Configurable RISC-V softcore processor for FPGA implementation

João Filipe Monteiro Rodrigues

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

November 2019

Acknowledgments

Em primeiro lugar quero agradecer aos meus orientadores, Professor Pedro Tomás e Professor Nuno Roma, por todo o apoio que me deram ao longo destes meses. Tenho a agradecer o tempo que dispensaram a ajudar-me a tentar resolver alguns problemas e a orientar e rever o meu trabalho. Agradeço também ao Nuno Neves por me ter ajudado diversas vezes a tentar resolver problemas relacionados com o Xilinx, por me ter disponibilizado alguns recursos que foram necessários para desenvolver este trabalho e também pela gestão das máquinas utilizadas. Por fim agradeço à minha família e amigos, em especial aos meus pais, que sempre me apoiaram ao longo destes 5 anos, principalmente durante a realização deste trabalho.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia under projects UID/CEC/50021/2019 and PTDC/EEI-HAC/30485/2017.

Resumo

Ao longo dos últimos anos, o mercado dos processadores tem sido dominado por arquiteturas proprietárias que implementam conjuntos de instruções que requerem licenciamento e pagamento de comissões monetárias para que possam ser utilizadas. A ARM é um exemplo de uma empresa que comercializa as suas microarquiteturas para que os fabricantes as possam implementar nos seus próprios produtos e não permite que o seu conjunto de instruções (ISA) seja utilizado noutras implementações sem estas serem licenciadas. O conjunto de instruções RISC-V surgiu com o objetivo de permitir o desenvolvimento de hardware ou software sem custos, através da criação de um ISA de código aberto. Deste modo, é possível que qualquer projeto que implemente o ISA RISC-V seja disponibilizado de forma aberta ou até mesmo implementado em produtos comerciais. No entanto, as soluções RISC-V que têm surgido não apresentam os requisitos necessários para que possam ser incluídas em projetos, nomeadamente de investigação, por estarem mal documentadas ou não oferecerem desempenhos adequados. Com este trabalho, pretendeu-se desenvolver um processador RISC-V que tenha como características a adaptabilidade, flexibilidade e funcionalidades que não estão hoje presentes nas soluções atuais. Como base do trabalho, utilizou-se um processador não-RISC-V, o MB-Lite, que foi modificado para implementar este ISA e foi dotado de funcionalidades que até agora não suportava, tais como: caches, transferência de dados via PCIe, módulo de comunicação em série (UART), contadores e temporizadores, e unidades funcionais multi-ciclo. A solução proposta foi implementada e testada numa FPGA, de modo a verificar o correto funcionamento do sistema e a obter a sua caracterização experimental.

Palavras-chave: RISC-V, conjunto de instruções (ISA), arquitetura softcore, FPGA

Abstract

Over the past years, the processor market has been dominated by proprietary architectures that implement instruction sets that require licensing and the payment of fees to receive permission so they can be used. ARM is an example of one of those companies that sell its microarchitectures to the manufactures so they can implement them into their own products, and it does not allow the use of its instruction set (ISA) in other implementations without licensing. The RISC-V instruction set appeared proposing the hardware and software development without costs, through the creation of an open-source ISA. This way, it is possible that any project that implements the RISC-V ISA can be made available open-source or even implemented in commercial products. However, the RISC-V solutions that have been developed do not present the needed requirements so they can be included in projects, especially the research projects, because they offer poor documentation, and their performances are not suitable. With this work, the main goal was the development of a RISC-V processor that has as characteristics the adaptability, flexibility, and features that are not yet present in the current solutions. As the base of this work, it was used a non-RISC-V processor, the MB-Lite, that was modified to implement this ISA, and it was improved with new functionalities, such as caches, data transfer through the PCIe bus, a serial communication module (UART), counters and timers, and multi-cycle functional units. The proposed solution was implemented and tested on an FPGA in order to validate the system's correct operation and to obtain its experimental characterization.

Keywords: RISC-V, instruction set (ISA), softcore architecture, FPGA

Contents

List of Tables

List of Figures

Acronyms

ALU Arithmetic and Logic Unit.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BAR Base Address Register.

BRAM Block Random-Access Memory.

BSP Board Support Package.

CSR Control and Status Registers.

DMA Direct Memory Access.

DRAM Dynamic Random-Access Memory.

DSP Digital Signal Processing.

EX Execute.

FF Flip-flop.

FPGA Field Programmable Gate Array.

FPU Floating-Point Unit.

GPIO General-Purpose Input/Output.

I2C Inter-Integrated Circuit.

ID Instruction Decode.

IF Instruction Fetch.

ISA Instruction Set Architecture.

JTAG Joint Test Action Group.

LRU	Least Recently Used.
LUT	Look-up Table.
LUTRAM	LUT Random-Access Memory.
MEM	Memory.
MIG	Memory Interface Generator.
PC	Program Counter.
PCIe	Peripheral Component Interconnect Express.
RAW	Read after Write.
ROM	Read-Only Memory.
RTL	Register Transfer Level.
SIMD	Single-Instruction Multiple-Data.
SoC	System on Chip.
SPI	Serial Peripheral Interface.
UART	Universal Asynchronous Receiver/Transmitter.
WAR	Write after Read.
WAW	Write after Write.
WB	Write-Back.

1 Introduction

Contents

Over the years, the CPU market has been dominated by INTEL and AMD, in what concerns their x86 architectures, and by ARM, which is not a manufacturer but develops processor's designs mainly for mobile devices and microcontrollers. Those ARM designs are licensed to the manufacturers who integrate them into their products or use them to develop and sell their own System on Chips (SoCs). These processors are typically closed-source, and even their instruction sets cannot be used to develop third-party architectures without permission and the payment of royalty fees. All these limitations make the development of new processors more expensive and increase the difficulties to have a significant market share.

In 2010, the development of a new Instruction Set Architecture (ISA), called RISC-V, was driven at the University of California, Berkeley. RISC-V has the purpose of creating an extensible and open-source instruction set, not only for academic use but also for commercial products. Under this assumption, it is sometimes referred to as the "Linux" of processors. As a consequence, RISC-V has received significant support from the open-source community, with the adaptation and development of programming tools, such as a GCC compiler with GDB support [1], LLVM toolchain [2], GNU MCU Eclipse [3], C libraries (newlib, glibc) [1], an official ISA simulator (Spike) [4], and a simulator in QEMU [5]. In terms of operating systems, it has already support for the Linux Kernel, FreeBSD, and ports of Debian. The RISC-V Foundation controls the RISC-V evolution, and its members are responsible for promoting the adoption of RISC-V and participating in the development of the new ISA. In the list of members are big companies like Google, NVIDIA, Western Digital, Samsung, or Qualcomm. Currently, NVIDIA and Western Digital are working on their RISC-V microcontrollers [6, 7] to incorporate them in their commercial products, as an alternative to their current ARM solutions.

With the RISC-V development, it is expected that the actual paradigm will change through the creation of a universal and open-source software ecosystem with the contribution of everyone. The main goal is to support multiple implementations, giving the freedom to the companies, universities, or research centers to develop their own hardware solutions, knowing that this software layer exists, and it can be used without restrictions. In the future, if successful, RISC-V may be present in processors of different platforms, such as computers, smartphones, and in other types of microcontrollers.

1.1 Motivation

This thesis was developed under the ambitious premise of creating the means to the development of a massively parallel processor using the RISC-V instruction set. The main goal of that targeted ambition was the implementation of a multicore system with several RISC-V cores. When the existing RISC-V core implementations started to be studied in order to select the one to serve as the base of that work, multiple faults were identified in the available solutions, related to the lack of documentation, low performance, or no support for Field Programmable Gate Array (FPGA) implementations.

The initial approach for that project considered the Rocket [8] core, developed by the authors of RISC-V. The first step was the FPGA implementation, but it was realized that the existing implementable versions were developed for simulation purposes related to existing commercial projects.

Another option that was taken into consideration was the Pulpino [9] SoC. Once again, the performance and the maximum operating frequency were insufficient since it was developed for RTL simulation or Application Specific Integrated Circuit (ASIC) implementations. Other cores available at that time were analyzed, and they will be presented in the next chapter, together with the Rocket and the Pulpino.

Based on that, the idea of developing a new RISC-V core that is easily modified and mitigates those implementations problems arose. It seemed to be a fundamental step to allow the development of future projects that include multiple RISC-V cores, with particular attention being given to the multi-core systems and support for FPGAs implementations. A non-RISC-V core with a robust architecture was selected to be modified and used as the base of this work.

1.2 Objectives

The main goal of this work is the development of a RISC-V softcore processor to be implemented in an FPGA, using a non-RISC-V core as the base of this architecture. The main objectives can be summarized in the following points:

- Implementation of the RISC-V ISA on a non-RISC-V processor;
- Hardware support for the RISC-V Integer Multiplication and Division Extension;
- Creation of hardware support to allow the integration of caches and external memories in the processing structure;
- Development of peripherals (UART module, timer, and counter);
- Give support for high throughput data transfers from the host CPU to the developed core via PCIe;
- Implementation of the developed architecture in a Xilinx Virtex UltraScale+ VCU1525 FPGA board;
- Development of tools to help the software development for the proposed processor;
- Evaluation of the implementation results in terms of resources, operating frequency, and power requirements.

1.3 Contributions

The work that was conducted in the scope of this thesis allowed the development of a new RISC-V processor with new features when compared to the current solutions. Beyond that, the proposed solution was focused on solving the problems and limitations identified in the other RISC-V cores that were analyzed in this thesis, especially in terms of the adaptability and flexibility, allowing future modifications according to the applications needs and requirements. The software support was also taken into consideration, and several tools and libraries were developed to make the software development easier. The complete system was developed, ensuring support for FPGA implementation.

1.4 Thesis Outline

This thesis is divided into 6 chapters. The remaining of this document is organized as follows:

- Chapter 2 presents the RISC-V ISA and the existing extensions, followed by the conducted analysis and comparison of several RISC-V cores. In the end, a reference to other non-RISC-V cores is also done.
- Chapter 3 describes the development process related to the proposed architecture and its features, as well as the decisions that were taken across the work.
- Chapter 4 presents the developed software tools to interact with the processor, including the developed libraries to access the peripherals, compilation scripts, and practical examples to demonstrate how they can be used.
- Chapter 5 includes the implementation process of the developed system in an FPGA, and it also presents the set of experimental results that were obtained.
- Chapter 6 contains the conclusions of this work and ideas that can be developed in future work.

2 RISC-V specification and implementations

Contents

This chapter begins with the analysis of the RISC-V ISA specification, focusing on the base integer instruction set, the existing extensions, and how they work. A particular emphasis will be given to the Integer Multiplication and Division extension which is implemented in the processor developed in this thesis.

Then, several existing RISC-V cores are studied and compared. It is essential to understand what are the functionalities and drawbacks of each one to understand what is missing in the existing solutions and what is important to implement in a new solution. This analysis is essentially focused on the FPGA support, the documentation quality, and how easily they can be adapted and modified.

Finally, as a consequence of the current state of the existing RISC-V processors, other non-RISC-V cores are referenced.

2.1 RISC-V ISA

RISC-V includes a base integer ISA with 32, 64, and 128-bit variants (RV32, RV64, and RV128, respectively). The RV128 variant will be implemented thinking in the future, when an address space larger than 64 bits will be needed [10]. Additionally, it supports optional extensions giving the flexibility to use what is necessary to implement in a specific application. Currently, the RISC-V Foundation has not ratified the standard ISA in the latest release of the ISA Manual [10]. However, in the newest draft release [11], some of the modules suffered significant changes, and others are already ratified, including the Base Integer Instruction Set, RV32I, and the Multiplication and Division extension. Currently, the RV32I features fewer instructions than what was set in the original version, and this means less complexity and resources when it is time to implement them in underlying processor architectures. Based on that, the version "20190608-Base-Ratified", published in June of 2019, was followed across this work [11].

As long as this processor implements the RV32I extension and offers support for integer multiplication and division, it is necessary to know what are the instructions that should be supported and implemented, in order to define the hardware requirements such as the memory size, the register file structure, the Arithmetic and Logic Unit (ALU), the multiplier, and the divider.

2.1.1 Base Integer Instruction Set

The RV32I uses 32 32-bit wide registers (x0-x31) to store integer values, where x0 is a read-only register holding the value zero. The source registers, *rs1* and *rs2*, as well as the destination register, *rd*, are always in the same position in every instruction. In the base ISA, there are 6 formats of 32-bit long instructions: R, I, S, B, U, and J. S and B formats are similar, the only difference is how the immediate term is organized within the instruction. The same happens with U and J formats. Figure 2.1 summarises the 6 instruction formats.

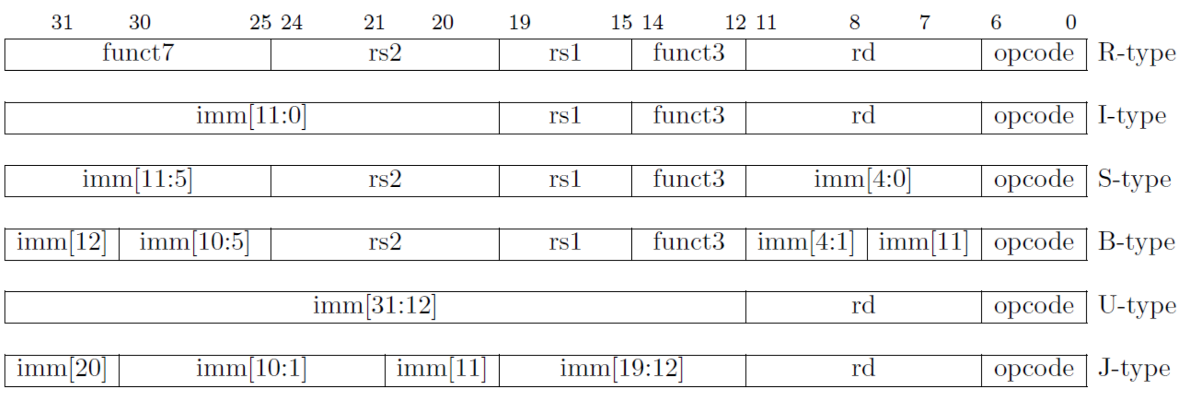


Figure 2.1: The RISC-V base instruction formats [11]. The registers are always in the same position in all formats. The immediate is encoded in different ways depending on the format. Each bit is labeled with its position in the immediate value, imm[x].

Currently, the RV32I (version 2.1) instruction set features 40 instructions [11], and it was designed to reduce the hardware resources, providing support for small implementations. At the same time, this ISA is recognized as a compilation target and is supported by several operating systems. The previous versions of the RV32I included Control and Status Registers (CSR) instructions to access timers and counters in unprivileged code [10]. Currently, these instructions are not mandatory, and a dedicated extension was created. The set divides the instruction into the following five groups:

- 21 Integer Computational Instructions:

  Arithmetic (addition, subtraction, and bitwise shifts), logical (bitwise Boolean operations) and comparison (arithmetic magnitude comparisons) instructions.

- 8 Load and Store Instructions:

  Load byte (signed and unsigned), Load half-word (signed and unsigned), Load Word, Store byte, Store half-word and Store word.

- 1 Ordering Instruction:

  The FENCE instruction is used to order memory accesses by other cores and/or external devices.

- 8 Control Transfer Instructions:

  Both conditional branches and unconditional jumps are supported. Branch instructions compare the values in two registers and assume the conditional branch range, given by the 12-bit signed immediate, leading to a branch range of $\pm$ 1K instructions. Furthermore, there are two jump instructions: JAL and JALR. JAL (jump-and-link) changes the Program Counter (PC) to anywhere in a range of $\pm$ 256K instructions. JALR uses rs1 and a 12-bit signed immediate to do an indirect jump to the address in rs1 plus the immediate.

- 2 Environment Call and Breakpoints instructions:

  System Instructions are used to invoke the debugger or to make system calls. The EBREAK instruction is used to stop the execution and return to debugger control. The ECALL instruction is used to make a call to the environment system.

2.1.2 Extensions

One of the RISC-V main goals is to be suitable for all types of applications, starting from low-end applications, with hardware restrictions, where the base integer instruction set is enough, but also reaching high-performance processors which might require, as an example, floating-point operands or support for division and multiplication operations. This goal can be achieved by defining extensions to the instruction set, which can be implemented or not, according to the requirements and needs of each architecture. Since the extensions are independent, they can be developed in parallel, and they can be created by third parties, without affecting the base ISAs. Table 2.1 features the list of all extensions under development by the RISC-V Foundation and it is possible to observe several extensions already ratified, such as Integer Multiplication and Division, Control and Status Register instructions, single, double and quad-precision floating-point. The extensions marked as *Draft* are in an early stage of development and changes are expected in the future. In this group are included Single-Instruction Multiple-Data (SIMD) instructions and Vector operations, which are essential for data-level parallelism.

Table 2.1: RISC-V ISA Extensions [11].

Extension	Description	Version	Status
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Zicsr	Control and Status Register (CSR) Instructions	2.0	Ratified
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.0	Frozen
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
C	Standard Extension for Compressed Instructions	2.0	Ratified
Ztso	Standard Extension for Total Store Ordering	0.1	Frozen
Counters	Performance Counters and Timers	2.0	Draft
L	Standard Extension for Decimal Floating-Point	0.0	Draft
B	Standard Extension for Bit Manipulation	0.0	Draft
J	Standard Extension for Dynamically Translated Languages	0.0	Draft
T	Standard Extension for Transactional Memory	0.0	Draft
P	Standard Extension for Packed-SIMD Instructions	0.2	Draft
V	Standard Extension for Vector Operations	0.7	Draft
N	Standard Extension for User-Level Interrupts	1.1	Draft
Zam	Standard Extension for Misaligned Atomics	0.1	Draft

By taking in consideration the main objectives of this thesis, a particular attention is worth giving to the Integer Multiplication and Division Extension. This extension allows multiplication and division between two operands stored in two registers. In particular, we focus only on the 32-bit instructions (RV32M), since there are also instructions for 64-bit operands, but they were not used in this work. These instructions use the integer registers to store the results directly instead of using special registers for these type of operations, reducing the latency caused by moving the results from those registers to integers ones. The instructions which comprise this extension are present in Table 2.2.

Multiplication Instructions:

Four multiplication instructions are available, one of which is used to obtain the lower 32 bits of the result and the remaining three allow to obtain the upper 32 bits, according to the sign of the operands. If both lower and upper bits are required, the instructions MULH[S][U] and MUL should be executed sequentially, assuring that the operands registers are preserved. In the architecture level, both operations can be executed using just one multiplication operation, reducing the latency to get both parts of the result.

Division Instructions:

There are two instructions for division, signed and unsigned, and two instructions to obtain the remainder. In these instructions, rs1 stores the dividend and rs2 stores the divisor. The division should round the quotient towards zero, and the remainder should have the dividend's sign, according to the C99 standard. When the quotient and remainder are needed, both instructions should be executed or, as an alternative, the micro-architecture could implement a mechanism to store both results, since a division is an arithmetic operation that takes several clock cycles to get the result. In the RISC-V ISA, it was decided to not trigger any exception when we try to divide by 0. To detect these situations, a branch instruction should be inserted right after the division to check and change the program flow if it is required.

Table 2.2: Integer Multiplication and Division instructions [11].

Instruction	**Meaning**
mul rd, rs1, rs2	rs1 x rs2 multiplication and returns the lower 32 bits
mulh rd, rs1, rs2	rs1 x rs2 (signed x signed) multiplication and returns the upper 32 bits
mulhsu rd, rs1, rs2	rs1 x rs2 (signed x unsigned) multiplication and returns the upper 32 bits
mulhu rd, rs1, rs2	rs1 x rs2 (unsigned x unsigned) multiplication and returns the upper 32 bits
div rd, rs1, rs2	rs1 / rs2 signed division
divu rd, rs1, rs2	rs1 / rs2 unsigned division
rem rd, rs1, rs2	Remainder of *div*
remu rd, rs1, rs2	Remainder of *divu*

2.2 RISC-V Cores

The number of processors based on the RISC-V ISA has increased over time. Currently, there are processors to be used as simple microcontrollers, coprocessors, more advanced SoC with the capability to boot Linux and even more advanced implementations with multicore support and integration of customized accelerators. Several of them were selected to be studied in this section and compared based on their features, maximum operating frequency, FPGA support, resources usage, and documentation. These are the RISC-V cores that will be analyzed and compared:

- PicoRV32;
- Rocket Chip;
- ORCA;
- Potato;
- PULPino;
- VexRiscv;
- SweRV.

2.2.1 PicoRV32

PicoRV32 [12] is a 32-bit RISC-V core written in Verilog and suitable for FPGAs implementation. It supports the following ISA configurations: Integer Reduced (RV32E), Integer (RV32I), Integer Compressed (RV32IC) and Integer Multiplication and Division (RV32IM). The core exists in three variants, with different interfaces to connect peripherals. The *picorv32* provides a simple native memory interface to be used in simple applications. The second variant, *picorv32_axi*, features an Advanced eXtensible Interface (AXI) Lite Master interface, useful for applications which use already AXI to connect peripherals. The third one, *picorv32_wb*, provides a Wishbone master interface.

Each core can have three configurations: small, regular, and large. The small architecture uses about 760 Look-up Tables (LUTs) on Xilinx 7-Series FPGAs, but it does not implement counter instructions, two-stage shifts, and detection of misaligned memory accesses and illegal instructions. The regular configuration implements the features missing in the base configuration, requiring less than 920 LUTs. Finally, the large configuration supports the multiplication and division extension, compressed instructions, barrel shifter, custom instructions for interrupt request and a feature used to implement non-branching instructions in external cores, not requiring more than 2020 LUTs. Regarding the timing evaluation, a maximum frequency of 714 MHz is achieved for the Xilinx Virtex UltraScale+ board with the xcvu3p-ffvc1517-3-e FPGA.

Despite all the technical information available regarding its features and evaluation reports, there is no description available of its internal architecture. It is hard to understand how the processor was built by just analyzing the Verilog code even by the fact that the core was written in just one file without comments. Beyond that, this RISC-V core is not pipelined, which is a disadvantage when compared with other alternatives, as it will be seen further.

2.2.2 Rocket Chip

The Rocket Chip [8] is an open-source SoC generator created at the University of California, Berkeley. It uses the Chisel hardware construction language embedded in Scala to generate cores, caches, and interconnections, to produce a complete SoC in an object-oriented way. The Chisel can generate fast cycle-accurate C++ simulators, low-level Verilog Register Transfer Level (RTL) for FPGA or ASIC emulation or synthesis using standard tools [13]. The Rocket Chip is essentially a library of generators that can be configured and connected in different ways, originating different SoC designs. It can generate an in-order or an out-of-order core, Rocket and BOOM, respectively, which can be attached to coprocessors using an interface named RoCC [8]. Figure 2.2 shows an example of a Rocket Chip instance with all the generators and interfaces represented.

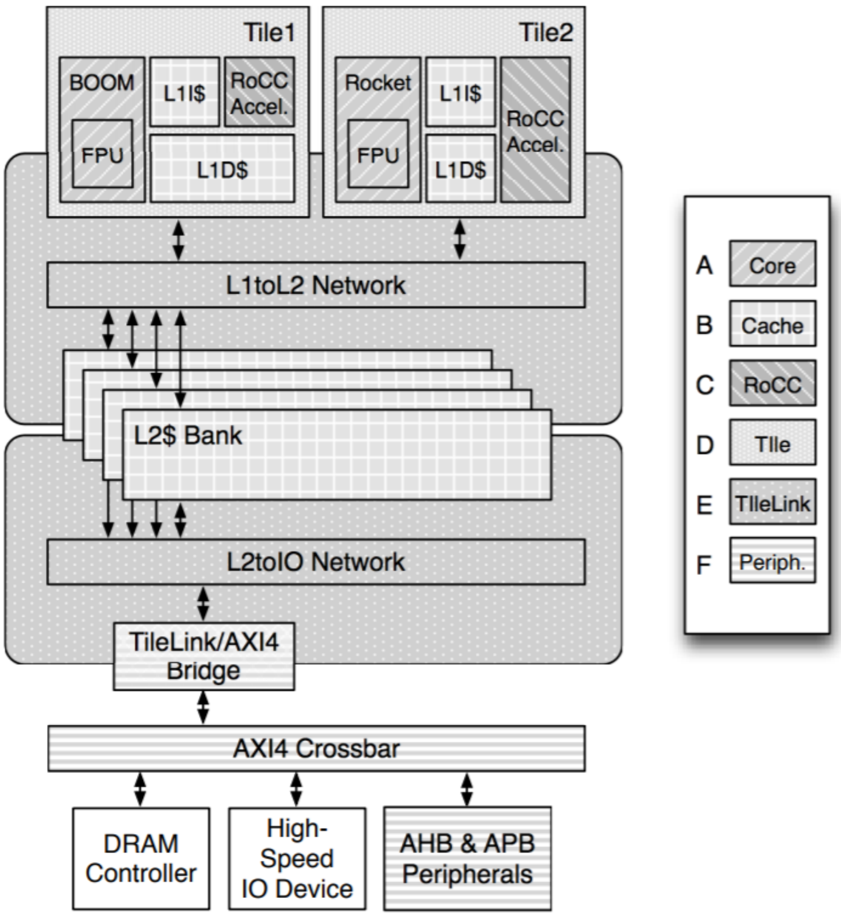


Figure 2.2: Example of a Rocket Chip instance [8]. Core - Rocket or BOOM generator, with an optional FPU. Caches - Cache and TLB generators with configurable sizes and policies. RoCC - The Rocket Custom Coprocessor interface to implement coprocessors. Tile - A tile generator for cache-coherent tiles. It allows defining the type of cores and accelerators. TileLink - A generator for networks of cache-coherent agents and controllers. Peripherals - Generators for AMBA-compatible buses.

The Rocket [8] configuration is a 5-stage in-order core, with support for the RV32I and RV64I ISAs, and it can be configured to include the M, A, F, and D extensions. This core is highly parameterized, offering the possibility to configure different modules, such as the number of floating-point pipeline stages, optional extensions, and the caches and TLB sizes. It has a configurable branch prediction with support for a branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). The Rocket pipeline is represented in Figure 2.3.

With the RoCC interface, the Rocket is able to control external coprocessors and accelerators (vector processors, crypto units, image processing units). The processor attached to the RoCC interface accepts instructions from the Rocket's Write-Back stage, and it also has access to the main core's data cache. According to the authors, Rocket can be thought of as a "library of processor components" because several modules can be easily re-used in other projects [8].

