signal from the DFU, testing the case in which there are two write operations from both sources, with different addresses. That test was successful as the remaining registers were written as intended.

Comparator

The comparator block generates a six-bit signal that gives the necessary information about both input signals. It was implemented by casting both signals as signed or unsigned and using the built-in comparators to set specific bits in one or zero.

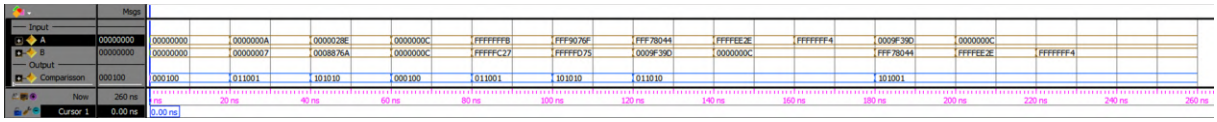


Figure 4-3: Comparator testbench

The testbench begins with both signals having the same value, then it sets the value of A to a bigger value than B. This results in a change of the output, which now sets the necessary bits to express that the inputs are now different, and A is indeed larger than B, regardless of whether both numbers are signed or not. Then the other case is tested at 40 ns, where A is set to a lower value than B and therefore the result shows that information by setting the necessary bits. All tests from that point onward test once again the previous results, but with different inputs. As is the case from 120 ns onwards, testing numbers that will yield different results on the signed and unsigned comparisons.

Control and Status Registers (CSR)

The CSR is implemented as a collection of ten 32-bits registers, four hardwired constants and a series of multiplexers. The implemented registers are the ones with total or partial writing permissions, so to write any of them, the only way is by specifying an operation, address, and the desired data to operate. The main consideration here is that all operations use “GprData” and one of the CSR as their arguments. This block uses the multiplexers to enable the write operation on each CSR independently and in that process, it also grants the writing permission to certain bits depending on the selected CSR.

Some CSRs have special features, for instance there are four different MEPC registers, numbered from zero to three. As those registers store the next PC value to use it after an interruption, it is necessary to switch that value from the register MEPC0 to MEPC1 and so on, when a nested interruption occurs. This mechanism works similarly to a stack memory, allowing the data to be popped (read) or pushed (written) while being controlled by the “CycleMePc” signal. Since those registers have that special feature, only the “MePc0” can be written normally.

The CSRs are relatively like the GPR as they both are registers and have the same writing circuits, with some differences. Consequently the testbench ends up testing one of the main features of this block, the possibility to cycle the values up and down inside the MePc registers. It started with the first two MEPC registers filled, and then two more values were pushed into them in 480 ns and 500 ns respectively. Afterwards, all MePc registers were left as they were for a while and in 560 ns the “CycleMePc” signal indicated that the values were to be extracted. Therefore the first value stored in MEPC0 were read and lost, moving the other values up the MEPC registers, the same happened in 580 ns, 600 and 620 ns.

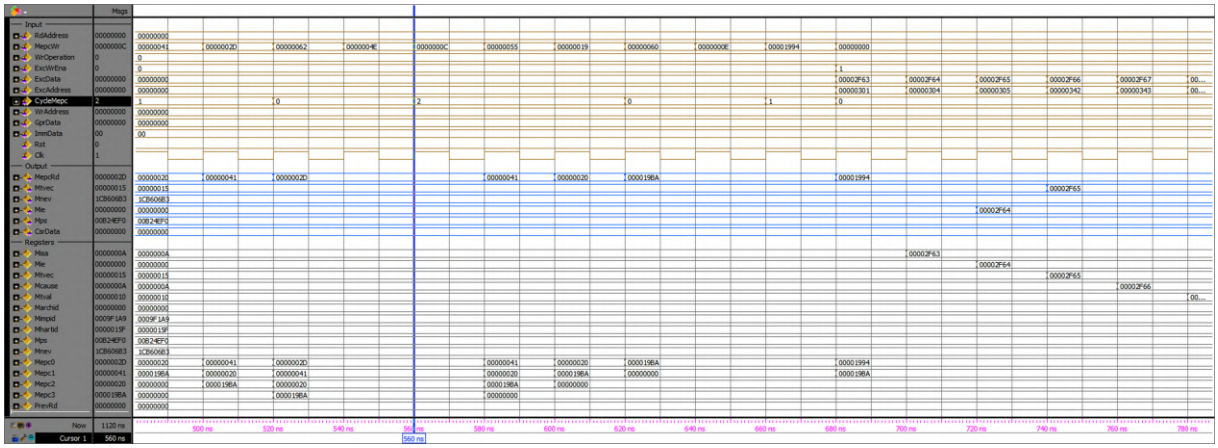


Figure 4-4: CSR testbench

4.1.3 Alu Stage

The implementation of this stage was made by summoning two different blocks, the ALU itself and a custom pipeline register used to enable or disable the pipelining mode in this stage. This register cannot be erased by any signal, but it can be disabled if needed. Apart from those blocks, some signals were set to go to the previous stage to do the necessary CSR operations, essentially making a write back data path to that block.

Arithmetic Logic Unit (ALU)

The arithmetic logic unit consists of multiple elements, each responsible for a specific operation. However, it was not implemented using a dedicated block for every functionality, instead, only the multiplier, adder/subtractor, and shifters were created as standalone blocks. Logical operations such as AND, OR, and XOR were implemented using a FOR GENERATE statement due to their highly regular interconnection. Conversely, the signals required for the divisor were defined but hardwired to zero since the divisor itself was not implemented.

Taking a closer look at the Adder/Subtractor, it was implemented as a carry-lookahead adder. Nevertheless, a secondary architecture was also defined, describing a ripple-carry adder. This was done to provide some initial flexibility in the processor implementation, allowing an easy way to switch between the two architectures if needed.

The multiplier is an array multiplier, beginning by performing an AND operation for each bit of each input and then summing the results up in four layers of adders. Each layer employs different adder sizes to minimize the number of serial operations by doing more parallel sums. As not all bits were added in each layer, and some carry signals are generated in each one, it is important to include them to the adding process, when possible to, once again, keep the number of serial adders as low as possible with this architecture.

The final block described is the shifters, which offer three different operations: logical right shift (adding zeros to the least significant bits), logical left shift (adding zeros to the most significant bits), and arithmetic right shift (preserving the original MSB from the input to maintain the sign). Each operation was implemented using a multiplexer, with the "Shamt" input serving as the selection signal to set the desired outputs. Subsequently, another multiplexer selects one from the three possible operations based on the value of the "ArithRlN" signal to determine the result.

Given that there are three different modules within the ALU, a test protocol was defined for each module, looking at different possible inputs and their respective outputs. As expected, each module successfully carried out their function, even when receiving its maximum or minimum values. Knowing that the inner modules are working, now the ALU block was tested. To do so, each possible operation defined by the signal "Operation" was selected, and within each case, the possible Sources were also defined. The values assigned to the data signals in this simulation were randomized, since the desired cases were already tested in the previous simulations.

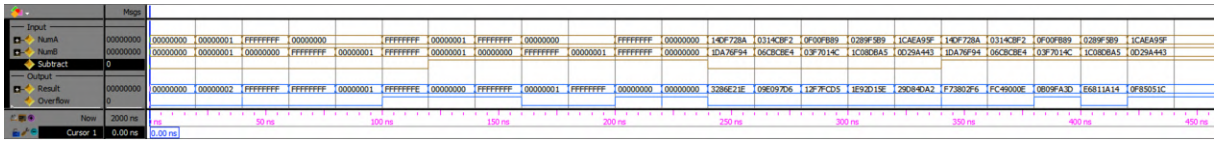


Figure 4-5: Adder / Subtractor Testbench

The adder / subtractor block was tested in its two operating modes. Firstly by making several additions of various numbers where the results were successful as seen in the time frame from the beginning of the test to the 140 ns time stamp and in the output signals "Result" and "Overflow". Right after that instant the "Subtract" input is set to one, indicating a subtraction operation meaning that the second input, called "B" is a negative number. In this case as the output signal can be negative or positive, it is important to consider that the shown "result" must be read like a signed signal, in two's complement. For that test, the output results were all given according to what was expected and therefore proved the adder and subtractor correct behavior.

The multiplicator test was performed by entering some special values at first and then random values to test functionality. Since the input is composed of two 32-bits signals, the output should be a 64-bit signal. However that output was divided into the lower part of the mentioned 64-bit signal and the higher part of it. An overflow port was also added, but as seen in the 20 ns result, no multiplication will ever use that port, since there is not any result that big. From 40 ns to 80 ns the multiplication by zero was tested and after that, from 80 ns to 100 ns a commutative set of numbers were selected, and their results were identical. Finally, random numbers were selected to test that both results ports were correct in all cases and as expected the results were satisfactory.

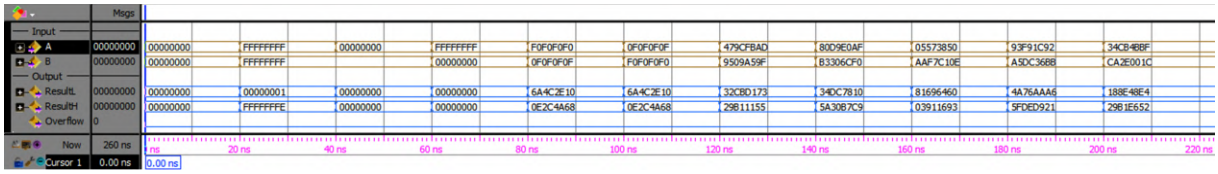


Figure 4-6: Multiplicator testbench

The shifters were tested for their three operations: Shift Left Logically (SLL), Shift Right Logically (SRL) and Shift Left Arithmetically (SRA). In Figure 26 one of those cases is shown, in which the system receives the number 1 as input and then the "Shamt" input is cycled to a set of possible values, and for all of them the results were successful. The other two testbenches are available on Appendix D.

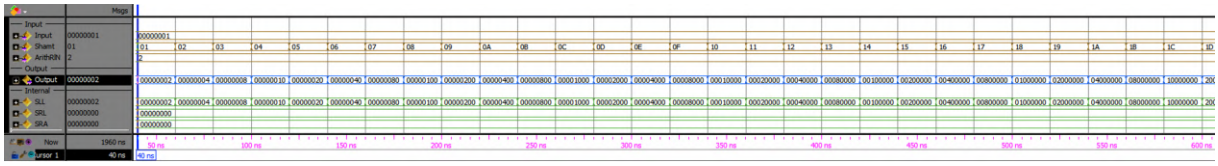


Figure 4-7: Shifter testbench (SLL).

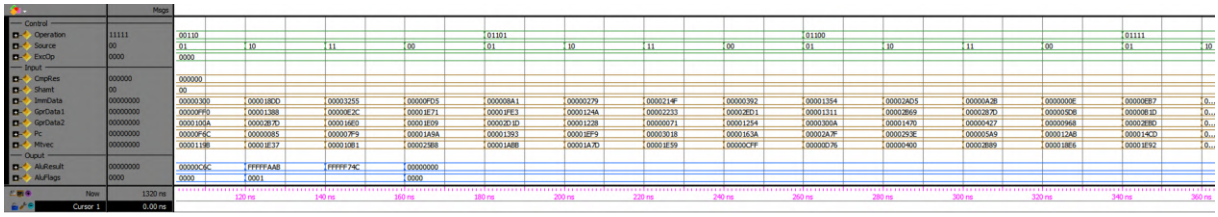


Figure 4-8: ALU testbench.

Finally, a general test of the whole block was performed. In this test all possible operations and source combinations were tested, to ensure that under no conditions this block would fail any operation. In the segment shown the operations are unsigned subtraction, signed and unsigned division, and signed remainder.

4.1.4 Memory Stage

This stage is composed of one main block, the MDR/MAR. Even so in this stage a pipeline register is also found. It can be disabled to follow the correct execution when needed, but it cannot be erased. Some of the signals in this block are rerouted to the DFU, as those signals can be forwarded in some cases to improve time performance.

Memory Data Register / Memory Address Register (MDR / MAR)

The memory data register and memory address register are composed of several multiplexers, with different functions. For starters, this block generates the "RdWrEnable" signal based on the operation it receives. That same input is also used to decide which input source it should read or write.

Since some instructions read or write eight. Sometimes sixteen or thirty-two bits are necessary to fill the most significant bits of them to complete the required 32-bit word. To do so the system uses the received operation to determine if that gap is filled with zeros, and therefore assumes that the data is unsigned, or if the Most significant bit received is used to fill that space, representing the data as signed.

As this block operation goes hand to hand with the cache memory its test was made alongside the main test on chapter 5.

4.1.5 Write Back Stage

This is the last stage of the pipelined system and as such it only has one block, and no pipeline registers.

Write Back Selection Block

This block is implemented as a multiplexer, that receives all the possible inputs that could write information to the GPR, and to control said multiplexer there is a signal that indicates which one will be written.

The only special mention in this block is the negative value, which is transformed to a 32-bit zero value, if its value is zero, or to a 32-bits value of one otherwise.

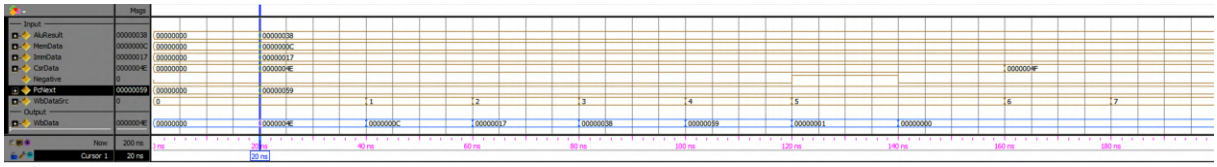


Figure 4-9: Write Back Selection Block testbench.

This block was tested by setting some constant values and selecting all possible values of sources. In the case of the negative source, both cases were tested to verify that the zero and 1 results were working flawlessly.

4.1.6 Other Blocks

Control Unit

The control unit is composed of two main blocks: the micro program memory, and a finite state machine. The first is a series of three multiplexers that uses different longitudes of the instruction opcode to establish what instruction was received and return the control word, a group of signals encapsulated in records that correspond to each stage. The values set into the control word are responsible for the behavior of the different blocks in the pipeline stage. Since there are three different signals generated from the muxes, a final mux is used to decide where the desired control word comes from. This block is also used to set some important values to the "BranchType" signal and "GprAddress".

The second block is the finite state machine, which is composed of thirty-one unique states, encapsulated in the different functionalities shown in the previous chapter. To implement this machine, a register was made to store a value of an enumerated type, that way each state will have its own name, and it is easier to keep track of them. Then, using a CASE statement each state was defined with its possible transitions and designated outputs. In this block several masks are also found used to easily read and write data to the desired CSRs.

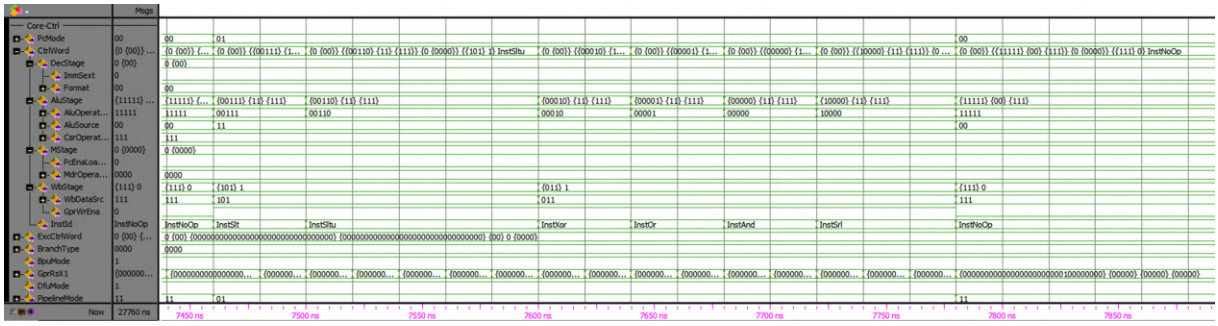


Figure 4-10: Control unit and Micro Program Memory testbench.

The micro program memory was tested by inputting several real instructions, along with the control unit, and the results shown in Figure **4-10**. It depicts the record structure inside the control word, with one field for each pipeline stage and an instruction ID to keep track of the current instruction.

The finite state machine usually oscillates between the cache miss state and the normal operation state. However in the test shown in Figure **4-11** multiple states are being used, since the program was executing several branch operations and jump operations that were overlapping. The behavior of the state machine is, in this case, correct.

