PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER ENGINEERING UNDERGRADUATE PROGRAM

PUC-RS5: A RISC-V PROCESSOR CORE FOR EMBEDDED USES

WILLIAN ANALDO NUNES

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Dr. Ney Laert Vilar Calazans
Co-Advisor: Marcos Luiggi Lemos Sartori

Porto Alegre
2022

For God and my family, their sacrifice and support made this journey possible.

“The things that we love tell us what we are.”
(St. Thomas Aquinas)

PUC-RS5: UM NÚCLEO PROCESSADOR RISC-V PARA USOS EMBARCADOS

RESUMO

A arquitetura RISC-V é modular e extensivel, sendo versátil para aplicar em múltiplos usos. A capacidade de atender interrupções e tratar exceções é essencial em sistemas embarcados como os usados em computação robótica, controle industrial ou sistemas veiculares, A arquitetura privilegiada do RISC-V define níveis de privilégio de operação e um conjunto de registradores de controle e de armazenamento de informação, estes últimos conhecidos como CSRs. Através de CSRs se dá o controle dos modos de privilégio e que se gere ações necessárias para realizar a manipulação de *armadilhas* (interrupções e exceções). Este trabalho descreve o projeto e implementação de PUC-RS5, um núcleo processador que implementa a ISA RV32I com a extensão Zicsr, para dar um suporte mínimo à arquitetura privilegiada RISC-V. O modo de privilégio de máquina é suportado pelo núcleo PUC-RS5; o suporte a outros modos é considerado um relevante trabalho futuro. A organização PUC-RS5 é um pipeline com emissão única, de 4 estágios, dotado de organização Harvard. Define-se para este núcleo uma interface com ambiente de execução (EEI), que é prototipada em uma plataforma baseada em FPGA juntamente com o núcleo. Neste ambiente integram-se mecanismos capazes de gerar interrupções, como timers e botões externos para validar as rotinas de tratamento de armadilhas. PUC-RS5 é uma implementação de código aberto, minimalista, extensível e de baixo custo em área. Quando comparado com processadores descritos na literatura disponível, apresenta resultados comparáveis aos que lhe são próximos em funcionalidade. Sua operação é demonstrada sobre a plataforma Nexys A7 com um FPGA xc7a100tcsg324-1 da Xilinx, utilizando neste dispositivo 1542 LUTs e 814 FFs. O PUC-RS5 é uma evolução de trabalhos anteriores, demonstrando funcionalidade adicional, qual seja a possibilidade de tratar interrupções e exceções. O núcleo PUC-RS5 foi validado por simulação e em hardware, executando com sucesso a suíte de Berkeley, o conjunto de programas de teste Coremark, e outros aplicativos simples desenvolvidos para este fim.

Palavras-Chave: Sistemas embarcados, RISC-V, Arquitetura do conjunto de instruções, ISA, Arquitetura privilegiada, Núcleo processador.

PUC-RS5: A RISC-V PROCESSOR CORE FOR EMBEDDED USES

ABSTRACT

The RISC-V architecture is modular and extensible, being versatile enough to apply in multiple uses. The ability to service interrupts and handle exceptions is essential in embedded systems such as those used in robotic computing, industrial control, or vehicular systems. The privileged architecture of RISC-V defines operating privilege levels and a set of control and information storage registers, the later known as CSRs. Through CSRs, the privilege modes are controlled and the necessary actions are generated to carry out the manipulation of traps (interrupts and exceptions). This work describes the design and implementation of PUC-RS5, a processor core that implements the ISA RV32I with the Zicsr extension, to provide minimal support for the privileged RISC-V architecture. The machine privilege mode is supported by the PUC-RS5 core; support to other modes is a relevant future work. The PUC-RS5 organization is a 4-stage, single-issue pipeline with Harvard organization. An execution environment interface (EEI) is defined for the core, which is prototyped on an FPGA-based platform along with the core. In this environment, mechanisms capable of generating interrupts are integrated, such as timers and external buttons to validate trap handling routines. PUC-RS5 is an open source, minimalist, extensible and with low area cost implementation. When compared to processors described in the available literature, it presents results comparable to those that are close to it in functionality. Its operation is demonstrated on the Nexys A7 platform with an FPGA xc7a100tcsg324-1 from Xilinx, using 1542 LUTs and 814 FFs in this device. PUC-RS5 is an evolution of previous works, demonstrating additional functionality, namely the capacity to handle interruptions and exceptions. The PUC-RS5 core has been both simulation- and hardware-validated, successfully running the Berkeley suite, the Coremark test suite, and other simple applications developed for this purpose.

Keywords: Embedded systems, RISC-V, Instruction set architecture, ISA, Privileged architecture, Processor core.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS

ALU – Arithmetic Logic Unit

APPINJ – Application Injectors

ARM – Advanced RISC Machines

ASCEND – Asynchronous Standard Cells for ‘n’ Designs

ASIC – Application Specific Integrated Circuit

BGEZ – Branch if Greater or Equal to Zero

BOOM – Berkeley Out-of-Order Machine

BRAM – Block Random Access Memory

CISC – Complex Instruction Set Computer

CLB – Configurable Logic Block

CLIC – Core Local Interrupt Controller

CLINT – Core Local Interrupter

CPU – Central Processing Unit

CSR – Control and Status Register

DMA – Direct Memory Access

DMNI – Direct Memory Network Interface

FF – Flip-Flop

FIFO – First-In-First-Out

FPGA – Field Programmable Gate Array

FSM – Finite State Machine

GUI – Graphic User Interface

GPIO – General Purpose Input/Output

GPPC – General Purpose Processing Cores

GPU – Graphics Processing Unit

HART – Hardware Thread

HDL – Hardware Description Language

HRT – Hard Real-Time

IC – Integrated Circuit

ID – Instruction Decoder

IOT – Internet of Things

IP – Intellectual Property

IR – Interrupt Register

IRQ – Interrupt Request

ISA – Instruction Set Architecture

LUT – Look Up Table

MBC – Memory and Bus Controller

MCAUSE – Machine Cause Register

MEMPHIS – Many-core Modeling Platform for Heterogeneous SoCs

MEPC – Machine Exception Program Counter

MIE – Machine Interrupt Enable

MIP – Machine Interrupt Pending

MPIE – Machine Previous Interrupt Enable

MPP – Machine Previous Privilege

MPSOC – Multi-Processor System-on-Chip

NI – Network Interface

NRT – Non-Real-Time

PC – Program Counter

PE – Processing Elements

PL – Programmable Logic

PLIC – Platform Level Interrupt Controller

PS – Processing System

PUC-RS5 – PUCRS RISC-V Privileged Architecture

PUCRS – Pontifical Catholic University of Rio Grande do Sul

PUCRS-RV – PUCRS RISC-V

PULP – Parallel Ultra Low Power

QDI – Quasi Delay Insensitive

RAM – Random Access Memory

RF – Register File

RISC – Reduced Instruction Set Computer

ROM – Read Only Memory

ROS – Robotic Operation System

RTL – Register Transfer Level

SOC – System-on-Chip

SPI – Serial Peripheral Interface

SRT – Soft Real-Time

SW – Store Word

UART – Universal Asynchronous Receiver Transmitter

VLSI – Very Large Scale Integration

XU – Execute Unit

CONTENTS

1. INTRODUCTION AND MOTIVATION

Simply said, a *processor* is a piece of digital hardware capable of executing software. Today, processors are essential parts to build most, if not all, digital systems. The variety of distinct processors available is huge and everyday sees the proposal of new processors for specific or general purpose uses.

The study of processors is now a well-established and fundamental segment of courses like Computer Science and Computer Engineering. Such study often comprises approaching processors from two intertwined abstract points of view: *processor architecture* and *processor organization*. *Processor architecture* studies processors as machines that contain a set of relevant registers to store data and control information, that can execute a set of instructions, with specified formats (source and binary), and that interacts with external storage components (memories) where sequences of instructions and data lie. *Processor organization*, in turn, relates more closely to digital hardware design and with the process of mapping a processor architecture to a set of digital hardware components that implement the architecture. The present work is about the implementation of a specific processor core organization, called PUC-RS5 of a given processor architecture, the RISC-V RV32I. The PUC-RS5 has as target its use in digital embedded systems for multiples uses, typically in sensor-rich environments like robotic equipment and/or in multi- and many-core chip multiprocessors for generic or specific applications.

A *processor core* or just a *core* as this work often refers, is a software executing unit that can be a part of multiple environments. A core is often a fraction of an integrated circuit (IC), but not necessarily. Also it often connects to external components that communicate with it and complement its functionality. These components can be memories, communication modules, or even other processor cores. When this last situation occurs the overall system might be referenced as a *multi-core*. Given the current advance in microelectronics technologies, even complex cores can be embedded in large numbers inside a single small silicon subtract with say just one or a few square millimeters.

An *embedded system* is a computational system that combines hardware and software and is designed to perform a specific set of functions. These systems can have a fixed functionality or be programmable. Embedded systems can vary in complexity, they can be simple systems such as those present in many devices of daily use such as remote controls, keyboards, digital watches, earphones, etc. But they are also present in complex and critical applications, such as industrial machinery, complex medical equipment, airplanes, ground vehicles etc. Embedded systems usually count with sensors and/or actuators that exchange information with the environment. Sensors are used to obtain environmental parameters that might be relevant for the correct operation of the system. They convert physical data into electric signals that can be encoded, interpreted and processed. Embedded systems are often used in Internet of Things (IoT) devices, which are Internet-connected devices that can be accessed remotely and thus do not require humans to locally operate them. IoT devices are becoming popular and keep boosting e.g. the popularization of smart houses, smart buildings, and smart electric grids. More critical applications often rely on a Real-Time Operating System (RTOS) a kind of operating system that has the ability to handle multiple concurrent events and respect a set of defined deadlines for the execution of some or all of its tasks.

Processor organizations can vary widely depending on the uses expected by the processor. For example, it can include a simple single core used only for communicating data to and from larger systems, or they can compose complex many-core systems responsible for an convoluted computations such as those executed by artificial neural networks. The features to implement in an processor core organization must be decided at a very early stage in the project of a system.

The RISC-V Instruction Set Architecture (ISA) set of specifications has a big advantage over commercial ISAs when it comes to access to hardware, due to its openness and associated features. The main reason for the rapid RISC-V popularization is that it is defined as an open standard, integrated ISA set, which implies the ample availability of public documents targeting every detail of each feature and its respective implementation, and also a huge amount of worldwide contributors to the standard definition. The active community of researchers, engineers, laboratories and enterprises is able to respond and contribute to the improvement of RISC-V in very fast ways. Another reason that makes RISC-V an excellent choice when it comes to projects is the modularity and extensibility of its definitions. RISC-V is divided into very well-defined and well-documented parts called *base* and *extension* modules [WA19]. A *base module* is in fact a complete ISA. Extensions are specific sets of instructions, grouped by functionality. The base RISC-V ISA modules differ among them in the overall characteristics such as instruction length and register bank configurations. Maybe the most emblematic RISC-V module is the RV32I Base Integer Instruction Set. This is the simplest ISA and its instructions are mandatory in every implementation of a RISC-V core. RV32I defines instructions used for the most operations in general software such as arithmetic and logic computations, control flow operations, and memory access actions. Besides RV32I, the latest version of the RISC-V Instruction Set Manual (Unprivileged ISA) [WA19] defines two other ratified ISAs, RV64I and RISC-V Weak Memory Ordering (RVWMO), and also announces two draft ISAs, RV32E and RV128I.

RISC-V *extension modules* are optional and bring specific functionalities that complement some base ISA. The current Unprivileged ISA Manual (V.20191213) lists RISC-V 17 extensions, 8 of these already ratified, one frozen (i.e. with little room for changing before ratification) and 8 in draft status. A more recent information, available at the site https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions (as of November, 2022), lists 14 new ratified classes of extensions that count with 54 extension variations as a whole.

These numbers attest the dinamicity of the research and development community around the RISC-V effort in bringing a strong open source alternative to processor architecture standardization. Among the most used ratified extensions it is worth citing here: the Multiplication and Division Extension (M), which introduces instructions for integer multiplication and division; the Single-Precision Floating-Point Extension (F) introduces instructions that can handle floating-point operations and that is the base for the another two extensions, Double-Precision (D) and Quad-Precision (Q). Among the extensions defined for supporting the interaction between a RISC-V core and the external world, there is the Zicsr extension that defines instructions that manipulate Control and Status Registers (CSRs) and defines the structure and function of the CSRs themselves. The Zicsr instructions perform atomic operations in CSRs and are essential for the development of RISC-V organizations able to interact in an orderly fashion with peripheral devices and systems. The fact that RISC-V is an open-source standard allows the community to interfere in the production of new extensions accordingly to specific needs, which allows proposing instructions and modules applicable to specific domains, to achieve performance gains or other gains without the need of rely on standard instructions only.

RISC-V also counts with a privileged architecture definition [WAH21] that standardizes the set of CSRs, and also defines and standardizes three privilege levels a RISC-V core can operate. Privilege modes add the ability to have different levels of trust for executing code, which enables the separation of user applications from operating system (OS) modules, among other software classes. The CSR set defined in the privileged architecture is also used extensively in interrupt and exception handling. At this point it is interesting to introduce a precise definition of interrupts, exceptions and traps. First, it is necessary to establish some previous RISC-V definitions [WA19], those of *core* and *hart*:

> A component is termed a *core* if it contains an independent instruction fetch unit. A RISC-V compatible core might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading.

Now, citing the Volume I RISC-V manual [WA19] there is said:

> We use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart. We use the term *interrupt* to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term *trap* to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.

This work adopts the above definitions. Here, interrupt and exception handling is collectively called *trap handling*. Trap handling uses status flags present in CSRs and obeys to predefined behaviors. The privileged architecture defines the steps to use in a handling routine, and all CSRs changes triggered by every trap. The ability to treat interrupts and exceptions through traps adds robustness to a processor core and allows its use in critical applications. Interrupts are an essential part of a core design as it adds to the core the possibility of efficient communication with sensors, actuators and timers, which are all integral parts of a real-world environment.

1.1 Motivation

The previous paragraphs already advanced a generic set of motivations to employ and invest in dominating RISC-V.

A more internal motivation for developing an extensible RISC-V processor core arises from the long term participation of the Author in research activities in laboratories of hardware design of the Pontifical Catholic University of Rio Grande do Sul (PUCRS), which can benefit from the contributions developed here. These activities relate to applications operating on environments which require interaction with sensors and actuators. Using cores already available in the literature revealed challenging in several occasions, leading to the proposal of an in-house RISC-V core implementation.

A RISC-V core must be easy to embed in systems and have low area usage, to allow its use in multiple environments requiring core replication and/or low power footprint. Abundance of FPGA boards in the laboratories guarantee the possible to test cores easily in real-word devices.

Another motivation for this Work is to provide a guide on how to use and extend the proposed PUC-RS5 core. Developing source code in a simple and well documented way enables other members of the laboratories to use and extend the core seamlessly. This will bring advantages with regard to external open-source RISC-V implementations, such as those available as PULP Platforms (e.g. RI5CY, Ibex). These implementations are often full of complex parameters to enable the construction of multiple organizations from a single code base, with features that can be added or removed, bringing difficulties for the code understanding and extension.

Writing this work in English is a good way for the Author to practice with technical terminology and writing skills, a challenge that motivates the production of a technical report as a Bachelor Thesis.

1.2 Objectives and Contributions

The main contribution of this work is a processor core that implements the RISC-V RV32I ISA with minimal support to efficiently handle external peripherals. It is implied in this support the capacity to handle interrupts and exceptions in a standard and efficient way, as described in the RISC-V privileged architecture specification [WAH21]. The generated core is called PUC-RS5.

A requirement for achieving efficient and standard external peripherals manipulation is to integrate all standard Zicsr extension instructions into a RISC-V pipeline organization and provide a set of control and status registers (CSRs) to control input/output (I/O) processes.

Note that, although the basic support to implement multiple privilege levels as described in [WAH21] is present in the PUC-RS5 core, there is no effort to implement privilege levels distinct from the *machine level*. Support to other privilege levels that can enable the core to be used in environments with adequate operating systems (OSs), supervisors or hypervisors is a relevant future work.

An objective of the work is that the proposed processor core, PUC-RS5, be a valid description, amenable to hardware implementation. To guarantee the achievement of this objective a hardware prototyping process demonstrates the PUC-RS5 operation. The process uses an FPGA-based platform.

The rest of this document is organized in seven additional chapters. Chapter 2 brings some important concepts for understanding this work, including a brief overview of the RISC-V effort, a previous simple core implemented by the Author, PUCRS-RV, and addresses a few application fields that PUC-RS5 can target. Chapter 3 briefly reviews the literature on processor cores with target similar to those of PUC-RS5. It also explores some works that employ RISC-V cores for robotics applications and many-core environments, expectedly good target fields for PUC-RS5. Chapter 4 presents a small history of the process that leads to the PUC-RS5 proposition and gives a generic set of aspects that differentiate PUC-RS5 from previous efforts. Chapter 5 details the process of implementing all PUC-RS5 proposed features, serving as a guide on understanding and extending the PUC-RS5 core. Chapter 6 covers the validation environment developed for PUC-RS5 core its FPGA prototyping process. Chapter 7 compares implementation data obtained during the PUC-RS5 prototyping with results from similar RISC-V cores and details a set of conducted experiments in running software with it. Chapter 8 brings a set of conclusions of the work and presents some relevant topics to explore as immediate future work.

2. BACKGROUND

This Chapter provides an introduction to topics required to fully understand this work: Section 2.1 introduces the concept of Instruction Set Architectures (ISA) in general and a more detailed overview of the RISC-V architecture [WA19]. Section 2.2 addresses some details on the RISC-V privileged architecture [WAH21]; Section 2.3 introduces PUCRS-RV, an implementation of the RISC-V RV32I ISA as a processor core developed by the Author, and which is the base of this work. Section 2.4 introduces some possible embedded uses where RISC-V processor cores and other cores can be used.

2.1 Instruction Set Architectures - ISAs

An ISA is an abstract model of a processor. A specific ISA establishes sets of instructions, registers, memory access features, and other characteristics which defines a software-executing abstract machine. The hardware that implements the abstractions defined by an ISA is called a *processor organization*. Such hardware is capable of decoding and executing programs written using *instructions* of some ISA. An ISA defines the functionality of instructions without stating how to precisely implement such functionalities, leaving implementation details to be defined by the organization design. For example, an ISA that contains a instruction that multiplies two integers defines the sizes of operands, where and how to find these, the size of the operation result and where to store it. Different organizations for this ISA can implement these instructions in different forms, with a wide variation of hardware sizes, performance and power characteristics.

This enables multiple implementations of the same ISA, allowing different kinds of machines to be able to run the same code.

A common criterion used to classify ISAs relies on how the machine accesses external memory during instruction execution. The criterion falls into two ISA classes:

1. Complex Instruction Set Computers (CISC) - these correspond to an older way to implement processor that emphasizes: (i) hardware-powerful implementations; (ii) multi-step complex instructions in addition to single-step instructions. A single CISC instruction can e.g. dictate the fetch of multiple data from external memory, perform some operation (say arithmetic or logic) on these data and store results back in memory; (iii) smaller program sizes; (iv) small number of architectural working registers (typically from 1 to less than 10 data registers);

2. Reduced Instruction Set Computer (RISC)[1] - this is a more modern way to define processors, which emphasizes: (i) software-efficient implementations; (ii) only simple instructions, easily executed by simple hardware. One of the main characteristics that differentiate RISC from CISC machines is the use of a *load-store* philosophy, where reading and writing from/to external memory is a task of specific instructions, completely separated from instructions that process data with e.g. arithmetic and logic instructions; (iii) RISC organizations usually require larger program sizes compared to CISC organizations; (iv) RISC organizations rely on a large number of working registers (16, 32 or even much more).

The RISC-V ISA stands out by its simplicity. It is a RISC set of ISAs, implying instructions easy to understand and implement in hardware. It is a modular ISA, which makes its design usually more flexible and extensible. It is an open standard, implying the enabling of royalty-free processor core development

2.1.1 The RISC-V ISA Set

The most popular commercial ISAs today are the Intel X86 and the ARM set of ISAs, both of which are proprietary and subject to the payment of royalties for their use. Of course, there are some advantages in adopting these ISAs, since they are popular architectures, have a big set of available documentation and there is a huge software base built with these as target.

Open standards, on the other hand, give the community the ability to develop the technology. They attract many people from academia and industry, because are free to use and also the engaged community grows fast, with increasing numbers of people working to make it more accessible and more popular. The RISC-V is a modern, open, and extensible ISA set, initially designed at the Berkeley University that was created aiming to be an open standard ISA. Today RISC-V is controlled by RISC-V International, a global non-profit organization (see https://riscv.org/about/). RISC-V aims at being simple, flexible and extensible, being generic enough to be scalable for a huge variety of applications, which is achieved by defining a few basic set of instructions that is mandatory in every implementation and by defining several optional extensions.

The RISC-V ISA set is defined in its two-volume Instruction Set Manual [WA19, WAH21]. Table 2.1, extracted from [WA19] contains the current list of modules reinforced by the RISC-V standard. The Base modules, except for RVWMO, correspond to the ISAs options available for implementation. RVWMO defines one option (the simplest one) for pro-

[1]Note that the RISC name is today a misleading denomination, even if the first RISC machines displayed such characteristic (having a small number of instructions compared to CISC architectures). RISC instruction sets need not, and often are not, smaller that those of CISC architectures.

viding a memory consistency model to RISC-V implementations, which is specially relevant for multiprocessor designs and is ignored here.

Table 2.1 – The currently defined RISC-V modules, divided into Base and Extension ones. Note that modules can be in the Ratified, Frozen or Draft status.

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*

Extension	Version	Status
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

Extensions are ways of improving a base module with additional sets of standard instructions. The most relevant extensions are M, A, F and D, which are collectively known a the G extension set. These extensions add integer multiply and division instructions (M), atomic read-modify-write memory instructions (A) single-precision and double-precision floating point registers, and instructions (F and D). The extension Zicsr adds a set of instructions that operate atomically in the core CSRs. Among the other ratified extensions are the Quad-Precision Floating-Point extension (Q), Compressed Instructions extension (C) and Instruction-Fetch Fence (Zifencei). To inform the compiler of which extensions are available in the target core, compilation flags are used indicating the sets of instructions that the generated code can contain.

This work describes a minimal hardware implementation, including just the RV32I base module and the Zicsr extension.

The RISC-V project that begins in 2010 continues to receive new features and new extensions. See, for example [RIS22], a repository that contains the latest changes in the

RISC-V and shows the set of extensions that will be eventually merged into the final specifications in the future. All extensions are optional and each one has a status of development. The modules marked as *Frozen* are not expected to change significantly before being ratified. The modules marked as *Draft* are expected to change before ratification. The Modules marked as *Ratified* are ratified at the present time and will (hopefully) never change.