The support for FPGAs implementations is also given, but it is primarily used for tests and simulations, achieving frequencies in the range of 25-100 MHz, depending on the configuration and the FPGA used for implementation. Currently, Rocket is being used by SiFive, a company created by several RISC-V authors, which produces silicon RISC-V processors based on Rocket, and it offers tools to develop, test, and evaluate their designs.

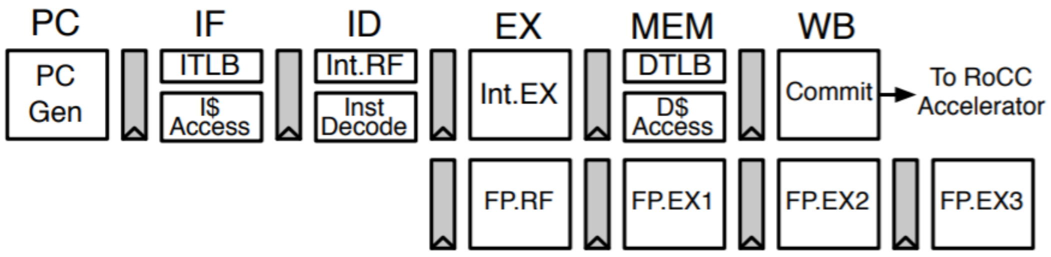


Figure 2.3: The Rocket Core pipeline [8] with 5 stages: Program Counter, Instruction Fetch, Instruction Decode, Execute, Memory, and Write-Back. It is also represented the Floating-point unit and the RoCC interface after the Write-Back stage.

BOOM [14] is an out-of-order core that implements the RV64G ISA. It was built to use the Rocket Chip generator infrastructure, so several modules created for Rocket are also used by BOOM, including the caches, the functional units, and the TLBs. To instantiate a BOOM core in a design generated by the Rocket Chip SoC, the Rocket core tile is replaced by the BOOM core tile. Conceptually, BOOM is a 10-stage processor, but some of them are combined. The stages correspond to the Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Write-Back, and Commit phases. Branch speculation is also supported by BOOM, and it works in the following way: each instruction in the pipeline receives a tag that marks which branch is responsible for the instruction that is being executed. If the branch was mispredicted, then all the instructions that depend on that branch are removed from the pipeline.

BOOM is primarily optimized for ASIC, but it is also usable on FPGAs. It supports the FireSim flow [15], an open-source FPGA-accelerated cycle-accurate hardware simulator that runs on FPGAs on AWS EC2 F1 instances, running at 90+ MHz [16].

Unfortunately, Rocket Chip is not the best solution for someone who needs to modify and implement a RISC-V core on an FPGA, due to the lack of documentation about how to create a clean project and how to modify the processor. Because it offers too many features, it is hard to understand an manipulate, the generated Verilog is not directly suitable for FPGA implementation, and the existing FPGA implementations were only created to test commercial products offering very low operating frequencies.

2.2.3 ORCA

ORCA [17] is an RV32IM RISC-V core written in VHDL and developed by VectorBlox to be the host processor for a proprietary solution, although it can also be used as a standalone processor. According to its authors, ORCA was designed to be suited for FPGAs and to be highly parameterized. The pipeline, present in Figure 2.4, can be configured to have 4 or 5 stages, by merging the Forward and Ex/Mem stages, improving speed or area, depending on what is more important for each implementation. Beyond the number of pipeline stages, the forwarding paths available in the architecture are also fully configurable, as well as the possibility to implement the M extension with hardware support, the number of cycles to compute the shift operations and the use of data and instruction caches. The memory interfaces are designed to support multiple FPGAs vendors. The connection to memory caches is made via AXI. On the other hand, uncached accesses can be made via AXI4-Lite or using an auxiliary interface according to the used FPGA vendor, which can be configured as either WISHBONE, Intel Avalon, or Xilinx LMB.

In terms of area and speed, the RV32I implementation requires about 1620 LUTs, and it can achieve a maximum frequency of 109 MHz in an Altera Cyclone IV FPGA. For the same FPGA, the processor version with hardware support for the multiplication and division extension uses about 2350 LUTs and the maximum frequency increases to 125 MHz.

ORCA is claimed to offer the advantage of being targeted for FPGAs with support for multiple vendors, and its high degree of customization allows the easy creation of different core versions. On the opposite side, there is a lack of documentation, and implementation samples for different FPGAs are not available.

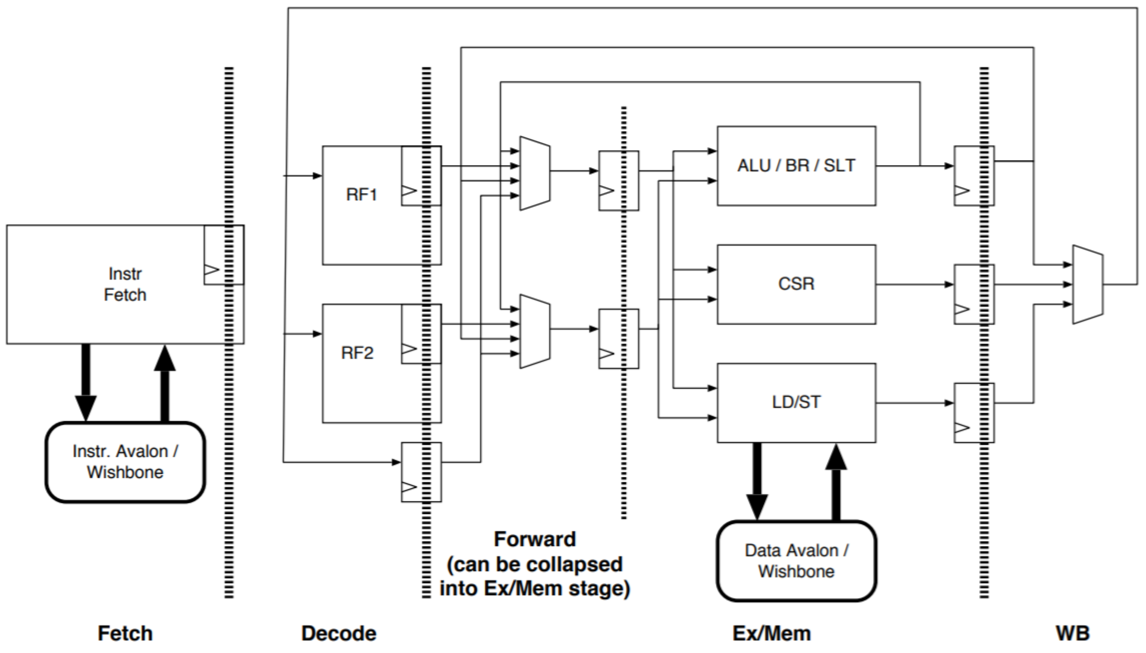


Figure 2.4: ORCA pipeline [18] with 4 or 5 stages. On the Fetch stage it is represented the interface to the instruction memory. The Decode stage features two register files, one for each operand. The Forward and Ex/Mem stages can be merged, and here there are three units: ALU/BR/SLT unit to perform arithmetic, logic, branches and shift operations; CSR unit; and LD/ST unit with access to memory interfaces to load and store data. The Write-Back stage (WB) stores the data on the destination register.

2.2.4 Potato

Potato [19] is a simple RISC-V core written in VHDL with a complete implementation of the RV32I subset and support for the CSRs instructions. Its pipeline has 5 stages, and its block diagram is represented in Figure 2.5. The Wishbone bus interface is used to allow the connection of peripherals and memories. In the source code, it is made available a SoC using Potato, to be implemented in a Arty FPGA running at 50 MHz, with support for a 32-bit timer with compare interrupt, a Universal Asynchronous Receiver/Transmitter (UART) module with configurable baud rate, a General-Purpose Input/Output (GPIO) module, memories using Block Random-Access Memories (BRAMs), and an optional instruction cache.

When compared with other alternatives, this core follows an interesting approach, by using a familiar interface (the Wishbone bus) to connect different peripherals and memories, making the process of adding new peripherals easy. Unfortunately, Potato is not documented, and implementations for different FPGAs with resources usage and timing requirements are not provided.

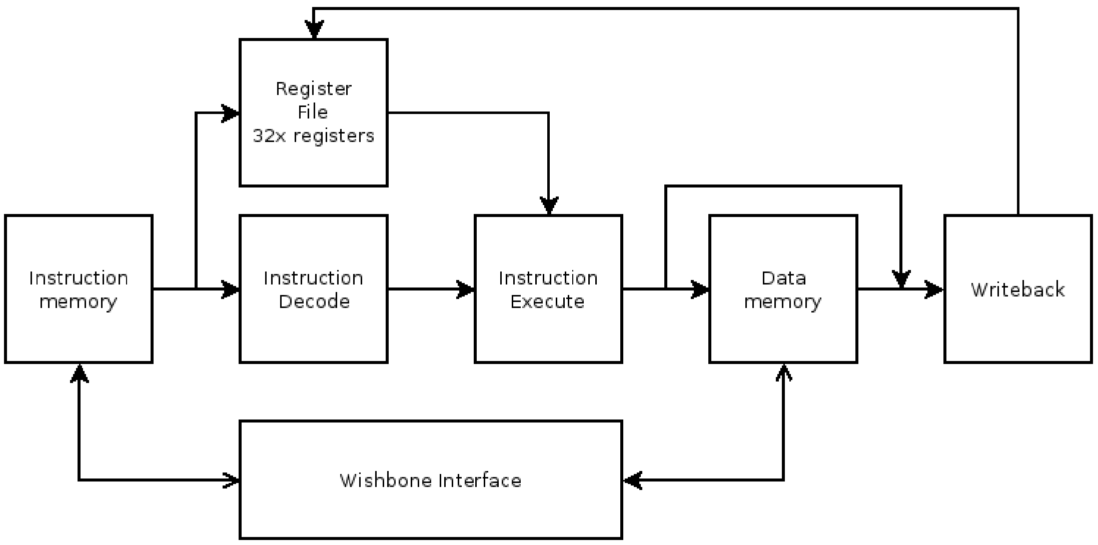


Figure 2.5: Potato block diagram [19]. The pipeline has 5 stages: Instruction Memory (the same as Instruction Fetch), Instruction Decode, Instruction Execute, Data Memory, and Write-Back. The Instruction Memory and Data Memory stages have access to the Wishbone Interface, where the memories and peripherals are connected.

2.2.5 PULPino

PULPino [9] is a 32-bit SoC written in System Verilog, developed in ETH Zurich, which can implement two types of RISC-V cores with the same external interfaces: RI5CY and ZERO-RISCY. This SoC supports several peripherals through AXI, such as UART, Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), 32-bit timers, as well a boot Read-Only Memory (ROM), and data and instructions RAMs. PULPino also offers a debug unit which provides external access to its memory and peripherals via Joint Test Action Group (JTAG).

RI5CY [20], whose architecture is represented in Figure 2.6, is a 32-bit in-order 4-stage core that implements the RV32I and the M and C extensions, Multiplication/Division and Compressed instructions, respectively. This core also supports the floating-point extension, by enabling an optional IEEE-754 single-precision Floating-Point Unit (FPU). Additionally, RIS5Y implements several custom instructions, such as hardware loops, post-incrementing load and stores, ALU operations, and packed-SIMD instructions.

ZERO-RISCY [21], which is now denominated as Ibex and it is maintained by lowRisc, is focused on efficiency and area optimization, and it can be used as a processor for control applications. Figure 2.7 illustrates its architecture based on a 2-stage 32-bit in-order pipeline, which implements the RV32I or a different version with just 16 integer registers called RV32E and the M and C extensions. According to the requirements and specifications of each implementation, the M extension can be disabled and the RV32E can be selected, reducing its area and improving its efficiency.

The SoC and both cores are well documented; each one has its manual, which describes the architectures, the offered features, the interfaces between components, the existing peripherals, and the configuration options. However, even though a version of PULPino for FPGA also exists, PULPino is mainly targeted for RTL simulation and ASICs, so the FPGA implementation can only be used as an emulation platform because the performance is not optimized and the maximum frequency is not higher than 40 MHz.

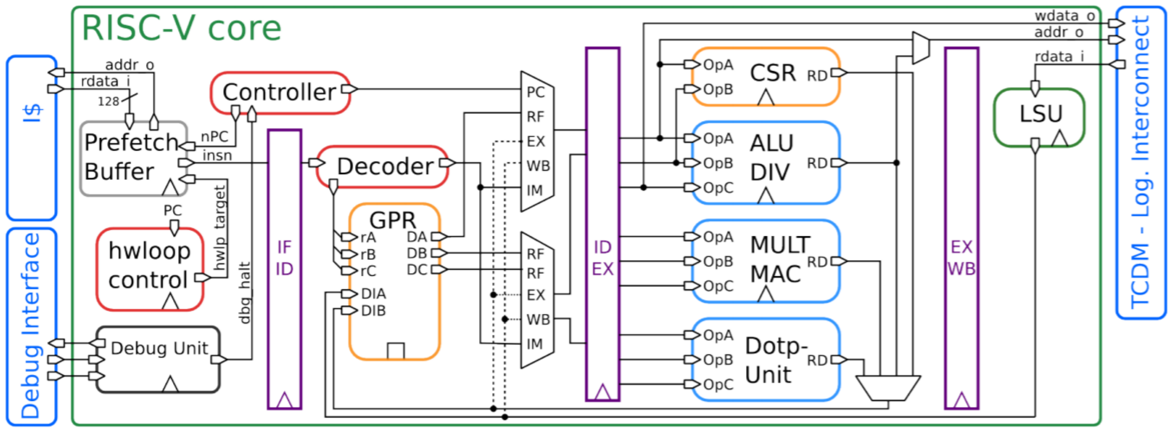


Figure 2.6: RI5CY block diagram [9]. RI5CY has a pipeline with 4 stages: Instruction Fetch, Instruction Decode, Execute, and Write-Back. It offers support for the RV32I, multiplication and division extension, compressed instructions, floating-point operations, and several custom instructions. As external interfaces, it has an instruction cache, debug access and data memory interconnection.

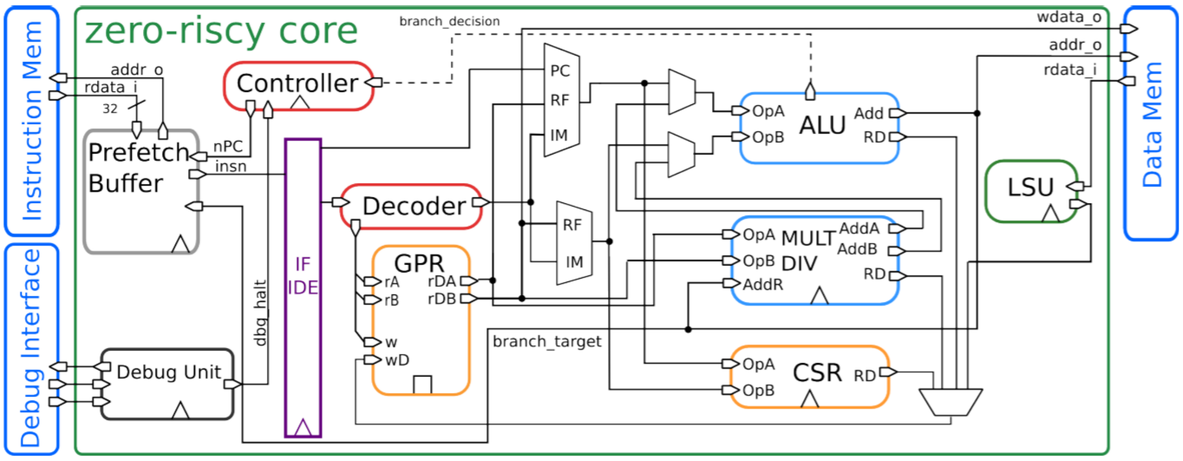


Figure 2.7: ZERO-RISCY block diagram [9]. This core was designed to be efficient and area-optimized, providing a 2-stage in-order pipeline with support for the RV32IEMC. It offers external access to instruction and data memories and a debug interface.

2.2.6 VexRiscv

VexRiscv [22] is a 32-bit core whose pipelined architecture can be configured with 2 to 5 stages (Fetch, Decode, Execute, Memory, Write-Back). It is descibed with SpinalHDL, an open-source high-level description language regarded as a fork of Chisel. This processor supports the RV32IMCA instruction set, and it is optimized for FPGAs of different vendors. VexRiscv was built using a software-oriented approach to develop its hardware description, and so almost all the components are plugins, such as the instruction and data caches, debug extension, multiplication and division, as well as the number of pipeline stages, interrupts and exception handling.

Several configurations are supported, leading to different values of area usage and maximum frequency. The smallest configuration (without any optional module) requires just 496 LUTs and 505 Flip-flops (FFs) and achieves a frequency of 324 MHz on a Artix 7 FPGA. The complete core, called VexRiscv full max, with multiplication and division support, debug module, 16 KB of data and instruction caches, and exceptions, makes use of 1758 LUTs and 1094 FFs, achieving 193 MHz on the same FPGA.

The hardware description language that was used to build this core is not well known by the community, creating barriers to understand and modify the code. Also, the core architecture is not described in the available documentation.

2.2.7 SweRV

Western Digital also decided to develop a RISC-V processor named SweRV [23, 24], to be part of some of their products instead of using a paid and closed-source solution. The SweRV is a 32-bit core written in System Verilog with support for the RV32I and the M, C, Zifencei, and Zicsr extensions. Figure 2.8 shows the SweRV 9-stage pipeline, structured as a dual-issue, superscalar, and mostly in-order pipeline, although with some out-of-order capability. According to the SweRV Manual [24], the pipeline has 4 stall points: Fetch 1, Align, Decode, and Commit. The Fetch 1 stage stalls due to instruction cache miss and if the remaining stages are full. The Align stage has 3 fetch buffers to temporarily store the instructions when it is necessary to stall. The Decode stage can simultaneously decode up to 2 instructions from 4 instruction buffers. Finally, in the Commit stage, up 2 instructions are committed at the same clock cycle. The core also features a load and store pipeline with 3 stages, 2 execute pipes, I0 and I1, each one with 2 ALUs on EX1 and EX4, a 3-cycle latency pipelined multiplier, and a 34-cycle out-of-pipeline divider. Furthermore, it has a branch predictor and 4 interfaces to instructions fetch, data accesses, debug accesses, and external Direct Memory Access (DMA) via AXI4 or AHB-Lite.

The documentation is complete, focusing on several aspects, such as memory accesses (address regions, memory protection, exception handling), power management (power states, power control), performance monitoring (counters, events), and caches. The SweRV can achieve a frequency of 1 GHz for 28 nm nodes, but unfortunately, it is not suited for FPGA implementations.

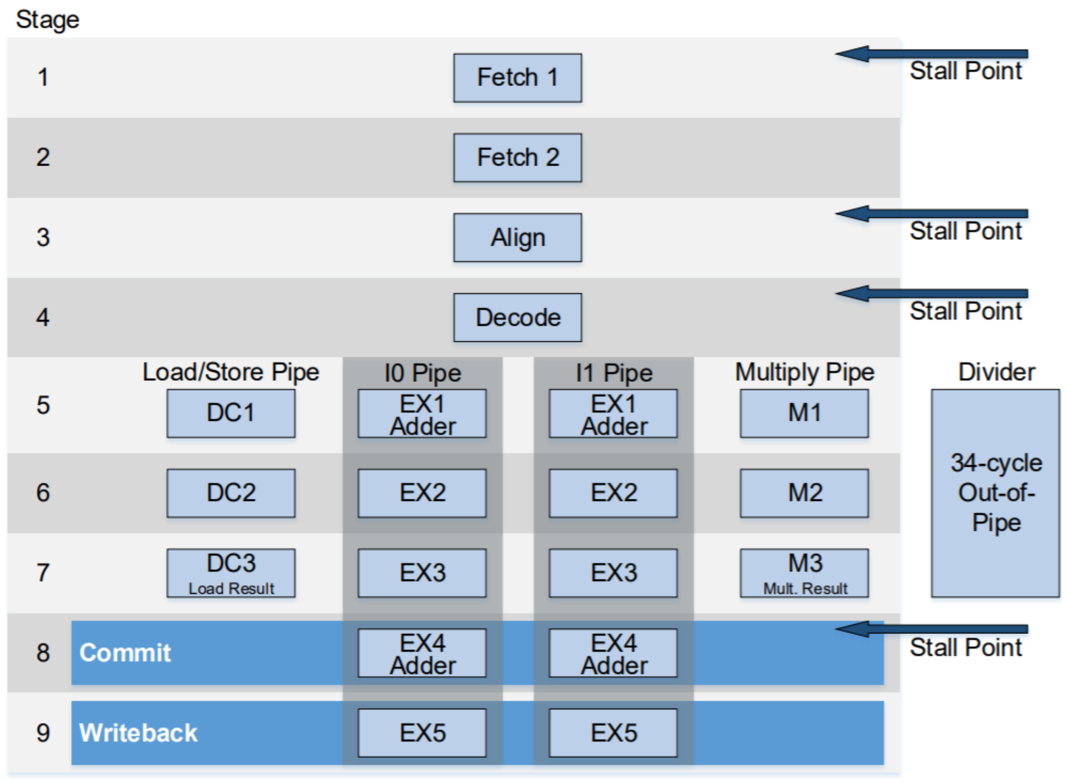


Figure 2.8: SweRV pipeline [24] with 9 stages. The decoder features two instructions pipes on its output, allowing the simultaneous issue of two instructions to the execute stage. The number of pipeline stages of each functional unit is represented, as well as the divider, which has a latency of 34 cycles but is not pipelined.

2.2.8 Discussion

Seven RISC-V processors were separately analyzed, focusing on the architecture, the supported RISC-V extensions, the offered features, and the implementation results. To simplify the comparison, this information was extracted and resumed in Table 2.3. Throughout this section, the collected information will be used along the discussion, in an attempt to find the most suitable RISC-V softcore processor to be used in research projects using FPGAs. The discussion is based on the following criteria:

- Extensibility/customization;
- FPGA implementation support;
- Available documentation.

Extensibility/customization:

If a processor is being used in some project, it certainly requires a way to communicate with other components such as memories, standard input/output, or custom IPs. To make this possible, the core should support a communication protocol like AXI, Wishbone, UART, or SPI. The connection to other peripherals is not the only topic related to the extensibility of a processor. A processor can be considered extensible if its architecture is designed in a way that makes the configuration and the modification simple. This characteristic can be achieved by dividing the design into well known and defined components, using similar interfaces and conventions across the entire architecture, and by adopting the use of generics to set the configurations.

From the set of processors previously reviewed, the ORCA and the VexRiscv support three different interfaces: AXI, Intel Avalon, and Wishbone. This way, these two processors are compatible with different interfaces used by different vendors. The PicoRV32 supports AXI and Wishbone, but it is not compatible with Intel Avalon. The Rocket Chip, the PULPino, and the SweRV are only compatible with AXI. The Potato only offers the Wishbone interface. These interfaces can be used to connect to external memories, DMAs, accelerators, or other modules like UART, I2C, and SPI as it is done in the PULPino.

In terms of the architecture, the Rocket Chip is the most extensible and modifiable processor because it supports multicore implementations with different types of cores, caches, and accelerators. Due to the existing interconnections, which allow the development of the SoC in an object-oriented way, each component uses a specific interface and can be easily attached and modified. The remaining cores, except the SweRV, are also configurable. In general, they have different variants where the type of interfaces, the supported ISA extensions, the caches, and the type of cores (specifically in PULPino) can be configured. The SweRV was designed to be used in commercial projects, so the available design is fixed when compared with the others, but nothing prevents it from being modified since it is open-source.

FPGA implementation:

In research projects, the softcores are mainly used to be implemented on FPGAs. Therefore, their design should be suitable for FPGA implementation, and whenever possible, with support for multiple vendors. Sometimes, the developers provide support for FPGAs but only to perform tests and simulations, mainly due to the fact that the architecture was developed without following the most suitable methodologies, causing bad synthesis results with high minimal clock periods and high usage of resources.

Almost half of the analyzed processors have maximum operating frequencies under 100 MHz. The Potato, which was developed for FPGAs, features a maximum frequency of 50 MHz. The PULPino is mainly targeted for simulation and ASIC implementations, achieving a frequency of 40 MHz on FPGAs. The existing implementations of the Rocket Chip for FPGAs are limited to 100 MHz since it is also targeted to ASIC. On the other hand, the PicoRV32 is the processor with the highest frequency, 714 MHz in a specific Xilinx UltraScale+ device and a value between 250 and 450 MHz on the 7-Series Xilinx FPGAs. The VexRiscv achieves a maximum operating frequency that can range between 193 and 336 MHz, according to the configuration, when implemented on the Xilinx Artix 7. Finally, the ORCA processor, which was developed to support multiple vendors, is limited to 125 MHz on the

Altera Cyclone IV FPGA. The SweRV does not offer any implementation for FPGA.

Documentation and Support:

The documentation of a processor is essential to understand how it was developed, the internal architecture, its features, its purpose, and how it can be used. The documentation can be a technical report, a manual, the code comments, or a guide to help who is using the processor for the first time. The existence of scripts to help the compilation process, tools to test processor, and project examples are also valuable.

From the processors referred above, the SweRV from Western Digital has the complete documentation certainly because it will be used in commercial products. The PULPino is also well documented, having documentation for the SoC and both cores are available. The Rocket Chip offers a good description of both cores, Rocket and BOOM, but on the other hand, the platform itself is not well documented because it is not easy to understand how a process can be created. The remaining cores, the PicoRV32, the ORCA, and the VexRiscv are poorly documented. The PicoRV32 internal architecture description does not exist, and its code is not well organized. The VexRiscv core is not described, and it uses an HDL language that is not very well known by the community.

Conclusion:

The platform with the best documentation is the SweRV, followed by the PULPino. However, when an existing FPGA implementation is a requirement, the SweRV does not offer support, and PULPino is not able to run on high frequencies. The cores with the best FPGA support are the PicoRV32 and the Vexriscv. Nevertheless, PicoRV32 is not pipelined, and both are not well documented. The Rocket Chip is the best solution when it comes to the extensibility, but it is not targeted to FPGAs, and the documentation about the SoC is poor.

Hence, it is reasonable to conclude that, the existing RISC-V softcores are not suitable to be used in research projects, since it was not possible to find a solution that reasonably meets all of these three topics.

Table 2.3: Reviewed RISC-V Cores comparison.