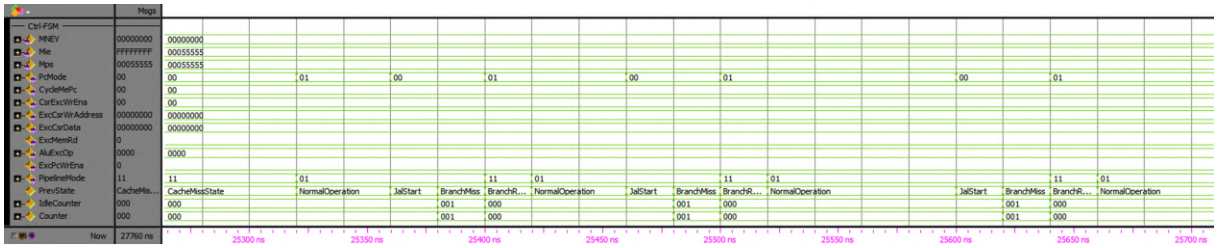


Figure 4-11: Control Unit's finite state machine testbench.

Data Forwarding Unit (DFU)

This block was implemented as a series of registers keeping track of all the current instructions that are being executed. Each time a new instruction is received, the DFU stores both read addresses, and the write address. That information is provided by the control unit using the "GprAddress" signal. Then, in the next cycle, the data is transferred to the next register to open space for the new information. In total there are always four sets of addresses stored, corresponding to the instructions executed in the Register Read Stage, ALU Stage and Memory Stage. This block is disabled completely when the core changes from the pipelined mode to the monocycle mode.

Since this block must signal the pipeline registers, BPU and the control unit if there is a dependency, it is necessary to detect it as soon as possible. To do so, it is always comparing the data in the read and write addresses. If any read address has the same value as a write address corresponding to a further stage, the dependency signal is set to one.

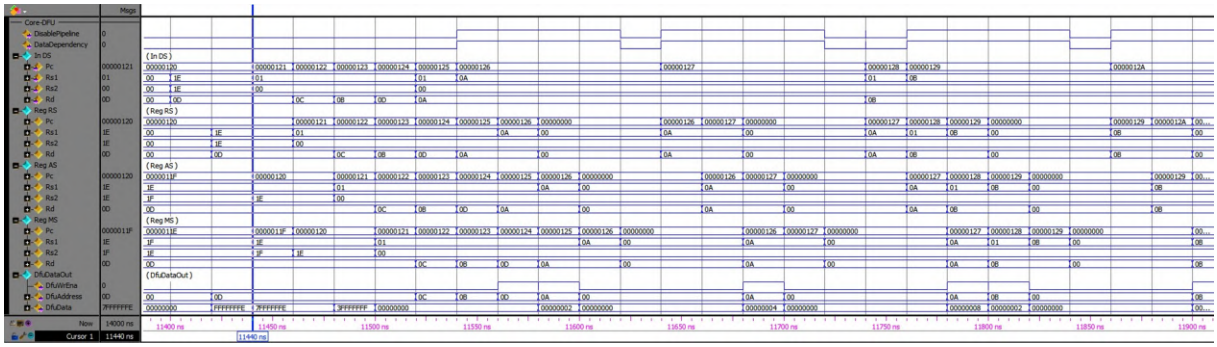


Figure 4-12: DFU testbench.

The data dependency test describes a series of chained dependent instructions. Meaning that the second executed instruction depended on the first one, the third depended on the second and so on. Chains from two dependencies and up to five were made to test the limits of this unit. In Figure **4-12** is shown how at 11520 ns a new package of addresses is received in "In DS". Then 20 ns later is transmitted to "Reg RS" to allow new information to get in. Nevertheless the previous data has the write address set to 0x0A, and the new data also has

the read address set to the same address. For that reason, the output "DataDependency" is set to one. This behavior is seen multiple times from that moment onwards. Before that moment it is possible to see a normal set of address without dependency go through easily.

Branch Prediction Unit (BPU)

The branch prediction unit is composed of a finite state machine, six registers and some multiplexers. When a branch is detected the control unit sends the "BranchType" signal to this block, where multiple things happen. First a prediction is made based on the finite state machine, which determines if the core must jump to the specified address and start executing the instructions found there, or if the core should keep executing the next instructions. No matter the choice, the other option address is stored in this block and it waits the instruction execution to get to the ALU Stage, where the comparison is read and then, depending on the branch type received, one of seven masks is selected and applied to the received result. That way, the real result of the branch is checked.

At that point there are two possible options, either the prediction was correct, or it was not. The first case is easy to handle as there is no need to do anything else, but if the prediction was wrong, it means that some instructions are being executed in the first two pipeline stages and they should not be there. Therefore, it is necessary to clean those two stages, and to do so, the "ClrPipeline" output is set to one. Furthermore, a failed prediction is reported to the finite state machine, that changes its state accordingly to try and avoid failing next time.

The testbench for this block was made by testing many branches that should not jump in a row succeeded by one, or many, that should do the jump, and vice versa. This approach was made to test the finite state machine, but also to test how the block reacts to several branches in a row, as it should be capable of handling such cases. In Figure **4-13** is seen how after a successful prediction, or an unsuccessful one, the finite state machine changes its state, as intended, to try and succeed next time. There the registered signals are also shown in the groups "BpuData0" and "BpuData1", and, as seen in the testbench, the signals in them are propagating from one to the other.

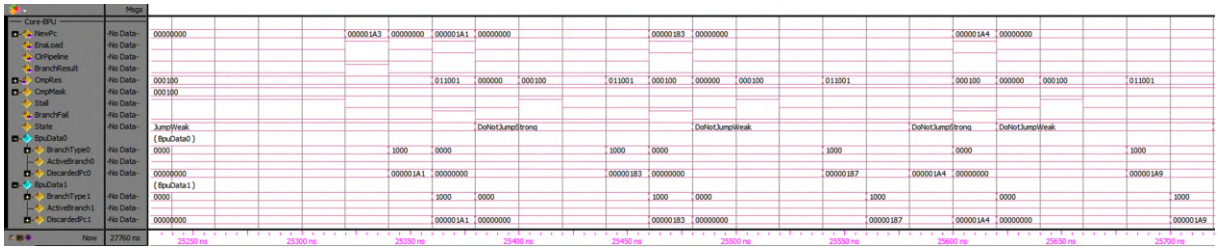


Figure 4-13: BPU testbench.

4.2 Cache Memory

The cache is composed of three main blocks, the ARCU, data cache and instruction cache. The ARCU is a finite state machine that oversees the replacing operations of the cache, and as such it is implemented using a CASE statement and a registered signal to store the state value. In this same block there are three different counters, one to generate the sixteen necessary addresses found in a slice, the second used to count the delay needed to read or write data from memory, and the third to give another delay, this time to change from read to write operations. The maximum values of the last two counters are easily configurable to match the memory specifications used.

In both data cache and memory cache there are sixteen memory slices, which act as memory blocks used to store sixteen words each. Furthermore there is also a block used to select the lesser rating from all the slices, implemented by doing several comparisons to get the absolute minimum value. From there that value was built into a sixteen bit one-hot signal that designate the slice associated with this value. This process is done continuously; but this signal is only used when the ARCU issues the replace signal.

Inside each slice there is a structure of the already mentioned sixteen registers. To read or write them the slice assumes that the four least significant bits of the address carry the information to select one of them. The slice also has a rating counter made to increase its value by twelve each time the core access any associated field of the slice, and each clock cycle that the slice has not used, its rating counter descends by one, This counter was made in such a way that the value is always within the limits of the number of bits. Meaning that it is impossible for the counter to reset its value by its own. Since if its value is zero and the slice is not being used, the rating counter would decrement to negative one, but instead it saturates the value at zero. Inversely when the counter is at a high value and its value should increase by twelve, its maximum value will never go beyond 255, instead saturating the output to that number. The last component of the slice is a 33-bit register used to store the read address associated with the cache. This value can only be edited when the ARCU is making a replacement, and the slice is selected as the one with the lesser rating. This register is made with an extra bit to indicate whether the slice is valid or not. As the cache starts with unknown data in all its registers, the idea is to start the address register with the most significant bit set to one to indicate that the slice does not contain any data.

Figure **4-14** shows the test performed on the data cache. As previously stated, both parts of the cache memory are similar, and their implementation is almost completely symmetrical, therefore they work similarly. At the start of the test the first slice of the data cache has the most significant bit of the address set to one, which indicates that this slice does not contain any data. However the core required data located in the 0x000001D0 to 0x000001DF range and therefore this slice is assigned to that value. The ARCU generates the replace signal and the replace address by concatenating the starting address of the block with the

values generated by its counter. This address are sent, not only to the slice that is being written, but also to the memory, that returns the desired data. This time, the ARCU did not write anything to the memory as it read that this slice data was invalid, yet if that were not the case it would have written the sixteen fields to the memory before reading the new information. The whole process, from the moment when the cache generates the miss signal at 16460 ns to the end of said signal at 17440 ns elapsed 980 ns; all that time the core was stopped, nonetheless, since all the necessary information is now ready it would not need to stop anymore for a time, until it required data from another memory block.

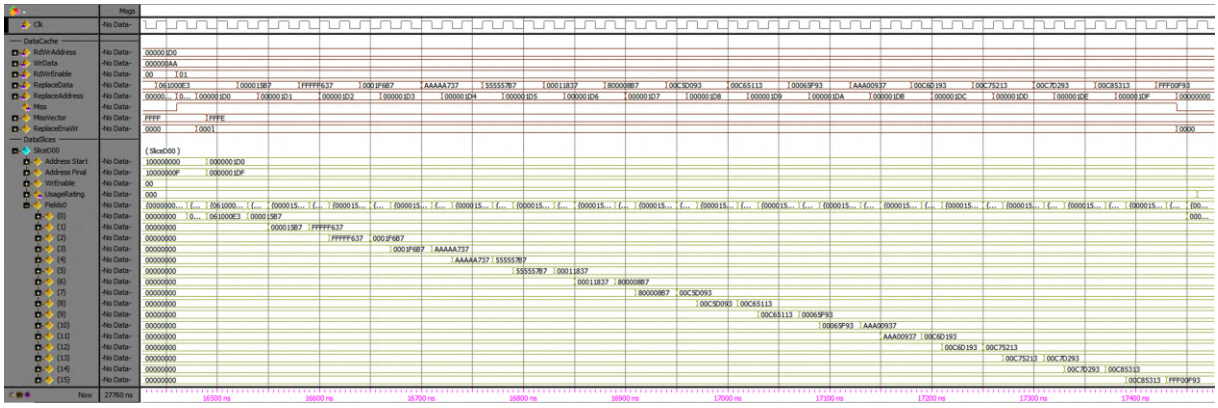


Figure 4-14: Cache Memory testbench.

4.3 Assembler program

To ease the test of the full processor an assembler code was made in the programming language C. This code reads an ASM file, in assembler language that must be present alongside the executable file. Once executed, the program presents a main menu, which by default tells the user that it is impossible to export the results until the assembly and compilation are completed.

The compilation reads the assembler file character by character and classifies each line as a comment, an instruction, or a segment declaration. Comments are expressed by the character "@" at the beginning of them, and from that character to the end of line the compiler will ignore everything. The instructions are case insensitive and depending on the instructions, the code verifies that the necessary fields, such as registers, addresses or numbers, are present. The code also verifies that the immediate data provided for some instructions is valid, by checking that the provided value is less than the maximum value for the number of bits specified for that instruction, and it does the same to the registers' address, those are identified by either the letter R, once again case insensitive, or the character "$". While a segment declaration starts with the character "&" and is used to help the code organize

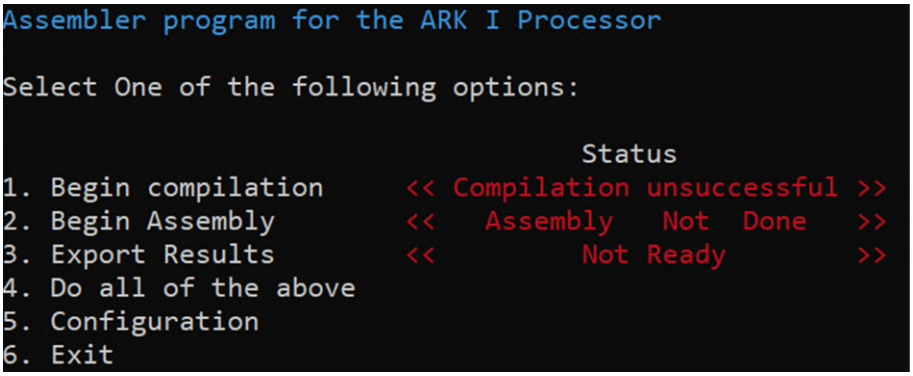


Figure 4-15: Main menu of the compiler.

the instructions according to the cache slices size. The code is also capable of suppressing trailing spaces, tabs and zeros from numbers and registers.

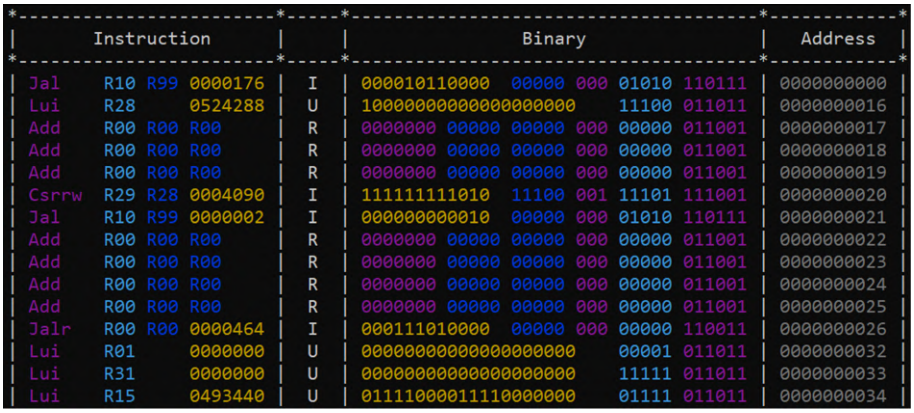
```
  3 | @ Each one of the following segments is intended for max 16 instructions
  4 | & StartCore0
  5 | @ Pc initial value = 0d000 = 0x00
  6 | JAL    R10      176 @ Jump to MainCore1                          | Pc =   0 =000
  7 |
  8 | & StartCore1
  9 | @ Pc initial value = 0d016 = 0x10
 10 | LUI    R28   524288 @ R31 <= 0008 0000 << 12 = 8000 0000        | Pc =  16 =010
 11 | NoOp                @                                           | Pc =  17 =011
 12 | NoOp                @                                           | Pc =  18 =012
 13 | NoOp                @                                           | Pc =  19 =013
 14 | CSRRW  R29 R28 4090 @ Toggle core to act as a single cycle core  | Pc =  20 =014
 15 | JAL    R10        2 @                                           | Pc =  21 =015
 16 | NoOp                @                                           | Pc =  22 =016
 17 | NoOp                @                                           | Pc =  23 =017
 18 | NoOp                @                                           | Pc =  24 =018
 19 | NoOp                @                                           | Pc =  25 =019
 20 | JALR   R00 R00  464 @ Jump to SecondCore                         | Pc =  26 =01A
 21 |
 22 | & ErrorRoutine
 23 | @ Pc initial value = 0d032 = 0x20
 24 | LUI    R01        0 @ R01 <= 0000 0000 << 12 = 0000 0000        | Pc =  32 =021
 25 | LUI    R31        0 @ R31 <= 0000 0000 << 12 = 0000 0000        | Pc =  33 =022
 26 | LUI    R15   493440 @ R15 <= 0007 8780 << 12 = 7878 0000        | Pc =  34 =023
```

Figure 4-16: Example of a piece of an ASM code shown during the compilation process.

At the end of the compilation process and before the assembly, the code counts the number of instructions and according to the different segments defined'. it assigns an address to each instruction. The main goal in this process is that each segment of the ASM code begins at the very start of a memory block.

The assembly reads the compilation results and generates the binary code for each instruction, this is shown to the user highlighting each piece of instruction with a different color to associate it to the respective instruction argument.

After the assembly is completed ,the code allows the user to export the results. There are three possible formats: one is to export them to a CSV file, which was made in case that in some future the software in the processor could be uploaded to it; the second option is



Instruction		Binary	Address
Jal R10 R99 0000176	I	000010110000 00000 000 01010 110111	0000000000
Lui R28 0524288	U	10000000000000000000 11100 011011	0000000016
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000017
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000018
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000019
Csrrw R29 R28 0004090	I	111111111010 11100 001 11101 111001	0000000020
Jal R10 R99 0000002	I	000000000010 00000 000 01010 110111	0000000021
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000022
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000023
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000024
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000025
Jalr R00 R00 0000464	I	000111010000 00000 000 00000 110011	0000000026
Lui R01 0000000	U	00000000000000000000 00001 011011	0000000032
Lui R31 0000000	U	00000000000000000000 11111 011011	0000000033
Lui R15 0493440	U	01111000011110000000 01111 011011	0000000034

Figure 4-17: Assembly process.

to export it as a VHDL block, which would be composed of a multiplexer that returns the desired data, but acts like a read only memory. The third option is to export it as a MIF, which is used to initialize the on-chip memories and since the present project uses one of those as its main memory, this was the most used export option.