Table 2.1 and the [RIS22] repository shows how the RISC-V is an evolving concept. It has stable features, but also contains several volatile features and is today in a process of continuous evolution. One proof of this is that there are 13 extensions ratified in years 2021 and 2022 (see https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions) but were not yet integrated yet into the formal specifications.

2.2 The RISC-V Privileged Architecture

The RISC-V privileged architecture is a complement of base ISAs. It defines aspects such as privilege levels and functionalities required to run operating systems and attach external devices.

Privilege levels encapsulate different levels of software permissions. There are three defined privileges: (i) Machine (M) - this mode is the most privileged level and its implementation is mandatory in a privileged architecture. Code running in machine mode is inherently trusted; (ii) Supervisor (S) - is the second privilege level and is often used by operating systems; (iii) User (U) - this mode is the less trustable mode and has the lowest privilege, its operation is restricted. The privilege levels and their encoding is presented in Table 2.2. Note there is an undefined but reserved for future use level, the level 2 privilege.

Table 2.2 – RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	`00`	User/Application	U
1	`01`	Supervisor	S
2	`10`	*Reserved*	
3	`11`	Machine	M

All RISC-V hardware implementations must support the M mode, as this is the most privileged mode, which has access to all the hart functionalities. All code that runs in machine mode is considered trusted, it will not provide any protection against incorrect or malicious application code. The user mode U is the most common option besides M, since it protects the rest of the system from application code. The hart usually initializes in the M mode, where boot configurations take place. Next the mode switches to U, if available. The U mode runs applications until a trap occurs, forcing the privilege to change to a higher level.

When a trap occurs the privilege changes and the hart branches to a trap handler routine. The trap handler filters the cause of the trap and performs adequate treatment of that kind of trap.

The privileged architecture specifies the Control Status Registers (CSRs) responsible for keeping track of the actual privilege and the status of the hart. It requires the RISC-V Zicsr extension, which introduces instructions to access and control the status of CSRs. All Zicsr instructions are atomic, meaning they read and write CSRs in a single instruction. Each privilege level has its own set of CSRs, and each set can be accessed by its privilege or by a higher privilege one. If an application tries to access a CSR of a set that belongs to a higher privilege mode, this causes an exception.

2.2.1 RISC-V Interrupt Organization Modes

The ability of handling interrupts can be implemented in different forms, depending on the desired application and they vary in complexity. Unfortunately, this is a subject not yet regulated in a ratified way by RISC-V formal specifications. There are drafts of interrupt modes currently under study, and some have already been used by more than one RISC-V implementation. This work chooses to employ interrupt treatment by adopting one of the interrupt treatment proposition options pleaded by the SiFive, Inc. company a long-term adopter of RISC-V. SiFive describes interrupt treatment in detail in [SiF20]. It describes three interrupt controller types: (i) the Core Local Interrupter (CLINT); (ii) the Core Local Interrupt Controller (CLIC); and (iii) the Platform Level Interrupt Controller (PLIC). The first two are local interrupt controllers that provide low-latency interrupt treatment to the hart and the last is a global interrupt controller that can act at a platform level. The differences and some details on each approach are the subject of next sections.

Core Local Interrupter (CLINT)

CLINT uses a fixed priority for software, timer, and external interrupts. The interrupt ID represents the fixed priority value of each interrupt and is predefined. There are two different CLINT modes of operation, the direct mode and the vectored mode. To configure CLINT modes, the *Mode* field of the MTVEC CSR holds the actual configuration. For direct mode, the Mode field is set to 0, and for vectored mode Mode field is set to 1. The MTVEC CSR in the *Base* field holds the base address for trap handling in both modes. When the Mode is set to “direct”, all interrupts and exceptions trap to the same handler located at the base address in the MTVEC CSR. This value will be loaded into the Program Counter register, branching to a generic trap handler that will evaluate the trap cause and jump to the specific trap handler. In the vectored mode, a vector table is used for lower interrupt handling latency. When a trap occurs, the trap cause code is used to index the vector table for the corresponding handler, based on the interrupt ID. This means that when an interrupt occurs, program execution jumps directly to the place pointed to by the vector table offset of the corresponding interrupt.

Core Local Interrupt Controller (CLIC)

CLIC has a more flexible configuration than CLINT. However, CLINT has a smaller design overall. CLIC can display reverse compatibility with CLINT when programmed in the CLINT mode. CLIC has programmable interrupt levels and priorities, introducing new direct and vectored modes. CLIC introduces new CSRs that are used for CLIC's own configuration.

The CLIC direct mode takes an approach similar to CLINT direct mode, but with the addition of a feature called *selective vectoring*. Selective vectoring allows each interrupt to be configured for CLIC hardware vectored operation, while all other interrupts use the CLIC direct mode. The CLIC CSR *CLICINTCFG* is used to configure selective vectoring. CLIC direct modes use MTVEC as the base address for exception and interrupt handling, but introduces CLIC CSR *MTVT* as the base address for the interrupts configured for selective hardware vectoring.

The vectored mode is a concept similar to the CLINT vectored mode, where an interrupt vector the table is used for specific interrupts. In CLIC vectored mode, the handler table contains the address of the interrupt handler instead of a jump instruction. CLIC vectored mode uses MTVEC exclusively for exception handling and the CLIC CSR MTVT is used to define the base address for all vectored interrupts.

CLIC allows programmable interrupt levels and priorities for all supported interrupts. For that, it replaces the MIE, MIP, and MIEDELEG with new CLIC CSRs: CLICINTIE, CLICINTIP, and CICINTCFG.

Platform Level Interrupt Controller (PLIC)

PLIC is used to manage global interrupts and route them to one or several CPUs in the system. PLIC can route a single interrupt source to multiple CPUs. In PLIC, each interrupt has a configurable priority from 1-7, with 7 being the highest priority and the value 0 disables an interrupt. PLIC can be combined with CLINT and CLIC in multi-CPU systems, being the manager of the external interrupts for the local interrupt controllers.

2.3 The PUCRS-RV Processor Core

The PUCRS-RV is a processor core that implements the RISC-V RV32I ISA, developed by the Author. It was first described in [LNZ+22], consisting of a single-issue, five-stage pipeline organization as follows:

1. Instruction fetch;
2. Instruction decoding;
3. Operand fetch;
4. Execution;
5. Retirement.

The instruction flow of PUCRS-RV is controlled by a system of tags. Every time it fetches an instruction, it is associated with a tag that will follow it until retirement. The Retirement Stage has a tag initialized with the same value as the tag in the Instruction Fetch Stage. Every time a jump/branch is taken, the tag is incremented in the Retirement Stage first, since it is this stage that performs the branch. When a branch is obtained in the Instruction Fetch Stage its tag is also incremented. Instructions in the pipeline that follow an instruction that branches are associated with the older branch. When these arrive at the Retirement Stage, its tag and the Retirement Stage tag will not match, and the effects of the instruction will not be effectuated. For this system to work, the branch prediction strategy must use a "never taken" policy.

To detect Data Hazards the PUCRS-RV processor core has in the third stage a queue mechanism that stores the registers having pending writes (instructions on the fourth or fifth stage of the pipeline that will write on that register), this register is called a "locked register". Every time an instruction enters the third stage and one of its source registers is a locked register that indicates a data conflict/Hazard, in this case, a bubble (a NOP instruction) is issued forward until the conflict is resolved and the instruction can read a proper operand.

2.4 Examples of Embedded Uses for Processors

Processor cores can be used in many kinds of applications. The most common and known processor cores are used in general-purpose computers. Such cores have to display features supporting many kinds of applications and needs that a general user may have. Embedded processor cores take a different approach as they seek to be cheaper and are more application specific. To be cheaper, one thing these cores seek is to have a low area, to reduce production costs. Another required feature of embedded cores is low power consumption, due to the fact that embedded platforms many times operate with limited power resources, usually a battery or even less than that. An embedded core that consumes less battery extends or even enables its use. This said, the design of a compact and low-power processor core is essential to consider its use in embedded applications. This Section brings an overview of two possible classes of applications that a core can be applied and how they relate to the PUC-RS5 proposed core. Section 2.4.1 explores needs present in robotic applications and Section 2.4.2 presents an overview of the use of cores in many-core platforms.

2.4.1 Robotic Applications

The idea of developing the PUC-RS5 core for robotic applications arises from the participation of the Author in the *Laboratório de Sistemas Autônomos (LSA)* of the PUCRS University. This lab aims to build autonomous systems that can be applied in real-life situations, such as autonomous boats, drones, and robots. The team of this lab is integrated by undergraduate students, graduate students and researchers. As often systems are built and embedded in environments, the idea of having a customizable core took shape and encouraged the enhancement of the Author's available PUCRS-RV core. PUC-RS5 can already be used as an FPGA prototype to provide hardware acceleration for specific tasks.

Robotic applications are usually sensor-rich and input-output rich, as the environment where they operate generally produces large amounts of information that are required to process for a robot to realize its mission autonomously. Sensors can produce multiple kinds of interrupts and exceptions, thus the ability to treat exceptions and serve interrupts is essential for this kind of application. These features are present in the PUC-RS5 as a main goal.

Sensor data are used by algorithms for determining the robot operation. For example, a software implementing machine learning for image and location recognition may be a target. Also, packages like the Robotic Operating System (ROS) may be used in conjunction with sensor data for robot navigation in the environment. Sensor data are read in interrupt handling routines and processed. Next, actuator data can then be generated and applied to external peripherals. These actions can used to control speed, position, etc. of a robot. Section 5.6 briefly discusses the minimum required software stack to provide support to interrupt and exception handling.

2.4.2 Multi-Core/Many-Core Applications

Multi-core or Many-core applications are those that use more than one instance of one or more processor cores through association of heterogeneous cores or replication of identical cores, respectively. Cores can be connected through a shared bus that implements some protocol such as ARM AXI or through an intrachip network, also known as a network-on-chip (NoC) that uses routers for message exchange among cores. Cores interact with each other for information exchange, bus control, synchronization etc. This can be performed in systems with shared memory or by having a system based in message exchange only. For enabling core replication in embedded systems, it is necessary to have cores with low area usage to enable the possibility of having Networks-On-Chip (NoCs) with a many-core approach.

In many applications that use multi- or many-core systems, a method called Direct Memory Access (DMA) is used for data transfer that allows peripherals (or other cores) to access and write data directly to the memory of a core without the need for the target core branching to treatment routines until the transfer is complete. When the transfer operation is complete, the DMA controller communicates the core the end of the transfer operation through an interrupt. The core then can employ the transferred data and operate on them. The DMA method is quite useful in scenarios requiring the transfer of large amounts of data into the core memory. It only takes that the core program the DMA controller to perform the transfer, reducing the core computational load.

One devised future application for the PUC-RS5 core is to become the new processor core to use in the Memphis many-core [RCFM19]. This many-core is a configurable system where a possibly large numbers of cores is integrated by an NoC, creating a bi-dimensional mesh arrangement of cores into processing tiles. Each tile comprises four modules: (i) a processor core; (ii) a local scratchpad memory, a DMA/Network Interface module (name DMNI); and (iv) a 5-port NoC router. Each processor runs a multi-tasking micro-kernel software, containing the basic processing features to allow operations such as inter-core communication, external system communication, user applications reception, execution and migration, etc.

3. STATE OF THE ART

This Chapter brings an overview of the state of the art on some RISC-V privileged architecture cores similar to the PUC-RS5 propose here, in Section 3.1. Next, it gives an initial overview of the few works that already employed RISC-V architectures in robotic environments in Section 3.2, and revises an effort of the GAPH research group able to benefit from this work, namely the Memphis platform in Section 3.3.

3.1 Similar Processor Cores

Several projects proposing RISC-V cores already exist, and many such efforts are under way. This Section revises implementations similar in features to the PUC-RS5 cores, so that a comparison of PUC-RS5 to these is applicable.

3.1.1 The Parallel Ultra Low Power (PULP) Platform

The Parallel Ultra Low Power (PULP) Platform is a project created in 2013 as a joint effort of ETH Zurich and the University of Bologna, dedicated to develop RISC-V cores with target on low power consumption. This is an open-source platform that released 32 and 64-bit implementations of the RISC-V ISA and also some Systems-on-Chip (SoCs) that uses these cores. Some of its RISC-V implementations have been taped-out as ASICs in silicon that are used on IoT and low-power projects.

There are three cores currently provided by PULP: *CV32E40P*, *CVA6*, and *Ibex*. The *CV32E40P* was formerly called *RI5CY*, it is a 32-bit, 4-stage, in-order RISC-V core that implements the RV32I ISA with extensions M and C, which has an optional 32-bit FPU hardware supporting the F extension and instruction set extensions for DSP operations. It implements some custom extensions to achieve higher performance and energy efficiency. The *CVA6* core, formerly called *Ariane* is a 6-stage, single issue, in-order 64-bit CPU which implements the RV64I ISA and extensions M, C and D. CVA6 implements the three privilege levels M, S, and U, to fully support Unix-like operating systems. The Ibex Core is more detailed in Section 3.1.2.

The single-core SoCs provided by PULP are two: *PULPino* and *PULPissimo* PULPino is an open-source single-core micro-controller that uses either the Ibex core or the CV32E40P core and contains a set of peripherals, including interfaces I2S, I2C, SPI, and UART. PULPissimo is a more advanced and more complex architecture compared to PULPino. It includes the support to allow peripherals to copy data directly to memory using a simple DMA called µDMA.

3.1.2 Ibex core

Ibex [Low21] was initially developed as part of the PULP platform which was formerly called "Zero-riscy". It is currently maintained by the lowRISC non-profit company located at the University of Cambridge Computer Laboratory (https://lowrisc.org/) and is under active development. Ibex is a 32-bit RISC-V core written in System Verilog. It is heavily parameterized allowing it to be used in many embedded applications. Ibex supports the Integer (RV32I) or Embedded (RV32E) ISAs and offers support to the integer multiplication and division (M), compressed (C), and b (Bit Manipulation) extensions. It is a 2-stage, in-order, single-issue pipeline. The Ibex core implements the privileged architecture and supports the M and U privilege levels.

Figure 3.1 presents the general organization of the Ibex 2-Stage pipeline.

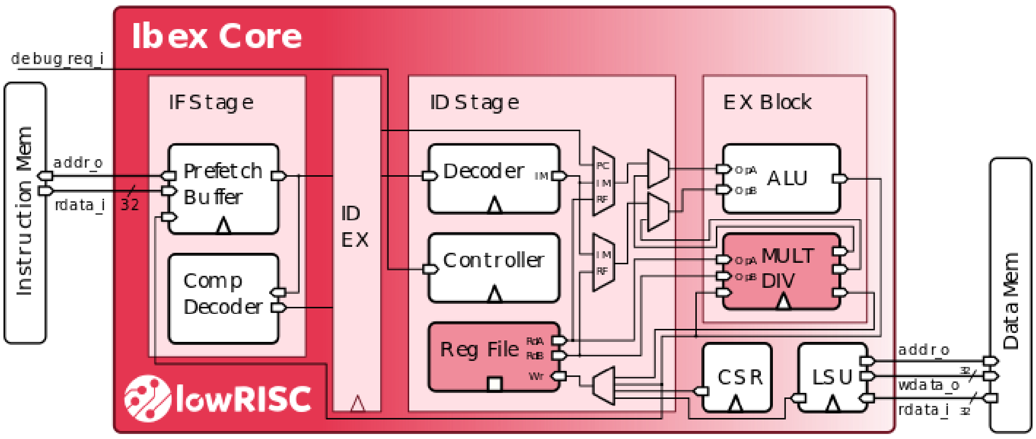


Figure 3.1 – The Ibex RISC-V core pipeline organization [Low21].

The Ibex core can be used in FPGA RTL simulation and ASIC synthesis, having the characteristics of being *ultra-low-power* and *ultra-low-area*. It can also be configured in a way to reduce the register bank size, implementing the RV32E, instead of the RV32I ISA.

3.1.3 SCR1 core

The SCR1 core [Syn22] is maintained by the Syntacore company from Cambridge (UK). SCR1 is the simplest of seven cores offered by Syntacore (from SCR1 to SCR7) and it overall organization is depicted in Figure 3.2.

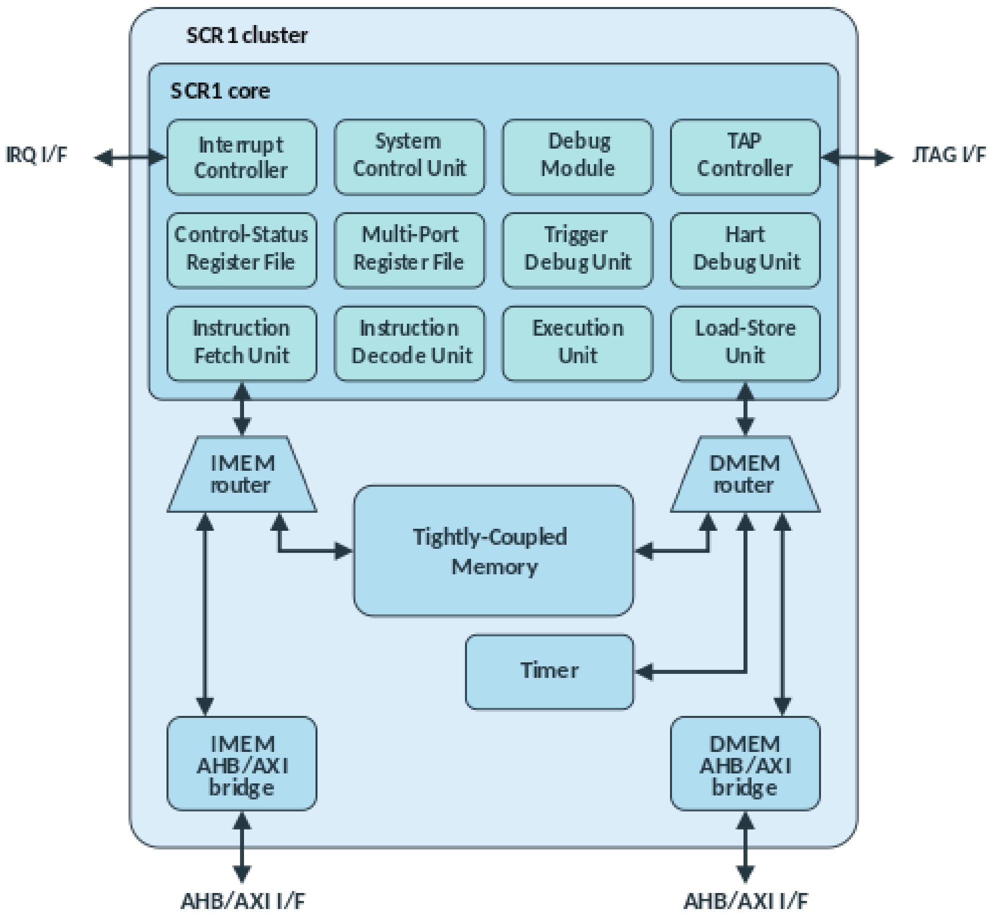


Figure 3.2 – The SCR1 RISC-V core organization [Syn22].

SCR1 is a RISC-V core that implements the RV32I ISA and the extension Zicsr to support the RISC-V privileged architecture. It also offers the integer multiplication and division (M) and the compressed (C) as optional extensions. It also offers the possibility to reduce the register bank in order to implement the Embedded RV32E ISA. SCR1 can be configured to have from 2 to 4 pipeline stages and offers many optional resources for communication with the environment. It is industry-grade core and has been silicon-proven. It is the simpler RISC-V core designed by Syntacore and the only one with open-source code.

3.1.4 The Steel RISC-V core

The Steel core is a processor core developed as a Bachelor Thesis at the UFRGS Brazilian university [Cal20]. It is a 3-stage, single-issue, in-order pipeline that implements the RV32I ISA and the Zicsr extension. It implements the Privileged architecture and supports just the machine (M) privilege level as PUC-RS5. It was validated in an FPGA board, but does not count with an interrupt controller nor details any software adaptation for the privileged architecture support. The organization of the Steel pipeline is presented in Figure 3.3. Steel Performs branches on the second stage of the pipeline.

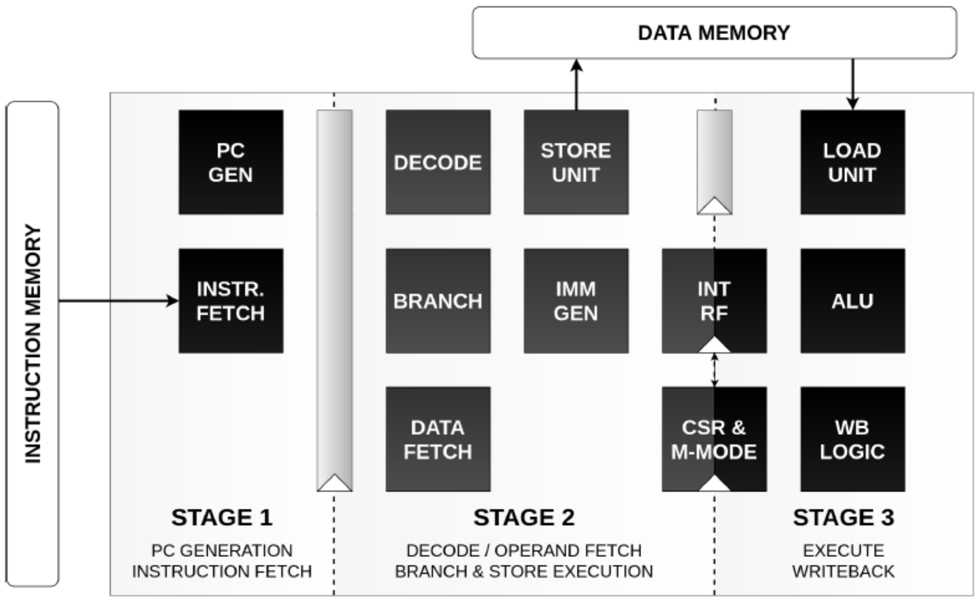


Figure 3.3 – The Steel RISC-V core organization [Cal20].

3.2 RISC-V Applied to Robotics Applications

Previous sections presented RISC-V cores comparable to PUC-RS5 and their characteristics. This Section approaches some Works using RISC-V cores for robotic applications.

3.2.1 Siwa: custom RISC-V based SOC for low power medical applications

Garcia-Ramirez et al [GRCRMR+20] introduce *Siwa*, a RISC-V RV32I-based SoC built around RISC-V, designed for highly integrated implantable biomedical applications, targeting implantable or wearable applications, that was implemented on a commercial *0.18 µm* (HV) CMOS technology. It includes control for integrated sensing and stimulation interfaces, as well as standard communication interfaces like a Universal Asynchronous Receiver/Transmitter (UART), eight general purpose input/output ports (GPIO), and a serial peripheral interface (SPI) as displayed in Figure 3.4.