		Cores								
Features		**PicoRV32**	**Rocket Chip**		**ORCA**	**Potato**	**PULPino**		**VexRiscv**	**SweRV**
			Rocket	BOOM			RIS5Y	ZERO-RISCY		
Architecture		32-bit	32/64-bit	64-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit
RISC-V modules		I, E, M, C	I, M, A, F, D	I, M, A, F, D	I, M	I	I, M, C, F	I, E, M, C	I, M, C, A	I, M, C, Zifencei, Zicsr
Pipeline	Type	-	In-order	Out-of-order	In-order	In-order	In-order	In-order	In-order	Mostly in-order
	Stages	-	5	10	4/5	5	4	2	2/3/4/5	9
Hardware Description Language		Verilog	Chisel		VHDL	VHDL	System Verilog		SpinalHDL	System Verilog
FPGA support		Yes	Yes, for simulation purposes		Yes	Yes	Yes, for simulation purposes.		Yes	No
Max. Frequency		714 MHz	100 MHz		125 MHz	50 MHz	40 MHz		small: 324 MHz full: 193 MHz	1 GHz for 28 nm nodes
FPGA resources usage		small: 761 LUTs base: 917 LUTs large: 2019 LUTs	-	-	2350 LUTs	-	-	-	small: 496 LUTs, 505 FFs full: 1758 LUTs, 1094 FFs	-
Peripherals		AXI, UART, SPI	AXI		AXI, Avalon, Wishbone	UART, GPIO module	UART, I2C, SPI, timers, JTAG debug		AXI, Avalon, Wishbone	AXI, JTAG
Suited for FPGAs		Yes	No		Yes	Yes	No		Yes	No
Documentation		Insufficient	Insufficient		Insufficient	Insufficient	Excellent		Insufficient	Excellent

2.3 Other non-RISC-V softcores

Currently, there are also available multiple non-RISC-V open-source processors that were developed in the past decade. Several of them have what is missing in the current RISC-V cores: proper documentation, FPGA support with high operating speeds and low resources usage, and a design which can be easily modified and adapted. Consequently, if we select one of these processors and modify the architecture to implement the RISC-V ISA, in the end, it should be theoretically possible to obtain similar results in terms of speed and area.

2.3.1 Non-RISC-V softcores review

Over the years, different researches were made to evaluate the existing cores based on different criteria. In [25], the comparison and analysis of different processors were focused on the stability and usability, the compiler, the ISA, the implementation, and the quality of the available documentation. In [26, 27], some of the existing open-source processors were compared with commercial solutions. The main goal was the development of a portable and customizable microprocessor for rapid system prototyping, based on an existing core, which should be suitable for different research projects. It was defined a list of requirements that a processor of this type should meet, such as the fact that the processor RTL description should be open-source, reliable and with a high-quality design, configurable, small (in terms of resources), and it should follow a design methodology to make it easier to understand and modify.

The list of analyzed processors includes the aeMB, the LEON3, the OpenFire, the OpenRisc 1200 [28], and the Plasma. These processors were compared based on synthesis results (area and speed), toolchain results using different benchmarks, configurability, and design quality. At the end, the list was reduced to just two cores, LEON3, and aeMB, where both candidates showed a similar performance during the benchmarks. The LEON3 was better in terms of configurability and VHDL code quality, but it was too large and complex, requiring time to understand it. As a result, it cannot be quickly integrated into different projects. Hence, the aeMB was initially figured as a convenient option. However, due to some implementation errors and the lack of documentation, the process of modifying and improving the aeMB architecture faces several difficulties.

In this domain, another alternative can also be considered: the development of a processor from scratch. For this purpose, the MB-Lite [26, 27] softcore represents a promising possibility when focusing on the implementation of the requirements initially proposed and following the two-process design methodology [29], by J. Gaisler. Furthermore, it proved to be a reliable alternative in the field of the open-source softcores. It has a small size, but at the same time, it can achieve similar frequencies as its competitors. The way the code is organized and commented, the modularity and the interfaces it offers, along with the provided documentation and good practices followed during its development contribute to make this processor an excellent option to use in research projects. Due to its particular suitability to endure the objectives address by this thesis, the MB-Lite architecture and implementation results will be presented with more detail in the next subsection.

The two-process methodology used by the MB-Lite suggests the usage of two processes per entity: one process contains all the combinatorial logic, and the other contains the sequential logic, instead of having just one with all the logic mixed. Thus the combinatorial process controls the state and the outputs, based on the inputs and values stored in registers controlled by the sequential logic. This method increases the code organization and the readability, the abstraction level, provides a uniform way to encode the algorithms, and it could even simplify debugging, due to the clear separation between the sequential and the combinatorial logic.

In [30], the work was focused on the development of a new open-source core, the SecretBlaze. The authors were worried about the fact that the existing processors are not protected against powerful cryptanalysis techniques, called Side-Channel Attacks (SCAs). The SecretBlaze was developed following the same methodologies used by the MB-Lite, and its design quality and modularity were emphasized and recognized by the SecretBlaze's authors. The SecretBlaze was implemented on a Xilinx Spartan-3 FPGA, and it was able to achieve 90 MHz of maximum frequency. In terms of resources, it required about 630 FFs and 1180 LUTs. These results are not so good as what the MB-Lite can offer, as it will be seen later.

2.3.2 MB-Lite

The MB-Lite [26, 27] softcore was developed as part of T. Kranenburg's Master Thesis, at Delft University of Technology. It implements the Xilinx's 32-bit RISC MicroBlaze ISA, and it can execute programs compiled by the standard compiler without modifications. However, not all the MicroBlaze instructions were implemented to simplify the architecture. Some instructions that can be configured by compiler parameters, such as the hardware multiplier and barrel shifter, can be added to the processor by modifying the configuration parameters, according to the application requirements. As an alternative, these units can be replaced by software libraries.

The implementation results reported for a Xilinx Virtex 5 FPGA board showed that a basic version of the MB-Lite processor without multiplier and barrel-shifter is capable of achieving a frequency of 222 MHz, requiring 843 LUTs and 355 FFs. When the multiplier and the barrel-shifter are enabled, the frequency reduces to 65 MHz, and the resources increase to 1450 LUTs. MB-Lite, when compared to the original MicroBlaze, is faster and smaller, and it was also faster than almost all the softcores analyzed by T. Kranenburg, losing only to the aeMB, which has a maximum frequency of 279 MHz.

The MB-Lite architecture, represented in Figure 2.9, is based on the MIPS processors, featuring a 5-stage pipeline with a modular implementation design, and two separated instruction and data memories. Each stage was carefully developed and described in separated components, following the same hardware description methodologies and signals naming. The exception is the Write-Back stage, which was implemented in the same component as the Instruction Decode, due to its connection to the Register File, but in this schematic, they were represented separately to show the traditional view of the pipeline. The five pipeline stages are the following:

Instruction Fetch:

The Instruction Fetch stage is responsible for controlling the program counter and for fetching the next instruction from the instructions memory. The program counter has three possible sources: during the normal execution, the last program counter value is incremented; when a branch occurs, the next program counter uses the branch target address; finally, when the reset signal is asserted, the program counter is set to zero. When a hazard occurs, the program counter needs to remain unchanged. The program counter value is directly connected to the instruction memory address port, and it should be ready before the rising edge of the clock because the instruction needs to be available in the next cycle in the Instruction Decode stage.

Instruction Decode:

The Instruction Decode stage determines all the control signals required for the execution of the instruction in the following pipeline stages. Based on the operation code (opcode) the instruction is identified, the registers addresses and the immediate value are decoded, the dependencies are verified, and the control signals for the remaining pipeline stages are generated. Due to a specific MicroBlaze ISA instruction, STORE WORD, which uses three register values (A and B for the address and D for the value to be stored), the MB-Lite requires a true dual-port 32x32-bit register file to be able to read three operands at once.

The reset, flush, and stall signals should be evaluated to determine if the instruction can be issued in the next cycle. When the reset signal is asserted, the current program counter and the instruction are changed to zero. There is also a mechanism to detect hazards between the current instruction and previous ones. This mechanism stores the current program counter and the instruction, and being decoded inserts a NOP instruction to stall the pipeline. If the instruction decode stage receives a flush order from the execute stage when a branch is executed, the current instruction must be cleared before the next clock cycle.

As it was previously said, the connections between the Write-Back stage and the Register File are implemented in this stage to store the execution results or the values loaded from the data memory in the destination registers. In this way, the VHDL code is simplified because the Register File signals are all defined in the same component.

Execute:

The execute stage is responsible for performing the aimed ALU operation and for obtaining the branch result according to the branch conditions. Before the execution of the operation, the operands should be selected. They are obtained either from the register file, the program counter, the immediate value encoded in the instruction, or from the forwarding values coming from the Write-Back stage, the Memory stage or the previous ALU result, depending on what is the most recent value. The ALU offers support for arithmetic and logical operations, branch evaluation, and it can also be configured to include a single-cycle barrel-shifter and an integer multiplier.

Memory:

The Memory stage provides access to the data memory to store and load values. The ALU result defines the memory address and the value to be stored corresponds to operand D. The memory access can be made using different sizes: byte, half-word, and complete word. In rare cases, a data hazards occurs. In particular, when a store instruction is immediately executed after a load instruction, and the loaded value is immediately stored to the memory. This situation could be simply solved by forwarding the loaded value to the Memory stage, but due to the additional logic, the performance would be affected. As an alternative, a stall can be inserted in the pipeline when this case is verified. This forwarding can be enabled or disabled in the configuration file.

Write-Back:

The Write-Back stage receives either the ALU result (from the Memory stage) or the output of the Data memory (when there is a load instruction), and one of these is written back to the destination register. According to the control signals, it chooses the correct value and it selects the destination register to store the value in the register file. Some of the signals that contain the output memory data and the ALU result are also used as forwarding data signals to the previous stages.

By using VHDL generics, the core uses different configuration parameters that can enable the multiplier, the barrel shifter, the external interruption, different memory settings, and memory-mapped peripherals. To support the connection of multiple devices, the MB-Lite also offers a configurable address decoder that can be directly connected to the data memory bus. This bus is composed of the data input and output signals, the address, the enable, and the write enable signal, with byte selection. In the configurations file, the memory map and the number of peripherals can be configured using generics. When the address decoder receives a request, the address is decoded and the corresponding slave's chip-enable signal is activated according to the memory map. One of the slaves can be a Wishbone bus adapter, allowing the connection of Wishbone compliant IPs with different latency cycles. This adapter is also responsible for disabling the core until the end of a transaction during a Wishbone cycle. In Figure 2.10 it is shown an example of a connection between the core and external devices using the memory map address decoder.

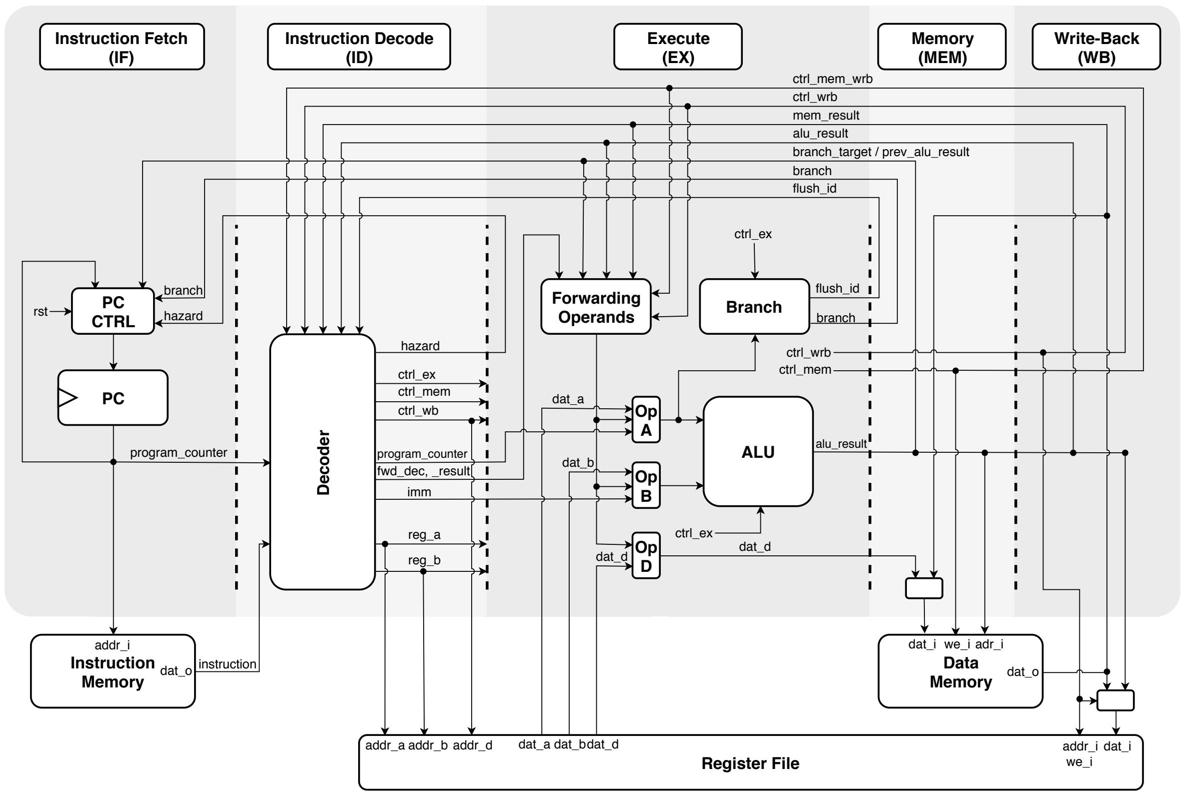


Figure 2.9: The MB-Lite architecture. The pipeline consists of 5 stages: Instruction Fetch, Instruction Decode, Execute, Memory, and Write-Back. The core itself is connected to the external memories and the Register File, also present in this diagram. Some signals and logic are not represented to simplify the figure.

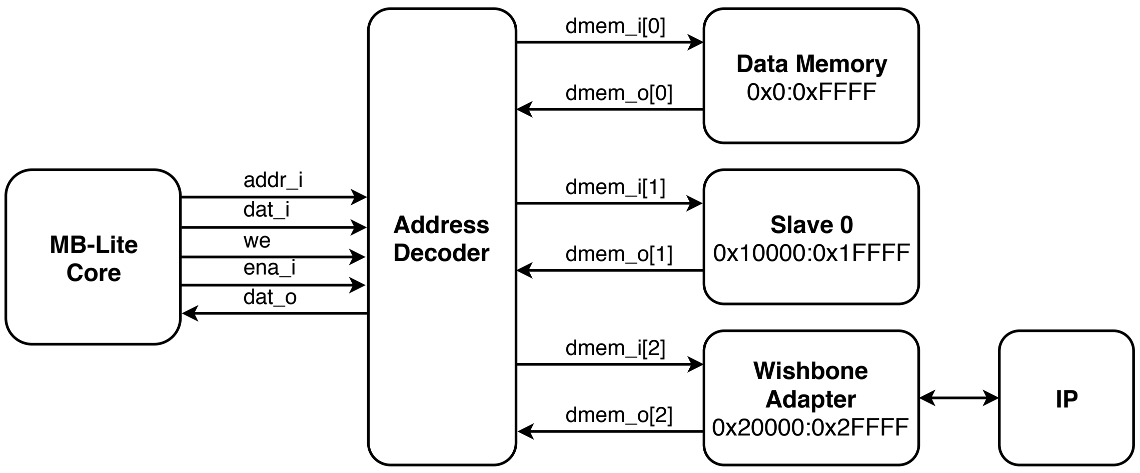


Figure 2.10: The MB-Lite memory map address decoder. In this example, the connections between the core and three external peripherals are represented: the data memory, a generic slave, and a Wishbone IP connected through the Wishbone Adapter. The memory map address decoder uses the data memory interface to establish the connection between the peripherals and the core, according to the memory map set by VHDL generics.

2.4 Summary

Throughout this chapter, the state-of-the-art related with the RISC-V ISA was analyzed, with a special attention given to the instruction set and some of the existing implementations. It started by presenting the RISC-V evolution, comprehending not only the ISA but also the support provided by the open-source community, companies, and universities. Regarding the instruction set, the base integer set and the integer multiplication and division extension were discussed.

Looking to the existing RISC-V core implementations, seven were presented and compared, using as criteria the available documentation, the maximum working frequency, the FPGA support, and resources usage. Based on this evaluation, it was possible to conclude that none of them were able to meet all the required aspects. It was also suggested the hypothesis of modifying a non-RISC-V core which already meets these requirements to support the RISC-V ISA. From the existing non-RISC-V cores, the MB-Lite was considered the most suitable according to the proposed requirements, so its architecture and implementation results were studied at the end of this chapter.

3 Proposed Architecture

Contents

This chapter presents the architecture development of the proposed processor. The main focus is on the modifications that were introduced to the MB-Lite architecture and the implementation of the RISC-V ISA, as well as the new features, and the taken decisions related to the FPGA implementation. This presentation is introduced by an architecture overview of the processor, where the main structural changes are highlighted. In the following sections, each component is explained in more detail.

3.1 Architecture Overview

By analyzing the MB-Lite original architecture, described in the previous chapter, one realizes that it offers fewer features than the existing RISC-V cores. Stands out the support for multi-cycle functional units, caches, and pre-built peripherals, such as UART, timers, and counters. Based on this observation, the solution that was developed during this thesis was focused not only on the implementation of the RISC-V ISA but also on keeping the MB-Lite characteristics, namely the methodologies used and its reliability. The new features of the architecture beyond the support for the RISC-V ISA include:

- Development of a multi-cycle divider and a multiplier and providing support for additional functional units.
- Support for the execution of some instructions out-of-order, after the ID stage.
- Development of a specialized unit for conflict detection and handling, caused by the out-of-order execution and other structural hazards.
- Creation of an extra writing port on the Register File to write-back the non-memory instructions directly after the EX stage, avoiding an extra stage.
- Development of peripherals (UART module, counter, timer, and data cache).
- Addition of a buffer that works as a waiting queue between the EX and the MEM stage, to store memory request instructions until they can enter to the MEM stage, in order to keep the issuing of instructions.

The processor architecture, represented in Figure 3.1, shows the changes that were introduced on the MB-Lite pipeline. The new signals and units are represented in orange. To keep the schematic more straightforward, some simplifications were made. The EX stage now has support for a multi-cycle pipelined divider and multiplier, with configurable latencies, allowing out-of-order execution in the remaining stages. To detect and solve the hazards introduced by these modifications, it was necessary to develop and include a Dependency Handler unit on the ID stage. This unit is responsible for analyzing the current instruction type, the operands dependencies, and the state of the functional units, in order to issue or stall the processor whenever required.

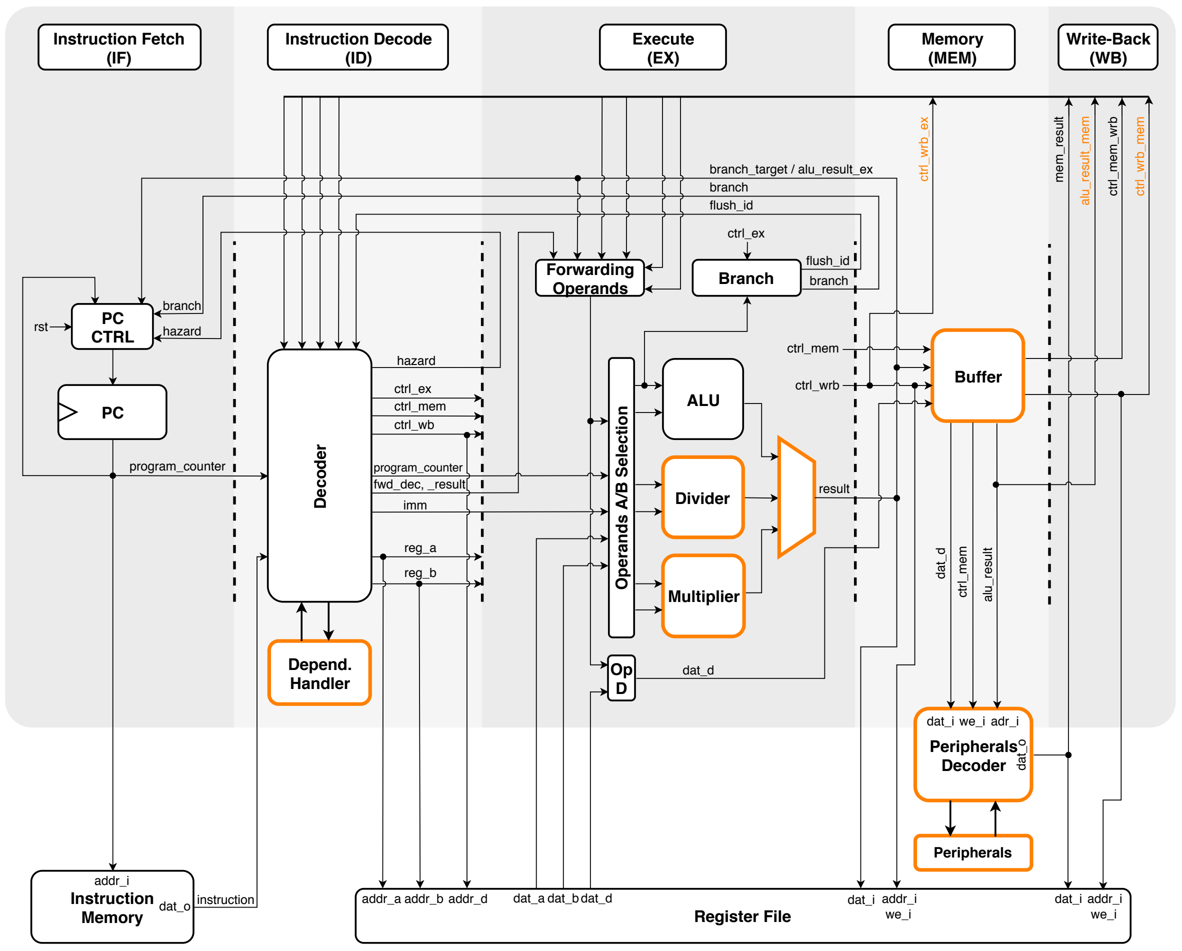


Figure 3.1: Processor Architecture schematic with simplifications. The new units and signals are represented in orange. A dependency handler unit was included on the ID stage to detect hazards. The EX stage features a multi-cycle pipelined divider and a multiplier. The Register File was modified to support two write ports since the EX stage can now write directly to the destination registers. The MEM stage features a buffer to store instructions that are waiting for their request to be processed. By using an Address Decoder, it is possible to connect different peripherals such as the data cache or the UART module, not represented here.

The MEM stage was modified to support memory requests with unknown latency. This requisite was necessary to support the interconnection of a data cache. By using an address decoder, it is possible to set the address map and connect the cache and other peripherals using the same interface.

To avoid stalls on the processor's pipeline every time an instruction is waiting on the MEM stage until the memory request is complete, different ways to continue the execution of instructions (without dependencies) were identified. The first modification consists in the implementation of a new write port on the RF and forwarding non-memory request instructions directly from the EX stage to the WB. The second improvement was the creation of an instructions buffer (Figure 3.2) with a configurable size on the entrance of the MEM stage to allow the issue of new instructions without dependencies, even when there is one instruction waiting on the MEM stage. If there is an empty space inside the buffer, the ID stage will continue issuing memory instructions.

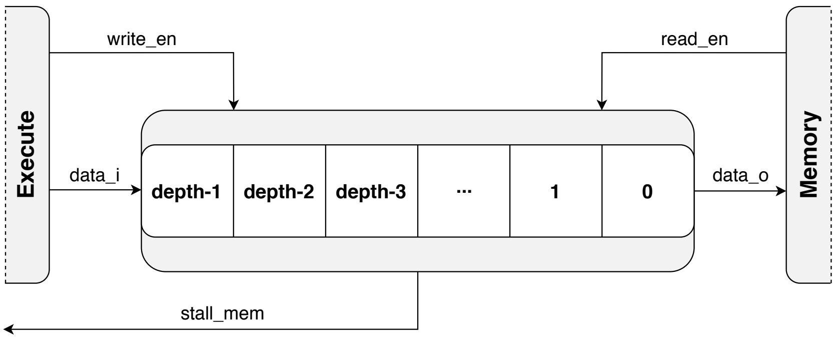


Figure 3.2: Memory stage instructions buffer. Based on the occupation of this queue, the processor will continue issuing new memory instructions in case of having free space. Whenever the buffer is full, the pipeline stalls waiting for the acknowledgment of the instruction that is being processed.

3.2 RISC-V support

As referred to in the previous chapter, the MB-Lite processor is an open-source implementation of the MicroBlaze ISA [31]. To implement the RISC-V ISA, the required changes were mainly focused on the instruction decoder because it was necessary to change the parsing of each instruction due to differences in the formats.

The MicroBlaze ISA has only two types of instructions, Type A (used for register-register instructions) and Type B (used for register-immediate operations), as represented in Figure 3.3. On the other hand, the RISC-V ISA has six different types of instructions, as shown in Figure 2.1. New entries were added inside the decoder to parse the new types of instructions. Each instruction type is identified with a specific opcode since instructions of the same type share the same opcode. Then, after the type has been identified, the specific instruction is discovered using the *funct3* and *funct7* parameters, and the operands are retrieved.

The ALU also required some modifications due to new operations or changes in the way they should be executed. The decision to offer support for the Integer Multiplication and Division extension also required new entries inside the decoder and new operations defined in the Execute control signals. The list of RISC-V supported instructions is available in Table A.1.

For comparison purposes, the MicroBlaze instructions implemented by the MB-Lite softcore are present in Table A.2. It is noticeable the reduction of instructions on the RISC-V ISA when compared to the MicroBlaze. Mainly, the memory requests and branches, since in these operations, RISC-V uses only one register and the immediate as operands, but MicroBlaze also supports two registers.

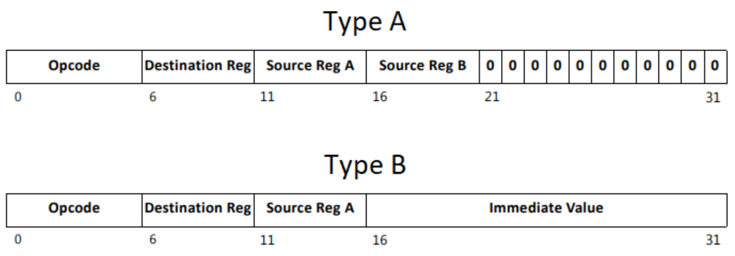


Figure 3.3: MicroBlaze instruction types [31].

3.3 Memory Structure

In a Harvard architecture, instructions and data are stored in different memories. This model allows a simultaneous access to data and instructions, but it is not possible to mix the program and data addressing space, for instance, to load a new program as data before the execution. Consequently, this processor is designed to be used as a slave, because each program needs to be previously transferred before its execution.