Finally, the code can be configured to show some debugging information, but also to change the number of words per slice and the total number of words present in the on-chip memory. These values are set to always be 16 words / slice and 4096 words in memory by default, to better fit this implementation.

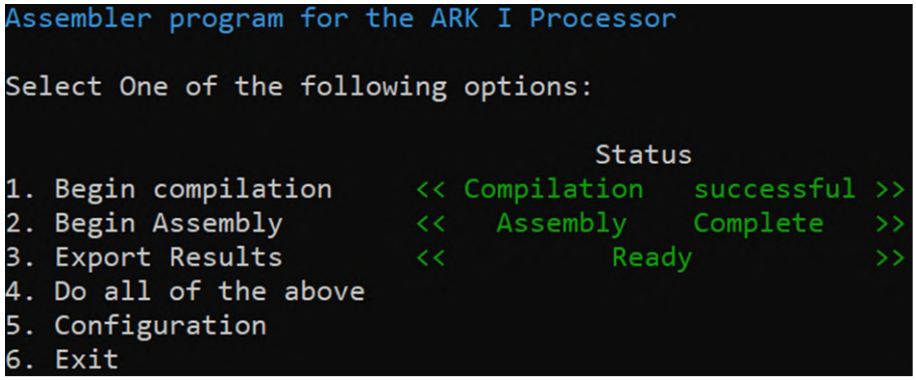


Figure 4-18: Assembler code ready to export data.

Chapter 5

Test and result analysis

Once the processor is implemented and all the blocks have been tested, the next step was to do a general testbench, that executes all instructions one after the other and that ensures that all functionalities are working as expected. To do so a code was made, to test the processor features. It starts by using both cores simultaneously, the first one is immediately redirected to the main area of the code, where all instruction are executed at least twice, in that same section of the code the data forwarding and branch prediction tests that were seen in chapter 4 are present. Meanwhile, the second core starts by configuring itself as a monocycle core and once that configuration is over it is redirected to a compressed version of the main code, where most instructions are executed. Once both cores end their program they end in the final section, where an instruction makes them jump to the same address repeatedly, effectively preventing the execution of any other instruction.

Finally, the pipelined core receives a series of interruptions to check how it manages that situation, for that purpose the addresses three to eight contain the necessary information for the core to find the necessary interruption routines, which are composed of some instruction designed to write specific data to some registers.

5.1 Pipelined mode tests

The pipelined configuration is harder to verify than the monocycle configuration, this is since there are up to five instructions that are being executed simultaneously, and sometimes the BPU, DFU or control unit must intervene to keep everything running smoothly. Figure **5-1** shows a general overview of the testbench start, this shows that just at the beginning there is a cache miss, as there is no data inside the cache and the initial instructions must be fetched from memory, also is shown how after some time the core will ask for more instructions, and while the cache gets that information from the memory, the core must wait.

Fetching and decoding an instruction on any mode takes two main steps, first the cache will evaluate if the PC address is stored within the instruction cache, if it is then the desired data is sent to the core, but if it is not present then the cache must bring the whole block that contains the desired instruction and then give the instruction to the core. The second step is when the core receives the instruction, then the instruction register, and control unit get that data and generate the necessary addresses, immediate data, and control word. The time consumed between the two steps depends on whether the cache has the data or not, for instance the process in the first scenario can be completed in under one clock cycle (20 ns), but if the data is note present it could take up to 1000 ns.

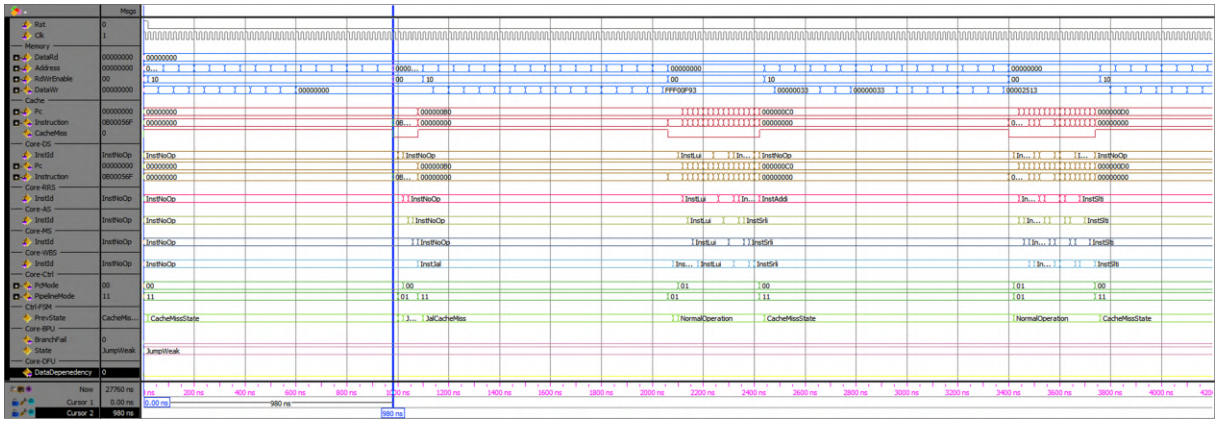


Figure 5-1: Pipeline testbench general overview.

Figure **5-2** shows the execution of several instructions, for instance the instruction "Csrrs", which reads a specific CSR, and then writes to that same register a value while performing an operation. This instruction has memory address 0x00000142 and is received at the Decode Stage at 15200 ns, in that moment the control word is generated by the control unit, and the necessary data and addresses from the instruction are extracted, the next clock cycle, 20 ns later, the same instruction is propagated to the Register Read Stage, as shown with a yellow arrow, in that same moment another instruction arrives to begin its execution, however the "Csrrs" instruction is indifferent to that, as it is now in a different stage in which its control word is getting data from the GPR. 20 ns later, the instruction goes to the ALU Stage, as shown with the green arrow, the instruction in this stage writes information to the CSR, then the next cycle it goes as indicated by the blue arrow to the Memory Stage. This specific instruction does not use, nor needs to do anything in this stage, but this step is imperative, as there is another instruction in the next stage, and potentially there could be a case where the read information from the CSR is needed, and from the memory stage the data forwarding is possible. The final step is done at 15280 ns when following the red arrow, the instruction gets to the Write Back Stage, where the CSR value is written to a GPR. In total the execution of the instruction took 100 ns, from start to finish and in that same timespan, and for this specific case, only another instruction finished its process, but at the

same time four other instruction started its execution and in the next 80 ns will complete their own processes.

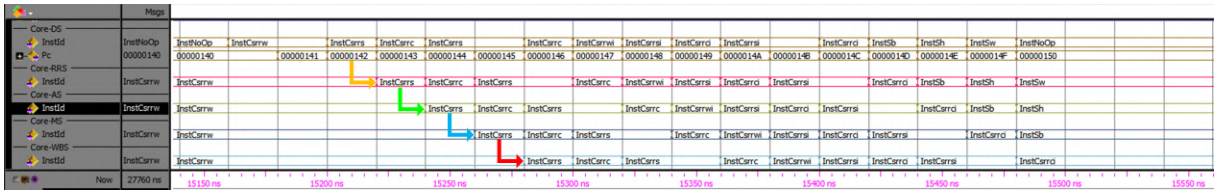


Figure 5-2: Pipeline instruction propagation.

However, sometimes is impossible to fill the pipeline as shown in Figure **5-2**, this could happen for many reasons, but all of them involve the BPU, DFU, control unit or the cache memory. The situation happens due to certain instructions, such as "Jal" or "Jalr", but it also occur due to branch mispredictions or data dependencies, as seen in chapter 4 in the DFU and BPU sections, or even due to exceptions. This last topic will be covered in section 5.3, so for now the testing of stalls induced by instructions is the main topic. Figure **5-3** shows one example of this behavior, since the instruction with address zero is "Jal", and it triggers the finite state machine, disables the pipeline registers, and program counter, and takes over the control unit to start issuing a "No Operation" instruction to the Decode Stage. As "Jal" is an absolute jump, it requires to finish any previous instruction, and once that is completed, the new value is uploaded to the PC, this could make a cache miss, and this is such case, for that reason a while after the finite state machine finishes its procedure and then the core resumes its normal operation. In most cases the decision to stop the core to do something is bound to the necessity of freeing the pipeline stages, to use the most recent data to make a decision, load the most recent data somewhere or just avoid breaking atomic segments of the code.

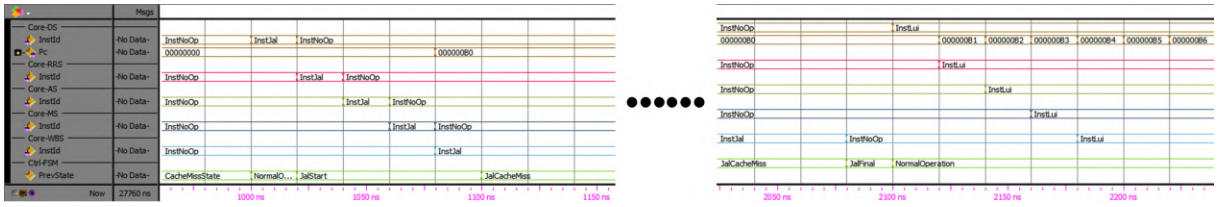


Figure 5-3: Instruction execution introducing a stall to the pipeline stages.

5.2 Monocycle mode test.

The monocycle mode is not the default configuration of any core in this processor, therefore, to test it is necessary to first set it up. To do so the second core starting program executes

the necessary operations, Figure **5-4** shows this process, by first executing at 1000 ns the load upper immediate instruction which loads the value 0x80000000 to the twenty-eighth GPR (R28), after that instruction three "no operation" instructions are executed to prevent any data dependency. The value stored to R28 is vital, as to change the core from monocycle mode to pipelined mode the MPS CSR most significant bit must be set to one, to do so the instruction "Csrrw" instruction is executed at 1080 ns, this instruction writes the contents of R28 to the specified CSR, which in this case was the machine pipeline status register. However, the instruction takes a while to be executed and some other instructions can be received and executed before the MPS value is changed, however once the most significant bit is set, and the control unit's finite state machine end any process, the core will begin its process to configure itself in monocycle mode. In 1240 ns the finite state machine begins the execution of the states associated with this process, which block the core from reading any other instruction for a while, let any previously received instruction to complete its execution, and set the new pipeline status in the MPS, which in turn disables the pipeline registers, DFU and BPU. This whole process takes 160 ns, meaning that after 1400 ns the core now acts as a monocycle core and therefore any instruction is now executed in just one cycle.

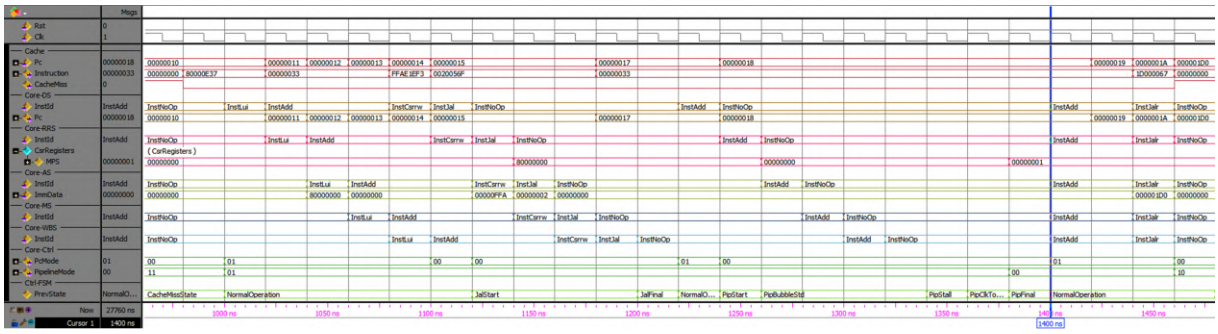


Figure 5-4: Pipeline mode to monocycle mode transition.

Once the core is in monocycle mode, most instructions are executed, as previously in the pipelined simulation all instructions were used multiple times, and each block was also tested, this time the testbench was centered on some specific instructions, to prove the functionality of the monocycle mode. Figure **5-5** shows both cores, executing their own instructions, the upper core is the one operating in pipelined mode and executing the previously mentioned testbench of its own, while down the monocycle core is completing its own instructions. The instructions that both cores are using are the same, but this figure makes rather evident the difference in execution mode, depending on the core configuration. The monocycle instructions, as seen at 2460 ns go through the different stages immediately, reading data, performing the necessary ALU operation, interacting with the memory (and waiting for it if necessary), and writing data back to the registers, all in one single clock cycle.

As previously stated, the monocycle mode deactivates the DFU and BPU, and therefore

Figure 5-5: Pipelined core and monocycle core execution.

can only be stopped by cache misses, exceptions, and the mode change process. This last possibility is depicted in Figure **5-6**, as the monocycle testbench comes to and end and the core cycles back to pipelined mode, the code begins this process by using the same instruction and the same data that were used when the core was going from pipelined to monocycle, meaning using the GPR that has the wanted value (R28 with value 0x80000000) and the "Csrrw" instruction. The same value used to go from pipelined mode to monocycle mode is used to go the other way around, this is due to the way the MPS was implemented, as the most significant bit indicates that the core's mode is going to toggle. At 5140 ns the finite state machine began the same process described previously, and at 5300 ns, 160 ns later, the core is configured once more as a pipelined core.

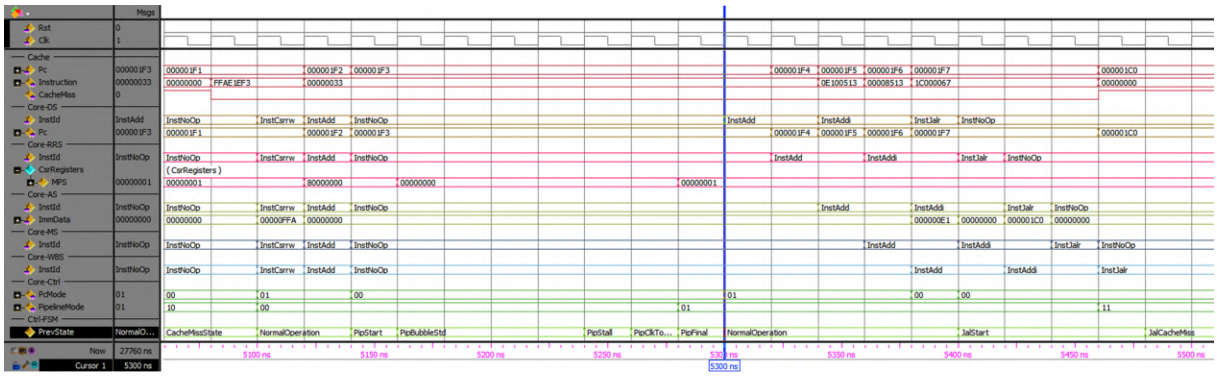


Figure 5-6: Monocycle mode to pipelined mode transition.

5.3 Exceptions management

The Ark-I processor can identify and manage exceptions coming from up to nine different sources, eight of which are triggered by the peripheral modes and their IRQ signals, which

are distributed equally between both cores, and the identification of an unknown instruction, which triggers the error exception. In any case the finite state machine inside the control unit has a series of states specialized in handling all kinds of exceptions.

For starters the MIE is used to enable interruptions, if its eleventh bit is zero, then no exceptions can occur. Once an exception is received and the MIE is active, the MIP CSR sets a specific bit based on the received exception, to acknowledge the reception of said event, then, once the control's finite state machine reaches the "NormalOperation" state, the exception begins its execution, this whole process can be seen in Figure **5-7**. In which is shown that the exception is received at 28440 ns, where the least significant bit of the IRQ signal is set to one, the next cycle the MIP value, called "MipRd" by the control unit, sets the value 0x00000002, indicating that there is an interrupt pending. However, in that moment the finite state machine is executing the "Jal" instruction and therefore is not available, but at 28580 ns that process finishes, and the exception begins its execution. Part of that execution process is the establishment of the priority level of this exception, and getting the memory address where the exception subroutine is found. To do so, the processor must get the value stored in MTVAL, which is the base address of the table that contains the addresses for all exception subroutines, add to that base address an offset based on the specific exception, and then get the subroutine address from memory using that information. That way the PC is set to the desired address, and the exception can begin its execution at 28760 ns.

Figure 5-7: Single peripheric exception.

All exceptions should have a return instruction at their subroutines end, to resume the main

code execution, however the assembler program does not enforce it. The return instruction could be “Mret”, “Sret” or “Uret”, as the processor does not differentiate between them, and this instruction is also handled by the finite state machine. Figure **5-8** shows this process, since at 29840 ns the core receives the instruction, the state machine then waits until the pipeline stages are cleared, then pops out of the MNEV register the priority set when the interruption first occurred, and reestablish the program counter to continue the main code operation.