Siwa is a customized implementation version of the RV32I architecture, with several CSRs modified from its definition and some not even implemented at all. Interrupts disable

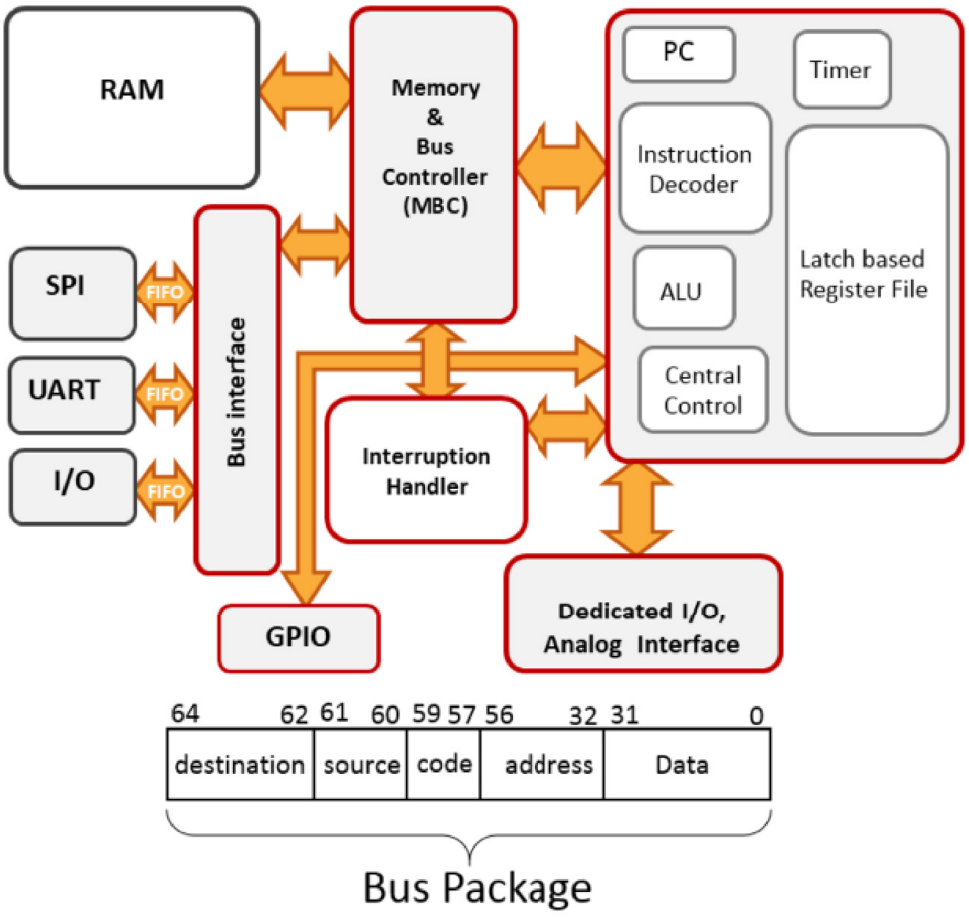


Figure 3.4 – The Siwa SoC organization [GRCRMR+20].

the handling of further interrupts until the service routine is completed and a return to normal executions is performed, except for system exceptions that are always accepted.

The centrally controlled organization is composed of five general blocks: the memory and bus controller (MBC), in charge of interfacing the CPU with the general bus and the system's memory; an instruction decoder (ID) that translates instructions into micro-coded data going to the arithmetic and logic unit (ALU) and the register file (RF), and command fields serving as the input to the CPU's controller finite state machine (FSM); a 32-word, 32-bit RF that also includes all the special CSRs of the system; 32-bit ALU capable of signed addition, word rotation, and standard the Boolean logic functions defined by the RV32I specification; and the central micro-coded FSM which coordinates the whole CPU.

3.2.2 CompROS

Dehnavi et al. [DKN+21] propose a hardware-software architecture called *CompROS* for ROS2-based robotic development in a Multi-Processor System-on-Chip (MPSoC) platform. The proposed hardware architecture consists of a Hard Real-Time (HRT) RISC-V-based subsystem implemented in programmable logic (PL) part of the MPSoC platform, a Soft Real-Time (SRT) ARM-based subsystem in the Processing System (PS) part of the MP-SoC platform and a Non-Real-Time (NRT) PC. Figure 3.5 shows the CompROS hardware architecture.

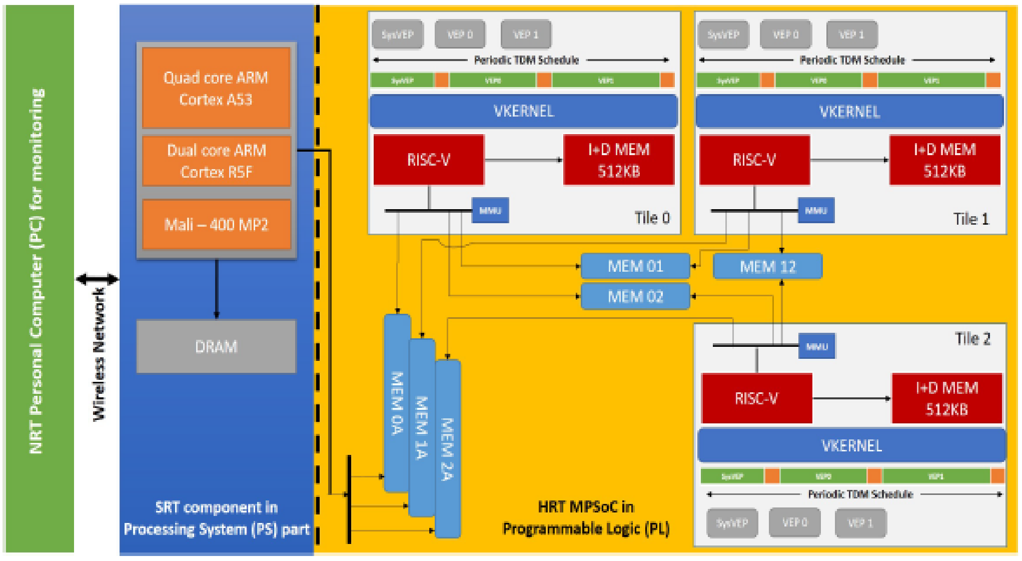


Figure 3.5 – The CompROS hardware architecture [DKN+21].

CompROS consists of three subsystems including an HRT subsystem for real-time control, an SRT ARM subsystem for supervisory control, and an NRT personal computer for system monitoring.

As an HRT subsystem, CompROS hardware implements a 3-tile CompSOC platform on a Xilinx ZCU102 board that hosts a Zynq UltraScale+ as FPGA devide. However, given the modularity of the platform, it is easily scalable to more tiles. Each CompSOC tile embeds a 32-bit RISC-V processor that runs at 100 MHz. Each tile includes 512KBytes of Instruction+Data (I+D) memory implemented through FPGA BRAMs. There is a real-time dual-ported shared memory with a capacity of 64KBytes between any two tiles of the platform that is used for inter/intra-tile communication. For RISC-V to/from ARM communication, and to dynamically load programs, there is a 32KBytes dual-port shared memory between each tile and the SRT ARM processor on the processing system side.

The second subsystem of CompROS is an SRT subsystem that consists of a quad-core ARM Cortex-A53, a dual-core Cortex-R5F real-time processor, and a Mali-400 MP2 graphics processing unit (GPU). This subsystem is used for supervisory control (such as in robot navigation), programming the HRT subsystem, and connecting the robot to the outside world.

The last CompROS subsystem is an NRT PC node that is used to monitor multiple robots, each of which hosts its HRT+SRT subsystems. The communication from the monitoring system to the supervisory control system is done through wireless network infrastructures.

3.2.3 RISC-V FPGA Platform toward ROS-based Robotics Application

Lee et al. [LCYK20] describe the design of an FPGA platform for a RISC-V processor able to run robotic applications on the Robot Operating System (ROS). This platform allows exploring the design space of RISC-V CPU for robotic applications and get insight into RISC-V CPU architectures for optimal performance and a secure system.

The platform implements on an FPGA board the RISC-V Berkeley Out-of-Order Machine (BOOM), a synthesizable and parameterizable open-source RV64GC RISC-V core written in the Chisel hardware construction language running ROS and applications. Internally, a RISC-V Core is connected with other peripherals through a system bus, so it can communicate with the outer world by using UART, I2C, SPI, and USB.

Figure 3.6 show the employed hardware and software stacks.

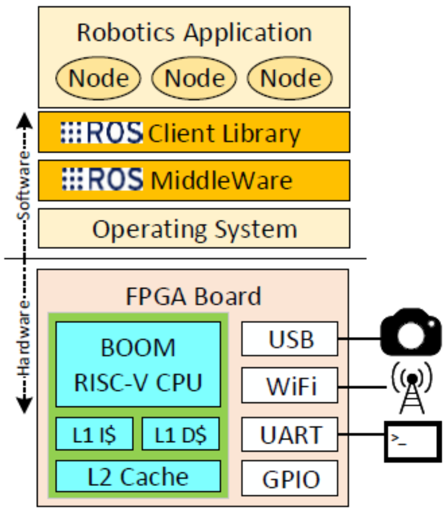


Figure 3.6 – The hardware-software stack proposed by Lee et al. [LCYK20].

The FPGA board which implements the RISC-V platform is connected to a vacuum cleaner robot via an USB interface. It can process the input image from a camera on the robot and navigate as a result.

3.3 The Memphis Platform

The Many-core Modeling Platform for Heterogeneous SoCs (Memphis) is a platform developed by the GAPH research group at PUCRS, described for example in [RCFM19]. Memphis is a many-core architecture used in academic/research environments. It is divided into two regions General Purpose Processing Cores (GPPC) and Peripherals.

GPPC is a set of identical Processing Units/cores called Processing Elements (PE) in the platform. PEs can be processors with different architectures, such as MIPS, ARM, and RISC-V. To adapt a processor core to become a Memphis PE little effort is required regarding connection to memory. The memory used by PEs is a true dual-port memory, storing code and instructions, and is possible to have shared memories connected to the system, as peripherals. The Direct Memory Network Interface (DMNI) is a component that merges two functions, a Network Interface (NI) and a Direct Memory Access (DMA). Memphis adopts a Network-on-Chip (NoC) That uses packet switching routers with the following characteristics: XY routing, round-robin arbitration, input buffering, and credit-based flow control. The DMNI directly connects the NoC router to the PE internal scratchpad memory.

Peripherals provide an input/output interface or hardware acceleration for tasks running on the GPPC region. Examples of peripherals include shared memories, accelerators for image processing, communication protocols (e.g., Ethernet, USB), and Application Injectors (AppInj). The NoC differentiates data packets from peripheral packets. Data packets are those exchanged by tasks running in PEs, and peripheral packets are those transferred between a task and a peripheral.

Memphis provides hardware applications for modeling, implemented in VHDL and SystemC. On its software stack, Memphis provides a distributed micro-Kernel capable of running the applications and is responsible for real-time task scheduling and communication among tasks.

4. WORK PROPOSAL

This Chapter presents and justifies the work proposal for this end-of-term work. Technically, it consists in producing improvements to the previously presented PUCRS-RV processor core, briefly described in Chapter 2.3 and detailed in [SC17, NSC22a]. The improvements target the provision of a processor core with support to deal with exceptions and interrupt handling. The work explores standard features of the RISC-V architecture on how to support such features. The basis for the work is the definition of the RISC-V privileged architecture [WAH21].

The scope of this end-of-term work is restricted to the development of improvements over the PUCRS-RV processor core and the implementation of the RISC-V Zicsr extension, alongside with the privileged architecture and one or more of the privileged modes. All changes made in the original architecture will be tracked into a branch with no privileged modes and Zicsr extension to keep track of the impact that these features bring to the processor core when it comes to area usage and performance.

This work also seeks to be a guide on how to extend the developed processor core, documenting the needed changes and their impacts on the original PUCRS-RV core pipeline structure. This will bring support for future users and developers to easily use and extend the PUC-RS5 core, and this is one of the goals of this project. For that, a detailed overview of the improvements and of the validation process will be brought in Chapters 5 and 6.

To ensure the feasibility of using the proposed organization in real system implementations, this work also describes the design and implementation of an FPGA prototype of the PUC-RS5 processor core on top of a commercial platform, and demonstrates its operation. The PUC-RS5 core FPGA prototyping process is detailed in Chapter 6.

The rest of this Chapter includes four sections. It explores the previous work that helped the Author in defining the specification of the PUC-RS5 core in Section 4.1. Next, Section 4.2 details the changes necessary on the PUCRS-RV to transform it into the PUC-RS5 core. A central feature of the PUC-RS5 core is that it follows the RISC-V Privileged Architecture standard, by adding the Zicsr extension to the RV32I ISA, and this is the subject of Section 4.3. Lastly, Section 4.4 explores how this work deals with the standard RISC-V interrupt organization modes depicted in Section 2.2.1.

4.1 PUCRS-RV Organization: History and Structure

The PUCRS-RV core development started in the year 2017 with ARV, a high-level organization specification implemented in the Google Go language [SC17, SC21]. The ARV specification and its unfolding into a RISC-V organization is described in Section 4.1.1.

PUCRS-RV in itself is a 5-stage pipeline organization intended to support the design of a fully asynchronous organization, code-named ARV (an acronym for *Asynchronous RISC-V*).

Figure 4.1 gives an overview of the PUCRS-RV organization, briefly discussed next and approached in more detail in Section 4.1.2.

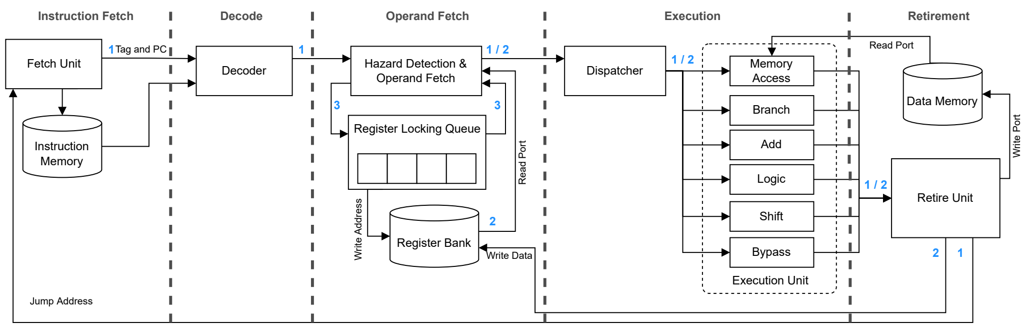


Figure 4.1 – Block diagram of the PUCRS-RV organization. Blue numbers over arrows indicate the three operational loops of the organization: (1) Control Loop; (2) Datapath Loop; (3) Register Locking Loop.

One of the main features of PUCRS-RV is its tag system, which tracks instructions fetched along the pipeline until their retirement. Retirement here refers to the process of instruction execution conclusion. An instruction is only retired if the tag associated with it is the same tag present in the Retire Unit. The Retire Unit has an internal tag that is increased every time a branch instruction is retired. The Fetch Unit creates the tag, which is increased when a branch is accepted. Instructions fetched from the new address jumped to are associated with the new tag, causing a match in the Retire Unit tag. Instructions fetched after fetching the instruction that causes the branch may be fetched before knowing these are not to be executed, and the tags of these instructions will not match any Retire unit tag, when these arrive at the Retire Unit, justifying to discard them. This system is described as the organization *"Control Loop"*, and involves the system of tags and the branching control tied to the tags. The loop can be observed in Figure 4.1, and is identified by number 1 on the arrows composing it.

Another feature present in the PUCRS-RV organization is a queue of locked registers, called Register Locking Queue (RLQ). The RLQ keeps track of every register of the Register Bank with a pending write to it. A pending write register is a register that has an instruction on the pipeline that will write its result to it. When an instruction tries to read from a register that has a pending write this implies a data hazard. When a data hazard is detected, a bubble is issued in the pipeline and the instruction that caused the hazard is held until the hazard is resolved. The hazard is considered resolved after the instruction that targets the locked register writes its result to the locked register, which then contains a valid value to be read and is thus unlocked. This register-locking mechanism also creates a loop, due to its cyclical usage in data hazard detection and its relationship with the Register Bank, since this pipeline stage is the last one in the organization, and which writes to the Register Bank. The "Register Locking Loop" is presented in Figure 4.1 and is identified by number 3.

The third and last loop in the PUCRS-RV pipeline organization is the *"Datapath Loop"*. It comprises the path that fetches instruction operands from the Register Bank and ends with the write-back operation performed by the Retire Unit. This loop is identified in Figure 4.1 by number 2.

4.1.1 The ARV Specification and ASIC Implementation

The ARV specification was proposed by Sartori and first described in [Sar17], consisting in a Go language abstract description. Go is a modern concurrent language designed by Google, which incorporates modeling principles derived from the Communicating Sequential Processes (CSP) paradigm.

The main goal of the ARV specification was to provide the functionality of a complete asynchronous design of a RISC-V processor core. This goal was achieved by using a series of go-routines communicating through channels. ARV was validated using the RISC-V Berkeley suite that performs unit tests for each instruction, achieving compliance with the ISA specification. The complete ARV specification is freely available at github [SC21].

Later, the Author entered the research project and was responsible for developing the Pulsar ARV (PARV) processor core [NSC22b], an RTL-like System Verilog description. PARV is amenable to automatic synthesis by using the Pulsar asynchronous synthesis tool [SWMC19, SMC20c, SMC20a], targeting an Application Specific Integrated Circuit (ASIC) implementation using the ASCEnD-FreePDK45 standard cell library [SMC20b]. To learn the System Verilog language the Author started by implementing a synchronous RTL version of the ARV specification, which then becomes the PUCRS-RV processor core.

The Go language serves to describe the ARV organization and also to validate its asynchronous operation. To use this processor core, nonetheless it is necessary to translate it in some description that is amenable to be used as source for automated hardware development. This triggered a work of implementing the architecture in a Hardware Description Language (HDL). The chosen HDL was System Verilog, due to its powerful validation features and also for its compatibility with the [SWMC19] flow that was under development to support synthesis of the asynchronous organization. The Pulsar flow transforms an RTL-like description into a *Quasi Delay Insensitive (QDI)* asynchronous circuit. The PULSAR tool under development at the time required more complex circuit descriptions for demonstrating its correct operation. Its development was implicitly aligned with new challenges that would arise from more complex circuit constructions. For that purpose, the System Verilog version of the ARV organization (PARV) was defined and implemented.

The initial strategy was to develop the simpler units first such as the Execute Unit, next tackling units of increasing complexity. As the simpler units were built and validated using the PULSAR flow, the integration started to be performed between units, to the point where the asynchronous core was entirely built and validated by post-layout simulation.

As the main goal was to validate the Pulsar flow and the development of an entirely asynchronous synthesizable core and the RTL-like source code was very close to an RTL design of a synchronous processor core, the PUCRS-RV core was created in parallel with the asynchronous organization. This processor core was first published in [LNZ+22] where it was used as a target for a RISC-V soft error reliability analysis.

4.1.2 The PUCRS-RV System Verilog Design and Implementation

The PUCRS-RV organization is briefly introduced in Section 4.1, while its first specification in Go and later asynchronous RTL-like implementation are treated in Section 4.1.1. This is the basis for the work realized herein by the Author.

The PUCRS-RV core is a complete synchronous implementation of the RV32I ISA, and has a pipeline organization similar to the Go-language specification. It was first developed with the goal to learn more about the RISC-V architecture and its hardware development, and to increase circuit design knowledge. The target was to apply the acquired knowledge to further the design of an asynchronous version of RISC-V. PUCRS-RV was quickly recognized as a valuable asset, providing the research group of the Author with a processor core capable of being adapted to multiple uses, such as to build embedded robotic systems and/or constitute a processor core to use many-core integrated circuit design.

4.2 Changes from PUCRS-RV to PUC-RS5

The PUCRS-RV processor core consists of a simple pipeline. This pipeline has no capability to support interrupts of the instruction execution flow, gives no support for traps or trap handling, and was validated only through simulation. Its validation relies on the use of HDL specific and or generic testbenches built to validate arbitrary software execution. This approach can simplify memory access behavior, by assuming asynchronous and instantaneous access to memories for both, instructions and data. This led to difficulties when trying to prototype the core design using real-world memories, which often operate synchronously and which access latency can be long, even longer than the core processor clock period. To provide a processor core effectively usable in real-world systems there arises the need for a series of changes and additions to the original organization. The PUCRS-RV core organization, depicted in Figure 4.1 is essentially a single issue five-stage pipeline.

This work proposes changes to the PUCRS-RV organization to extend its applicability, by adding support to handle interrupts and exceptions. The main organizational change over the PUCRS-RV core organization is combining the Instruction Fetch and Decode stages in a single one. This is intended to decrease the pipeline depth, lowering area usage (by reducing the number of flip-flops used as time barriers to separate stages). Besides that, the PUC-RS5 core is expected to support the Zicsr extension, the only requirement to provide a standard implementation of the RISC-V privileged architecture. The Zicsr extension basically provides atomic read-modify-write instructions that operate on the control and status registers (CSRs) [WAH21].

The change in the pipeline depth is motivated by the implementation of the privileged architecture, which foresees the addition of a set of CSRs. These registers are the core of any privileged architecture, once they control and keep track of the state and status of the processor core and also add some features, such as trap handling. Trap handling is an essential part of a processor core, because it adds the aptitude for interruption handling. CSRs can be classified as mandatory or optional. Once there are several mandatory CSRs, these will be grouped in a CSR Bank that has the capability of reading and writing from/to CSRs through instructions of the Zicsr extension.

Since privilege modes are not mandatory for every RISC-V implementation, this work will restrict attention to a pipeline without any support to privilege modes other than the machine mode, with complete support for the Zicsr extension, and with a minimal CSR bank. This will allow comparisons between PUC-RS5, PUCRS-RV and other cores in the literature, allowing to achieve an estimation of the minimum overhead required for counting with a RISC-V privileged architecture.

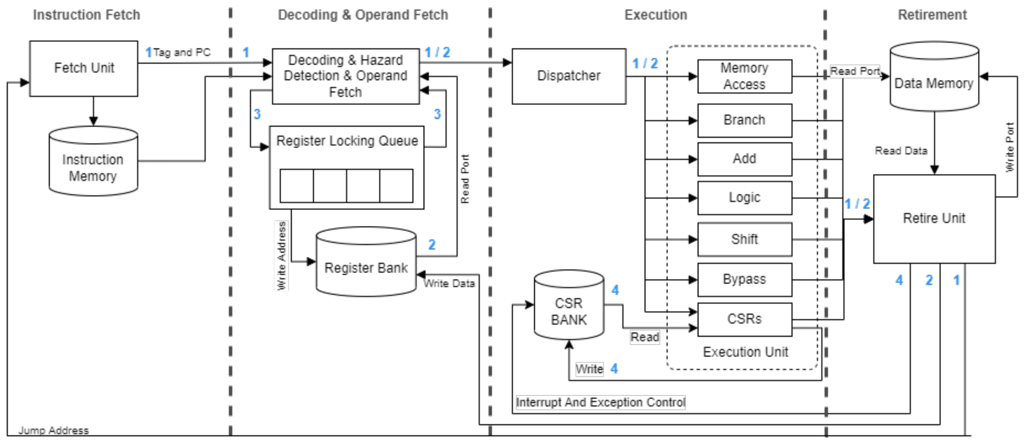


Figure 4.2 – Proposed block diagram for the PUC-RS5 organization. Note the new CSR register bank and the new connections to enable the privileged architecture.