As an example, when a C program is compiled, the compilation toolchain produces a binary where the program instructions (also known as text) and the data are stored, originating the memory layout represented in Figure 3.4. Typically, the data part includes the initialized and uninitialized data, the heap, and the stack, requiring a bigger address space when compared with the text part.

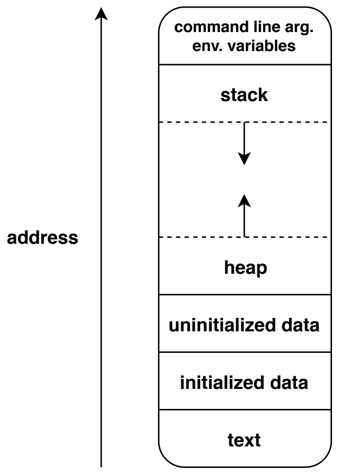


Figure 3.4: Memory layout resulting from the compilation of a C program.

Modern processors use caches to mitigate the trade-off between available storage and access speed. In particular, it is common to see implementations with multiple levels of caches, where the low level is faster but has small capacity, and the up-levels have more capacity and are faster enough regarding the main memories. Multicore processors also benefit from multi-level caches. Typically each core features its non-shared L1 cache because it would increase the latency and the circuit complexity and consequently reduce the performance of each core. In the higher levels, unified caches are desirable since they allow the sharing of data between multiple cores outside the main memory and reduce the complexity of cache coherence protocols. The initial goal was the implementation of two caches, one for instructions and the other for data. Due to time limitations, only one was implemented and the priority was given to the data cache because, as it was said, the data part has tendency to be larger than the instructions part. The connections between the core and both the instruction memory and the data cache are represented in Figure 3.5.

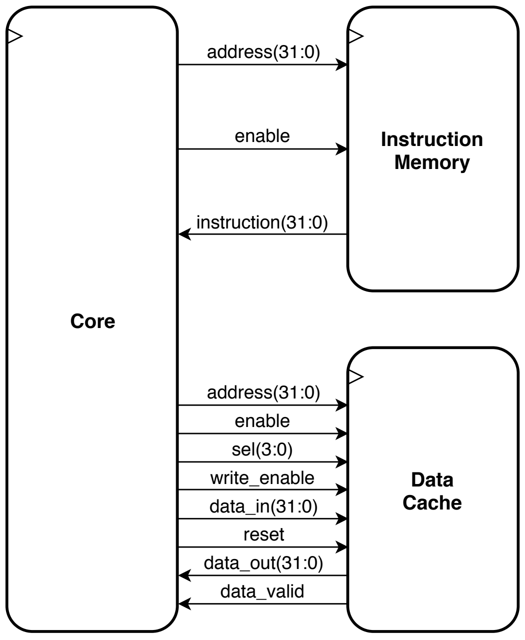


Figure 3.5: Connections between the core and both the instruction memory and the data cache.

To store the program and data on the FPGA, it is required a physical memory that can be BRAMs or other memory types such as Dynamic Random-Access Memories (DRAMs), depending on the FPGA board support. The BRAMs are small capacity memories inside the FPGA chip, used to store data. Multiple BRAMs can be aggregated to create bigger memories and other complex structures such as FIFOs and Dual Port BRAMs with just 1 cycle output latency. However, to store large amounts of data, this solution cannot be applied due to two main constraints: the number of BRAMs inside an FPGA chip is limited, and they are spread across the FPGA, which could originate timing violations with a consequent decrease of the operating frequency when the used BRAMs are too far from each other. On the other hand, the external DRAM provides more storage capacity, but the latency to read and store data is higher. The FPGA board that was used to implement and test this processor supports an external DRAM device, having in total 4 $\times$ 16GB DDR4 DIMMs.

Beyond the instructions and data memories, the processor supports peripherals that are part of the address space. They are accessible via memory instructions, which are handled by the address decoder and forwarded according to the address map. More details about the implementation of peripherals are available in Section 3.6. The default address space represented in Table 3.1 features 16KB for instructions, 256B for peripherals, and nearly 4GB for data, allowing a large configuration space according to the user's preferences.

Table 3.1: Address Map.

Region	Start Address	End Address	Size
Instructions	0x0000 0000	0x0000 3FFF	16KB
Data (cached)	0x0000 4000	0xFFFF FEFF	4GB
Peripherals	0xFFFF FF00	0xFFFF FFFF	256B

3.3.1 Instruction memory

Since the instruction cache was not implemented, the program instructions are stored inside BRAMs. The memory should have one port to receive and send data across the Peripheral Component Interconnect Express (PCIe) interface and a second port connected to the processor to fetch the instructions. Xilinx offers a BRAM generator [32] through the Vivado's IP Catalog, which supports several types of RAMs with configurable sizes. According to the architecture requirements, the True Dual-Port BRAM is the right option because it provides two read and write ports for different addresses simultaneously and two independent clocks. This way, it is possible to connect one of the ports to the receive the data that is sent via PCIe, but due to implementation requirements, this interface must be compatible with AXI4. So, we need to use the Xilinx's AXI Block RAM Controller IP [33] to make the conversion between both interfaces, because BRAMs are not compatible with AXI.

The connections between the AXI Block RAM Controller and the instruction memory, implemented using a True Dual-Port BRAM, are represented in Figure 3.6, where is also represented the core connected to the other port. It is represented the two independent clock signals, one from the PCIe logic and the dedicated core clock.

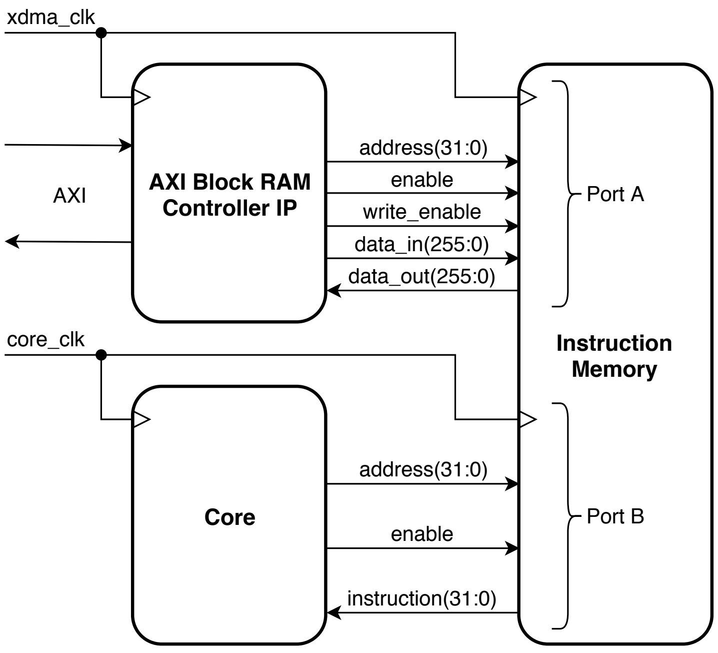


Figure 3.6: Connections between the Core, the Instruction Memory (implemented with a True-Dual Port RAM IP), and the AXI Block RAM Controller IP.

The BRAM Port A has data input and output ports of 256 bits, corresponding to 8 instructions received simultaneously from the AXI BRAM Controller. On the other hand, Port B data is 32-bit wide because the processor only needs one instruction each clock cycle. The depth of each port is defined according to the corresponding line width and total memory size. When the user decides to change the processor's instruction memory size, the BRAM depth should be updated in both ports, as well as the mapping of the AXI BRAM Controller on the AXI bus, to be compliant with the Address Map of Table 3.1.

3.3.2 Data memory and cache implementation

The adopted cache uses an existing implementation as its base structure, requiring several modifications to support 32-bit words, lines with a larger dimension, a configurable number of lines, and data invalidation when the processor is reset. The cache can be configured to use two different configurations: direct-mapped or 2-way set associative with the Least Recently Used (LRU) data replacement policy. Both types use the Write-Back policy, where the date is only written to the main memory when the cache line needs to be replaced by a new loaded line. This write policy allows reducing the number of memory accesses when compared to the Write-Through policy, since this last policy keeps the memory always updated. Figure 3.7 shows the structure of the default cache with 2 ways of associativity and 16 lines, each with 256 bits, featuring one valid and one dirty bit per line. This cache has a total capacity of 1KB, corresponding to 256 32-bit words.

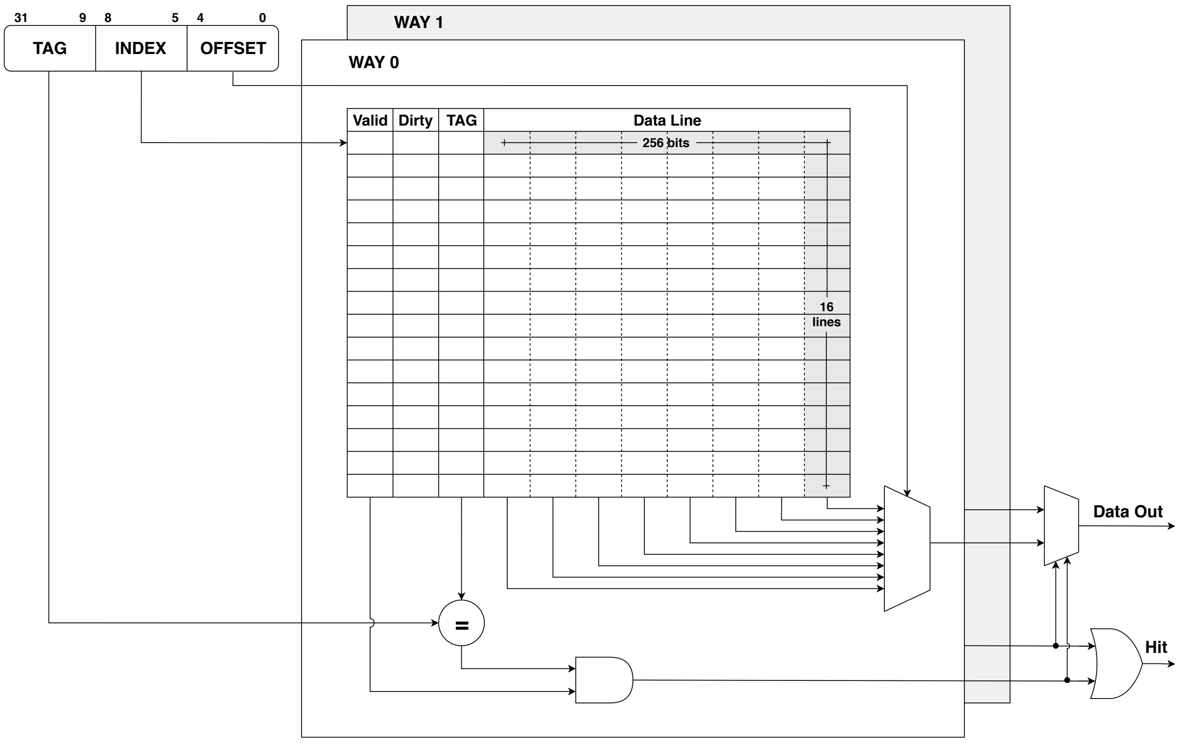


Figure 3.7: 2-way set associative cache structure. Each way is comprised of 16 lines with 256 bits each, the tag, a valid bit, and a dirty bit which indicates if the line was changed and it needs to be updated on the main memory before it is replaced.

Depending on the program type and the amount of data needed in the beginning and during its execution, the main data memory could require a storage space higher than the available BRAMs can provide. The solution goes through the use of DRAMs present on the board to store the processor's execution data.

The connection between the design implemented on the FPGA and the DRAM is made through the Xilinx Memory Interface Generator (MIG) [34]. This IP is responsible for generating memory controllers and interfaces, simplifying the design process by providing the constraints and Verilog or VHDL design implementations automatically. The memory controller accepts user transactions through a native interface or AXI4 Slave interface, which are converted and sent to the physical layer internally. The MIG was configured to include the AXI4 Slave interface instead of the native interface to transfer data between the PCIe endpoint and the DDR4 memory, since both IPs are compatible with this protocol. Its AXI base address was mapped according to the processor's address map.

To connect the cache and the MIG, an AXI4 Master [35, 36] interface for the cache was developed. This interface is primarily based on the Finite State Machine present in Figure 3.8 that reads the cache output signals to detect new requests and makes the conversion to the AXI4 protocol. Until the cache sends a read or write request, the state machine is kept on the IDLE state. When the request is made, the AXI transaction is initialized by sending the memory address (WRITE ADDR and READ ADDR states). As soon as the slave sets the valid signal to high, the next state is changed to READ or WRITE, depending on the initial cache request. The read request finishes when the interface receives valid data, and the cache input valid signal is set to high. The write request has an additional state (WRITE RESP) used to receive the slave valid write response signal before the cache receives the valid signal, which indicates the end of the write transaction.

The introduction of DRAMs as data memory solves the available storage limitation but increases the latency for every access by dozens of clock cycles. This is a consequence of the DRAM internal latency and the delays created by the AXI interconnections. The minimum latencies introduced by both the cache and the main memory are 22 clock cycles in case of a Read Miss or a Write Miss, 3 clock cycles for a Write Hit, and finally, 2 clock cycles when a Read Hit occurs.

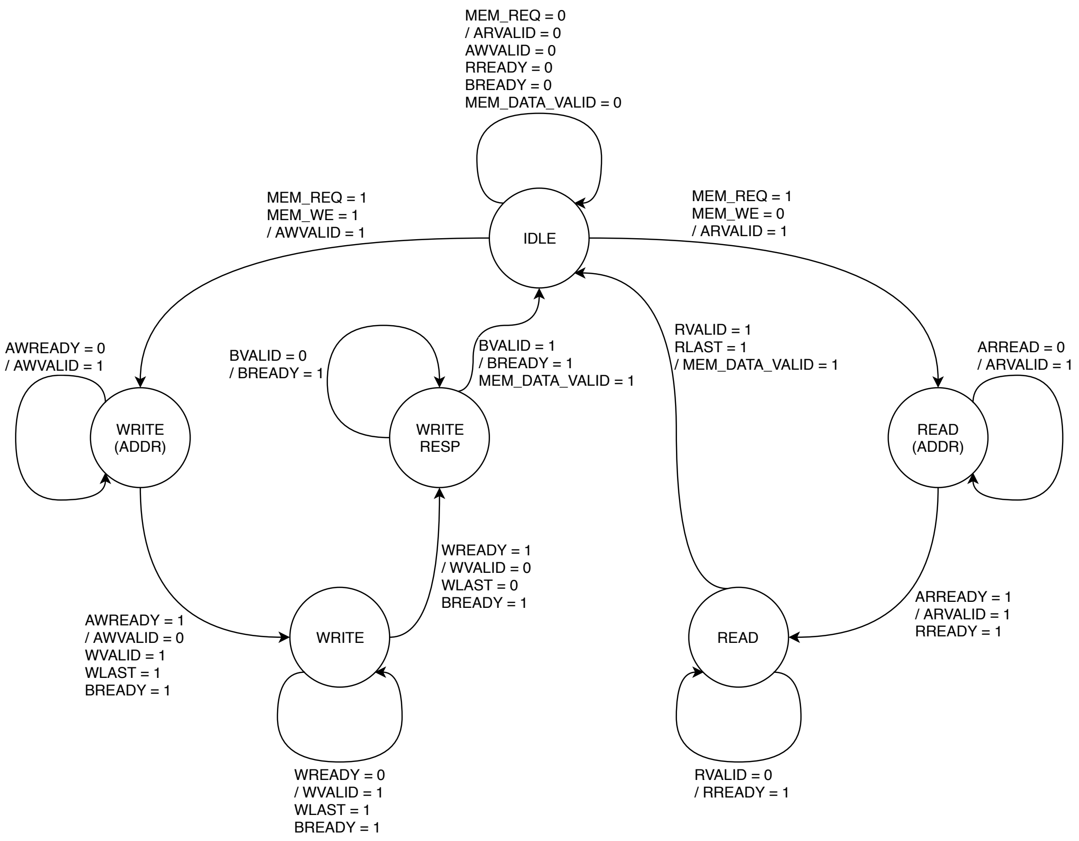


Figure 3.8: Cache AXI4 master interface Finite State Machine. This state machine is responsible to control the interface that translates the cache read and write requests to AXI4 and vice-versa.

By using the Xilinx IP Integrator, a specific IP was created to make the process of integrating the cache easier. By making the cache independent of the core, the user has more control over its implementation and faces fewer difficulties if he decides to add his cache or memory instead of what was originally available. By observing the IP in Figure 3.9, one can identify the inputs corresponding to the control signals from the core, and as outputs, the AXI4 master interface to be connected to the MIG, and the data and valid signals that are connected to the core.

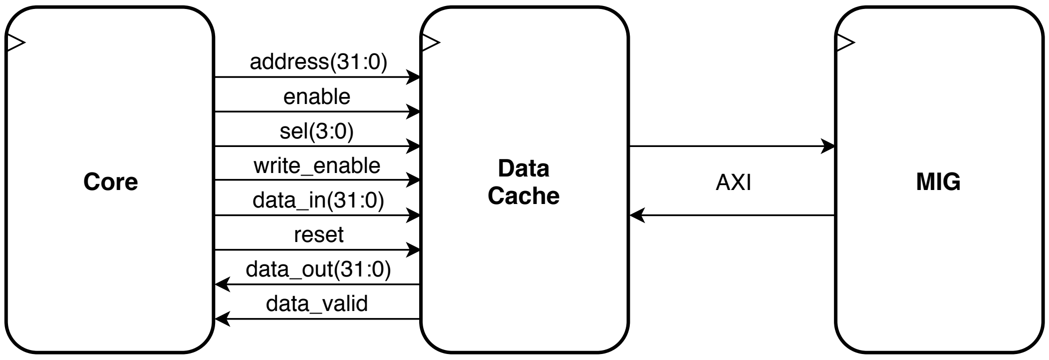


Figure 3.9: Connections between the Core, the Data Cache, and the MIG.

3.4 Multi-cycle Functional Units

With the addition of support for the RISC-V Integer Multiplication and Division extension, the core requires the implementation of a multiplier and a divider. Typically the multiplication and division are too complex to be executed on a single clock cycle without reducing the frequency. The multi-cycle functional units solve this problem because they need several clock cycles to generate the result. By using the Xilinx IPs, a pipelined multiplier unit and a pipelined divider with configurable latency were developed to be implemented in the Execute stage in parallel with the ALU. The multiplier IP is configured with a latency of 6 clock cycles, and the divider latency was set to 30 clock cycles. It is higher due to the complexity of the division algorithms.

3.4.1 Integer Multiplier

The integer multiplier unit developed for this processor includes a multi-cycle pipelined Xilinx IP multiplier [37]. The IP settings allow the configuration of the type of the operands (signed or unsigned), its sizes, and the latency required to compute the result. The RISC-V multiplication instructions (see Table 2.2) use 32-bit operands, signed or unsigned. Since the multiplier unit must support both types of operands, but the IP, after generated, only support one type, it was decided to define the operands always as signed. Then, by using an extra bit, the sign can be extended when the operand is signed, but otherwise it has the value zero. This selection is made by the control signals a_sign and b_sign, which are dependent on the operation type. Thus, the multiplier IP inputs are 33-bit wide, and the output result is 66-bit wide (2×33 bits), but the two most significant bits are discarded. According to the multiplication instruction, the output value can be the lower or the higher 32 bits of the result.

There is a control signal (low_high) that chooses the desired part. Figure 3.10 shows the structure of this unit and Table 3.2 shows the control signals for each supported multiplication instruction.

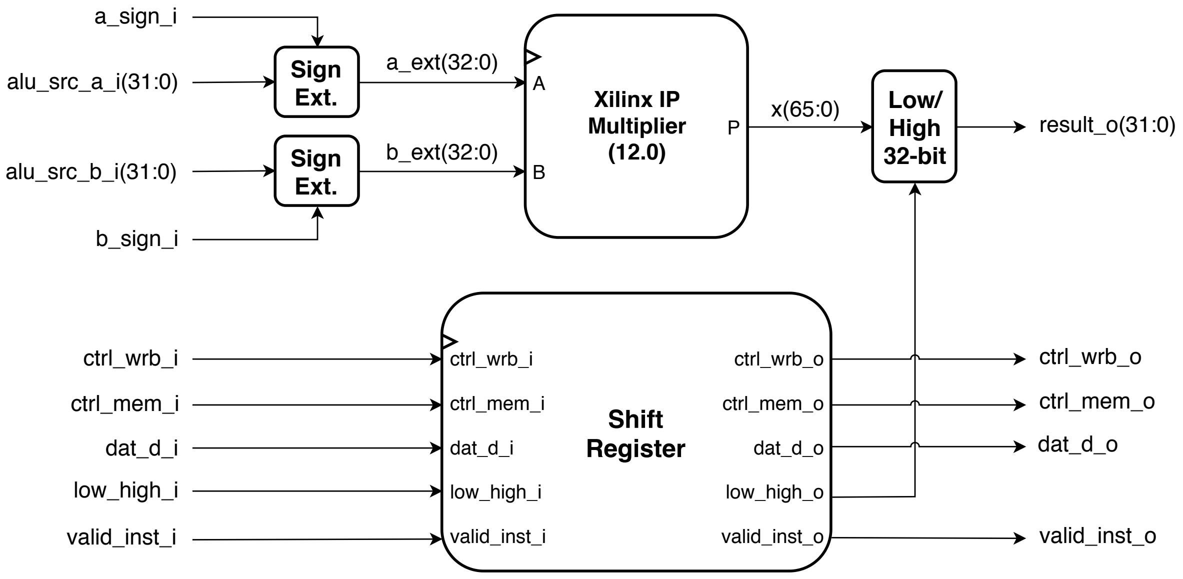


Figure 3.10: The Multiplier Unit. This unit uses a Xilinx IP to execute the multiplication operations. There is always a signal extension in the operands because the multiplier is configured to do sign operations. When the instruction corresponds to an unsigned multiplication, the extension bit is zero. The output value can be the upper or the lower 32 bits of the result, and the signal low_high is used to select the correct part. Since the multiplier is pipelined, it is necessary to have a shift register to store the control signals of each operation.

Table 3.2: Control signals for each supported multiplication instruction.

Instruction	**Control signals**		
	a_sign	**b_sign**	**low_high**
MUL	0	0	0
MULH	1	1	1
MULHSU	1	0	1
MULHU	0	0	1

Each instruction that arrives in the Execution stage is comprised of different control signals, which are needed for the following pipeline stages. Since the multiplier is pipelined, it can simultaneously execute a number of multiplications equal to its latency, so this unit must be capable of storing all the control signals from each instruction. To do that, the unit uses a shift register with the multiplier's latency. To know when the multiplier output is valid, there is a valid signal (named valid_inst) that is set to high when a valid operation arrives, and due to the fixed latency, the valid signal is at the exit of the shift register when the valid result is in the multiplier's output.

3.4.2 Integer Divider

The development of the integer divider unit, represented in Figure 3.11, followed the same principles of the multiplier. This time it was used a multi-cycle Xilinx IP divider, with different division implementations: LutMult, Radix-2, and High Radix. According to the IP's manual [38], the LutMult solution limits the dimension of the dividend and the divisor to 17 and 12 bits, respectively; the High Radix only supports fractional remainders; the Radix-2 allows dividends and divisors up to 64 bits, fractional and integer remainders and it is recommended for applications which require high throughput. Therefore, Radix-2 is the only solution compliant with the ISA requirements.

The divider IP always computes the quotient and the remainder, and both are available in its outputs. Depending on the instruction, the desired result can be the quotient or the remainder. This selection is made by the signal div_rem, generated in the Decode stage. According to the C99 standard [39], the quotient is rounded towards zero, while the remainder's signal is equal to the dividend's. The control signals used to select the signs of the operands or the desired result, for each supported instruction, are described in Table 3.3.

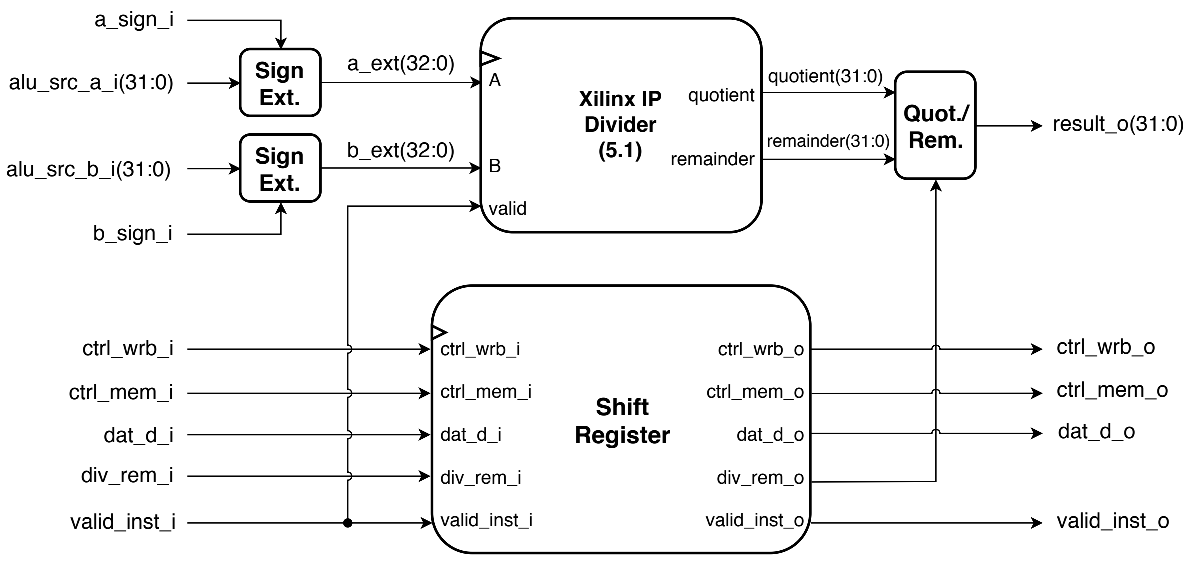


Figure 3.11: The Divider Unit. This unit supports the division of 32-bit integer signed/unsigned operands, and its output can provide the remainder or the quotient. It uses a multi-cycle pipelined divider from Xilinx with a latency of 30 cycles, and it is configured to use the Radix-2 division algorithm. A shift register with the same latency is used to store the control signals during the division operations.

Table 3.3: Control signals for each supported division instruction.