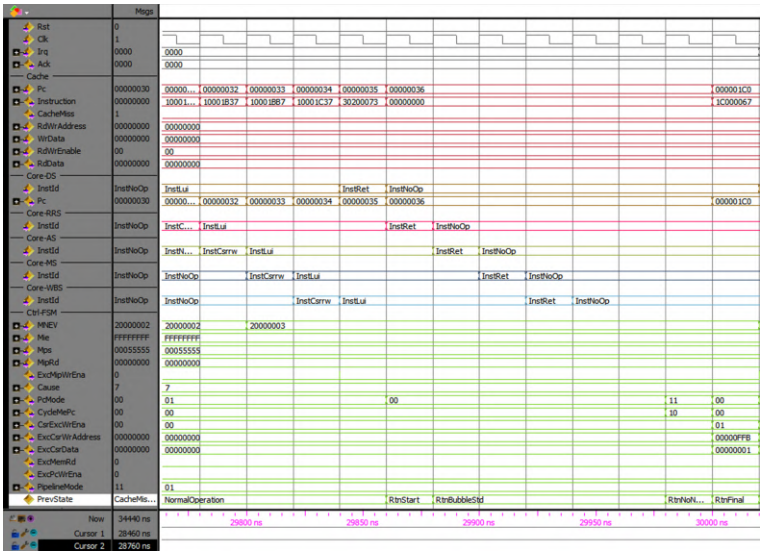


Figure 5-8: Single exception return.

The previously depicted method is used only to execute the exception subroutine, however, the core was made to allow multiple interruptions at once, nesting one into the other, to do so, the control unit and CSR has a priority system and four different MEPC registers. Figure **5-9** shows how at 30100 ns four different peripherals have requested an interruption for themselves, the MIP saves the information each one, and at 30140 ns the first interruption begins its execution, which clears part of the MIP, at the same time the control unit sets the “MnevLevel” to one, as that is the current nesting level and sets the current priority level to the one associated to this first interruption. The maximum priority level, and therefore the exception that cannot be interrupted, is the error exception, from there the interruption associated to the most significant bit of the IRQ is the next with higher priority, and then the next IRQs in descending order have lower priorities associated. Each time a new nested exception is being executed the finite state machine also saves the previous PC to the MEPC registers, which can hold up to four different values in order of arrival.

Once each exception completes its own subroutine the finite state machine executes the return states, however, to do all the necessary changes an extra state is added, as this time the MEPC registers must give the old PC value to the core and move all the other saved values one register up, in addition to lower the “MnevLevel” and restore the previous priority

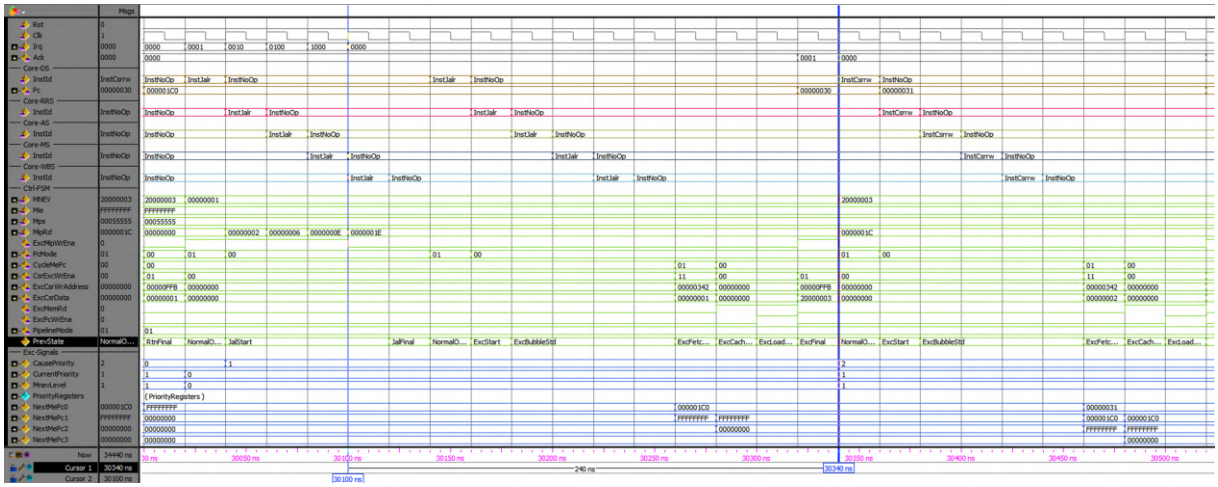


Figure 5-9: Nested exceptions.

level. All these changes are made to restore the control flow of the former subroutine, but is important to consider that under any circumstances the core does save any register value, meaning that the information in the GPR before the interruption, nested or not, are probably different to those after the interruption, which is why is vital for the programmer using this feature to implement the necessary code to store the data context of the program at the beginning of each exception.

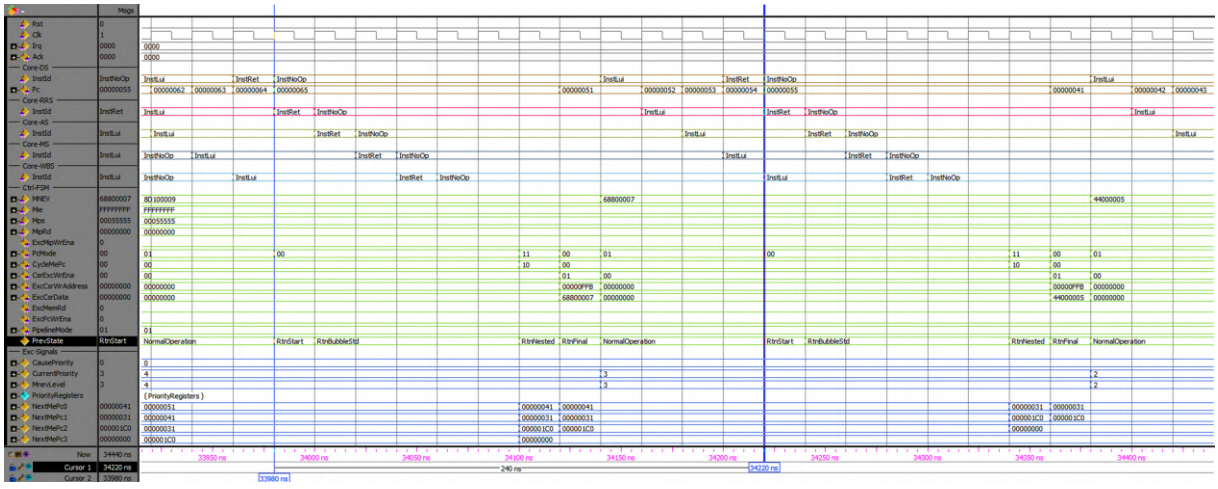


Figure 5-10: Core returning from nested exceptions.

Chapter 6

Conclusion and future work

The design process of the ARK-I processor was a vital step of the project, as the variable pipelining system required modifications to almost every block, in which the key element was to follow a top-down methodology, and according to that the main blocks were defined, and then into each one of them the necessary considerations were done, to have both operative modes working. To do so, the starting point was to analyze how each block would behave in certain scenarios, or by evaluating the theoretical response to specific sets of inputs and sometimes even consecutive inputs. The core design depended heavily on the pipeline stages and therefore the initial outline was the key factor to discover some interesting combinations of inputs that could cause errors; to prevent such failures the core and the control unit were modified until a satisfactory design was achieved, one in which all foreseeable problems were manageable. On the other hand, the design depended heavily on the selected ISA, as the RISC-V standard has several modules, variations, and implementation possibilities, thus each instruction required to be fully understood to break its execution into atomic requirements that were assigned to different blocks to complete every single instruction's execution.

Consequently the control unit was one of the most meticulously designed blocks, as in it the instructions were decoded into their respective control words, and several other signals are sent to control the instruction flow in the pipeline stages, decide the next instruction address, and when to stall the core and insert "no operation" instructions to maintain the correct execution. The control unit's finite state machine was also designed carefully, as its function is vital to the core functioning. Its main function of managing the exception transitions and returns, while considering the nesting priority level, nesting level, and cause, without interfering with the instructions that are already being executed, and accounting for possible cache misses, branch mispredictions and data dependencies. All the design features were selected to minimize the possible errors found in the processor's implementation stage, but some of them had in mind the idea of extracting some extra performance from the core,

to achieve a better throughput or to simplify the execution process.

On the other hand, the cache memory design and implementation proved to be a simple way to optimize on some cases the memory operations, which directly affects the processor's performance positively. By having a data cache and memory cache apart, both have enough space and there is no possibility of losing important information of any kind in case one need more space than the other, furthermore, is possible to edit both sizes easily, to tune then to the demands of the task at hand. Its other remarking features is that it can change the memory delay time to adjust to most memories requirements, and has the capability to fetch blocks of necessary data from memory, writing the old data block that will be replaced and getting the new one, thus preventing data loss and getting not only the one necessary 32-bit word, but also many others in case they are needed anytime soon.

As the design process was made with such care, the implementation process ended up being a matter of following the previously designated features of each block, with some exceptions such as the adder, multiplier, shifters, and counters, which specific architecture were not defined in the design process, but rather their performance goal. For this implementation process VHDL was the key element, as it allows full control over what FPGA's elements are being used. The implementation process was bonded with simulations for each block which were in some cases extensive and therefore, keeping track of the desired results could be difficult, however, the usage of automatic testbench using CSV files greatly simplified this process and was, along with Model Sim, the most important tools to easily test each block functionality.

The assembler program was also one of the most important advancements in this project, as it moves the instructions address automatically to match the cache slice size and memory size, as well as to quickly traduce the assembler code to machine code, while detecting invalid instructions, data or registers addresses and generate the necessary code sections to make it easy for the programmer to identify which instructions are being executed when the code is running. Furthermore, the structure of the assembler was made to allow students and people that are getting familiar with the processor, to understand how the final machine code is generated, and what bit positions correspond to each field. This program was also key to test all the processor's features, as is easier to change arguments and instructions in assembler language than it is to do so in binary.

The final testbench tested that both operating modes proposed, monocycle and pipelined, were completely operational, executing the given instructions with the intended results and managing all the possible problems that arose in the process, such as data dependencies and branch predictions for the pipelined core, mode transitioning, and pipeline register deactivation and data rerouting for the monocycle and cache misses and exceptions for both modes. While not evident in this case, both modes have their own advantages and disadvantages, as the monocycle core is better at dealing with code segments with highly

amounts of data dependencies between instructions or branches, while the pipelined core can execute instructions faster, achieving a better throughput.

Even though several considerations were made to get a good performance from this core, there are many aspects that could be worked on to improve it. For starters, the already mentioned pipeline mode runs at the same clock frequency as the monocycle mode, making it slower than it should be. The local clock block and the theoretical necessary states in the control's finite state machine are in place to change the clock frequency for both modes, nonetheless the ARK-I does not have this feature yet, which probably would get a better throughput from the processor. On the same topic is important to calculate the maximum frequency of both modes, as the 50 MHz clock frequency used in this version is inherited from the base frequency of the Cyclone V FPGA, and therefore with a timing analysis could be possible to set a better frequency for both cases that increases the processor performance; this frequency however should be rounded to get that the monocycle clock frequency would be a multiple of the pipelined clock frequency. The local clock module was not the only one left empty, as the divisor was also not implemented, but the space for it is available, as well as all the instructions, operation codes and datapath to do so. That block would require performing the 32-bit integer division and remainder operation of unsigned and signed numbers in under one clock cycle, for which the author made a theoretical combinational design of an integer divisor module, nevertheless said design was not implemented, and in reality any divisor that satisfies the timing requirements can be used.

The processor is implemented in VHDL for an FPGA with the previously mentioned DE1-SOC board on mind, but loading the described hardware to the chip was not done in this project and therefore is left as future work to get this processor to the fully functional on chip stage.

The assembler code can also be upgraded in future versions to calculate memory address easily based on the code segments tags, and even to analyze the code to detect highly dependent clusters of successive instructions or even multiple successive branch instructions and change the core from pipelined mode to monocycle mode to extract even more performance of the processor from the assembly stage. This, however, would require analyzing the time impact of the data dependency and branch misprediction stalls to know how many of those are needed to justify the round-trip mode toggle, which by itself consumes a considerable amount of time.

The cache memory, as previously stated, was a good addition to the project to avoid losing time on memory read and write transactions, however, the way the cache works could be improved, to use the time in which it is idling to bring new data blocks that could be used in the near future, reducing the time that the core is stalled waiting for the cache miss to be solver in some scenarios. This block could also be improved by adding the necessary resources to allow data writing to the main memory more frequently or even on demand by

the other cache, letting both cores to share data and hence execute the same code, divided into two different threads that are not independent. Similarly, the DFU and BPU units could also be upgraded, the first one to allow data forwarding from the ALU stage, and the second one, by improving the prediction system.

Bibliography

[1] M. T. Banday, "A study of current trends in the design of processors for the internet of things," ACM Int. Conf. Proceeding Ser., Tech. Rep., 2018. [Online]. Available: doi:10.1145/3231053.3231074

[2] C. Papadimitriou, "Algorithms, complexity, and the sciences," Proc. Natl. Acad. Sci. U. S. A., Tech. Rep. vol. 111, no. 45, pp. 15881–15887, 2014. [Online]. Available: doi:10.1073/pnas.1416954111.

[3] R. S. T. I. R. N. J. Schuchart, D. Hackenberg and M. K. Patterson, "The shift from processor power consumption to performance variations: fundamental implications at scale," Comput. Sci. - Res. Dev., Tech. Rep. ol. 31, no. 4, pp. 197–205, Nov 2016. [Online]. Available: doi:10.1007/s00450-016-0327-2

[4] S. Bhunia and M. Tehranipoor, "System level attacks countermeasures," Hardware Security, Elsevier, Tech. Rep. pp. 419–448., 2019.

[5] J. Davis and R. Reese, "Calculating maximum clock frequency," FAO, Tech. Rep. 1–22, 2008.

[6] J. L. H. D. A. Patterson, "Computer organization and design, fifth edition: The hardware/software interface 5th, ed," Morgan Kaufmann Publishers Inc, Tech. Rep., 2013.

[7] A. Akram and L. Sawalha, "A study of performance and power consumption differences among different isas," Proc. - Euromicro Conf. Digit. Syst. Des. DSD, Tech. Rep. pp. 628–632, 2019. [Online]. Available: doi:10.1007/s00450-016-0327-2

[8] J. M. E. Blem and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," Proc. - Int. Symp. High-Performance Comput. Archit, Tech. Rep. pp. 1–12, 2013. [Online]. Available: doi:10.1109/HPCA.2013.6522302.

[9] "History of risc-v," Tech. Rep., accessed Feb. 02, 2023 [Online]. [Online]. Available: https://riscv.org/about/history/

[10] S. Greengard, "Will risc-v revolutionize computing?" Commun. ACM, Tech. Rep. vol. 63, no. 5, pp. 30–32, 2020. [Online]. Available: doi:10.1145/3386377.

[11] A. Waterman and K. Asanovic, "The risc-v instruction set manual volume i: User-level isa," Tech. Rep., 2017.

[12] ——, "The risc-v instruction set manual , volume ii: Privileged architecture, vol ii," Hanna, Tech. Rep., 2017.

[13] B. Venu, "Multi-core processors - an overview," Tech. Rep., 2011.

[14] C. X. J. Bui and S. Gurumurthi, "Understanding performance issues on both single core and multi-core architecture," Tech. Rep., 2007.

[15] J. L. Hennessy and D. a Patterson, "Computer architecture, fourth edition: A quantitative approach," Tech. Rep. no. 0. 2006.

[16] M. Y. B. Youne, "Branch prediction in pipeline computer system," University of Jordan, Tech. Rep., 2008.

[17] M. G. H. C. Chang, B. Li and K. W. Cameron, "How processor speedups can slow down i/o performance," Proc. - IEEE Comput. Soc. Annu. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS, Tech. Rep. pp. 395–404, February 2015. [Online]. Available: doi:10.1109/MASCOTS.2014.55

[18] G. S. Rao, "Performance analysis of cache memories," J. ACM, Tech. Rep. vol. 25, no. 3, pp. 378–395, 1978. [Online]. Available: doi:10.1145/322077.322081

[19] Y. L. et al, "A fully associative, tagless dram cache," ACM SIGARCH Comput. Archit. News, Tech. Rep. vol. 43, no. 3S, pp. 211–222, 2016. [Online]. Available: doi:10.1145/2872887.2750383

[20] kammoh, "Orca risc-v," YARA, Tech. Rep. [Online]. Available: https://github.com/riscveval/orca-1

[21] YosysHQ, "Picorv32," Konicaminolta, Tech. Rep., 2019 [Online] accessed Feb. 09, 2022. [Online]. Available: https://github.com/YosysHQ/picorv32

[22] C. C. et al, "Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product," Proc. - Int. Symp. Comput. Archit, Tech. Rep. vol. 2020-May, pp. 52–64, 2020. [Online]. Available: doi:10.1109/ISCA45697.2020.00016

[23] J. S. B. R. and A. V. A., "Implementacion de un procesador risc-v en vhdl para aplicaciones academicas , soportando programacion a alto nivel y perifericos a la medida," Pontifcia Universidad Javeriana, Tech. Rep., 2021.