Figure 4.2 depicts the proposed organization. The pipeline is reduced by one stage, becoming a pipeline with the following stages:

1. Fetch;

2. Decode;
3. Execute
4. Retire.

The Fetch unit addresses the instruction memory that returns the requested instruction directly to the Decode Unit on the next clock cycle. The Register Bank is accessed in the second stage of the pipeline, and its operands will be returned in the same cycle, asynchronously. The Execution Unit is still responsible for memory readings, but the data requested by the execution stage will be provided directly to the Retire Unit on the next clock cycle. Writes to memory are performed by the Retire Unit in the fourth stage. The new CSR Bank is placed between the third and fourth stages. This allows access to CSR Bank with the same behavior as Data Memory accesses, which brings the possibility to use some CSR mapped in memory to reduce the use of registers inside the processor core. The Retire Unit represents the fourth stage, and is responsible for performing the trap handling protocol, to be discussed in Chapter 5. Due to this requirement, it is in constant communication with the CSR Bank.

The communication between the Retire Unit and the CSR Bank is represented by the wire with label 4 in Figure 4.2. The CSR Execute Unit communicates directly with the CSR Bank to perform reads and writes from/to it. The Retire unit communicates with the CSR Bank to perform the control of interrupts and exception traps.

4.3 RISC-V Privileged Architecture and the Zicsr Extension

The RISC-V Privileged Architecture is established in Volume II of the RISC-V Instruction Set Manual [WAH21]. It defines the privileged architecture itself, which uses CSRs and privilege levels that are encoded as operation modes inside CSRs. In the privileged architecture, a RISC-V core is also referred to as a hardware thread (or hart). The privilege levels provide protection to the core from different components of the software, intercepting and eventually preventing attempts to perform operations that are not permitted on the current privilege mode. When such attempts happen, it causes an exception, that causes a trap to a higher privilege level to treat the exception. Table 4.1 shows the RISC-V supported combinations of privilege modes.

The machine level is the highest privilege level and is the only level that is mandatory for a RISC-V privileged implementation. The machine level is encoded as machine mode (M-Mode) and has access to all the functionalities of the core. All the code that is run in M-Mode is inherently trusted, as it has low-level access to the machine implementation.

The supervisor level is encoded as supervisor mode (S-Mode) and the user level is encoded as user mode (U-Mode). These levels are not mandatory and a RISC-V im-

Table 4.1 – Supported combinations of privilege modes.

Level Number	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

plementation might provide from 1 to 3 privilege modes. Simpler embedded systems usually implement only the machine mode because of the prioritization of reduced area usage though this will provide no protection against incorrect or malicious application code. Some RISC-V implementations will also support the user mode to provide separation of the software stacks and protect the system from application code. More complex systems that run Unix-like operating systems usually implement all three levels to provide isolation between a supervisor level operating system and the supervisor execution environment. A hart usually runs application code in user mode until a trap occurs, forcing a switch to a higher privilege level through a trap handler that will eventually resume execution in user mode.

Each privilege mode has its own set of CSRs with a predefined range of addresses. The two higher bits of the CSR address (csr[11:10]) define whether the register is read/write or read-only. The following two bits (csr[9:8]) encode the lowest privilege level that has access to the CSR. When an instruction tries to access a inexistent or non-implemented CSR, an exception will be generated. An exception is also raised when there is an attempt to access a CSR without the minimum privilege level or when attempting to write into a read-only register. The privileged architecture also provides the possibility to implement custom CSRs in non-reserved addresses. Some CSRs have no mandatory implementation based on the privilege levels that the hart implements. For example, in a hart that has only the machine mode it makes no sense to implement the Machine Trap Delegation Registers (medeleg and mideleg).

Access to the CSRs is made by the instructions defined in the Zicsr extension. This extension is mandatory in a privileged architecture. It defines a set of six Instructions that operate on CSRs. All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in a 12-bit immediate field of the instruction. The immediate form of instructions employs a 5-bit zero-extended immediate encoded in the rs1 field. This extension will be further discussed in Section 5.2, where all instructions in this extension will be detailed and their implementation will be presented.

The privileged architecture and the CSRs are primordial for the aptitude of trap handling. It is the main goal of this work to implement a RISC-V core capable of handling exceptions and especially interrupts.

4.4 The PUC-RS5 Interrupt Organization Mode

The PUC-RS5 processor core implements just the simplest interrupt organization mode, CLINT (see Section 2.2.1 for details on the RISC-V available interrupt modes). The Core Local Interrupter (CLINT) mode for interrupt control has a fixed priority and two modes of operation, direct and vectored. PUC-RS5 is expected to implement only the direct mode.

5. THE PUC-RS5 CORE IMPLEMENTATION

This Chapter presents the design of the PUC-RS5 processor core in detail All changes made to the start core (PUCRS-RV) are discussed and justified.

The PUC-RS5 core interface to the external world is displayed in Figure 5.1. Signals represented to the left of the Figure are related to instruction memory access and are further described in Section 5.8. Signals to the right are used for data memory access and are also discussed in Section 5.8.

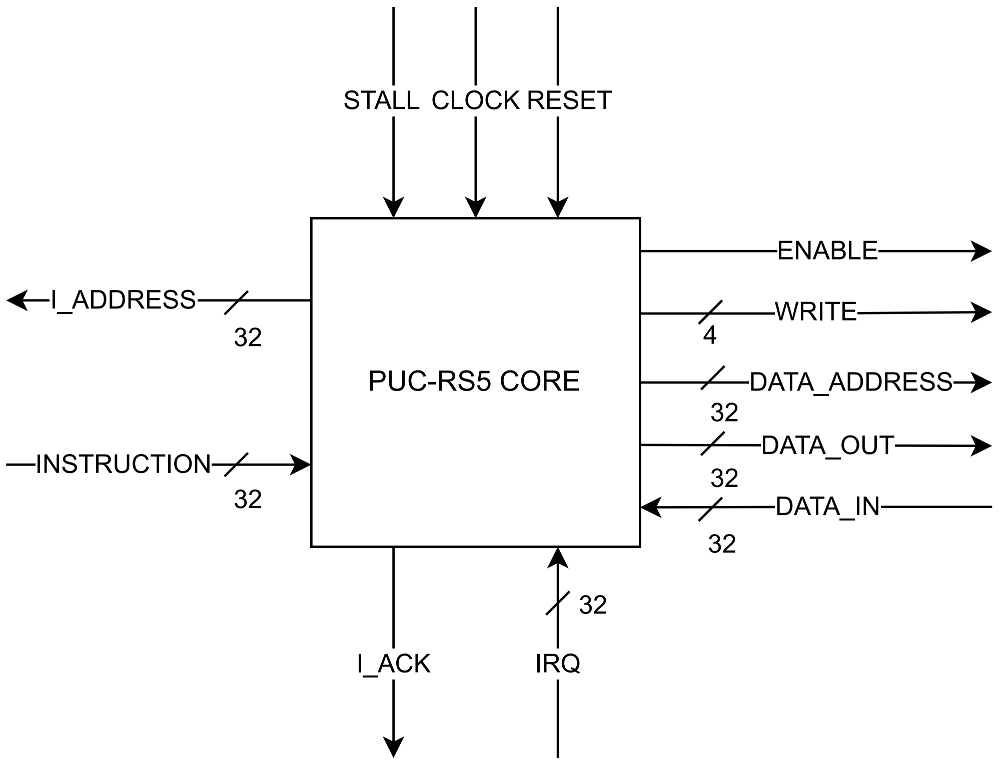


Figure 5.1 – The PUC-RS5 processor core interface, precisely depicting all core pins.

The three signals represented on the top of the Figure are global control signals and are detailed in Table 5.1. The two signals located at the bottom of the figure are related to the interrupt control and are also somehow detailed in Table 5.1.

Table 5.1 displays a brief description for each control signal. The *CLOCK* signal represents the synchronization signal that operates at a constant frequency. The *RESET* signal sets the state of the core to an initial known state and is synchronous to the CLOCK signal. The *STALL* signal is further discussed in Section 5.9. The *I_ACK* and *IRQ* signals are related to the interrupt controller and are further explored in Section 5.4.5.

The following nine Sections detail the PUC-RS5 design. Section 5.1 presents an overview of the PUCRS-RV pipeline changes, focusing on the reduction of one stage. Section 5.2 brings a detailed view of the Zicsr extension and its instructions. Section 5.3 details

Table 5.1 – Description of control signals.

Signal	Width	Direction	Description
CLOCK	1 Bit	input	Clock Synchronicity signal
RESET	1 Bit	input	Reset the state of the core
STALL	1 Bit	input	Stall the pipeline State
IRQ	32 Bits	input	Interrupt Request Indicator
I_ACK	1 Bit	output	Interrupt Acknowledgment

the implementation of the PUC-RS5 CSR unit. Section 5.4 focus on the implementation of the CSRs that are grouped as a CSR bank. Section 5.5 presents the integration and communication of the CSR bank with the rest of the pipeline. Section 5.6 presents the software stack developed to support privilege modes. Section 5.7 presents the changes made to the structure of the data register bank. Section 5.8 describes the changes made to the external data memory interface. Finally, Section 5.9 discusses the new *STALL* signal, responsible for stalling the PUC-RS5 core pipeline.

5.1 Changes to the PUCRS-RV Pipeline

The organization implemented by the PUCRS-RV pipeline has 5 stages. One of the bigger changes implemented in the PUC-RS5 processor core is reducing one pipeline stage. This was made by merging the second stage, the Decode unit, and the third stage, the Operand Fetch Unit into a new and more complex Decoder unit that comprises the two stages. The older Decode unit was responsible for just identifying the instruction and its format, also detaching from the instruction code the address of the operands and forwarding it to the next stage. The Operand Fetch stage received the pre-processed signals from the Decode unit and processed them. It forwarded the address of the operands to the register bank, detected data hazards, and also performed the extraction of immediate data, based on the instruction format. With this explained it is clear the possibility of identifying the unnecessary use of additional registers and therefore an increase of area usage. One of the cases where there is unncessary use of a register is in the detachment of the operand addresses, which have a fixed position on the instruction code and was just assigned to a register to be passed to the register bank. For reducing area, it was needed to pass forward, by registers, the instruction, and also the instruction format.

In the data hazard detection, the cost of a hazard was the same, once hazards are detected only in the stages in front of the unit that performs the register bank access. In both cases there were two stages, execute and retire. In the PUC-RS5 core, hazard detection is made considering the execute stage and the retire stage. The retire stage is responsible for the conclusion of the instructions, performing jumps, and memory writes, and also performing write-back in the register bank. The retire unit is considered in hazard detection because it currently writes in the same clock edge that the read occurs, the rising edge of the clock. If the register bank is a master-slave or has a write-then-read architecture or works in two clock edges, then the data hazard detection can be simplified to consider locked only the destination register of the instruction present in the execute stage.

The branch prediction for both PUCRS-RV and PUC-RS5 pipelines is never jump. It keeps the pipeline simpler and takes less area. When the prediction is wrong, the pipeline is filled with instructions that must be discarded until the instructions of the new instruction flow have arrived and then fill the pipeline again. The cost of discarding the wrong-fetched instructions is higher in a 5-stage pipeline than in a 4-stage pipeline, since the maximum number of discards is lower in the later pipeline.

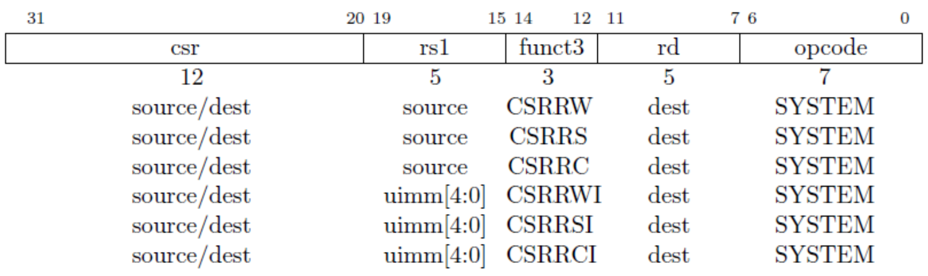
The PUC-RS5 Decoder unit becomes more complex with more responsibilities but with fewer registers being used as time barriers.

Some other changes relate to memory communication delays. The PUCRS-RV considered that memory data fetch was available in the same cycle. This was a less common behavior for real-world memories and is hard to reproduce in practice. The more usual behavior is to have a 1-cycle delay from the address signal being sent and the requested data is available. This way, the new pipeline addresses the instruction memory at the Fetch unit and the Decoder unit receives the instruction directly from memory. The older specification addressed the instruction memory in the Fetch unit and considered that it received the instruction immediately, in an asynchronous way, bringing the need for an Instruction Register (IR) as a time barrier for the instruction being provided to the second stage. The same happened in memory reads that are performed on the Execute unit. These expected the data to appear in the same cycle and then registered it to pass it to the Retire unit. In PUC-RS5 the Execute stage only outputs the read address and the data is received by the Retire unit that processes it and then writes to the register bank. The Retire unit, besides receiving data read from memory also provides the data for writing to memory.

One of the intentions of this work was also to keep track of the area increase caused by the use of the privileged architecture and the CSRs.

5.2 The Zicsr Extension Implementation

The RISC-V Instruction Set Manual [WA19] specifies several extensions for RISC-V. Among them, there is the Zicsr extension. This extension defines 6 new instructions that are used to access CSRs. These CSRs are defined in the Volume 2 of the RISC-V Manual [WAH21] and will be further explained in Section 5.4. The CSR instructions can atomically read from and write to a single CSR. The format of the six instructions is displayed in Figure 5.2. The CSR address is encoded on a 12-bit immediate field held in the field *csr* located in bits 31-20. The 12-bit field reaches an address space of 4096 CSRs on each hart. The instruction encoding provides two register addresses: a source register, *rs1*, and a destiny register, *rd*. In the immediate forms of the instructions, the field *rs1* holds a 5-bit immediate that is zero-extended. The fields *funct3* and *opcode* are used for instruction decoding.



31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 5.2 – Definition of Zicsr instructions format.

Based on the source and target registers, the functionality of the instruction can be reduced by not performing one of its actions, such as reading from or writing to the CSR. These changes in the behavior of the instruction are illustrated in Table 5.2. This behavior is needed to be able to just read or just write a register, which is useful in cases such as reading a read-only CSR without writing to it and causing an exception. The CSR reads and writes can have some side-effects such as exception raising (e.g. when the user tries to access a CSR with higher privilege required or performing a write to a read-only register) and other CSRs being modified implicitly (e.g. *instret* CSR).

Table 5.2 – Conditions determining whether a CSR instruction reads or writes to a CSR.

Register operand				
Instruction	*rd* is `x0`	*rs1* is `x0`	Reads CSR	Writes CSR
CSRRW	Yes	–	No	Yes
CSRRW	No	–	Yes	Yes
CSRRS/CSRRC	–	Yes	Yes	No
CSRRS/CSRRC	–	No	Yes	Yes
Immediate operand				
Instruction	*rd* is `x0`	*uimm*=0	Reads CSR	Writes CSR
CSRRWI	Yes	–	No	Yes
CSRRWI	No	–	Yes	Yes
CSRRSI/CSRRCI	–	Yes	Yes	No
CSRRSI/CSRRCI	–	No	Yes	Yes

5.2.1 CSRRW Instruction

The CSRRW instruction is an atomic Read and Write instruction that performs a swap of the CSR and the source register. It reads the value of the CSR and writes it in the destiny register and writes the source register content to the CSR. When the destiny register is the *x0* (the zero register) the instruction does not perform the read, only the write. In this case, the instruction can also be known as the CSRW assembly pseudo-instruction.

5.2.2 CSRRWI Instruction

The CSRRWI instruction is a variant of the CSRRW that uses the *rs1* field as an unsigned 5-bit immediate. This instruction has the same behavior as CSRRW when the *rd* field is *x0*. In this case, the instruction can also be known as the CSRWI assembly pseudo-instruction.

5.2.3 CSRRS Instruction

The CSRRS instruction is an atomic Read and Set instruction that performs a CSR read and sets the intended bits of the CSR. The initial value in the source register, *rs1*, is used as a bit mask that specifies the CSR bit positions that must be set. Any bit high in *rs1* causes the corresponding bit to be set in the CSR only if that CSR bit is writable. When the *rs1* field holds the value of the register *x0* the instruction does not write to the CSR. In this case, the instruction can also be known as the *CSRR* assembly pseudo-instruction.

5.2.4 CSRRSI Instruction

The CSRRSI instruction is a variant of the CSRRS that uses the *rs1* field as an unsigned 5-bit immediate. This instruction has the same behavior as CSRRS when the *rd* field is *x0*.

5.2.5 CSRRC Instruction

The CSRRC instruction is an atomic Read and Clear instruction that performs a CSR read and clears the intended bits of the CSR. The initial value in the source register, *rs1*, is used as a bit mask that specifies the CSR bit positions that must be cleared. Any bit high in *rs1* causes the corresponding bit to be cleared in the CSR only if that CSR bit is writable. When the *rs1* field holds the value of the register *x0* the instruction does not write to the CSR.

5.2.6 CSRRCI Instruction

The CSRRCI instruction is a variant of the CSRRC that uses the *rs1* field as an unsigned 5-bit immediate. This instruction has the same behavior as CSRRC when the *rd* field is *x0*.

5.3 The CSR Unit

To implement the Zicsr extension the pipeline required changes, which are:

1. The inclusion of Zicsr instructions in the instruction type enumeration;
2. The inclusion of a new unit XU type enumeration;
3. The creation of the CSR operation type;
4. Zicsr instruction decoding;
5. The CSR Unit creation and integration in the execute stage;
6. The CSR Bank integration.

Topics 1, 2 and 3 are described in Section 5.3.1, topic 4 is described in the Section 5.3.2, topic 5 is described in Section 5.3.3 and topic 6 is described in Section 5.5

5.3.1 CSR Enumerations

The first step to implementing the Zicsr instruction is to define them in the enumerations that define the type of instructions. These definitions are made in a package module that defines every type and constant that the organization uses, this package is defined in the file *"pkg.sv"* that is presented in Listing 5.1.

```
1  package my_pkg;
2
3      const int            MEMORY_SIZE = 65535;
4
5      typedef enum  logic[2:0] {R_type, I_type, S_type, B_type, U_type, J_type}
    fmts;
6
7      typedef enum  logic[2:0] {
8                                  OP0, OP1, OP2, OP3, OP4, OP5, OP6, OP7
9                                } op_type;
10
11     typedef enum  logic[5:0] {
12                            NOP = 6'o00,LUI,SRET,MRET,WFI,ECALL,EBREAK,INVALID,
13                            ADD = 6'o10,SUB,SLTU,SLT,
14                            XOR = 6'o20,OR,AND,
15                            SLL = 6'o30,SRL,SRA,
16                            BEQ = 6'o40,BNE,BLT,BLTU,BGE,BGEU,JAL,JALR,
17                            LB  = 6'o50,LBU,LH,LHU,LW,SB,SH,SW,
18                            CSRRW=6'o60,CSRRS,CSRRC,CSRRWI,CSRRSI,CSRRCI
19                            } i_type;
20
21     typedef enum  logic[2:0] {bypass, adder, logical, shifter, branch, memory,
    csri} xu;
22
23     typedef enum  logic[1:0] {NONE, WRITE, SET, CLEAR} csr_ops;
24
25     typedef enum  logic[1:0] {USER, SUPERVISOR, HYPERVISOR, MACHINE = 3}
    Privilege;
26
27 endpackage
```

Listing 5.1 – Enumeration definitions for the Zicsr instructions.

The instruction encoding of the PUC-RS5 core is made in a way that it can be split afterward into simpler types that defines the Execute Unit (XU) for the instruction and the corresponding operation in that Execute Unit.

The execute Unit type is defined by the enumeration called *"xu"* that is defined in line 21 of Listing 5.1, it is a 3-bit type and corresponds to the 3 highest bits of the instruction type.

The operation type is defined by the enumeration called *"op_type"* that is defined in line 7. It corresponds to the 3 lower bits of the instruction type. This operation is only used inside the operation units and is a way of sparing some bits in the management of the operation to perform.

To add a new execute unit the only change needed is to add a new field in the enumeration *xu* called *"csri"* which stands for "CSR Instructions".

The instructions are defined as a 6-bit type called *"i_type"*. This type has the execute unit as the 3 higher bits and the operation as the 3 lower bits. Because of that, the octal format that comprises 3 bits is used to separate instructions by unit. This definition is located in line 11. The new instructions definitions are located in line 19 and they were made using the instruction name as the enumeration key.

A new type was defined for the CSR operations, this type is called *"csr_ops"* and is defined in line 24. It is a 2-bit type that defines the three operations that can be performed in a CSR: Write, Set and Clear. The remaining bit is used as a No-Operation code.

The definition of privileges is also shown in line 24. Although the machine privilege mode is the only one currently supported, others are already defined for future utilization.

5.3.2 CSR Instructions Decoding

Instruction decoding is made at the second stage of the pipeline. This stage is defined in the file *"decode.sv"* The decode is made based on the instruction object code received from the memory. The decode of the Zicsr extension is made based on the fields *funct3* and *opcode* that correspond respectively to bits 14-12 and 6-0 of the instruction word. The definition of these codes is made in the RISC-V specification. To perform the decoding of the instruction object code into the previously defined instruction enumeration types, presented in Section 5.3.1, other conditions are added into the main decoding *"if"* operation, these conditions are displayed in Listing 5.2.

```
if(instruction[6:0]==7'b0110111) i<=LUI;
                        .
                        .
                        .
else if(instruction[14:12]==3'b001 & instruction[6:0]==7'b1110011)i<=CSRRW;
else if(instruction[14:12]==3'b010 & instruction[6:0]==7'b1110011)i<=CSRRS;
else if(instruction[14:12]==3'b011 & instruction[6:0]==7'b1110011)i<=CSRRC;
else if(instruction[14:12]==3'b101 & instruction[6:0]==7'b1110011)i<=CSRRWI;
else if(instruction[14:12]==3'b110 & instruction[6:0]==7'b1110011)i<=CSRRSI;
else if(instruction[14:12]==3'b111 & instruction[6:0]==7'b1110011)i<=CSRRCI;
                        .
                        .
                        .
else i<=INVALID;
```

Listing 5.2 – The Zicsr instruction decoding process.