Instruction	**Control signals**		
	a_sign	**b_sign**	**div_rem**
DIV	1	1	0
DIVU	0	0	0
REM	1	1	1
REMU	0	0	1

3.5 Hazards

The modifications made in the processor's pipeline, specifically the introduction of a data cache and the multi-cycle functional units, caused some new hazards that had to be solved. This section studies the various types of hazards originated by those changes, and it also explains how they were solved. Data hazards are the first type to be considered, followed by Control hazards, and lastly, the Structural hazards. At the end of this section, it is presented a scoreboard unit that was developed to detect and solve hazards during the processor execution.

3.5.1 Data Hazards

Data hazards occur when the dependencies of operands between instructions are not respected, and eventually, some of them will use wrong operand values. The pipeline architecture can introduce these hazards when instructions with dependencies between them are too close, and some of them need operands computed by the previous instructions that are not written yet to the Register File (Read after Write). The solution for this problem is the introduction of stall cycles, and the usage of forwarding paths like the original MB-Lite does. There are three types of data hazards, being classified as follows:

- Read after Write (RAW): It occurs when an instruction requires an operand written by a previous instruction but it is not available yet on the RF.
- Write after Write (WAW): It occurs when the instructions arrives at the Write-Back stage out-of-order or allows multiple simultaneously writes.
- Write after Read (WAR): It occurs when an instruction is writing to its destination register before a previous instruction has read its operands from the same register, receiving an incorrect value. This type of data hazards does not occur on processors with in-order instructions issue.

The introduction of multiple functional units with different latencies, working in parallel in the Execution stage, may result in the instructions leaving that stage out-of-order, causing WAW hazards. This situation can also originate RAW hazards when an instruction that is entering the EX stage uses operands that were not written yet because a previous instruction is still inside the EX stage. The memory load instructions can also originate RAW hazards since the instructions can be several clock cycles waiting for the requested value. To identify and solve these situations, a dedicated unit was developed and implemented inside the ID stage, responsible for storing information about each instruction that leaves the ID stage and uses the multi-cycle functional units or accesses the memory. These hazards are detected and solved by the scoreboard unit, that is going to be presented later.

3.5.2 Control Hazards

Control hazards appear as a result of not knowing which instruction should be executed when a conditional branch or jump instructions enter the pipeline, considering that the processor needs to continue fetching instructions. The original MB-Lite has its branch control unit in the Execution stage, and it uses a static not taken branch prediction technique. This approach considers that no branch is taken, and the processor continues to fetch the instructions that follow the branch or jump. If the prediction was correct, the execution continues, but if it was incorrect, the instructions that meanwhile entered into the Fetch and Decode stages are cleared, and the correct target instruction is fetched.

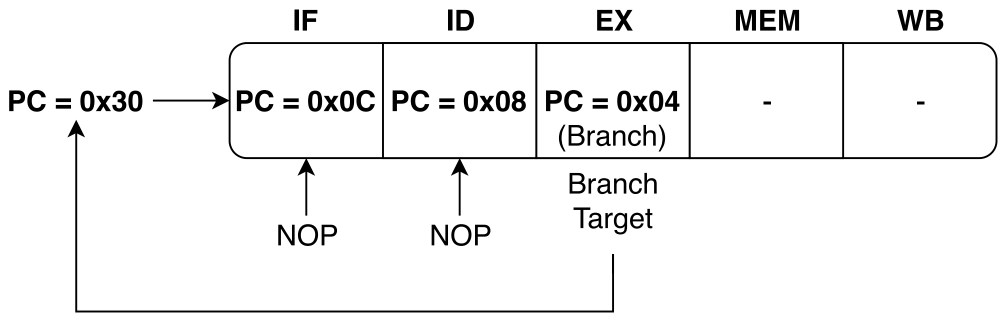


Figure 3.12: Example of an incorrect branch prediction, where the instructions that entered into the pipeline are removed and the correct program counter is fetched.

3.5.3 Structural Hazards

Structural hazards are caused by hardware limitations, preventing the correct execution of the instructions. With the modifications implemented on this processor, some structural hazards may appear inside the EX stage, whenever more than one functional unit finishes the execution on the same cycle. Consequently, this stage can only output one instruction per cycle. The solution to this problem requires knowing the position of each instruction inside each functional unit and introduce stalls in specific cycles, instead of issuing the instruction. By using the scoreboard mechanism explained in the next sub-section, the processor is capable of predicting and solving these hazards based on the occupancy of each unit and its known latency.

3.5.4 Dependencies Handling

As it was previous referred, the need to have a control mechanism capable of identifying and handling the dependencies between instructions and the usage of multiple functional units arise. The considered unit was inspired by the Scoreboard concept, introduced in the CDC 6600 computer [40], in 1963. In that computer, the Scoreboard was responsible for monitoring each instruction dependencies and the available hardware resources in order to solve WAW, RAW, and structural hazards. Due to the similarities behind the goal of both units, the name “Scoreboard” was also used in this unit.

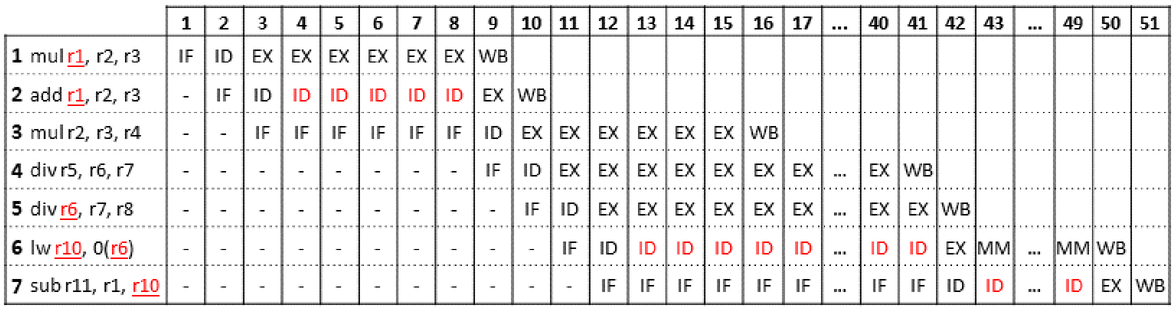
Inside this unit, represented in Figure 3.13, there are 3 main blocks. The top one is a set of shift registers responsible for storing the expected latency of each register operand when an instruction inside a functional unit makes use of that same register as its destination. The verification of dependencies occurs as soon as the instruction enters the ID stage. By decoding its used registers identification, it is verified if the corresponding position on the shift register is set to ’1’. If so, the scoreboard indicates that a hazard exists on its output and the processor stalls. For memory request instructions, there is a specific bit that is set to ’1’ when a load instruction is issued. Consequently, instructions dependent on that register are stalled. As soon as the load is finished, that bit is cleared, and instructions waiting on the ID stage are ready to be issued if no other hazards exist.

The other 2 blocks of the scoreboard (see Figure 3.13) are responsible for storing the position of each instruction inside the multiplier and the divider. It is necessary to provide the ID stage with information about when it should stall, in order to avoid structural hazards, which happen when one of these units has an instruction in its penultimate stage, and the instruction about to be issued uses the ALU. Since the ALU execution only takes 1 clock cycle, a collision in the EX output would occur. Another situation occurs when the instruction present on the ID stage requires the multiplier, but inside the divider there is an instruction in a position of its meta-pipeline that would finish simultaneously. The solution goes through the introduction of a 1 cycle stall instead of issuing the instruction.

To demonstrate the operation of this unit, Figure 3.14 depicts a simple program that contains 3 data hazards. In this case, we considered a latency of 6 clock cycles for the multiplier and 30 for the divider. In the second instruction of this program there is a WAW hazard on register R1 because the first instruction writes to the same register. But despite being issued first, the mul instruction stays 6 cycles inside the multiplier. To solve this hazard, the processor stalls until instruction 1 leaves the EX stage. Figure 3.15 represents the values that are written to the scoreboard during cycle 3, corresponding to the latency of R1 (6 cycles) and the entrance of the instruction on the multiplier.

In the sixth instruction, it is observed a RAW data hazard, since it uses as operand the register R6, which is the destination register of the previous division operation. Once again, the processor stalls. In Figure 3.16, (cycle 12) it is possible to observe the scoreboard content resulting from the execution of two divisions (instructions 4 and 5) and multiplication (instruction 3). The R1’s shift-register is empty since the multiplication of instruction 1 has already finished (at cycle 8).

After cycle 41, the load instruction is finally free to continue across the pipeline. As such, the Memory field of register R10 is set to ’1’ until the hit signal is received (Figure 3.17). Consequently, instruction 7 should wait to avoid a RAW hazard. Finally, at cycle 50, the dependency is solved, so the execution continues, and the Memory bit of R10 is set to ’0’, as shown in Figure 3.18.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	40	41	42	43	...	49	50	51
1 mul r1, r2, r3	IF	ID	EX	EX	EX	EX	EX	EX	WB																	
2 add r1, r2, r3	-	IF	ID	ID	ID	ID	ID	ID	EX	WB																
3 mul r2, r3, r4	-	-	IF	IF	IF	IF	IF	IF	ID	EX	EX	EX	EX	EX	EX	WB										
4 div r5, r6, r7	-	-	-	-	-	-	-	-	IF	ID	EX	EX	EX	EX	EX	EX	EX	...	EX	WB						
5 div r6, r7, r8	-	-	-	-	-	-	-	-	-	IF	ID	EX	EX	EX	EX	EX	EX	...	EX	EX	WB					
6 lw r10, 0(r6)	-	-	-	-	-	-	-	-	-	-	IF	ID	ID	ID	ID	ID	ID	...	ID	ID	EX	MM	...	MM	WB	
7 sub r11, r1, r10	-	-	-	-	-	-	-	-	-	-	-	IF	IF	IF	IF	IF	IF	...	IF	IF	ID	ID	...	ID	EX	WB

Figure 3.14: Example of dependencies handling. For each instruction is shown its execution stage over time. On the first column, the registers involved in data hazards are underlined and written in red, as well as the cycles when the instruction stalls on the ID stage due to dependency conflicts.

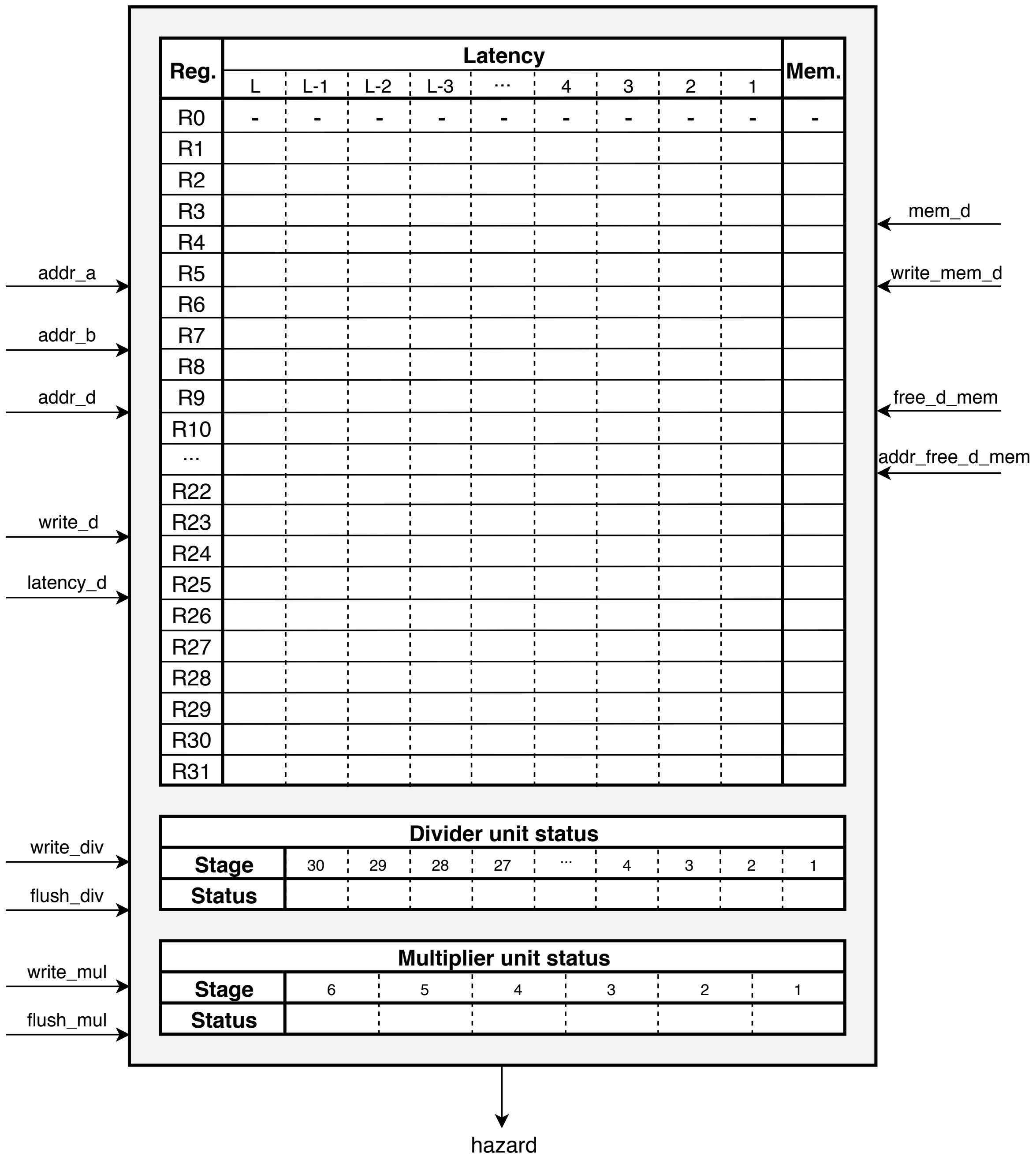
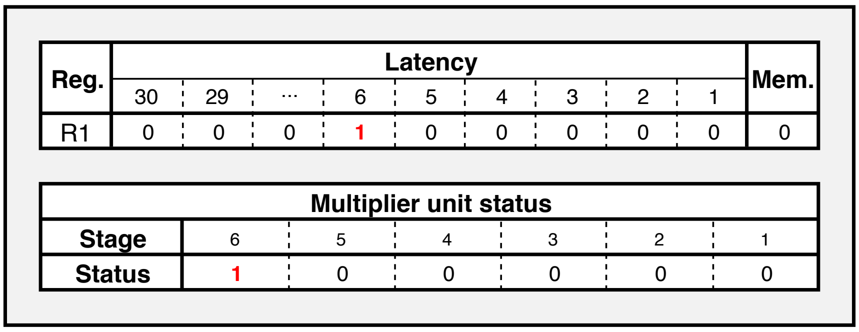


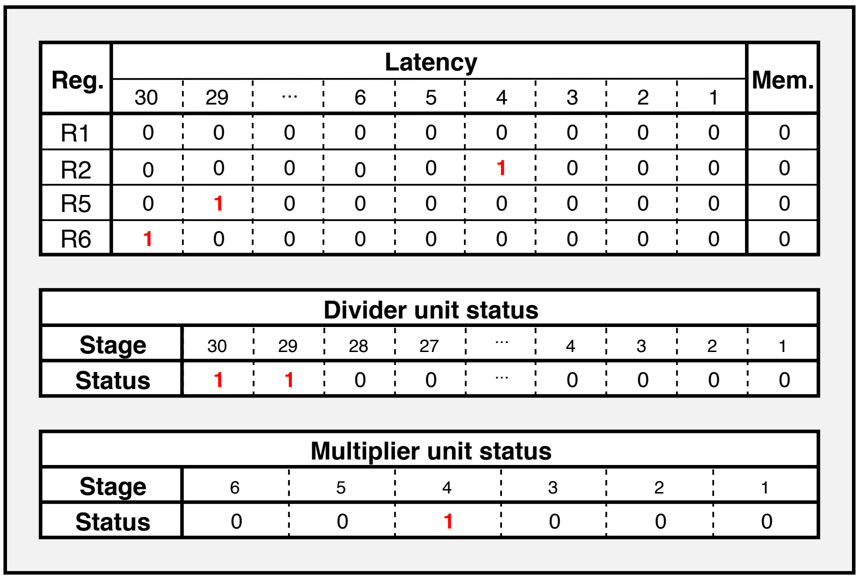
Figure 3.13: Representation of the scoreboard unit. It is responsible for informing the ID stage when data and structural hazards occur, by marking the destination registers in use and storing the state of each multi-cycle functional unit.



Reg.	Latency									Mem.
	30	29	···	6	5	4	3	2	1	
R1	0	0	0	1	0	0	0	0	0	0

Multiplier unit status						
Stage	6	5	4	3	2	1
Status	1	0	0	0	0	0

Figure 3.15: Scoreboard state at clock cycle 3. Register R1 is marked on the scoreboard with the multiplication latency. The position of this instruction inside the multiplier is also stored.

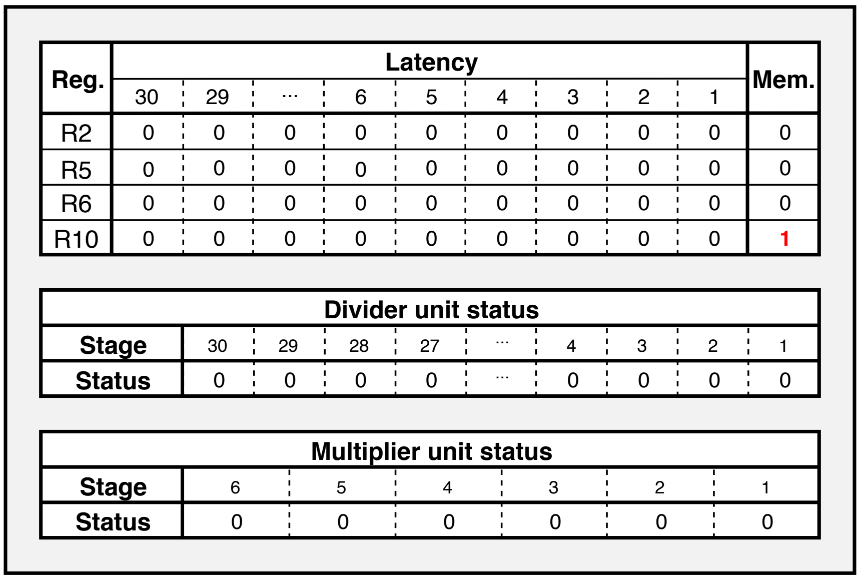


Reg.	Latency									Mem.
	30	29	···	6	5	4	3	2	1	
R1	0	0	0	0	0	0	0	0	0	0
R2	0	0	0	0	0	1	0	0	0	0
R5	0	1	0	0	0	0	0	0	0	0
R6	1	0	0	0	0	0	0	0	0	0

Divider unit status								
Stage	30	29	28	27	···	4	3	2	1
Status	1	1	0	0	···	0	0	0	0

Multiplier unit status						
Stage	6	5	4	3	2	1
Status	0	0	1	0	0	0

Figure 3.16: Scoreboard state at clock cycle 12. At this moment, the dependency of R1 was already solved. Instructions 3, 4, and 5 are being processed at the EX stage, so the destination registers were marked, as well as the corresponding functional units.

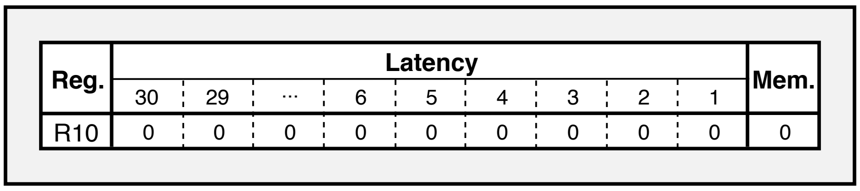


Reg.	Latency									Mem.
	30	29	···	6	5	4	3	2	1	
R2	0	0	0	0	0	0	0	0	0	0
R5	0	0	0	0	0	0	0	0	0	0
R6	0	0	0	0	0	0	0	0	0	0
R10	0	0	0	0	0	0	0	0	0	1

Divider unit status									
Stage	30	29	28	27	···	4	3	2	1
Status	0	0	0	0	···	0	0	0	0

Multiplier unit status						
Stage	6	5	4	3	2	1
Status	0	0	0	0	0	0

Figure 3.17: Scoreboard state at clock cycle 42. The divider and the multiplier units are empty, so none of the registers are marked with latencies. Instruction 6 left the ID stage and because it is a load request, its destination register, R10, has its memory bit on scoreboard set to '1'.



Reg.	Latency									Mem.
	30	29	···	6	5	4	3	2	1	
R10	0	0	0	0	0	0	0	0	0	0

Figure 3.18: Scoreboard state at clock cycle 50. The memory load request received a hit on the previous cycle, so the mark on the scoreboard relatively to the register R10 was removed.

3.6 Peripherals

Peripherals are used to expand the processors' capabilities to interact with its external environment, according to the application's requirements. This section presents three memory-mapped peripherals that were developed for this processor and which are compatible with the MB-Lite address decoder, as shown in Figure 2.10. The first peripheral is a UART module, to be used as standard input and output, together with a set of software functions. The remaining peripherals are a cycle counter and a timer, useful to measure time on programs and evaluate the processor's performance. It is also explained how other peripherals can be added and the requirements they should meet.

3.6.1 UART for Standard Input/Output

UART is an asynchronous serial communication peripheral that uses only two independent lines to transmit (TX) and receive (RX) data. Unlike other protocols, the supported serial communication protocol does not need any clock signals because both devices should be configured to use the same transmission rate (or baud rate) and the same data format. The most common baud rates are 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, and 115200 bps. The UART is used for low distance and slow communications, and almost all the available microcontrollers support it to connect peripherals, for debugging or for standard input and output communication.

The serial data transmission, represented in Figure 3.19, begins with the start bit, which usually has the logic value low because the lines are typically high during the idle state. This bit is essential for the receiver to detect the incoming data and to synchronize its internal clock. The start bit is followed by the 8 bits of data, sequentially. After the last data bit, there is an optional parity bit used for error detection. Finally, the transmission ends with at least one stop bit, which has the logic value high, and the line is kept high until a new transmission starts.

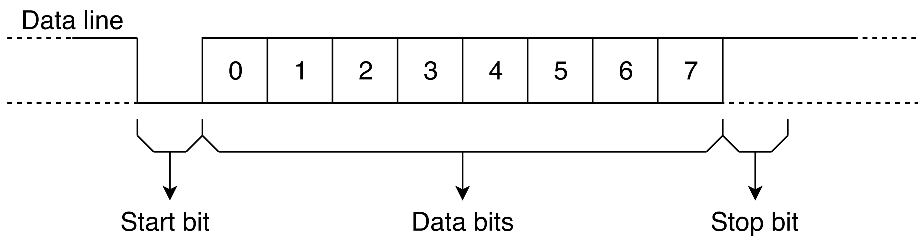


Figure 3.19: Representation of a serial transmission.

Since the processor's operating frequency is usually higher than the transmission rate, when the processor wants to write a string to the TX line, it cannot be waiting that each byte transmission ends to write the next byte. The solution for that is the use of a FIFO to store the bytes until they are sent. To know when the FIFO is full, there is a signal named FIFO_TX_FULL with that indication mapped in memory, which can be read by the processor to stop the next transmission and wait. In the same way, there is a signal to indicate when the FIFO is empty so that the UART TX module knows when there is nothing to transmit (FIFO_TX_EMPTY). On the reception side, when the processor is waiting for a string, it should wait for each byte so it can only read the RX line when the reception FIFO has content, and this is done by reading the address correspondent to the FIFO_RX_EMPTY signal. The architecture of the considered UART module is present in Figure 3.20. The Address Decoder is responsible for controlling which FIFO signal is going to be read or written, according to the memory map present in Table 3.4. The TX and RX modules, available on [41], are responsible for transmitting and receiving the bits from the FPGA's lines, and the TX Controller initiates the transmission when the TX FIFO has content, by setting the TX_VALID signal to high, following the finite state machine present in Figure 3.21.

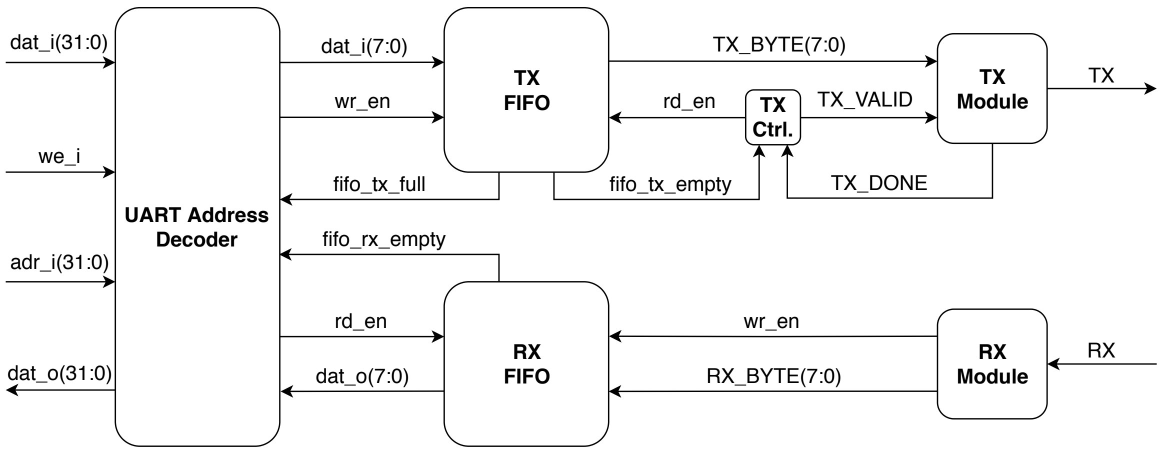


Figure 3.20: UART module. The FPGA's TX and RX signals are connected to this module which is comprised of the UART Address Decoder, the TX and RX FIFOs, the TX Controller, and the TX and RX modules.