[24] O. S. E. S. I. R. Diaz Gamarra and M. D. L. Ortiz, "Diseño de un prototipo de controlador pid en drones con procesador a la medida en arquitectura risc-v para implementación en fpga," Pontifcia Universidad Javeriana, Tech. Rep., 2020.

[25] T. M. P. Jamieson and E. Schonauer., "Framework and tools for undergraduates designing risc-v processors on an fpga in computer architecture education," Proc. - 6th Annu. Conf. Comput. Sci. Comput. Intell. CSCI 2019, Tech. Rep. pp. 778–781, 2019. [Online]. Available: doi:10.1109/CSCI49370.2019.00148

Appendix A

Appendix: RV32IM instruction set

The RV32IM instruction set is composed of sixty different instructions, of which six of them were not implemented, as they were used in specific cases that were considered not necessary for the current implementation. Those instructions were: FENCE, FENCE.I, ECALL, EBREAK, WFI and FENCE.VM.

The other fifty-four instructions were implemented, and were classified into ten different groups shown below:

U-Type instructions

Table A-1: U-Type instructions

Instruction	Format	Name	Implementation
LUI	U	Load Upper Imm	$X[rd] = Imm << 12$
AUIPC	U	Add Upper Imm to PC	$X[rd] = pc + Imm << 12$

The type U instructions are used to either load immediate information to a specific register or to load information to a specific register from the PC and immediate information. These instructions are characterized by having the longest immediate data field, of 20 bits.

Immediate arithmetic-logical instructions

These instructions comprise I and R format. I format is usually for immediate instructions, with a long 12-bit immediate field, however in this case the R instructions present replace the RS2 field with the SHAMT field, used to indicate the number of shifts that should be performed. This group of instructions is one of three ALU oriented groups and is usually utilized when one of the operators is not found in any registers and is small enough to fit the immediate field.

Arithmetic-logical instructions

Table A-2: Immediate arithmetic or logical instructions

Instruction	Format	Name	Implementation
ADDI	I	Add Imm	$X[rd] = X[rs1] + sext(Imm)$
SLTI	I	Set Less Than Imm	$X[rd] = 1?X[rs1] < sext(Imm)$
SLTIU	I	Set Less Than Imm Unsg	$X[rd] = 1?X[rs1] < uext(Imm)$
XORI	I	Xor Imm	$X[rd] = X[rs1]XORsext(Imm)$
ORI	I	Or Imm	$X[rd] = X[rs1]\|sext(Imm)$
ANDI	I	And Imm	$X[rd] = X[rs1]\&sext(Imm)$
SLLI	R	Shift Left Logical Imm	$X[rd] = X[rs1] << shamt$
SRLI	R	Shift Right Logical Imm	$X[rd] = X[rs1] >> ushamt$
SRAI	R	Shift Right Arith. Imm	$X[rd] = X[rs1] >> sshamt$

Table A-3: Arithmetic-logical instructions

Instruction	Format	Name	Implementation
ADD	R	Add registers	$X[rd] = X[rs1] + X[rs2]$
SUB	R	Subs registers	$X[rd] = X[rs1] - X[rs2]$
SLL	R	Shift Left Logical	$X[rd] = X[rs1] << X[rs2]$
SLT	R	Set if Less Than	$X[rd] = 1?X[rs1] < sX[rs2]$
SLTU	R	Set if Less Unsigned	$X[rd] = 1?X[rs1] < uX[rs2]$
XOR	R	Xor registers	$X[rd] = X[rs1]XORX[rs2]$
SRL	R	Shift Right Logical	$X[rd] = X[rs1] >> uX[rs2]$
SRA	R	Shift Right Arith	$X[rd] = X[rs1] >> sX[rs2]$
OR	R	Or Registers	$X[rd] = X[rs1]\|X[rs2]$
AND	R	And registers	$X[rd] = X[rs1]\&X[rs2]$

This group of instructions only uses the R format, and therefore its operations always have two different registers being used, RS1 and RS2. This is the second group of instructions that are ALU oriented, and are usually used to do consecutive calculations, using previous instructions results.

CSR operations

Table A-4: CSR operations

Instruction	Format	Name	Implementation
CSRRW	I	CSR Read Write	$CSR[Imm] = X[rs1]$ $X[rd] = CSR[Imm]$
CSRRS	I	CSR Read and Set	$CSR[Imm] = CSR[Imm] \mid X[rs1]$ $X[rd] = CSR[Imm]$
CSRRC	I	CSR Read and Clear	$CSR[Imm] = CSR\& \ X[rs1]$ $X[rd] = CSR[Imm]$
CSRRWI	I	CSR Read Write uImm	$CSR[Imm] = uDat$ $X[rd] = CSR[Imm]$
CSRRSI	I	CSR Read and Set uImm	$CSR[Imm] = CSR[Imm] \mid uDat$ $X[rd] = CSR[Imm]$
CSRRCI	I	CSR Read and Clear uImm	$CSR[Imm] = CSR\& \ uDat$ $X[rd] = CSR[Imm]$

The CSR operations, as the name implies, are used to read, and write to the control and status registers, all of them use the I format, however in some cases the RS1 field is replaced by the Data field, which contains a 5-bit immediate data to be written.

Return instructions

Table A-5: CSR operations

Instruction	Format	Name	Implementation
URET	?	Return from exceptions in U-Mode	$Pc <= MePc$
SRET	?	Return from exceptions in S-Mode	$Pc <= MePc$
MRET	?	Return from exceptions in M-Mode	$Pc <= MePc$

The return instructions are defined by not having any specific format, there are not any fields to be filled by the user and as such do not have any arguments. The three instructions are executed identically by the ARK-I core. These instructions are used to return from subroutines.

Memory data load instructions

Table A-6: Memory data load instructions

Instruction	Format	Name	Implementation
LB	I	Load Signed Byte	$X[rd] = sext(M[X[rs1] + sext(Imm)][7:0])$
LH	I	Load Signed Hex	$X[rd] = sext(M[X[rs1] + sext(Imm)][15:0])$
LW	I	Load Word	$X[rd] = sext(M[X[rs1] + sext(Imm)][31:0])$
LBU	I	Load Unsigned Byte	$X[rd] = uext(M[X[rs1] + uext(Imm)][7:0])$
LHU	I	Load Unisgned Hex	X[rd] = uext(M[X[rs1]+uext(Imm)][15:0])

These I format instructions are used to load information from the memory (M) to a specific register. They allow to load the whole word, or just part of it depending on what needs the programmer. The twelve immediate bits used address the memory allows for up to a 4095 positions address offset.

Memory data store instructions

Table A-7: Memory data store instructions

Instruction	Format	Name	Implementation
SB	S	Store Byte	$M[X[rs1] + sext(Imm)] = x[rs2][7:0]$
SH	S	Store Hex	$M[X[rs1] + sext(Imm)] = x[rs2][15:0]$
SW	S	Store Word	$M[X[rs1] + sext(Imm)] = x[rs2][31:0]$

These S format instructions are used to store information in the memory, by reading it from a specific register. They allow different sizes of data to be stored, depending on the selected instruction.

Jump instructions

Table A-8: Jump instructions

Instruction	Format	Name	Implementation
JAL	I	Jump to Address	$X[rd] = PC + 4$ $PC = PC + sext(Imm)$
JALR	I	Jump to Address(Retrn)	$X[rd] = PC + 4$ $PC = (X[rs1] + sext(Imm))$

These instructions utilize the I format and their function is to move the PC from one memory address to another, effectively jumping from one section of the code to another. These jumps do not require any kind of confirmation, and are executed immediately after they are received, meaning that the ARK-I cores do not need to use the BPU as there are no branches to predict, hence, they are executed by the control's finite state machine.

Branch instructions

Table A-9: Branch instructions

Instruction	Format	Name	Implementation
BEQ	S	Branch if Equal	$PC = PC + sext(Imm)?X[rs1] == X[rs2]$
BNE	S	Branch if not Equal	$PC = PC + sext(Imm)?X[rs1]/ = X[rs2]$
BLT	S	Branch if $<$	$PC = PC + sext(Imm)?X[rs1] < X[rs2]$
BGE	S	Branch if $>=$	$PC = PC + sext(Imm)?X[rs1] >= X[rs2]$
BLTU	S	Branch if $<$ Unsg	$PC = PC + sext(Imm)?X[rs1]u < uX[rs2]$
BGEU	S	Branch if $>=$ Unsg	$PC = PC + sext(Imm)?X[rs1]u >= uX[rs2]$

These instructions utilize the S format and their function is to jump to a certain address depending on a condition, meaning that the jump does not always occur, thus they heavily depend on the BPU for they execution on pipelined mode.

Multiplication and division instructions

Table A-10: Multiplication and division instructions

Instruction	Format	Name	Implementation
MUL	R	Multiply	$X[rd][31:0] = X[rs1]xX[rs2]$
MULH	R	Multiply high	$X[rd][63:32] = X[rs1]sxsX[rs2]$
MULHSU	R	Multiply High Sgn. Unsg	$X[rd][63:32] = X[rs1]sxuX[rs2]$
MULHU	R	Multiply High Unsg	$X[rd][63:32] = X[rs1]uxuX[rs2]$
DIV	R	Integer Divide	$X[rd] = X[rs1]s/sX[rs2]$
DIVU	R	Unsigned Divide	$X[rd] = X[rs1]u/uX[rs2]$
REM	R	Reminder	$X[rd] = X[rs1]\ s\%sX[rs2]$
REMU	R	Unsigned Reminder	$X[rd] = X[rs1]u\%uX[rs2]$

These are the final instructions that are ALU oriented and are also the ones that are part of the multiplication module of the RV32IM standard. They use the R format to use two registers as inputs, and even if the division and reminder instructions are present, they are not implemented in the processor.

Appendix B

Appendix: Implemented CSRs

The RISC-V standard can address up to 4096 32-bit CSR, limitation imposed by the CSRs instruction on the instruction set that only allow twelve bits for addressing, however not all the possible addresses are defined by the standard and some are left empty for future implementations or for designers to create their own CSRs according to their needs.

Table B-1: Control and Status Registers

Name	**Address**	**Description**
MISA	0x0301	Machine ISA
MIE	0x0304	Machine Interrupt Enable
MTVEC	0x0305	Machine Trap Vector Base Address
MCAUSE	0x0342	Machine Exception Cause
MTVAL	0x0343	Machine Trap Value
MIP	0x0344	Machine Interrupt Pending
MARCHID	0x0f12	Machine Architecture ID
MIMPID	0x0f13	Machine Implementation ID
MHARTID	0x0f14	Hardware Thread ID
MPS	0x0FFA	Machine Pipeline Status Register
MNEV	0x0FFB	Machine Nested exception value
MEPC0	0x0FFC	Machine Exception Program Counter
MEPC1	0x0FFD	Machine Exception Program Counter
MEPC2	0x0FFE	Machine Exception Program Counter
MEPC3	0x0FFF	Machine Exception Program Counter

Machine ISA (MISA): This read only register has the required bits set to one to indicate that the processor uses the RV32IM instruction set, which translates to the value 0x40001100.

Machine Interrupt Enable (MIE): this read/write register is used to enable or disable excep-

tions and interruptions, it is composed of several fields associated to user mode, supervisor mode and machine mode. This processor only uses the eleventh bit to indicate whether the interruptions are enabled or not.

Machine Trap Vector Base Address (MTVEC): This read/write register stores the memory address where the first subroutine's address is located. The following four memory addresses from MTVEC hold the other subroutine's addresses.

Machine Exception Cause (MCAUSE): This read/write register stores the last exception cause; each exception has a unique cause ID that is stored in it during the exception management states in the control's finite state machine.

Machine Trap Value (MTVAL): This read/write register stores the last unknown instruction that triggered the error exception. Its value is set in the exception management states.

Machine Interrupt Pending (MIP): This read/write register is responsible for setting a one in each bit associated to a specific exception source, the current ARK-I implementation only uses the five least significant bits and is read by the control unit to start the necessary processes to execute the corresponding exception subroutine.

Machine Architecture ID (MARCHID): This read only register stores the architecture's identifier, for this implementation the value stored is 0x65827573, corresponding to the four characters of the processor's name "ARKI".

Machine Implementation ID (MIMPID): This read only register stores the version of the processor implementation, as this is the first implementation its value is set to one.

Hardware Thread ID (MHARTID): This read only register is used to identify each hardware thread, or in other words each core, the ARK-I starts counting from core 0 to 1, as there are only two cores implemented.

Machine Pipeline Status Register (MPS): This read/write register was created to store the value of the current pipeline mode, and to be used to change the core's operating mode, from monocycle to pipelined and vice versa. Its Most significant bit is used to toggle from one mode to the other, while its least significant bit stores the current operating mode. This bit is changed by the control unit when the core is transitioning from one mode to the other.

Machine Nested exception value (MNEV): This read/write register was made to store the relevant information associated to the nested exceptions. Its twelve most significant bits are reserved to store the four possible levels of priorities, each one associated to an exception nesting level, its bit 19 through 16 are reserved and must always be set to zero. Finally, its bits three to one store the number of nested exceptions present and the least significant bit is used to determine whether the core accepts nested exceptions or not.

Machine Exception Program Counters (MEPC): These four registers are set to be read only, except for MEPC0, which is a read/write register. They store the previous values of the PC,

to restore them after an exception. The four registers are built to allow inputting information, and pushing the previous data from register to the next, and reading information, while moving data one step from the final register to the first, essentially working as a first in, last out architecture, that is managed by itself.

Appendix C

Appendix: Control unit's finite state machine

Inside the control unit there is a finite state machine, previously shown simplified in Figure **3-13** on chapter 3. As this machine is used to control some of the core most important features, such as exceptions, and mode cycling, is relevant to dig deep into it.

The machine is separated into five different states groupings and has two states that do not belong to any group whatsoever. The individual states are the "Normal operation" state and the "Cache miss" state, the first is active in most cases as this state is used when any common instruction is being executed, and the core does not need to execute any of the cases that are managed by the other states. The "cache miss" state, as its name implies, is activated when the cache memory generates the miss signal, and its purpose is to avoid any state change during this period. Even if it is not shown in Figure **C-1**, most final states of each group can bypass the "Normal operation" state to go to the start any other group, if needed. This possibility to bypass that state was done to save one clock cycle in some cases.

For starters, there is a subgroup of states that is used four different times, that is the "Bubble states", these states are used when the control unit needs to finish executing all the current instructions before doing anything else. Figure **C-2** shows the different states and transitions of this subgroup, the first thing it does it to assess if the core is in monocycle mode or if it is rather in pipelined mode, while issuing the order to stall the PC. This first division between both modes is important to accelerate the process because the core in monocycle mode does not need to empty the pipeline stages due to all instructions being executed in one clock cycle. In the case where the core is pipelined, then the other states handle the emptying of the stages, to do so the machine counts to five, one per each stage, while stalling the PC, in this mode no new instructions are received, while the previous are completing its execution. In this stage there are some eventualities that could happen, firstly a data dependency could be found, meaning that some pipeline stages stop their processes and therefore more time

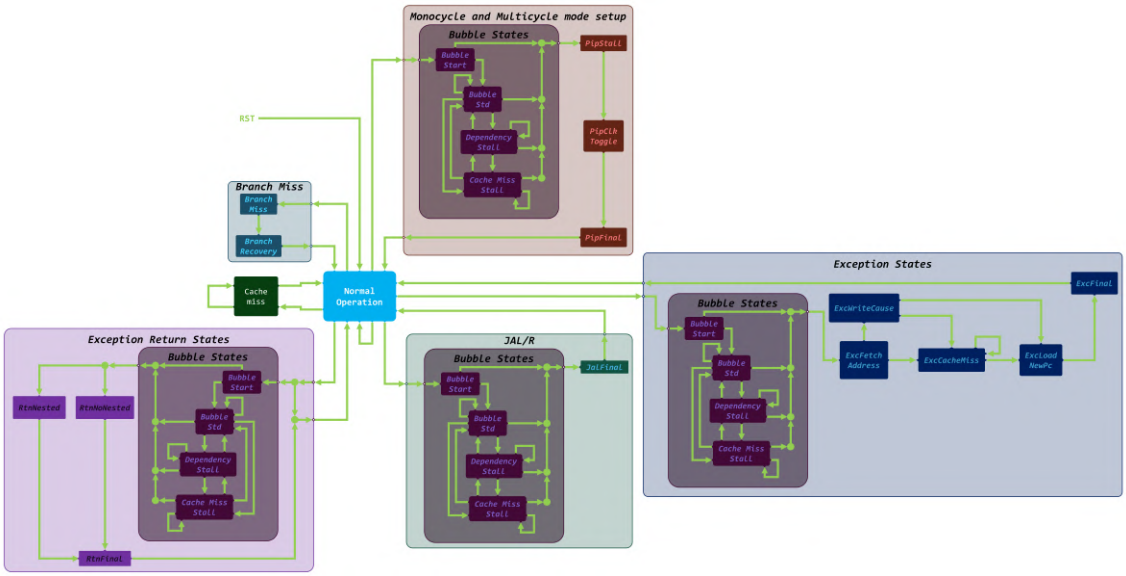


Figure C-1: Complete control unit's finite state machine.

is needed to completely empty them. The other possible eventuality is that one operation causes a cache miss, in which case the processor must be stopped, including the finite state machine. A branch misprediction is not considered, as that will only cause a change in the PC, which is stopped anyways, and it would eliminate some of the instructions that are being executed, leaving everything else identical. Once the machine counts to five, and there are no more eventualities, then it goes to the next state, this is the base condition to leave this subgroup of states and is possible to do so from three different states, as long as the conditions are met.