The instruction format is already decoded as a *"I_type"* based on the *opcode* field. The format is responsible for the generation of the immediate and also for assigning the correct operands to the execute stage. The execute unit is implicitly defined in the instruction type as explained in Section 5.3.1.

5.3.3 CSR Execute Unit

The first step to add a new Execute unit to the PUC-RS5 core is to instantiate it in the *"execute.sv"* file. This file defines the third pipeline stage and is responsible for the instantiation of the Execute units and the demultiplexing of the results. To do that, the unit must be instantiated with the input signals being associated with the operands received from the Decoder stage. The outputs are then assigned to an array of results where each position is an *Execute unit* output. This array is demultiplexed based on the instruction being executed. In the case of the CSR Unit, the output to the *Retire unit* is the data read from CSR bank. The other CSR unit outputs are directly linked to the CSR bank.

The CSR Execute unit is defined inside the file *xus.sv*, together with the other execute units, its interface is shown in Listing 5.3. The CSR Unit receives the two operands and the instruction operation. The first operand is the *opA* input and is the data read from the Register bank. The second operand *instruction*, is the instruction code, the immediate and the CSR address is extracted from this signal. The third operand, *i*, is the operation the instruction performs.

The Unit has 6 outputs. The first two, *csr_rd_en* and *csr_wr_en*, are the read and write enables for the CSR Bank. The *csr_op* is the output that defines the operation to be executed in the CSR Bank. The *csr_addr* is the address of the CSR. The *csr_data* is the data to write or used in the set and clear operations. The *csr_exception* is a signal that indicates an exception in the CSR operation.

```
module csrUnit (
    input logic [31:0]  opA,
    input logic [31:0]  instruction,
    input op_type       i,
    output logic        csr_rd_en,
    output logic        csr_wr_en,
    output csr_ops      csr_op,
    output logic [11:0] csr_addr,
    output logic [31:0] csr_data,
    output logic        csr_exception
);
```

Listing 5.3 – The CSR unit interface.

Listing 5.4 displays the code used for CSR access exception detection. A CSR exception is raised in cases where either the instruction tries to write to a read-only CSR or if the instruction tries to access a CSR with a Higher privilege than the actual privilege. As the current implementation has only the machine mode implemented, the privilege is fixed in a hard-coded way.

```
always_comb
    // Raise exception if CSR is read-only and write enable is true
    if ((csr_addr[11:10] == 2'b11) && (csr_wr_en_int == 1))
        csr_exception <= 1;
    // Check Level privileges
    else if ((csr_addr[9:8] < privilege) && ((csr_rd_en_int == 1) || (csr_wr_en_int == 1)))
        csr_exception <= 1;
    // No exception is raised
    else
        csr_exception <= 0;
```

Listing 5.4 – CSR instructions exception detection.

Listing 5.5 displays the process of the CSR Bank read enable and write enable alongside its operands. From the instruction input, the internal operands *rd* (destiny register), *rs1* (register source 1), and *csr_addr_int* are extracted (lines 1-3). These internal signals are then used in the conditional operator *"if"* in lines 8-26. This Conditional operator is responsible for generating the internal read and write enables based on the behavior of each instruction facing the *rs1* and *rd* addresses as listed in Figure 5.2. These internal enables are then assigned to the output after a bitwise AND operation with the inverse of the *csr_exception* signal. When an exception is detected no operation is allowed, since the bitwise AND with the enables will result in a not enable state.

```
1  assign rd  = instruction[11:7];
2  assign rs1 = instruction[19:15];
3  assign csr_addr_int = instruction[31:20];
4
5  assign csr_rd_en = csr_rd_en_int & !csr_exception;
6  assign csr_wr_en = csr_wr_en_int & !csr_exception;
7
8  always_comb begin
9      if (i==OP0 || i==OP3) begin // CSSRW or CSSRWI
10         csr_wr_en_int = 1;
11         if (rd==0)
12             csr_rd_en_int = 0;
13         else
14             csr_rd_en_int = 1;
15
16     end else if (i==OP1 || i==OP2 || i==OP4 || i==OP5) begin     // CSRRS/C and CSRRS/CI
```

```
        csr_rd_en_int = 1;
        if (rs1==0)
            csr_wr_en_int = 0;
        else
            csr_wr_en_int = 1;

    end else begin
        csr_rd_en_int <= 0;
        csr_wr_en_int <= 0;
    end
end
```

Listing 5.5 – Generation of operands and read and write enables for CSR Bank.

Listing 5.6 displays the code that is used to identify if the data to be written in the CSRs is the data from the register bank or the immediate from the instruction code. It also shows the operation decoding based on the previously decoded *OP* input signal. Lines 1-2 define a block that assigns the internal signal to the output *csr_addr*. Lines 4-8 describe the assignment to the output *csr_data* either the operand from the register bank or the zero-extended immediate from the instruction that was located in the *rs1* field. Lines 10-19 define the operation output *csr_op* the operation based on the instruction operation received.

```
 1 always_comb
 2     csr_addr <= csr_addr_int;
 3
 4 always_comb
 5     if (i==OP0 || i==OP1 || i==OP2)
 6         csr_data <= opA;
 7     else
 8         csr_data <= ’0 & rs1;
 9
10 always_comb
11     if (i==OP0 || i==OP3)               // WRITE
12         csr_op <= WRITE;
13     else if (i==OP1 || i==OP4)          // SET
14         csr_op <= SET;
15     else if (i==OP2 || i==OP5)          // CLEAR
16         csr_op <= CLEAR;
17     else                                // NONE
18         csr_op <= NONE;
19     end
```

Listing 5.6 – Data and operation definition for CSR Bank.

5.4 The CSR Bank

Once the PUC-RS5 core correctly implements the Zicsr extension, the CSR bank can be developed. For that, the first step is to define CSR names and addresses. This was done in the package definition in the way previously explained in Section 5.3.1 and appears in Listing 5.7. Line 1 defines the two Trap Modes that can exist, direct and vectored. Direct Mode always traps (branch to a routine address) to a fixed address independently of the kind of trap. Vectored Mode will have a dynamic trap address, once it is based on the cause of the trap, it will add the base trap address to the code of the interrupt or exception resulting in the exact trap handler for that trap. Line 3 defines the CSRs names and addresses for each. It is defined as a type that has a 12-bit length giving a CSR address space of 4096 addresses. The defined CSRs are the mandatory Machine mode CSRs defined in the privilege mode specification. All of these have a predefined address. Line 5 and line 7 define the exception and interrupt codes used in the trap handlers.

```
1 typedef enum  logic[1:0] {DIRECT, VECTORED} TRAP_MODE;
2
3 typedef enum  logic[11:0] { MVENDORID = 12'hF11, MARCHID, MIMPID, MHARTID,
     MCONFIGPTR,  MSTATUS = 12'h300, MISA, MEDELEG, MIDELEG, MIE, MTVEC,
     MCOUNTEREN, MSTATUSH = 12'h310, MSCRATCH = 12'h340, MEPC, MCAUSE, MTVAL, MIP,
      MTINST = 12'h34A, MTVAL2, CYCLE = 12'hC00, TIME, INSTRET, CYCLEH=12'hC80,
     TIMEH, INSTRETH} CSRs;
4
5 typedef enum  logic[4:0] { INSTRUCTION_ADDRESS_MISALIGNED,
     INSTRUCTION_ACCESS_FAULT, ILLEGAL_INSTRUCTION, BREAKPOINT,
     LOAD_ADDRESS_MISALIGNED, LOAD_ACCESS_FAULT, STORE_AMO_ADDRESS_MISALIGNED,
     STORE_AMO_ACCESS_FAULT, ECALL_FROM_UMODE,ECALL_FROM_SMODE, ECALL_FROM_MMODE =
      11, INSTRUCTION_PAGE_FAULT, LOAD_PAGE_FAULT, STORE_AMO_PAGE_FAULT = 15, NE}
     EXCEPT_CODE;
6
7 typedef enum  logic[4:0] { S_SW_INT = 1, M_SW_INT = 3, S_TIM_INT = 5, M_TIM_INT
     = 7, S_EXT_INT = 9, M_EXT_INT = 11} INTERRUPT_CODE;
```

Listing 5.7 – Definition of CSR addresses and codes for exceptions and interrupts.

The implementation of the CSR Bank is made in the file *"CSRBank.sv"*. Each CSR address mapped in the package has a 32-bit registered signal corresponding to it, except the read-only that are constants assigned to the output. During the core reset, CSRs receive their initial value. Each CSR has also a write mask that can prevent some fields to be written in each register as defined in each specification. To perform operations over a CSR, the current CSR value is assigned to an internal signal called *current_val*. A switch-case operator based on the CSR address is responsible for assigning the current value and the write mask to the respective signals.

Table 5.3 presents the implemented unprivileged CSRs that were implemented in the register bank, they are unprivileged because they can be accessed in any privilege mode without raising an exception. Column *Privilege(Access)* shows the privilege required for accessing the CSR and the access type of the respective CSR, in these cases being Unprivileged and (U) and read-only (RO). The *cycle* CSR is a read Only CSR that is incremented at every clock cycle and can be read by the Zicsr instructions or by the *RDCYCLE* pseudo-instruction. The *instret* CSR is a read-only CSR incremented every time the Retire unit retires an instruction. This CSR can be read by the Zicsr instructions or by the *RDTIME* pseudo-instruction. The *time* CSR is a read-only CSR that implements a real-time timer, it was not implemented directly in the CSR Bank but its behavior was implemented in the peripherals as will be explained in Section 6.2.6.

Table 5.3 – Implemented RISC-V unprivileged CSRs.

Number	Privilege(Access)	Name	Description
Unprivileged Counter/Timers			
`0xC00`	U(RO)	`cycle`	Cycle counter for RDCYCLE pseudo-instruction
`0xC02`	U(RO)	`instret`	Instructions-retired counter for RDINSTRET
`0xC01`	U(RO)	`time`	Timer for RDTIME pseudo-instruction

Table 5.4 presents the machine-level CSRs that were implemented in the register bank, they are exclusive for machine-level privilege access and any attempt to access it with a lower privilege level will raise an exception. There are five CSR that are Read Only, these CSRs are related to core-specific information for registered commercial RISC-V distributions, such as Vendor, Architecture, and Implementation Identifiers, they all return the value 0 as the PUC-RS5 distribution is not yet classified in the RISC-V canonical implementations. Column *Privilege(Access)* shows the privilege required for accessing the CSR as machine level (M) and classifies the CSR according to the access being read-only (RO) or read-write (RW).

The *mstatus* CSR is a read-write CSR that holds the current status of the core, it is used on privilege mode changes and has some fields related to enabling functionalities such as the field Machine Interrupt Enable (MIE). The *misa* CSR is a read-write CSR that holds information about the ISA and extensions implemented by the core. This is important for generic software to identify which extensions are implemented in hardware, including to define if that kind of operation should be treated in software or can be performed by hardware. The *misa* CSR writes allows the possibility to disable some implemented extensions or to change the core instruction length from 64 to 32, for example. The *mie* CSR holds the enabled interrupts, each bit of this register corresponds to a kind of interrupt and is directly mapped with the *mip* CSR, its behavior will be explored in further Sections. The *mtvec* CSR holds the trap handler base address and is set in boot routines as will be further explained in

Table 5.4 – Implemented RISC-V machine-level CSRs.

Number	Privilege(Access)	Name	Description
Machine Information Registers			
`0xF11`	M(RO)	`mvendorid`	Vendor ID
`0xF12`	M(RO)	`marchid`	Architecture ID
`0xF13`	M(RO)	`mimpid`	Implementation ID
`0xF14`	M(RO)	`mhartid`	Hardware thread ID
`0xF15`	M(RO)	`mconfigptr`	Pointer to configuration data structure
Machine Trap Setup			
`0x300`	M(RW)	`mstatus`	Machine status register
`0x301`	M(RW)	`misa`	ISA and extensions
`0x304`	M(RW)	`mie`	Machine interrupt-enable register
`0x305`	M(RW)	`mtvec`	Machine trap-handler base address
Machine Trap Handling			
`0x340`	M(RW)	`mscratch`	Scratch register for trap handlers
`0x341`	M(RW)	`mepc`	Machine exception program counter
`0x342`	M(RW)	`mcause`	Machine trap cause
`0x343`	M(RW)	`mtval`	Machine bad address or instruction
`0x344`	M(RW)	`mip`	Machine interrupt pending

Section 5.6. The *mscratch* CSR is a register used for swapping an integer register contents with its value in the trap handler routines. The *mepc* CSR receives the PC address of the instruction causing an exception or the PC address of the instruction that was interrupted by the trap to an interrupt handling routine. The *mcause* CSR holds the value of the cause of the trap. It is read by software to isolate the cause of the trap handling trigger event to jump to a proper handler routine. Values that the *mcause* CSR can receive and its meanings are shown in Table 5.5. The *mtval* CSR is an auxiliary register that receives either the instruction that caused the instruction or the address of the instruction that caused the exception. The *mip* CSR holds the interrupts that have pending requests, it is associated directly with the *mip* register. The behavior of these CSRs will be further explored in the following Sections, associated with related software routines.

5.4.1 Read Operation

The read operation is made by a block of code that first checks if the read operation is enabled and if the instruction is not going to be killed (this will be further explained in Section 5.5). After validation, the contents of the addressed CSR is assigned to the output. This is done by a switch case command indexed by the CSR address. Read-only registers are constants that are directly assigned to the output port and the other CSRs are indexed by the output.

5.4.2 Write Operation

As previously said, each CSR has a write mask, and its current value is assigned to a hardware internal signal. Signals are combined with the data input to generate the data to be written to the addressed CSR as displayed in Listing 5.8, which has different behavior for each operation that must be performed.

To actually perform a write operation as a result of executing an instruction, it does have the lowest precedence. This happens because other operations have a higher impact on the core state. The higher precedence writes are originally from interrupts, exceptions, and trap returns. These cases imply in general more than one CSR to be written. The write is made if the write enable is set and if the instruction is not going to be killed. In this case, the addressed CSR receives the value calculated and assigned to the *wr_data* signal.

```
always_comb
    if(csr_op==WRITE)
        wr_data <= data & wmask;
    else if(csr_op==SET)
        wr_data <= (current_val | data) & wmask;
    else if(csr_op==CLEAR)
        wr_data <= (current_val & (~data)) & wmask;
    else
        wr_data <= 'Z;
```

Listing 5.8 – Generation of the data to be written in the CSR Bank.

5.4.3 Machine Return

The trap return, also known as *MACHINE_RETURN*, is caused by an *MRET* instruction, which is used at the end of a trap handling routine. It causes the core to return to the previous stage before the trap occurred. It usually rollbacks the privilege to the privilege that it had before the trap (as the PUC-RS5 core has only the machine privilege implemented, it stays the same). It does that by restoring the field *Machine interrupt enable (MIE)* of the *mstatus* CSR with the value stored in the field *Machine Previous Privilege (MPP)* of the same register.

5.4.4 Exception Trap

When an exception occurs, several CSR fields and CSRs are written at the same time. The mcause CSR will hold the cause of the exception. Bit 31 of mcause CSR will be set to zero to indicate to the handler that the trap cause is an exception. The other 31 bits will receive the code of the exception according to predefined values that are shown in Table 5.5. The field *MPP* of the *mstatus* CSR receives the current privilege. The current privilege will be set to machine mode. The field *Machine Previous Interrupt Enable (MPIE)* receives the field *MIE* both from the *mstatus* CSR and then the *MIE* is set to zero. The *mepc* CSR receives the current PC if the exception cause was an ECALL instruction, otherwise it receives PC+4. The *mtval* CSR receives the instruction that caused the exception if the exception was caused by an illegal instruction, otherwise it receives the PC that fetched the instruction causing the exception.

5.4.5 Interrupt Trap

The CSR bank receives the interrupt request signal (IRQ) from the environment and stores it in the *Machine Interrupt Pending (MIP)* CSR. The *MIE* CSR holds the enable for each kind of interrupt, where each bit corresponds to an enable bit relative to the MIP CSR. Based on the MIP CSR and in the relative enable from the MIE, a signal called *interrupt_pending* is raised to the core. When the core is able to accept the interrupt it will return a signal called *interrupt_ack* signaling the trap to the interrupt handler.

When an interrupt is acknowledged, several CSR fields and CSRs are written at the same time. The mcause CSR will hold the interruption code. Bit 31 of the mcause CSR will be set to one to indicate to the handler that the trap cause is an interrupt. The other 31 bits will receive the code of the interrupt according to predefined values that are shown in Table 5.5. The field *MPP* of the *mstatus* CSR receives the current privilege. The current privilege will be set to the machine mode. The field *Machine Previous Interrupt Enable (MPIE)* receives the field *MIE* both from *mstatus* CSR and then the *MIE* is set to zero. The *Machine Exception Program Counter (MEPC)* CSR receives the current PC.

5.5 The CSR Register Bank Integration

The CSR Bank integration was made by instantiating the module called *CSRBank* described in Section 5.4, in parallel with the Execute unit, at the third stage of the pipeline.

Table 5.5 – Machine cause register (`mcause`) values after trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	$\geq$16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	$\geq$64	*Reserved*

It receives the signals from the CSR Unit of the Execute stage and communicates with the Fetch stage and also with the Retire stage.

5.5.1 Integration with Fetch Unit

The Fetch unit is the first stage and is responsible for fetching instructions and assigning a tag to them. The Program Counter, before the changes to PUCRS-RV had only two paths, or it received the PC plus 4 or it received a new address when a branch occurred. The process used for PC generation is displayed in Listing 5.9

```
always @(posedge clk)
if (reset)
    PC <= start_address;
else if (MACHINE_RETURN)
    PC <= mepc;
else if (EXCEPTION_RAISED || Interrupt_ACK)
    PC <= mtvec;
else if (jump)
    PC <= result;
else if (!hazard && !stall)
    PC <= PC_plus4;
```

Listing 5.9 – Integration of the CSR Bank with Fetch Unit.

The CSR Bank integration resulted in more branching options for the PUC-RS5 organization. When an MRET instruction executes, the PC receives its older value, previously stored in the MEPC CSR. When an exception is detected or an interrupt is acknowledged, the PC receives the MTVEC CSR value that holds the trap handler address. The other cases are the standard ones and already existed. This work only increased the validation of the stall *stall* along the hazard verification. This topic will be discussed in Section 5.9.

Another required change in the Fetch unit is the increment of the tag in these cases. Before, the tag was only incremented when a branch was taken, now it is also incremented when an MRET is executed or when an exception is detected, or when an interrupt is acknowledged. In other words, the tag is increased every time the PC receives a new value which is different from an increment by 4.

5.5.2 Integration with the Retire Unit

The *Retire unit* has also a tag system, used to validate instructions to retire. The tag is also increased when the execution flow of the core is changed, in the cases mentioned in Section 5.5.1. Since the two tag systems must have the same behavior regarding the pipeline operation, this unit is modified in the same way as the Fetch unit.

The *Retire unit* is essential to the CSR Bank functionality. It validates executed instructions and raises exceptions in case they do not work properly. Also, it is responsible for the ECALL, EBREAK, and MRET instructions. These instructions operations are triggered by the Retire unit by raising an exception or indicating an MRET to the CSR Bank and passing proper data such as exception codes and arguments.

Interrupts are also taken by this unit. After the recognition of an *interrupt_pending* signal and finding the right moment to accept the interrupt, it generates an acknowledgment for the interrupt source. This acknowledgment triggers the required changes in the CSR Bank and also causes a branch to the handler routine in the Fetch unit.

5.6 Software Support for the Privileged Architecture

The hardware side of the privileged architecture is just a base for the privileges modes, that have their behavior defined by the software. The software is responsible for defining the routines that implement the trap handling. One of the software's responsibilities is setting the initial state of the required CSRs. Also, in the direct mode, when a trap occurs it applies a filter to determine what is the trap cause and then jumps to the respective handler. The software was built in the RISC-V assembly language and is compiled by a GCC compiler into a binary. As the focus of this work is the hardware side of the implementation, the built software is basic and simple, with goal to validate the implemented organization.

Figure 5.3 represents the trap handling flow. Initially, an event occurs which triggers the peripheral to generate a trap request. Next, the CLINT accepts the trap request, signaling an acknowledgment and changing the core state to a handling state, while stores the previous state. After that, the PC receives the generic trap handling address. The trap handler routine decodes the trap cause and branches to the trap specific handler. Finally, the return handler executes a machine return (MRET) instruction that restores the previous PC and the previous state in the CSRs and CLINT.

5.6.1 Boot Configuration

The assembly code in Listing 5.10 displays the boot routine. At startup, the software enters a code section with the label *"boot"*. This section is responsible for setting the initial state of all CSRs. The first step is to define the address that the PUC-RS5 core must branch when a trap occurs. As defined in the specification the mtvec CSR holds the trap handler

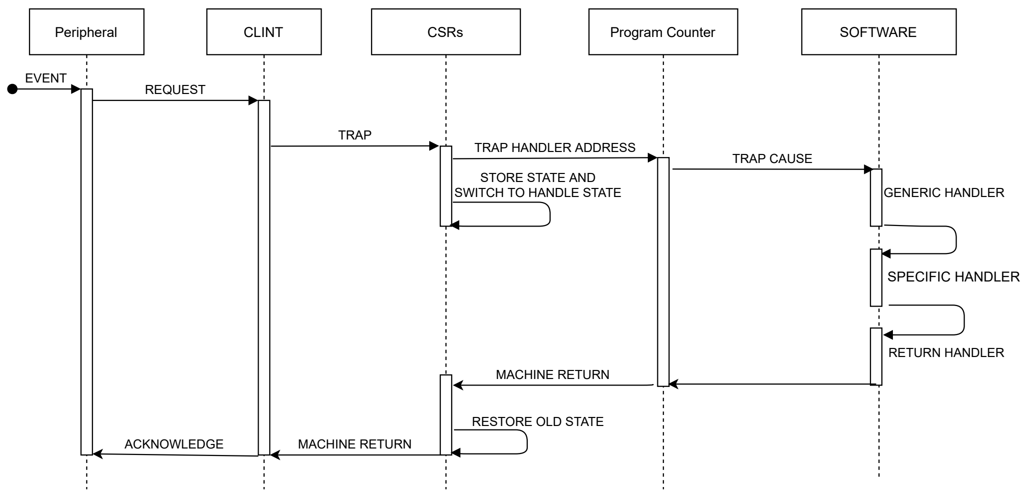


Figure 5.3 – The PUC-RS5 trap handling flow.

value. Initially, the address of the trap handler label is loaded into register *t0*, and then it is written to the mtvec CSR with the CSRW Zicsr instruction.

```
.globl boot

    # Set the trap handler address
    la t0, trap_handler          # Trap Handler address is loaded into t0
    csrw mtvec, t0               # and then written in the MTVEC CSR

    # Enable Global Interrupts
    li t1, 0x80                  # Sets MPIE (bit 7)
    csrs mstatus, t1             # of MSTATUS CSR

    # Enable External Interrupts
    li t2, 0x888                 # Set bits 11, 7 e 3 (MEIE, MTIE e MSIE)
    csrs mie, t2                 # of MIE CSR

    # Adjust mscratch
    la t1, reg_buffer            # Loads address of a memory region
    csrw mscratch, t1            # in the MSCRATCH CSR

    # Changes to USER MODE
    csrr t1, mstatus             # SET MPP field
    li t2, ~0x1800               # of MSTATUS CSR
    and t1, t1, t2               # with user mode (00)
    csrw mstatus, t1

    la t0, main                  # Loads main address
    csrw mepc, t0                # in MEPC CSR
    mret                         # Execute a Machine Return
```

Listing 5.10 – Code for the boot routine.