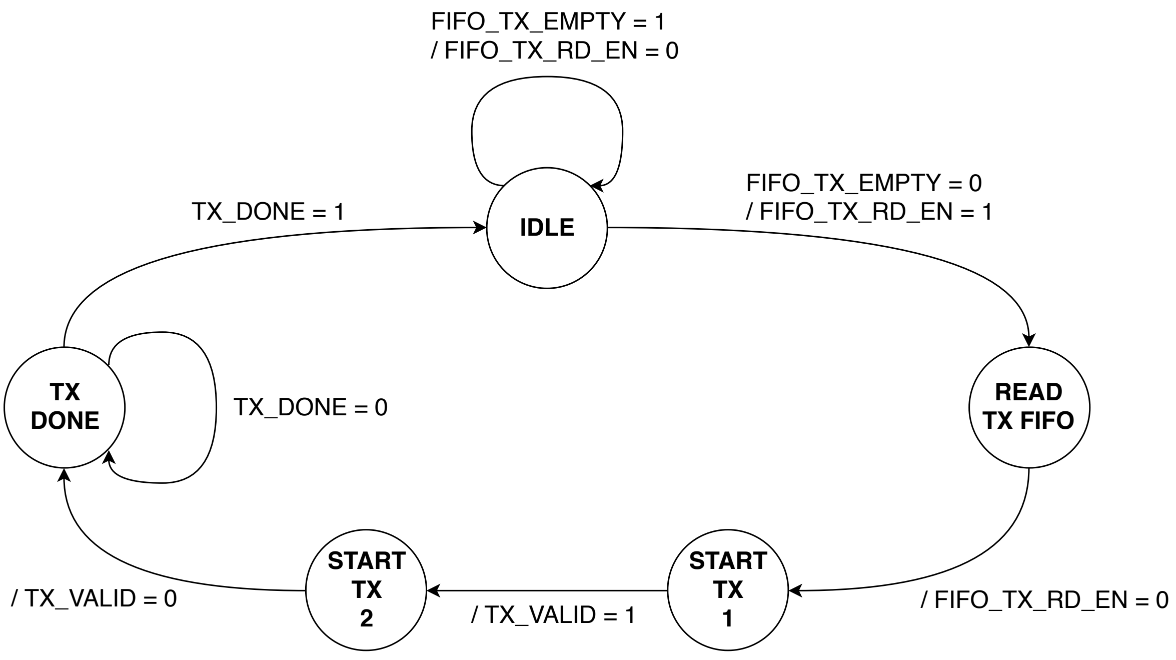


Figure 3.21: TX Controller Finite State Machine. The control waits for some content stored in the TX FIFO by checking the signal FIFO_TX_EMPTY in the IDLE state. When the FIFO is not empty, the controller sets the FIFO_TX_RD_EN to high to get the next byte to be sent. Then, the TX_VALID signal is enabled and the transmission of each bit starts in the TX module. The controller is kept in the TX_DONE stage until the TX_DONE signal is set to high.

To simplify the use of the UART module for standard input and output operations through the serial line, it is available a library with some implementation of the standard C stdio library adapted for this processor, in particular the following functions: putchar(), printf(), puts(), getchar(), and gets(). Those functions make use of the UART Module Memory Map (Table 3.4). A detailed explanation about those functions is available in Chapter 4.

Table 3.4: UART Module Address Map. These are the addresses where the input and output data signals are mapped as well as the FIFO's status signals. The processor can read the status of each FIFO to know if there is something to read from the RX FIFO or if the TX FIFO has free space.

Name	Address	Type	Description
RX Empty	0xFFFF FFF0	Read	Indicates the status of the RX FIFO. It holds the value '1' if it is empty or '0' if not.
TX Full	0xFFFF FFF4	Read	Indicates the status of the TX FIFO. It holds the value '1' if it is full or '0' if not.
RX Data	0xFFFF FFF8	Read	Address where the received bytes can be read.
TX Data	0xFFFF FFFC	Write	Address where the output bytes should be written.

3.6.2 Cycle Counter

Counters are used in processors to count events or the number of clock cycles, and they are very usefully for basic performance analysis allowing the comparison of benchmarks across different processors with different operating frequencies. The RISC-V manual reserves in the Control and Status Registers Extension some unprivileged read-only CSR registers for performance counters and timers. Since this implementation does not offer support for the Zicsr extension, the counter is implemented as a memory-mapped peripheral, being accessed with memory requests instructions instead of using the CSR instructions included in the Zicsr extension.

To tackle this need, it was developed a 64-bit cycle counter compatible with the same memory interface as the other peripherals. The counter features a 64-bit accumulator that is always counting when the processor is enabled, and it is reset when the processor is also reset. The choice of a 64-bit counter (instead of a 32-bit) is to reduce the chance of a overflow while the count is being considered. According to Table 3.5, the user accesses to the 64 bits (separately) using two 32-bit reads. The selection of the output part is easily made internally using the input address, as represented in Figure 3.22.

To obtain the 64-bit result, first, the user should read the upper 32 bits; then, the lower 32 bits; finally, the upper 32 bits again to compare with the first value to detect if any overflow occurred. If both values are different, the process should be repeated. By following this process, it is possible to get the 64-bit result using 32-bit instructions. In Chapter 4, it is presented a list of functions that can be used in C programs to access the cycle counter.

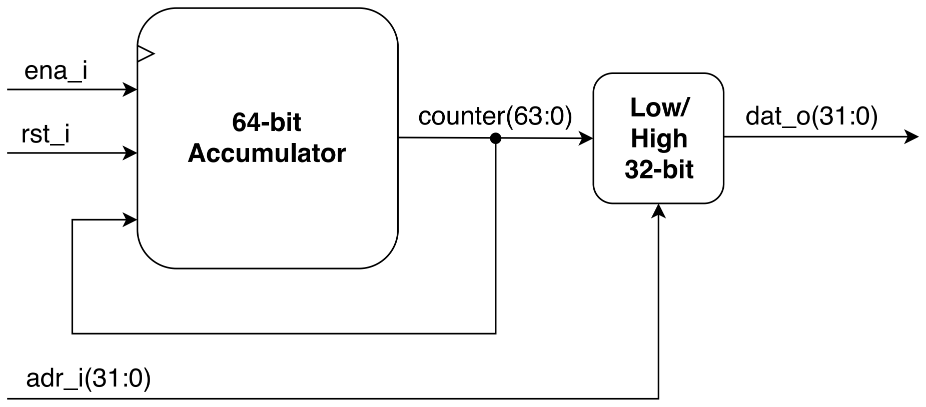


Figure 3.22: The cycle counter block diagram. This 64-bit counter uses an accumulator and it is compatible with the processor's memory interface. The selection of the required result part is made by the input address.

Table 3.5: Cycle Counter Address Map. The 64-bit value should be obtained by reading the upper and the lower 32 bits individually.

Name	Address	Type	Description
counter[31:0]	0xFFFF FF00	Read	Location of the counter's lower 32 bits.
counter[63:32]	0xFFFF FF04	Read	Location of the counter's higher 32 bits.

3.6.3 Timer

Timers are used to measure how much time a specific task needs to be executed, being a valuable tool to evaluate the processor's performance. Its hardware is the same used in the cycle counter, but it requires a dedicated clock with a well-known frequency instead of the clock used in the processor. Despite being set by the user before the synthesis and implementation, the processor's clock can suffer minor changes during the synthesis process, which will compromise the precision of the results, since the time results are calculated using the relation between the number of clocks counted by the accumulator and the frequency.

The timer range and resolution depend on the clock frequency. Increasing the frequency results in a higher resolution, but on the other hand, a higher frequency limits the total time that can be measured. The accumulator width is also an important parameter to calculate the range. As an example, using a 64-bit accumulator and a frequency of 500MHz, the range value is more than 1100 years.

Focusing specifically on the FPGA used for the current implementation, there are four 300 MHz clock sources available, all of them implemented with differential pairs. The user must select one of the free clocks and, using a clock generator, connect it to the timer clock input. The output frequency can be defined according to his preferences.

The process that is used to access the timer is the same that was explained for the counter cycle. To get a time value, two reads at different addresses are needed, according to the address map of Table 3.6. On Chapter 4, the implemented functions to help the user to access the timer and to obtain an absolute time value are presented.

Table 3.6: Timer Address Map. The 64-bit value should be obtained by reading the upper and the lower 32 bits individually.

Name	Address	Type	Description
timer[31:0]	0xFFFF FF80	Read	Location of the timer's lower 32 bits.
timer[63:32]	0xFFFF FF84	Read	Location of the timer's higher 32 bits.

3.6.4 Support for more memory-mapped peripherals

In the development of this processor, it was considered the addition of other memory-mapped peripherals. The peripheral should follow the VHDL interface despicted in Listing 3.1, to be compatible with the MB-Lite Address Decoder. Beyond that, it must follow a set of conditions, as follows:

- When the peripheral has valid data in its output (**dat_o**), the signal **ena_o** should be set to high to inform the processor that valid data is available to be read on the MEM stage.
- The input and output signals must be sensitive to the processor's clock (**clk_i**).
- The **clk_i**, **adr_i** and **ena_i** signals are mandatory. The signals **wr_i**, **dat_o** and **ena_o** are mandatory only if the peripheral supports the read mode. The **rst_i** signal is optional.
- Extra input and output ports can be added according to the peripheral needs.

The number of peripherals and the base address of each one is defined through the constants CFG_NUM_SLAVES and CFG_MEMORY_MAP present on the configuration file "cfg_Pkg.vhd", as shown in the example present in Listing 3.2.

Listing 3.1: VHDL compatible peripheral interface.

```vhdl
entity peripheral is
  port (
    dat_o    : out std_logic_vector(31 downto 0);
    ena_o    : out std_logic;
    dat_i    : in  std_logic_vector(31 downto 0);
    adr_i    : in  std_logic_vector(31 downto 0);
    we_i     : in  std_logic;
    clk_i    : in  std_logic;
    rst_i    : in  std_logic;
    ena_i    : in  std_logic
  );
end peripheral;
```

Listing 3.2: VHDL peripherals address space definition.

```
constant CFG_NUM_SLAVES : positive := 3;
constant CFG_MEMORY_MAP : memory_map_type(0 to CFG_NUM_SLAVES+1) := (
  X"00000000", --Initial address
  X"FFFFFF00", --Cycle counter base address
  X"FFFFFF80", --Timer base address
  X"FFFFFFF0", --UART base address
  X"FFFFFFFF"  --Final address
);
```

3.7 Data Transfer (PCI Express)

The PCIe interface is a high-speed serial data transfer standard bus created by Dell, HP, IBM, and Intel in 2003, featuring a shared address space, with data and control lines communication in packets. This interface is currently the standard to connect peripherals on motherboards, such as graphic cards, network cards, accelerators, and hard drives, due to its high-bandwidth.

The PCIe link of a device can have different widths, also called lanes. A lane corresponds to two differential pairs, one to receive and the other to transmit, providing a full-duplex communication between the host and the peripheral. The link width can be x1, x4, x8, or x16 lanes, requiring connectors of different form factors. Over the years, newer PCIe generations were released with the purpose of increasing the bandwidth, but at the same time, providing backward compatibility. In Table 3.7 it is presented a list with the raw bit rate and bandwidth of each PCIe generation.

Table 3.7: Raw bit rate and bandwidth values for different generations of the PCIe standard.

PCIe generation	Raw bit rate/lane/way	Bandwidth [GB/s] (unidirectional)			
		x1 lane	x4 lanes	x8 lanes	x16 lanes
1.0	2.5 GT/s	0.25	1.0	2.0	4.0
2.0	5 GT/s	0.50	2.0	4.0	8.0
3.0	8 GT/s	0.95	3.94	7.88	15.75
4.0	16 GT/s	1.97	7.88	15.75	31.51

The PCIe devices are mapped into the host address space. When the host starts up, the BIOS or the operating system identifies all the devices connected to the bus and program their Base Address Registers (BARs) with the assigned base address, which is valid until the system is on. Each non-bridge PCIe device can implement up to 6 different BARs, where each BAR represents a different region, having a different base address.

Xilinx provides an IP [42] that allows the connection between the user logic and the PCIe bus through the FPGA PCIe interface. The DMA/Bridge Subsystem for PCI Express, also referred as XDMA, can be configured as a DMA data mover or as Bridge between PCIe and AXI memory. The

DMA data mover is ideal for moving blocks of data, and it can act as an AXI memory mapped interface or as an AXI streaming interface to allow direct connection to RTL logic. The PCIe bridge is used to convert PCIe packets into AXI traffic and vice versa without offering support for data streaming or a DMA. In this design, the DMA data mover configuration was used, acting as an AXI memory-mapped interface, since the main goal is the data transference between the host and the FPGA to specific addresses.

The XDMA diagram, represented in Figure 3.23, shows the different alternatives to access the user logic from the PCIe bus. The AXI DMA can be configured to have up to 4 upstream and downstream channels, which are useful for multi-thread parallel data transmission. Beyond that, the host can bypass the DMA and access the user logic directly through an AXI4 memory-mapped interface with 32-bit data requests. There is also another interface which allows the host to access the configuration and status registers in the user logic through an AXI4-Lite interface. On the host side, each interface has its BAR and a dedicated base address.

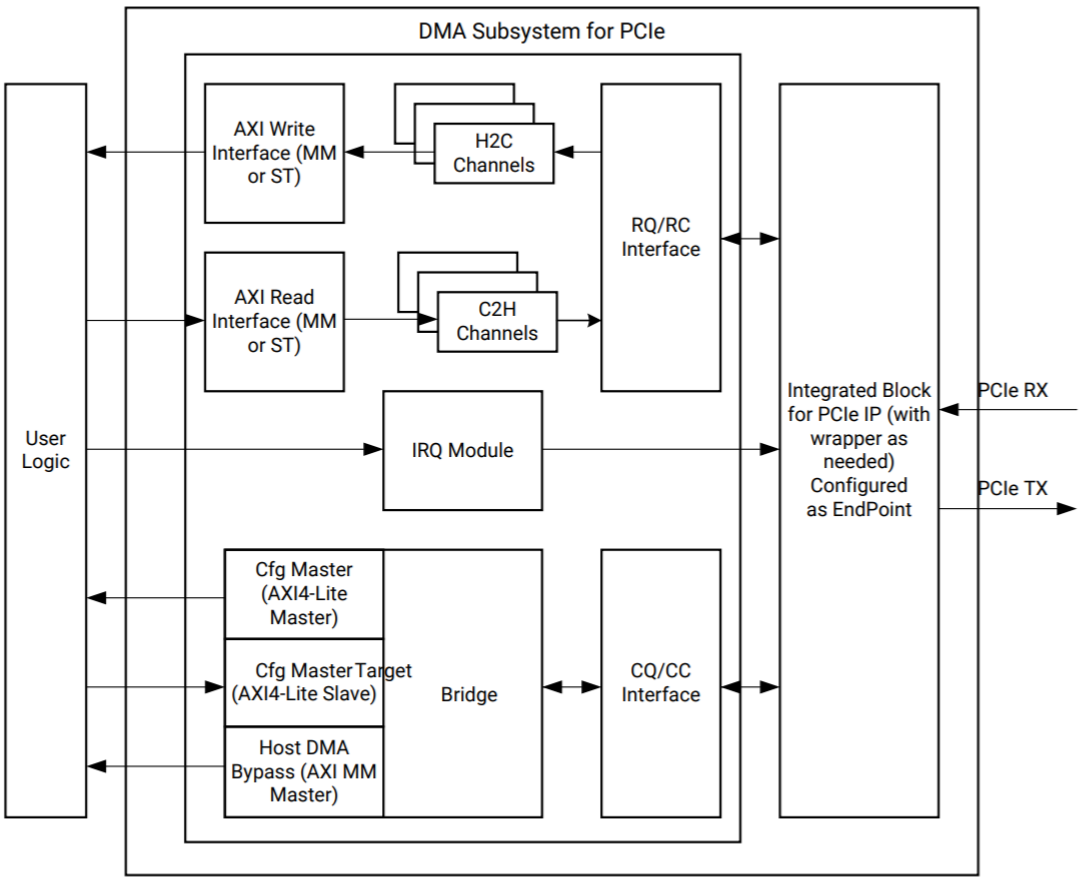
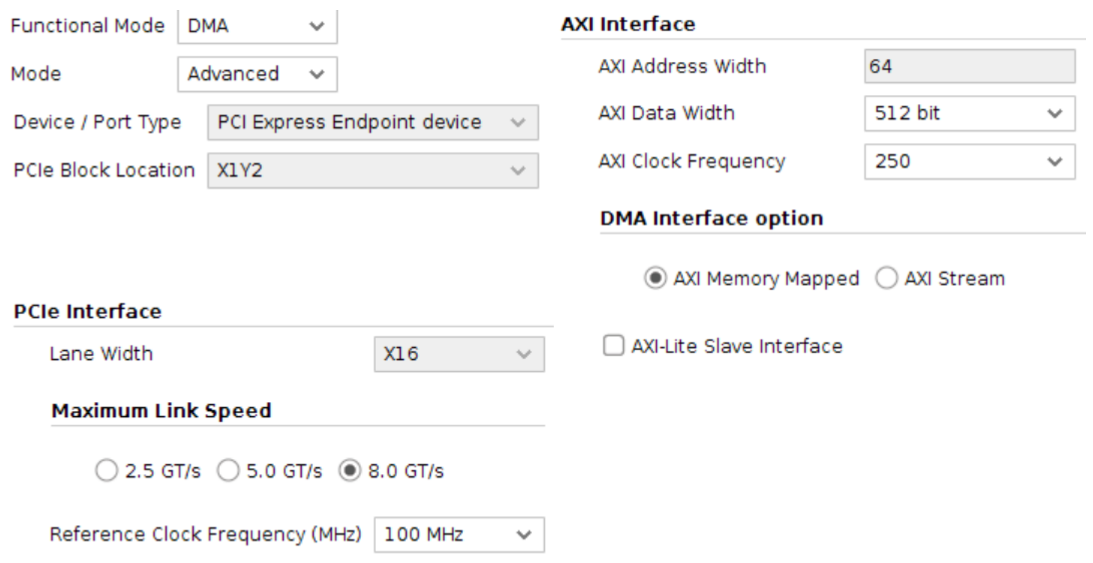

DMA Subsystem for PCIe
User Logic
AXI Write Interface (MM or ST)
H2C Channels
AXI Read Interface (MM or ST)
C2H Channels
RQ/RC Interface
IRQ Module
Cfg Master (AXI4-Lite Master)
Cfg Master Target (AXI4-Lite Slave)
Host DMA Bypass (AXI MM Master)
Bridge
CQ/CC Interface
Integrated Block for PCIe IP (with wrapper as needed) Configured as EndPoint
PCIe RX
PCIe TX

Figure 3.23: DMA Subsystem for PCIe diagram. The AXI4 DMA interface between the host and the FPGA with multiple channels is represented on top of the diagram. On the bottom, the diagram shows the interfaces used for the PCIe traffic to bypass the DMA and access the user logic directly using an AXI4-Lite interface or an AXI4 Master interface.

When configuring the XDMA IP on Vivado, the user must select the functional mode as “DMA” and the AXI interface as “Memory Mapped”. The configuration tool allows the selection of different PCIe link speeds and the number of lanes, according to the support offered by the host and the FPGA. The

AXI Data Width available options are affected by the PCIe interface bandwidth.

During this work, the selected link speed value and the number of lanes correspond to the maximum available, 16 lanes, and 8.0 GT/s (PCIe 3^{rd} Gen.), which represents a bandwidth of 15.75 GB/s in each direction. Hence, by using these PCIe settings, it was possible to use an AXI Data Width of 512 bits. Figure 3.24 exemplifies the configuration that was done.


Functional Mode: DMA
Mode: Advanced
Device / Port Type: PCI Express Endpoint device
PCIe Block Location: X1Y2

PCIe Interface
Lane Width: X16

Maximum Link Speed
○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s

Reference Clock Frequency (MHz): 100 MHz

AXI Interface
AXI Address Width: 64
AXI Data Width: 512 bit
AXI Clock Frequency: 250

DMA Interface option
● AXI Memory Mapped ○ AXI Stream

☐ AXI-Lite Slave Interface

Figure 3.24: DMA/Bridge Subsystem for PCI Express IP configuration on Vivado Block Automation. This IP was configured to use 16 lanes and a maximum link speed of 8 GT/s, which is equivalent to the PCIe 3^{rd} Generation. The AXI bus supports 512-bit words, running at a frequency of 250 MHz. The DMA is configured as an AXI Memory Mapped device.

The data flow between the host and the processor is shown in Figure 3.25. The binary is transferred from the host memory to the XDMA on the FPGA via PCIe. Depending on the destination address, the data that is being received is sent to the DRAM (via MIG IP) or the BRAM (via AXI to BRAM IP) using the AXI4 protocol. The core uses simple interfaces (write, enable, address, data in, data out) to access data on the instruction memory or the data cache. The data cache communicates with the MIG via AXI4 when it needs to access the DRAM to read or store data.

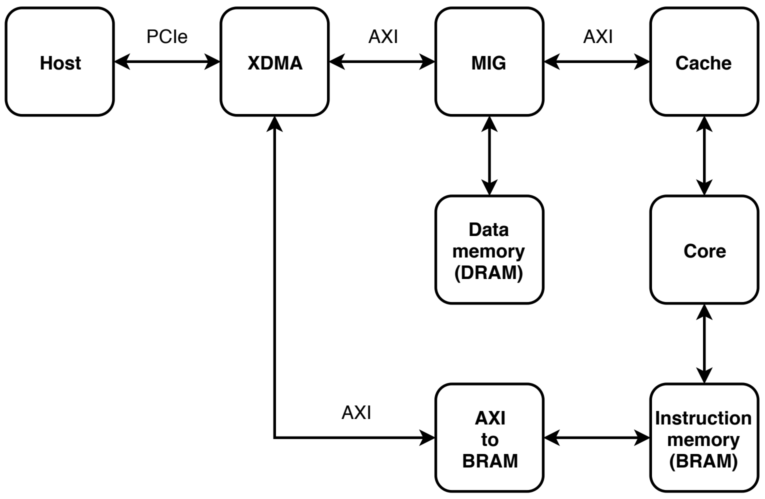


Figure 3.25: Data flow between the host and the processor's memories.

3.8 Adaptation for other FPGA vendors

Currently, the FPGA market is dominated by Xilinx. Each vendor offers different IPs, some of them closed-source, and different tools for design implementation, creating ecosystems compatible just with their products. Even inside their ecosystem, there are different versions of the same IP, depending on the FPGA family, requiring changes in the designs when implemented on different models.

Once the development of this processor settled on the Xilinx environment, some of the IPs used in the design implementation were generated using the Xilinx configuration tools. Consequently, if the user chooses to implement this core on a different FPGA, he should follow the FPGA documentation to find similar alternatives for the incompatible IPs. If the IP interfaces are too different, a wrapper may be needed.

During the development of the processor, the problem of incompatibilities was always present. The use of vendor IPs was avoided whenever it was feasible. In some instances, due to the complexity issues and optimizations, it was mandatory to use vendor IPs, instead of developing custom solutions from scratch to increase the compatibility between different platforms. The Xilinx blocks used in the implementation are:

- PCIe XDMA;
- DRAM MIG;
- Multiplier;
- Divider;
- FIFOs;
- AXI to BRAM controller;
- BRAM generator.

The two major Xilinx blocks that were used are the PCIe XDMA and the DRAM MIG, both compatible with the AXI protocol. If the alternative IPs available for the FPGA where the processor is going to be implemented are compatible with this protocol, the necessary changes are likely to be minimal, since the existing interfaces on the core would be compatible, especially the cache AXI interface and instructions memory interface. In case of the alternative IPs not offering support for the AXI protocol, these interfaces on the core must be adapted.

Beyond these, the multiplier, the divider, and some of the FIFOs also require attention. The adaptation process of these IPs is more straightforward in comparison with the XDMA and the MIG because the number of connections is smaller and standardized.

3.9 Summary

Throughout this chapter, the development of the RISC-V softcore based on the MB-Lite processor was presented. The first section includes an overview of the proposed architecture that highlights the significant changes that were introduced to the original architecture. The second section analyzes the memory structure, where it is shown the instruction and data memories, as well as the processor's address map. Next, it is given special attention to the new multi-cycle functional units and the mitigation of hazards that appear as a result of these changes in the pipeline. To expand the processor's resources and functionalities, several peripherals were developed, sharing the same interface, which increases the flexibility and adaptability in different scenarios. Then, it is explained how the programs are transferred to the processor's memory using the FPGA board PCIe interface. Some of the changes required propriety IPs compatible only with Xilinx. As a consequence, a reflection about the adaptation to other vendors was presented at the end of the chapter.

4 Software Tools

Contents

To complement the proposed architecture, a set of software tools is required to allow the development of programs and the interaction between the core (on the FPGA) and the host machine (personal computer). This chapter focuses on explaining which tools should be installed on the machine to allow the software development and the access to the PCIe bus, to transfer the binary and the data. Beyond that, several other tools were developed, including a Board Support Package, a makefile, libraries to interact with the processor's peripherals, and scripts to transfer the binaries and read or clear the processor's memories, according to the user intention.

4.1 Development Workflow

The development workflow includes the software installed on the host machine to compile programs (RISC-V GCC compiler), the required drivers to enable the connection between the host and the XDMA through the PCIe bus, the support files created to allow the software development (Makefile and Board Support Package) and scripts to transfer data between the host and the core.

4.1.1 Software Requirements

GCC Compiler:

RISC-V offers an official GNU toolchain with support for the GCC compiler, GDB, newlib, and glibc [1]. Due to some bugs on the official tools, it was decided to use as an alternative the xPack GNU RISC-V Embedded GCC [3], which is a pre-compiled modified version specially adapted to produce bare-metal applications and can be easily installed on different operating systems. According to the xPack GNU RISC-V Embedded GCC release notes [3], it is fully compatible with the official specifications.

Since RISC-V is a set of base ISAs and extensions, the compiler must know which modules are being used on the implementation. To provide the supported set of instructions, the user must include the GCC options **-march** and **-mabi** when the compiler is called. For the particular case of the developed processor, with support for the 32-bit base integer ISA and the Integer Multiplication and Division extension, the first option is **-march=rv32im**. The second option, **-mabi**, is used to define the integer ABI in use, **-mabi=ilp32**, supporting 32-bit "int", "long", and pointers, as well as 64-bit type "long long", 8-bit "char", and 16-bit "short". Some commands used for compilation are available in Listing 4.1.

Listing 4.1: GCC compiler commands examples.

```
/../bin/riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -std=gnu11 -Wall -Iinclude
-fno-common -c -o bsp/entry.o bsp/entry.S

/../bin/riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -std=gnu11 -Wall -Iinclude
-fno-common -c -o bsp/exit.o bsp/exit.c

/../bin/riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -std=gnu11 -Wall -Iinclude
-fno-common bsp/entry.o bsp/exit.o -o elf/printf_test.elf src/printf.c src/main.c
-T bsp/ls.lds -nostartfiles -static -L/../lib/gcc/riscv-none-embed/8.2.0/ -lgcc
```

XDMA PCIe drivers:

To allow the detection and data transfers between the host and the XDMA on the FPGA, using the PCIe bus, Xilinx provides specific Windows and Linux drivers [43]. Along with the drivers, Xilinx also offers test scripts to validate the correct operation of the implemented design. Some of the test scripts were modified to allow the transference of any binary file with any arbitrary size to execute or store on the core, with the possibility to set the desired memory address. The user can also use the scripts to clear or dump the memory data using specific flags when executing the script. Listing 4.2 shows how data can be sent and received using the scripts to access the PCIe interface.