The exception states are the ones responsible for managing the exceptions and interruptions. Their process begins by emptying the core's stages, using the previously described bubble states. Once all the current instructions are executed, the machine evaluates the exception to determine the cause, and starts fetching the memory address in which said exception begins its subroutine. To do so the control unit issues a signal to the ALU to add the MTVEC to the cause ID. Each case is identified by a unique code, that is used, not only to find its subroutine in memory, but also to establish its priority and in the specific case of the error exception, to store the faulty or unknown instruction to MTVAL. Once the control unit gets the memory address of the subroutine it loads it to the PC to begin the execution of that piece of code. These states also comprise the possibility of a cache miss in the middle of the fetching process for the subroutine address. Lastly the machine issues the acknowledge signal to the peripheral and sets the new nesting value. This last action is done despite the nesting feature being enabled or not, because at least single nesting level is needed to return from exceptions. This whole process is shown in Figure **C-3**.

Once an exception is concluded, the program must return to where it was before the inter-

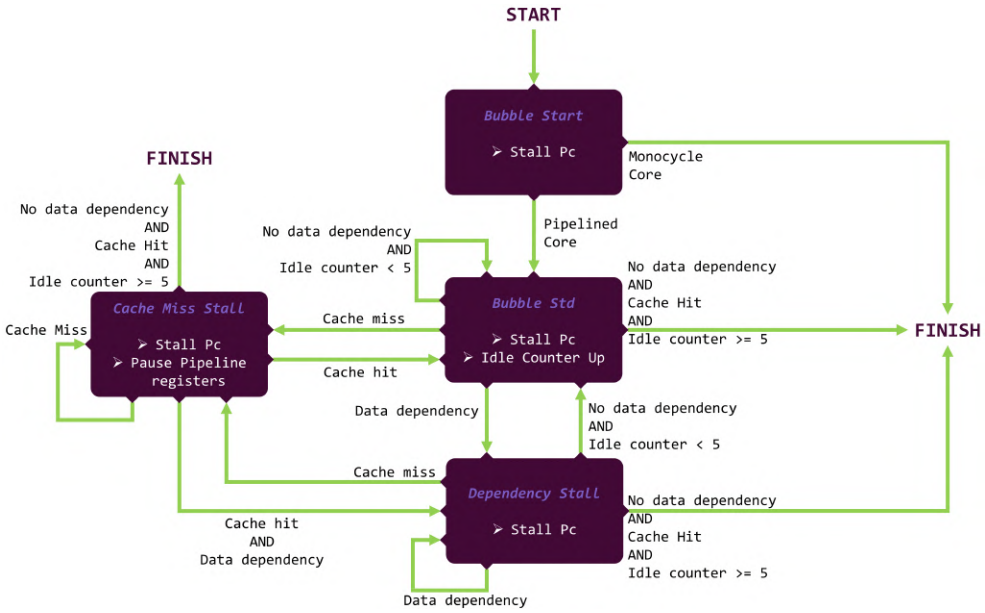


Figure C-2: Bubble states.

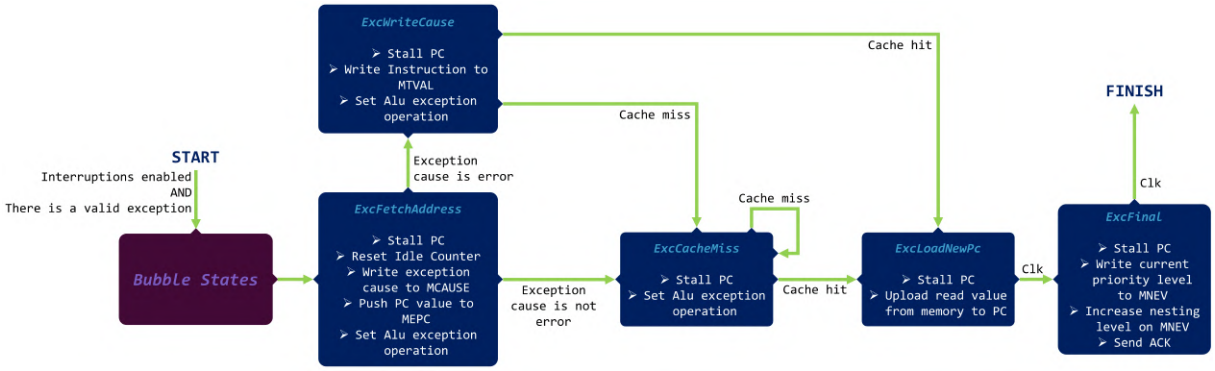


Figure C-3: Exception states.

ruption, to do so, the instructions URET, SRET and MRET are available, and all three are seen as the same by the core, and trigger the return states shown in Figure **C-4**. These states began by emptying the core's stages, but once that process is completed, there are two different states that could be executed depending on the nesting level of the core. If the core was just executing a simple exception that was not nested, then the "RtnNoNested" state is executed, which will restore the program counter and set the nesting level back to zero. However, if the core was executing a nested exception, then the nesting level is lowered by one and the program counter is popped from the MEPC registers. Subsequently in both cases the priority level is also popped in the MNEV CSR. These states also consider the possibility that the programmer just used the return instruction while no exception is being executed, in such case the machine does not execute any state and keeps its previous state,

"Normal operation", as the return instruction was not valid, This validation is done using the nesting level.

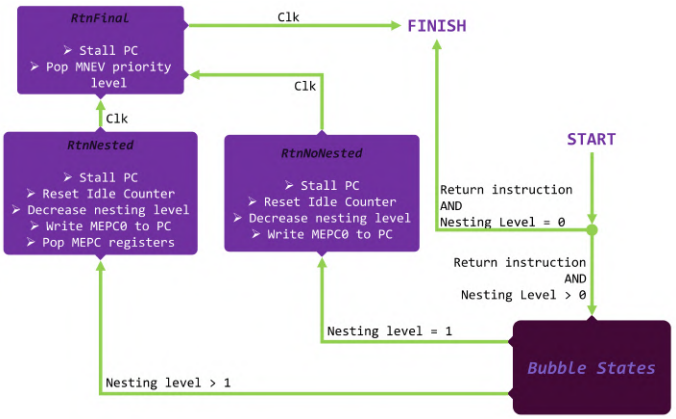


Figure C-4: Return states.

The only others instructions apart from return that need the finite state machine to complete their execution are JAL and JALR. Both are jumps that do not use the BPU, as there is no doubt that the PC will change, however while the new PC is being calculated, the core is stalled. This waiting is due to the possibility that any other instruction that came previously than the jump could change the data used to calculate the final address of this jump. Once all other instructions are executed the machine ensures that the new address gets to the PC from the ALU. This group of states are rather simple but necessary to avoid any mistake in the instruction execution.

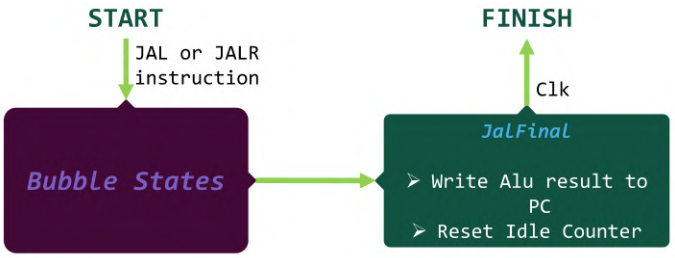


Figure C-5: Jal/r states.

One of the most important groups of states are the Monocycle and Pipelined mode setup states, shown in Figure **C-6**. These states are used for one of the main features of this core, the possibility to change from monocycle mode to pipelined mode. To do so, once again, the pipeline stages must be empty, otherwise when going from pipelined mode to monocycle mode the instructions that are present in each stage would be ignored as long as the core is in that mode, and only after another mode change they would be executed, potentially several cycles after the moment when they were supposed to be executed, breaking the code

flow. Once the stages are empty a stall cycle is given to the core, to then change the pipeline mode by writing the necessary data in the MPS and setting the clock configuration, this second action does not have any effect as there is only one clock frequency implemented in the current version. Finally another stall is given to the core to then proceed to the "Normal operation" state.

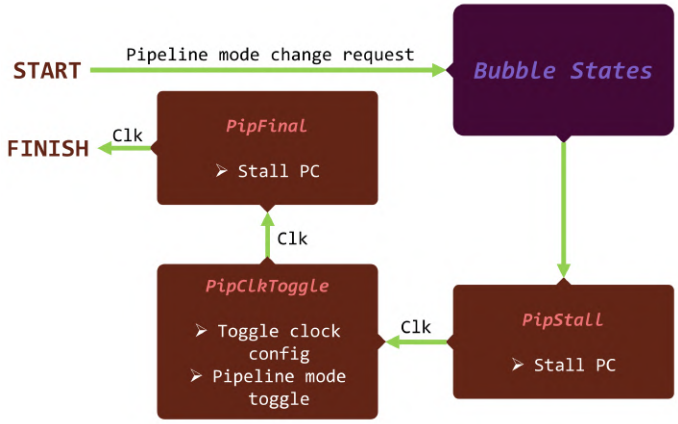


Figure C-6: Monocycle and Pipelined mode setup states.

The last aggrupation is also the only one that do not require to use the bubble states to empty the pipeline stages, as these states are used to synchronize the core after a branch fail, giving time to erase the instructions that must not be executed and to start fetching the correct instructions just after the first pipeline register is cleared. There are only two states in this group, as seen in Figure **C-7** and their actions are simple but effective to let the core run smoothly after a misprediction by the BPU.

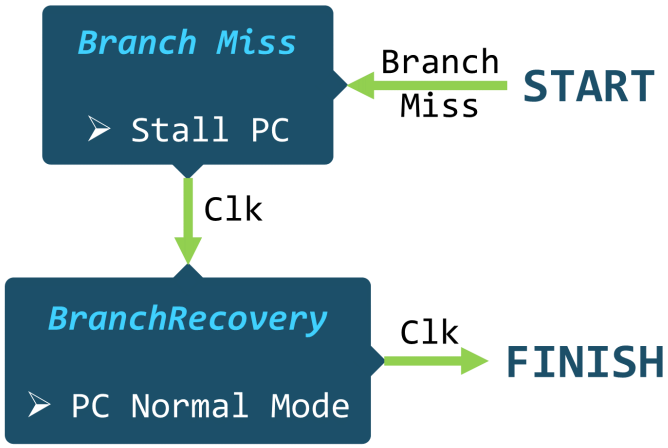


Figure C-7: Branch miss states.

Appendix D

Appendix: Shifter testbenches.

The RV32IM standard defines three possible shifts in the supported instructions, one to the left and two to the right, each one was tested, and the results are found in this section.

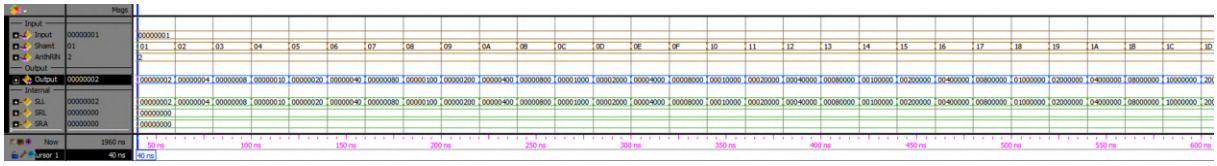


Figure D-1: Shift Left Logical testbench.

The left shift is always a logical shift, meaning that the bits are just moved from their original position to the new one, which is "Shamt" times to the left, and the spaces left behind are filled with zeros. Figure **D-1** shows the test performed to this instruction in which most "shamt" values were tested, assuming an input of 1, and the results were satisfactory, as the bit were moved the required number of positions to the left.

Figure D-2: Shift Right Arithmetical testbenchs.

The first right shift is the arithmetical one, which is characterized by the way it fills the spaces left behind, using the most significant bit of the input number. Figure **D-2** shows this operation's testbench, and as seen the input is 0x80000000, which is just the thirty-second bit set to one and the rest to zero, and in every possible shift done, the spaces to the left of the final position of this bit set to one, are also set to one. Is important to consider that if a number does not have the most significant bit set to one, then this shift will fill the

spaces with zeros. Thus, shift is also known as signed shift, as it conserves the number sign in its most significant bit.

Figure D-3: Shift Right Logical testbench.

The last shift is also a logical, but this time to the right. It works identically to the shift left logical, and in this case the most important feature is that the spaces left behind are always filled with zeros, which is shown in Figure **D-3** for all the "shamt" values that were tested.

1st Workshop:
ARK-I Basic usage

The ARK-I processor is a dual core soft processor described in VHDL in Quartus prime. The processor itself features the RISC-V instruction set, specifically including the I and M modules of it.

To compile a code and run it in the ARK-I processor there are some steps that must be followed:

1. The desired code must be written in assembler language, to ease the process the core comes with a file to import the language and some processor-specific characteristics to notepad ++[1]. The main feature of this code is the usage of different segments, indicated by the character "&", each segment of code has, at least, sixteen different instructions and it is recommended to associate each instruction with its corresponding memory address, as shown in Figure 1.

```
& StartCore0
@ Pc initial value = 0d000 = 0x00
JAL     R10       176 @ Jump to MainCore1                                | Pc =   0 =000
```

Figure 1

The code can have as many segments as needed, but there should be at least three of them present at all times. The first is the code segment with the initial instructions of the first core, it goes from the memory address zero to the memory address fifteen, as sixteen assembler instructions are not enough to do much, the idea is to use this space to include a jump instruction, such as JAL or JALR, or even a branch instruction to another segment of the code. Figure 1 is an example of such.

```
& StartCore1
@ Pc initial value = 0d016 = 0x10
LUI     R28    524288 @ R31 <= 0008 0000 << 12 = 8000 0000          | Pc =  16 =010
NoOp                  @                                             | Pc =  17 =011
NoOp                  @                                             | Pc =  18 =012
NoOp                  @                                             | Pc =  19 =013
CSRRW   R29 R28 4090  @ Toggle core to act as a single cycle core   | Pc =  20 =014
JAL     R10         2 @                                             | Pc =  21 =015
NoOp                  @                                             | Pc =  22 =016
NoOp                  @                                             | Pc =  23 =017
NoOp                  @                                             | Pc =  24 =018
NoOp                  @                                             | Pc =  25 =019
JALR    R00 R00   464 @ Jump to SecondCore                          | Pc =  26 =01A
```

Figure 2

The second segment that is always required to be present in the code is the one associated with the second core, this must always start at the memory address number sixteen, but it could be theoretically extended as much as needed. Figure 2 shows an example of such segment, and in this case, it also shows the necessary instructions to toggle the core's mode, configuring it as a pipelines mode by editing the control and status register (CSR) number 4090, which is the machine pipeline status (MPS) register.

[1] That file is found in "Codes/ArkI.xml"

The third necessary segment of code is the error handling subroutines, that triggers if the processor tries to execute an unknown or faulty instruction. This segment can be found in any memory address, but once set in a specific location, that same address mut be set to the memory in a location marked by the machine trap vector (MTVAL), a CSR with the address 835. This CSR has the default value of 0, as both cores expect to find the error subroutine at that location by default. In Figure 3 is an example of this subroutine. It is important to consider that all subroutines must end with a return instruction, such as MRET, URET or SRET, and that the three are identical for the processor's execution.

```
& ErrorRoutine
@ Pc initial value = 0d032 = 0x20
LUI    R01        0 @ R01 <= 0000 0000 << 12 = 0000 0000          | Pc =  32 =021
LUI    R31        0 @ R31 <= 0000 0000 << 12 = 0000 0000          | Pc =  33 =022
LUI    R15   493440 @ R15 <= 0007 8780 << 12 = 7878 0000          | Pc =  34 =023
JALR   R00 R00  448 @                                            | Pc =  35 =024
MRET               @ Return from exception                       | Pc =  36 =025
```

Figure 3

The desired code should also have the other eight interruptions subroutines, associated to each one of the interruptions sources. These subroutines do not have any default location, so it is necessary to add that location to the program at a later stage.