Register mstatus is initialized by default with interrupts disabled, to guarantee that an interrupt does not happen during boot. One of the boot steps is then to set the bit that globally enables interrupts. To keep it disabled until the boot process is complete the correct way is to set the field *MPIE*. This is done by setting bit 7 of the mstatus CSR. When the MRET instruction is executed, the MPIE field is loaded into the MIE field of the same CSR. When that is done interrupts are thus enabled.

To actually enable the interrupts, the mie CSR must be configured. Each bit of this CSR corresponds to an enable to a kind of interrupt that is requested by setting that bit in the mip CSR. As the three predefined interrupt types are external, timer and software interrupts,the bits corresponding to these are set. The enable bits for these 3 interrupt types are bits 11, 7, and 3 respectively, which are set during the boot process in line 12 and line 13.

The mscratch CSR is a CSR that holds a memory address is reserved for use during trap handling to store the context. This CSR is loaded with the address of the "reg_buffer" label, a predefined memory region reserved for this purpose.

Lines 19 to 23 load the mstatus CSR content into the *t1* register and then sets the MPP field to the user privilege. This field will be loaded with the current privilege when a machine return occurs. As the PUC-RS5 core has only the machine mode implemented, this change does not cause any effect, but it is kept to display how mode change is expected to occur in future core versions.

The last three lines conclude the boot process, by loading the label *"main"* into the mepc CSR. This CSR content will be loaded into the PC when a machine return instruction is executed. Lastly, the MRET instruction is executed, by applying all effects programmed by the *Boot Routine*, concluding the process of putting the CSRs to a known state.

5.6.2 The Trap Handler

The code in Listing 5.11 displays the assembly code used in the trap handling setup routine. This routine is responsible for the identification of the trap cause. The first thing it does is to swap the *a0* register and the mscratch CSR. The value loaded in a0 is used as the address to store the current context. The values of the integer register *a1-a5* are then stored in memory, to enable their use in the routines.

```
trap_handler:
    csrrw a0, mscratch, a0

  sw a1, 0(a0)
  sw a2, 4(a0)
  sw a3, 8(a0)
  sw a4, 12(a0)
  sw a5, 16(a0)
```

```

    csrr  a1, mcause        # a1 is not used in programs

    bgez a1, exc_handler    # Branch to an Exception handler if is an exception
    andi a1, a1, 0x3f       # Isolate interrupt cause
    li a2, 11               # a2 = External Interrupt
    beq a1, a2, MEI_handler # If is a MEI branch to MEI_handler
    li a2, 3                # a2 = software Interrupt
    beq a1, a2, MSI_handler # If is a MSI branch to MSI_handler
    li a2, 7                # a2 = Timer Interrupt
    beq a1, a2, MTI_handler # IF is a MTI branch to MTI_handler

    j return                # Else just returns
```

Listing 5.11 – Code for the trap handler routine.

The routine reads the mcause CSR into the *a1* register. Next, it works in isolating the cause. If bit 31 is set to 0 the Cause value is positive, which implies an exception since bit 31 of the mcause CSR is the field that defines an exception when is set to 0 or an interrupt (when set to 1), as explained in Section 5.4.4. If the Cause value is positive then the *Branch if Greater or Equal to Zero (BGEZ)* will branch to the exception handler.

The other lines perform field isolation and comparisons with the interruption cause codes. When a match is found, the routine branches to the respective handler. If no match is found, it just returns to the normal application flow.

5.6.3 The Exception Handler Routine

The code in Listing 5.12 displays the assembly code used in the Exception Handler routine. This routine simply treats the return address located in the mepc CSR. In Exceptions the mepc holds the address of the instruction that caused the exception, as the most common exceptions are ECALL instructions then the regular flow must be retaken and for that, the return address must be the next instruction. Therefore the mepc value is loaded into an integer register and then increased and then placed in mepc again. After the mepc is increased the return process is called.

```
exc_handler:

    # Adjust Mepc to return from a ECALL
    csrr a1, mepc # Loads return address
    addi a1, a1, 4 # Sums 4 to the return address
    csrw mepc, a1 # Updates MEPC to new value

    j return
```

Listing 5.12 – Code for the exception handler routine.

5.6.4 The Interrupt Handler Routine

The code in Listing 5.13 displays the assembly code used in the interrupt handling routine. The interrupt handlers are specific to each kind of interrupt. In this case, to keep it simple they are almost the same, being different only in the string output that is sent to the memory-mapped output register to be printed on the screen.

```
MTI_handler:
    li a1,0x80001000
  addi a2,zero,'\n'
  sw a2,0(a1)
  addi a2,zero,'M'
  addi a3,zero,'T'
  addi a4,zero,'I'
  addi a5,zero,'\n'
  sw a2,0(a1)
  sw a3,0(a1)
  sw a4,0(a1)
  sw a5,0(a1)

    j return
```

Listing 5.13 – Code for an interrupt handler routine.

Initially, the output register address is loaded into register *a1*. After that, some ASCII codes are loaded into integer registers to be used in stores later. The *Store Word (SW)* instruction uses the address stored in register *a1* and the ASCII codes are loaded in registers. These stores send the data to the memory-mapped output register. The output register is responsible for printing the data on the screen. This is made by either just calling a System Verilog print Function or by sending it over a UART to the host computer screen, as will be explained in Section 6.2.4.

5.6.5 The Return Handler Routine

The code in Listing 5.14 displays the assembly code used in the Return Handler routine. This routine undoes the context store made in the trap handler routine as explained in Section 5.6.2. First, it reads the 5 values stored in the memory into their origin and then undo the swap of the *a0* register and the mscratch CSR.

```
return:
 lw a1, 0(a0)
 lw a2, 4(a0)
 lw a3, 8(a0)
 lw a4, 12(a0)
 lw a5, 16(a0)

   csrrw a0, mscratch, a0
   mret
```

Listing 5.14 – Code for the return handler routine.

5.7 Register Bank Changes

The Register Bank suffered some changes in its implementation aiming for less area usage. On the PUCRS-RV core, the register bank had a tri-state logic for assigning values to the bank outputs. The zero register was also emulated to preserve the use of one 32-bit register, but this behavior required the tri-state organization. The write address was made by a 31-bit one-hot signal, where each bit represented the enable for the corresponding register.

The first change was to remove the tri-state logic since it usually increases area usage. After that, the write address was simplified to a 5-bit encoded signal. This was made to keep the pattern between the two usages of the register bank that will be discussed in Section 6.2.1.

5.7.1 Locked Registers Queue

The one-hot write address of the register bank was made in this format because of the locking register queue. This mechanism is responsible for keeping track of the register that has pending writes. They were stored in a queue and the positions of the queue were merged into a wired-OR signal.

After the change of the one-hot signal to an encoded address, the locked registers queue had to change its behavior to accommodate 5-bit signals instead of the old signal with 31 bits. The locked registers queue then did not merge itself in a unique signal anymore and the test must address each position of the queue individually.

These changes were made mainly aiming at the usage of LUTRAMs during the core FPGA prototyping. The project of the LUTRAM-based register bank is discussed in Section 6.2.1. To also be able to use a regular register bank when the simulation is made in a non-prototyping environment, compiler directives were created. This Directive relies on a *DEFINE* directive called "PROTO" in the TOP level RTL. This directive must be set when the intended version is the prototyping and unset when RTL simulation is the intended usage. This directive is used in an *"ifdef"* block that switches between the two versions of the register bank.

5.8 Changes to the Memory Interface

The memory interface suffered changes to accommodate the BRAM usage in prototyping, discussed in Section 6.2.2. These changes addressed the need to make aligned memory accesses, which was not the case in PUCRS-RV. Before, the interface had separate addressing signals for the data memory reads and writes. This organization would have caused the need to use a single port BRAM just for the instruction memory and another dual-port BRAM for the data memory. As the intended behavior of the circuit is to keep the area and resources as low as possible, the interface changed to have only one set of ports for data access. Figure 5.4 presents a diagram of the input and output signals related to memory and Table 5.6 presents a brief description of each signal.

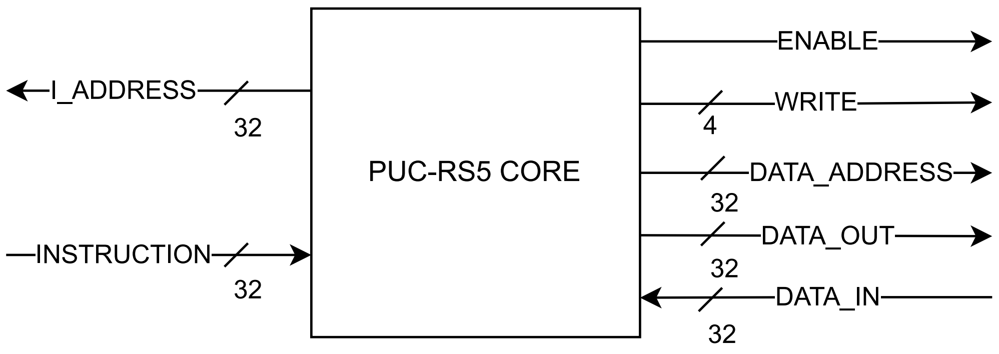


Figure 5.4 – The PUC-RS5 memory interface.

To access the instruction memory there are only two signals. The PUC-RS5 core outputs the *I_ADDRESS* signal which contains the address of the requested instruction. In the next cycle, the *INSTRUCTION* signal is provided by the memory.

The data Memory interface has five signals, one input, and 4 outputs. These signals are detailed in Table 5.6. The input signal is the *DATA_IN* that is used in memory reading operations. Memory access is only allowed when the *ENABLE* signal is set to one, otherwise, all other signals are ignored. The *WRITE* signal indicates to the memory the intended operation, indicating a write when its value is set to 1, and a read is set to 0. The *DATA_ADDRESS* signal is the signal that addresses all data memory operations. Finally, signal *DATA_OUT* is the output data to be written in memory in a write operation.

Table 5.6 – Description of the memory interface signals.

Signal	Width	Direction	Description
I_ADDRESS	32 bits	output	Instruction request address
INSTRUCTION	32 bits	input	Requested instruction
ENABLE	1 bit	output	Enable for memory operations
WRITE	4 bits	output	Byte-wide Write enable
DATA_ADDRESS	32 bits	output	Data operation address
DATA_OUT	32 bits	output	Data to write in memory
DATA_IN	32 bits	input	Data read from memory

As the data memory uses the same address signal for reads and writes the core has to manage the use of this resource. Memory reads are made by the Execute unit of the pipeline and the writes are made by the Retire unit, which can lead to concurrency for the memory signals. To avoid the concurrency and possible failures by read and write operations trying to access the memory at the same time, a data hazard detection mechanism is employed. Memory conflict detection keeps track of the presence of a memory write instruction in the pipeline in a queue, similar to the locked register queue, but with a single bit indicating if a store instruction is present in that stage. This mechanism allows memory instructions of the same kind, stores or loads, to be propagated in the pipeline, but does not allow a store instruction to be followed by a read to avoid the operations concurrency.

Listing 5.15 displays the data memory access control based on the core internal read and write signals. The memory enable signal is only set to one if an internal read or write signal has a value set to a value different from zero. If the write signal is different from zero, this means that a write operation must be made. The memory addressing signal *Data_address* receives the internal *write_address* signal, otherwise it receives the *read_address* signal. The lowest two bits of the address signal are always set to zero as the data access is always aligned to a 32-bit word frontier.

```
always_comb begin
    if(write!='0)
        DATA_address[31:2] <= write_address[31:2];
    else
        DATA_address[31:2] <= read_address[31:2];

    DATA_address[1:0] <= '0;

    if(write!='0 || read==1)
        enable <= 1;
    else
        enable <= 0;
    end
```

Listing 5.15 – Control of the data bus access.

5.9 Pipeline Stall

To be able to perform some operations such as communication with the environment, the core must be able to hold its state until the operation is ready/complete. For that, the *stall* signal was created. The *stall* signal is responsible for holding the pipeline stages propagation when it is set to one. For that, this signal is used in all the pipeline registers as an enable signal. In this way, when the stall signal is set then all registers hold their previous state on subsequent clock edges, until stall is reset.

The *stall* signal in the first two stages is used alongside the *hazard* signal since it also works as an enable for the pipeline's first two stages registers, in the bubble propagation mechanism. The bubble propagation is different from a stall state. In bubble propagation, it is necessary that the Execute and the Retire stage proceed with their operation to solve a data conflict (hazard) identified in the register locking queue. In a stall, the Execute and the Retire stage must hold their state, since probably the stall state is caused by a memory access operation. The core stages must hold their state until the resource is clear or until the data is ready to be read.

Cases that can lead to a stall state are related to the core environment. The most common possibility is when a write to the memory-mapped output register is tried and the output register buffer is full. In this case, the write signals must be held until the output register completes the current operation and releases space in the buffer. Another case can be when the core is trying to read data from a peripheral that takes longer than 1 clock cycle to provide the requested data. This case can be described by the reading of a peripheral that has to first process the value to be returned.

6. THE PUC-RS5 CORE VALIDATION PROCESS

This Chapter presents the PUC-RS5 core validation process. It uses two environments, one to validate the core only through simulation, detailed in Section 6.1. This is called *testbench environment*. The second environment validates the core in a minimalist FPGA prototyping environment. The later is the target of Section 6.2, and is accordingly called *prototyping environment*.

To support the process of using the PUC-RS5 core, Table 6.1 presents a set of memory-mapped registers employed by both validation environments. These registers are mapped to addresses higher than the hexadecimal address *0x80000000*. The testbench environment emulates the functionality of these registers using System Verilog functions and the prototyping environment implements their behavior using components described in the following sections.

Table 6.1 – Memory-mapped registers employed in the PUC-RS5 validation process.

Address	R/W	Size	Description	Usage
0x80000000	W	8 bits	Application end register	Indicates to the environment that the application finished
0x80001000	W	8 bits	Output register	Prints data on screen (sent to the UART)
0x80004000	W	8 bits	Output register	Prints data on screen (sent to the UART)
0x80006000	R	32 bits	Real-time timer	Returns the core running time in μs.

6.1 The PUC-RS5 Testbench Environment

For the validation of the core only, the environment used is just a regular Hardware Description Language (HDL) testbench that emulates peripherals and memory to simulate their behavior in conjunction with the core. The testbench provides the *clock* and *reset* signals to the core. It instantiates the processor and an abstract RAM module. The memory-mapped registers have a behavior modified to simplify their use. This is made to have a simple and fast environment to validate the core, since this can be simulated in regular simulators such as *Mentor Modelsim or Questa, and Cadence Xcelium*. The testbench helps validating minor changes in the core, allowing to build the validation environment much faster through the use of command-line scripts. The testbench environment also offers more powerful debug functionalities, such as logging all the memory operations into external files, which help the bug tracking process.

6.1.1 The Testbench RAM Module

The RAM module consists in System Verilog RTL code that implements the interface described in Section 5.8. It emulates a true dual-port memory with synchronous reads and writes. Read operations return the requested data on the next clock cycle and a similar behavior is implemented for writes. The RAM has a word-addressed behavior and allows byte-wide writes.

The RAM module initially opens a binary file containing the compiled software to be executed by the core and loads these file contents into an array of 65536 byte-wide positions. After memory initialization, it creates debug file pointers. Each pointer accesses a file located in a sub-folder called *Debug*. Every log entry contains the simulation time of the operation alongside with its contents. There are three debug files:

1. *instructions.txt* - keeps track of every instruction fetch, the address and the instruction fetched are logged to it;
2. *reads.txt* - this file contains the log of every data read from memory and its address;
3. *writes.txt* - receives a new entry every time a write is performed and outputs the address of the write and the new value, alongside with the write signal that indicates what bytes of the addressed word received new data.

6.1.2 The Testbench Memory-Mapped Registers

The memory-mapped registers presented in Table 6.1 have their implementation individually implemented in the testbench as described below.

Applications that print characters on the screen employ output registers mapped to the hexadecimal addresses *0x80004000* or *0x80001000*. It is expected that these memory-mapped registers receive a byte-wide signal. The PUC-RS5 testbench environment implements the register functionality using System Verilog *$write* function. This function prints a byte in the screen/console of the program running the simulation.

A write to address *0x80000000* indicates the end of program execution. In the testbench, the Verilog *$finish* function is called after a finish message is logged to the console. This function ends the simulation and usually closes the program running the simulation.

The register mapped to address *0x80006000* behave as a real-time timer whose value can be assessed by reads to its address. This value returns the simulation time and is implemented by the use of the System Verilog *$time* function. As the software expects the value returned by the timer to express time in microseconds units and the testbench timing precision is set to nanoseconds, then the value returned to the core is converted from nanoseconds to microseconds by dividing the value returned by the *$time* function by 1000. This timer is used by the Coremark benchmark for performance measurements.

6.2 PUC-RS5 Prototyping

The PUC-RS5 core validation in an environment with real-world components behavior relies on an FPGA hardware prototyping process. The prototyping process employs a *Digilent NEXYS A7* FPGA board with the *Xilinx xc7a100tcsg324-1* FPGA device in it. All components described in this Section use resources from this board.

This environment is instantiated by a module called **PUCRS5_With _Peripherals** which organization is depicted in Figure 6.1. This Figure groups the signals described in Table 5.6 and showed in Figure 5.4.

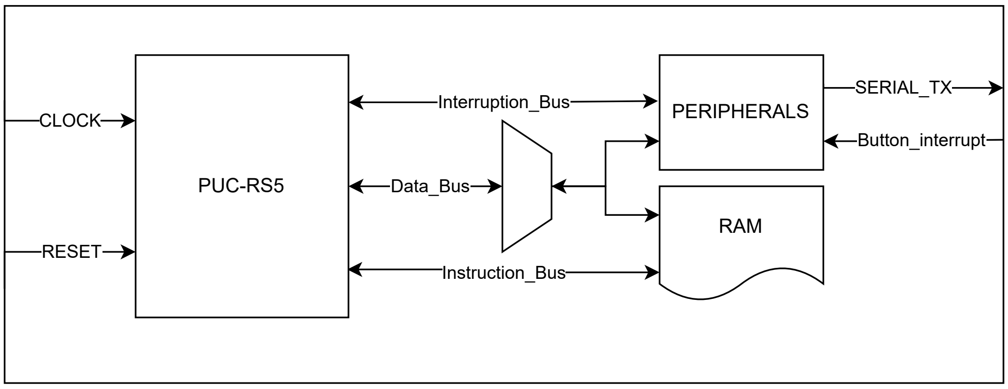


Figure 6.1 – The PUC-RS5 core validation prototyping environment block diagram.

In the prototyping process, the environment relies on the use Xilinx BlockRAMs available in the FPGA device, instead of using emulated RAMs as described in Section 6.1.1. The memory subsystem corresponds to the block called RAM in Figure 6.1. BlockRAMs or BRAMs in short are high-density configurable memory hardware modules available in Xilinx FPGAs, usually containing from 16Kbits to 36Kbits of memory. For the core register bank implementation, an alternative version of the PUC-RS5 description was produced. The new System Verilog description relies on specific Xilinx device libraries and uses a Xilinx FPGA primitive device called LUTRAM instead of generic System Verilog code to describe the 32 32-bit register bank. LUTRAMs are a much more area- and power-efficient way to produce register and register files in FPGAs. Note that CSRs logic is too convoluted to justify their implementation with LUTRAMs.

Besides the core memory subsystem built with BRAMs, and a new core register bank description optimized for FPGAs, the prototyping environment contains a universal asynchronous receiver transmitter (UART) module to exchange data with the board host computer. This UART has a FIFO implemented using LUTRAM registers to reduce the core pipeline stalling during communication with the external world. The environment also counts with a timer that generates timer interrupts. One of the input buttons of the board is mapped as an external interrupt source for the core. The blocks described in this paragraph correspond to the block called PERIPHERALS in Figure 6.1.

Still regarding Figure 6.1, the core instruction-related signals are grouped in a line called *Instruction_Bus* and the line called *Data_Bus* represents the grouped data-related signals. The same is true for the IRQ and interrupt acknowledgment lines, which are grouped into the *Interrupt_Bus* composite signal.

Finally, it is possible to note in Figure 6.1 the PUC-RS5 core instantiation and signals connecting it to the RAM subsystem, detailed in Section 6.2.2 and the Peripherals module, detailed in Section 6.2.3.

The *TOP* module, that constitutes the whole prototyping environment has its structure somehow abstracted in Figure 6.1. It in fact manages data enable signals that are passed among the main hardware modules, based on decoding address actions from address lines produced by the core. The RAM subsystem implements the same RAM behavior presented in Section 6.1.1. It receives the enable signal when the core-generated address is within the range of 0 to 65536, as this is the prototyped memory size. When the Address is larger than 65536 the processor is assumed to be addressing a peripheral. In this case, the enable signal is then passed to the *Peripherals* module. The four ports used of the prototyping environment design shown in Figure 6.1 are three inputs and one output, being the output the SERIAL transmitter port and the inputs the signal coming from the external button, and the clock and reset control signals.

6.2.1 The PUC-RS5 Core LUTRAM Register Bank Version

The register bank often takes more than half of the area usage of small processor core. To reduce this area usage in FPGA boards, a feature called LUTRAM is available. It uses the *Look Up Tables (LUTs)* of the board to produce multi-bit registers, instead of using the output registers of the *Configurable Logic Block (CLB)*, which can take a large number of CLBs to implement the entire register bank. A LUTRAM can be inferred by the tool in the optimization process or can be directly configured and instantiated in the design.

Initial attempts to infer LUTRAMS for PUC-RS5 register bank were made but without success. This difficulty to infer LUTRAMs is due to the register bank interface counting with two read ports and one write port. The easiest way to solve this point was to rely in the explicit declaration and configuration of registers as LUTRAMs. To enable an easy access to the later choice, a tool is available in the Intellectual Property (IP) modules Catalog or *"IP Catalog"* inside the Xilinx Vivado design environment. The specific tool is called *"Dis-*

tributed Memory Generator". It generates an IP hardware module (also called IP core) that implements LUTRAM registers that can then be instantiated in any design.

