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

Ivan Ricardo Diaz Gamarra
Ponticia Universidad Javeriana
Bogota, Colombia
diazivan@javeriana.edu.co

2nd Given Name Surname
Ponticia Universidad Javeriana
Bogota, Colombia
diego-mendez@javeriana.edu.co

3rd Given Name Surname
Ponticia Universidad Javeriana
Bogota, Colombia
egerlein@javeriana.edu.co

***Abstract*—Complex algorithms used by scientists require high-performing processors, which can be implemented in many ways by 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 the pipelined mode and the monocycle mode to leverage the strengths of each. The most important simulations are presented alongside the main features of the implementation.**

***Index Terms*—VHDL, pipeline architecture, monocycle core, multi core, cache memory, RISC-V, open-source**

I. Introduction

RISC-V is an open-source, modern, and versatile instruction set architecture (ISA) used in various types of applications and processor implementations. It was designed to be modular, with each module serving a specific purpose [1]. For instance, the I module, also known as the base integer instruction set, includes all the necessary features for a basic processor to execute a code . It contains instructions to control code flow, perform algorithmic and logic operations, and load and store data to and from memory, and so on. On the other hand, the M module is also useful, as it comprises multiplication and division instructions for integers, which are common operations in many complex algorithms developed by researchers today [2].

The current paper presents the design and implementation of the ARK-I processor, an open-source dual-core RV32IM processor with a variable pipeline architecture and cache memory. This initial version of the processor was created as a simple tool to handle the challenges of running demanding research algorithms and as an educational resource to learn the design and implementation process of a processor with these characteristics. Since it is implemented in a hardware description language, namely VHDL, it allows for edits as needed to achieve specific required performance. It also enables proper simulation to verify that every instruction is working as expected

II. Related Works

As this project aims to complete an open-source processor using a hardware description language as the main implementation method, it is important to review similar cores that have already been implemented and are ready to be used. A noteworthy implementation of a processor is the ORCA [3], an RV32IM processor made in VHDL and Verilog by Kammoh. It features a 4 to 5-stage pipeline, a single core, and can execute matrix operations. However, its documentation is limited to its interface buses, interruptions, and some registers.

Another core is called PicoRV32 [4], made by Yosys. This processor features a Verilog implementation of a variable ISA, allowing the user to select from the RV32E, RV32I, RV32IC, RV32IM, and RV32IMC. It is optimized for size but not for raw performance, has a single core, and some instructions like jumps can take up to 6 clock cycles to complete.

Another completed processor was made by Chen Chen and Xiaoyan Xiang [5]. Their processor has a 12-stage pipeline with out-of-order execution to achieve high performance with a single core. They used the RV64GCV ISA, which allows to use compressed instructions, vector operations, and the standard I, M, A, F, and D extensions. None of the previous processors provide complete documentation on how they were designed or implemented. They only allow minor changes to their architecture, making them poor subjects for people who want to learn the specifics of a processor or want to edit them for their own purposes, such as optimizing certain algorithms.

There are other examples of soft processors made with full documentation. Whilst their main purpose is not to execute complex algorithms with superior performance, it is important to consider the characteristics introduced into these processors. For instance, Abelardo V. [6] made a single-core processor using the RV32I ISA that handles all the basic functions needed. It was made as an educational project and includes interruption management like all the other processors mentioned so far, and the possibility of using the GCC compiler.

Ivan D. [7] also made a processor designed to be used as a drone controller, specifically aiming to execute the PID algorithm. This processor features a single core and no pipeline at all, but the documentation about the architecture is sufficient to understand what is happening in each instruction.

Finally, P. Jamieson [8] created a framework for undergraduates to learn about computer architectures using an RV32I architecture in VHDL. It has the option to implement one or two cores but no pipeline at all. None of these processors report any usage of cache memory to speed up memory operations.

Many features used in the ARK-I processor were inspired by Hennessy and Patterson's textbook [9], which describes pipelined architecture in detail with its many advantages while also addressing solutions to its main problems using branch prediction and data forwarding. The textbook also illustrates the basic requirements for cache memory implementation and some key features of a monocycle core.

III. System Description

None of the previously mentioned processors have a key characteristic that is implemented in the ARK-I processor: its core-independent ability to execute instructions in pipelined mode or in monocycle mode, along with the capability to switch between both modes while executing code. This characteristic, called variable pipelined architecture, allows the core to extract the best of each mode depending on the program structure, which could potentially result in better performance than using one mode or the other independently.

A. A. Top-Level Overview

This processor is composed of two identical but independent RV32IM cores with a variable pipeline. Each core has its own cache memory connected to the cores by a block called the address resolver, which is used to separate the addressing of memory elements and peripherals. These peripherals are memory-mapped to up to 65535 addresses, but there are only four interruption request lines per core. Each cache memory is also connected to a shared on-chip memory. This architecture allows the ARK-I to run two independent programs simultaneously and to easily access their own peripherals.

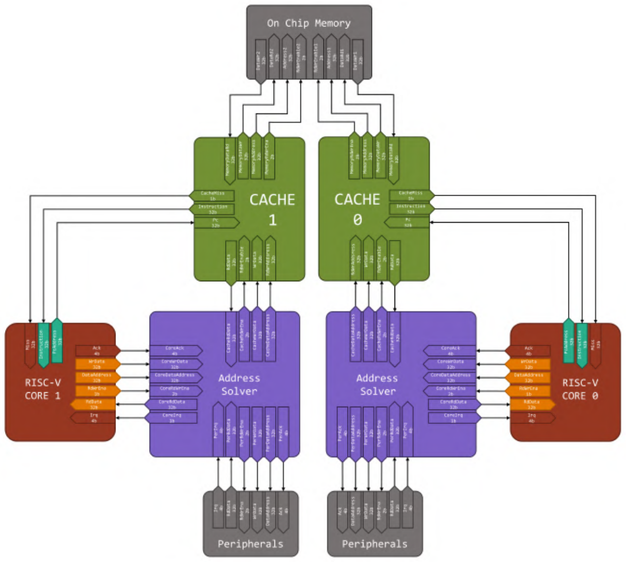

Fig. 1. Multi core block diagram.

Any kind of memory could be used to store the program. For this specific case, the selected memory was the dual-port on-chip RAM memory from the Quartus IP library. This choice allows for easy definition of contents and can be uploaded to the core along with the rest of the implementation. Because the memory has two separate read and write ports, it is possible to feed both cache memories simultaneously allowing seamless downloading or uploading of data to and from the cache, as each core has its own communication bus to the memory.

B. Single core characteristics

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.

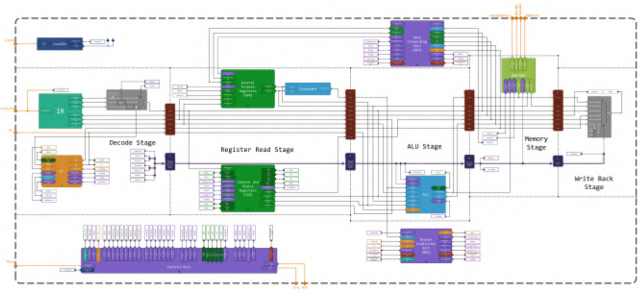

Fig. 2. Multi core block diagram.

Figure 2 shows the block diagram of a single ARK-I core. There are two main areas: the one in the middle surrounded by the dotted line is the variable pipeline, which is divided into five stages. 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.

The variable pipelining architecture allows, as previously stated, a pipeline mode or a monocycle mode. The first allows the execution of up to five instructions at a time, each at a different stage of execution, therefore, at a different stage of the pipeline system. But, in this mode, 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. These are designed to provide reasonable solutions to these problems, either by trying to bring the necessary data one stage ahead in time, by controlling the correct program flow when needed, or by 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.

The monocycle mode, on the other hand, allows an easy way to execute instructions in one clock cycle, avoiding all the issues associated with data dependencies and branch misprediction, but losing out when an interruption or cache miss occurs.

To allow both modes to coexist in the core, between each stage of the variable pipelined architecture, there are blocks called pipeline registers. These blocks allow information to be retained from one stage to the next in pipelined mode, but in monocycle mode, they are configured to let the data go through them immediately. This allows the use of the same resources for both modes.

C. Control Unit

This is one of the most important blocks of the ARK-I core, as it 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. The finite-state machine oversees particular cases such as interruptions, exceptions, or issues like bubbles and stalls when needed. These - could be caused by cache misses, data dependencies, or branch mispredictions, including the JAL and JALR instructions in pipelined mode. The microprogrammed control unit manages all other implemented instructions in the RV32IM instruction set architecture (ISA).

It's important to clarify that even though the selected ISA is composed of sixty different instructions, six of them were not implemented as they were used in specific cases that were considered unnecessary for the current implementation. These instructions were: FENCE, FENCE.I, ECALL, EBREAK, WFI, and FENCE.VM. Additionally, the four division and remainder instructions have the necessary signals and operation codes, but the associated integer division module was not implemented. The other fifty instructions were implemented and are completely functional.

D. Data Forwarding Unit (DFU)

This block aids the control unit by overseeing the namespaces that are being used by the currently executed instructions in pipelined mode. It detects a data dependency each time an instruction's register read address is the same as another instruction's register write address in a later stage. In those cases, the DFU blocks the core to avoid receiving any more instructions and executing the dependent operation until the necessary instruction reaches the memory read stage, where the required data is forwarded to the registers to be written, and then normal core operation resumes.

The selection of the memory read stage as the place from which data can be forwarded was made because most instructions in the RV32IM ISA depend on the ALU to be completed, and in the memory stage, those results are already stable to be written.

E. Branch Prediction Unit (BPU)

This block also aids the control unit, this time by managing the possible outcomes of a branch instruction in pipelined mode. Its function, as the name suggests, is to predict the outcome of a branch instruction before its execution is completed. To do so, the BPU is equipped with a finite state machine (FSM) that decides whether to commit to jumping to a new memory address, assuming that the branch will be successful, or to keep the current next program counter address and assume that the branch result will be unsatisfactory. The FSM evaluates its choice every time a branch instruction is completed, and if its prediction fails two times in a row, it will change its prediction from one choice to the other.

This block also stores the memory address that was not used. If the prediction fails, the BPU must erase whatever progress was made in the pipeline stages and restore the unused memory address to execute the correct instructions from scratch.

F. Cache memory

Each core has its own fully associative cache memory, and both caches are identical. These caches are not meant to run the same code, as they always assume that the main memory, or each one of them, has the most recent version of the data. This means that if one core modifies the information at a given memory address, that data will not be available to the other core until the first cache writes that information to the main memory.

The cache memory is composed of two different segments: one is dedicated to storing instructions, known as the instruction cache; and the other is for storing data, known as the data cache. The only difference is that the data cache is designed to be read and written, while the instruction cache can only be read. Both caches are made up of sixteen slices, each independent from one another. Inside each slice, there is space for sixteen 32-bit words where the information is stored. To know which address is associated with each slice, there is another field that stores this information alongside a bit to indicate whether that slice has valid data or not. Lastly, each slice has a counter whose value goes up when it is being used and goes down when it is not. This counter is used to compare slices and select the least used one to be replaced.

To manage the replacement process, the Address Replacement Control Unit (ARCU) was created. It checks if there is a miss in either of the two caches and, using that information, 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 replacing the instructions and then the data. The data replacement process considers that if a slice without an invalid address is replaced, the data must be written to memory before writing the new information.

For this project an assembler code was also made, which considered the on-chip memory size and number of words per slice to organize the code in memory according to the given characteristics. That program is also used to detect errors on the incoming ASM code.

IV. Results

Each one of the implemented blocks was tested, but a general testbench was conducted on the processor using a code that executed all instructions at least once. In this test, one core was set in pipelined mode, while the other was configured in monocycle mode. Each one ran a different program, and in the end, they both arrived at an instruction that makes them jump to the same instruction repeatedly to finish the code and prevent the processor from executing more instructions. However, even in that state, the ARK-I can still receive exceptions or interruptions, and as such, that feature is also tested.

A. Pipelined mode

The pipelined mode executes up to five instructions at a time, limited only by the number of stages implemented. Figure 3 shows this process by following a single “CSRRS” instruction through the different stages, even if in some of them, as happens in this example, the instruction does nothing, and just waits for the next cycle to continue its execution. In the shown simulation, after the highlighted instruction is received, more instructions enter the pipeline stages, allowing for more instructions per second to be completed. But after a while the decode stage sets its current instruction to “No Operation” This happens as a consequence a cache miss, which stops the core and forces the previously mentioned operation for a while, until the situation is solved. Then the correct instruction is received and the core resumes its procedure. This behavior is seen at the start of the simulation, in which the “No Operation” instruction is replaced by the first CSRRW seen.

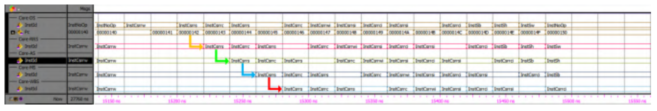

Fig. 3. Multi core block diagram.

B. Monocycle mode

The monocycle mode executes the instructions using the same resources as the pipelined mode; therefore, their functioning was flawless. However, as the cores are configured by default to be in pipelined mode, it is necessary to first set the desired operation mode. Figure 4 shows this process, which is done by the control unit’s FSM and depends completely on the machine pipeline status (MPS) register, one of the custom-designed control and status registers (CSRs) in this implementation. The most significant bit of MPS is used to toggle between modes, and in this example, the CSRRW instruction is used to write to that bit and begin the mode transition. But before that, the control unit’s FSM must be in the “Normal Operation” state, which eventually will be reached. At that point, the mode change begins, starting by allowing the current instructions in the pipelined stages to be executed while preventing new ones from being received. Then, over three states, the pipeline mode is changed, and from 1400 ns onwards, it is observed that the instructions do not propagate through the stages but rather across them all in a single clock cycle. The process to switch from monocycle mode to pipelined mode is identical, also relying on the MPS register.

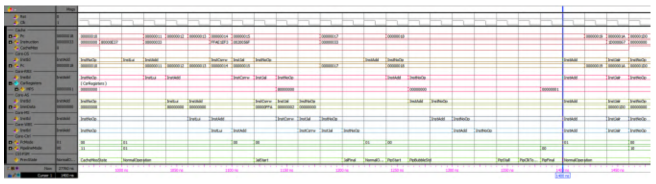

Fig. 4. Multi core block diagram.

C. Exceptions

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 Interruption request signals. These are distributed equally between both cores, and the identification of an unknown instruction, which triggers the error exception. In any case the FSM inside the control unit has a series of states specialized in handling all kinds of exceptions or interruptions, and for the core they both are the same.

The processor also supports nested exceptions, meaning that one exception can be interrupted by another, if the second has a higher priority level than the first one. The priority levels of the exceptions are bound to their IRQ line, and in the case of the error exception, its given priority value is the highest. The nesting feature can be enabled or disabled using the least significant bit of the machine nested exception value (MNEV) CSR. The same goes for interruptions in general, as they can be disabled by setting to zero the eleventh bit of the machine interrupt enable (MIE) CSR.

Figure 5 shows the process of managing an interruption, or in this case four of them. The control unit’s FSM starts by emptying the pipeline registers, then fetches the address of the interruption’s subroutine by using a CSR and an exception code associated with the received interruption. Then the program counter is set to that address and the acknowledge signal is sent to the peripheral that issued the interruption.

If the interruption was nested, then the FSM also sets the new priority level and saves the previous address to return to it later. The process of starting an exception subroutine does not ever save the core’s context and that task is left for the programmer to do. The ARK-I processor assumes that at the end of each subroutine there is a return instruction, which will trigger the return states from the control unit’s FSM. At this point, The FSM will undo the steps of the exception states to go back to the previously stored address, to continue the former subroutine or the main core’s code, depending on the situation.

Fig. 5. Multi core block diagram.

V. Conclusions And Future Work

The top-down design methodology ensured that most potential issues were addressed at that stage, leading to a faster and smoother implementation phase and avoiding any major redesigns.

By providing an open-source core with both modes available, students interested in both operational modes can learn about them, compare them, and even modify them for fitting their needs, such as to expand their knowledge of computer architecture topics or compare both operating modes. Furthermore, since the core uses a popular ISA like RISC-V, people can learn about its main features or, if they are already familiar with it, understand how it integrates with this architecture and optimize it with better clock frequencies, algorithmic or arithmetic blocks, or even new instructions to execute their algorithms.

Using both modes alongside the cache memory and the dual-core implementation could enhance the processor's performance. But, since the clock frequency between both modes is fixed, the monocycle mode is faster in most scenarios. The capability to add this variable clock feature is present in the core and it is one of the main planned additions for future revisions of this project.

References

[1] K. Asanović and D. A. Patterson, "Instruction Sets Should Be Free The Case For RISC-V," Aug. 2014.

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

[3] kammoh, "ORCA RIC-V." https://github.com/riscveval/orca-1.

[4] YosysHQ, "PicoRV32," 2019. https://github.com/YosysHQ/picorv32 (accessed Feb. 09, 2022).

[5] C. Chen 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., vol. 2020-May, pp. 52–64, 2020, doi: 10.1109/ISCA45697.2020.00016.

[6] J. S. Barbosa R. and A. Valdivieso 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, 2021.

[7] I. R. Diaz Gamarra, O. S. Espinel Santamaria, and M. D. Latorre 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, 2020.

[8] P. Jamieson, T. McGrew, 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, pp. 778–781, 2019, doi: 10.1109/CSCI49370.2019.00148.

[9] J. L. H. David A. Patterson, Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, 5th ed. Morgan Kaufmann Publishers Inc., 2013.