The example code used in this workshop is found in the processor's files in "Codes\TestbenchCode.asm".

2. Once the code is completed the second step is to compile and assemble it. To do so, in "Codes\Assembler" there is an executable file, which is the assembler program for this processor, to use it the desired code must be pasted into the "AsmCode.asm" file found in that directory.

   The program begins by showing a menu in which is possible to do a step-by-step compilation, or just execute all steps at once. There is also an option to configure some features and the exit option, as seen in Figure 4.

```
Assembler program for the ARK I Processor

Select One of the following options:

                                    Status
1. Begin compilation      << Compilation   successful >>
2. Begin Assembly         <<   Assembly    Complete   >>
3. Export Results         <<     Export  Complete     >>
4. Do all of the above
5. Configuration
6. Exit
```

Figure 4

The configuration option, shown in Figure 5, lets the user decide the number of words, or instruction per cache slice, the maximum number of instructions in the defined memory and toggle some extra information regarding the compilation process. The default value for the cache memory slice size and the memory size is set to match the default values found in the current implementation of the ARK-I processor.

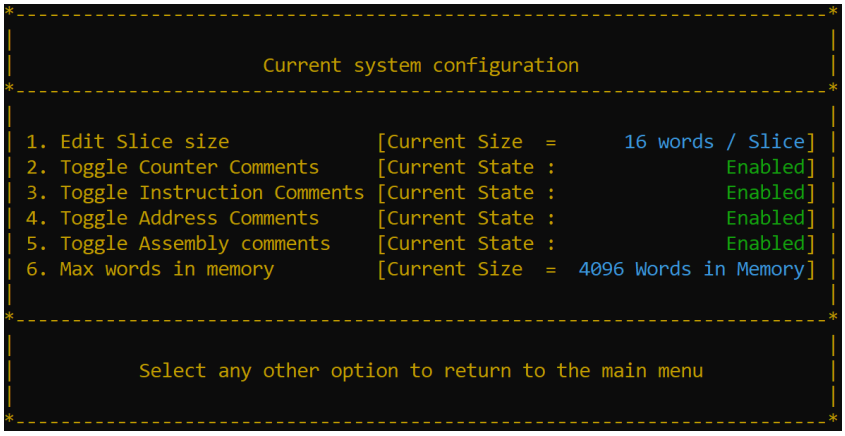


Figure 5

The beginning of the compilation shows, by default, the number of code lines and addresses detected, followed by the entire code. The assembler ignores the comments, but it associates the given instructions to each segment and checks that each instruction is spelled correctly, ignoring the differences between uppercase and lowercase letters and multiple consecutive spaces and tabulator characters. The program also checks if each instruction's arguments are present and within the expected values. If any unknown instruction is found, or if any argument is incorrect, the program will mark them, in red and then continue the compilation process. In such case, at the end of this first step, each line that contains an error will be summarized and the compiler will exit to the main menu. Figure 6 shows, at the left side a code that is correct, and at the right side, the same code but with some minor mistakes. Each mistake is not only marked but as seen it also explains briefly what was the found mistake, such as, for example, invalid instruction, invalid argument, or missing argument.

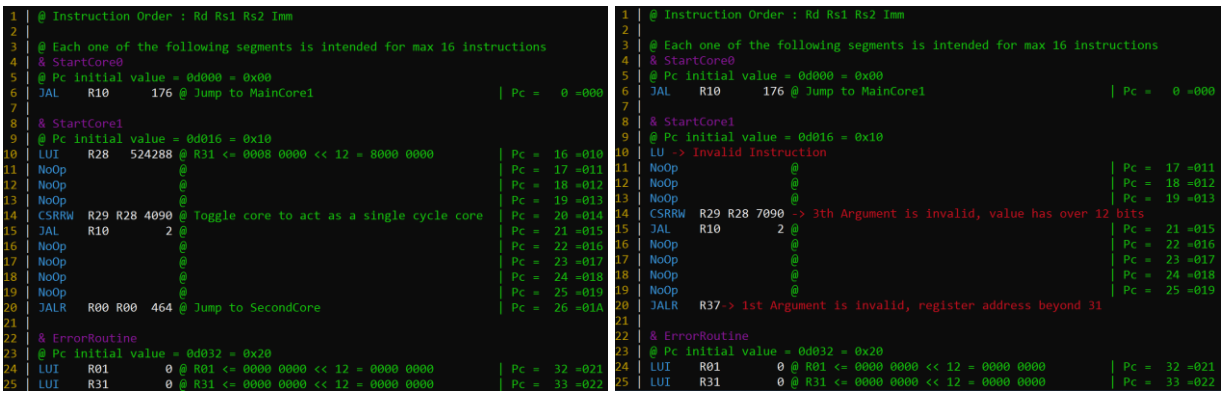
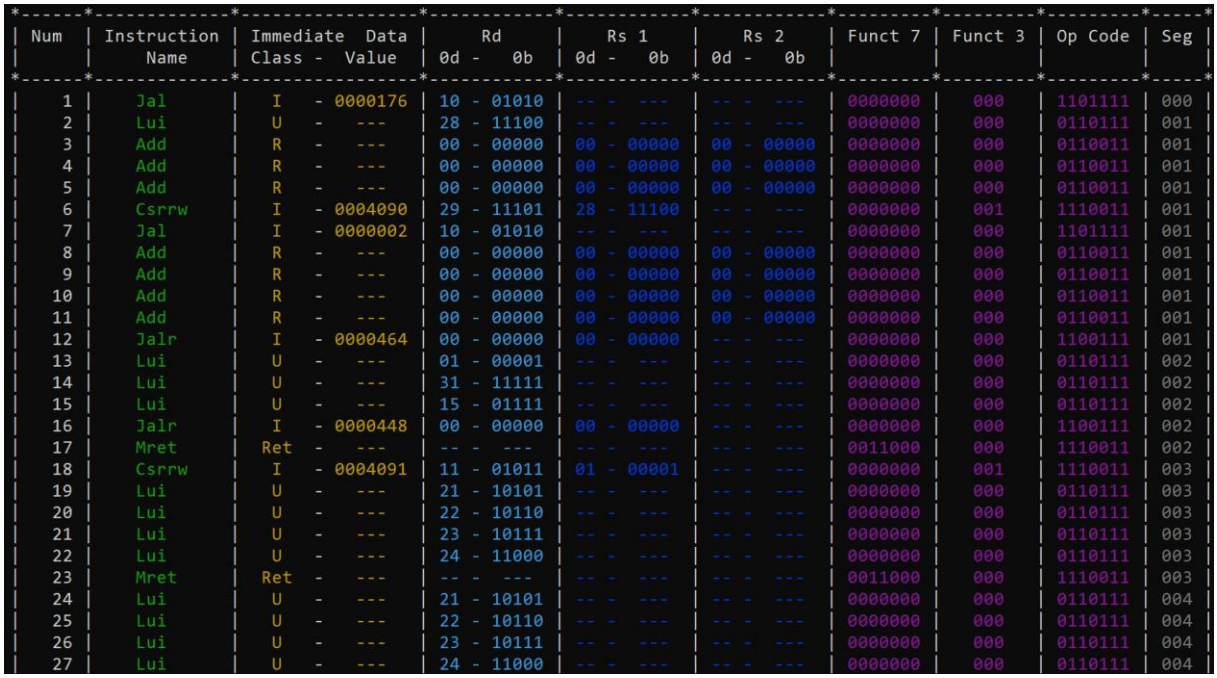


Figure 6

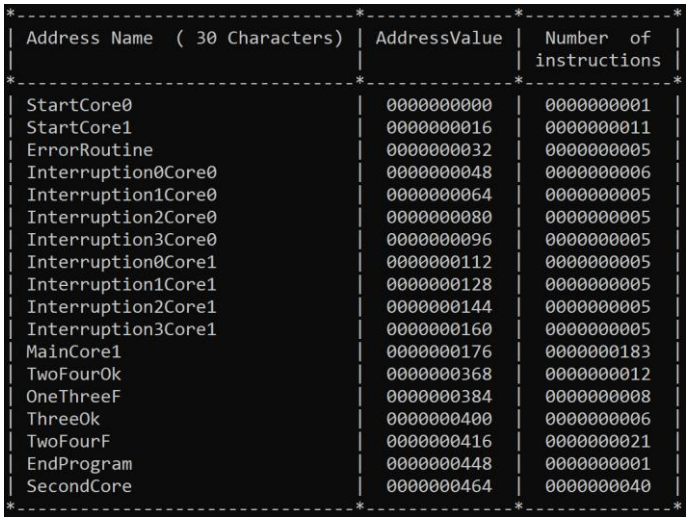
After the code is read correctly, the assembler shows the instructions in a table, in which each instruction is related to its immediate type and data, but also showing its register write address (Rd), both of its register read addresses (Rs1 and Rs2), their respective operation code and Funct3 and Func7 codes, and finally the segment associated with each instruction. Most of these values are shown in decimal and binary, with the decimal being indicated by the "0d" on top of a column and the binary by a "0b". Figure 7 shows the example code in this table, and as seen some instructions have dashes in the Rs1 and Rs2 fields, and some others in the immediate data fields. This happens as not all instructions requires all these arguments, and the system recognizes it and empty them as necessary.



Num	Instruction Name	Immediate Data Class - Value	Rd 0d - 0b	Rs 1 0d - 0b	Rs 2 0d - 0b	Funct 7	Funct 3	Op Code	Seg
1	Jal	I - 0000176	10 - 01010	-- - ---	-- - ---	0000000	000	1101111	000
2	Lui	U - ---	28 - 11100	-- - ---	-- - ---	0000000	000	0110111	001
3	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
4	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
5	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
6	Csrrw	I - 0004090	29 - 11101	28 - 11100	-- - ---	0000000	001	1110011	001
7	Jal	I - 0000002	10 - 01010	-- - ---	-- - ---	0000000	000	1101111	001
8	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
9	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
10	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
11	Add	R - ---	00 - 00000	00 - 00000	00 - 00000	0000000	000	0110011	001
12	Jalr	I - 0000464	00 - 00000	00 - 00000	-- - ---	0000000	000	1100111	001
13	Lui	U - ---	01 - 00001	-- - ---	-- - ---	0000000	000	0110111	002
14	Lui	U - ---	31 - 11111	-- - ---	-- - ---	0000000	000	0110111	002
15	Lui	U - ---	15 - 01111	-- - ---	-- - ---	0000000	000	0110111	002
16	Jalr	I - 0000448	00 - 00000	00 - 00000	-- - ---	0000000	000	1100111	002
17	Mret	Ret - ---	-- - ---	-- - ---	-- - ---	0011000	000	1110011	002
18	Csrrw	I - 0004091	11 - 01011	01 - 00001	-- - ---	0000000	001	1110011	003
19	Lui	U - ---	21 - 10101	-- - ---	-- - ---	0000000	000	0110111	003
20	Lui	U - ---	22 - 10110	-- - ---	-- - ---	0000000	000	0110111	003
21	Lui	U - ---	23 - 10111	-- - ---	-- - ---	0000000	000	0110111	003
22	Lui	U - ---	24 - 11000	-- - ---	-- - ---	0000000	000	0110111	003
23	Mret	Ret - ---	-- - ---	-- - ---	-- - ---	0011000	000	1110011	003
24	Lui	U - ---	21 - 10101	-- - ---	-- - ---	0000000	000	0110111	004
25	Lui	U - ---	22 - 10110	-- - ---	-- - ---	0000000	000	0110111	004
26	Lui	U - ---	23 - 10111	-- - ---	-- - ---	0000000	000	0110111	004
27	Lui	U - ---	24 - 11000	-- - ---	-- - ---	0000000	000	0110111	004

Figure 7

The last step of the compilation process just shows a table with all the defined segments, their starting address and the number of instructions found in each one. This table allows for sections names of up to thirty characters, as seen in Figure 8.



Address Name (30 Characters)	AddressValue	Number of instructions
StartCore0	0000000000	0000000001
StartCore1	0000000016	0000000011
ErrorRoutine	0000000032	0000000005
Interruption0Core0	0000000048	0000000006
Interruption1Core0	0000000064	0000000005
Interruption2Core0	0000000080	0000000005
Interruption3Core0	0000000096	0000000005
Interruption0Core1	0000000112	0000000005
Interruption1Core1	0000000128	0000000005
Interruption2Core1	0000000144	0000000005
Interruption3Core1	0000000160	0000000005
MainCore1	0000000176	0000000183
TwoFourOk	0000000368	0000000012
OneThreeF	0000000384	0000000008
ThreeOk	0000000400	0000000006
TwoFourF	0000000416	0000000021
EndProgram	0000000448	0000000001
SecondCore	0000000464	0000000040

Figure 8

After the compilation process is completed, the program returns to the main menu, and there the message next to the first option should now be green, and it should indicate that the compilation has been completed successfully. Then the compiled code must be assembled or translated into a machine code. The second option of the main menu allows to do this process. In it another table is shown, once again with all the defined instructions, but this time it also shows the binary translation of each instruction, highlighting each argument in a color to show where each part of

the instruction is being implemented into the binary word. The final column shows the instruction address in decimal. Figure 9 shows this table for the example code.

Instruction		Binary	Address
Jal R10 R99 0000176	I	000010110000 00000 000 01010 110111	0000000000
Lui R28 0524288	U	10000000000000000000 11100 011011	0000000016
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000017
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000018
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000019
Csrrw R29 R28 0004090	I	111111111010 11100 001 11101 111001	0000000020
Jal R10 R99 0000002	I	000000000010 00000 000 01010 110111	0000000021
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000022
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000023
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000024
Add R00 R00 R00	R	0000000 00000 00000 000 00000 011001	0000000025
Jalr R00 R00 0000464	I	000111010000 00000 000 00000 110011	0000000026
Lui R01 0000000	U	00000000000000000000 00001 011011	0000000032
Lui R31 0000000	U	00000000000000000000 11111 011011	0000000033
Lui R15 0493440	U	01111000011110000000 01111 011011	0000000034
Jalr R00 R00 0000448	I	000111000000 00000 000 00000 110011	0000000035
Mret	Ret	0011000 00010 00000 000 00000 111001	0000000036
Csrrw R11 R01 0004091	I	111111111011 00001 001 01011 111001	0000000048
Lui R21 0065537	U	00010000000000000001 10101 011011	0000000049
Lui R22 0065537	U	00010000000000000001 10110 011011	0000000050
Lui R23 0065537	U	00010000000000000001 10111 011011	0000000051
Lui R24 0065537	U	00010000000000000001 11000 011011	0000000052
Mret	Ret	0011000 00010 00000 000 00000 111001	0000000053
Lui R21 0131074	U	00100000000000000010 10101 011011	0000000064
Lui R22 0131074	U	00100000000000000010 10110 011011	0000000065

Figure 9

At this point the code has been compiled and assembled, however the idea is to export it to load it into the processor. To do so the third option in the menu now allows to select one of three possible export options: CSV, VHDL and MIF. The first option creates a file with all the defined addresses in decimal and in front of them the instructions in binary. The second type of output generates a VHDL entity, which contains a multiplexer relating all the addresses and instructions, and acts as a read only memory. Finally, the MIF file is a memory initialization file, that describes the complete contents of a memory. That last option must be chosen to generate the required file for the processor to execute the code. When that option is selected the result is shown in the console, and the corresponding file is created at “Codes\Assembler\BinaryOuput.mif”, as shown in Figure 10.

That file can be uploaded to the processor and executed, however, sometimes there are some raw data that must be present for the processor to function as expected, for example somewhere in the memory there must be information that indicates what address is associated with each one of the exceptions and interruptions, or perhaps the code just requires a set of data to work correctly.