The configuration of the desired block is intuitive, using a graphic user interface (GUI). It Initially offers options for sizing the dimensions of the block, the *Data Depth* and the *Data Width*. The register bank for a RISC-V 32I core has 32 data registers, that is the block *Data Depth*, each register having 32 bits, which corresponds to the block *Data Width* dimension. Input and output port configurations are made in a way that they replicate the behavior of the regular register bank. Reads are made asynchronously and writes are synchronous. At the reset, all registers receive the constant value of zero.

In the *"Distributed Memory Generator"* tool, the memory type can be defined in 4 ways, one option is as a ROM memory and three are available for read and write (RAM) configurations. The option selected was to have a simple Dual Port RAM, where one port is for writing and the other is for reading. As the register bank works much more efficiently by having two read ports, the register bank was duplicated. The first register bank is responsible for the read of the A operand and the second for the B operand. To keep the data in both blocks identical for the reads, every write performed in the register bank is executed simultaneously in both blocks.

6.2.2 The BRAM-based PUC-RS5 Memory Subsystem

To implement the PUC-RS5 core memory, several possibilities are available. Using the testbench environment RAM module would be an option, if it was written as a prototypable piece of code. But this would most certainly use much of FPGA resources, maybe making the prototyping unfeasible in the available device. Another option would be to use the large external memory devices available in the Nexys 7 board outside the FPGA. The Nexys A7 board contains two external memory devices: 128 Mbytes of DDR2 SDRAM and 16MBytes of nonvolatile serial Flash [Dig19]. Besides, it contains a slot where an SDCard can be inserted, supporting still larger amounts of memory. However, any of these options would imply developing or retrieving memory driver hardware and software, increasing the time taken beyond that available to develop this work. A third option, which also avoids the high use of CLB slice registers to implement memory, is to employ memory IPs available in FPGA devices, called BlockRAMs or BRAMs. BRAMs can store medium to large amounts of data, without the need to recur to use external devices. They are used for building large First-In-First-Out (FIFO) memories and are configurable as single or true dual-port memories. Dual-port memory supports simultaneous reads and writes in any two independent addresses, while a single port has only one access, used for read or write operations in a half-duplex mode. In simple dual-port RAMs, one port is used for reading and another for writing, while in true dual-port RAMs both ports can be used for reads and writes.

The RAM subsystem used in the prototyping process was generated using a tool present in the same *"IP Catalog"* of the Xilinx Vivado environment. The tool in question is the *"Block Memory Generator"*. This tool produces a memory block based on the configurations inserted in the tool GUI. The generated block was configured with the *True Dual Port* option, with both RAM subsystem ports sharing a same clock. The RAM subsystem Write width was configured with the 32-bit value and the write enable as byte-wide signal, which means that the enable signal for the write has a width of 4 bits. Each bit of the enable signal allows the write operation of the corresponding byte(s) of the word sent for writing. This allows to have word-aligned memory accesses and still be able to perform byte-wide writes. The memory depth was set to 16384 words. As each word has 4 bytes (32 bits), the resulting total size of the RAM subsystem is 64 Kilobytes or 65536 Bytes.

The first port of the RAM subsystem is used for instruction fetching and is connected directly with the processor signals. This port only performs read operations. The second port is used for the DATA interface and is a read/write port. It receives signals directly from the processor except for the enable. The enable that the RAM subsystem receives is a signal generated by the TOP module that instantiates the core and the RAM subsystem, the signal is generated based on the address signals produced by the PUC-RS5 core. As the RAM subsystem was configured with 64 Kbytes of space, the RAM subsystem enable is only set when the address is within this range. Otherwise, the processor is addressing a peripheral. Refer here to Listing 6.1. The RAM subsystem is initialized using a *".coe"* file, which holds the contents to be loaded into memory.

```python
with open("input.bin", "rb") as file_in:
    rd = file_in.read(-1)
    byteArray = bytearray(rd)

    while len(byteArray) < 65536:
        byteArray.append(0)

with open("./output.coe", "w") as file_out:
    file_out.write("memory_initialization_radix=16;\n")
    file_out.write("memory_initialization_vector=\n")

    for i in range(0, len(byteArray)-4, 4):
        file_out.write(str(hex(byteArray[i+3])[2:].zfill(2)))
        file_out.write(str(hex(byteArray[i+2])[2:].zfill(2)))
        file_out.write(str(hex(byteArray[i+1])[2:].zfill(2)))
        file_out.write(str(hex(byteArray[i+0])[2:].zfill(2)))
        file_out.write(",\n")

    file_out.write("00000000;")
```

Listing 6.1 – Memory Initializer Generator.

The content is placed in the *coe* file with a memory position per line in the header-specified encoding. As the memory *Data Width* was configured as 32-bit width, each memory position holds 32 bits. Thus, each line of the file will hold a 32-bit content. The encoding of each line is set on the header of the file on the *memory_initialization_radix* statement. The encoding can be set to binary(2), octal(8), decimal(10), and hexadecimal(16). The file contents must then agree to the selected encoding.

The ".coe" file is generated by a Python program that takes the compiled binary as input and generates the *coe* output file. This program was created in the context of this work. First, it reads the binary file content into a byte array with a length of 65536 positions (the same size configured in the RAM subsystem). After the content is read, it writes the header of the ".coe" output file, setting the number format to hexadecimal, and then initializes the contents vector. The contents vector receives a word per line. For that, the loop iteration is made by steps of 4 and the 4 bytes are then arranged into a 32-bit word. After the loop iteration, the program ends the initialization vector with the *";"* character and finishes execution.

The RAM subsystem is instantiated by the TOP module RTL code used for FPGA prototyping. This RTL connects the signals from the PUC-RS5 core and the BRAM. It is also responsible for generating the RAM subsystem enable, based on the address generated by the core.

6.2.3 Peripherals

To separate the RAM subsystem from the other prototyping environment features, a module called Peripherals was created in the *Peripherals.sv* file. This module is instantiated by the TOP module, called *PUCRS5_With_Peripherals*, as presented in Figure 6.1. The peripherals module receives the core signals always when the processor tries to access an address with value larger than the predefined memory size. This keeps a separation between what is access to the RAM subsystem and what is related to the external world provided by the prototyping environment.

Figure 6.2 presents a block diagram of the Peripherals module. It contains a data access controller, which controls the writes to the peripherals memory mapped registers. This is used only by the UART module, it sends write signals to the FIFO Buffer based on the FIFO status signals. This controller also manages the reads from register-mapped addresses such as the real-time timer.

The Peripherals module instantiates the UART transmitter alongside with the FIFO used as a Buffer for the UART. It contains a timer set to generate an interruption every 5 seconds. The module also contains a Button Debouncer module and a hardware that generates a single interrupt request per button press. This interrupt generator module is

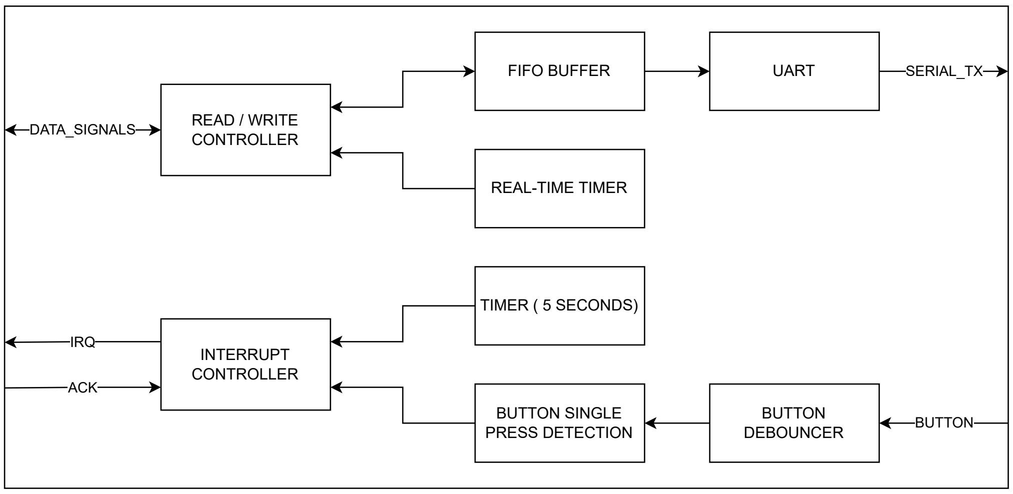


Figure 6.2 – The Peripherals module block diagram.

controlled by a hardware that generates the *Interrupt Request (IRQ)* signal to the core and manages the acknowledgment received from it.

The mentioned modules are briefly detailed in the following sections. Section 6.2.4 explores the UART and its Buffer, while Section 6.2.5 presents the interrupt controller and Sections 6.2.6 and 6.2.7 present the Timers and Button related modules implementation, respectively.

6.2.4 The Prototyping Environment UART

The UART is a module used for sending data to the outside of the PUC-RS5 prototyping environment. The UART acts in response to writes to a memory-mapped register used for printing on the screen. The UART serial output signal is connected to one of the serial output ports of the FPGA board. This signal is then passed through the same cable user for FPGA configuration, connected to the host computer. In the host computer, a terminal emulator program is configured and executed to receive serial data. This terminal input is set to connect to the serial port connected to the board. This way, data sent through the serial port by the PUC-RS5 UART is just printed on the terminal screen.

The prototyping environment UART module used is a UART transmitter that receives a byte as input and transmits it using a serial protocol. The protocol used is the 8N1 standard that has no parity bit and has 1 stop bit for each data byte sent. The baud rate used is configured to 9600bps.

As previously said, the UART transmission is triggered by writes in the memory-mapped register used in the prints present in RISC-V code software. When a write occurs to these mapped registers data is sent to the UART if the UART is not busy. If the UART is still busy sending the last byte, then the core would need to hold its state until the UART is ready to receive a new byte to transmit. This is made by setting the PUC-RS5 *stall* signal until the UART becomes available, which forces the core to hold the write signals in the same state, ensuring that no data is lost.

To avoid the need of stalling the pipeline at every write, a FIFO is instantiated to add throughput to UART writes on the core side. This FIFO was configured using a tool called *FIFO Generator* from the *IP Catalog* of the Xilinx Vivado environment. The tool generates an IP that can be instantiated into the design. The configuration screen gives various alternatives for FIFO implementations, such as BRAM, LUTRAM, or shift registers. The chosen alternative was to use a FIFO implemented in LUTRAMs. The FIFO configured as a standard FIFO with a Data width of 8 bits and a depth of 64 positions. When the core writes to the UART-mapped register the data is passed to the FIFO and then the FIFO passes it to the UART. When the FIFO is full and a write occurs, the stall of the processor is unavoidable. This management process is made based on the signals empty and full of the queue that indicate the FIFO status.

6.2.5 The Prototyping Environment Interrupt Controller

There is a hardware block in the prototyping environment that generates the signal *Interrupt Request (IRQ)*, which is sent to the PUC-RS5 core. Listing 6.2 shows the main excerpt of System Verilog code for this hardware block.

```
always @(posedge clk)
    if (reset)
        IRQ <= '0;
    // EXTERNAL Interrupt
    else if (IRQ[11]==1 && Interrupt_ACK)
        IRQ[11] <= 0;
    else if (Button_request==1)
        IRQ[11] <= 1;
    // TIMER Interrupt
    else if (IRQ[7]==1 && Interrupt_ACK)
        IRQ[7] <= 0;
    // 5 seconds = 100,000,000 * 10ns * 5 =  0x1DCD6500
    else if (counter>=32'1DCD6500)
        IRQ[7] <= 1;
```

Listing 6.2 – Generation of interrupt request signal.

The IRQ signal is stored in the MIP CSR that holds pending interrupts. When an interrupt is accepted and is about to be treated by software, the acknowledgment signal is sent to the environment, indicating that an interrupt was accepted. The three kinds of implemented interrupts obey to a predefined order of precedence as the interrupt controller is a CLINT controller (See Section 2.2.1 for details). The order is: first, external interrupts;

next software interrupts; lastly timer interrupts. This precedence is ensured by the ordering of the if-else statement.

The code above presents the process of generation of the IRQ signal. During reset, IRQ is set to zero. After detecting an Acknowledgment signal the signal that corresponds to the ack is set to zero representing that the request was accepted. Otherwise, the conditions to raise the request are checked. The generation of the condition for generating an external interrupt appears in line 7 and the generation of the signal *Button_request* is discussed in Section 6.2.7. The counter implementation used in line 13 as a condition is discussed in Section 6.2.6.

6.2.6 The Prototyping Environment Timers

The Peripherals module implements a 32-bit clock counter that acts like a timer. This counter is increased by one at every clock cycle. As the clock operates at a frequency of 100MHz, the corresponding clock period is 10 nanoseconds. When it reaches 100,000,000 clock counts then a second has passed. The timer is configured to generate an interruption when 5 seconds have elapsed. When this occurs and the interrupt is treated, the timer is reset to zero to start a new 5s-cycle.

Another timer is used as a real-time timer. It is implemented by a 64-bit register that is increased at each clock cycle. As the clock period is 10 nanoseconds, the value is increased by 10, this way, the register holds the value of the elapsed nanoseconds since the start of the count. This is used in some applications that read the elapsed time of execution such as some of the Coremark benchmarks. Coremark expects this value to be returned in microseconds, which brings the need to divide the read value by 1,000. As the core does not have a divider module, the value read from the timer is shifted in 10 positions to the right, performing a division by 1024, which results in an approximation for the division by 1,000 that can be accepted since this is not used in any critical operation.

6.2.7 Button Press Detection and Interrupt generation

The signal *Button_request* used for external interrupt requests is generated by the pressing of a button on the FPGA board. To implement such a request, the first thing to do was the mapping the FPGA board button signal to the core input. After that, the signal is passed to the Peripherals module for treatment. The input signal is pre-processed by a Debouncer Module that ensures the button was pressed long enough and is clean from glitches. This is made e.g. to avoid imperfections of the button signal when responding to a physical button press by a human, a very slow event. The filtering process simply registers any change in a button signal and checking if it remains constant for a defined number of clocks. The output signal of the debouncer is registered and used in a comparison operation that detects changes in the de-bounced signal. When the de-bounced signal is set to one and the registered signal is 0 then a press of the button occurred. This last step is made to ensure that a single press of the button generates a single interrupt request. This signal is then used to generate the *Button_request* signal used in the IRQ generation process discussed in Section 6.2.5.

6.3 The PUC-RS5 Prototyping Process

As previously stated, the prototyping process employs a Nexys A7 Board. The entire prototyping system used the Xilinx Vivado environment for design, simulation, and board prototyping. First functional simulation validates the design and after synthesis back-annotated simulations help ensure the correct timing functionality. The testbench is rather simple as it only needs to implement the clock/reset generation and a UART receiver for printing the data sent to it.

After the core and its environment are determined as correct by simulation, the output *bitstream* file generation takes place and is loaded into the board FPGA device using also resources from Vivado. The board connects to the host computer by means of the configuration cable, that later acts as a serial communication interface for the UART prototyping environment module. On the host computer, a serial interpreter is configured to listen to the prototyping environment uART and print everything it receives to the host screen. Through this, the Berkeley suite and the Coremark benchmarks were used as software validating tools and the physical buttons were used to validate the external interrupts of the core. The prototyping environment also communicate the occurrence of the timer-generated interrupts through the UART to the host.

7. EXPERIMENTS AND RESULTS

This Chapter discusses experiments conducted over the PUC-RS5 processor core and the obtained results. The basic validation software used for validation was the Berkeley suite that performs unit tests for each RISC-V instruction, as previously presented in Chapter 6. Section 7.1 presents area usage results assuming an specific FPGA part, the one present in the prototyping board adopted for the experiments. For performance measurements, the Coremark benchmark was used and the associated results are the target of Section 7.2, where additionally comparisons with similar cores are also described.

7.1 Area and Power Results

The area results are all taken after logic and physical synthesis using the AMD/Xilinx Vivado environment, version 2021.1. The target is part xc7a100tcsg324-1, the largest FPGA device that fits in the Digilent Nexys A7 board (https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/), the adopted prototyping platform. All resources mentioned herein, such as number of LUTs and BRAMs refer to the cited FPGA part.

As mentioned in Chapter 4 this work deals with two processor cores: PUCRS-RV (a RISC-V core without privileged architecture) and PUC-RS5 (a minimalist implementation of the RISC-V privileged architecture). PUCRS-RV has no CSRs and no feature related to a privileged architecture, it only implements the bare RV32I ISA. PUC-RS5 counts with a CSR Bank containing all mandatory CSRs and makes available the Zicsr extension. The PUCRS-RV core will be used as a base of comparison for results. Clearly, the comparison gives a clear idea of minimum overheads implied by the use of a RISC-V privileged architecture.

Area results can be interpreted in two ways, the first one is considering only the PUC-RS5 core resources usage, and the second one is considering the entire environment with external peripherals and memory(ies). Core comparison considers only the core resources, ignoring peripherals and environment features such as memories or Universal Asynchronous Receiver-Transmitter (UART) mechanisms. The results used to compare the PUC-RS5 core with similar processors from the literature are those considering the entire environment used in FPGA prototyping, to be fairer since most literature results consider the entire environment as well, even if the environment might comprise fewer resources.

7.1.1 Comparison between PUCRS-RV and PUC-RS5 Cores

This Section shows a comparison between the two processor cores developed by the Author, PUCRS-RV and PUC-RS5. Both core organizations employ exactly the same Register Bank organization, which takes 44 LUT RAMs to implement a three-port (two read ports and one write port) memory. Note that in general LUTs correspond to combinational logic, while FFs stand for sequential logic. Using the AMD/Xilinx LUT RAM feature enables a more efficient implementation of medium-sized memories such as register banks [XIL19]. The 44 LUT RAMs of the Register Bank are part of the sequential logic, although made up mostly of LUTs.

Table 7.1 presents the resource usage for the non-privileged PUCRS-RV core organization. The core uses 1,017 LUTs in its implementation, which represents just 1.60% of the LUTs available on the FPGA. The Decoding stage is the stage that consumes more combinational resources. The decoder unit uses 841 LUTs and 183 registers, which represents about 82.7% of the LUTs used in the entire core, and about 54% of the registers used by the entire core. This is due to the complexity of the unit, that performs the instruction decoding, operands fetching, register locking logic, and bubble issue.

Table 7.1 – PUCRS-RV resource usage.

Resource	Utilization	Available	Utilization (%)
LUT	1,017	63,400	1.60
FF	339	126,800	0.27

Table 7.2 shows the resource use for the PUC-RS5 core. It uses a total of 1,542 LUTs, which represents about 2.43% of the total LUTs available on the FPGA. The number of LUTs compared to PUCRS-RV grows by 525 LUTs, which represents an increase of about 51.60%. The flip-flop use also increases significantly, from 339 to 814 FFs,which represents an increment of 475 FFs, or about 140% more FFs.

The decoder unit uses 1,121 LUTs, which represents about 73% of the total LUT usage. This represents an increase of 280 LUTs, which compared with the decoder unit of PUCRS-RV is about a 33% increase. This increment is caused by the addition of the decoding logic for the Zicsr extension new instructions.

The Execute Unit is another unit that shows a significant increase in LUT usage. It uses 124 LUTs and 80 FFs in PUCRS-RV, and the privileged architecture uses 246 LUTs and 112 FFs. This increase represents about 98% more LUTs and 40% more FFs. This increment is caused by the addition of the unit responsible for executing the instructions of the Zicsr extension and performing its communication with the CSR Bank.

The CSR Bank represents the biggest usage of FFs in the design, it uses 421 registers and 106 LUTs. The LUT usage is about 7% of the total LUTs. The FF usage is about 52% of total FFs.

Regarding on-chip power consumption, the PUCRS-RV core consumes an estimated 0.129W, while the PUC-RS5 core consumes an estimated 0.136W, which represents an increase of about 8% in the privileged organization.

Table 7.2 – PUC-RS5 resource usage.

Resource	Utilization	Available	Utilization (%)
LUT	1,542	63,400	2.43
FF	814	126,800	0.64

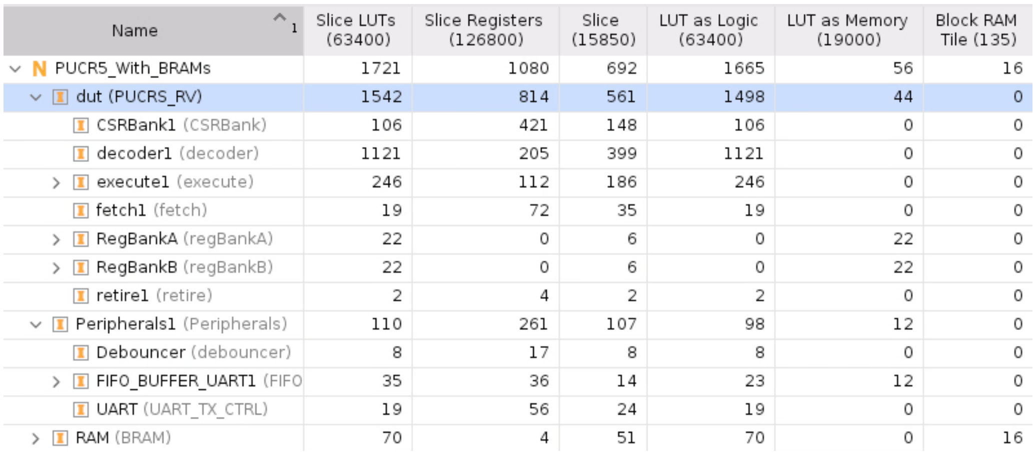
7.1.2 Comparison between PUC-RS5 and Similar Cores

To compare this work with similar cores, the entire environment used in the FPGA prototyping is considered to provide a fairer comparison with other cores, even though the PUC-RS5 counts with more environmental resources than others core cited in the literature such as including timers and interrupt controller hardware. The environment comprises RAMs and a peripheral set, the later of which include features such as UARTs, timers, and interrupt controllers as presented in Section 6.2.3. Steel, Ibex, and SCR1 processors were targeted at the same FPGA chip used in this work, as reported by Calçada in [Cal20]. All three cores are implementations of the RISC-V RV32I ISA with some degre of privileged architecture. Steel resulted from the Bachelor Thesis of Calçada [Cal20]. Ibex is a freely available core developed by the non-profit company lowRISC [Low21, Low22] and SCR1 [Syn22], provided by Syntacore, an enterprise dedicated to develop RISC-V-based Intellectual Property (IP) cores. Thus, using the data reported in this reference provides a good and fair comparison.

Figure 7.1 presents the detailed hierarchical area usage for the PUC-RS5 processor core.