Listing 4.2: Examples of commands to write and read data from the processor's memories using the PCIe interface.

```
//command to transfer the printf_test.bin file to the base address 0x00000000
./write_pcie.sh pritff_test.bin 0

//command to read 1024 bytes from the address 0x00004000 (16384)
./read_pcie.sh 1024 16384
```

4.1.2 Support Files

Board Support Package:

The Board Support Package (BSP) is a set of software parts responsible for executing hardware-specific routines. This processor requires an entry (entry.S) and an exit file (exit.c), provided in Appendix B. The entry file (Listing B.2) is responsible for doing the initial setup before and after the program execution, such as global and stack pointers initialization, bss section cleaning, main function execution, and call the exit. The exit function (Listing B.3) is responsible for finishing the processor execution with a jump instruction that ends in a loop and is identified in the ID stage.

Beyond these two files, the BSP also includes a Linker Script. A Linker Script file is used to inform the linker (after the program compilation) of how the memory structure (Figure 4.1) is organized. It reflects precisely the processor's memory map. When the memory map is changed, the Linker Script should be updated with the new memory base addresses and sizes. An example of a Linker Script configuration is available in Listing B.1.

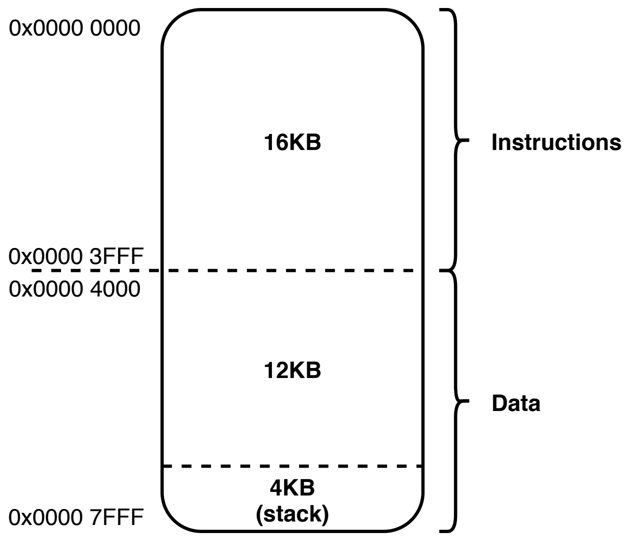


Figure 4.1: Memory structure configured on the Linker Script.

Makefile:

The developed makefile, available in Appendix C, includes a set of directives that allow the compilation and build process automation. The binary file generation is the result of the compilation process described in Figure 4.2. First, the BSP files are compiled and assembled, producing two different objects. Then, the ELF file is created as a result of the source files and BSP compilation and linked with the Linker Script. The RISC-V toolchain path must be provided by setting an environment variable.

The binary output file is created using the RISC-V objcopy, available in the toolchain. It was also included the possibility to generate a unique binary with the entire memory content or two separated binaries, one with the instructions part and the other with the data.

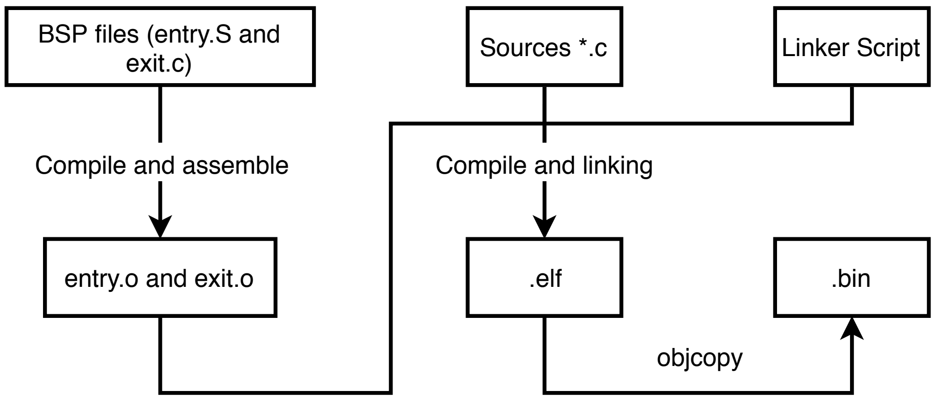


Figure 4.2: Binary generation workflow.

4.2 Developed Libraries

Standard Input and Output:

The set of developed functions that are herein presented have the purpose of making the development process easier when the access to the standard input and output is required. They act as a software layer between the user code and the hardware implementation of the UART module.

Listing 4.3 shows some of the developed functions to print data through the serial interface. The user calls the printf() function as if he was using the standard printf() from the stdio.h library. The printf() function (and others not represented here) are responsible for creating the output string and sending each char to the putchar() function. The putchar() function calls the uart_putchar() function that is responsible for writing the char to the UART TX FIFO memory address, always verifying if the TX FIFO is full or not.

Listing 4.3: Developed Standard Output most relevant functions.

```
#define UART_TX_FULL (*((volatile unsigned int*)(0xFFFFFFF4)))
#define UART_TX_DATA (*((volatile unsigned int*)(0xFFFFFFFC)))

...

//sends each char to the UART TX line
int uart_putchar(char ch){
    //new line detection
    if(ch == '\n'){
        uart_putchar('\r');
    }

    //checks the UART TX FIFO status
    while(UART_TX_FULL == 1);

    //writes the char to the TX line
    UART_TX_DATA = ch;

    return 0;
}

//prints just one char
char putchar(char c){
    uart_putchar(c);

    return c;
}

//printf function
int printf(const char* format, ...){

    ...

}

...
```

To read data from the UART RX line, the user should call the gets() function available in Listing 4.4, passing as argument a pointer to a string and its maximum size. Then, each char stored in the RX FIFO is read (provided that the RX FIFO is not empty). The read process ends when the user writes a new line. All the content written to the UART RX is replicated in the terminal by simply writing it back to the TX line.

Listing 4.4: Developed Standard Input functions.

```
#define UART_RX_EMPTY (*((volatile unsigned int*)(0xFFFFFFF0)))
#define UART_RX_DATA (*((volatile unsigned int*)(0xFFFFFFF8)))

//reads one char each time
int getchar(){
    //wait until data is written to the UART RX FIFO
    while(UART_RX_EMPTY == 1);

    return UART_RX_DATA;
}

//reads a string with the maximum size defined by s
void gets(char *p, int s){
    int c = 0;

    //read next char
    while(--s){
        c = getchar();

        //detect new line
        if(c == '\n' || c == '\r'){
            break;
        }

        //writes the char to the TX to get feedback
        putchar((*p++ = c));
    }

    //writes a new line
    putchar('\n');
    *p = 0;
}
```

Cycle Counter and Timer:

Both the Cycle Counter and the Timer hardware implementation are similar, so the corresponding functions to access both peripherals only differ on the memory addresses that are accessed. Listing 4.5 shows the developed functions getcounter() and printcounter(), used to obtain the current 64-bit counter value and print it to the terminal, respectively. The code detects if the counter overflows between the reading of both parts, and if this happens, it will try again. Listing 4.6 shows similar functions used for the timer (gettimer() and printtimer()).

Listing 4.5: Developed Cycle Counter functions.

```
#define CYCLE_COUNTER_LOW (*((volatile unsigned int*)(0xFFFFFF00)))
#define CYCLE_COUNTER_HIGH (*((volatile unsigned int*)(0xFFFFFF04)))

//gets the current cycle counter value
void getcounter(unsigned int result[2]){
    unsigned int result0, result1, result2 = 0;
    int *cnt_l = (int*) CYCLE_COUNTER_LOW;
    int *cnt_h = (int*) CYCLE_COUNTER_HIGH;

    do{
        result0 = *cnt_h;
        result1 = *cnt_l;
        result2 = *cnt_h;
    }while(result0 != result2);

    result[0] = result1;
    result[1] = result2;
}




//prints the number of clock cycles passed between cycles c1 and c2 in hex.
void printcounter(unsigned int c1[2], unsigned int c2[2]){
        unsigned int r[2];

    r[0] = c2[0] - c1[0];
    r[1] = c2[1] - c1[1];

    if((unsigned int)(c2[0]) < (unsigned int)(c1[0])){
        r[1]--;
    }

        printf("\n\n0x%8X %8X cycles\n\n", r[1], r[0]);
}
```

Listing 4.6: Developed Timer functions.

```
#define TIMER_LOW (*((volatile unsigned int*)(0xFFFFFF80)))
#define TIMER_HIGH (*((volatile unsigned int*)(0xFFFFFF84)))

//gets the current timer value
void gettimer(unsigned int result[2]){
    unsigned int result0, result1, result2 = 0;
    int *timer_l = (int*) TIMER_LOW;
    int *timer_h = (int*) TIMER_HIGH;

    do{
        result0 = *timer_h;
        result1 = *timer_l;
        result2 = *timer_h;
    }while(result0 != result2);

    result[0] = result1;
    result[1] = result2;
}
```

```c
//prints the number of timer clock cycles passed between the instants t1 and t2 in hex.
void printtimer(unsigned int t1[2], unsigned int t2[2]){
    unsigned int r[2];

    r[0] = t2[0] - t1[0];
    r[1] = t2[1] - t1[1];

    if((unsigned int)(t2[0]) < (unsigned int)(t1[0])){
        r[1]--;
    }

    printf("\n\n0x%8X %8X timer clock cycles\n\n", r[1], r[0]);
}
```

Use cases:

To demonstrate how those functions can be used inside the programs, several pieces of code will be presented. The first example (Listing 4.7) shows how the user can get the number of cycles an algorithm takes to be executed. The second example (Listing 4.8) is similar, but this time is for the timer functions. The last example (Listing 4.9) calculates and prints the quotient and the remainder by dividing two integers written by the user on the terminal.

Listing 4.7: Example 1: Counter functions.

```c
int main(){
    unsigned int c1[2], c2[2];
    int disks = 20;

    getcounter(c1);
    hanoi(disks, 'A', 'C', 'B');
    getcounter(c2);

    printcounter(c1, c2);

    return 0;
}
```

Listing 4.8: Example 2: Timer functions.

```c
int main(){
    unsigned int t1[2], t2[2];
    int a = 0;
    int n = 40;

    gettimer(t1);
    a = fib(n);
    gettimer(t2);

    printf("F(%d) = %d\n\n", n, a);

    printtimer(v1, v2);

    return 0;
}
```

Listing 4.9: Example 3: Standard I/O functions.

```c
int main(){
    char a[10];
    int n = 0;
    int dividend = 0;
    int divisor = 1;
    int result = 0;
    int remainder = 0;

    while(divisor != 0){
        printf("Dividend:\n");
        gets(a, 100);
        dividend = atoi(a);

        printf("Divisor:\n");
        gets(a, 100);
        divisor = atoi(a);

        result = dividend / divisor;
        remainder = dividend % divisor;

        printf("Quotient = %d\n", result);
        printf("Remainder = %d\n\n", remainder);
    }

    return 0;
}
```

4.3 Summary

This chapter presented the developed software tools that can be used as a complement to the hardware. It starts with the presentation of the required software toolchain (the RISC-V GCC Compiler and the Xilinx XDMA PCIe driver), and the support files that are essential for the compilation process: the Board Support Package files and the Makefile. The second and last section presents and explains the developed functions necessary to interact with the processor's peripherals, with some usage examples.

5

Implementation and Experimental Results

Contents

To test and evaluate the developed processor, different system configurations were implemented and prototyped in an FPGA, to be compared and evaluated in terms of resources, operating frequency, and power requirements.

Moreover, some software tests were executed to verify the correct system operation and to obtain quantitative evaluation from the execution of algorithms that can be used as benchmarks for comparing its performance with other softcores. Those tests were compiled using the RISC-V compiler. In Appendix D, part of the disassembled code of a test program is shown for both RISC-V and MicroBlaze ISAs. It is visible that both assembly codes are not too different, justifying the approach of modifying the ID stage decoder.

Based on the obtained results and taking into account all the implemented features, a discussion is presented, at the end of the chapter, to compare the proposed solution with the other RISC-V cores presented in Chapter 2.

5.1 Prototyping framework

The developed processor was implemented on a Xilinx Virtex Ultrascale+ VCU1525 device [44], featuring an XCVU9P FPGA. This board is mainly targeted for computationally intensive applications due to the amount of resources offered by its FPGA and the available communication interfaces. Looking for its technical features, stand out the Gen3 x16 / Gen4 x8 PCIe interface, offering high transfer rates, four DDR4 DIMM slots, each one with 16GB memories, and finally, a UART interface via USB. Regarding the FPGA resources, the number of available LUTs, FFs, BRAMs, LUT Random-Access Memories (LUTRAMs), and Digital Signal Processing (DSP) units are listed in Table 5.1 [45].

Table 5.1: XCVU9P FPGA resources.

LUT	1182240
FF	2364480
BRAM	2160
LUTRAM	591840
DSP	6840

During the development phase, several versions of the softcore were implemented on the FPGA, as new features were added. This process started with the PCIe XDMA implementation, followed by modifications on the architecture to add multi-cycle functional units support, hazards handling, peripherals, and cache support. The last step was the connection to the MIG, to start using the DRAM as data memory. The complete solution was implemented according to what was initially purposed and idealized. The simplified system diagram, presented in Figure 5.1, illustrates the major components and the connections between them.

The XDMA IP is responsible for connecting the host processor and the core's memory system, by converting the PCIe data packets received from the host into AXI requests. Inside the FPGA, the data is transferred, according to the address map, via AXI to the data memory (and placed in the external DDR4 memory through the MIG IP) or to the instruction memory, implemented with BRAMs. The data cache, the UART module, and other peripherals are connected to the core data bus through the address decoder. Hence, this address decoder is responsible for forwarding the memory requests according to the memory address. Finally, the user has access to the core's standard input and output by using the implemented UART, in particular, by connecting the host to the board's USB port, since it provides UART over USB conversion.

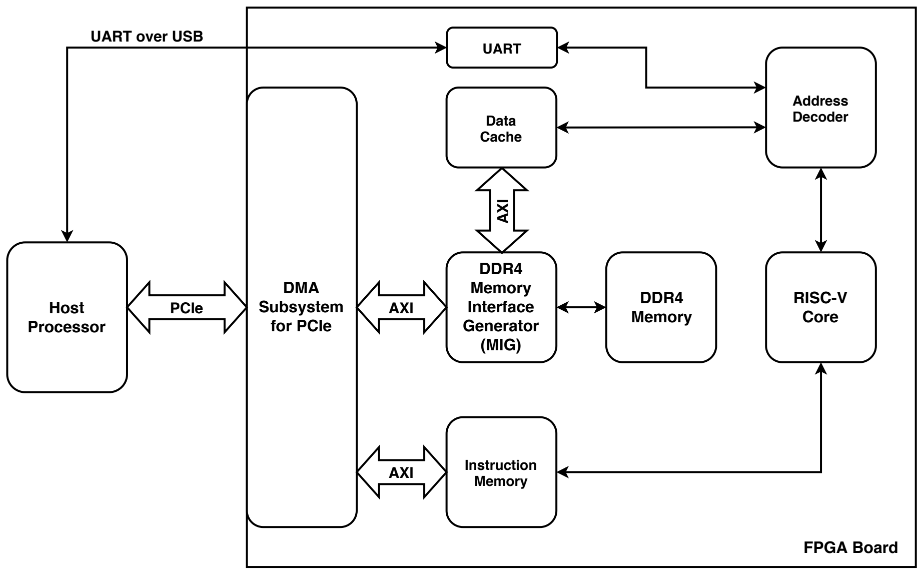


Figure 5.1: Simplified diagram of the proposed system. This diagram is an overview of the connections between the main blocks inside the FPGA and the host processor.

To make the implementation process more manageable, the core and the data cache were packaged into custom IPs. The complete version was implemented with the help of Vivado IP Integrator design automation tool, which is responsible for the auto-configuration of some of the IPs (such as, the XDMA, and the MIG) and establishing connections between them. Those tools also add extra logic when the IPs cannot be directly connected; for instance, when they are driven by different clocks, the AXIs interfaces have different configurations, or when it is necessary to connect multiple AXI blocks to the same bus.

The Block Design created for the full implementation is available in Appendix E. There, we can find all the processor required IPs (Core, Cache, MIG, and XDMA) and the additional ones (Clock generators, System Reset, Inverters, and AXI SmartConnect).

The following variants in the developed softcore were implemented:

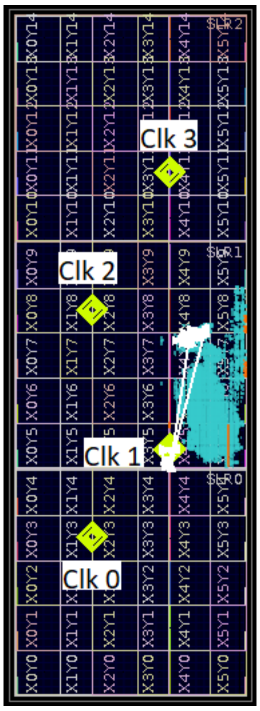
- Core: this is the simplest implementation, that only includes the core with the modifications in the architecture. Different versions with and without the multiplier and the divider were also implemented to evaluate the resources used by these units.
- Core + XDMA: base version including the XDMA IP.
- Core + XDMA + Cache: base version including the XDMA IP and the data cache.

- Core + XDMA + Cache + DRAM: this version is the complete system, featuring the connection to the DRAM using the MIG IP, as shown in Appendix E.

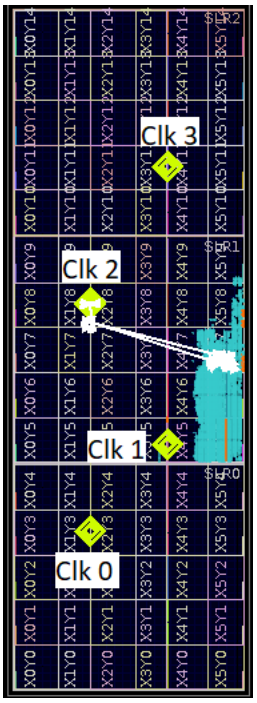
5.1.1 Area and timing constraints analysis

By using the Xilinx Vivado 2019.1 tool, the proposed architecture was submitted to the implementation process to understand how the modifications impacted the resources usage and operating frequency. The small version of the implemented RISC-V core is a minimal implementation with all the changes in the pipeline, and optionally, the divider and the multiplier. However, the extra IPs, such as the XDMA, the MIG, the data cache, and the peripherals, were removed. Looking at Table 5.2, we can observe that the minimal implementation, as expected, uses less resources than the others and it supports a maximum operating frequency of 250MHz. The multiplier has low logic usage since it is implemented with DSPs. However, the divider increases the area with significant impact in the FFs and LUTs usage because the Radix-2 algorithm exploits the FPGA logic, in opposition to the High Radix algorithm that uses DSPs.

With the addition of the XDMA, the used resources increased significantly. The XDMA is a complex IP that requires a large FPGA area, especially in terms of LUTs and Flip-flops. The frequency was reduced from 250MHz to 200MHz. An explanation for that can be the fact that the logic placement is near the PCIe pins on the FPGA, but the four clock pins are relatively far (as shown in Figure 5.2), which increases the critical path and forces higher clock periods. This implementation also includes a data memory implemented with BRAMs, which explains the use of 75 BRAMs when compared to the base core.



(a) Implementation using the clock 1.



(b) Implementation using the clock 2.

Figure 5.2: Vivado post-implementation device view showing the critical path with 200MHz clocks.

When the pre-developed cache integrated into a custom IP was added to the processor, the clock frequency was reduced to 100 MHz. The main reason for this reduction is the fact that the considered cache implementation is not optimized, leading to an increase in the critical path. According to information reported by Vivado, the critical path starts on the logic responsible for selecting the processor output data and ends on the Write-Back stage memory data input.

The last implementation presented in Table 5.2 corresponds to the complete system, which includes the connection to the DRAM using the MIG IP. The operating frequency was kept in 100MHz, but the used area increased about three times, mainly due to the resources required by the MIG implementation.

Table 5.2: Resources usage and operating frequency of the different configurations of the developed processor.

Design	**Resources**					**Frequency**
	LUT	**LUTRAM**	**FF**	**BRAM**	**DSP**	**[MHz]**
Core	5800	2407	2346	0	0	250
Core (+ MUL)	5921	2434	2439	0	4	250
Core (+ DIV)	7234	2419	5230	0	0	250
Core (+ MUL + DIV)	7336	2446	5307	0	4	250
Core + XDMA	28856	2367	31931	75	4	200
Core + XDMA + Cache	33137	3699	33872	59	4	100
Core + XDMA + Cache + DRAM	94343	9924	102055	110.5	7	100

5.1.2 Power analysis

Table 5.3 presents the results of the power analysis estimations that were provided by Vivado tool after the post-implementation process. The resources used and the routing, as well as other circuits, affect the power results. More resources typically mean more energy consumed.

Looking at these the power results, it can be seen that the developed core's small implementation require less power than the others, since it has the lowest area occupancy. On the other hand, the two implementations with the XDMA IP, as expected, have higher power values. The complete system, with the DRAM connection, uses three times more resources than the previous implementation, causing an increase of about three times in the power consumption.

Table 5.3: Power results for each implementation.

Design	Dynamic Power [W]
Core	0.27
Core (+ MUL + DIV)	0.30
Core + XDMA	1.31
Core + XDMA + Cache	1.32
Core + XDMA + Cache + DRAM	3.91

5.2 Benchmarks results

To evaluate the processor performance and reliability, a set of benchmark tests were executed. These tests, written in C, implement simple algorithms that require some processor effort to finish the tasks. The number of clock cycles and the corresponding execution time were obtained for each test.

The set of benchmark tests includes the Tower of Hanoi, Fibonacci sequence element calculation, and vector-vector division and multiplication. The first two tests implement recursive algorithms that can be used to confirm the processor's reliability, Due to these algorithms' execution grow rates, the processor runs during millions of clock cycles without failing. The others are useful for testing data memory accesses, by enabling the verification of hardware resources, such as the data cache system operation, the main memory accesses communication, and the implemented divider and multiplier.

The advantage of using such algorithms is because they are simple, and the obtained results can be easily verified, being useful to validate the correct operation of the developed solution.

Tower of Hanoi:

The Tower of Hanoi puzzle consists of n disks and three rods. The disks are placed in one of the rods, and the goal is to move them from one rod to another rod. The rules say that only one disk can be moved each time, in each move the top disk is removed and placed on top of another rod, and a disk cannot be placed on top of a smaller disk. The time complexity for this algorithm is exponential, $\mathcal{O}(2^n)$, meaning that this is a computationally costly algorithm. In terms of memory, the complexity is $\mathcal{O}(n)$ since it only stores n return pointers because it is the maximum number of recursive calls. The used algorithm implementation is presented in Listing 5.1.

Several tests were executed using a different number of disks. For each test, the number of clock cycles and the time needed to finish the algorithm execution was measured. The results are present in Table 5.4. For each test with n disks, if we divide the number of clock cycles by the algorithm complexity, 2^n, we will get a constant approaching 56 the higher the number of disks. So, for this processor, the number of clock cycles for n disks is equal to 56×2^n.

Listing 5.1: Tower of Hanoi recursive implementation.

```
void move(int n, char from, char to, char aux){
        if(n == 1){
                return;
        }
        move(n-1, from, aux, to);
        move(n-1, aux, to, from);
}
```

Table 5.4: Number of clock cycles and time duration for different tests with different number of disks.

#Disks	**5**	**10**	**15**	**20**	**25**
#Clock cycles	1851	57403	1835193	58720597	1879048995
Time	18,51 us	574,03 us	18,35 ms	587,21 ms	18,79 s

Fibonacci Sequence:

The series of Fibonacci numbers is represented by the following sequence: $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$ and $F_1 = 1$. From a computational point of view, using a recursive algorithm to compute the n^{th} element without compilation optimizations, the complexity is given by $\mathcal{O}(1,618^n)$. The memory complexity is linear and it is given by $\mathcal{O}(n)$, since the recursive call stack has the maximum size equals to n. The C code used in this test to implement the Fibonacci sequence algorithm to get the n^{th} element is available at Listing 5.2.

The Fibonacci algorithm execution was tested for the first 40 elements, and the results were correct. The number of clock cycles and the execution time required to run some of the tests are presented in Table 5.5. By doing the same analysis that was done with the Hanoi Towers' results, the number of clock cycles is given by a constant multiplying by the complexity. For these results, the constant is approaching 71,4, so the number of clock cycles for this processor can be estimated by $71,4 \times 1,618^n$. With this test, it was also possible to guarantee the 64-bit cycle counter correct operation, since values represented by more than 32 bits were acquired successfully.

Listing 5.2: Fibonacci Sequence recursive implementation.

```
int fib(int n){
    if (n == 0){
        return 0;
    }
    else if(n == 1){
        return 1;
    }
    else{
        return fib(n-1) + fib(n-2);
    }
}
```

Table 5.5: Number of clock cycles and time duration for different tests with a different number of elements.

n	5	10	15	20	30	40
#Clock cycles	830	8825	97591	1080782	132888429	16344128821
Time	8,30 us	88,25 us	975,91 us	10,81 ms	1,33 s	163,44 s

Vector-vector division and multiplication:

To evaluate the correct operation of each functional unit and the access to the data memory, several tests involving vectors were executed. In each test, two arrays with the same size were divided or multiplied element by element, as show in Listing 5.3. The number of clock cycles and time of each test were measured. The results for the division are available in Table 5.6 and for the multiplication are in Table 5.7. The tests were executed several times for arrays with different sizes. When the number of elements varies, the execution time is affected in the same proportionality.

Listing 5.3: Division and multiplication between the elements of two arrays.

```
void vvdiv(int n, int a[], int b[], int c[]){
    int i;
    for(i = 0; i < n; i++){
        c[i] = a[i] / b[i];
    }
}

void vvmul(int n, int a[], int b[], int c[]){
    int i;
    for(i = 0; i < n; i++){
        c[i] = a[i] * b[i];
    }
}
```

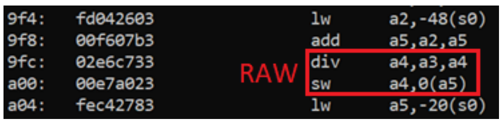
Table 5.6: Vector-vector division: number of clock cycles and time duration for different vector sizes.

#Elements	1k	10k	100k	1M	10M
#Clock cycles	109316	1092009	10919860	109200045	1091999939
Time	1,09 ms	10,92 ms	109,20 ms	1,09 s	10,92 s

Table 5.7: Vector-vector multiplication: number of clock cycles and time duration for different vector sizes.

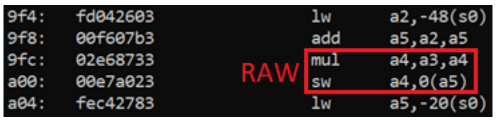
#Elements	1k	10k	100k	1M	10M
#Clock cycles	85588	854095	8541234	85410105	854099854
Time	855,88 us	8,54 ms	85,41 ms	854,10 ms	8,54 s

The pipeline capability of each multi-cycle functional unit tested here is not improving the results, since a RAW hazard occurs just after the issue of the corresponding arithmetic instruction when the store instruction that receives the result arrives at the ID stage and the processor stalls until the result is out the EX stage. Part of the Assembly code of both tests is shown is Figure 5.3, where the RAW hazard is highlighted in both cases.



```
9f4:   fd042603          lw     a2,-48(s0)
9f8:   00f607b3          add    a5,a2,a5
9fc:   02e6c733          div    a4,a3,a4
a00:   00e7a023          sw     a4,0(a5)
a04:   fec42783          lw     a5,-20(s0)
```

(a) SW and DIV RAW hazard.



```
9f4:   fd042603          lw     a2,-48(s0)
9f8:   00f607b3          add    a5,a2,a5
9fc:   02e68733          mul    a4,a3,a4
a00:   00e7a023          sw     a4,0(a5)
a04:   fec42783          lw     a5,-20(s0)
```

(b) SW and MUL RAW hazard.

Figure 5.3: RAW hazard every time each element result is stored

5.3 Discussion

The comparison of this solution with other softcores is not an easy task. As it was previously said, the other RISC-V cores presented in this work have several particularities. Some are not suitable for FPGAs, so they cannot be implemented and tested under the same conditions. Others, announced as suitable for FPGAs, are difficult to implement due to the lack of documentation or are attached to specific boards. The obtained results that were achieved with this processor in terms of frequency and resources usage also cannot be directly compared, because all of them were implemented in different FPGAs, leading to different implementation results.

What it is possible to compare are the functionalities and the characteristics each one offers. Looking at Table 2.3, we can observe that the developed architecture is the first one offering support for data transfer via PCIe, it integrates a data cache, supports the Multiplication and Division Extension, AXI connection to the external DRAM memory, several peripherals, including a UART module, being prepared to support more due to a well-defined interface.

Beyond that, due to its robust architecture inherited from the MB-Lite softcore, the proposed solution keeps its adaptability and flexibility. Most of the modifications were made thinking in future customizations, such as the addition of more functional units with variable latencies, the addition of new peripherals, or other types of cache structures.

5.4 Summary

This chapter presented the implementation and evaluation procedures of the developed system on an FPGA. It started by presenting the prototyping framework, where the FPGA that was used for the implementation was presented. Then, the implementation process was described, and the different implementation versions were compared in terms of resource usage, operating frequency, and power requirements. To evaluate and test the processor execution on the FPGA, different algorithms were executed, and the performance results were listed. Finally, a reflection was made to discuss the experimental results and compare what was achieved over this work with the other RISC-V cores analyzed in Chapter 2.

6 Conclusions

The number of RISC-V processors has increased in the last years. This fact is explained by the recent efforts that were made on the RISC-V development with the support offered by the open-source community and companies that integrate the RISC-V Foundation. It is expected that in the future, several commercial products will include RISC-V cores instead of the current closed and proprietary solutions.

Despite the recent work done in the available RISC-V implementations, the integration of one of the existing cores in projects is not easy. Several problems in the current solutions were found, such as the lack of support for FPGA implementation with suitable performances, poor documentation and information explaining how it operates and what are the steps to execute the implementation process successfully.

To contribute to the RISC-V development, a new RISC-V softcore processor for FPGA implementation was proposed in this work. Throughout the development, the limitations found in the existing processors were taken into consideration.

The development started after the analysis of the RISC-V ISA, to identify the required instructions for a base implementation and the existing extension, giving more attention to the Integer Multiplication and Division Extension. Research about the current state of the RISC-V processors was also done. Seven cores were compared, using as evaluation parameters the available documentation, the degree of customization, and the existing FPGA implementation support.

Based on the information collected from the initial analysis, other non-RISC-V cores were analyzed, with the expectation of finding one with a well-designed architecture, proper documentation, and good performance to serve as the base of this architecture. The MB-Lite softcore, which originally implements the Xilinx's MicroBlaze ISA, was selected due to its well-designed architecture and the methodologies used during its development related to the HDL implementation.

The first step was the implementation of the RISC-V ISA on the MB-Lite architecture. Due to differences in both ISA's instructions formats, the decoder was modified, so the RISC-V instructions were well parsed. After that, to support the Integer Multiplication and Division extension, a multi-cycle multiplier and a divider were included in the Execute stage. The memory system was improved with the implementation of a data cache, support for external memories, and data transfer via PCIe. Three peripherals (a UART module, a counter, and a timer) were developed, as well as software libraries to help the software development. The complete system was successfully implemented and tested in a Virtex UltraScale+ VCU1525 FPGA board.

6.1 Future Work

For future work, this processor can be used in the development of a multicore system, just like the motivations that originated this work. Before that can be achievable, there are some recommendations to improve this softcore:

- Create support for the implementation of an instruction cache, since it would be essential to support the implementation of a multi-level cache system, fundamental in a multicore processor.
- Improve the current cache implementation by reducing its critical path, expecting improvements in the operating frequency.
- Development and implementation of a debugger unit to enable the GDB debugging.
- Implement more RISC-V extensions, such as the floating-point extension and the atomic instructions.
- Test the developed implementation in other FPGAs.

References

[1] RISC-V. GNU toolchain for RISC-V, including GCC. Available: `https://github.com/riscv/riscv-gnu-toolchain`, 2019. [Online].

[2] L. Project. LLVM 9.0.0 Release Notes. Available: `http://releases.llvm.org/9.0.0/docs/ReleaseNotes.html`, 2019. [Online].

[3] T. xPack Project. xPack GNU RISC-V Embedded GCC. Available: `https://xpack.github.io/blog/2019/07/31/riscv-none-embed-gcc-v8-2-0-3-1-released/`, 2019. [Online; accessed 10-October-2019].

[4] RISC-V. Spike, a RISC-V ISA Simulator. Available: `https://github.com/riscv/riscv-isa-sim`, 2019. [Online].

[5] QEMU. Documentation/Platforms/RISCV. Available: `https://wiki.qemu.org/Documentation/Platforms/RISCV`, 2019. [Online].

[6] B. Zimmer, R. Venkatesan, Y. S. Shao, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, et al. A 0.11 pj/op, 0.32-128 tops, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16nm. In *2019 Symposium on VLSI Circuits*, pages C300–C301. IEEE, 2019.

[7] W. Digital. RISC-V: Accelerating Next-Generation Computing Architectures. Available: `https://www.westerndigital.com/company/innovations/risc-v`, 2019. [Online].

[8] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The Rocket Chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[9] A. Traber and M. Gautschi. *PULPino: Datasheet.* ETH Zurich, University of Bologna, 2017.

[10] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Technical report, RISC-V Foundation, 2017.

[11] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified. Technical report, RISC-V Foundation, March 2019.

[12] C. Wolf. PicoRV32 - A Size-Optimized RISC-V CPU. Available: https://github.com/cliffordwolf/picorv32, 2015. [Online; accessed 16-June-2019].

[13] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221. IEEE, 2012.

[14] K. Asanovic, D. A. Patterson, and C. Celio. *The Berkeley Out-of-Order Machine (BOOM) Design Specification*. University of California, Berkeley, 2016.

[15] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 29–42. IEEE Press, 2018.

[16] C. Celio, J. Zhao, A. Gonzalez, K. Asanovic, B. Korpan, and D. Patterson. BOOM: Berkeley Out-of-Order Machine. Available: https://github.com/riscv-boom/riscv-boom, 2019. [Online; accessed 25-June-2019].

[17] VectorBlox. VectorBlox ORCA. Available: https://github.com/VectorBlox/orca. [Online; accessed 25-June-2019].

[18] VectorBlox. ORCA FPGA-Optimized RISC-V. 2016.

[19] K. Skordal. The Potato Processor: A simple RISC-V processor for use in FPGA designs. Available: https://github.com/skordal/potato. [Online; accessed 25-June-2019].

[20] A. Traber, M. Gautschi, and P. D. Schiavone. *RI5CY: User Manual Revision 4.0*. ETH Zurich, University of Bologna, 2019.

[21] P. D. Schiavone. *zero-riscy: User Manual Revision 0.2*. ETH Zurich, University of Bologna, 2018.

[22] VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation. Available: https://github.com/SpinalHDL/VexRiscv. [Online; accessed 01-July-2019].

[23] W. Digital. SweRV EH1 core. Available: https://github.com/chipsalliance/Cores-SweRV, 2019. [Online; accessed 02-July-2019].

[24] W. Digital. RISC-V SweRV^{TM} EH1 Programmer's Reference Manual. Available: https://github.com/chipsalliance/Cores-SweRV, 2019. [Online; accessed 26-June-2019].

[25] R. Jia, C. Y. Lin, Z. Guo, R. Chen, F. Wang, T. Gao, and H. Yang. A survey of open source processors for FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2014.

[26] T. Kranenburg. Design of a portable and customizable microprocessor for rapid system prototyping. 2009.

[27] T. Kranenburg and R. Van Leuken. MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 997–1000. European Design and Automation Association, 2010.

[28] D. Lampret. OpenRISC 1200 IP core specification. *September June*, 2001.

[29] J. Gaisler. Fault-tolerant microprocessors for space applications. *Gaisler Research*, pages 41–50, 2012.

[30] L. Barthe, L. V. Cargnini, P. Benoit, and L. Torres. The secretblaze: A configurable and cost-effective open-source soft-core processor. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 310–313. IEEE, 2011.

[31] Xilinx. MicroBlaze Processor Reference Guide. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf`, 2018.

[32] Xilinx. Block Memory Generator. Available: `https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf`, 2017. [Online].

[33] Xilinx. AXI Block RAM (BRAM) Controller. Available: `https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf`, 2017. [Online].

[34] Xilinx. UltraScale Architecture-Based FPGAs Memory IP. Available: `https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf`, 2019. [Online].

[35] ARM. AMBA® AXI™ and ACE™ Protocol Specification, 2011.

[36] Xilinx. AXI Reference Guide. Available: `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf`, 2017. [Online].

[37] Xilinx. Multiplier. Available: `https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf`, 2015. [Online].

[38] Xilinx. Divider Generator. Available: `https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf`, 2016. [Online].

[39] ISO/IEC 9899:1999. Programming languages — C. Standard, International Organization for Standardization, Dec. 1999.

[40] J. E. Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40. ACM, 1964.

[41] nandland. UART, Serial Port, RS-232 Interface. Available: https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html. [Online].

[42] Xilinx. DMA/Bridge Subsystem for PCI Express. Available: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf, 2019. [Online].

[43] Xilinx. Xilinx PCI Express DMA Drivers and Software Guide. Available: https://www.xilinx.com/support/answers/65444.html, 2019. [Online; accessed 01-October-2019].

[44] Xilinx. VCU1525 Reconfigurable Acceleration Platform - User Guide. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf, 2019. [Online].

[45] Xilinx. Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit. Available: https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html, 2019. [Online].

A

Implemented RISC-V instructions

Table A.1: List of the implemented RISC-V instructions.

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Table A.2: List of the MicroBlaze instructions implemented by the MB-Lite softcore.

000000	Rd	Ra	Rb	00000000000	ADD
000001	Rd	Ra	Rb	00000000000	RSUB
000101	Rd	Ra	Rb	00000000001	CMP
000101	Rd	Ra	Rb	00000000011	CMPU
001000	Rd	Ra	Imm		ADDI
001001	Rd	Ra	Imm		RSUBI
100000	Rd	Ra	Rb	00000000000	OR
100001	Rd	Ra	Rb	00000000000	AND
100010	Rd	Ra	Rb	00000000000	XOR
100011	Rd	Ra	Rb	00000000000	ANDN
101000	Rd	Ra	Imm		ORI
101001	Rd	Ra	Imm		ANDI
101010	Rd	Ra	Imm		XORI
101011	Rd	Ra	Imm		ANDNI
101100	00000	00000	Imm		IMM
100100	Rd	Ra	0000000000000001		SRA
100100	Rd	Ra	0000000000100001		SRC
100100	Rd	Ra	0000000001000001		SRL
100110	00000	0(1)0000	Rb	00000000000	BR(BRD)
100110	Rd	10100	Rb	00000000000	BRLD
100110	00000	0(1)1000	Rb	00000000000	BRA(BRAD)
100110	Rd	11100	Rb	00000000000	BRALD
100110	Rd	01100	Rb	00000000000	BRK
101110	00000	0(1)0000	Imm		BRI(BRID)
101110	Rd	10100	Imm		BRLID
101110	00000	0(1)1000	Imm		BRAI(BRAID)
101110	Rd	11100	Imm		BRALID
101110	Rd	01100	Imm		BRKI
100111	0(1)0000	Ra	Rb	00000000000	BEQ(BEQD)
100111	0(1)0001	Ra	Rb	00000000000	BNE(BNED)
100111	0(1)0010	Ra	Rb	00000000000	BLT(BLTD)
100111	0(1)0011	Ra	Rb	00000000000	BLE(BLED)
100111	0(1)0100	Ra	Rb	00000000000	BGT(BGTD)
100111	0(1)0101	Ra	Rb	00000000000	BGE(BGED)
101111	0(1)0000	Ra	Imm		BEQI(BEQID)
101111	0(1)0001	Ra	Imm		BNEI(BNEID)
101111	0(1)0010	Ra	Imm		BLTI(BLTID)
101111	0(1)0011	Ra	Imm		BLEI(BLEID)
101111	0(1)0100	Ra	Imm		BGTI(BGTID)
101111	0(1)0101	Ra	Imm		BGEI(BGEID)
110100	Rd	Ra	Rb	00000000000	SB
110101	Rd	Ra	Rb	00000000000	SH
110110	Rd	Ra	Rb	00000000000	SW
111100	Rd	Ra	Imm		SBI
111101	Rd	Ra	Imm		SHI
111110	Rd	Ra	Imm		SWI
110000	Rd	Ra	Rb	00000000000	LBU
110001	Rd	Ra	Rb	00000000000	LHU
110010	Rd	Ra	Rb	00000000000	LW
111000	Rd	Ra	Imm		LBUI
111001	Rd	Ra	Imm		LHUI
111010	Rd	Ra	Imm		LWI
010000	Rd	Ra	Rb	00000000000	MUL
011000	Rd	Ra	Imm		MULI
010001	Rd	Ra	Rb	00000000000	BSRL
010001	Rd	Ra	Rb	01000000000	BSRA
010001	Rd	Ra	Rb	10000000000	BSLL
011001	Rd	Ra	00000000000 & Imm5		BSRLI
011001	Rd	Ra	00000010000 & Imm5		BSRAI
011001	Rd	Ra	00000100000 & Imm5		BSLLI

B

Board Support Package

Listing B.1: Linker Script example

```
OUTPUT_ARCH ( "riscv" )

ENTRY ( _start )

MEMORY
{
  rom (x!rw) : ORIGIN = 0x00000000, LENGTH = 16K
  ram (wxa!ri) : ORIGIN = 0x00004000, LENGTH = 16K
}

SECTIONS
{
  __stack_size = 4K;

  .init : {
    KEEP (*(SORT_NONE(.init)))
  } >rom

  .text : {
    *(.text .text.*)
  } >rom

  .fini : {
    KEEP (*(SORT_NONE(.fini)))
  } >rom

  PROVIDE ( __etext = . );
  PROVIDE ( _etext = . );
  PROVIDE ( etext = . );

  .rodata : {
    *(.rdata)
    *(.rodata .rodata.*)
  } >ram

  .data : {
    *(.data .data.*)
    *(.sdata .sdata.*)
    PROVIDE( __global_pointer$ = . );
  } >ram

  PROVIDE ( _fbss = . );
  PROVIDE ( __bss_start = . );
  .bss : {
    *(.sbss)
    *(.bss .bss.*)
    *(COMMON)
  } >ram

  PROVIDE ( _end = . );
  PROVIDE ( end = . );

  .stack ORIGIN(ram) + LENGTH(ram) - __stack_size : {
    PROVIDE ( _heap_end = . );
    . = __stack_size;
    PROVIDE( _sp = . );
  } >ram
}
```

Listing B.2: entry.S

```
    .section .init
    .globl _start
    .type _start,@function
_start:
    # Initialize the global pointer
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop

    # Initialize the stack pointer
    la sp, _sp

    # Clear the bss section
    la a0, __bss_start
    la a1, _end
    bgeu a0, a1, 2f
1:
    sw zero, (a0)
    addi a0, a0, 4
    bltu a0, a1, 1b
2:
    auipc ra, 0
    addi sp, sp, -16
    sw ra, 8(sp)

    # Start the program
    li a0, 0
    li a1, 0
    call main

    # Finish the execution
    tail __exit
```

Listing B.3: exit.c

```
void __exit(void) {
    asm volatile ("nop");
    asm volatile ("nop");
    asm volatile ("nop");
    asm volatile ("jalr zero, zero, 0x0");
}
```

C
Makefile

Listing C.1: Makefile.

```
# RISCV environment variable must be set
CC=$(RISCV)/bin/riscv-none-embed-gcc
OBJCOPY=$(RISCV)/bin/riscv-none-embed-objcopy
CFLAGS=-march=rv32im -mabi=ilp32 -std=gnu11 -Wall -Iinclude -fno-common

# Set board support package path
BSP_PATH=bsp

# Output name
OUT=$(shell basename $(CURDIR))

# Linker Script Flags
LINKER_SCRIPT=$(BSP_PATH)/ls.lds
LDFLAGS=-T $(LINKER_SCRIPT) -nostartfiles -static -L$(RISCV)/lib/gcc/riscv-none-embed/8.2.0/ -lgcc

# BSP files (entry.S and exit.c)
ASM_SRC=$(BSP_PATH)/entry.S
C_SRC=$(BSP_PATH)/exit.c
ASM_OBJS=$(ASM_SRC:%.S=%.o)
C_OBJS=$(C_SRC:%.c=%.o)
LINK_OBJS=$(ASM_OBJS) $(C_OBJS)

# Source files
SRC_FILES=$(wildcard src/*.c)

.PHONY: all dirs clean
all: dirs BIN

# Create the output directories
dirs:
	mkdir -p elf/
	mkdir -p bin/

# Generate the ELF file
ELF: $(LINK_OBJS) $(LINKER_SCRIPT)
	$(CC) $(CFLAGS) $(LINK_OBJS) -o elf/$(OUT).elf $(SRC_FILES) $(LDFLAGS)

# BSP objects
$(ASM_OBJS): %.o: %.S
	$(CC) $(CFLAGS) -c -o $@ $<

$(C_OBJS): %.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $<

# Generate the binaries based on the ELF using the objcopy
BIN: ELF
	$(OBJCOPY) -O binary elf/$(OUT).elf bin/$(OUT).bin
	$(OBJCOPY) -O binary elf/$(OUT).elf -R .rodata -R .data bin/$(OUT)_text.bin
	$(OBJCOPY) -O binary elf/$(OUT).elf -j .rodata -j .data bin/$(OUT)_data.bin

# Remove the BSP objects and the output ELF and BIN files
clean:
	rm -rf $(ASM_OBJS) $(C_OBJS) elf/$(OUT).elf bin/$(OUT)*.bin
```

D

RISC-V assembly code example

Listing D.1: RISC-V assembly code example.

```
00000994 <fib>:
 994:        fe010113          addi        sp,sp,-32
 998:        00112e23          sw        ra,28(sp)
 99c:        00812c23          sw        s0,24(sp)
 9a0:        00912a23          sw        s1,20(sp)
 9a4:        02010413          addi        s0,sp,32
 9a8:        fea42623          sw        a0,-20(s0)
 9ac:        fec42783          lw        a5,-20(s0)
 9b0:        00078a63          beqz        a5,9c4 <fib+0x30>
 9b4:        fec42703          lw        a4,-20(s0)
 9b8:        00100793          li        a5,1
 9bc:        00f70863          beq        a4,a5,9cc <fib+0x38>
 9c0:        0140006f          j        9d4 <fib+0x40>
 9c4:        00000793          li        a5,0
 9c8:        0380006f          j        a00 <fib+0x6c>
 9cc:        00100793          li        a5,1
 9d0:        0300006f          j        a00 <fib+0x6c>
 9d4:        fec42783          lw        a5,-20(s0)
 9d8:        fff78793          addi        a5,a5,-1
 9dc:        00078513          mv        a0,a5
 9e0:        fb5ff0ef          jal        ra,994 <fib>
 9e4:        00050493          mv        s1,a0
 9e8:        fec42783          lw        a5,-20(s0)
 9ec:        ffe78793          addi        a5,a5,-2
 9f0:        00078513          mv        a0,a5
 9f4:        fa1ff0ef          jal        ra,994 <fib>
 9f8:        00050793          mv        a5,a0
 9fc:        00f487b3          add        a5,s1,a5
 a00:        00078513          mv        a0,a5
 a04:        01c12083          lw        ra,28(sp)
 a08:        01812403          lw        s0,24(sp)
 a0c:        01412483          lw        s1,20(sp)
 a10:        02010113          addi        sp,sp,32
 a14:        00008067          ret
```

Listing D.2: MicroBlaze assembly code example.

```
0000068c <fib>:
 68c:       3021fff0        addik       r1, r1, -16
 690:       f9e10000        swi       r15, r1, 0
 694:       fa610008        swi       r19, r1, 8
 698:       fac1000c        swi       r22, r1, 12
 69c:       12610000        addk       r19, r1, r0
 6a0:       f8b30004        swi       r5, r19, 4
 6a4:       b0000000        imm       0
 6a8:       e86009b0        lwi       r3, r0, 2480        // 9b0 <c>
 6ac:       30630001        addik       r3, r3, 1
 6b0:       b0000000        imm       0
 6b4:       f86009b0        swi       r3, r0, 2480        // 9b0 <c>
 6b8:       e8730004        lwi       r3, r19, 4
 6bc:       bc030014        beqi       r3, 20                // 6d0
 6c0:       e8730004        lwi       r3, r19, 4
 6c4:       a8630001        xori       r3, r3, 1
 6c8:       bc030010        beqi       r3, 16                // 6d8
 6cc:       b8000014        bri       20                 // 6e0
 6d0:       10600000        addk       r3, r0, r0
 6d4:       b800003c        bri       60                 // 710
 6d8:       30600001        addik       r3, r0, 1
 6dc:       b8000034        bri       52                 // 710
 6e0:       e8730004        lwi       r3, r19, 4
 6e4:       3063ffff        addik       r3, r3, -1
 6e8:       10a30000        addk       r5, r3, r0
 6ec:       b9f4ffa0        brlid       r15, -96         // 68c <fib>
 6f0:       80000000        or       r0, r0, r0
 6f4:       12c30000        addk       r22, r3, r0
 6f8:       e8730004        lwi       r3, r19, 4
 6fc:       3063fffe        addik       r3, r3, -2
 700:       10a30000        addk       r5, r3, r0
 704:       b9f4ff88        brlid       r15, -120         // 68c <fib>
 708:       80000000        or       r0, r0, r0
 70c:       10761800        addk       r3, r22, r3
 710:       e9e10000        lwi       r15, r1, 0
 714:       10330000        addk       r1, r19, r0
 718:       ea610008        lwi       r19, r1, 8
 71c:       eac1000c        lwi       r22, r1, 12
 720:       30210010        addik       r1, r1, 16
 724:       b60f0008        rtsd       r15, 8
 728:       80000000        or       r0, r0, r0
```

E

Design Implementation

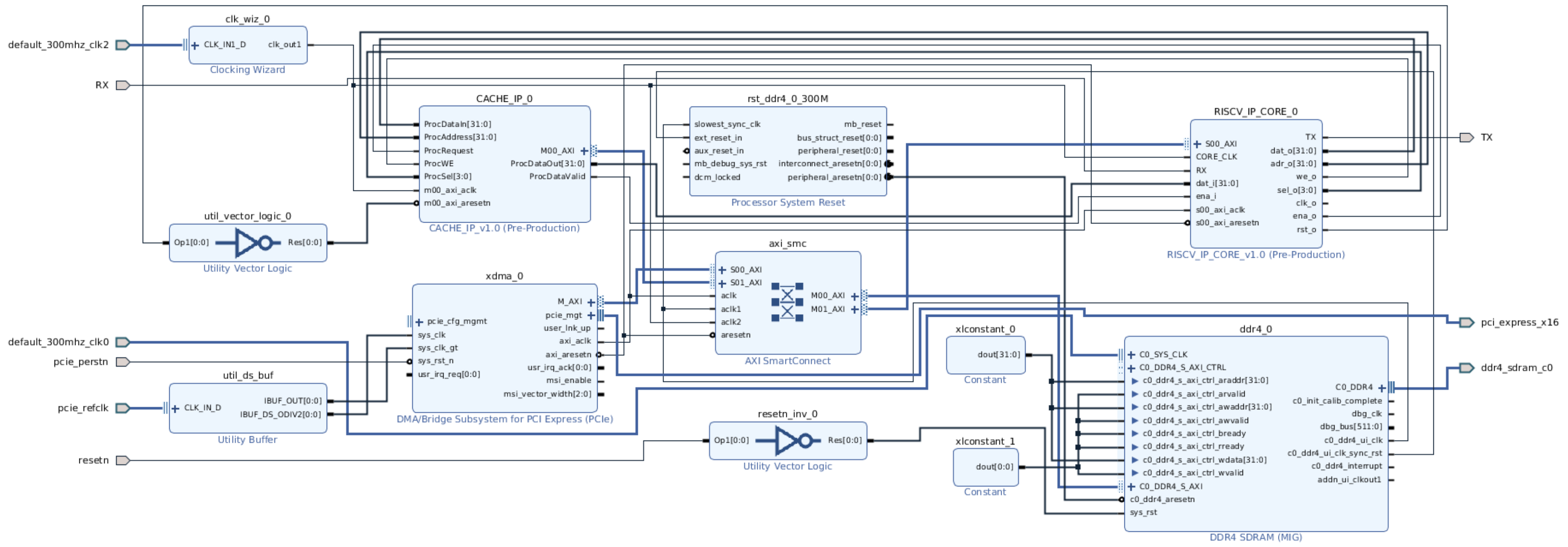


Figure E.1: Block design implementation of the proposed system created using Xilinx IP Integrator.