That information must be added to the memory file directly, in the specific case of the example, the values for each exception are found on the addresses one to five, and the addresses six to eight just have some data that is used in the execution of this code. Figure 11 shows how to add that information to the memory files without damaging the program. First the area selected was empty, Figure 10 shows that the addresses from one to fifteen were set to zero, thus adding new information to this specific area of the code does not affect the program itself. Then the bracket

[1..15] must be changed into the bracket [9..15] and the previous addresses must be filled with the desired information.

```
-- ArkI Assembler ganarated Memory Initialization File (.mif)

WIDTH = 32;
DEPTH = 4096;

ADDRESS_RADIX = UNS;
DATA_RADIX = BIN;

CONTENT BEGIN
0 : 00001011000000000000010101101111;
[1..15] : 00000000000000000000000000000000;
16 : 10000000000000000000111000110111;
17 : 00000000000000000000000000110011;
18 : 00000000000000000000000000110011;
19 : 00000000000000000000000000110011;
20 : 11111111101011100001111011110011;
21 : 00000000001000000000010101101111;
22 : 00000000000000000000000000110011;
23 : 00000000000000000000000000110011;
24 : 00000000000000000000000000110011;
25 : 00000000000000000000000000110011;
26 : 00011101000000000000000001100111;
[27..31] : 00000000000000000000000000000000;
32 : 00000000000000000000000010110111;
33 : 00000000000000000000111110110111;
34 : 01111000011110000000011110110111;
35 : 00011100000000000000000001100111;
36 : 00110000001000000000000001110011;
```

Figure 10

```
CONTENT BEGIN
0 : 00001011000000000000010101101111;
1 : 00000000000000000000000000100000;
2 : 00000000000000000000000000110000;
3 : 00000000000000000000000001000000;
4 : 00000000000000000000000001010000;
5 : 00000000000000000000000001100000;
6 : 00000000000000000000000010101010;
7 : 00000000000000001010101010101010;
8 : 10101010101010101010101010101010;
[9..15] : 00000000000000000000000000000000;
16 : 10000000000000000000111000110111;
17 : 00000000000000000000000000110011;
18 : 00000000000000000000000000110011;
19 : 00000000000000000000000000110011;
20 : 11111111101011100001111011110011;
21 : 00000000001000000000010101101111;
```

Figure 11

3. Uploading the machine code to the processor is a two-step process, first the mif file must be placed in the directory “VHDL\3-Memory” with a unique name. This same name must be set in the processor’s VHDL files.

This process is done by accessing the file "VHDL\3-Memory\RamComponent\Ram2Port.vhd" and changing the line number 77, shown in Figure 12. The string in that line must point to the code that the processor must run. Afterwards the whole project must run the analysis and synthesis compilation in Quartus.

```
init_file => "./VHDL/3-Memory/TestbenchCode.mif",
```

Figure 12

4. With the code uploaded to the processor, the final step is to simulate the processor's behavior. This is done in Model Sim, a software from altera. Upon opening the program, the first thing to do is to change the actual directory. This is done by clicking in "File" in the upper right part of the window, and then in "change directory", this will create another window in which the ARK-I Main directory must be selected.

   After that, the correct simulation settings must be selected. Every needed simulation has a "do" file that contains all the necessary settings to add the desired signals and set their names and colors. Furthermore, these settings also group some signals together to improve readability, and it also sets the cursors and simulation time. Altogether the "do" files are the easiest way to replicate any simulation and store its parameters.

   For this case in the lower part of the Model Sim window there is a section called the "Transcript", in it the following command must be written to start the simulation:
   - "`do simulations/pipelinedCoreTest.do`"

   This command will compile all the processor files, and then will create a wave window with the desired signals in it, then it will simulate for a certain amount of time and will graph the output.

   Figure 13 shows the resulting simulation. In it the first thing to do is to scroll back to the beginning of the simulation, as the program by default sets the view at the end. The beginning does not show much, as there is a cache miss, since the cache does not have the required instructions, and that will be a common occurrence from now on, as from time to time the core will ask the cache for more instructions and the cache will generate a miss while it fetches said information from memory.

   This testbench code goes through all the available instructions in pipelined mode and uses the second core to execute some of those instructions in monocycle mode. And, as there are many different interesting things that can be seen in this specific simulation, it was impossible to show them all in just one configuration. Thus, there are five different simulation configurations:

   - "`do simulations/MonocycleCoreTest.do`"
   - "`do simulations/PipelinedCoreTest.do`"
   - "`do simulations/CacheMemoryTest.do`"
   - "`do simulations/DataForwardingTest.do`"
   - "`do simulations/ExceptionsTest.do`"

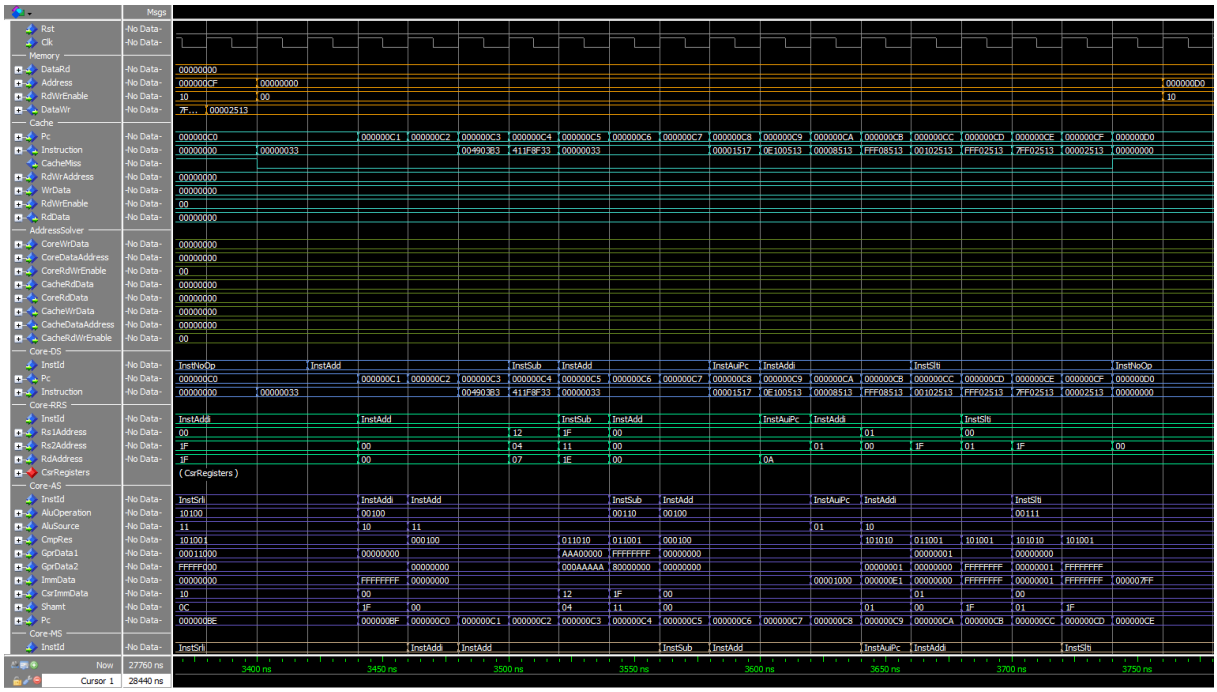


Figure 13

Exercise

Using the code "Codes/Fibonacci.asm", compile it, load it into the ARK-I processor and run the following simulation:

- "`do simulations/ FibonacciTest.do`"

Show the results from the cursor onward, assuming the top part of the simulation corresponds to the first core, running in monocycle mode, and the lower part to the second core, running in pipelined mode, highlight the execution differences for each core. Pay special attention to the instruction propagation through the stages of each core and the execution of branches.

Which core completes the task faster?

What would happen if the core in pipelined mode had a clock frequency five times higher than the core in monocycle mode?

2nd Workshop: ARK-I Interruptions

The ARK-I processor can manage exceptions and interruptions. Both terms refer to the same thing, an occurrence that makes the processor stop its work to execute a different segment of code. The only exception found in this processor is the unknown instruction exception, that is executed when it encounters an instruction whose operation code is not recognized, this usually happens with faulty instructions or instructions that are not supported in the RV32IM standard. An interruption is when a peripheral circuit requires attention, and they depend on the interruption request (IRQ) signals to inform the processor of this. Once the processor begins attending the peripheric, then an acknowledge (ACK) signal is sent, to synchronize both.

Each core of the processor has four different IRQ and ACK signals, meaning that up to eight peripherals can try to interrupt the processor at the same time, however each ARK-I core has a built-in priority system, meaning that in such scenario only one interruption would be executed, and then the others.

To test how the interruptions system works, there is a demonstration code within the ARK-I files, which executes an algorithm to find the factors of a given number. This algorithm is found in "Codes\Factors.asm" and it has the following characteristics:

- Single core usage, the second core is stuck as its first instruction is to jump to the same address.
- Execution in pipelined mode only, as there are no data dependencies this algorithm can run in pipelined mode easily from start to finish.
- Three main segments of code, the first one dedicated to initializing values, the second is the interruption subroutine to read the unsigned number given by the user and store it, and the third is the algorithm which finds the factors of said number.
- The code is a closed loop, meaning that once the algorithm is complete the program begins its execution again. In case that the user does not input a number, the code will wait on the final instructions after the initialization stage.

To simulate this code there is a dedicated version of the ARK-I processor that has the necessary files to create the desired peripheric circuit, and to give the desired values to it. It is the "5. ArkI-Processor - Peripherics".

To input the number which factors will be found, the file "\VHDL\Testbench\2-Core\PeripheralsTestProtocol.vhd" must be edited. On line number 59 the IRQ signal is set, it is important to set a reasonable time, as the code starts with interruptions disabled, meaning that if said time is too early, then the simulation could not get it. It is important to set the IRQ back to zero after a while, preferably after 20ns, as if the signal is left for too long on one, it could trigger multiple interruptions back-to-back. On line 62 the desired number must be set in hexadecimal, this value

could be ser for any period, as the peripheral circuit manages it according to the core's instructions. Figure 1 shows an example of this configuration.

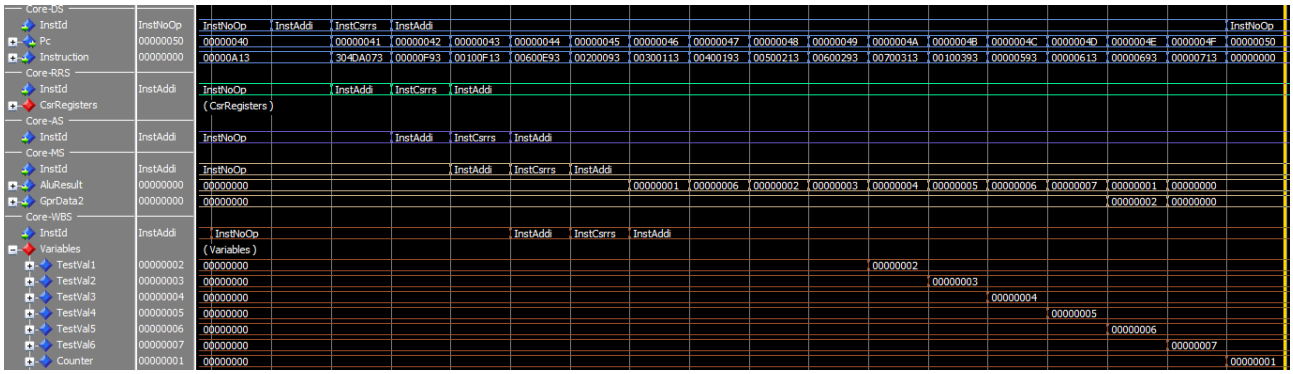
```
59  IRQ         <= (x"1",x"0") AFTER 4540 ns,
60                 (x"0",x"0") AFTER 4560 ns;
61
62  PerRdData <= (x"0000000C",x"0000000C") AFTER  4540 ns,
63                 (x"00000008",x"00000008") AFTER 50000 ns;
```

Figure 1

Once the testbench file is ready, on ModelSim, the simulation must be executed, for that matter, the following command is available which will load the entire simulation.

"`do simulations/factors.do`"

The code starts by jumping to the address 0x40 and it enables interruptions and sets some important values on registers. As this code was made to prevent any kind of data dependency it will calculate six possible factor values at the same time, and those values are stored on the "TestVal" variables, associated with the general-purpose registers (GPR) 1 to 6.



Then the code is left in a cycle waiting for the user to start an exception, which eventually happens as seen in Figure 2. After the interruption is received there are two cache misses, in which the core is getting the subroutine associated with the interruption. That segment of code reads data from the address 0xFFFF0000, which is detected as part of the peripherals mapped data, and therefore the information is read from that source, and then stored into the processor. The code then sets the necessary values on the registers to begin the execution of the algorithm.

The algorithm starts by disabling interruptions and is composed of two cycles, one which increments the counter by one each time, multiplies the counter by the "TestVal" registers, and checks if any of the results is the same as the number whose factors are being searched. If the multiplication result is the same as that number, then the associated "TestVal" is stored in a peripheral register, if not then nothing happens. This first cycle is completed once the counter reaches the value of half the selected number plus six. The second cycle increases each one of the "TestVal" by six. If the last "TestVal" has a higher value than half the selected number plus six, then the code is over, and the core returns to the address 0x40 and disables the execution of the algorithm until another exception is found. Figure 3 shows two cycles of the algorithm, in which two different factors of twelve (0xC) are found: four and three.

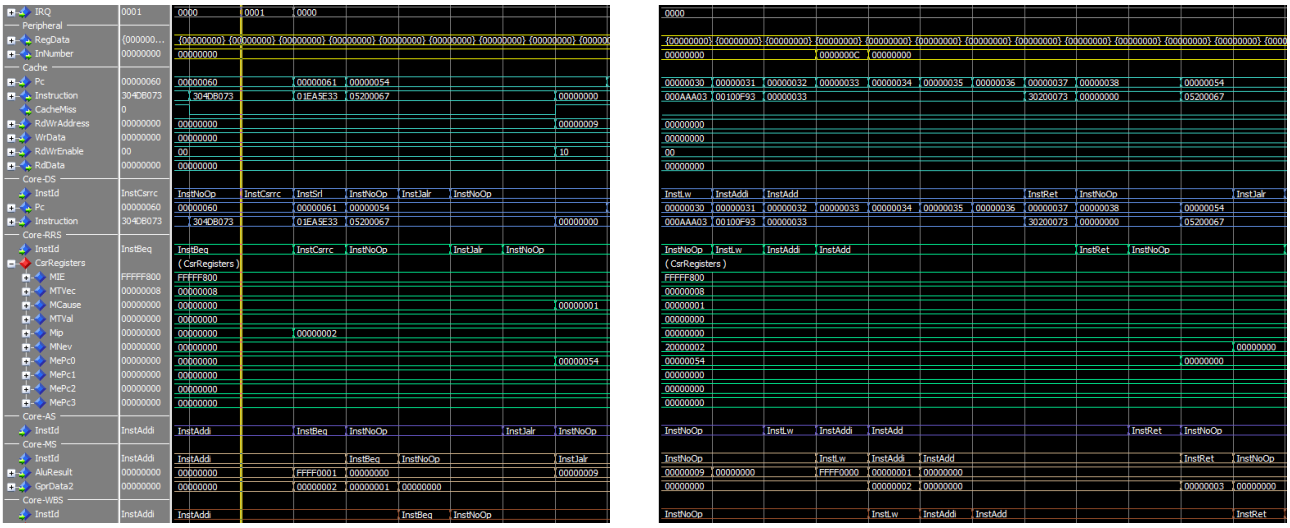


Figure 2

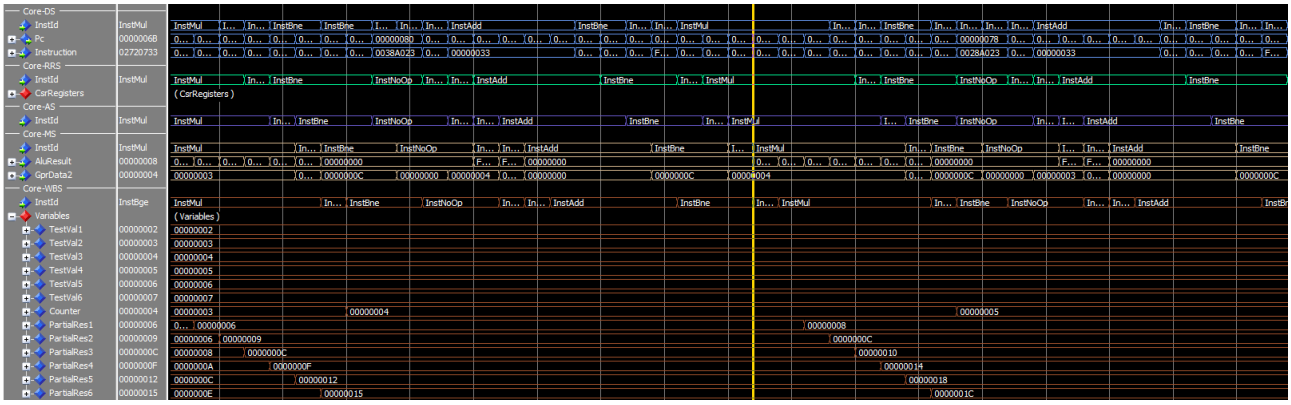


Figure 3

Exercise

1. Using the previously provided algorithm modify the “Factors.mif” file to set a different number to find its factors, a reasonable number is suggested, as the simulation size increases drastically with bogger numbers. Show the results on the peripheral registers (Yellow)

2. Set two different interruptions, the first to find the factors of four, and the second to find the factors of six. What happens if the second interruption is set during the first algorithm execution?

3. Edit the MIF file to set a faulty instruction to the code, at the address 0d01, by setting all values to zero, document the outcome. Does the unknown instruction subroutine trigger? Why?

4. Finally edit the MIF file to set a faulty instruction to the code, at the address 0d68, this time by setting all values to one, document the outcome. Does the unknown instruction subroutine trigger? Why?