The FPGA prototyping environment organization is presented in Chapter 6. The core uses 1,542 LUTs and 814 FFs (note that Vivado´Reports calls a FF by the term “register”, avoided here to prevent confusion). The peripherals consume 110 LUTs and 261 FFs. The high number of FFs used in this unit is due to the implementation of the two counters discussed in Section 6.2.6. There are 70 LUTs and 4 FFs used in the BRAM access. Observing the overall results, the core takes around 90% of all used LUTs and about 75% of the FFs used in the design.

Next, Table 7.3 presents the overall account of resources used in the PUC-RS5 and its FPGA prototyping environment.



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)
PUCR5_With_BRAMs	1721	1080	692	1665	56	16
dut (PUCRS_RV)	1542	814	561	1498	44	0
CSRBank1 (CSRBank)	106	421	148	106	0	0
decoder1 (decoder)	1121	205	399	1121	0	0
execute1 (execute)	246	112	186	246	0	0
fetch1 (fetch)	19	72	35	19	0	0
RegBankA (regBankA)	22	0	6	0	22	0
RegBankB (regBankB)	22	0	6	0	22	0
retire1 (retire)	2	4	2	2	0	0
Peripherals1 (Peripherals)	110	261	107	98	12	0
Debouncer (debouncer)	8	17	8	8	0	0
FIFO_BUFFER_UART1 (FIFO	35	36	14	23	12	0
UART (UART_TX_CTRL)	19	56	24	19	0	0
RAM (BRAM)	70	4	51	70	0	16

Figure 7.1 – PUC-RS5 detailed hierarchical area usage.

Table 7.3 – PUC-RS5 and prototyping environment overall resource usage report.

Resource Type	Utilization	Available	Utilization (%)
LUTs	1721	63400	2.71
FFs	1080	126800	0.85
LUTRAMs	56	19000	0.29
BRAMs	16	135	11.85
IO pins	4	210	1.90

Percentually, the most demanded resource by the PUC-RS5 FPGA prototyping environment are BRAMs, since the design reserves 64Kbytes of memory for program and data, taking 11.85% of the BRAMs available on the FPGA. The number of BRAMs can of course be reduced, the 64Kbyte value was arbitrarily chosen to support rather large compiled source codes and/or rather large data. The LUT RAM usage increased from 44 to 56 from the core-only values, due to the addition of 12 LUT RAMs used as a UART output buffer. Note that the FPGA prototyping environment design uses just 4 FPGA IO pins as already discussed in Section 6.2 and depicted in Figure 6.1.

To put the above results in perspective with the available literature, the Table 7.4 presents a resource comparison among the PUC-RS5 core and Steel [Cal20], Ibex [Low21, Low22] and SCR1 cores.

Table 7.4 – Comparison of PUC-RS5 with similar processor cores.

	PUC-RS5	**Steel** [Cal20]	**Ibex** [Low21]	**SCR1** [Syn22]
LUTs	1721	1626 (-5,5%)	2010 (+17%)	2359 (+37%)
LUTRAMs	56	48	48	0
FFs	814	624 (-23,3%)	790 (-3%)	1392 (+71%)
Extensions	RV32I + Zicsr	RV32I + Zicsr	RV32I + Zicsr + C	RV32I + Zicsr
Modes	M	M	M, U	M

It is important to point out that the compiled Ibex counts with the RISC-V Compressed instructions extension ("C") and the User mode, features that are not implemented on the three other cores. Another important point to consider is that SCR1 is a generic HDL description, which does not benefit from building the Register Bank with specific FPGA memory resources such as LUT RAMs. This explains in part the significantly higher number of FFs and LUTs in the synthesized SCR1 design.

The comparison results do not indicate that an implementation is better or worse than others. There are always trade offs in the design and each one has its particularities. The environments also are not exactly the same, since some implement interrupt controllers inside the core and others on the environment, while still others do not even implement this. In general, it is hard to isolate just the core from their environments or even equalizing the same environment features for performing exact comparisons without re-implementing all cores. Remember, all data in Table 7.4 except for data on the PUC-RS5 come from [Cal20].

SCR1 takes significantly higher resource usage than PUC-RS5, 37% more LUTs and 71% more FFs. SCR1 is the most similar to the PUC-RS5 and Steel cores, since it implements the same privilege modes and the same Extensions. Based on this, it is possible to consider that PUC-RS5 and Steel take less area than SCR1.

The comparison between Steel and PUC-RS5 cores is the easier and fairer to perform. Although Steel uses 5.5% less LUTs and 23.3% less FFs, the PUC-RS5 implements more features than it. It is also important to point out that Steel is a 3-stage pipeline while PUC-RS5 is a 4-stage pipeline. Steel does not implement an interrupt controller nor makes available the interrupt generation mechanisms used in PUC-RS5 for validation of the Machine mode. Meanwhile Steel implements a UART receiver that consumes around 30 LUTs and 53 FFs. PUC-RS5 in turn implements two timers (one as a 32-bit register and another as a 64-bit register), plus button debouncers and single press detection. It besides makes available a buffer for UART writes, to increase UART write performance. This additional features in the PUC-RS5 brings a significant increase in the resource use.

Based on these results it is possible to conclude that even with an additional pipeline stage compared to Steel, and bringing more features such as an interrupt controller and timers, the PUC-RS5 obtains a good compromise in terms of area occupation.

7.2 Performance Results

The PUC-RS5 performance was evaluated using the EEBMC Coremark [EDN22]. Coremark is a widely used benchmark for core performance evaluation. It produces a normalized score for a core, which allows simple comparison between several processors. The Coremark programs need to run for about 10 seconds to obtain a result that minimizes the impact of the time spent in trap handling routines that are not programmed, such as interrupt handling. Coremark was ported to the PUC-RS5 software platform and was executed in the prototyping environment (the xc7a100tcsg324-1 FPGA part of the Digilent Nexys A7 board, using the board main clock at 100MHz frequency.

The Coremark performance data for PUC-RS5 and for the three compared cores is depicted in Table 7.5.

Table 7.5 – Performance comparison of PUC-RS5 to similar processor cores.

Core	Iterations/MHz
PUC-RS5	0.670
Steel	1.360
Ibex	0.904
SCR1	1.270

Coremark performance presented results of about 67 iterations per second in the mentioned platform. For comparison with other cores this score must be normalized by dividing it by the operating frequency (expressed in MHz), which is needed because the iterations score is frequency dependent, meaning that if the frequency is doubled the number of iterations per second would probably double the previous value. The PUC-RS5 score normalization resulted in the value of 0.67 Iterations per MHz.

PUC-RS5 obtained the least performance score among all compared cores. It is important to cite that the PUC-RS5 design did not initially targets high performance. Analyzing the sources of this low performance and suggesting organization enhancements based on such an analysis is a relevant future work.

As performance results comparatively below expectations, some quick hypotheses and tests are advanced to determine the possible sources of inefficiencies, to enable increase the PUC-RS5 performance by eventually solving these.

A first hypothesis was that much time is spent waiting for the solution of data hazards. To try to minimize this hazard impact, a data forwarding mechanism is useful. This allows the register locking queue to reduce in size by one position, generating fewer hazards and issuing fewer bubbles. The forwarding mechanism uses a condition that directly assigns data coming from the Retire Unit to the Decoder Unit based on read and write addresses for instructions in the pipeline. If the write address matches one of the read addresses of a following instruction already fetched, the data the Decoder Unit gets can come directly from the Retire Unit before the Register Bank update. This change was tested and brought an increase in performance from about 67 iterations per second to about 78 interactions per second, an increase of 16.5% in performance. As this mechanism is quite simple and takes neither a lot of resources nor big complexity to the pipeline, it can be implemented easily on future versions of the core.

Even with the improvements just cited, performance is still poor compared to the other cores. As a result, this works concludes that hazard detection and bubbles are not the only organization issue responsible for the low performance.

Another hypothesis that can be raised is the cost of missed branches caused by the never-branch policy. Remember branches are performed only at the last stage, in the Retire Unit. This causes the need of discarding up to three instructions wrongly fetched. A possibility to increase performance related to the branches is implementing some branch prediction mechanism to alter the never-branch policy. But this can be costly. Another possibility is performing the branch in an earlier stage of the pipeline, reducing the cost of a missed branch as is made in the Steel core that branches always in the second stage of the pipeline. However, for conditional branches, this can make the second stage quite complex, since in this stage operands are obtained, branch addresses need to be computed, and the comparison required by the conditional branch needs to be realized. And finally, the result of the comparison needs to be used for executing (or not) the branch. Clearly, such chain of sequential actions can imply a long critical path in this stage, which might explain why Steel is clocked at only 50MHz.

8. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

This work describes a new RISC-V processor core, PUC-RS5 with a minimal privileged architecture and supporting the RV32I ISA. The main feature of the newly developed core is the support to deal with a basic set of interrupts and exception types, providing the implementation of a set of trap handlers for these. It also validates PUC-RS5 in a real-world environment using FPGA prototyping. This work is expected to serve as a base for future works on the development of embedded systems. The core is simple, cleanly code to facilitate its understanding, and built for enabling easy extension of its functionalities.

The core tests revealed some implementation caveats, due to the effort to make the initial core version a simplest of implementations. The mechanism of the data forwarding discussed in Chapter 7 alongside other performance improvement opportunities, can help reducing the number of issued bubbles in the pipeline and thus increase the core performance. The PUC-RS5 implementation showed good performance improvement with the forwarding logic tests, but it is not yet fully integrated in the first version of the pipeline. The full integration of a data forwarding unit in the core is a first future work.

Another improvement suggested in Chapter 7 is the implementation of branch prediction mechanisms to alter the never-branch policy, or changing the pipeline to perform the branch in an earlier stage of the pipeline, reducing the cost of a missed branch. These two possibilities are still pending evaluation of their trade-offs relating performance improvement and the resulting area overhead. Concerning performance improvements there is an unexploited opportunity to increasing the core operating clock frequency. A hint that this is indeed a relevant way to improve the core is the fact that absolutely no effort was made to control the design critical paths and yet, the 100MHz running frequency was achieved by an ordinary synthesis process. Probably PUC-RS5 still has some room to accommodate a much higher operating clock frequency due to its positive slack. It is expected that with little adjustments and attention to critical path analysis, the clock can be increased to higher frequencies such as 150 or 200MHz or even more. This is another interesting future work.

Another possibility is to expand the core implementation to support other privilege modes, starting by the user mode and next supervisor mode, designing the required set of CSRs. It is important to point out that other privileged modes implementation will obviously increase core area and the software side needs, since they are mostly used by more complex software stacks.

Another set of future works is the implementation of other RISC-V extensions. One of the RISC-V extensions that can be addressed is the Multiply extension ("M"), which introduces instructions for multiplication and division, operations that performed by compiler-generated code in PUC-RS5. This can be interesting for applications that require multiplications and divisions in large scale, but it is important to point that this is expected to consume a lot of area if fast versions of multiply and divide modules are added. Another extension is the atomic (A) extension. This extension introduces instructions capable of performing atomic operations that are widely used in multi-processed environments to control bus access and create semaphores. Also, the compressed extension ("C") introduces interesting instructions for reducing the software memory footprint, although the decoding process becomes way more complex for this extension, once the instruction width becomes variable, this extension does not implement any new operation to the pipeline beyond the possibility of code reduction.

All such extensions and their respective instructions are reported in Appendix A, alongside the implemented extensions. Appendix A also presents a suggested list of RISC-V assembly pseudo-instructions. This Appendix has the intention of being a guide to the implemented extensions and a link to those extensions that are interesting to be implemented in the context of near-future works. The Appendix also presents a brief description of each instruction and their performed operation, and lists the correlated pseudo instructions for software development guidance.

The software stack is also a pretty big subject that can be explored in future works, as it is intrinsic to the privileged architecture usage and is directly related to the applications where it can be used. The possibility of integrating real-time systems or a ROS system into the software stack is determinant for the use of this processor core in robotic applications.

REFERENCES

[Cal20] Calçada, R. d. O. “Design of Steel: a RISC-V Core”, Bachelor’s Thesis, Federal University of Rio Grande do Sul (UFRGS), Computer Engineering Course. Informatics Institute, Porto Alegre, RS - Brazil. Advisor: Ricardo A. L. Reis, 2020, 83p.

[Dig19] Digilent, Inc. “Nexys A7 FPGA Board Reference Manual”, 2019, Captured in: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual?redirect=1.

[DKN+21] Dehnavi, S.; Koedam, M.; Nelson, A.; Goswami, D.; Goossens, K. “CompROS: A composable ROS2 based architecture for real-time embedded robotic development”. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021, pp. 6449–6455.

[EDN22] EDN Embedded Microprocessor Benchmark Consortium. “CoreMark - An EEMBC Benchmark”. Captured in: https://www.eembc.org/coremark/index.php, Jul 2022.

[GRCRMR+20] Garcia-Ramirez, R.; Chacon-Rodriguez, A.; Molina-Robles, R.; Castro-Gonzalez, R.; Solera-Bolanos, E.; Madrigal-Boza, G.; Oviedo-Hernandez, M.; Salazar-Sibaja, D.; Sanchez-Jimenez, D.; Fonseca-Rodriguez, M.; et al.. “Siwa: A custom RISC-V based system on chip (SOC) for low power medical applications”, *Microelectronics Journal*, vol. 98, 2020, pp. 1–8.

[LCYK20] Lee, J.; Chen, H.; Young, J.; Kim, H. “RISC-V FPGA platform toward ROS-based robotics application”. In: 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), 2020, pp. 370–370.

[LNZ+22] Lodéa, N.; Nunes, W.; Zanini, V.; Marcos Luiggi Lemos Sartori, L. C. O.; Calazans, N. L. V.; Garibotti, R. F.; Marcon, C. A. M. “Early Soft Error Reliability Analysis on RISC-V”, *IEEE Latin America Transactions*, vol. 20–9, 2022, pp. 2139–2145.

[Low21] LowRISC C.I.C. “Ibex: An Embedded 32-bit RISC-V CPU core”. Captured in: https://ibex-core.readthedocs.io, May 2021.

[Low22] LowRISC C.I.C. “Ibex RISC-V Core, open-source hardware project”. Captured in: https://github.com/lowrisc/ibex, Nov 2022.

[NSC22a] Nunes, W. A.; Sartori, M. L. L.; Calazans, N. L. V. “PUCRS-RV”. Captured in: https://github.com/gaph-pucrs/pucrs-rv, May 2022.

[NSC22b] Nunes, W. A.; Sartori, M. L. L.; Calazans, N. L. V. “Pulsar ARV - A QDI Asynchronous RISC-V Implementation”. Captured in: https://lesvos.pucrs.br/williannunes/PARV, Jul 2022.

[RCFM19] Ruaro, M.; Caimi, L. L.; Fochi, V.; Moraes, F. G. “Memphis: a framework for heterogeneous many-core SoCs generation and validation”, *Design Automation for Embedded Systems*, vol. 23–3, 2019, pp. 103–122.

[RIS22] RISC-V International. “RISC-V Instruction Set Manual”. Captured in: https://github.com/riscv/riscv-isa-manual, Nov 2022.

[Sar17] Sartori, M. L. L. “ARV: Towards an Asynchronous Implementation of the RISC-V Architecture”, Bachelor’s Thesis, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Computer Engineering Course. School of Technology, Porto Alegre, RS - Brazil. Advisor: Ney L. V. Calazans, 2017, 57p.

[SC17] Sartori, M. L. L.; Calazans, N. L. V. “Go Functional Model for a RISC-V Asynchronous Organisation - ARV”. In: ICECS, 2017, pp. 381–348.

[SC21] Sartori, M. L. L.; Calazans, N. L. V. “ARV - Go High-level Functional Model”. Captured in: https://github.com/marlls1989/arv, May 2021.

[SiF20] SiFive, Inc. “SiFive Interrupt Cookbook, Version 1.2”, 2020, Captured in: https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

[SMC20a] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2020, pp. 114–123.

[SMC20b] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “ASCEnD-FreePDK45 - A Free Standard Cell Library for SDDS-NCL Circuits”. Captured in: https://github.com/marlls1989/ascend-freepdk45, Jun 2020.

[SMC20c] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “Pulsar - A Flow to Support the Design of QDI Asynchronous Circuits”. Captured in: https://github.com/marlls1989/pulsar, Jun 2020.

[SWMC19] Sartori, M. L. L.; Wuerdig, R. N.; Moreira, M. T.; Calazans, N. L. V. “Pulsar: Constraining QDI Circuits Cycle Time Using Traditional EDA Tools”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2019, pp. 114–123.

[Syn22] Syntacore. “SCR1 Microcontroller Core”. Captured in: https://syntacore.com/page/products/processor-ip/scr1, Apr 2022.

[WA19] Waterman, A.; Asanović, K. “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA”, Technical Report 20191213, University of California, Berkeley, 2019, 238p.

[WAH21] Waterman, A.; Asanović, K.; Hauser, J. “The RISC-V Instruction Set Manual Volume II: Privileged Architecture”, Technical Report 20211203, University of California, Berkeley, 2021, 155p.

[XIL19] XILINX, Inc. “7 Series FPGAs Memory Resources - User Guide”, 2019, UG473, v1.14, https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources.

APPENDIX A – AN EXCERPT OF A RISC-V REFERENCE CARD

A.1 - Basic RISC-V Formats and an Embedded Instruction Set - RV32I

A.1.1 - Core Instruction Formats

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

A.1.2 - The RV32I Base Integer Instruction Set, Version 2.1

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

A.2 - Some Extensions for Simple Embedded Systems

A.2.1 - “Zicsr” - Control and Status Register (CSR) Instructions, Version 2.0

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7

Inst	Name	FMT	Opcode	funct3	Description (C)
`CSRRW`	CSR Read/Write	I	`1110011`	`0x1`	`rd = csr, csr = rs1`
`CSRRS`	CSR Read and Set Bits	I	`1110011`	`0x2`	`rd = csr, csr = csr | rs1`
`CSRRC`	CSR Read and Clear Bits	I	`1110011`	`0x3`	`rd = csr, csr = csr & rs1`
`CSRRWI`	CSR Read/Write Immediate	I	`1110011`	`0x5`	`rd = csr, csr = rs1(imm)`
`CSRRSI`	CSR Read and Set Bits Immediate	I	`1110011`	`0x6`	`rd = csr, csr = csr | rs1(imm)`
`CSRRCI`	CSR Read and Clear Bits Immediate	I	`1110011`	`0x7`	`rd = csr, csr = csr & rs1(imm)`

A.2.2 - “M” - Standard Extension for Integer Multiplication and Division, Version 2.0

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	`0110011`	`0x1`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	`0110011`	`0x2`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	`0110011`	`0x3`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`divu`	DIV (U)	R	`0110011`	`0x5`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	`0110011`	`0x7`	`0x01`	`rd = rs1 % rs2`

A.2.3 - “A” - Standard Extension for Atomic Instructions, Version 2.1

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
`lr.w`	Load Reserved	R	`0101111`	`0x2`	`0x02`	`rd = M[rs1], reserve M[rs1]`
`sc.w`	Store Conditional	R	`0101111`	`0x2`	`0x03`	`if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }`
`amoswap.w`	Atomic Swap	R	`0101111`	`0x2`	`0x01`	`rd = M[rs1]; swap(rd, rs2); M[rs1] = rd`
`amoadd.w`	Atomic ADD	R	`0101111`	`0x2`	`0x00`	`rd = M[rs1] + rs2; M[rs1] = rd`
`amoand.w`	Atomic AND	R	`0101111`	`0x2`	`0x0C`	`rd = M[rs1] & rs2; M[rs1] = rd`
`amoor.w`	Atomic OR	R	`0101111`	`0x2`	`0x0A`	`rd = M[rs1] | rs2; M[rs1] = rd`
`amoxor.w`	Atomix XOR	R	`0101111`	`0x2`	`0x04`	`rd = M[rs1] ^ rs2; M[rs1] = rd`
`amomax.w`	Atomic MAX	R	`0101111`	`0x2`	`0x14`	`rd = max(M[rs1], rs2); M[rs1] = rd`
`amomin.w`	Atomic MIN	R	`0101111`	`0x2`	`0x10`	`rd = min(M[rs1], rs2); M[rs1] = rd`

A.2.4 - “C” - Standard Extension for Compressed Instructions, Version 2.0

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	CR-type
funct3	imm	rd/rs1		imm		op	CI-type
funct3	imm			rs2		op	CSS-type
funct3	imm				rd’	op	CIW-type
funct3	imm		rs1’	imm	rd’	op	CL-type
funct3	imm		rd’/rs1’	imm	rs2’	op	CS-type
funct3	imm		rs1’	imm		op	CB-type
funct3	offset					op	CJ-type

Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd’, (4*imm)(rs1’)
c.sw	Store Word	CS	00	110	sw rs1’, (4*imm)(rs2’)
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs’, x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs’, x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd’, sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd’, rd’, imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd’, rd’, imm
c.andi	AND Imm	CB	01	100x10	andi rd’, rd’, imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd’, rd’, rs2’
c.or	OR	CS	01	10001110	or rd’, rd’, rs2’
c.xor	XOR	CS	01	10001101	xor rd’, rd’, rs2’
c.sub	SUB	CS	01	10001100	sub rd’, rd’, rs2’
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

A.3 - A Set of RISC-V Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning

`la rd, symbol`	`auipc rd, symbol[31:12]`
	`addi rd, rd, symbol[11:0]`
		Load address

`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]`
	`l{b|h|w|d} rd, symbol[11:0](rd)`
		Load global

`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`s{b|h|w|d} rd, symbol[11:0](rt)`
		Store global

`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{w|d} rd, symbol[11:0](rt)`
		Floating-point load global

`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{w|d} rd, symbol[11:0](rt)`
		Floating-point store global

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine

`call offset`	`auipc x1, offset[31:12]`
	`jalr x1, x1, offset[11:0]`
		Call far-away subroutine

`tail offset`	`auipc x6, offset[31:12]`
	`jalr x0, x6, offset[11:0]`
		Tail call far-away subroutine

`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`CSRR rd, csr`	`CSRRS rd, csr, x0`	Read CSR
`CSRW csr, rs`	`CSRRW x0, csr, rs`	Write CSR
`CSRS csr, rs`	`CSRRS x0, csr, rs`	Set bits in CSR
`CSRC csr, rs`	`CSRRC x0, csr, rs`	Clear bits in CSR
`CSRWI csr, uimm`	`CSRRWI x0, csr, uimm`	Write CSR, immediate
`CSRSI csr, uimm`	`CSRRSI x0, csr, uimm`	Set bits in CSR, immediate
`CSRCI csr, uimm`	`CSRRCI x0, csr, uimm`	Clear bits in CSR, immediate

A.4 - The RISC-V Register Naming Conventions

Register	ABI Name	Description	Saver
`x0`	`zero`	Zero constant	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0 / fp`	Saved / frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Fn args/return values	Caller
`x12-x17`	`a2-a7`	Fn args	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP args/return values	Caller
`f12-17`	`fa2-7`	FP args	Caller
`f18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller