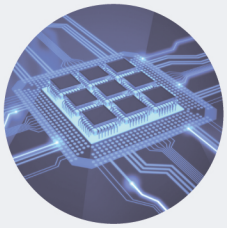
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Monitorização de Desempenho de Aplicações de Tempo Real em Plataformas RISC-V

NUNO FILIPE PESSOA SOARES
outubro de 2024

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Performance Monitoring of Real-Time Applications in RISC-V Platforms

Nuno Filipe Pessoa Soares

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: Tiago Diogo Ribeiro de Carvalho
Co-Supervisor: Luís Miguel Rosário da Silva Pinho

Evaluation Committee:
President:
Luís Lino Ferreira, Professor Coordenador, Instituto Superior de Engenharia do Porto

Members:
José Gabriel de Figueiredo Coutinho, Investigador Associado, Imperial College London, UK
Tiago Diogo Ribeiro de Carvalho, Investigador Associado, Instituto Superior de Engenharia do Porto

Porto, October 17, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, October 17, 2024

Dedicatory

This thesis is dedicated to all the people of contributed during this phase of my life and helped me going trough the daily hardships I faced.

To my parents and brother, whose unwavering love and support have been constants throughout my whole life, playing an essential role in me everything I have achieved today.

I would like to express my gratitude to my supervisors, Tiago Carvalho and Luís Miguel Pinho for their invaluable support and guidance during the writing of this thesis. Their insights and suggestions were essential in the development, structuring and finalization of the work delivered in this document.

To my friends who contributed in miscellaneous ways during my academic and personal life paths. A special thanks to Ricardo Rodrigues who accompanied me during this master's program; his support was essential in helping me complete this phase of my academic path.

To my girlfriend, Bia, whose love, patience and encouragement have been a constant source of inspiration and strength in my life. Your belief in me and in my ability to achieve my goals has been crucial throughout this journey. You have stood by my side in my best and worst moments, helping me stain sane and focused in my goals and dreams. You make my life's foundation sturdier and you are one of the reasons I can get up and smile each morning. Without you unwavering support, completing this phase of my life would have been much more difficult. Thank you for everything you have done and continue to do each day to make my life better.

Finally, I would like to leave a message to all those who have dreams and goals: stay strong, work hard and never give up. Life is challenging for everyone, and hardships are inevitable, but that shouldn't be an excuse. Keep moving forward - every step you take brings you closer to your goals, even if the future seems uncertain. Strive to achieve your goals so that in the end, you have no regrets. There is a line between your dreams and reality, and you are the one who is responsible for doing something to cross that line, and bring your dreams closer to reality.

Abstract

Real-time systems are an important field of research, specially when considering that these systems interact with the real-world and their tasks have direct impact in human lives and society. It is imperative that the traceability and performance of these systems can be ensured.

RISC-V is an open-source instruction set architecture, gaining interest from both industry and academia, specially in the context of real-time systems development due to its versatility and customization capabilities. The performance analysis of RISC-V systems emerges as a critical aspect, with the need to evaluate metrics, such as system efficiency and scalability, in order to deal with increasing workloads and tasks complexity.

There is a lack of support for performance analysis, specially in real-time operating systems for RISC-V architectures, usually not providing full support to the latest performance monitoring related specifications, or being tangled to a specific operating system.

This thesis provides a comprehensive study on performance analysis approaches on RISC-V systems, exploring tools and solutions for performance monitoring of applications, that provide access to the system performance counters. As a solution, this work proposes an API capable of retrieving performance metrics from the hardware performance counters, accessible via code instrumentation over a target application.

The API proposed works as a proof-of-concept for the development of more sophisticated tool capable of facilitating the retrieval and configuration of performance metrics from a RISC-V system. To access the API performance, some introductory tests have been performed to showcase that the API is capable of interacting and managing the hardware performance counters present on the system.

Additionally, the overhead generated from the API was also analysed briefly, showcasing that the API has limited impact on the overall system's performance when retrieving metrics from the hardware performance counters.

Keywords: RISC-V, Performance Analysis, Performance Counters, Hardware Performance Monitoring, Performance Monitoring API

Resumo

Sistemas de Tempo Real têm-se tornado uma importante área de investigação, visto que, estes interagem diariamente com o mundo, e possuem um impacto direto sobre vidas humanas e a sociedade, torna-se assim imperativo assegurar a rastreabilidade e desempenho do sistema.

RISC-V é uma arquitetura de conjunto de intruções versátil, customizável e de código fonte aberto. Devido às suas caracteristicas inerentes e a sua natureza *open-source*, tem ganhado muita atenção por parte da industria e da academia, tornando-se uma arquitetura ideal para desenvolver diferentes sistemas de tempo-real.

Com isto, a análise de desempenho de um sistema de tempo real, emerge como um aspeto crítico, onde é necessário avaliar diferente métricas de desempenho, tais como eficiência e escabilidade, com o intuito de garantir que o sistema é capaz de suportar os aumentos de cargas de trabalho e de complexidade das tarefas a executar.

Esta tese proporciona um estudo compreensivo sobre abordagens utilizadas para realizar análise de desempenho em sistemas RISC-V. Diferentes ferramentas e soluções que permitem acesso aos *hardware performance counters* do sistema são exploradas. Este trabalho propõe uma solução de uma API capaz de permitir a recolha de métricas de performance diretamente e configurar os *hardware performance counters*, através de instruções programáticas colocadas diretamente no código fonte da aplicação em questão.

A API proposta funciona como uma prova de conceito para o desenvolvimento de uma ferramenta mais sofisticada que é capaz de recolher e configurar diferentes métricas de desempenho de um sistema RISC-V. Para analisar o desempenho da API, alguns testes introdutórios foram realizados para demonstrar as capacidades da API de interagir e manipular os *hardware performance counters* presentes num sistema RISC-V.

Adicionalmente, o overhead gerado pela utilização da API foi analisado brevemente, o que demonstrou que a API tem um impacto limitado no desempenho global do sistema, quando se recorre à recolha de métricas diretamente dos *hardware performance counters*.

Contents

List of Figures

List of Tables

List of Source Code

List of Abbreviations

ABI	**A**pplication **B**inary **I**nterface
AEE	**A**plication **E**xecution **E**nvironment
API	**A**pplication **P**rogramming **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BIOS	**B**asic **I**nput/**O**utput **S**ystem
CNN	**C**onventional **N**eural **N**etworks
CPU	**C**entral **P**rocessing **U**nit
CSR	**C**ontrol and **S**tatus **R**egister
DBT	**D**ynamic **B**inary **T**ranslation
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GPU	**G**raphics **P**rocessing **U**nit
HAL	**H**ardware **A**bstraction **L**ayer
HBI	**H**ypervisor **B**inary **I**nterface
HDL	**H**ardware **D**escription **L**anguage
HEE	**H**ypervisor **E**xecution **E**nvironment
HPM	**H**ardware **P**erformance **M**onitor
I/O	**I**nput/**O**utput
IoT	**I**nternet **o**f **T**hings
ISA	**I**nstruction **S**et **A**rchitecture
MLP	**M**ulti-**L**ayer **P**erceptron
OOP	**O**bject-**O**riented **P**rogramming
OS	**O**perating **S**ystem
PAPI	**P**erformance **A**pplication **P**rogramming **I**nterface
PMU	**P**erformance **M**onitoring **U**nit
RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead **O**nly **M**emory
RTL	**R**egister-**T**ransfer **L**evel
RTOS	**R**eal **T**ime **O**perating **S**ystem
SBI	**S**upervisor **B**inary **I**nterface
SIG	**S**pecial **I**nterest **G**roup
SEE	**S**upervisor **E**xecution **E**nvironment
SoC	**S**ystem **o**n a **C**hip
TLB	**T**ranslation **L**ookaside **B**uffer
VMM	**V**irtual **M**achine **M**onitor

Chapter 1

Introduction

Real-time systems are an important field of research [1], and one of the most relevant topics for cyber-physical systems. These systems have to interact with the real-world and perform tasks that should execute within time limits (deadlines), with specified periods of execution, and considering different execution priorities.

Program traceability is an imperative feature for real-time applications [2] in all phases of its lifetime, allowing the analysis, debugging and optimization of these applications. Some examples of tracing measurements are execution time, energy usage, and system-specific performance counters metrics (such as cache hits and misses, instructions executed, branch predictions and exceptions) [2, 3].

RISC-V is an instruction set architecture designed as an open standard with a Reduced Instruction Set [4]. It is an architecture that has gained a lot of interest from both industry and academics, allowing designers to customize and extend the base architecture and build their own processor, shaped to their operating systems and target end applications [5]. In particular, RISC-V has been object of interest for real-time systems due to its modularity, simplicity and flexibility. In fact, as ISA extensions can be added or removed as see fit. It is possible to implement or choose the extensions that better fit the requirements of the system, leading to a better overall system performance. When we are dealing with real-time systems where it is crucial to have a system capable of performing its tasks correctly and efficiently, the capability of tailoring the instructions to the requirements comes in handy to boost the systems performance.

1.1 Problem statement and Motivation

Performance analysis in RISC-V is, then, an important aspect, especially when considering the execution of RISC-V platforms running a real-time operating system (e.g., FreeRTOS [6] or Zephyr [7]). Understanding the traceability of RISC-V platforms, how tracing mechanisms can be designed and how to use them is still a popular research topic.

RISC-V is part of a growing open-source community; several specifications have been ratified over the years, and many are still being developed. Even when a specification exists and has already been ratified, currently available solutions may not support the entire ISA specification, as it takes time for specific implementations to catch up with ratified specifications. This scenario is even harsher when the specifications are still under development.

When it comes to performance analysis there are some extensions being drafted to address system performance monitoring, such as the Counters extension, that deals with the standardization of the hardware performance counters [8] on a RISC-V system, and Zicsr,

already ratified, which deals with access to the Control and Status Registers (CSRs), such as performance counters [9, 10].

Furthermore, multiple work groups were created focusing in different topics related to RISC-V platforms. One such group is the RISC-V Performance Analysis SIG (Special Interest Group), which is responsible for driving the strategy and coordinating the development of end-to-end solutions for monitoring, reporting, while also analysing the performance of software executing on RISC-V systems [11].

Currently, the community is still focusing with standardization when it comes to performance analysis on RISC-V systems. Most of the existing solutions for performance monitoring, such as Perf [12], only provide support for Linux systems, which despite being capable of having real-time capabilities, is on its core a general purpose operating systems.

With Real-time systems becoming a target to RISC-V development, it is crucial to ensure the availability of performance monitoring capabilities, in order to ensure that the systems has the necessary capabilities to perform as intended inside its time-constraints.

1.2 Objectives

There are two main goals for this work. First, it aims to provide a comparative study between the research approaches and methodologies being developed and used in the context of performance analysis on systems running a Real-Time Operating System on a RISC-V platform. The feasibility of retrieving different performance metrics directly from the hardware performance counters of the system for the development of an API will be analysed. Second, the development of an API capable of providing access to the performance counters in a convenient, programmatic fashion. The solution should focus on the use of standardized ISA, considering the minimal number of extensions as possible. Additionally, the potential of being operating system independent will be analysed, to provide a more portable and versatile solution.

The survey should detail the native RISC-V features for tracing, as well as available tools, frameworks and libraries specially designed for tracing applications running in RISC-V architectures. The tracing tool shall allow the specification of code tracepoints, allowing the tracing of specific measurements (such as execution time and CPU cycles), in specific code regions.

1.3 Research Questions

To better define and understand the research topic, some research questions were defined to be answered in this thesis, namely:

- **RQ1 -** Is it possible to extract performance metrics of applications running on RISC-V microprocessor? What about real-time applications?
- **RQ2 -** Is it possible to configure the system to measure different performance metrics based on chosen events, on a RISC-V microprocessor?
- **RQ3 -** Can an approach extract and configure performance metrics of real-time applications running on RISC-V microprocessor, considering the execution in a RTOS, while providing a low, close to negligible, overhead?

- **RQ4** - Can the approach be developed considering a minimum of standardized extensions from the RISC-V ISA specifications?
- **RQ5** - Can the solution be provided as an API, easily deployable in different RTOS and RISC-V systems?

1.4 Contributions

This thesis contributes to the study and analysis of the existing work related with performance monitoring in RISC-V Systems, with a focus on the ISA Specifications for both Unprivileged and Privileged, specially related with hardware performance counters. It also delves into the research and development of an API designed to answer the proposed research questions.

This thesis contains an comprehensive study on the RISC-V ISA Specifications, focusing on the Performance Monitoring features in RISC-V System, and discussing the rise of RISC-V popularity in both industry and academia, in recent years. Additionally, it explores the benefits of using emulation during the development of a solution, the role of Real-Time Operating Systems, metrics used for Performance Analysis, and analysed various approaches available to be used for performance monitoring across different platforms.

One of the primary aim is to answer the research questions outlined in the previous subsection, by providing a solution capable of facilitating the retrieval of performance metrics, and configuration of the RISC-V System to allow the desired metrics to be easily retrieved.

The proposed approach in this thesis is developed as an API that is designed to be easily ported to different platforms and environments, giving specially consideration in reducing the implementation specific details from both operating systems and the CPU architecture support. By providing an abstraction to these dependencies, it enhances the API portability to other systems.

The API offers different configuration options and is designed with extensibility in mind, special considering custom expansions that may exist within the system for performance monitoring, enhancing the API adaptability and versatility.

Furthermore, the thesis highlights several challenges, limitations and potential directions for future work. It also includes an analysis and demonstration on the API's usage, showcasing some examples on the API usability and the advantages it provides to users.

1.5 Thesis Organization

The current study is distributed by seven chapters, each dealing with an essential topic of this thesis. The current chapter, Chapter 1, provides a small introduction to the research topic, as well as the definition of the goals and objectives of this research. Chapter 2 provides a brief literature review about the topic, unraveling some state of the art technologies and solutions that shall be the backbone of the provided solution. In Chapter 3, an overview on performance monitoring in RISC-V systems, showcasing the RISC-V PMU features and how it can be accessed in according to both unprivileged and privileged ISA specifications. Later on, in Chapter 4 the developed RISC-V Performance Counters API is discussed, providing an overview on its architecture, core functionalities, configuration options and some implementation details. Chapter 5 explores the system setup during the API development, detailing

the software and hardware environment used, along with an explanation regarding the API Porting process and how it can be achieved, followed by the coverage of some lessons learnt throughout the API development. In Chapter 6 the API is analysed and demonstrated, showcasing how the API can be used along with some use case examples. This chapter also includes a brief evaluation on the API generated overhead, memory requirements during execution and some challenges encountered while testing the API. The chapter concludes with an overview of how the API and the topics discussed in the thesis answered the research questions. Finally, Chapter 7 reflects about the entire thesis, discussing the API limitations and potential areas for future improvements.

Chapter 2

State of the Art

This chapter contains an extensive study of the current state of the art, focusing on the RISC-V ISA and its rise in recent years. It also explores the role of emulation in solution development, highlighting its benefits and tools that can be utilized. Additionally, the chapter provides an overview of Real-Time Operating Systems, detailing various paradigms, characteristics, and their support for RISC-V.

Finally, further insights into performance analysis, emphasizing key metrics commonly used and hardware performance monitoring. It concludes with an overview of available approaches for performance measurements in the community and a comparison amongst them.

2.1 RISC-V ISA

RISC-V is a new open-source Instruction-Set Architecture (ISA) that "builds and improved on the original Reduced Instruction Set Computer" and aims to be a "clean, simple, and modular ISA that is well suited to low-power embedded systems and high-performance computers alike", which was originally designed to support computer research and education at Berkeley University [4].

In [4] Andrew Waterman describes the reasoning behind the creation of RISC-V ISA. Almost all of the popular commercial ISAs are proprietary and thus their vendors that "have a lucrative business selling implementations of theirs ISAs, be it in the form of IP cores or silicon, and so they do not welcome freely available implementations that might erode their profits". While this on itself doesn't inhibit "all forms of academic computer architecture research", it "erects a barrier to the commercialization of successful research ideas" [4].

Other than that, popular commercial ISAs have a "massive complexity" and are "quite difficult to implement in hardware" and there is "little incentive to create simpler subsets" [4]. If a complete implementation of an ISA is not available in hardware, then "unmodified software cannot run" which undermines the need to use an existing ISA [4]. While "some degree of complexity is necessary" and even beneficial, "these instructions sets tend not be complicated for sound technical reasons" and "simpler instructions sets can lead to similarly performant systems" [4].

While assessing the feasibility of using an existing ISA, Waterman analysed multiple ISAs on Chapter 2 in [4]. During this assessment he looked into some key components that are "essential for a modern general-purpose ISA", among them were the licensing and pricing, 64-bit addressing support, floating-point support following IEEE 754-2008 and virtualization capabilities, the result can be visualized at Table 2.1.

Table 2.1: ISAs comparison - Retrieved from [4]

	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		✓				✓	
64-bit Addresses	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

As it can be seen, all of the ISAs analysed did not support at least two important technical features and the one that comes closest is ARMv8, which is a proprietary standard. The only two free and open-source architecture sets, "SPARC and OpenRISC, lack several crucial architectural features". Almost all the ISAs have some properties that substantially increase implementation complexity, especially for high-performance implementations [4].

2.1.1 Key principles and designs

RISC-V was developed on the "legacy of the RISC-I, RISC-II, SOAR, and SPUR projects" and because it was "the fifth major RISC ISA design effort at UC Berkeley" it was named RISC-V, conveniently, the roman number 'V' serves as an acronymic pun for "Vector", and goes into accord with the intention of supporting "research in data-parallel architectures" [4].

Goals

The core goal that guided the definition of the whole RISC-V instruction set "was to make an ISA **suitable for nearly any computing device"** [4]. To achieve this, there are two principles that need to be taken into consideration:

- **"RISC-V should not be over-architected for any microarchitectural pattern, implementation fabric or deployment target"** [4]
  This is crucial to ensure that a particular feature, only demanded by a specific domain, will be provided with optimizations at the costs of expenses on other features. As "different application domains demand different microarchitectural styles, and such features complicate some of those implementations", the opposite is also true, because "not all domains demand all of the features of a rich ISA" by including them anyway it would only "increase cost and reduce efficiency"[4].
- **RISC-V "must be open and free to implement"** [4]
  Open-source development provides multiple benefits but "the most important one is the potential abundance of processor implementation" [4]. By eliminating vendor lock-in, users become empowered and capable of choosing implementations from various vendors without sacrificing compatibility. Furthermore, it keeps promoting a diverse and competitive "free-market competition between open and proprietary implementations" that should "spur microarchitectural innovation" [4]. Another important point is that "the barrier to academic-industrial interactions is lowered" because both worlds will "share common standards and implementations". Finally, this also addresses another important topic in security concerns as "entities that do not trust

certain implementations of the standard perhaps due to fears of industrial espionage, or of meddlesome governments—can instead devise their own" [4].

With the intention of making RISC-V a widely available and used instruction set, some specific technical goals were defined. As presented on [4], the following are highlighted:

- *Small base ISA with separated optional extensions.*
- *32-bit and 64-bit address spaces support.*
- *Custom ISA extensions.*
- *Variable-length instruction set extensions support.*
- *Efficient hardware support for modern standard, such as IEEE-754 2008 floating point standard and C11 anc C++11 programming languages.*
- *Orthogonal user ISA and privileged architecture.*

Base ISAs

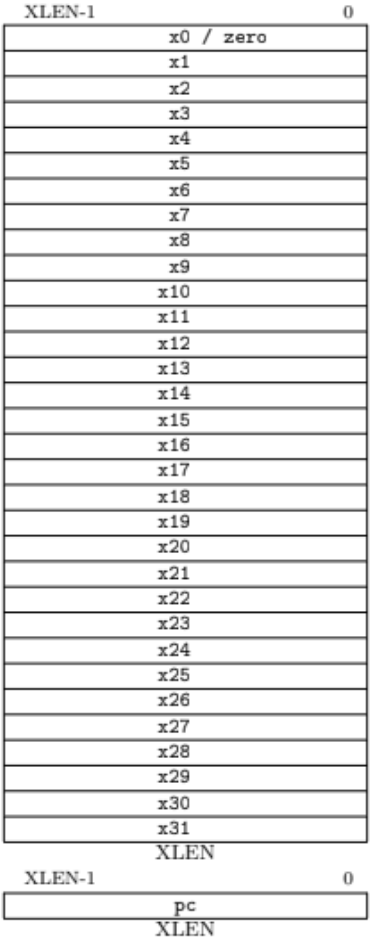


Figure 2.1: RISC-V Base Unprivileged integer register state - Retrieved from [9]

The RISC-V ISA defines a base implementation which can be increased with additional implementations, reducing the core of the specification to the strictly necessary operations

that are required to serve most purposes of modern computing. With this segregation, which allows for further definitions of ISA extensions, there will be an increase in performance as instructions can be implemented for each specific domains without bringing a negative impacts to other domain applications.

This flexibility provided by the usage of a lean base implementation and the option to introduce ISA extensions to the base ISA, helps making RISC-V "an ISA suitable for resource-constrained low-end implementations and high-performance ones alike" [4].

The RISC-V base ISAs is "simple and straightforward to implement, yet complete enough to support a modern software stack", it is segregated in three base specifications, being them RV32I, RV32E and RV64I, where "RV32I and RV64I differ primarily in the width of the registers and the size of the memory address space" and "RV32E is a variant of RV32I with fewer registers, meant for deeply embedded systems where every transistor counts" [4].

Due to a fast decline in memory prices, the required memory in a system increased rapidly, a factor that was not taken into consideration when designing the PDP-11, lead to a need to evolve the architecture to support more memory space address [13]. In [13], Gordon Bell and Bill Strecker, two of the designers of the PDP-11, mentioned that "There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management".

Learning from their predecessors, in [4] a 128-bit extension, RV128I, is also defined. This extension follows the same approach as the one defined for 64 bit support, RV64I, whereas in this case, the width of the registers was doubled, from the original 64-bit sized specification, with new operations being added to operate using the full 128 bits registers. In Figure 2.1, its possible to observe the register state for the integer base implementation, where XLEN corresponds to the number of bits.

Its important to mention that in [4], Waterman expects that the 64 bits of address space is ample enough for decades to come; at the time of writing is paper, in 2016 the fastest supercomputer was Tianhe-2, it had 1.3 PiB of memory, which takes around 51 bits to be completely byte-address. Currently, in early 2024, the fastest supercomputer is Frontier [14] with 9.2PiB of addressable memory, this would take around 54 bits to completely address. This represents an increase in 3 bit in less than a decade, every day new technological advancements are being created, such an example is DeepSouth, a supercomputer that aims to mimic the human brain biological processes, "using hardware to efficiently emulate large networks of spiking neurons at 228 trillions synaptic operations per second - rivalling the estimated rate of operations in the human brain" [15] as Professor van Schaik said, “This platform will progress our understanding of the brain and develop brain-scale computing applications in diverse fields including sensing, biomedical, robotics, space, and large-scale AI applications.” [16].

As time goes on and the complexity of tasks increase, the computational resources required to achieve these challenges are becoming increasingly stringent and demanding, which will ultimately lead to a greater need for memory resources, which increments the necessary number of bits to address all the memory available on the system. Despite being far away from meeting the limit of 16384 PiB of memory, which is the maximum capacity of addressable memory available for a 64-bits, a 128-bit extension makes RISC-V a future proof architecture.

ISA Standard Extensions

To provide the ISA with enough flexibility to handle multiple application and system domains, RISC-V defines four standard extensions at[4]:

- **M** for Integer Multiplication and Division operations.
- **A** for Atomic Memory Operations, which is particularly important for multiprocessors synchronization.
- **F** for Single-Precision Floating-Point operations.
- **D** for Double-Precision Floating-Point operations.

Despite all of this, "RISC-V like its RISC predecessors, is not a particularly densely encoded instruction set architecture" and its "fixed-width encoding simplifies pipelined microarchitectures helping to make the ISA suitable for reasearch and educational purposes". However, this same benefit can also bring an increase to the source code size, that may lead to a lot of complications, specially in domains where code size is a major concern due to scarce and limited resources, such as embedded systems [4].

To confront this problem, the RVC extension, also known as "RISC-V Compressed" was designed in [4] and as it is later mentioned, "C" extension as mentioned at [9]. This extension "increases RISC-V applicability to all of these domains by improving code density by 25%-30%, resulting in smaller programs than all of the commercially popular 64-bit instruction sets". This is achieved by introducing "dual-length instructions to the base ISA", which "reduces static code size and dynamic fetch traffic of RISC-V programs by encoding the most frequent instructions in a denser format"[4].

Later in 2021, at [9] more new extensions were added and are still currently being defined. Regarding already ratified versions we have the following extra:

- **Q** for Quad-Precision Floating-Point operations.
- **C** for Compressed Instructions.
- **Zicsr** for Control and Status Register (CSR) Instructions.
- **Zifencei** for Instruction-Fetch Fence.

Aside from these, there are many more being defined that are currently only on its drafted format, awaiting approval to be ratified by the RISC-V Foundation. Some examples are:

- **Counters** for Hardware Performance Counters, base counters and timers.
- **L** for Decimal Floating-Point operations.
- **B** for Bit Manipulation.
- **V** for Vector Operations.

Privileged Architecture

All of the previously mentioned base ISAs and Standard Extensions, reflect the user-level ISA definition. In order to meet the goal of orthogonalize the RISC-V Architecture, the definition of a privileged architecture is missing. With a clear separation of unprivileged (user-level) and privileged architecture, it is possible to:

- **Allow the user ISA to be shared across a wide variety of systems, each one with different requirements**. A simple example of this is an embedded real-time system with a trusted code base, this system will hold different features than an hypervised server with I/O virtualization. By separating both the Privileged and User-level ISA, its possible to use the same user ISA on both, and only change the implementations regarding the privileged specification, achieving a reduction on development costs [4].
- **Facilitates experimentation in privileged architectures.** This allows for researchers to experiment with different implementations of privileged architectures or operations, such as, memory translation and/ore protection schemes without the need to rewrite application code, which will follow the user-level ISA [4].
- **Simplifies full virtualization implementation.** When privileged features are exposed to unprivileged software, the hardware-assisted virtualization will have additional complexity, making classical virtualization impossible. Such an example "is the x86 FLAGS register, which holds both user-visible condition codes and privileged fields, such as the interrupt-enable flag. User-mode writes to the privileged portion of the FLAGS register are ignored. This silent failure foils classical virtualization, in which the guest operating system runs in user mode: guest attempts to disable interrupts, for example, cannot be intercepted and emulated by the host OS. VMware heroically worked around this limitation with an intricate dynamic binary translation scheme" [4]. By clearly segregating unprivileged and privileged specifications, its possible to perform the virtualization of each specification without caring for the impact on the other specification.

An important point to highlight from the chosen design choices is that with solid definition of a user-level and privileged architecture designs, it is currently possible to change the privileged ISA completely without affecting the user-level ISA [9], this makes it possible to completely replace it without the need to make changes on application code.

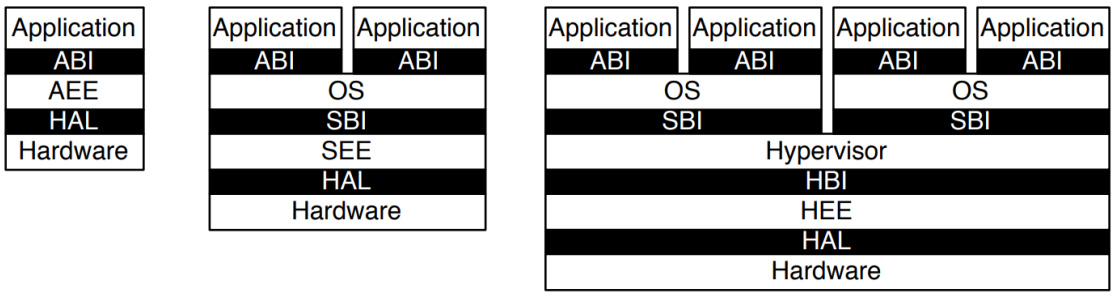


Figure 2.2: Different implementations stacks supporting various forms of privileged architectures on Waterman 2016 Design - Retrieved from [4]

Usually, "in a conventional general-purpose system, applications make requests of the operating system using a system-call convention in an application binary interface (ABI)" [4], the application code does not need to know the implementation details of the "underlying layers that implement the system call" it is sufficient to only know the interface used for the communication. By making this abstraction it is possible to improve the system modularity, other than that, it is even possible that a user application does not even know the "identity of its application execution environment (AEE)", which most of the time is an operating system, but it can easily be an emulator implementing the same ABI [4].

An interesting note is that usually, in most privileged environments the OS directly perform actions such as arming timers and routing interrupts by directly interacting with the hardware [4]. By adding another abstraction layer between the OS and the underlying hardware, it is possible to improve OS implementation flexibility and portability. In [4] and other specifications for the privileged ISA, such as [9], the supervisor execution environment (SEE) is abstracted, allowing the OS to communicate through a supervisor binary interface (SBI).

By making use of a "single SBI across all SEE implementations allows a single OS binary image to run on any SEE", this "SEE can be a simple boot loader and BIOS-style IO system in a low-end hardware platform or a hypervisor-provided virtual machine in a high-end server" or even a "thin translation layer over a host operating system in a architecture simulation environment" [9].

The analogy continues to the hypervisor, which interacts with the hypervisor execution environment (HEE) via a hypervisor binary interface (HBI). This design simplifies the implementation of recursive virtualization."[4].

However, this abstraction can't continue indefinitely, in [4] defines the "For native RISC-V hardware systems the lowest-level execution environment interacts with the hardware via a hardware abstraction layer (HAL). The HAL isolates the execution environment from the implementation details of the hardware platform, such as the location of control registers in the address map. Hiding platform-specific details improves the reusability of execution environment software". Figure 2.2 shows the initially defined different implementations examples.

Later on this additional abstraction from the Hardware seems to have been dropped from the ISA, as per the [9] Privileged ISA review, there is no mention to this HAL, and the examples shown are present in Figure 2.3, and as it can be seen, the HAL layer has been removed.

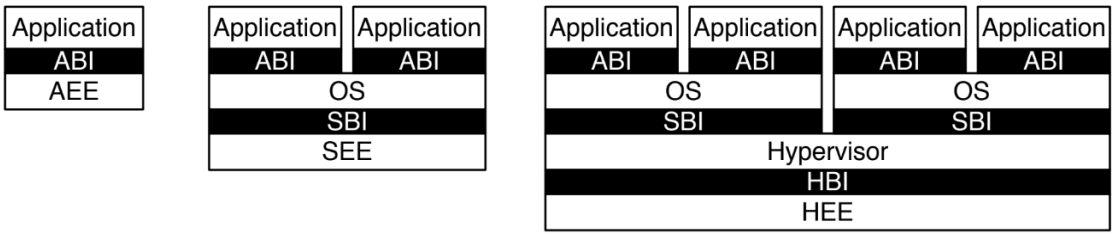


Figure 2.3: Different implementations stacks supporting various forms of privileged architectures on RISC-V Privileged Specification - Retrieved from [9]

Initially, at [4] four levels of privilege were defined, being them:

- **U** User, the least privileged, in which application code usually executes.
- **S** Supervisor, providing basic exception processing and virtual memory support, meant to execute OS level code.
- **H** Hypervisor, placeholder for a privileged mode designed to host a virtual machine monitor (VMM).
- **M** Machine, with unfettered access to all hardware features.

Usually, providing only the M-mode would be sufficient for most embedded system, although it would provided some isolation for system and application software if a U-mode is also be provided together. A system with M/S/U modes is capable of supporting a traditional Unix-like OS, while adding the H-mode will provide additional support for hardware-assisted hypervisors [4]. Table 2.2 shows the available levels in [4].

Table 2.2: RISC-V Privileged levels on Waterman 2016 Design - Retrieved from [4]

Level	Description
U	User/Application
S	Supervisor
H	Hypervisor
M	Machine/Trusted

Number of Levels	Supported Modes
1	M
2	M, U
3	M, S, U
4	M, H, S, U

In its latest version of the ISA reviewed privileged manual [9], the only modes defined are U, S and M, and the old H is currently being held as a reserved mode for later implementations. Table 2.3 shows the available levels defined in [9].

Table 2.3: RISC-V Privileged levels on RISC-V Privileged Specification - Retrieved from [9]

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

Additionally, a D-mode, standing for Debug mode, was also included as a way "support off-chip debugging and/or manufacturing tests"[9]. A separated Debug specification proposal describes the operation of a RISC-V system in debug mode, but at the moment of writing is only on its drafted version [1].

Key design principles

By adhering to these principles, RISC-V enables the development of processors that can be tailored to specific applications and use cases, providing a high degree of customization and optimization [17] [18].

Recapping, the key design principles can be summarized on the following attributes:

- **Modularity:** Due to the addition of its extension ISAs, RISC-V has a modular structure. It is possible to create standard extensions for specific application domains, allowing designers to tailor the architecture to meet the requirements of a wide range of applications and systems domains.

[1] RISC-V Debug Specification GitHub Repository

- **Simplicity:** RISC-V follows the RISC design philosophy, favoring a reduced set of simple and standardized instructions and reducing the complexity of the hardware, letting software take the burden, this leads to a simpler to implement and faster microprocessor.
- **Scalability:** The architecture is designed to be scalable, supporting both 32-bit and 64-bit instruction set widths, which makes it suitable for a variety of computing devices, from embedded systems to high-performance servers on the cloud. It even defines the 128-bit instruction support as future proof.
- **Extensibility:** RISC-V allows for the addition of custom instructions, enabling the tailoring of the architecture on specific tasks, improving the overall performance on specific applications. In fact, only taking into consideration the Base ISAs for 32-bits, it only uses around 1/4 of the available operation codes space size, reserving 3/4 of the total operation codes to be added by extensions and custom instructions.
- **Open Ecosystem:** Like previously mentioned, RISC-V was created with the intention of being an open and free ISA. Nowadays, the RISC-V Foundation, now part of the Linux Foundation [19], oversees the development and standardization of RISC-V. The open nature of RISC-V has encouraged a diverse ecosystem of hardware and software developers, which foster innovation and collaboration.

2.1.2 Rise of RISC-V

Since its conception and release to the public, RISC-V has been rapidly gaining a foothold in the technology industry and is already widespread and adopted across diverse sectors of industry and education/academia. This demonstrates the potential and appeal that this technology has for multiple parties across all domains, being them either a small and independent developer, researcher or student or a big or small corporations.

Internet of Things (IoT)

When it comes to the Internet of Things (IoT), RISC-V has gained a significant traction, its open-source nature allows for an efficient customization, making RISC-V a well-suited instruction set for the heterogeneous requirements that are demanded by IoT devices.

Due to its flexibility, extensibility and scalability, RISC-V ISA has fomented new possibilities in the IoT field. This ISA enables for an optimized performance in resource-constrained environments, due to its multiple extensions targeting scarce resource devices, and as ISA instructions and extensions can be added or removed to better fit the requirements of the IoT applications, this tailoring capabilities makes it easier to design IoT hardware and software solutions [20].

Furthermore, it is even possible to make use of the compressed ISA on even stricter devices, so that the usage of the resources present in the system are optimized to its maximum. Other than that, it is also possible to virtualize some hardware components required by the ISA due to its multi-leveled privilege support design. On a native RISC-V hardware system, "the lowest-level execution environment interacts with the hardware via a hardware abstraction layer"[4]. By using this abstraction layer, specific hardware component can be virtualized, and applications can make use of it together with the real physical hardware.

Edge Computing

Regarding edge computing, RISC-V has demonstrated to be a frontrunner due to its flexibility and scalability capabilities that make it an ideal choice for edge devices that demand both efficiency and adaptability. Like for the IoT field, it is possible to make use of the multiple defined ISA extensions, or even define a new specific extension, that can provide a more efficient approach for specific operations in the edge computation domain.

Edge computing applications, which range from smart cameras to industrial sensors, benefit from this RISC-V's ability to handle specific tasks with tailored instructions, enhancing overall system performance. With the coming of AI, edge computing is been seen as the key to unlock the capabilities of this technology. Some studies are implementing Neural Networks on edge devices, and have been verified to be capable of achieving less energy dissipation, higher security and lower latency when comparing to currently available implementations [21].

On [22], the RISC-V Vector extension is used on an existing RISC-V Core to augment a vector unit for accelerating Machine Learning inference on edge devices, the benchmarking performed has demonstrated that the vector processor has gained a speedup on the overall process of image classification of about 40 times on the usage of Multilayer Perceptron (MLP) and 20 time for Conventional Neural Networks (CNN) and energy savings up to 33 times on MLP and 16 times on CNN when compared to exiting solutions. This demonstrates the enormous potential and capabilities that RISC-V brings to these fields.

Datacenters

Data centers can be described as the foundation of modern computing. RISC-V has also entered this field, and it is becoming increasingly possible to see positive feedback in this domain. The ISA modular design architecture that allows for customization, can be exploited to meet the specific demands of data center workloads. Some of the more notable implementations in this field, include processors specifically designed for high-performance computing tasks.

Ventana Micro Systems, has introduced such a high performance RISC-V processor that competes with privately owned ISAs [23, 24] contributing to the growing ecosystem of RISC-V solutions for data-intensive applications.

Proprietary Domains

Despite being designed to be open and free to be used, everyone is allowed to defined their own ISA. Obviously, this is applied to both proprietary and community domains. Multiple success stories are rising across all sectors of the industry, with companies showcasing the effectiveness of RISC-V in meeting the unique and various needs across all domains.

For instance, SiFive, that is leading RISC-V processor IP provider, has played a pivotal role in driving the adoption of RISC-V across various applications [25]. Western Digital has incorporated RISC-V in its storage controllers, demonstrating the architecture's relevance in critical components of data storage systems [26].

Moreover, in [27], Ruobing Han et al., were capable of executing NVIDIA CUDA Source code on a RISC-V GPU architecture device, "with multiple features, including multi-thread, multi-block, atomic and synchronization", this showcases the architecture's versatility, even

in traditionally complex and proprietary domains. All of these success cases serve to empathize the impact that this technology has been achieving in recent years and the foothold being gained across almost all industries, competing with some older and well established proprietary ISAs.

Space Industry

RISC-V has also been introduced to Space, providing an alternative to the existing proprietary solutions, allowing for a new generation of architecture frames for on-board embedded systems. The modularity and open-source nature of this ISA helps reducing the resources, time, and complexity required for the development of custom SoC design.

Additionally, as detailed information about open-source IP cores can be found by inspection and ad-hoc improvements or modifications for security are much easier to be performed, it is possible to avoid the need to design everything from scratch and thus ultimately increase reusability of components in this field. These capabilities make RISC-V an excellent alternative that may probe even more reliable than the already existing solutions [28].

2.2 Emulation

Emulation has had a big impact in the development of systems and applications, specially when it comes to microcontrollers. With the proliferation of embedded systems, where resources are usually scarce, and every transistors counts, it is crucial to ensure the system is behaviouring as intended.

2.2.1 Overview and advantages

By performing early tests into the defined architecture, it is possible to accelerate the development process. The earliest the model of a specific hardware or software design can be validated the faster its development process will become and the more confidence will it gain.

In [29], hardware emulation was used to map a model of the design of a K5™AMD microprocessor, "onto hardware resources and execute it at high speeds". By emulating the microprocessor, the debugging process of the chip, becomes facilitated, as it provides a more "friendlier and productive environment" compared to debugging silicon, allowing for hardware failure isolation and bug detection and fixes [29].

Across the years, multiple development approaches were applied to maximize performance and velocity, and reduce development costs. The chosen development approach is specially important when the topic is embedded systems, were developers are required to develop a low cost, high performance system, while reducing the "time-to-market" to a minimum. In [30] some development approaches were discussed:

- **Hardware First**: The first step is to complete the design and specification of the system functionalities and requirements, after that it comes the development of the hardware and software, and finally the complete system will be tested and assessed to ensure that it's working as intended. As this is a sequential approach, we can only verify the correctness of the system on a later phase, if an irrecoverable error is detected then the whole system may be discarded.

- **Hardware/Software Co-Design**: This approach is based on formal specification of a system design, by using mathematics and computer science basis, it is possible to describe the system's behaviour and derive multiple implementations in order to perform a set of tasks. By iteratively analysing multiple alternatives, it is possible to merge the best of each approach in order to attain a better final solution. Note that this iterative process can be done in parallel for both hardware and software specifications. Finally, comes the hardware and software integration and testing, if the system complies with the specifications then the development ends, if not the other alternatives will be analysed and merged. As this approach has an algorithmic nature, it allows for an early verification of the systems design, however, the most complex the algorithm, the harder will it be to perform the development, which can become a limiting factor, specially if the system is complex.
- **Emulation Based Methodology**: This approach was proposed in [30], and it starts by first assessing and ranking components that can be used for the system design, after this, the chosen components, both hardware and software, will be emulated together, if the components meet the system requirements then the development ends, if not the next ranked components can be emulated and assessed, and finally the whole system will be physically integrated and tested. By emulating the system components it is possible to rapidly verify if the system has serious problems, such as real-time invalid behaviours.

Other than allowing for a faster development, it also has a relevant impact for teaching microprocessors, in [31] a simple and inexpensive development environment, that emulated microprocessor bus activities, was created. It enabled students to retain an high interest during the project development and the conceptual basis of the course, as they were capable of easily interacting with the acquired knowledge by developing different projects on the emulated environment.

In [32] emulators were used to perform a performance analysis on Real-Time Operating System (RTOS). The emulation gave the author a detailed analysis of the internal behaviour of the RTOS on an early phase, and as it closely reflects the target system it was also possible to assess its real-time capabilities, and the influence of different caches on the performance. If this assessment was not emulated, it would be necessary to obtain multiple implementation with and without the different caches to assess their impact on system performance, whereas by using emulation it can be easily turned on and off by specifying the cache used prior to the start of the emulation.

When a system is being emulated both software and hardware components are replicated in another system, this provides access to a systems ecosystem in another device. This is specially relevant when a specific hardware component is not available or not yet developed. It provides a safer debugging environment and allows for early testing phases when components are not yet physically available.

2.2.2 Tools and Languages comparison

When developing for devices based on the RISC-V architecture, it is especially important to use emulation for development and testing. Due to being a relatively new ISA, there is a low availability of hardware components. Furthermore, as it is being constantly modified and improved, the hardware development is always behind the newest ISA standard specifications, making it difficult and even sometimes impossible to use some of the most recent ISA

functionalities. Due to this, by making usage of emulation it is possible to use the latest ISA (if already implemented on an emulator) before an hardware implementation has been developed, optimizing the development of solutions for this ISA, as it enables a faster system development.

Other than this, it is important to take into consideration the necessities of each project, and use the adequate level of detail for the emulator implementation. When designing an emulator, different techniques can be used upon implementation, it can simply remain as a pure CPU simulation, or it can be a full-system emulation [33].

Existing RISC-V emulators can be fit under two categories, detailed Register-Transfer Level (RTL), which is a design abstraction of the system that can model a synchronous digital circuit signals between hardware registers and its correspondent logical operations, or Dynamic Binary Translation (DBT), which translate the target ISA instructions, to the current system architecture on-the-fly [33, 34].

As RTL provides a more detailed emulation of the hardware, it is usually slower, but has less errors when performing system analysis, when compared to DBT, which favours simulation speed over level of detail. Depending on the level of detail and error tolerance that is allowed, the balance between emulation accuracy and speed needs to be defined. In Figure 2.4 it can see a comparison between the simulation accuracy and speed, when dealing with RTL and DBT, other than that, some tools and languages are also marked taking into consideration its capabilities and functionalities regarding both RTL and DBT techniques support.

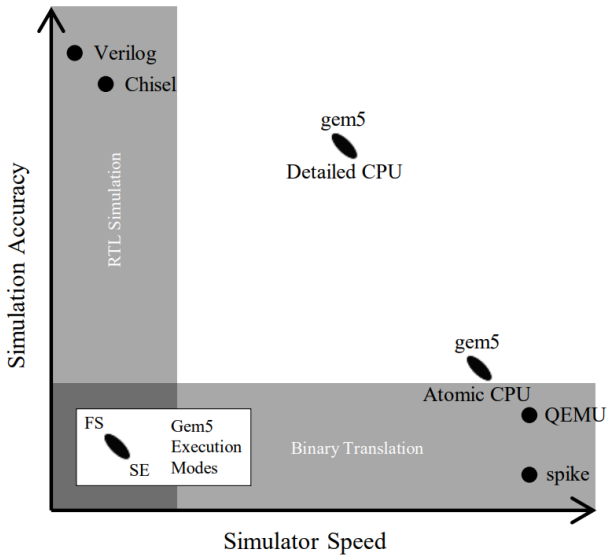


Figure 2.4: Simulation accuracy vs speed - Retrieved from [34]

In [34] Alec Roelke and Miercea R. Stan, discuss the gem5 simulator [35], which other than providing advanced simulation features such as system call emulations and checkpoint definition, it also provides detailed performance data. Furthermore, it supports a wide range of ISAs, ranging from proprietary ISAs such as x86 and ARM, with RISC-V among the supported ones, including a wide variety of different system-level architectures and processor models. This tool enables full system emulation and can be integrated with detailed functional models, such as CPU pipeline, instead of only using abstract timing models, which are sufficient when considering a pure performance evaluation. It is important to notice

that as the complexity increases, so does the amount of resources required to execute the simulations, which may lead to a decrease in performance, when comparing to other tool using only timing models [33].

Regarding RTL translation, Chisel [36] is a hardware design language based on Scala, that defines hardware datatypes and routines and by taking advantage of multiple Object-Oriented Programming (OOP) concepts, such as inheritance and polymorphism, that can be used later as hardware generators, that can convert the hardware descriptions to a C++ simulator or low-level Verilog code that can be latter mapped to a FPGA or used in an Application Specific Integrated Circuit (ASIC) flow [34, 36].

A tool that also supports RISC-V and provides full-system emulation is QEMU [37]. This tool mainly recurs to the usage of DBT techniques and so is designed to execute rapidly in order to provide functional validation of complex systems [33]. QEMU also provides other features that are highly valuable as an emulator, such as, it is noticeable portability that supports multiple architecture such as x86 and RISC-V; User mode emulation; exception support; and hardware interrupts, among others [38]. These features provide a good environment to perform debugging during the development process.

In table 2.4 a feature comparison between the previously mentioned tools is shown.

Table 2.4: Features and Compatibility of emulation tools, adapted from [34]

Feature	gem5	Chisel	QEMU
Bynary Translation	✓		✓
Checkpoints	✓		✓
Multicore simulation	✓	✓	✓
Performance statistics	✓	✓	
RTL imulation		✓	
System Call emulation	✓	✓	✓
ASIC synthesis		✓	
FPGA tools		✓	
Phase Analysis	✓		
Stats-based tools	✓	✓	

2.3 Real-time Operating Systems in RISC-V

This section provides an overview in Real-Time Operating Systems, discussing their paradigms, characteristics and support for RISC-V Systems.

2.3.1 Real-Time Operating Systems

Real-Time Operating Systems (RTOS) have been exposed to multiple mutation and evolution across the years, evolving from a single-use specialized systems, designed to support a respective set of tasks that met a goal, to a wide variety of more general purpose systems, such as real-time variants of Linux systems. Furthermore, apart from being completely predictable to support safety-critical applications, they are also developed to support soft real-time applications, such as video or audio streams [39].

The current market for RTOS is highly heterogeneous and range from proprietary kernels to real-time versions of popular Operating Systems (OS) such as Linux and Windows. A RTOS emphasizes the predictability, efficiency and need to support timing constraints [39]. Different application requirements led to further specialization of a RTOS based on the target system inherent characteristics, some kernels are designed to be small and fast, having the goal of reducing the run-time overhead incurred by the kernel when interacting with the system.

As John A. Stankovic and R. Rajkumar mentioned in [39] "fast is a relative term and not sufficient when dealing with real-time constraints", because of this when choosing the systems kernel its necessary to take into consideration the workload it will be subjected to and also the amount of resource and tasks that will be necessary for its execution. Depending on this, and the criticality of the system, a kernel that allows our system to perform as intended and is capable of meeting the tasks real-time constraints needs to be chosen.

Each kernel employs multiple techniques to deal with and ensure that timing requirements are met, such as, supporting multi-tasking, priority based and preemptive scheduling mechanisms and maintaining a high-resolution real-time clock, are some of the most widely used. While further details and remarks can be taken into consideration depending on the system requirements, some of the more relevant real-time related functions mentioned in [39] are:

- timers.
- priority scheduling.
- shared memory.
- real-time files.
- semaphores.
- inter-process communication.
- asynchronous event notification.
- process memory locking.
- asynchronous Input/Output (I/O).
- synchronous Input/Output (I/O).
- threads.

2.3.2 RTOS Paradigms

Additionally, it is also critical that the RTOS is capable of ensuring **Hard and Soft Real-Time Guarantees**, by analysing and designing the system taken into consideration that all inputs and system details known, it is possible to meet hard deadline requirements, furthermore, it is typical to maintain a margin that does not exceed 60% resource utilization when dealing with safety critical systems to give some margin of maneuver if an unexpected problem happens [39].

Another important function is the **Admission Control**, which decides if a given task should be admitted into the system or not. Note that some hard real-time systems are scheduled based on heuristics while other are statically analysed, so it is important to be aware of the system resources and the incoming request characteristics to ensure a good evaluation, to

employ online timing analysis that allow on-the-fly dynamic scheduling or a correct static analysis if its the case [39].

Additionally, it is also crucial to take into consideration the **resource reservation and allocation**, ensuring that for a given task, the necessary resources are sufficient and reserved. If this resource reservation is employed carefully, it is even possible to avoid conflicts of mutual access, via scheduling instead of ensuring it with semaphores or other similar mechanisms, a precise reservation is a valuable characteristic when timing issues are relevant [39].

Reflection increases OS flexibility and can be used to obtain better performance, by interpreting the system and applications metadata it is possible to improve online decisions and guarantee a better performance for dynamic environments [39].

Finally, **Resources Kernel** is another paradigm that introduces the notion of "portable" resource kernel, by providing unique interfaces to create or destroy resource sets, which are basically a virtual machine where the resources are guaranteed and bound to one or more applications. By allowing its reservation, allocation/deallocation, resizing, binding/unbinding and detail acquisition it is possible to allow an application to use "more resources" than the actually available in the system [39].

Usually, when an application requires resources it will lock them to its usage and sometimes will not make use of all allocated resources. By virtually allocating resources allows for mapping actual used resources to an application, allowing for the use of unused resources if they are not being used, ensuring the application maintains the necessary resources if needed.

2.3.3 RISC-V support

In [40], a guide provided by Antmicro [41], a RISC-V Foundation member, presents a guide in how to get started with RISC-V development. In there two different operating system are showcased, Zephyr [42], a Real-Time operating system, and Linux, more specifically a Debian distribution [43], a General-Purpose Operating System with support for real-time operations via extensions. Another operating system that provides support for RISC-V is FreeRTOS [44], another open-source Real-Time operating system.

Zephyr

Zephyr is a RTOS that has been developed with security in mind, the project is overseen by Linux Foundation [19], inserting this operating systems on a large community of open-source development, ensuring the long-term sustainability of this project [45].

The key features of this operating system are [45]:

- **Board Support:** Support a wide variety of common architectures and instruction sets including ARM, x86, and RISC-V.
- **Security and reliability:** Its robust and secure kernel, provides supports for multiple scheduling algorithms, memory protection, and fault handling. Additionally, other security features such as cryptography, secure boot, and firmware updates are also provided.
- **Driver libraries:** The included built-in driver interfaces and libraries provide support for a wide variety of sensors and devices, as well as power management functionalities to ensure a long battery life.

- **Modularity and configurability:** A configurable and modular approach was used when developing the operating system, allowing developers to customize it according to their requirements and preferences, being capable of running on devices with as little as 8 KB of RAM, or scaled up to support complex applications and hardware.
- **Open source and community:** Being an open source project backed by the Linux Foundation, it ensures a long-term sustainability and vendor-neutral governance, with an active community, that provides large amounts documentation, tutorials, forums, and mailing lists.

FreeRTOS

FreeRTOS was originally developed by Richard Barry around 2003 and was latter acquired by Amazon Web Services and licensed under an MIT open-source code license. Its development is transparent and community-driven in GitHub, and does not require any special tools or development practices [6].

The key features highlighted for this operating system are [6]:

- **Architecture and Framework Support:** Notable for its support for multiple architectures, making it versatile for various hardware platforms. Additionally, it has a growing set of tools and libraries, indicating ongoing development and enhancement efforts. This continuous active development ensures that FreeRTOS remains relevant and adaptable to evolving project requirements.
- **Simplicity and Low Memory Footprint:** Designed with a focus on having a small memory footprint, becoming an ideal choice for resource-constrained embedded systems. Its modular design allows developers to include only necessary components, further reducing memory usage. The core RTOS kernel is contained in just 3 C files, demonstrating its simplicity. Moreover, the minimal ROM, RAM, and processing overhead ensures an efficient resource utilization, furthermore, the RTOS kernel binary image size ranges from 6K to 12K bytes.
- **Integration with Middleware:** Can be easily integrated with a wide variety of middleware components such as TCP/IP stacks, file systems, and USB stacks. This integration makes it easier for complex embedded systems development, which require different functionalities such as networking, storage, and other more advanced functionalities. By leveraging the chosen middleware components, it is possible to provided a tailored solution that enhances the capabilities of the systems, so that they can meet the target system applications requirements effectively.
- **Free and Supported:** Available for use in commercial applications, it offers both a commercial licensing option for businesses and an open-source free to use under MIT license. Moreover, the professional support and porting services, provided by their partners, ensure a reliable assistance when dealing with complex systems. Its extensive documentation also aids developers on their understanding of the operating system features, while the active free support forum serves as a valuable resource where users can seek assistance and share insights among their peers. For organizations requiring dedicated assistance, commercial support is available, ensuring reliable support options tailored to specific needs.

2.4 Performance Analysis

Due to the increasing complexity of software and hardware, performance aspects have been becoming more and more important, specially when we are referring to embedded systems. The earlier a performance analysis is conducted over the development process, the faster we can ensure that the system is functioning properly and has the necessary hardware and software capabilities. However, this is not an easy topic to address as there is a lot of subjectivity involved regarding what is acceptable or not, furthermore, execution times and events are not easily measured due to the lack of functionalities present on a system, which makes it harder to do so accurately [46].

Ensuring that a system is capable of responding to internal or external events dynamically is specially important on systems that needs to handle multiple tasks, and if the target is a real-time system that addresses critical topics, this necessity becomes even higher, as if one of the tasks misses a deadline it can bring unpredictable negative impacts in some cases.

Another issue is the measurement process used during the development process, which frequently interferes with the system as a whole. This may lead to biases in the measuring results as the final deployed system will not have these interfering functionalities. As a result, it is expected that the production system, which is not impacted by interruptable functionalities such as debugging options or profiling, will be more efficient than the development system [46].

The following subsections will provide an overview of key metrics used to assess system performance, followed by a discussion of RISC-V's support for Hardware Performance Monitoring (HPM).

2.4.1 Key metrics

A variety of metrics can be used and analyzed to assess the performance of a system or software application. In fact, a wide range of factors can and should be used to assess the efficiency and effectiveness of a specific task or system. Some of the more commonly considered metrics include:

- **Throughput:** Can be defined as the number of tasks completed within a predetermined time period; the higher the throughput, the more efficient the task execution.
- **Latency:** It is defined as the time elapsed from the start to the completion of a single specific task. Low latency is critical for real-time applications that require high responsiveness.
- **Energy Consumption:** This represents the energy and power consumed during a system's operation or task execution. Lower power consumption is critical for energy-efficient systems with limited resources, such as embedded systems, particularly if they must be portable and not easily accessible.
- **Resource Utilization:** CPU and memory usage. It is critical to ensure efficient resource utilization for all systems. Typically, as efficiency increases, the power required to complete a task decreases, as does the time consumed in some cases.
- **Scalability:** Refers to a system's ability to handle increasing workloads; this is an important factor for a system in adapting to environments with increasing demands over time.

- **Fault Tolerance:**The ability to maintain a system's functionalities in the presence of faults or failures is critical when referring to critical systems, where incorrect data or results from a specific task or functionality can have disastrous consequences.

2.4.2 Hardware Performance Monitoring

Most modern processors include a Performance Monitoring Unit (PMU) that measures system performance and allows it to react accordingly. This unit can be implemented as a hardware unit, a software unit, or a combination of both. By gathering various statistics and events during system operation, it is possible to analyze processor and memory runtime execution and use those values when debugging or profiling code in order to react or adapt the system to meet the required metric. Intel and ARM define their own PMU functionalities in their developer manuals [46].

However, when it comes to PMUs for embedded system processors, it is clear that they are not commonly employed, with the majority of them only found in systems intended specifically to perform such tasks on a highly critical system [47]. Keeping this in mind, it is critical to ensure that such devices with restricted capabilities have the necessary hardware and software to perform as intended.

To address this important topic, the RISC-V International Foundation established the RISC-V Performance Analysis SIG (Special Interest Group), which is in charge of driving the strategy and coordinating the development of end-to-end solutions for monitoring, reporting, and analyzing the performance of software executing on RISC-V systems. This group is also responsible for selecting the software tools used in performance analysis, making sure the necessary ISA specifications are followed and that the hardware mechanisms that enable these software tools are present in all existing and in new RISC-V specifications [11].

Hardware performance counters are frequently used to analyse the performance of processing units. Performance counters are basically a specialized set of registers located within the processor whose sole purpose is to keep track of various events during a program's runtime execution. These counters provide information about a system's performance by monitoring specific metrics such as the number of instructions executed, cache hits and misses, and other relevant activities.

A RISC-V platform generally supports the monitoring of various hardware events. By relying on a limited number of hardware performance counters, it becomes capable of registering a wide range of events, including hardware general events such as the number of CPU cycles and instructions retired, till, branch prediction instructions, and cache hits and misses [48]. It can also track certain firmware events, such as the number of misaligned load/store instructions.

As previously stated, a Supervisor Binary Interface (SBI) is an interface that allows supervisor-level software, such as an operating system kernel or hypervisor, to interact with and control the underlying hardware in the RISC-V architecture.

The SBI Performance Monitoring Unit (PMU) extension is an interface for supervisor-mode to configure and use the RISC-V hardware performance counters when executing on the machine-mode level [48].

In general, these hardware performance counters can only be managed directly through machine-mode instructions. As a result, a machine-mode SBI implementation may choose

to disallow SBI PMU extension if the required hardware is not implemented by the specific RISC-V platform [48].

The RISC-V ISA is part of a continuous ecosystem development, from expanded software compatibility to an increasing number of hardware implementations. Alongside the software and hardware, the RISC-V specification is constantly evolving to meet the growing needs of the RISC-V ecosystem. Since RISC-V privileged specification version 1.7 [49], a minimal performance monitoring interface was defined for the Hardware Performance Monitor (HPM) (see Figure 2.5). From then, the specification has introduced additional counters and necessary features for access control and event multiplexing [50].

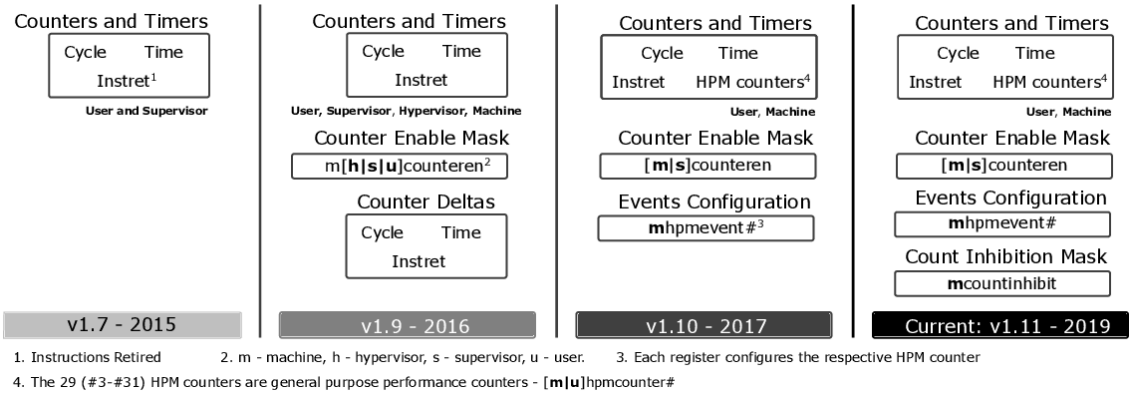


Figure 2.5: Evolution of RISC-V HPM specification - Retrieved from [50]

Version 1.9 [51] enhanced privileged counter access control by introducing counter-enable masks and delta counters to calculate differences between lower privilege counters and machine-level counters.

In version 1.10 [52], 29 additional Hardware Performance Monitor (HPM) counters have been included, allowing for extensive configurability with selectable events. Each HPM counter, from hpmcounter3 to hpmcounter31, can be individually configured using hpmevent registers, providing a high level of design flexibility.

Version 1.11 [53] introduced counter inhibition, enabling software to atomically sample events by stopping counting through the mcountinhibit register [50].

Ratified specification version 1.13 of the RISC-V privileged ISA, introduced no changes to performance monitoring when compared with version 1.11.

Further objectives might include supporting common event standardization, counting ISA-level metrics, and standardizing widely used micro-architectural metrics. In addition, features such as counter overflow interrupts can be used to accurately count events that exceed the counter capacity. The RISC-V specification is constantly refined to improve performance monitoring and meet changing requirements [50].

At the time of writing, RISC-V performance monitoring support is considerably simpler than its x86 counterpart and does not compare to dedicated performance analysis tools such as ARM's coresight or Intel's PCM-based monitoring solutions. Even so, the RISC-V HPM specification is a flexible generic performance monitoring solution, and its open-source nature allows any degree of implementation freedom [9, 50].

2.5 Approaches for performance measurements

There are already a lot of performance measurement solutions and tools available, such as Intel Performance Counter Monitor (PCM) [54], PAPI[55], Perf[12] and Extrae[56], yet not all of them support a RISC-V architecture. One such example is Extrae, a tool developed by Barcelona Supercomputing Center, that uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance that can be later analyzed after the application execution [56]. Another one is Intel PCM, this is a dedicated API that can be used to monitor performance and energy metrics of Intel Cores. Other than the API libraries that can be directly included on code, it also provides a command-line interface (CLI) and some graphical dashboards for better visualization [54].

Some of the most popular performance analysis tools available for RISC-V architectures are perf and PAPI.

- Performance analysis tools for Linux, or Perf, is an open-source tool for kernel profiling tool that is part of the Linux kernel and can be used to perform multiple performance analysis, including CPU profiling, memory profiling, and more. It is a command-line tool that allows the collection and analysis of a wide variety of performance data, the majority of which comes from hardware performance counters [12].
- The Performance Application Programming Interface (PAPI) is a tool used to gather performance counter metrics from various hardware and software components. It supports multiple architectures, and some of the top industry companies, together with other liaisons, ensure the integration of PAPI with new architectures at or near their release [55].

Aside from Papi and Perf, which are both open-source solutions, other solutions have emerged over time; some are built on top of them, while others are not.

In [57], J. Silveira et al. introduce Prof5, a profiling tool tailored for RISC-V architectures. Prof5 takes advantage of functional simulation with accurate energy and timing models derived from RTL simulation and power analysis. This tool will execute the provided binary application in a simulated environment, generating auxiliary files and reports for performance analysis, particularly power consumption.

In article [46], Tobias Scheipel, Fabian Mauroner, and Marcel Baunach developed a module in hardware description language (HDL) that allows for measuring execution times, task-aware and unaware events without interfering with the system. This module is designed to be directly integrated into the RISC-V architecture pipeline, allowing it to gather all system and task information without interfering with the system itself; additionally, it can access system memory. Moreover, besides minimizing hardware time usage, this approach allows for the setting up of a more detailed performance profile by providing information about the task being executed. Given that it is a non-intrusive approach, it has no impact on the worst-case execution time analysis.

The paper [50] and [58] propose new extensions and modifications to the existing Perf tool, along with an interface for the PAPI library. It aims to achieve full support for the most recent RISC-V performance monitoring specification by modifying the Linux kernel and integrating an OpenSBI API. This paper primarily focuses on providing specially designed mechanisms for reading and writing to machine-level privileged counters and registers.

All of the mentioned approaches provide wide support for retrieving performance metrics directly from the HPM in RISC-V systems. However, when considering Real-Time support they become somewhat lacking. Most of them are only usable under Linux systems, and support to RTOS was not encountered on the respective documentation. The implementation itself highly relies on the Kernel implementation and modifications, which might prove troublesome when trying to port new features to the system as it might impact the overall Operating System functionalities.

Additionally, the more widely used solutions, PAPI and perf, are not exclusively targeted to RISC-V systems. Despite this not being a huge issue, as it is ported across multiple systems and platforms, it is difficult for the API to rapidly support new features and platform that may arise during the years. The more platforms and system's architectures an API needs to support, the more hours and effort needs to be put into motion in order to keep the API up to date with the latest features. As RISC-V has been experiencing a huge development during the latest's years, these APIs don't provide support to the latest features and functionalities, and as times goes on and new specifications are released, it will be staying outdated and do not provide full support to the RISC-V architecture.

Furthermore, as RISC-V is an easily extendable ISA specification, manufacturers often implement their own extensions, this increases the effort even more in order to support the new extensions. As these solutions have already been under development for a long time, they provide a lot of functionalities that may not be useful for most cases, especially if we are considering resource constrained devices. Usually, in this systems, the manufacturer opts to develop its own custom extension to perform performance monitoring in order to save costs on development and also on an attempt to reduce the overhead generated by the PMU.

Such an example is the ESP32C6 board [59], which provides a custom extension on its High-Performance Core to deal with performance monitoring. This extensions uses the same principles applied for event configuration on the RISC-V ISA Privileged specification [9], with the *hpmevent* registers that are responsible to control the event that increments the respective counter, on a single performance counter. These customization would be inaccessible under the other solutions out-of-the-box and extending the existing implementations would often require kernel development which would limit the solution porting for other environments.

A solution that is capable of being easily deployed across different operating systems with minor changes, while also supporting new extensions and a wide range of configurations is missing. The simpler the porting process is, and the lower the reliance on external dependencies, the faster the API development can be done and also be kept up-to-date with the latest features of the RISC-V ISA.

Comparison

In Table 2.5, a more comprehensive comparison among the solutions mentioned that support RISC-V systems has been performed. In order make it clearer to read, the solution names, present on the first column have been shortened to ensure the table readability.

As it can be seen from the table a solution that provides support for RTOS is missing, this is a huge gap for RISC-V system as it has become a target architecture for critical systems, which may rely on RTOS. Despite both PAPI and Perf being potentially enabled to be ported to a RTOS, as they highly rely on the UNIX kernel for their functionalities, the porting process would be complex and time consuming.

Table 2.5: Comparison of performance monitoring solutions

Solution	Supported Operating System	Privilege Levels	PMU ISA Support	Extensible	Solution Type
Perf [12]	UNIX/Linux	M, U	yes, partial	yes	API
PAPI [55]	UNIX/Linux	M, U	yes, partial	yes	API
Prof5 [57]	General Purpose OS	M	no	no	FPGA + system emulation
System-Aware PMU [46]	OS agnostic	M	yes, additional features	no	PMU Hardware extension
RISC-V HPM Counter support on Perf [50] [58]	UNIX/Linux	M, U	yes	no	API extension of PAPI and Perf

It can also be observed that usually, supporting access to the full capabilities of the RISC-V ISA PMU only happens on custom solutions, both PAPI and Perf can attain this support by the usage of extensions such as described on [50] and [58]. As previously mentioned, supporting the latest features is crucial to perform a more accurate performance analysis. It is worth noticing that this analysis was performed taking into consideration the 2019 Specification for the Unprivileged ISA [10] and the 2021 Specification for the Privileged ISA [9], at the moment of writing the 2024 version have been released for both Unprivileged [60] and Privileged [61] ISAs, which further outdates the analysed solutions when taking into consideration the latests features added on these specifications for performance monitoring.

2.6 Summary/Analysis

As has been seen, RISC-V has gained popularity in a variety of industries; furthermore it has proven to be an excellent candidate for academic research. This ISA has been having a significant impact on computing's future and will continue to do so for many years. The ongoing development of RISC-V focuses on the development of multiple optional extensions that will enhance the system's performance for specific domain applications.

The RISC-V ISA is currently in constant development, so no complete implementation is available. When it comes to hardware implementations, only a few ISA base and extensions are supported for specific versions. This requires the employing of Emulation or Simulation systems, which have already demonstrated their credibility in a number of projects and solutions throughout the industry.

There is a lot of work that has not yet been ratified, particularly in the performance area; there is currently a drafted specification for the Counters extensions, which establish additional performance counters on the system. Despite this, nearly all of them are open and customizable, and must be defined based on implementations, with only three generics defined: *mcycle*, which tracks the number of cycles executed by the processor since it was reset or powered on; *mtime*, which is responsible for tracking the passage of time in real-time;

and *minstret*, which tracks the number of instructions executed by the processor since the reference point defined by *mtime*.

Some solutions for performing performance analysis on RISC-V have already been proposed, but only a few support the entire ISA specification for this topic, and the majority of them are only available for Linux systems. When it comes to RTOS, there is a lack of support, with only a few solutions available, none of which are as detailed as the Linux performance tool. Despite that, the research performed on this chapter provides some hints into both **RQ1** and **RQ2**. Existing solutions are already retrieving performance metrics from RISC-V systems, while providing configuration support. While few support was found for solutions related with both real-time applications and operating systems, it seems feasible to develop an approach that takes real-time constraints into consideration, as per questioned in **RQ3**.

Chapter 3

Performance Monitoring in RISC-V

This chapter discusses performance monitoring in RISC-V system, and how different PMUs functionalities can be configured and accessed depending on implementation specific details.

The RISC-V ISAs specifications define base counters and timers, along with custom general-purpose Hardware Performance Monitor (HPM) counters, which are used to continuously register performance metrics from the system during execution. The supervisor-level extensions specifies a rudimentary Performance Monitoring Unit (PMU). Meanwhile, in the unprivileged-level specification, the Counters extension, which encompasses the base timers and counters originally included under the Zicsr extension, was first drafter in version 2019 [10], and later ratified in version 2024 [60]. This ratification resulted in the establishment of two new extensions: *Zicntr* for the base counters and timers, and *Zihpm* for the additional customizable general-purpose hardware performance counters.

It is important to note that prior to the development phase of this work, the 2024 specification for both unprivileged [60] and privileged [61] ISAs had not yet to been released, and were only released while the thesis development was already been going on for a while. As a result, certain specifications that have an impact on the performance monitoring in RISC-V system - and can provide some clarifications and new functionalities in this context, specially when dealing with privilege levels and counters overflow - were not considered during the development of the API. Despite these specifications being referenced and will be taken into consideration for the future work on this API, as discussed on the following chapters, the present chapter was developed based on version 2019 of the Unprivileged ISA specification [10] and version 2021 of the Privileged ISA specification [9].

3.1 RISC-V PMU

The RISC-V PMU, as detailed on the ISAs specification, defines thirty two 64-bit performance counters and timers. The registers mapped to the 12-bit CSR address space will be different depending on the respective privilege mode.

As all of these registers are 64 bits wide, when the base ISA XLEN is 32 bits, they must be accessed in 32-bit portions. The register upper 32 bits are accessed via a different address space. To simplify referencing the upper 32 bits portion, a new mapping was created for each counter, with the naming convention that appends "h" to the counter name, such as *counterH*.

3.1.1 Base counter and Timers

The RISC-V ISAs specification defines several base counters and timer that can be used to retrieve basic system metrics. These base counters and timers should be implemented as dedicated CSRs within the system, each responsible for tracking specific events.

To this moment, only three base counters and timers are specified in the ISA, with the discussion for future additions of event-specific counters, such as those tracking cache hits or misses being underway. The currently defined counters are:

- *CYCLE*: Tracks the number of cycles that have occurred in the system since an arbitrarily defined point in time.
- *TIME*: Tracks the number of ticks that have occurred since an arbitrarily defined point in time.
- *INSTRET*: Tracks the number of instructions executed by a specific hart/core since an arbitrarily defined point in time.

CYCLE

The *CYCLE* counter is a 64-bit counter designed to return the number of cycles executed by the processor core, rather than by individual harts. It is intended to never overflow, and the rate at which it advances depends on the specific implementation and operating environment. Consequently, the execution environment should provide a mechanism to determine the current rate (*cycles per second*) at which the counter increments.

Defining a "core" can be challenging as it is highly dependant on implementation choices. Similarly, defining a "clock cycle" is also complex, as various considerations must be taken into account, leading to different implementation that may yield distinct definitions.

In systems with multiple cores or harts it can be difficult to distinguish individual cycles per hart. Therefore, the value in the cycles counter is intended to represent "*total per-core cycle*", a definitions that works well with the common implementation of a single hart/core case. With this in mind, the *CYCLE* counter should be primarily used for performance monitoring and *cycle-count/instructions-retired* should measure *Cycles-Per-Instruction (CPI)* for a hart.

Further definitions of cycle-count for systems with multiple harts/cores might be possible through the introduction of new hardware registers. However, this adds additional complexity, which contradicts the ISA's principle of simplification, it could be implemented as a new custom extension if justifiable by the system requirements.

There are many details specific to the execution environment and its implementation choices. For reference, what happens during "sleep" is not standardized across different executions environments and may have different implementations between systems. Despite that, if the entire core is paused during "sleep", it should not execute any clock-cycles, meaning the *CYCLE* counter would not increment.

TIME

The *TIME* counter is a 64-bit counter that tracks the wall-clock real-time elapsed from an arbitrary defined start time in the past. This counter increments by one with each tick of the real-time clock, and for realistic real-time clock frequencies, it should never overflow in

practice. The period of each tick should remain constant within a small error bound, and the environment should provide a way to determine the clock's accuracy (e.g. the maximum relative error between the nominal and actual real-time clock periods) as well as the period of a counter tick (*seconds per tick*).

On simpler platforms, the cycle count might represent a valid implementation of *TIME*, in which case the counters *TIME* and *CYCLE* counters may return the same value.

Given the wide variety of possible implementations across different platforms, it may be challenging to provide a strict mandate on clock period given. Therefore, the maximum error bound should be defined based on the platform's needs and requirements. Additionally, the real-time clocks of all harts in a single user application must be synchronized to within one tick of the system real-time clock.

INSTRET

INSTRET is a 64-bit counter that tracks the number of instructions retired by the current hart from an arbitrary starting point in the past. This counter should never overflow under normal conditions. Furthermore, instructions that cause a synchronous exception, such as *ECALL* and *EBREAK* instructions, are not considered to retire and hence will not increment the *INSTRET* CSR.

3.1.2 HPM counters

Apart from the base counters and timers, the ISAs specifications also allocate 29 additional 64-bit wide registers, designated as *hpmcounter3-hpmcounter31*, intended to be used as custom general-purpose hardware performance counters.

These counters monitor platform-specific events and may be configured via additional privileged registers. As a result, each manufacturer is responsible to provide the event configuration mapping, as well as the the width and number of registers used to configure the specific counter's events.

Similar to the base counters and timers, when the XLEN associated to the ISA is smaller than the length of the hardware performance counter, the complete value cannot be retrieved directly to the CPU registers, as the performance counter is wider than the CPU register.

In this case, additional mappings were created to access the upper 32-bits of the hardware performance counter, when XLEN equals 32. These mapping are named *hpmcounter3h-hpmcounter31h*, following the previously mentioned naming convention.

3.2 Accessing RISC-V PMU

Retrieving performance metrics usually requires access to the hardware registers, which can be accomplished either using the existing SBI, if available, or by directly access the registers with assembly instructions.

As previously mentioned, HPM counters and timers are mapped within the 12-bit CSR address space, allowing access via the instructions provided on the *Zicsr* extension, which handles all aspects related to Control and Status Registers in RISC-V system. By directly using the instructions from the *Zicsr* extension, we can reduce platform-specific and operating system dependencies, resulting in a more portable solution.

Configuring and accessing performance counters can be challenging sometimes, as different systems may implement different extension and registers, and may also use different interfaces for PMU configurations. Therefore, having an API that simplifies this process across multiple platforms with minimum changes can be extremely helpful for a system performance monitoring.

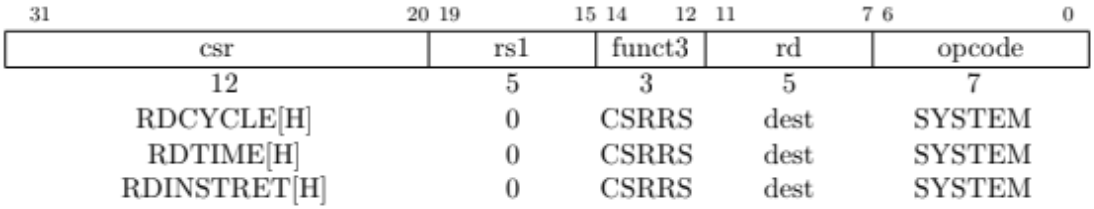
As mentioned earlier, while there are multiple extensions and features defined for performance monitoring across both Unprivileged and Privileged ISAs specifications, they essentially refer to the same core functionalities, differing in how they are accessed. Additionally, some custom extensions may also be used, providing different functionalities and configuration capabilities.

3.2.1 Unprivileged-level ISA specification

The Unprivileged-level ISA specification [10] provides a set of up to 32×64-bit performance counters and timers that are accessible thorugh unprivileged XLEN read-only CSR registers mapped under CSR memory address space.

Additionally, it defines several pseudo-instructions for accessing the base counters and timers: *RDCYCLE*, *RDTIME*, and *RDINSTRET*, each corresponding to the counter *CYCLE*, *TIME*, and *INSTRET*, respectively.

In fact, these pseudo-instruction are translated into "*CSRRS*" instructions from the "Zicsr" extension, using the specific CSR address where the counter is mapped. Therefore, it is possible to directly use the "*CSRRS*" instruction with the CSR memory address to retrieve the counter value.



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

Figure 3.1: Unprivileged pseudo-instructions to access base counters and timers. Retrieved from [10]

Depending on the base ISA XLEN, which determines the register size, the pseudo-instructions available behave differently:

- If XLEN is equal or higher than 64, the registers will have enough bits to retrieve the entire counter value, so the respective instructions will retrieve the full 64-bit value from the counter.
- If the XLEN is equal to 32, as is the case for the base *RV32I*, the *RDCYCLE*, *RDTIME* and *RDINSTRET* instructions will only access the lower 32-bit portion of the counter. To access the upper 32-bit of these counters, three new pseudo-instructions were introduced, following the previously mentioned naming convention. In this case, *RDCYCLEH*, *RDTIMEH* and *RDINSTRETH* will access the respective *CYCLEH*, *TIMEH* and *INSTRETH* counters, which correspond to the upper 32-bits of each counter.

Table 3.1: Currently allocated RISC-V unprivileged CSR Addresses for Performance Counters. Adapted from [10]

Unprivileged Counter/Timers			
Number	**Privilege**	**Name**	**Description**
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter 3.
0xC04	URO	hpmcounter4	Performance-monitoring counter 4.
...	...	...	...
0xC1F	URO	hpmcounter31	Performance-monitoring counter 31.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
...	...	...	...
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I only.

In table 3.1 the CSR mapping for the unprivileged counters and timers registers is displayed.

In [10] the unprivileged specification for the Hardware Performance Counters was drafted under the extension named "*Counters*" and was later ratified in [60]. In this ISA revision, everything related with the first three counters - cycles, time and instret - was defined under the extension "*Zicntr*" while the other 29 general-purpose hardware performance counters were defined under "*Zihpm*".

3.2.2 Privileged-level ISA specification

Under the privileged ISA specification, multiple hardware performance monitoring counters and registers are defined, each referring to a specific privilege level. It is important to note that aside from M-mode, which defines a basic hardware performance facility with some dedicated counters, both Supervisor and Hypervisor modes utilize the Unprivileged counters (User-mode).

Machine-mode Hardware Performance Monitor counters

The HPM counters defined for the machine-level ISA are the following:

- **mcycle** : a CSR that tracks the number of clock cycles executed by the processor core on which the hart is running.
- **minstret** : a CSR that records the number of instructions the hart has completed.

Both the *mcycle* and *minstret* registers provide 64-bit precision across all RV32 and RV64 systems. After a hart is reset, the counter registers hold arbitrary values, that can be set to a specific value by writing to them. Any write to a CSR takes effect only after the writing instruction has fully executed.

It is important to note that the *mcycle* CSR may be shared between multiple harts on the same core. In such cases, writes to mcycle will be visible to those harts. If this is the case. the platform should provide a way to indicate which harts share an mcycle CSR.

Additionally, the hardware performance monitor includes 29 additional 64-bit event counters, *mhpmcounter3* through *mhpmcounter31*, which can be configured using the event selector CSRs, *mhpmevent3* through *mhpmevent31*, which are MXLEN-bit WARL registers that determine which event triggers the corresponding counter to increment.

The definition of these events is platform-specific, but event 0 is universally defined as "no event". All counters should be implemented, even though a compliant implementation may set both the counter and its corresponding event selector to read-only (ergo 0).

When MXLEN is equal to 32, reads to the *mcycle*, *minstret*, and *mhpmcountern* CSRs returns bits 31–0 of the corresponding counter, and writing to them only affect these bits. Similarly, reading the *mcycleh*, *minstreth*, and *mhpmcounternh* CSRs returns bits 63–32 of the corresponding counter, and writing to them only modifies bits 63–32.

Figures 3.2 and 3.3 illustrate the Hardware Performance Monitor counters for M-mode.

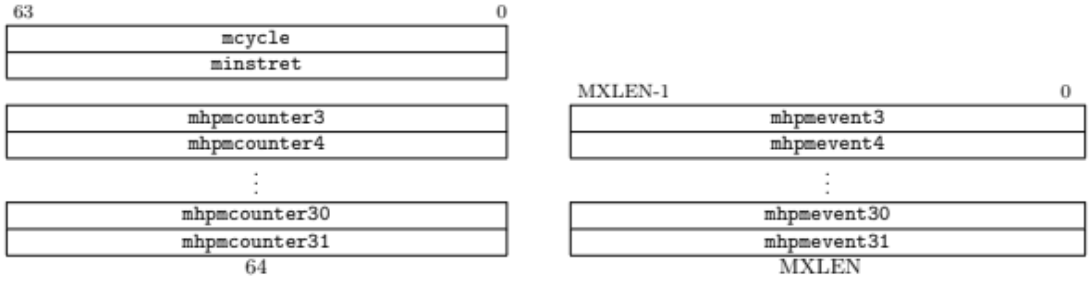


Figure 3.2: Hardware Performance Monitor counters for M-mode. Retrieved from [9].

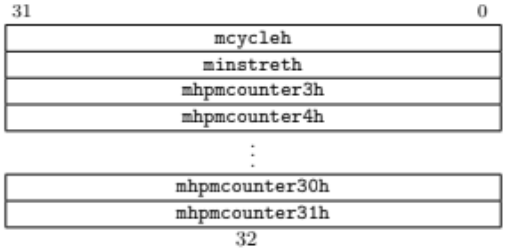


Figure 3.3: Upper 32 bits of Hardware Performance Monitor counters for M-mode. Retrieved from [9].

There are several important notes to take into consideration when using the HPM counters:

- The *cycle*, *instret*, and *hpmcountern* unprivileged CSRs are read-only shadows of *mcycle*, *minstret*, and *mhpmcountern*, respectively.
- On *RV32I* the *cycleh*, *instreth* and *hpmcounternh* unprivileged CSRs are read-only shadows of *mcycleh*, *minstreth* and *mhpmcounternh*, respectively.

- The unprivileged *time* CSR is a read-only shadow of the memory-mapped *mtime* register.
- On RV32I the unprivileged *timeh* CSR is a read-only shadow of the upper 32 bits of the memory-mapped *mtime* register, while time shadows only the lower 32 bits of mtime.

Machine Timer Registers

The privileged specification also defines the following machine timer registers:

- *mtime* : A memory-mapped, read-write register that increments at a constant frequency and has 64-bit precision on both RV32 and RV64 systems.
  - The platform must provide a method to determine the period of an *mtime* tick.
  - If the counter overflows, it will wrap around.
- *mtimecmp* : A 64-bit memory-mapped machine-mode timer compare register.
  - A machine timer interrupt is triggered whenever the value in *mtime* is greater than or equal to the value in *mtimecmp*, with both values treated as unsigned integers. This interrupt remains active until *mtimecmp* is updated to a value greater than *mtime*, which typically occurs when *mtimecmp* is written to.
  - The interrupt will only be taken if interrupts are enabled and the MTIE bit in the *mie* register (Machine Interrupt Enable register) is set. The MTIE bit specifically controls the enabling of machine timer interrupts, and setting this bit allows the timer interrupts to occur.

Figure 3.4 illustrates the two 64-bit wide *mtime* and *mtimecmp* registers.

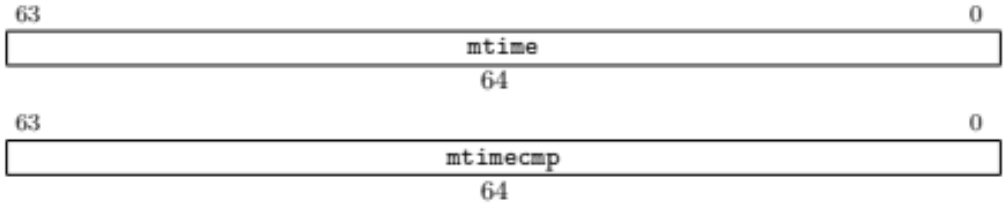


Figure 3.4: Machine time register and Machine time comapre register (memory-mapped control register). Retrieved from [9].

Availability control

The availability of each HPM counter can be controlled via dedicated CSRs for each privileged level. Each register is a 32-bit CSR that manages the availability of the hardware performance counters to the next-lowest privileged mode. Reading or writing to this register does not affect the underlying counters, which continue to increment regardless of accessibility.

If multiple privilege modes are implemented in the system, the ISA enforces that a Counter-Enable register needs to be defined to control the accessibility between privilege levels. However, if only M-mode is implemented. there is no need for a Counter-Enable register.

In figure 3.5, the bit organization for the counter-enable-register is displayed.

Figure 3.5: Counter-enable register. Retrieved from [9].

When the CY, TM, IR, or HPMn bit in the register is cleared, any attempt to read the unprivileged *cycle*, *time*, *instret*, or *hpmcountern* registers while operating in a lower-privilege level will result in an illegal instruction exception. If one of these bits is set, access to the corresponding register is allowed in the next implemented privilege mode.

According to [61], the following Counter-Enable registers were defined:

- *mcounteren* : Machine Counter-Enable Register
  - The register controls the accessibility of HPM counters in S-mode if implemented, otherwise U-mode.
  - If U-mode is not implemented then this register should not exist.
- *scounteren* : Supervisor Counter-Enable Register
  - This register controls the accessibility of HPM counters in U-mode
  - The setting of a bit in *mcounteren* does not affect whether the corresponding bit in *scounteren* is writable.
  - U-mode can only access a counter if the corresponding bits in *scounteren* and *mcounteren* are both set.
- *hcounteren* : Hypervisor Counter-Enable Register
  - This register controls the accessibility of HPM counters to the guest virtual machine.
  - In VU-mode (Virtual-User mode) a counter is not readable unless the applicable bits are set in both *hcounteren* and *scounteren*.
  - If the Hypervisor ISA is being used, *hcounteren* must be implemented.

Counter incremental inhibition

The privileged specification provides a mechanism to control which HPM counters are allowed to increment through the *mcountinhibit* register, as shown in Figure 3.6.

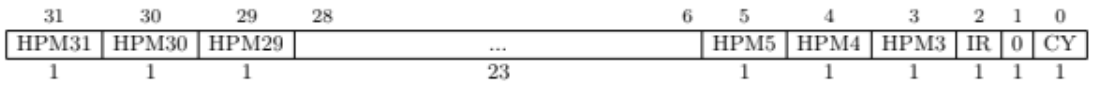


Figure 3.6: Counter-inhibit register *mcountinhibit*. Retrieved from [9].

This register only affects whether a counter increments and it does not influence the accessibility of the counter and the following points are taken into consideration:

- When the CY, IR, or HPMn bit in mcountinhibit is clear, the *cycle*, *instret*, or *hpmcountern* counters increment as usual. If the bit is set, the corresponding counter does not increment.
- As the *mcycle* CSR can be shared between harts on the same core, the corresponding bit in *mcountinhibit* is also shared, meaning any changes to this bit will be visible across those harts.
- If *mcountinhibit* is not implemented, it defaults to zero, allowing counters to increment as normal.
- Inhibiting counters when they are not needed is desirable for reducing energy consumption.
- As the *time* counter, may be shared across multiple cores, it cannot be inhibited using the *mcountinhibit* mechanism.

3.2.3 Custom extensions

RISC-V follows a modular approach and is known for its flexibility, portability and expandability, allows manufacturers to implement custom extensions tailored to deal with Performance Monitoring. There may be multiple reasons for choosing to develop their own custom extensions, such as cost reduction, by providing a reduced number of counters and adopting proprietary configuration methods, or to simply introduce new features not currently available or supported by the standard ISA specification.

Since performance monitoring mechanisms in the RISC-V ISA are relatively limited compared to more mature proprietary ISAs, manufacturer may find it beneficial to develop their custom extensions to enhance the system performance monitoring capabilities.

Due to this diversity, it is necessary to take into consideration the potential integration of custom extensions for performance monitoring context on any given platform. One example is the ESP32C6 board by Expressif, which incorporates a custom proprietary extension developed by its manufacturer, adding three new custom registers to enhance performance monitoring on its High-Performance core [59]:

- **mpccr** - A custom 32-bit machine performance counter register. The bit representation can be seen in Figure 3.7.

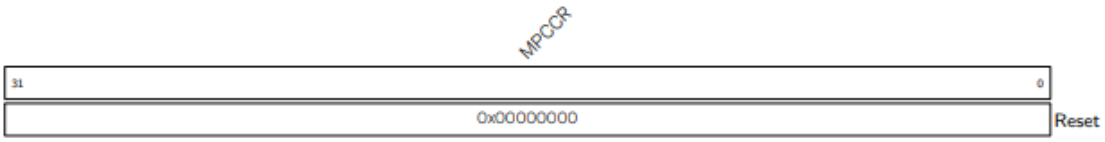


Figure 3.7: mpccr register. Retrieved from [59]

- **mpcmr** - A custom 32-bit register that configures counter saturation behaviour. If *COUNT_SAT* is set to 0 then the **mpccr** counter will overflow; otherwise, it will halt at its maximum value. It also enables **mpccr** by setting **COUNT_EN** to 1. The bit representation is shown in Figure 3.8.

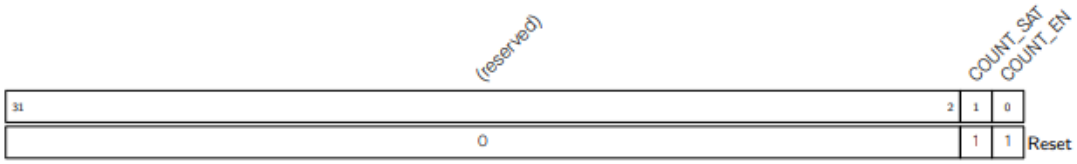


Figure 3.8: mpcmr register. Retrieved from [59]

- **mpcer** - A custom 32-bit event selector register for the **mpccr** performance counter. The bit representation is shown in Figure 3.9.

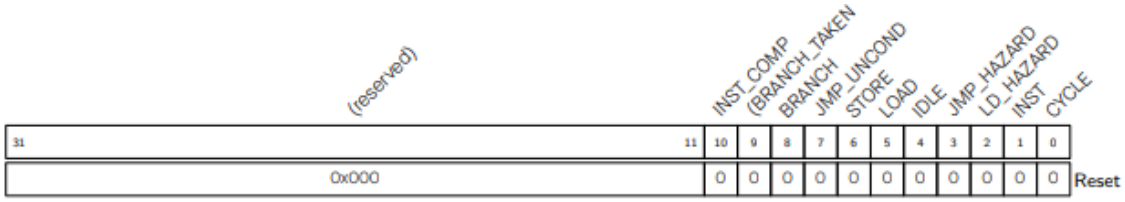


Figure 3.9: mpcer register. Retrieved from [59]

  Each bit set will cause the **mpccr** counter to increment when the corresponding event mapped to that bit occurs. Note that if multiple events are selected and occur simultaneously, then the counter will only increment by one. The bit mapping is as follows:

  - **CYCLE**: Count Clock Cycles. Cycle count does not increment during WFI (Wait-for-Interrupt) mode.
  - **INST**: Count Instructions.
  - **LD_HAZARD**: Count Load Hazards.
  - **JMP_HAZARD**: Count Jump Hazards.
  - **IDLE**: Count IDLE Cycles.
  - **LOAD**: Count Loads.
  - **STORE**: Count Stores.
  - **JMP_UNCOND**: Count Unconditional Jumps.
  - **BRANCH**: Count Branches.
  - **BRANCH_TAKEN**: Count Branches Taken.
  - **INST_COMP**: Count Compressed Instructions.

These custom registers allow the system to support event selection functionalities system when operating in high-performance mode. Further details will be given in Chapter 6.

Having a single configurable performance counter on the high-performance core, optimizes the system's overall performance. In system with multiple *hpmcounters* monitoring various events, an increase number of counters can introduce a significant overhead when retrieving those value. Even without retrieving the counter's values, the act of incrementing counters in response to events can increase power consumption, memory and the system's resources

overall consumption, a critical concern for embedded system operating in resource constrained environments. Moreover, this can also negatively impact the system performance, potentially causing missed deadlines and undesirable behaviour.

3.3 Configuring RISC-V PMU

Proper configuration of the RISC-V PMU is essential for accessing and retrieving different performance metrics. The configuration process is composed of several steps, depending on the system's available capabilities:

- **Event configuration** - Depending on the platform features, it is possible to configure which events will trigger the desired performance counter to increment. If available. It is crucial to configure the system to ensure that it tracks the performance metrics that the user wants to retrieve.
- **Counter inhibition** - Some systems offer the option to inhibit counters from incrementing, which helps in reducing resource consumption by the RISC-V PMU by disabling counters that are not currently in used.
- **Counter availability** - On systems with multiple privilege levels, it is possible to control which performance counter are available at each privilege level. By restricting counters access under a specific privilege level, enables the system to control which metrics are available based on the privilege level.

Event Configuration

Event configuration is a feature that is highly dependant on platform-specific implementation details. Some systems have fixed events assigned to each hardware performance counter and do not allow changes, while others may lack general-purpose HPM counters altogether. However, some platforms support event configuration, either through the usage of *hpmeventx* register associated with the respective *hpmcounterx* register, or via their own custom-defined extensions.

Usually, configuration is performed by modifying a CSR register that selects the event tracked by a given counter. According to the Privileged ISA [9], each *hpmcounterx* can have an associated *hpmeventx* register that determines the event that will cause the counter to increment. By writing the appropriate value to the *hpmeventx* register - mapped to the desired event - whenever that event occurs in the system the respective counter will increment.

The configuration can be achieved using the *CSRRW* instruction from the *Zicsr* extension, which can be used to write values to a specific CSR, using this instruction it is possible to write the desired event value to the CSR-mapped address of the register. However, the specific value and structure of the *hpmevent* register depend strongly on the platform's implementation details.

For example, on the ESP32C6 platform, event configuration is managed by the custom extension for performance monitoring, where manipulating the bits from the *mpcer* register, defines which events will cause the *mpccr* custom performance register to increment.

Counter inhibition

According to the Privilege ISA specification, it is possible to inhibit a counter from incrementing by setting or clearing the respective bit associated with the target counter in the *mcountinhibit* CSR register, if available. This can also be accomplished using the *CSRRW* instruction from the *Zicsr* extension.

Similar to event configuration, not all platforms implement the *mcountinhibit* register and some may implement a different approach for this functionality. Therefore, this feature should only be utilised if the target platform supports it, as attempting to write to an unsupported register can trigger an illegal instruction exception.

For the ESP32C6 platform custom's extension, counter inhibition for the *mpccr* register is controlled by the **COUNT_EN** bit (position 0) in the *mpcmr* register. Setting this bit to 1 enables the *mpccr* register to increment, while clearing stops the counter from incrementing. Additionally, the *mpcmr* register allows control over the counter's saturation behaviour, via changing the bit *COUNT_SAT*, which causes the *mpccr* counter to overflow when set to 0, and to halt at its maximum value otherwise.

Counter availability

Another feature available in the PMU is the option to configure the counter availability at specific privilege levels by setting or clearing the corresponding bit - associated with the target counter - in the *Counter-Enable* CSR register for the target privilege level. This configuration can be executed using the *CSRRW* instruction from the *Zicsr* extension.

When configuring counter availability, it is important to consider the supported privilege modes in the system. The availability of counters is managed by different registers depending on the implemented privilege modes. Furthermore, to enable a counter on a lower privilege level, it requires that on the next higher privilege level, that specific counter is also available; otherwise, it will not be accessible at a lower privilege level, due to the higher privilege level's precedence.

For the ESP32C6 custom extension, configuring counter availability across different privilege levels is not supported.

3.4 Using the RISC-V PMU

Using the RISC-V PMU highly depends on the specific implementation details of the platform being used. For systems that implement the Privileged ISA specification, the following steps can be performed to use the RISC-V PMU:

- **Enable Counters**: The first step is to enable the required hardware performance counters by manipulating the *mcountinhibit* CSR register. To enable a specific counter, the user may set to 1 the bits corresponding to the target counters, and reset the bits to 0 for counters that should be disabled.
- **Configure Events**: The next step is to configure which events will cause the respective counter to increment. To configure *hpmcounterx*, the user may set the respective value corresponding to the mapped event, under the *hpmeventx* CSR. Whenever the mapped event occurs in the system, the *hpmcounterx* will increment.

- **Set counter availability**: Control which metrics are available at different privilege levels, by configuring the appropriate *Counter-Enabled Register* (E.g., *mcounteren*, *scounteren*, *hcounteren*, etc.). Each CSR only controls the next lower privilege level counter's availability, and as privilege level precedence exists from higher to lower privilege levels to make a specific counter accessible at a lower privilege levels, it must also be enabled for all the higher privilege levels.
- **Access Counters**: Once all the configurations have been performed, the desired *hpmcounter* can be accessed on the desired privilege level.

For the ESP32C6 custom extension, similar steps are followed using the custom registers:

- **Enable Counters**: The first step is to enable the *mpcrr* custom performance register by setting the bit *COUNT_EN* in the *mpcmr* register. Optionally, configure the saturation behaviour of the register by setting or resetting the *COUNT_SAT* bit.
- **Configure Events**: Once the counter is enabled, configure which events will cause the *mpccr* to increment. This is done by setting or resetting the desired bits in the *mpcer* register, for the events required. For example, to track the number of loads and stores executed in the system, set the bits *STORE* and *LOAD* to 1 while resetting all the other bits to 0. This would cause the *mpccr* register to increment by one each time one of these two events occur in the system.
- **Access Counters**: To finish, access the *mpccr* register and handle its value according to the application's requirements.

Despite these approaches differing in some points, they share common steps: configuring the PMU, by enabling the necessary counters; selecting events; managing the counter's accessibility across different privilege levels; configure saturation behaviour (if available); and finally accessing the respective counter and dealing with the values accordingly.

It is worth noticing that, further developments have been made regarding counter behaviour and privilege filtering under the 2024 ISA Specification [61] [60]. This enhancements improve the PMU customizability but were not taken into consideration due to time constraints. Future work may explore these new functionalities.

3.5 Summary

Recapping, using the RISC-V PMU requires careful consideration to platform-specific implementation details. Manually configuring the PMU requires the user to be familiarised with both the platform, operating systems implementation details and also the respective ISA specifications implemented by the platform. By accessing and configuring the RISC-V PMU, it is possible to develop a solution that answers both **RQ1** and **RQ2**. In fact, directly retrieving values from the RISC-V PMU CSRs, and performing the necessary configurations, allows the development of a solution that can compute different performance metrics for the systems, independent of the running applications and operating systems. If the values are handled accordingly, it allows the extraction of performance metrics for real-time applications (**RQ1**) and by configuring the PMU to track different metrics based on specific events, allows the computation of different metrics based on the chosen configuration (**RQ2**).

However, the configuration process may also be challenging as it usually involves directly interacting with the RISC-V PMU using the *Zicsr* extension instructions. Incorporating

these instruction on the target application would usually require the use of RISC-V Assembly Language, since instructions are not directly mapped to high-level programming languages. Although some approaches may use the system SBI calls, the lack of a standard definition for an SBI in all systems, and its dependence on the specific target system, means that users may need to use different SBIs in different systems.

Using the *Zicsr* extension instructions to access and modify CSR values from the PMU, defined on the *Counters* extension, enables the development of a solution capable of performing performance monitoring capabilities. These two extensions - *Zicsr* for the instructions to read and write values on the counters, and *Counters* for defining the counters available on the PMU - represent the minimum number of standardized extensions needed to develop a solution for performance monitoring in RISC-V, addressing the **RQ4**. Since these extensions can be included in any RISC-V system, the solution could then be deployed to any system that provides at least those two extensions, suggesting a potential answer address portability in **RQ5**.

Developing a solution that abstracts the user from these complex and diverse implementation details under a unified interface, would significantly simplify the process to retrieve performance metrics from the system. The next chapter, Chapter 4, a solution designed to assist the user in configuring and using the RISC-V system PMU is presented.

Chapter 4

RISC-V Performance Counters API

This chapter presents the design, development, and implementation of the RISC-V Performance Counters API, a specialized tool built to facilitate the configuration and retrieval of metrics directly from the hardware performance counters in RISC-V systems.

This solution was designed to be programmatically intuitive, and developed based on the specifications provided by the Unprivileged RISC-V ISA from 2019 [10] and the Privileged ISA from 2021 [9], as these were the latest available at the time the API development began. In 2024, new specifications were ratified introducing additional features, addressing known problems and providing clarifications on certain topics. However, due to their late release, the limited time available for this thesis's development, and the absence of hardware supporting these new specifications, the primary source of reference remained the previously mentioned versions.

Accessing and configuring the Hardware Performance Monitoring features on a RISC-V system can be somewhat complex, as it requires interacting with specific CSRs. This requires that the developer understands how the ISA defines the features, while also takes into consideration platform-specific features and configurations. Using an API to simplify this configuration and access process, developers can more easily access and retrieve performance metrics from the system.

This chapter explores the API's design principles, core functionalities, and the implementation strategies employed. The subsequent subsections discuss:

- **API Architecture:** A detailed overview of the API's structure, focusing on its primary components and how they interact within a RISC-V environment.
- **Core Functionalities:** A detailed description of the API's essential functions, including how it enables the configuration and access of hardware performance counters and related events.
- **Configuration Capabilities:** An explanation of the API's customization options, highlighting how users can tailor the API for use on different RISC-V platforms with various requirements and resources.
- **Operations**: A closer look at the API's operational features, including the processes involved in interacting with and utilizing performance counters in real-time.
- **Implementation Details:** A review of the API's development, including insights into the coding process, challenges faced during implementation, and the solutions applied to overcome those issues.

By the end of this chapter, readers should have a clear overview of the RISC-V Performance Counter API, its capabilities, and its potential applications in RISC-V system performance analysis.

4.1 API Architecture

The API was developed with portability and modularity in mind, following the same approach as the RISC-V ISA. Depending on the platform's specific needs, it allows the user to include only the components necessary for the required features.

At the moment of writing, this API is composed of the following components that are present in Figure 4.1:

- MONITOR COMPONENT: Contains the core functionalities of the API.
- CONFIGURATION COMPONENT: Manages API's configuration and controls which features are enabled based on the platform needs.
- MEMORY COMPONENT: Handles all memory-related functions within the API, particularly the ones referencing to the heap management and allocation. Since memory allocation is usually performed via system calls to the underlying operating system, this component allows the API to configure the memory operations used throughout the whole API on the system. This reduces the API's dependency on the operating system by requiring only changes to the memory component.
- PLATFORM SPECIFIC CONFIGURATION COMPONENT: Aggregates the target platform's functionalities and configurations. By selecting the appropriate platform component, the API can be configured based on the platform's available features. This simplifies the process of porting the API across various system with different requirements and functionalities, such as custom extensions unique to a specific platform. In these cases, the usage of this extensions requires specific configuration steps that require the analysis of the platform's manuals.

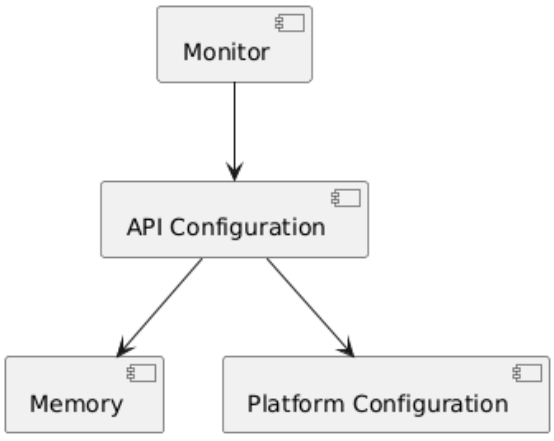


Figure 4.1: API Components

4.2 Core functionalities

The current API is still a draft and does not yet include all functionalities. However, the following features have been developed:

- Retrieval of performance counter values directly from hardware performance counters.
- Defining monitoring points on the target code.
- Configuration of hardware performance counters.

4.2.1 Retrieve performance counter values

To retrieve hardware performance counter values, the API makes use of the *Zicsr* extension. By using the *Zicsr* instruction with the CSR-mapped addresses for the respective HPM counters, a solution was implemented that can access and manipulate these CSRs. This approach can be easily extended throughout configuration files.

Since all HPM counters are CSRs, they must be mapped under the CSR space, on Appendix A, the mapping address mapping performed for the ISA standard CSRs is available. The API uses the *CSRRS* instruction, which performs an atomic read at the specified address, allowing the value present on the counter to be retrieved. In order to use this instruction, it is necessary to make usage of RISC-V Assembly, as these instructions are not mapped under any C instructions. To facilitate the usage on the API, a macro containing the asm instruction was included on the API.

```
1 #define READ_CSR(csr, result) ({ asm volatile("csrr %0, %1" : "=r"(
      result) : "i"(csr)); })
```

Listing 4.1: Macro definition to read CSRs.

RISC-V is known for its portability and versatility, allowing manufacturers to choose which extensions to implement on their platform. Furthermore, manufacturers may even create custom extensions to expand the platform capabilities if necessary. Since performance monitoring mechanisms in the RISC-V ISA are somewhat limited when compared to more mature proprietary ISAs, it may be beneficial for a manufacturer wants to develop their own extensions in order to enhance monitoring functionalities.

Taking into consideration the capability of customizing the RISC-V ISA, By directly utilizing *Zicsr* instructions, the API is designed to facilitate expansions that enable the API to use custom extensions when required.

One such example is the custom extension defined on the ESP32C6 board. This extension defines three new custom registers that can be used for performance monitoring related operations. By mapping the new addresses on new macros the user would be capable of using the already existing features to access the values on these new registers.

```
1 #define CSR_ADDRESS_mpcer 0x7E0
2 #define CSR_ADDRESS_mpcmr 0x7E1
3 #define CSR_ADDRESS_mpccr 0x7E2
```

Listing 4.2: Macro definition for new custom CSR addresses.

A simple example in how to retrieve the values from the hardware performance counters, using the *CSRSS* instruction from *Zicsr* extension can be found in the following listing.

```
uint32_t mpccr_value = 0, mcycles_value = 0, mcyclesh_value = 0;
READ_CSR(CSR_ADDRESS_mpccr, mcycles_value);
READ_CSR(MPFC_CYCLE, mcycles_value);
READ_CSR(MPFC_CYCLEH, mcyclesh_value);
```

Listing 4.3: C code example to retrieve HPM values using Zicsr extension.

Note that on that listing we are accessing both the upper and lower bits of the respective hardware performance counter when using the ISA *mcycles* register, while only accessing the *mpccr* register once. This is due to the ISA counters having a 64-bit length while the custom extension only considers a 32-bit length register as the ESP32C6 board is a 32-bit system. This is also a concern the user needs to take into consideration when retrieving values from the hardware performance counters.

4.2.2 Start-End points monitoring

To implement this feature, the API begins by retrieving the value of selected HPM counters at the starting point, and does the same thing at the ending point. The difference (delta) between these two values provides metrics regarding what occurred between the two defined points. This is achieved by using the previously defined macros for both *CSRSS* instruction and CSR addresses mapped, a code example can be observed in the following listing.

```
uint32_t mpccr_value_start = 0, mpccr_value_end = 0, mpccr_value_delta =
    0;
/// Starting point read
READ_CSR(CSR_ADDRESS_mpccr, mcycles_value_start);
///Code to evaluate
...
READ_CSR(CSR_ADDRESS_mpccr, mcycles_value_end);
mpccr_value_delta = mpccr_value_end - mpccr_value_start;
```

Listing 4.4: C code example to perform Start-End monitoring.

When analysing the previous code, it can be noticed that for each counter to be read a new *CSRSS* instruction needs to be performed. This can be easily managed if we are only taking into consideration a single counter value such as in the example, but when considering multiple counters being retrieved at the same time, manually adding the instructions for each counter may be a time-consuming task.

Furthermore, there are certain implementation-details that need to be taken into consideration during the result analysis, one example is that when the system has a 32-bit length core, the respective counter needs to be accessed twice, one for 32 upper bits and another for the 32 lower bits. Depending on the implementation details of the platform, usually having a different value on the upper bits would represent that an overflow has happened on the counter. Dealing with these situations may be troublesome as the user needs to be familiar with concrete implementation details for each platform.

In the future, it would be beneficial to expand the number of monitoring points that can be defined, allowing for the tracking of the HPM counter in multiple points of the application, while managing the different values automatically, depending on the user configuration. This enhancement would make the API's performance monitoring capabilities more robust and enable the retrieval of more complex metrics.

4.2.3 Hardware Performance Counters Configuration

Configuring the hardware performance counter requires the user to set some values on a specific CSR. To achieve this, the API makes use of the *Zicsr* extension. By using the *CSRW* instruction with the CSR-mapped addresses for the respective HPM counters, a solution was implemented that accesses the respective CSR and writes the appropriate values to the specific CSR.

To facilitate this process, some macros a macro containing the respective asm instruction were included on the API.

```
1 #define WRITE_CSR(reg, val) ({ asm volatile("csrw %0, %1" ::"i"(reg), "
    rK"(val)); })
```

Listing 4.5: Macro definition to manipulate CSRs.

In order to write a value to a specific CSR, the macro can be used according to the following example.

```
1 uint32_t value = 0x1;
2 WRITE_CSR(CSR_ADDRESS_mpcmr, value);
```

Listing 4.6: Code example to write to a CSR.

According to this example, the value *0x1* will be written to the CSR mapped to the *CSR_ADDRESS_mpccr* mapped address, which in this case corresponds to the *mpcmr* register. According to the *mpcmr* register implementation details, writing the value *0x1* should enable the *mpccr* register as it would set the *COUNT_EN* bit to 1. However, as the register being dealt with contains reserved bits, setting the register value to *0x1* could be change some of the reserved bits values, which might cause an illegal-exception to be thrown or cause undefined behaviours.

To mitigate this problem, when interacting with registers with reserved bits, the user should only make changes on the bits he is entitled to write. Considering this, the user should first retrieve the current value of the target CSR and then perform bit-wise operations to set and reset the corresponding bits.

To aid in this process, macros to set or reset bits on variables were also created.

```
1 #define SET_BIT(var, bit) ({ var = ((1 << bit) | (var)); })
2 #define RESET_BIT(var, bit) ({ var = (~(1 << bit) & (var)); })
```

Listing 4.7: Macros to set or reset a specific bit on a variable.

Taking all of this into consideration, a correct approach to set the *mpcmr* register can be found on the following code example.

```
1 uint32_t mpcmr_value = 0;
2 READ_CSR(CSR_ADDRESS_mpcmr, mpcmr_value);
3 SET_BIT(mpcmr_value, COUNT_EN);
4 RESET_BIT(mpcmr_value, COUNT_SAT);
5 WRITE_CSR(CSR_ADDRESS_mpcmr, mpcmr_value);
```

Listing 4.8: Code example to configure a CSR.

This code example ensures that the user is only updating the allowed bits, while setting *COUNT_EN* and resetting *COUNT_SAT*, which will cause the *mpccr* counter to increment and overflow when it reaches the maximum value allowed by a 32-bit variable.

This approach poses the same problem as the reading operations, while managing a single CSR would not be that complicated, if multiple CSR values need to be updated, specially if considering that usually a register may need to be enabled, have its accessibility controlled and event configuration performed. Manually performing these configurations would pose to be a time-consuming and error-prone task due to specific implementation details and reserved bits considerations.

4.3 Configuration

The API was designed with customization, versatility, and future expansion in mind. To achieve this, it provides multiple configuration's options based on the platform being used, making it easier to port across different platforms. Furthermore, the API also allows for selective inclusion of functionalities, enhancing versatility by enabling users to decide which features to include based on the system requirements. By including only the necessary features, it can reduce the system resource consumed by the API, which is especially critical for resource-constrained platforms.

4.3.1 Static configuration

Configuring the API is a key feature, as it enables the user to configure the API functionality based on their specific requirement, ensuring that only the required features are utilized. The following variables have been defined and can be configured by the user:

- *configSYSTEM_PLATFORM* - This variable is responsible to informing the API which platform is currently in use.
  - 0 - RISC-V ISA.
  - x - further platforms would have incremental ids.
- *configEVENTS* - This variable informs the API whether the platform supports event configuration, and wheter it follows the ISA or is configured through a custom extension.
  - 0 - No event configuration available.
  - 1 - RISC-V ISA event configuration.
  - 2 - Custom event configuration.
- *configSUPPORTED_PRIVILEGE_MODES* - This variable keeps track of which privilege modes are implemented in the system.
  - 1 - Machine.
  - 2 - Machine and User.
  - 3 - Machine, Supervisor and User.
- *configHEAP_ALLOCATION_ALLOWED* - This variable is responsible for configuring whether the API should allow heap allocation or not.
  - 0 - Not allowed.
  - 1 - Allowed.

- *configNUM_HPMCOUNTERS_REGISTERS* - This variable tracks the number of *hpmcounter* registers in the system.
- *configNUM_HPMEVENT_REGISTERS* - This variable tracks the number of *hpmevent* registers in the system.
- *configNUM_HPMCUSTOM_REGISTERS* - This variable tracks the number of *hpm* custom registers in the system.
- *configNUM_EVENTS* - This variable keeps track of the number of platform-specific events defined by the system that can be used to increment the counters.
- *configHPM_COUNTINHIBIT* - This variable informs the API whether counter inhibition feature is implemented or not.
  - 0 - Not implemented.
  - 1 - Implemented.
- *configHPM_COUNTEREN* - This variable is responsible for informing the API whether counter availability control feature is implemented or not.
  - 0 - Not implemented.
  - 1 - Implemented.

4.3.2 Platform configuration

One important feature of the API is the capability to configure the platform being used on the system. Each platform has its own unique implementation details, and by providing a mechanism to configure the system, the API simplifies the process of retrieving the desired performance metrics from the system.

This configuration defines system's available options and functionalities, such as:

- Base counters and timers available.
- Hardware Performance Monitoring counters available.
- Events and their respective configuration.
- Custom extension configuration.
- Memory management functions.

By tracking these system's configuration, the API can ensure that no operations are performed that might raise illegal exceptions when executed by the API. This is accomplished by preventing the API from utilizing feature that are not configured in the system; with this, if a feature is not configured, it means it is not present on the system.

It is also important to note that the entire API was designed taking into consideration the system's registers length. Currently, it supports both 32-bit and 64-bit addressing. To achieve this, all the structures used across the API use pre-processor macros to ensure compatibility. If *XLEN* - a CSR register that indicated the the register addressing length in the system - is equal to 32, all integer value are stored in *uint32_t* variables. Otherwise, *uint64_t* is used for handling values retrieved directly from hardware registers.

There are two types of configuration available on the API:

- **System configuration** - Defines the available options and functionalities of the system, such as available counter and timers, events, custom counter extensions, and heap management functions.
- **API behaviour configuration** - Controls what happens when using the API. Some configurable options are:
  - Counters to be read.
  - Events associated with each HPM counter.
  - Metrics to be computed.
  - How the information is handled (currently, only printing to the console is supported, but other options, such as cloud saving, might be included in the future).
  - System configuration to be used in the API.

System configuration

System configuration encompasses all the platform's available options and functionalities related to performance monitoring, allowing the configuration of ISA HPM CSRs and their respective settings, as well as custom extensions dedicated to HPM, along with their configurations.

```
typedef struct
{
    uint16_t address;
    reg_type_t type;
    csr_identifier_t id;
} register_cfg_t;
```

Listing 4.9: register_cfg_t struct type.

To configure the registers, the structure *register_cfg_t* was created, defined as follows:

- **address** : The CSR memory-mapped address of the corresponding register. This member is of type *uint16_t* as it refers to the 12-bit encoded CSR space, which supports up to 4096 CSRs.
- **type** : An enum value representing the type of the corresponding register. This member is a value from the enum *reg_type_t*.
- **id** : A unique CSR identifier used to reference the CSR in the API. This member is a value from the enum *csr_identifier_t*.

```
typedef enum
{
    reg_hpmcounter,
    reg_hpmevent,
    reg_countinhibit,
    reg_counteren,
    #ifdef configCUSTOM_REG_TYPE
        CUSTOM_REG_TYPE
    #endif
} reg_type_t;
```

Listing 4.10: reg_type_t enum type.

A CSR may have different associated types. For HPM-related CSRs, the relevant values defined in the RISC-V ISA specification are included on the enum *reg_type_t*. This enum also contains a macro that allows for custom extension-defined types. Currently, the available types are: **hpmcounter**, **hpmevent**, **countihibit** and **counteren**, which correspond to the register described in Section 4.2.3.

```
typedef enum{
    csr_cycle,
    csr_time,
    csr_instret,
    csr_hpmcounter3,
    ...
    csr_hpmcounter31,
    csr_hpmevent3,
    ...
    csr_hpmevent31,
    csr_counterinhibit,
    csr_counteren,
    #if configNUM_HPMCUSTOM_REGISTERS > 0
        CUSTOM_CSRS_ENUM
    #endif
    csr_num_events
} csr_identifier_t;
```

Listing 4.11: csr_identifier_t enum type.

For CSR identification, the enum *csr_identifier_t* was created to hold all the possible values for HPM-related CSRs in the system. Currently, some identifiers have defined for the CSRs described under the RISC-V ISA specifications. However, a macro has also been provided to allow for custom IDs from platform-dependent custom extensions.

```
typedef struct
{
    #if configNUM_HPMCOUNTERS_REGISTERS > 0
    register_cfg_t hpm_counters[configNUM_HPMCOUNTERS_REGISTERS];
    #if __riscv_xlen == 32
    register_cfg_t upper_hpm_counters[configNUM_HPMCOUNTERS_REGISTERS];
    #endif
    #endif
    #if configNUM_HPMEVENT_REGISTERS > 0
    register_cfg_t hpm_events[configNUM_HPMEVENT_REGISTERS];
    #endif
    #if configNUM_HPMCUSTOM_REGISTERS > 0
    register_cfg_t hpm_custom_csr[configNUM_HPMCUSTOM_REGISTERS];
    #endif
} system_cfg_t;
```

Listing 4.12: system_cfg_t struct type.

The *system_cfg_t* structure is responsible to configure the system itself. It contains information regarding the various registers in the system and also provides details regarding the monitored events and custom CSRs that may be available from platform's custom extensions.

API configuration

In addition to the system variables and configuration, the user can also control how the API behaves. Currently, the following options can be configured:

- Counters to be read.
- Events for each HPM counter.
- How to handle the output(currently, only printing to the console is available).
- System configuration to be applied within the API.

Privilege levels

```
typedef enum
{
    USER = 0x0,
    SUPERVISOR = 0x1,
    MACHINE = 0x3,
} privilege_levels_t;
```

Listing 4.13: privilege_levels_t enum type.

The enum *privilege_levels_t* contains the available options for privilege levels on the running platform. Currently, it holds the values defined in [9] for RISC-V privilege levels: **USER**; **SUPERVISOR**; and **MACHINE**, each corresponding to the respective values that are assigned on the ISA.

Event configuration

The API currently takes into consideration the wide range of possible extensions available on the system. Depending on the implementation details of the event configuration, it provides different options, structures and variables to be used within the API. This differences are controlled by the static variable *configEVENTS*.

```
#if configEVENTS == 0

#define configNUM_HPMEVENT_REGISTERS        0
#define configNUM_EVENTS                    0

/// @typedef typedef for empty event enums
typedef enum {
} events_enum_t;
#elif configEVENTS == 1
#define configNUM_HPMEVENT_REGISTERS        0
#define configNUM_EVENTS                    3

/// @typedef typedef for generic event enums
typedef enum {
    evt_first = evt_cycle,
    evt_cycle,
    evt_time,
    evt_instret,
    evt_last = evt_instret
} events_enum_t;
#endif
```

Listing 4.14: Event configuration.

In case the event configuration is not supported, the number of *hpmeventx* registers (stored in *configNUM_HPMEVENT_REGISTERS*), responsible for configurin the corresponding *hpmcounter*, and the number of events available on the system (stored in

configNUM_EVENTS) are both set to zero. Additionally, the *events_enum_t* enum, which holds the system's available events, remains empty.

If event configuration is supported and follows the RISC-V ISA specification [9], then the system will have only the three base counters and timers, which are: **cycles**, **time** and **instructions-retired**. These events are then defined inn the *events_enum_t* structure. Consequently, the number of events is set to three, and the number of configurable events register is set to zero, as these features are platform-dependent.

Data handling

Users may want to handle the data retrieved from the platform monitoring with different ways. However, due to the early stages of the API development, the only available option at present is to print the data directly to the console. In the future, it is expected that the data will be capable of being transmitted to the cloud or other devices, depending on the system's available components and features. To manage these data handling options, the *data_handling_config_t* enum, which includes the available methods for handling data on the system and API, was defined.

```
typedef enum {
    print_to_console,
} data_handling_config_t;
```

Listing 4.15: data_handling_t enum type.

Heap management

```
typedef struct {
    void* (*api_malloc)(size_t size);
    void* (*api_calloc)(size_t num_elements, size_t size);
    void (*api_free)(void* ptr);
} heap_management_functions_t;
```

Listing 4.16: heap_management_functions_t struct type.

Heap management is highly reliant on the underlying operating system, as different operating system will provide different mechanism and features for heap memory management. To reduce this OS dependency, the *heap_management_functions_t* was created. This structure holds function pointers to the corresponding OS heap memory management functions. By creating a port for each specific supported OS, it becomes possible to use this API across multiple ported operating systems. Currently, this structure maps the **malloc**, **calloc** and **free** operations from the OS to the API.

API config

```
typedef struct {
    privilege_levels_t privilege_mode;
    uint32_t counters_to_read;
    #if configNUM_HPMEVENT_REGISTERS > 0
    events_enum_t event_configuration[configNUM_EVENTS];
    #endif
    #if configNUM_HPMCUSTOM_REGISTERS > 0
    HPM_CUSTOM_CONFIGURATION_SETTINGS
    #endif
    #if configDEFAULT_METRICS_AVAILABLE
    uint32_t metrics_to_be_computed;
    #endif
```

```
13     data_handling_config_t data_config;
14     #if configHEAP_ALLOCATION_ALLOWED
15     heap_management_functions_t heap_func;
16     #endif
17     system_cfg_t sys_cfg;
18 } api_config_t;
```

Listing 4.17: api_cfg_t struct type.

The *api_config_t* structure was design to hold the API configurations for each execution. Multiple members were introduced on the configuring struct, including:

- *privilege_mode* - Holds the API's running privilege mode. Although the API provides an enum for multiple levels, currently, it only supports Machine mode privileges.
- *counter_to_read* - A bitset controlling which counters should be read during each read operation. It has a type of *uint32_t*, as the RISC-V ISA Specification defines that the PMU can have up to 32 general-purpose HPM counters.
- *HPM_CUSTOM_CONFIGURATION_SETTINGS* - A macro for additional custom extension configurations. Only available if custom registers exist.
- *metrics_to_be_computed* - Another bitset, this time for out-of-the-box metrics to be computed. Currently, this is a placeholder, as predefined metrics were not covered under the development of this thesis.
- *data_config* - Configures how the API handles the retrieved data. As previously mentioned, the only available option is printing to the console. In the future, the API could support additional options, such as cloud saving.
- *heap_func* - Contains the heap management functions that will be used by the API. By using the functions in this structure, the API can be configured to use memory-related OS functionalities across different operating systems, achieved by mapping the OS function pointers to this structure.

4.4 Operations

To allow the API to provide its core features, several default operations were defined. These operations are essential for the main functionalities of the API and are the following:

- Initialization [init operation] : Configure the entire API using the respective parameters.
- Counter read [read operation] : Accesses and retrieves the values from the counters that have been configured during the initialization operation;
- Monitoring operations [start/stop]: Captures the respective counter values and saves them to dedicated monitoring variables.
- Delta [delta operation] : Computes the delta between the start and end monitoring points and returns the computed values. Note that this operation computes the delta for all the retrieved counters.
- Print [print operation] : Outputs the monitoring structure depending on the chosen configuration.

4.4.1 Init operation

```
1 char perf_monitor_init(api_config_t conf);
```

Listing 4.18: perf_monitor_init function signature.

This operation takes the values provided in the *api_config_t* structure, passed as the parameter *conf*, and save them in memory for use during the API execution. These configurations cover the following points:

- Counter events;
- Counters to read;
- Counter-Enabling;
- Counter-Inhibition;
- Memory approach for monitoring structures;
- Data handling approach.

4.4.2 Read operation

```
1 char perf_monitor_read(perf_monitor_t*);
```

Listing 4.19: perf_monitor_read function signature.

This operation will retrieve the values of the counters configured at the time it is called and saves them on the structure *perf_monitor_t* provided as parameter.

The counter values are obtained using *Zicsr* extension, specifically the *CSRRS* instruction, which retrieves the value from the counter corresponding to the provided memory address.

4.4.3 Monitoring operations

Currently, two monitoring operations exist, being them, start and stop monitoring.

```
1 void perf_monitor_start();
2 void perf_monitor_stop();
```

Listing 4.20: perf_monitor_start and perf_monitor_stop functions signature.

These function will call the read operation and use dedicated variables to store the corresponding values.

- Start Monitoring - Reads and saves the counter values in a dedicated start monitoring variable;
- End Monitoring - Reads and saves the counter values in a dedicated end monitoring variable;

4.4.4 Delta operation

```
1 char perf_monitor_delta(perf_monitor_t*);
```

Listing 4.21: perf_monitor_delta function signature.

This operation takes the values from the dedicated start and end monitoring variables and computes the delta between the end and start results for each available counter.

The respective result will be return in the monitoring structure provided as a parameter.

4.4.5 Print operation

```
1 char perf_monitor_print(perf_monitor_t* monitor);
```

Listing 4.22: perf_monitor_print function signature.

This operation prints the provided monitoring structure based on the configuration set during the initialization operation.

It might be useful to create a print operation that takes the printing approach as a parameter, and another function that calls this one using the pre-configured approach.

4.5 Implementation details

The described API is still in its draft form, on an early stage of development. It was developed using the C language, which makes it highly portable across various platforms, as most of them natively support C. Additionally, C is a middle-level language that combines low-level hardware access with high-level abstractions, offering both control and efficiency. Moreover, its ability to easily integrate with Assembly instructions made it an ideal choice for developing this API.

4.5.1 RISC-V Extensions

It is important to consider that different system will have varying extensions and registers available. As a result, a performance monitoring API will require different configurations to function across multiple systems. To minimize the effort when porting the API to a different system, a configuration file for each architecture is necessary. This file will contain the HPM counters available and their respective configuration options and extensions available to the platform.

In order to use the API, the target system must have the following standard extensions implemented:

- *Counters* (now split into **Zicntr** and **Zihpm**) : Provides hardware performance counter implementation and features.
- *Zicsr*: used to access Control and Status Registers (CSRs).

When the development was started, the 2024 specification [61] [60] had not yet been released and was therefore not fully considered during both API's and Thesis development New extensions, beyond the previously mentioned *Zicntr* and *Zihpm*, were introduced in the updated ISA to address known problems. These include the standardization of counter

overflow handling and privilege mode filtering in the *Sscofpmf* extension, as well as the *Smcntrpmf* extension, which allows configuring the *cycle* and *instret* counter behaviour for privilege mode filtering.

4.5.2 Operating system independence - Portability

Currently the API, is completely OS independent, making it easily portable to any RISC-V system that includes the required extensions. The only feature that relies on the operating system is memory allocation, which was not fully developed in this thesis. Further details, will be provided in Chapter 5.

4.5.3 Future Expansion

To facilitate future expansions, instead of using the pseudo instruction, *RDCYCLE*, *RDTIME* and *RDINSTRET* to access the base counters and timers, the API uses the *CSRRS* instruction from the *Zicsr* extension with the corresponding memory-mapped addresses for the target hardware performance counter.

This approach requires that all available counters in the system need to be configured beforehand, ensuring that the API can use the appropriate mapped addresses during its execution.

By using the memory address instead of hardcoded register names, the API can also support additional counters provided by custom extensions from manufactures, as long as those counter are mapped within the API configuration.

Custom extensions for performance monitoring will normally include new CSRs used to count specific event metrics. Since the API relies on the memory-mapped addresses, by simply adding these addresses to the API configuration makes them available for use on the API.

Some systems have the events directly mapped to the HPM counters without allowing changes, while others have the *mhpmevent* CSRs to select and control event counting on the associated *hpmcounter* CSR. Additionally, some systems lack the *Zihpm* extension, preventing them from tracking additional custom events beyond the base ones. Furthermore, some systems do not even support the *Zicntr* extension for the base counters and timers. While these extensions where formally ratified in the Unprivileged ISA 2024, a drafted version with similar features was already present in earlier specifications.

4.6 Summary

The API proposed in this chapter aims to provide a wider support for the RISC-V performance monitoring capabilities, especially regarding the 2021 version of the Privileged ISA specification [9] and the 2019 version of the Unprivileged ISA specification[10]. Furthermore, it facilitates the retrieval of performance metrics, by enabling the user to configure the system PMU via the API configuration interface.

The API abstracts some of implementation specific details for both the ISA specifications and custom extensions, if configured, such as mapping all the CSR-addresses for the CSRs related with performance monitoring, provides an interface to select events that will trigger specific general-purpose hardware performance counters, and also enabling or disabling hardware performance counters and controlling their accessibility on different privilege levels. These

changes make the API more portable, as it allows the CSR-addresses to be mapped, reducing the dependencies on the platform itself.

Retrieving and modifying values from the CSRs is done via the *Zicsr*, *CSRSS* and *CSRRW* instructions, for reading and writing respectively. When using the API to write values to the CSRs, the API will take into consideration the reserved bits under the specific CSRs and perform bit-wise operations to ensure that only bits that the user is available to use are modified instead of the whole counter value. Using this instructions, while mapping the CSR-addresses, aids in the API portability as it reduces the dependencies on the systems own mappings. The diverse configuration capabilities also provide some support for this issue, by abstracting system specific implementation details on the API, and allowing the user to expand the API to support more custom extensions or details, aids this approach in achieving **RQ5**, which questions if a solution can be deployed as an API across different RTOS and RISC-V systems. Using these abstractions, allows the solution provided on this chapter to currently be capable of being ported accross different systems and platforms, if the correct porting process is done, more details in this porting is provided in the next chapter.

Chapter 5

System Setup and API Porting Process

This chapter describes technical details of configuring the system environment and porting the Performance Counters API to different RISC-V platforms. Since the API is designed to be flexible and functional across different operating systems and hardware configurations, significant emphasis was placed on ensuring portability and compatibility were given attention throughout the API development.

This chapter will cover the following topics:

- Details of the system environment, including the hardware and software utilized, as well as the role of emulation in the development process.
- Challenges and limitations encountered during the development phase.

5.1 System Environment

The system environment used during development and utilization of an API is crucial to ensure its proper functionality. This section, provides an overview of the hardware platform used during development, followed by a description of the software used, including the usage of QEMU as an emulator for creating a virtual RISC-V environment.

5.1.1 Software Environment

In order to develop the API, a cross-compilation environment was set up. An SDK was installed on the system to enabled the cross-compilation of the application from the host system to the target RISC-V platform.

In the initial phase, QEMU was used for its RISC-V support and ability to emulate performance monitoring related features. This was crucial for prototyping the API, allowing an early testing of basic concepts, such as retrieving of performance counter. As it was the first time experience with the RISC-V systems, becoming familiar with the architecture was essential for supporting the API's development.

Additionally, QEMU enabled the author to experiment with embedding RISC-V assembly instructions directly in C code to facilitate quick access to hardware registers.

During this phase, the QEMU virt machine was used as the target environment for the API. This machine provides a generic virtual RISC-V platform with the core RISC-V system

features. Using this platform as a target allowed testing of API features that are consistent across platforms, such as the usage of the *Zicsr* instructions to access or manipulate CSRs.

However, when it came to testing more specific features, such as configuring events for the HPM counters, some impediments emerged. This was due to the lack of defined event mappings required for the *hpmevent* values. These values are platform-specific, and since the QEMU virt machine is designed as a generic implementation, it lacked these definitions.

Another key consideration regarding the API development was the support for the target operating system. Platform compatibility with an operating system is crucial to ensure a proper system functionality. This also impacts the API development, particularly regarding memory-related features, device integration and other connectivity options, as the OS typically manages the interaction between software and hardware devices.

After evaluating the available operating systems, FreeRTOS was chosen due to its lightweight and modular kernel and its broad support for embedded systems.

5.1.2 Hardware Environment

The hardware environment used to developing and testing the API was the ESP32-C6 [59], a RISC-V-based microcontroller. It was selected for its balance of computational capabilities and its widespread use in embedded systems. Additionally, it supports the selected target operating system, FreeRTOS.

Key specifications [62]:

- **Processor**:
  - HP RISC-V processor with clock speed up to 160 MHz and a four stage pipeline.
  - LP RISC-V processor with clock-speed up to 20 MHz and two stage pipeline.
- **Memory**:
  - *L1 Cache:* 32 KB.
  - *ROM:* 320 KB.
  - *HP SRAM:* 512 KB.
  - *LP SRAM:* 16 KB.
- **Wireless Connectivity**: Supports both Wi-Fi 6 and Bluetooth 5.0 Low Energy (BLE), making it suitable for IoT applications.
- **I/O and Peripherals**: Offers a wide range of GPIOs, UARTs, SPI, and I2C interfaces, providing a flexible environment for embedded development and performance testing.
- **Low Power Modes**: Includes deep sleep modes, which are critical for evaluating the performance of power-constrained applications

As previously mentioned, the ESP32-C6 is built around a High-Performance (HP) and Low-power (LP) RISC-V CPU Cores, which makes it an ideal candidate for testing the API's interaction with the RISC-V PMU. This platform implements the RISC-V ISA's basic PMU capabilities, with hardwired events mapped to their respective *hpmcounter*. However, these counters are only accessible when the core is operates in low-power mode, which is a limiting factor for the system.

In addition, the platform also provides the custom extension previously mentioned in Chapter 3, developed by the manufacturer, that introduces three new registers that can be used in HP-Core to retrieve performance metrics.

Another relevant functionality of the ESP32-C6 is its wireless connectivity, which might aid in testing API-related features, such as cloud storage and Bluetooth data transfer to other devices. The I/O and peripheral support (GPIOs, UARTs ans other interfaces) also helps testing various features commonly used in embedded development.

The board resource constraints also helps in refining the API to be usable on low-resource devices, which may be the primary target for the API. Ensuring that the API operates on this system - without significantly affecting the overall performance, despite its memory an processing limitations - helps making the API more robust and reliable.

The API was developed using the Espressif IoT Develpment Framework (ESP-IDF), which includes support for flashing and debugging directly on the ESP32-C6 board. This required adapting the compilation environment, as the API was initially developed on an emulated system using a locally installed SDK for cross-compilation. ESP-IDF installations provides a separate installation that includes all the required tools for development

For debugging the applications, JTAG was used for real-time monitoring of the hardware, enabling the retrieval of *hpmcounter* values, while validating the API's functionality.

5.2 API Porting

Taking into consideration that the API was developed while using minimal dependencies, any platform-specific or OS-dependent features must have corresponding porting configurations. This design approach ensures that the API can be easily ported to different platforms and operating systems with minimal configuration efforts.

5.2.1 Operating System Support

Considering the currently available features of the API, the only OS-dependent functionality is memory management. To minimize dependency on a specific Operating System for handling memory management, a structure containing function pointer for memory related operations was created. These function should be used throughout the API whenever a memory-related function is performed. This setup is defined in the API's configuration file, and if heap allocation is enabled, the corresponding functions for memory allocation, reallocation and deallocation must be provided within the appropriate configuration structure member.

Currently, the only OS-dependent feature in the system is the heap management functionality. To demonstrate how porting is done, the target operating system was set to *FreeRTOS*, as it is a lightweight operating system.

In order to configure the FreeRTOS port, the following modifications were made:

- The macro $configFREE\_RTOS$ was added to the $api\_config.h$ file and set to 1 (enabled).
- The following modifications were made to the $api\_memory.h$ file, which contains all memory-related functions.

```
#ifdef configFREE_RTOS
#include "freertos/FreeRTOS.h"
#endif

#ifdef configFREE_RTOS and !(HEAP_FUNCTIONS)
static heap_management_functions_t heap_func = {
    .api_malloc = pvPortMalloc,
    .api_calloc = pvPortCalloc,
    .api_free = vPortFree
}
#define HEAP_FUNCTIONS heap_func
#endif
```

Listing 5.1: Updated api_memory_h file to include FreeRTOS configuration.

These changes include the *FreeRTOS* header file if *FreeRTOS* is configured and map the corresponding heap management functions to the specific structure. A macro is also defined to enable the use of these heap functions across the entire application. It is important to note that, depending on the heap management mode selected during *FreeRTOS* configuration, some memory-related operations, such as, *calloc* function may not be available. Therefore, the structure should be updated accordingly based on the *FreeRTOS* memory configuration.

With these minimal changes, the API can now use heap management functions within *FreeRTOS*. If different OS is configured, the corresponding heap management functions for that OS would be used.

5.2.2 Target System Architecture Considerations

To demonstrate the process of porting the API to a target platform, the ESP32-C6 board was selected, as it was used to test the API's functionalities. To achieve this, some changes were introduced to the API. The first step was to map the value 1 on the $configSYSTEM\_PLATFORM$ variable to be correspondent to ESP32-C6 platform. When this variable is enabled, the ESP32-C6 configuration file will be included on the API. This file, newly created under *platforms/esp32c6conf.h*, contains all the information related with this platform.

After this, the number of registers was configured on the API:

```
#define configNUM_HPMCOUNTERS_REGISTERS     13
#define configNUM_HPMEVENT_REGISTERS        0
#define configNUM_HPMCUSTOM_REGISTERS       3
#define configNUM_EVENTS                    13
#define configHPM_COUNTINHIBIT              1
#define configHPM_COUNTEREN                 0
```

Listing 5.2: Counter's configuration variables.

Another necessary step is to configure the events available on the system. Since the *hpm-counters* in the ESP32-C6 board are platform-specific, we need to created the enum responsible for the event configuration. The variable that tracks whether the configuration is platform specific is $configEVENTS$, and this variable needs to be set to 2. With this, the following changes were performed:

```
#if configEVENTS == 2

```

```
/// @typedef typedef for esp32c6 event enums
typedef enum {
    evt_cycle,
    evt_time,
    evt_instret,
    evt_wcycles_inst,
    evt_wcycles_mema,
    evt_wcycles_meminst,
    evt_mem_read_ret,
    evt_mem_write_ret,
    evt_ujmp_ret,
    evt_branch_ret,
    evt_branch_taken_ret,
    evt_comp_ret,
    evt_wcycle_mul,
    evt_wcycle_div,
    evt_last = evt_wcycle_div
} events_enum_t;

#endif
```

Listing 5.3: esp32c6 events enum.

Additionally, as this board includes a custom extension for performance monitoring, we need to add the respective configuration. This can be done with following code:

```
typedef enum {
    custom_evt_cycle,
    custom_evt_inst,
    custom_evt_ld_hazard,
    custom_evt_jmp_hazard,
    custom_evt_idle,
    custom_evt_load,
    custom_evt_store,
    custom_evt_jmp_uncond,
    custom_evt_branch,
    custom_evt_branch_taken,
    custom_evt_inst_comp
} custom_csr_events_t;

#define CUSTOM_CSRS_ENUM mpcer, mpcmr, mpccr,
#define CSR_ADDRESS_mpcer 0x7E0
#define CSR_ADDRESS_mpcmr 0x7E1
#define CSR_ADDRESS_mpccr 0x7E2

#define HPM_CUSTOM_CONFIGURATION_SETTINGS core_mode_t core_mode;
    uint32_t mpcer; uint32_t mpcmr;

#define configCUSTOM_REG_TYPE 1
#define CUSTOM_REG_TYPE reg_config_en,

typedef enum {
    COUNT_EN = 0,
    COUNT_SAT
} mpcmr_t;
```

Listing 5.4: Custom CSRs extension related configuration.

Furthermore, since the PMU functionality depends on the current CPU mode, the available CPU states were also mapped under the enum *core_mode_t*.

```
typedef enum {
    hp_core,
    lp_core,
    hp_core_w_lp_co_proccess
} core_mode_t;
```

Listing 5.5: core_mode_t enum type.

Finally, the system configuration was completed by mapping the corresponding registers, within the CSR addressing-space. This ensures that the API can interact with the relevant hardware registers.

```
#ifndef configSYSTEM_CFG
#define configSYSTEM_CFG  {                     \
    .hpm_counters = {                           \
        {                                       \
            .address = MPFC_CYCLE,              \
            .type = reg_hpmcounter,             \
            .id = csr_cycle                     \
        },                                      \
        {                                       \
            .address = 0x1808,                  \
            .type = reg_hpmcounter,             \
            .id = csr_time                      \
        },                                      \
        {                                       \
            .address = MPFC_INSTRET,            \
            .type = reg_hpmcounter,             \
            .id = csr_instret                   \
        },                                      \
        ...                                      \
    },                                           \
    .upper_hpm_counters = {                     \
        {                                       \
            .address = MPFC_CYCLEH,             \
            .type = reg_hpmcounter,             \
            .id = csr_cycle                     \
        },                                      \
        {                                       \
            .address = 0x1828,                  \
            .type = reg_hpmcounter,             \
            .id = csr_time                      \
        },                                      \
        {                                       \
            .address = MPFC_INSTRETH,           \
            .type = reg_hpmcounter,             \
            .id = csr_instret                   \
        },                                      \
        ...                                      \
    },                                          \
    .hpm_custom_csr = {                         \
        {                                       \
            .address = CSR_ADDRESS_mpcer,       \
            .type = reg_hpmevent,               \
            .id = mpcer                         \
        },                                      \
        {                                       \
```

```
46              .address = CSR_ADDRESS_mpcmr,    \
47              .type = reg_config_en,           \
48              .id = mpcmr                      \
49          },                                   \
50          {                                    \
51              .address = CSR_ADDRESS_mpccr,    \
52              .type = reg_hpmcounter,          \
53              .id = mpccr                      \
54          }                                    \
55      }                                        \
56  }
57  #endif
```

Listing 5.6: esp32c6 system's registers configuration and mapping.

The complete configuration can be found on Appendix B.

5.2.3 QEMU vs Real Hardware

Using QEMU offers both advantages and disadvantages during development. As a fully emulated system, it allows the users to completely customize the target platform, enabling the testing of multiple functionalities that may not yet have actual hardware support at the moment, but are already defined in the ISA specification. This is particularly beneficial for early prototyping of future features designed for a specific purpose. It is specially useful for performance monitoring-related features, which are often not fully supported across all systems and may be underdeveloped on certain platforms.

Moreover, performance monitoring on RISC-V has been having different advancements across the various ISA specifications versions, with new extensions introduced in later releases that do not yet have to corresponding hardware support. Emulated system like QEMU make it possible to test these features, provided they are properly emulated by the emulator.

However, since the purpose of this development is performance monitoring, it is challenging to verify if the values retrieved from the *hpmcounter* are accurate. Firstly, the emulation's implementation might differ from the actual hardware implementation. Other than this, there is no guarantee that the retrieved values are not being impacted by the different emulation layers on the system, which can become an issue, especially when dealing with timing -sensitive or events-specific monitoring.

A balanced development approach would be to use of both physical and emulated hardware platforms to complement the API development, ensuring a more robust and accurate testing process.

5.2.4 Lessons learnt

When first attempting to access performance-related features on the generic *virt* platform in QEMU, multiple illegal exceptions were thrown by the QEMU system. Debugging this issue was challenging, and it was eventually discovered that the event configuration was not mapped for the current target machine. Since the event configuration was undefined, attempts to write to the respective *hpmevent* registers resulted in these exceptions.

To address this, other target platforms were used to test the relevant API features. However, some additional configurations were missing in the QEMU emulator, which became an impediment for the API development.

As a solution, a physical hardware platform was acquired and used for testing, ensuring the proper functionality of the API.

This experience highlighted both the advantages and disadvantages of using an emulated system. Despite some functionalities might not be fully implemented or supported in emulation, the configuration process might be troublesome and delay the development. However, using a generic emulated system to test common API features allows for early prototyping, while real hardware can be used later to validate platform-specific functionalities.

Hardware-Specific Considerations

There are some hardware-specific implementation details that can have a significant impact on API development. One such example was retrieving performance values from the CPU depending on the respective CPU mode. Coming into contact with actual embedded systems development revealed that the choice of CPU hardware architecture can significantly impact software development.

The ESP32-C6 platform features a High-Performance CPU (HP-CPU) and a Low-Power CPU (LP-CPU), which interchange to handle different tasks efficiently and manage system resources. Despite both being RISC-V CPUs on the same platform, they implement different RISC-V extensions and provide different functionalities.

For instance, the High-Performance CPU includes custom extensions for performance monitoring, while the Low-Power CPU follows the ISA standard specification for event handling, but with a hard-wired configuration. These differences in handling the same functionality posed significant challenges during the API developing. Encountering custom extensions to handle performance monitoring, required refactoring the API architecture, which initially caused delays. However, this also resulted in a more robust API, now capable of being configured to support custom extensions depending on the target platform being used.

Custom CSR Usage

Despite having custom CSRs available on the API, it is the user's responsibility to ensure that they are being used only when the CPU is running on the respective power mode where these CSRs are accessible. Further developments might take this into consideration, incluing run-time checks to prevent the user from accessing unavailable CSRs. Currently, only compile-time verification ensures that certain features are not available on the API if they are not configured for current the target platform. These run-time verification could also be expanded to included additional API's features, such as only enabling cloud saving when an active network connection is present, rather than solely checking if the target platform has the required connectivity capabilities (Note that this feature is currently no available).

OS-Related configurations

When dealing with OS-related features, the wide variety of configurations usually available in a kernel can greatly impact of the applications running on top of it. Such an example is memory-management-related features. Depending on the kernel configuration, different memory management features may or may not be available. For instance, in FreeRTOS, depending on the heap mode chosen, the *calloc* function might not be available. If the application attempts to use *calloc* in this scenario, an error would be thrown. Making

sure that the OS configuration align with the API features is crucial to ensure its correct functionality. By implementing a configuration file on the API that tracks available OS features, the API becomes more versatile. This also enables support for different target operating systems, enabling the API to consider the different configurations within the OS.

Future Considerations

Currently, the API does not heavily rely on OS-related features, aside from a simple memory management porting mechanism used to test the concept. However, future API functionalities API may require closer interaction with the OS. Proper synchronization between the OS and API configuration could reduce the errors encountered when using the API. For example, if the API needs to access devices, it is crucial that the API is aware of the respective OS configuration for handling devices and the specific mapping used for those devices on the system.

5.3 Summary

Recapping, to facilitate the development process of this thesis, both hardware and emulators were used for the API development. Initially, the API was developed and tested under the QEMU emulator, under a generic RISC-V implementation. This proportioned an environment that was similar to almost all the RISC-V systems, at least in terms of core basic functionalities. However, when considering the retrieval of performance monitoring, some gaps were detected, such as the lack of implementation details for the general-purpose hardware performance counters, and their accuracy.

It was still possible to use the QEMU system to emulate a specific board, however, by doing that the API would be tested against an emulated environment, which might introduce variables that affect both performance and the API functionalities, affecting the evaluation process. Other than this, some errors were also encountered when trying to access the hardware performance counters on the QEMU emulated system, causing illegal exceptions to be thrown in some cases, and in other complete deadlock of the system, probably due to miss configurations.

To continue with the API development, a RISC-V board was chosen, ESP32C6. This platform features a High-Performance core, which has its own custom extension to deal with performance monitoring, and also a Low-Power Core that implements the RISC-V ISA specification, including the three base counters and timers, *mcycles*, *mtime* and *minstret*, that track the number of counter cycles, time ticks and instructions-retired, respectively, additionally, it also defines ten general-purpose counters that have hard-wired events that will cause the respective counter to increment when they happen, more details on these counters can be found in Table 6.1 in the next chapter.

Regarding the custom extension for the HP-Core, it adds three new custom registers, one that functions as a hardware performance counter, *mpccr*, another one that is responsible for configuring the counter enabling status and the overflow behaviour, *mpcmr*, and lastly, *mpcer*, which defines the event that will cause the custom counter to increment. This extension follows the same approach defined under the Privileged ISA specification for event selection and configuration, however, instead of being applied to a wider range of counters, that may range from *hpmcounter3* to *hpmcounter32* on the RISC-V ISA, it only works with a single custom hardware performance counter. This simplification may help reduce the

costs of production for the CPU, while also reducing the overhead generated by having a single counter tracking specific events instead of multiple counters tracking different events.

Incorporating this board under the solution development in this thesis, allowed for an improvement of the API portability, as it required refactoring the implementation to support new custom extensions, making the API more robust and versatile. Furthermore, during the whole development was made under the FreeRTOS operating system, all the functionalities that required operating system call, where abstracted for use on the API. By making usage of the API ported functionalities, it is possible to port the API across different systems, with the prerequisite that all the needed functionalities are already mapped under the API functions. For the moment, the only feature that required porting from the operating system was memory management operations, and as this functionality was not highly relevant for the API current development phase, it is not used for the moment.

This chapter partially answers the **RQ5**, by performing the specified porting process and the respective configurations, the solution defined in Chapter 4 can be rapidly ported to other platforms and execution environments..

Chapter 6

Analysis and Demonstration of the API

This chapter focuses on testing the core functionalities of the API, particularly its ability to retrieve performance monitoring values and configure the respective events associated with each *hpmcounter*. Additionally, the overhead introduced by the API and the memory requirements during its operation will also be analysed. By conducting thorough tests in real hardware, this chapter highlights into the API's strengths and limitations, particularly in terms of the API portability between environments.

6.1 How to use the API

To use the API, the file *"riscv_monitor.h"* must be included. This file will include the API required dependencies based on the configurations set on the *"riscv_configuration.h"* file. Within the configuration file, there are some static variables that can be defined to control and customize various features available on the API.

Upon including the file, the initialization function must be called with the corresponding API configuration. This allows the API to function as per the configurations set. After initialization, users can either use the monitoring functions - defining the starting and ending point for monitoring - or execute read operations to retrieve the HPM counters at specific points int the code base.

After retrieving the respective monitoring values, the user are responsible for handling them in order to process the data they need. Currently, the API does not providing any predefined metric computation. However, adding this capability in the future would enhance the API so that some out-of-the-box metrics could be computed by the API automatically when using monitoring and read operations.

6.1.1 API Interface

As previously mentioned, to use the API we need to include the *riscv_monitor.h* file. This file contains the dependencies necessary for the API execution and also the available functions and macros to use for performance monitoring.

Regarding macros, some macros containing the return values available on the API were defined, being them:

```
1 #define PERF_CSR_NOT_ACTIVE -4
2 #define PERF_ERROR_MEM_NOT_VALID -3
```

```
#define PERF_ERROR_MEM_NOT_INIT -2;
#define PERF_ERROR_MEM_ALLOCATION -1;
#define PERF_SUCCESS 1;
```

Listing 6.1: Macro definition for API return values.

Additionally, some macros that perform access to the performance counters using the *Zicsr* extension instructions were also introduced.

```
#define READ_CSR(csr, result) ({ asm volatile("csrr %0, %1" : "=r"(
    result) : "i"(csr)); })
#define WRITE_CSR_NAME(reg, val) ({ asm volatile("csrw " #reg ", %0" ::"
    rK"(val)); })
#define WRITE_CSR(reg, val) ({ asm volatile("csrw %0, %1" ::"i"(reg), "
    rK"(val)); })
```

Listing 6.2: Macto definition to manipulate CSRs.

Finally, macros to set or reset bits on variables were also created. These macros are useful to manipulate specific bits. Usually used for event or HPM enabling/disabling.

```
#define SET_BIT(var, bit) ({ var = ((1 << bit) | (var)); })
#define RESET_BIT(var, bit) ({ var = (~(1 << bit) & (var)); })
```

Listing 6.3: Macros to set or reset a specific bit on a variable.

Regarding the functions available, the API contains the following ones:

```
/// Initializes and configures the API functionalities.
char perf_monitor_init(api_config_t conf);

/// Start performance monitoring
void perf_monitor_start();

/// Stop performance monitoring
void perf_monitor_stop();

/// Read performance monitoring values
char perf_monitor_read(perf_monitor_t *);
char perf_monitor_readc(uint32_t, perf_monitor_t *);
char perf_read_config_csr(counter_t*);

/// Set config values on CSRs
char perf_set_config_csr(counter_t*);

/// Performs delta operations between start and stop monitoring points
char perf_monitor_delta(perf_monitor_t *);

/// Prints values
char perf_monitor_print(perf_monitor_t *monitor);
```

Listing 6.4: API common interface.

Some auxiliary functions have also been created that provide additional functionalities on the API. Some of them allow the user to see which CSRs are active, this uses the active bits on the corresponding bitset. Other than this, there are functions that aid the user in reading specific counters or setting specific events on a respective counter.

```
uint32_t _isActive(csr_identifier_t);
uint32_t _isActivec(uint32_t, csr_identifier_t);
char _perf_read_csr(counter_t *);
char _perf_read_csrc(uint32_t, counter_t *);
char _perf_read_custom_csr(counter_t *);
char _perf_read_custom_csrc(uint32_t, counter_t *);
char _perf_set_csr(csr_identifier_t, uint32_t);
char _perf_set_csrc(uint32_t, csr_identifier_t, uint32_t);
char _perf_set_custom_csr(csr_identifier_t, int32_t);
char _perf_set_custom_csrc(csr_identifier_t, int32_t);
```

Listing 6.5: API auxiliary functions interface.

Finally, some functions that shall be used for memory related operations, such as allocation and dealocation, were also added.

```
#if configHEAP_ALLOCATION_ALLOWED
perf_monitor_t *_perf_monitor_init();
perf_monitor_t *_perf_monitor_init_num(int);
void _perf_monitor_free(perf_monitor_t *);
void perf_monitor_free();
#endif
```

Listing 6.6: Heap related functionalities on API interface.

6.1.2 Example Use Cases

Following this, some example use cases and the respective implementation and results will be presented. This example will cover the basic features of the API, starting with the API configuration, then Start-End monitoring and Single point monitoring.

Use Case 1 - API Configuration

The following example considers how to set up the API configuration on the ESP32C6 board in different scenarios.

The first scenario considers the API execution under the HP-Core. As previously mentioned, the ESP32C6 HP-Core has its own custom CSR extension for performance monitoring. It defined three new registers:

- **mpccr** - A custom machine performance counter register.
- **mpcmr** - A custom register used to configure counter saturation behaviour (overflow or halt in maximum value) and enabling **mpccr**.
- **mpcer** - A custom event selector register for the **mpccr** performance counter.

Taking this into consideration, the first step is to define which events will be used to trigger the performance counter, in this case, "Wait cycles for multiplications and divisions were considered", they are mapped to the enum event "*evt_wcycle_mul* and *evt_wcycle_div*". To configure this, we need to set the respective bits under the *.mpcer* member, this can be done with the aid of the macro *SET_BIT*. Additionally, we use the same macro to enable the counter on the *.mpcmr* member.

Furthermore, we also set the API running privilege mode, data handling configuration, the core mode to *hp_core* and the system configuration.

```c
void init_monitor_hpcore()
{
    uint32_t mpcer_conf = 0;
    SET_BIT(mpcer_conf, evt_wcycle_mul);
    SET_BIT(mpcer_conf, evt_wcycle_div);

    uint32_t mpcmr_conf = 0;
    SET_BIT(mpcmr_conf, COUNT_EN);

    api_config_t config = {
        .privilege_mode = MACHINE,
        .data_config = print_to_console,
        .core_mode = hp_core,
        .sys_cfg = configSYSTEM_CFG,
        .core_mode = hp_core,
        .mpcer = mpcer_conf,
        .mpcmr = mpcmr_conf
    };

    perf_monitor_init(config);
}
```

Listing 6.7: Code example to configure API to monitor HP-Core.

The second scenario is configuring the API for the ESP32C6 LP-Core. The LP-Core implements the Privilege RISC-V ISA Specification with wired events. This can be seen on Table 6.1.

Table 6.1: Performance Counter and Low-Power CPU. Retrieved from [59].

Counter	**Counted Event**
mcycle	Clock cycles
minstret	The number of instructions
mhpmcounter3	Wait cycles for memory access
mhpmcounter4	Wait cycles for fetching instructions
mhpmcounter5	The number of memory read operations. An unaligned read is counted as two.
mhpmcounter6	The number of memory write operations. An unaligned write is counted as two.
mhpmcounter7	The number of unconditional jump instructions (jal, jr, jalr)
mhpmcounter8	The number of branch instructions
mhpmcounter9	The number of taken branch instructions
mhpmcounter10	The number of compressed instructions
mhpmcounter11	Wait cycles for multiplication instructions
mhpmcounter12	Wait cycles for division instructions

For the purpose of this example, the counters *mcycle*, *minstret* and *mhpmcounter10*. This was done by setting the member *counters_to_read* with the respective bits active. Furthermore, we also set the API running privilege mode, data handling configuration, the core mode to *lp_core* and the system configuration.

```c
void init_monitor_lpcore()
{
    uint32_t active_counters = 0;
    SET_BIT(active_counters, csr_cycle);
    SET_BIT(active_counters, csr_instret);
```

```
6       SET_BIT(active_counters, csr_hpmcounter10);
7
8       api_config_t config = {
9           .privilege_mode = MACHINE,
10          .counters_to_read = active_counters,
11          .data_config = print_to_console,
12          .core_mode = lp_core,
13          .sys_cfg = configSYSTEM_CFG
14      };
15
16      perf_monitor_init(config);
17  }
```

Listing 6.8: Code example to configure API to monitor LP-Core.

Use Case 2 - Start to end monitoring

After performing the API configuration we can then perform other actions. Regarding a start-to-end monitoring, we need to make use of the functions *perf_monitor_start* to capture the selected HPM counters on the starting point, and then *perf_monitor_stop* to capture the stopping point on the API. These two functions should be set before and after the portion of code the user wants to analyse.

After this, the user should create a *perf_monitor_t* structure that will be used as an input to the *perf_monitor_delta* function, and be filled with the delta variation between the start and stop points set previously.

Finally, it should use the *perf_monitor_print* to output the concrete delta values.

```
1  perf_monitor_start();
2
3  //Portion of code to monitor
4
5  perf_monitor_stop();
6
7  perf_monitor_t delta_value;
8  perf_monitor_delta(delta_value);
9  perf_monitor_print(delta_value);
```

Listing 6.9: Start-to-end monitoring code example.

Use Case 3 - Multiple monitoring points

In order to perform multiple monitoring points, the user can use the *perf_monitor_read* function. This functions retrieves the values from the configured active counters and places the values on the *perf_monitor_t* structure. Like the Start-to-End monitoring, single monitoring should also be performed after configuring the API.

By using this function, the user can place the function on the points he wants to monitor, and is then responsible for managing what to do with those values. The print function can be used to print the values present on the selected points.

```
1  perf_monitor_t pnt1, pnt2, pnt3;
2
3  perf_monitor_read(pnt1);
4  //code to monitor
```

```
5  perf_monitor_read(pnt2);
6  //code to monitor
7  perf_monitor_read(pnt3);
8
9  perf_monitor_print(pnt1);
10 perf_monitor_print(pnt2);
11 perf_monitor_print(pnt3);
```

Listing 6.10: Multiple points monitoring code example.

If the user does not want to make use of the API configuration features, he can replace the *perf_monitor_readc* with the *perf_monitor_readc* function that receives an additional bit set as a first parameter with the counters to read.

6.2 Evaluation

The following section will cover concrete use cases on the ESP32C6 application. The first scenario it will be shown how to initialise and use the API to retrieve HPM counter values while executing on LP-Core, with the current available API implementation. The second scenario shows how to configure the API on the HP-Core to select specific events and the respective results.

6.2.1 Retrieve HPM counters values - with counter inhibition

Considering that LP-Core has low resources, the API is used without using the predefined configuration functions. To do this, the user can makes use of the auxiliary functions that allow for a custom usage of the API. The first step is to create two variables of the structure*perf_monitor_t* to hold the start and end points of the application, under the global memory space.

After this the available counters are configured on the monitoring structure, configuring the counters ids for each counter. Then it is necessary to select the counters that will be actively read and inhibit the counters that are not read for performance purposes.

As certain bits on different registers are reserved and may contain values that are managed by the CPU and should not be manually modified, in order to configure a certain CSR, it should first retrieve the current value on the counter and only change the bits the user can modify.

In this case, the user should configure the available counters throughout the *mcountihibi* which has the bit on position 1 as reserved. This is because, usually, the second CSR for performance monitoring is the time counter. As previously mentioned, the *mtime* counter cannot be inhibited by the user as it is necessary for the system normal operation. With this, *mcounthibi* sets the second bit as reserved.

To retrieve configuration CSRs, the user should make usage of the *perf_read_config_csr* function to read configuration CSRs and *perf_set_config_csr* to set their respective values.

After manually configuring the bits, we can then set the respective HPM to the new value to configure the CSRs available on the system.

The following listing contains an example in how this approach was tested.

```c
void init_lp_core_config()
{
    perf_monitor_t esp32c6_lp = {
        .counters = {
            {.id = csr_cycle, .value = 0, .upper_value = 0},
            {.id = csr_time, .value = 0, .upper_value = 0},
            {.id = csr_instret, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter3, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter4, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter5, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter6, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter7, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter8, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter9, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter10, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter11, .value = 0, .upper_value = 0},
            {.id = csr_hpmcounter12, .value = 0, .upper_value = 0}}};
    start = esp32c6_lp;
    end = esp32c6_lp;

    perf_read_config_csr(&mcouninhibi);

    active_counters = mcouninhibi.value;
    SET_BIT(active_counters, csr_cycle);
    SET_BIT(active_counters, csr_instret);
    RESET_BIT(active_counters, csr_hpmcounter3);
    RESET_BIT(active_counters, csr_hpmcounter4);
    RESET_BIT(active_counters, csr_hpmcounter5);
    RESET_BIT(active_counters, csr_hpmcounter6);
    RESET_BIT(active_counters, csr_hpmcounter7);
    RESET_BIT(active_counters, csr_hpmcounter8);
    RESET_BIT(active_counters, csr_hpmcounter9);
    SET_BIT(active_counters, csr_hpmcounter10);
    RESET_BIT(active_counters, csr_hpmcounter11);
    RESET_BIT(active_counters, csr_hpmcounter12);

    mcouninhibi.value = active_counters;

    perf_set_config_csr(&mcouninhibi);
    mcouninhibi.value = 0;
    perf_read_config_csr(&mcouninhibi);

}
```

Listing 6.11: API usage and configuration to retrieve HPM counter values.

After executing this code on the LP-Core the following values were retrieved.

```
[mcouninhibi]: 61 - 0x0:ffffe405
```

The value retrieved from the *mcounthibi* register is as expected taking into consideration the bits that were made active. This indicates that the API is capable of accessing and changing the values present on the CSRs for the ISA specification.

6.2.2 Event configuration for HPM counters

Event configuration is performed by configuring a dedicated register that controls which events will trigger an increment on the respective counter. For this purpose. it will be shown

how to perform event configuration on the HP-Core Custom CSRs. For this purpose, the user starts by retrieving the value of the *mpecer* CSR, and then setting or resetting the bits to the events the user wants to configure.

To achieve this, the events cycle and instructions retired were chosen. After settings the respective bits, the user then needs to update the CSR value to the updated value.

After this, the *mpcrr* counter needs to be enabled, for this the *mpcmr* register is used to enable the counter and configure the overflow strategy, in this case to overflow.

After configuring these two register, then read operations can be performed under the *mpccr* custom CSR.

As this is a custom usage of the API and it is a fully custom extension, it made usage of the function *_perf_read_custom_csrc* to read custom extension CSRs and *_perf_set_custom_csrc* to set custom extension CSRs.

```
1  void app_main(void)
2  {
3      _perf_read_custom_csrc(0, &mpcer_c);
4
5      SET_BIT(mpcer_c.value, custom_evt_cycle);
6      RESET_BIT(mpcer_c.value, custom_evt_inst);
7      RESET_BIT(mpcer_c.value, custom_evt_ld_hazard);
8      RESET_BIT(mpcer_c.value, custom_evt_jmp_hazard);
9      RESET_BIT(mpcer_c.value, custom_evt_idle);
10     RESET_BIT(mpcer_c.value, custom_evt_load);
11     RESET_BIT(mpcer_c.value, custom_evt_store);
12     RESET_BIT(mpcer_c.value, custom_evt_jmp_uncond);
13     RESET_BIT(mpcer_c.value, custom_evt_branch);
14     RESET_BIT(mpcer_c.value, custom_evt_branch_taken);
15     RESET_BIT(mpcer_c.value, custom_evt_inst_comp);
16
17     _perf_set_custom_csrc(mpcer, mpcer_c.value);
18
19     _perf_read_custom_csrc(0, &mpcmr_c);
20
21     SET_BIT(mpcmr_c.value, COUNT_EN);
22     RESET_BIT(mpcmr_c.value, COUNT_SAT);
23
24     _perf_set_custom_csrc(mpcmr, mpcmr_c.value);
25
26     mpcer_c.value = 0;
27     mpcmr_c.value = 0;
28
29     _perf_read_custom_csrc(0, &mpcer_c);
30     _perf_read_custom_csrc(0, &mpcmr_c);
31
32     printf("HPM: mpcer: 0x%lx - mpcrr: 0x%lx\n", mpcer_c.value, mpcmr_c.
       value);
33
34     counter_t start = {.id = mpccr}, end = {.id = mpccr};
35     for (int i = 0; i < ITERATION_COUNT; i++)
36     {
37         TickType_t start_ticks, end_ticks;
38         unsigned long elapsed_ms;
39         start_ticks = xTaskGetTickCount();
40
41         _perf_read_custom_csrc(0, &start);
```

```
42
43          test_matrix();
44
45          _perf_read_custom_csrc(0, &end);
46          end_ticks = xTaskGetTickCount();
47          elapsed_ms = (end_ticks - start_ticks) * portTICK_PERIOD_MS;
48
49          printf("Elapsed time: %lu ms\n", elapsed_ms);
50          printf("mpccr[start]: 0x%lx\nmpccr[end]: 0x%lx\n", start.value,
    end.value);
51      }
52
53      for (int i = 0; i < ITERATION_COUNT; i++)
54      {
55          TickType_t start_ticks, end_ticks;
56          unsigned long elapsed_ms;
57          start_ticks = xTaskGetTickCount();
58
59          test_matrix();
60
61          end_ticks = xTaskGetTickCount();
62          elapsed_ms = (end_ticks - start_ticks) * portTICK_PERIOD_MS;
63
64          printf("Elapsed time: %lu ms\n", elapsed_ms);
65          printf("mpccr[start]: 0x%lx\nmpccr[end]: 0x%lx\n", start.value,
    end.value);
66      }
67  }
```

Listing 6.12: Event configuration code example.

In order to test the API impact on the system performance, the previous code was ran with 10000 iterations, while analysing the value retrieved by the custom extension from the *mpcrr* counter, which was set to track *cycles*, before and after calling the *test_matrix* function. This function performs matrix multiplication between two 15x15 matrices. Due to the low resources of the board chosen, and to increase the time taken during the execution, instead of only performing a single multiplication, it was performed 20 times per iteration. After retrieving the elapsed time from the executions with and without the API, the Table 6.2 was elaborated.

Table 6.2: Detailed Comparison of API vs No API Data

Metric	**API Data (ms)**	**No API Data (ms)**
Average	22.538986	22.523204
Maximum	23	23
Minimum	22	22
Standard Deviation	0.503276	0.503389

Comparing the differences between the executions times when using the API vs without using the API, it can be seen that the difference is minimal between the API execution data and a No-API, with the API having a slightly higher average of $22.538986(\pm 0.503276)ms$, compared to $22.523204(\pm 0.503389)ms$ without API.

The overhead percentage generated by the API, can be computed by using the average times from both the API and No API execution times, with the following formula:

$$\text{Overhead} = \frac{\text{API Average Time} - \text{No API Average Time}}{\text{No API Average Time}} \times 100$$

Substituting the exact values:

$$\text{Overhead} = \frac{22.538986 - 22.523204}{22.523204} \times 100$$

$$\text{Overhead} = \frac{0.015782}{22.523204} \times 100 \approx 0.07007\%$$

After analysing the overhead generated, it can be observed that it is minimal, $\approx 0.07007\%$ reinforcing the conclusion that the API's impact on performance is negligible.

6.2.3 Memory usage

Another important topic to take into consideration for an API targeting real-time systems, where they are working most of the time in resource constrained environments is memory usage. Currently, the API is being completely instantiated on the stack, and all memory is allocated at compile time. Because of this, the amount of memory used by the API remains constant during its execution. Depending on the API use cases, it will require different amounts of memory depending on the configurations performed during its execution, that will directly impact the size of the API structures. The more CSRs the system has available to be tracked, the more memory the API will require to work properly.

One of the structures that will be more widely used across the API execution will be the *perf_monitor_t* structure, defined in Chapter 4. This structure is composed of an multiple *counter_t* structures, one for each counter being tracked by the API, and as this structure will have different sizes depending on the CPU architecture it will also vary among different systems. Furthermore, the configuration structure *api_config_t* is also required for the API automatic configuration before its execution.

Considering the ESP32C6 Board, which is composed of 32-bit RISC-V core, the API structures will have the sizes present in Table 6.3.

Table 6.3: Sizes of API Structures

Structure Name	**Size (bytes)**
api_config_t	376
monitor_t	204
register_cfg_t	12
counter_t	12

As it can be seen from Table 6.3 the *api_config_t* structure is the biggest with 376 bytes of memory required, followed by the *monitor_t* with 204 bytes, and *register_cfg_t* and *counter_t* with 12 bytes required.

During the API life-cycle only one instance of the *api_config_t* structure will be present during the whole execution. This happens because this API contains all the information regarding the system configurations and how the API should behave, in section 6.1.2, on the Use Case 1, an example is provided in how to configure the API using this structure.

This will require the system to have at least 376 bytes available just to configure the API, event if the number of target counters being used is less then the available on the system. An additional note is that this structure will contain the configurations for both HP-Core and LP-Core for the moment, which might be inefficient if only one core is the target of analysis.

However, the user can always use the API without recurring to the initialization functions. If the system has low resources, it could manually configure the system using a *register_cfg_t* structure for each counter it wants to configure. This will reduce the memory required by the API as it will only need to use the amount corresponding to the target counters being currently analysed, instead of the whole counters in the system like for the initialization functionality.

By doing this, the user can reduce the memory required to the minimum number of counters that will be configured. This might prove beneficial if only one of the two cores (HP-Core and LP-Core) is being used, for example, as the HP-Core uses a custom extension for performance monitoring, with only three CSRs, it will only need three instances of the structure *register_cfg_t* which has size of 12 bytes, making it a total of only 36 bytes. For the LP-Core, which has thirteen counters plus one more for enabling the counters (*mcounteren*), the value would be 156 bytes for the counters and an additional 12 bytes for the counter-enable CSR, resulting in a total of 168 bytes for the configuration. It is important to note that despite the reduction in size, the API would not be working with all the available functionalities.

Other than this, the user could also make use of the API without any additional configuration for the memory addressing of the CSRs. For the moment, some auxiliary functions are present that uses hard-coded address values for ISA CSRs and some static configuration variables for custom addresses. This reduces the need of configuring the API, reducing the memory required. This was the approach used throughout this section.

Without the need to configure the API, and making use of the hard coded values on the auxiliary functions, the user will only be required to allocate memory to save the retrieved metrics from the hardware performance counters. As the API was not configured, the user needs to manually initialize the structure, filling the respective registers ids and settings the value members to 0.

This can be done in one of two ways, first using the *perf_monitor_t* structure, which is composed of one *counter_t* structure for each CSR available on the system, this included both HP-Core and LP-Core, or by manually creating the respective *counter_t* structures for each counter value retrieved.

By using the first approach the API will require 204 bytes of memory per structure, each structure is capable of holding the values from one monitoring point. This means that for each monitoring point set, the API will require a hard set value of 204 bytes.

The second approach is more flexible, as previously mentioned, due to the differences in CSRs in both HP-Core and LP-Core, by manually creating the counter structures the API will require: 36 bytes for the HP-Core and 168 bytes for the LP-Core, per monitoring point.

Recapping, considering an API execution where the user uses the API configuration function and performs a start-to-end monitoring, which requires two monitoring points, one before the portion of code being analysed, and another directly after it, the API will required the values depicted in Table 6.4.

Table 6.4: API Memory requirements for start-to-end monitoring

API Usage	Configuration Memory	Monitoring memory	Total
API initialization and start-to-end monitoring functions.	376 bytes	2x204 bytes	784 bytes
LP manual custom initialization and start-to-end monitoring functions	168 bytes	2x204 bytes	576 bytes
HP manual custom initialization and start-to-end monitoring functions	36 bytes	2x204 bytes	444 bytes
LP manual custom initialization and manual monitoring	168 bytes	2x168 bytes	504 bytes
HP manual custom initialization and manual monitoring	36 bytes	2x36 bytes	108 bytes
LP static initialization and start-to-end monitoring functions	0 bytes	2x204 bytes	408 bytes
HP static initialization and start-to-end monitoring functions	0 bytes	2x204 bytes	408 bytes
LP static initialization and manual monitoring	0 bytes	2x168 bytes	336 bytes
HP static initialization and manual monitoring	0 bytes	2x36 bytes	72 bytes

As it can be seen from the memory variations, depending on the effort and customization the user uses during the API usage, it can widely impact the memory required to used the API. Using the API out-of-the-box functionalities will require the most memory to execute, as the user will be using the API configuration for the whole system and won't need to take into consideration implementation details, it is expected that the amount of memory required would be higher. By increasing the customization from the user, the memory required will be lesser as the user will now be responsible for configuring the system itself, and will only be using the API as an interface to interact with the PMU, all the configuration work and metric handling needs to be performed by the user.

It is crucial to highlight that this analysis is only taking into consideration variables that need to be allocated during the API execution, and is not taking into consideration other memory requirements the API might bring to the application, such as API library size, functions and macro definitions, etc.

6.3 Challenges in Evaluation

During the API testing and evaluation there were multiple challenges that were faced. The first challenges started when trying to use the initial emulated environment to access the hardware performance counters specific to a RISC-V machine. Due to a lack of specification

on the QEMU *virt* machine, it was not possible to configure the *hpmevents* CSRs, which would prove a limitation in fulfilling the API objectives.

To overcome this, a more specific machine was chosen to be emulated on the machine, *sifive_u*, after analysing this machine implementation details and trying to access the respective CSRs, some illegal exceptions were thrown and in some cases the machine froze while not providing any outputs. This was probably due to a lack of configuration, but it proved to be a challenge to overcome. Faced with those challenges, it was decided to acquire a new RISC-V physical system, ESP32C6 board, that proved to have the necessary extensions and functionalities.

Later on, while testing the system, the board proved to be more complex than initially expected. After some analysis on the [59], it was detected that the HP-Core and LP-Core implemented different extensions individually, and one of them even implemented a custom extension for the performance monitoring capabilities. Incorporating this custom extension required a refactor on the API architecture, but also made it more versatile with the possibility to configure custom extensions via the API configuration files.

Another challenge was achieving proper synchronization between the HP-Core and LP-Core and also exchanging data between both cores. As the LP-Core is highly limited on the resources available, the API evaluation assessment was difficult. Initially, the tests performed were using the HP-Core to access the LP-Core registers and memory, achieving this synchronization was a challenge and sometimes lead to corrupt memory due to inappropriate accesses.

Other than this, the differences in architectures among HP-Core and LP-Core also difficult this process, especially considering that normally, the HP-Core enters in deep sleep for the LP-Core to start, and when the HP-Core resumes its operation, the LP-Core is disabled or put to sleep. Despite being possible to have the LP-Core acting as a co-processor with both active at the same time, it increments complexity.

Another challenge was in how to correctly evaluate the overhead that is introduced by the API. While the API simplifies the system's performance monitoring it also introduces an overhead on the system and may impact the system performance. Ensuring that the API does not negatively impact the system performance is crucial in real-time and resource-constrained environments.

One of the key features of this API is its configuration capabilities, however, in order to perform these configurations, they need to be stored in memory. This increase in memory consumption is a huge concern when dealing with resource-constrained devices. Managing the memory used by the API is essential to ensure the system's normal functionality.

Dealing with these challenges increased the complexity of the API development and led to an increase on the time taken to perform the evaluation. Due to the time limitations on the thesis development this had an impact on the overall thesis development.

6.4 Research Questions Analysis

The work presented throughout this thesis aimed to answer the research questions initially proposed in Chapter 1. The solution developed and presented in this thesis also provides an interface that is capable of fulfilling the research questions and the objectives initially proposed.

In Table 6.5 an overview in how the solution solved the respective question is presented.

Table 6.5: Research question analysis

Research Question	Solution on API
RQ1 - Is it possible to extract performance metrics of applications running on RISC-V microprocessor? What about real-time applications?	Read Operation that retrieves values from hardware performance counters.
RQ2 - Is it possible to configure the system to measure different performance metrics based on chosen events, on a RISC-V microprocessor?	Provides multiple mappings from the hardware to the API and provides an interface to configure the values present on the configuration CSRs.
RQ3 - Can an approach extract and configure performance metrics of real-time applications running on RISC-V microprocessor, considering the execution in a RTOS, while providing a low, close to negligible, overhead?	Configuring the hardware performance counters to be only enabled if necessary, and only the required number of counter values is retrieved.
RQ4 - Can the approach be developed considering a minimum of standardized extensions from the RISC-V ISA specifications?	Developed requiring only the *Zicsr* extension to write and read from the CSRs and the *Counters* extension that contains the performance counters definition.
RQ5 - Can the solution be provided as an API, easily deployable in different RTOS and RISC-V Systems?	By abstracting the dependencies external to the API implementation and allowing the user to configure and extend the API to support other extensions, makes it more portable and deployable if the respective configurations and portings are performed.

Regarding the research question **RQ1**, the research performed on the current State of the Art for RISC-V system's monitoring in Chapters 2 and 3, demonstrated that it is possible to extract performance metrics from applications running on a RISC-V system by accessing the values of the hardware performance counters CSRs present on ISA specifications for both unprivileged and privileged RISC-V systems. It is important to note that not all systems are required to have these specific CSRs implemented on the system. Additionally, the ISA specification also defined three default hardware performance counters: cycles, instructions-retired and time.

Multiple solutions have been developed that provide elementary support to access these hardware performance counters in RISC-V systems, however most of them are used on Linux systems and don't consider a running RTOS, where real-time applications usually execute. Despite not having a lot of support, it is also possible to extract performance metrics from a real-time application if the respective accesses to the hardware performance counters are performed correctly.

Regarding research question **RQ2**, in Chapters 2 and 3 it was described that there are multiple implementation-specific-details on a RISC-V system that may impact the behaviour and configuration on the RISC-V System's hardware performance counters, such as enabling or

disabling, and configuring which events will cause the counter to increment. Event configuration capabilities are documented under the RISC-V privileged ISA under specific CSRs that control the events that will trigger a specific hardware performance counter to increment.

For the research question **RQ3**, the solution proposed on this thesis in Chapter 4, is capable of performing the extraction and configuration of performance metrics from an application running on FreeRTOS, a RTOS, executing on a RISC-V system, with a minimal overhead as demonstrated in Chapter 6.

In order to achieve research question **RQ4**, the solution needs to be capable of interacting with performance monitoring related extensions from the RISC-V ISA. From the analysis performed in Chapters 2 and 3, it was identified that according to the RISC-V ISA there are two main extensions that deal with performance metrics, one is the *Zicsr* extension, which contains the instructions and details for accessing and using CSRs on a RISC-V system, and the other is the *Counters* extension which defines specific CSRs that function as hardware performance counters on a RISC-V system, that later originated *Zicntr* for the base counters and timers, and *Zihpm* for the additional customizable general-purpose.

This was taken into consideration during the development of the solution described in 4, which is capable of providing support to the identified extensions by making usage of the CSR-address mappings for the hardware performance counters, and the respective event mapping on the solution. By making usage of the mapped addresses and the instructions available on *Zicsr* extension, it is possible to access, manipulate and retrieve values directly from the hardware performance counters.

For research question **RQ5**, it was also shown in Chapters 5 and 6 that it is possible to perform the API porting between system platforms and operating systems, by making usage of the solution configuration capabilities described in Chapter 4, which makes the API more versatile and portable across different systems.

6.5 Summary

This chapter provided an overview on the current status of the API development. It highlighted functionalities that have already been implemented and how they behave, while also showcasing some challenges that appeared during development.

The API provides a simple interface that can be used by any user under the ported systems. If the target system does not have functionalities that contain a lot of divergent implementation details from the ISA specification, it may even be used as-is to retrieve values from the base counters and timers. However, if a more complex functionality is required, such as event configuration, which requires mapping the event values for each counter, or the usage of a custom extension, it is necessary to ensure that these functionalities have been correctly ported prior to their usage on the API, otherwise some unpredicted behaviours may occur.

In the future, it would be helpful to restrict the user from trying to use functionalities that have been not fully ported or implemented on the API, right from the configuration process or compilation, while also providing more documentation on the already implemented features and ported functions.

The examples use cases provided in this chapter showcase the three core functionalities that were initially proposed for the API, namely, perform the API configuration, which will in

turn configure the RISC-V PMU to perform as required, perform a start-to-end monitoring, programmatically, under a specific portion of code, and finally, retrieve the configured performance metrics from the HPMs in multiple defined monitoring points. The evaluation of the core functionalities of API provides an answer both **RQ1** and **RQ2**, by showcasing that the API is capable of retrieving and configuring the retrieval of performance metrics in real-time systems.

It was possible to test that the API is capable of configuring a PMU that contains custom extensions, which was the case for the ESP32C6 board, and also to retrieve specific counter values while disabling the unused counters to reduce system resource consumption. These tests showcase that the API is capable of fulfilling the initially defined objectives, while providing portability across platforms (**RQ5**).

The evaluation performed in this chapter partially answered **RQ3**. In fact, as more features and complexity is added to the API itself, the higher the overhead generated by its execution. While the overhead generated by the API could be better tested, it can be stated that it is possible to develop an approach that is capable of performing configuration and extraction of performance metrics, from RISC-V systems running real-time applications, while executing on a Real-Time Operating System, with a low overhead. The solution presented in Chapter 4 has demonstrated to have a low and close to negligible overhead ($\approx 0.07007\%$) after some tests, and some refinements can be performed on the API to optimize resource utilization.

Furthermore, by taking into consideration details that might impact performance, when performing the respective configuration of the API, such as, keeping the enabled hardware performance counters to the minimum required, and selecting the position of the monitoring points in the code, considering the events being tracked and if any traps are occurring. Dealing with multiple traps may increase the overhead of the API as it usually requires context-switches which inherently have some overhead, if the reading is done inside that trap, it may not only produce incorrect values, it could even increase the overhead if some of the metrics are shared across different cores.

Additionally, a memory usage analysis was also performed for the API execution. It was noticeable that depending on the amount of configuration the user needed to perform, the memory used would be lesser as the API would not need to have all the system configurations and the user could only configure the target CSRs it requires. This will require the user to have a more in-depth understanding on the inner-workings of the system and will in turn reduce the memory requirements for the API.

Chapter 7

Conclusions

Despite the time constraints for the thesis development, the developed API can work as a proof-of-context for the development of a more sophisticated tool capable of answering the research questions and fulfilling the objectives initially proposed.

As demonstrated, configuring and accessing the hardware performance counters can be a time-consuming task that requires through investigation and familiarity with the RISC-V ISA specifications and the corresponding platform-specific implementation details.

By making usage of the API, users can perform a simple configuration of the platform's PMU, enabling access and configuration of performance metrics directly from the hardware performance counters. This removes the need to interact with the platform-specific SBI or directly embed assembly instructions within the code, simplifying the process of performance monitoring.

Despite being somewhat limited, the API achieved the purposed objectives. It is capable of performing the configuration process of the RISC-V PMU, with both event configuration capabilities and CSRs enabling control.

The event configuration is dependant on each system implementation details. In fact, each manufacturer is allowed to develop their own RISC-V ISA extensions, that may be completely different from the ISA specifications if they deem necessary, to perform performance monitoring on their system.

Furthermore, some systems may have specific hardware performance counters that do not allow the system to configure the triggering events, having hard-wired specific events to each counter, that will cause the respective CSR to increment, while others may not event have configurable events and only the ISA defined default counters for cycles, instructions-retired and time.

Overall, using the API reduces the complexity to access the systems HPM counters. Together with its various configurations options, it simplifies the API porting process between system platforms and operating systems, making the API more robust and portable.

This porting is achieved by identifying all the operations that are reliant on a specific operating system and platform and perform the correct mapping on the API configuration variables. By using the API specific structures and macros, it is possible to abstract the user from the specific implementation details of the running environment and deploy the API across different systems and platforms.

The analysis performed on the memory requirements and overhead generated by the API were also satisfactory, as the API currently induces a close no negligible overhead, and the

memory requirements can vary depending on the level of customization the user performs. The less he manually configures the API, the higher memory requirements will be needed. Due to the API currently implemented features, it might be more valuable to only use the API as a direct interface to the System's PMU by making usage of the API auxiliary functions, specially if the user is familiarized with the target system implementation details and is able to manually configure the PMU. However, using the API still brings some advantages, by using the out-of-the-box configurations and functions it enables users to more easily retrieve performance metrics without having to manually configure the system.

7.1 Limitations

Currently, the memory management capabilities of the API are restricted. Despite the definition for the customization of the API memory management functions, it was not integrated into the whole implementation. Till this moment, all variables used on the API are variables on the stack and no dynamic allocation is performed. Including dynamic memory allocation would improve the management of the memory resources used by the application, as it would be capable of only using the required amount to achieve the system requirements. An example of this limitations is the configurations of the registers available in the system. Right now the API needs to perform the configuration to all the available HPM registers on the system, which makes the amount of memory resources used constant during the API lifecycle. This can be useful for some use cases, especially if all the CSRs related with performance monitoring are used during the API. However, if some of the registers are not necessary and will not be used during the systems' execution, it might be useful to only configure the required registers.

Other than this, when performing read operations on the API, all the values need to be handled by the user. Despite making the API more customizable to the user, sometimes it might be troublesome and time-consuming to perform a manual handling of the data retrieved. By including dynamic memory allocation, it would be possible to introduce structures more dynamic and versatile that would handle data retrieved by the API.

Despite this, the lack of dynamic memory does not restrict the API overall functionalities. Handling all the API data and configurations is more troublesome and complex in certain scenarios, such as, dealing with applications where the number of function calls is unknown. If the API is not aware apriori of how many times a read operation will be performed, if we are using static allocation we must either allocate a large number of variables that should be capable of handling the amount of values we predict we might need, or we can also make usage of a circular buffer that will automatically discard the earliest saves when overflowing. Despite this, it requires the user to define at compile-time a prediction to the amount of memory needs to be allocated. If the API was making usage of dynamic memory instead, it could define a limit upon from which it would start overwriting values, in order to control memory corruption due to the lack of memory resources, if the defined limit is not reached the API would only allocate the necessary memory.

Note that in return, using this features would increase the API complexity, as dealing with memory management is also a complex task. This would also increase the overhead generated by the API during the application' execution, especially if the number of allocations and deallocations operations were significant. On the other hand, defining everything on the stack during compile time, will ensure that no time is spent during run-time to perform those memory management operations. This is a trade-off between overhead and memory-usage,

that depending on the application specific requirements would empower the user to choose the option that fits his needs the best.

7.2 Future Work

Regarding future work, there are a lot of functionalities that could be added to enhance the API overall performance and versatility. Since improving memory management efficiency, to add new functionalities, such as, device support and out-of-the-box metrics computation.

A crucial aspect to the API development would be to increase the tests performed and better evaluate the overhead generated by the API, considering both resources usage and time consumption for the API usage.

Another important addition would be the definition of out-of-the-box metrics. By enabling the API to compute "standard" metrics used on performance monitoring, it would enable the user to be more efficient and faster when performing those operations. Additionally, a configuration interface could also be implemented where the user is capable of defining custom metrics, their requirements to be computed and how the assessment is done. This would make the API more robust and versatile, increasing the scenarios where it can be used.

Introducing a more user friendly configuration interface would also facilitate the API usage. Such an example would be to start importing the API configuration from a dedicated configuration file, and provide a different interface that aided on the API configuration process, such as an external program that provided a GUI interface to configure the API.

Other than this, it would also be helpful to include some scripts that would automatically scan the system for its specifications and configure the API in a way that would limit the user to not perform illegal operations considering the platform functionalities and devices available.

Integrating different data handling mechanism would also greatly improve the API functionalities. Enabling synchronization features with other devices or components, could help ease the requirements on the system's resources, especially when dealing with memory-constrained systems. A system with low resources could perform a recurrent sync with a specific device or service, which would keep the local resource usage limited and would allow the system to be used for a wider period of time without having to take into consideration the memory resources available system. This could be done by connecting the device used by the API with another one via different connectivity protocols, such as, Ethernet, GPIO, JTAG, Bluetooth, Cloud saving or other mechanism.

Enabling the API to handle different scenarios, such as dealing with concurrent and parallel applications which needs to take into consideration different cores CSRs, where some of them would be shared across multiple cores or even the whole system and other would be dedicated to each core. By being capable of dealing with these scenarios the API would be even more versatile and useful to the user, as it would simplify a highly complex problem that needs to take into consideration a wide range of platform-specific implementation details.

Performing a wider range of tests and evaluations on the API performance and behaviour would also be imperative to improve the API usability and functionalities. An evaluation on the minimum resources necessary to execute the API and the consumption during the API

usage depending on the requirements chosen would also be a good improvement for the API resilience.

Bibliography

[1] Edward A. Lee. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models". In: Sensors 15.3 (2015), pp. 4837–4869. issn: 1424-8220. doi: 10.3390/s150304837. url: https://www.mdpi.com/1424-8220/15/3/4837.

[2] M.G. Rodd and L. Motus. Timing Analysis of Real-Time Software. Elsevier Science, 1994. isbn: 9780080983967. url: https://books.google.pt/books?id=zaI7AAAAQBAJ.

[3] Karan Singh, Major Bhadauria, and Sally A. McKee. "Real time power estimation and thread scheduling via performance counters". In: SIGARCH Comput. Archit. News 37.2 (July 2009), pp. 46–55. issn: 0163-5964. doi: 10.1145/1577129.1577137. url: https://doi.org/10.1145/1577129.1577137.

[4] Andrew Shell Waterman. "Design of the RISC-V instruction set architecture". In: University of California, Berkeley (Jan. 2016). url: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf.

[5] et al Lee Yunsup. "An agile approach to building RISC-V microprocessors." In: (2016). ieee Micro 36.2, pp. 8–20.

[6] FreeRTOS. The FreeRTOS™ Kernel. https://www.freertos.org/RTOS.html.

[7] Zephyr. The Zephyr Project. https://www.zephyrproject.org/.

[8] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 20191213, SiFive Inc., EECS Department, UC Berkeley. Chap. 10.

[9] A. Waterman, K. Asanović, and J. Hauser. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203, SiFive Inc., CS Division, EECS Department, University of California, Berkeley, December 4, 2021.

[10] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 20191213, SiFive Inc., EECS Department, UC Berkeley.

[11] RISC-V Performance Analysis SIG. Performance analysis SIG. https://github.com/riscv-admin/perf-analysis/blob/main/CHARTER.md. Accessed Jan. 18, 2024.

[12] Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page.

[13] Gordon Bell and William D. Strecker. "Computer structures: What have we learned from the PDP-11?" In: 3rd Annual Symposium on Computer Architectures 4 (Jan. 1976), pp. 1–14. doi: https://dl.acm.org/doi/10.1145/633617.803541.

[14] Scott Atchley et al. "Frontier: Exploring Exascale". In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23. , Denver, CO, USA, Association for Computing Machinery, 2023. doi: 10.1145/3581784.3607089. url: https://doi.org/10.1145/3581784.3607089.

[15] Western Sydney University. World first supercomputer capable of brain-scale simulation. https://www.westernsydney.edu.au/newscentre/news_centre/more_news_stories/world_first_supercomputer_capable_of_brain-scale_simulation_being_built_at_western_sydney_university. Accessed: Jan. 17, 2024.

[16] Western Sydney University. ICNS to build brain-scale supercomputer. https://www.westernsydney.edu.au/icns/news/icns_to_build_brain-scale_supercomputer. Accessed: Jan. 17, 2024.

[17] S. Sharma. RISC-V Architecture: A Comprehensive Guide to the Open Source ISA. https://www.wevolver.com/article/risc-v-architecture-a-comprehensive-guide-to-the-open-source-isa. Accessed: Jan. 17, 2024. July 2023.

[18] Qualcomm. What is RISC-V and Why We're Unlocking Its Potential (July 2023). https://www.qualcomm.com/news/onq/2023/09/what-is-risc-v-and-why-were-unlocking-its-potential. Accessed: Jan. 17, 2024.

[19] Linux Foundation. Linux Foundation - Decentralized Innovation, built with trust. https://www.linuxfoundation.org/. Accessed Jan. 17, 2024.

[20] RISC-V Community News. Celebrating iotday: How RISC-V enables IOT innovation. https://riscv.org/news/2019/04/celebrating-iotday-how-risc-v-enables-iot-innovation/. Accessed Jan. 17, 2024.

[21] C. -H. Yang. "AI Acceleration with RISC-V for Edge Computing". In: (2020). International Symposium on VLSI Design, Automation and Test (VLSI-DAT), pp. 1–1. doi: 10.1109/VLSI-DAT49148.2020.9196404.

[22] V. N. Chander and K. Varghese. "A Soft RISC-V Vector Processor for Edge-AI". In: (2020). 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID), pp. 263–268. doi: 10.1109/VLSID2022.2022.00058.

[23] A. Daleiden. Ventana introduces Veyron, World's first data center class RISC-V CPU. https://riscv.org/news/2022/12/ventana-introduces-veyron-worlds-first-data-center-class-risc-v-cpu-product-family/. Accessed Jan. 17, 2024.

[24] K. Freund. Ventana micro brings RISC-V into the Data Center. https://www.forbes.com/sites/karlfreund/2022/10/24/ventana-micro-brings-risc-v-into-the-data-center/. Accessed Jan. 17, 2024.

[25] SiFive. https://www.sifive.com/. Accessed Jan. 17, 2024.

[26] Western Digital. https://www.westerndigital.com/en-gb. Accessed Jan. 17, 2024.

[27] Ruobing Han et al. Supporting CUDA for an extended RISC-V GPU architecture. 2021. arXiv: 2109.00673 [cs.PL].

[28] S. D. Mascio et al. "The case for RISC-V in space". In: SpringerLink (2020). Accessed: Jan. 17, 2024, pp. 1–1. url: https://link.springer.com/chapter/10.1007/978-3-030-11973-7%5C_37.

[29] Gopi Ganapathy et al. "Hardware emulation for functional verification of K5". In: Proceedings of the 33rd Annual Design Automation Conference. DAC '96. Las Vegas, Nevada, USA: Association for Computing Machinery, 1996, pp. 315–318. isbn: 0897917790. doi: 10.1145/240518.240578. url: https://doi.org/10.1145/240518.240578.

[30] C. Nitsch et al. "Embedded system architecture design based on real-time emulation". In: Proceedings 11th International Workshop on Rapid System Prototyping.RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668). June 2000, pp. 228–233. doi: 10.1109/IWRSP.2000.855241.

[31] Esam A. Qaralleh and Khalid A. Darabkh. "A new method for teaching microprocessors course using emulation". In: Computer Applications in Engineering Education 23.3 (2015), pp. 455–463. doi: https://doi.org/10.1002/cae.21616. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21616. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.21616.

[32] K. Weiss, T. Steckstor, and W. Rosenstiel. "Performance analysis of a RTOS by emulation of an embedded system". In: Proceedings Tenth IEEE International Workshop

on Rapid System Prototyping. Shortening the Path from Specification to Prototype (Cat. No.PR00246). June 1999, pp. 146–151. doi: 10.1109/IWRSP.1999.779045.

[33] Vladimir Herdt, Daniel Große, and Rolf Drechsler. "Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes". In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). Mar. 2020, pp. 618–621. doi: 10.23919/DATE48585.2020.9116522.

[34] Alec Roelke and Mircea R. Stan. "RISC5: Implementing the RISC-V ISA in gem5". In: Proceedings of Computer Architecture Research with RISC-V, Boston, Massachusetts USA, October 14, 2017 (CARRV'17). Oct. 2017. url: https://carrv.github.io/2017/papers/roelke-risc5-carrv2017.pdf.

[35] Nathan Binkert et al. "The gem5 simulator". In: SIGARCH Comput. Archit. News 39.2 (Aug. 2011), pp. 1–7. issn: 0163-5964. doi: 10.1145/2024716.2024718. url: https://doi.org/10.1145/2024716.2024718.

[36] Jonathan Bachrach et al. "Chisel: constructing hardware in a Scala embedded language". In: Proceedings of the 49th Annual Design Automation Conference. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. isbn: 9781450311991. doi: 10.1145/2228360.2228584. url: https://doi.org/10.1145/2228360.2228584.

[37] RISCV-QEMU. https://github.com/riscv/riscv-qemu.

[38] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: ATEC '05. Proceedings of the Annual Conference on USENIX Annual Technical Conference. Anaheim, CA: USENIX Association, 2005, p. 41.

[39] John A. Stankovic and R. Rajkumar. "Real-Time Operating Systems". In: Real-Time Syst. 28.2–3 (Nov. 2004), pp. 237–253. issn: 0922-6443. doi: 10.1023/B:TIME.0000045319.20260.73. url: https://doi.org/10.1023/B:TIME.0000045319.20260.73.

[40] Antmicro and RISC-V Foundation. The RISC-V Getting Started Guide. https://risc-v-getting-started-guide.readthedocs.io/en/latest/index.html. 2020.

[41] Antmicro. Antmicro. https://antmicro.com/.

[42] Zephyr. Zephyr RTOS featured in RISC-V Getting Started Guide. https://www.zephyrproject.org/zephyr-rtos-featured-in-risc-v-getting-started-guide/. 2019.

[43] Debian. RISC-V - Debian Wiki. https://wiki.debian.org/RISC-V. 2023.

[44] FreeRTOS. FreeRTOS for RISC-V RV32 and RV64. https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html.

[45] Stephen Berard. Why We Moved from FreeRTOS to Zephyr RTOS. Retrieved from: https://zephyrproject.org/why-we-moved-from-freertos-to-zephyr-rtos/. 2023.

[46] F. Mauroner T. Scheipel and M. Baunach. "System-Aware Performance Monitoring Unit for RISC-V Architectures". In: Euromicro Conference on Digital System Design (2017). Vienna, Austria, pp. 86–93. doi: 10.1109/DSD.2017.28.

[47] B. Sprunt. In: IEEE Micro, vol. 22, no. 4 (Aug. 2002). The basics of performance-monitoring hardware, pp. 64–71. doi: 10.1109/MM.2002.1028477.

[48] RISC-V Platform Specification Task Group. RISC-V Supervisor Binary Interface Specification. 2022. Chap. 11.

[49] A. Waterman et al. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.7, EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2015-49, 2015.

[50] J. M. Domingos, P. Tomas, and L. Sousa. "Supporting RISC-V Performance Counters through Performance analysis tools for Linux (Perf)". In: ACM (2021). doi: 10.48550/arXiv.2112.11767.
[51] A. Waterman et al. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.9, EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-129, 2016.
[52] A. Waterman et al. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1, EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-118, 2016.
[53] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified, RISC-V Foundation, June 2019.
[54] Intel Corporation. Intel Performance Counter Monitor (PCM). https://github.com/intel/pcm. Accessed: 2024-09-18.
[55] Innovative Computing Laboratory. Papi. https://icl.utk.edu/papi/.
[56] Barcelona Supercomputing Center. Extrae. https://tools.bsc.es/extrae.
[57] Jonathas Silveira et al. "Prof5: A RISC-V profiler tool". In: 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Nov. 2022, pp. 201–210. doi: 10.1109/SBAC-PAD55451.2022.00031.
[58] Joao Mario Domingos et al. "Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools". In: 2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP). July 2023, pp. 94–101. doi: 10.1109/ASAP57973.2023.00027.
[59] Espressif Systems. ESP32-C6 Technical Reference Manual. 2024.
[60] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 20240411, SiFive Inc., EECS Department, UC Berkeley.
[61] A. Waterman et al. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 20240411, RISC-V Foundation, June 2019.
[62] Espressif Systems. ESP32-C6 Series Datasheet. Version 1.2. 2024.

Appendix A

ISA CSR-Address mappings in C

```c
#ifndef _RISCV_CSR
#define _RISCV_CSR

#define UNPRIV_FLOATING_POINT_CSR_MASK          ((int) 0x001)  //CSR Starting address for Unprivileged Floating-Poing CSRs

#define UNPRIV_COUNTERS_TIMER_MASK              ((int) 0xC00)  //CSR Starting address for Unprivileged Counters/Timers
#define UNPRIV_UPPER_COUNTERS_TIMER_MASK        ((int) 0xC80)  //CSR Starting address for Upper 32 bits of Unprivileged
                                                               //Counters/Timers (RV32 Only)

#define MACHINE_COUNTERS_TIMER_MASK             ((int) 0xB00)  //CSR Starting address for Machine Counters/Timers
#define MACHINE_UPPER_COUNTERS_TIMER_MASK       ((int) 0xB80)  //CSR Starting address for Upper 32 bits of Machine
                                                               //Counters/Timers (RV32 Only)

#define MACHINE_COUNTER_INHIBIT_ADDRESS         ((int) 0x0320)  //CSR address for Machine Counter-Inhibit
#define MPM_EVENT_SELECTOR_START_MASK           ((int) 0x0320)  //CSR Starting address for Machine performance-monitor event selector


enum float_point_csr
{
    FP_FFLAGS_idx = 0,      //Floating-Point Accrued Exceptions
    FP_FRM_idx,             //Floating-Point Dynamic Rounding Mode
    FP_FCSR_idx,            //Floating-Point Control and Status Register (frm + fflags)
    FP_END_idx              //This should always be last!
};

#define UFP_FFLAGS          (FP_FFLAGS_idx      | UNPRIV_FLOATING_POINT_CSR_MASK)    //Unprivileged Floating-Point Accrued Exceptions
#define UFP_FRM             (FP_FRM_idx         | UNPRIV_FLOATING_POINT_CSR_MASK)    //Unprivileged Floating-Point Dynamic Rounding Mode
#define UFP_FCSR            (FP_FCSR_idx        | UNPRIV_FLOATING_POINT_CSR_MASK)    //Unprivileged Floating-Point Control and Status Register (frm + fflags)

```

```
30 #define UFP_END                (FP_END_idx          |
       UNPRIV_FLOATING_POINT_CSR_MASK)    //This should always be last!
31
32 enum counters_timers
33 {
34      PFC_CYCLE_idx = 0,         //Cycle counter for RDCYCLE Instruction
35      PFC_TIME_idx,              //Timer for RDTIME instruction
36      PFC_INSTRET_idx,           //Instructions-retired counter for
       RDINSTRET instruction
37      PFC_hpmcounter3_idx,       //PMC
38      PFC_hpmcounter4_idx,       //PMC
39      PFC_hpmcounter5_idx,       //PMC
40      PFC_hpmcounter6_idx,       //PMC
41      PFC_hpmcounter7_idx,       //PMC
42      PFC_hpmcounter8_idx,       //PMC
43      PFC_hpmcounter9_idx,       //PMC
44      PFC_hpmcounter10_idx,      //PMC
45      PFC_hpmcounter11_idx,      //PMC
46      PFC_hpmcounter12_idx,      //PMC
47      PFC_hpmcounter13_idx,      //PMC
48      PFC_hpmcounter14_idx,      //PMC
49      PFC_hpmcounter15_idx,      //PMC
50      PFC_hpmcounter16_idx,      //PMC
51      PFC_hpmcounter17_idx,      //PMC
52      PFC_hpmcounter18_idx,      //PMC
53      PFC_hpmcounter19_idx,      //PMC
54      PFC_hpmcounter20_idx,      //PMC
55      PFC_hpmcounter21_idx,      //PMC
56      PFC_hpmcounter22_idx,      //PMC
57      PFC_hpmcounter23_idx,      //PMC
58      PFC_hpmcounter24_idx,      //PMC
59      PFC_hpmcounter25_idx,      //PMC
60      PFC_hpmcounter26_idx,      //PMC
61      PFC_hpmcounter27_idx,      //PMC
62      PFC_hpmcounter28_idx,      //PMC
63      PFC_hpmcounter29_idx,      //PMC
64      PFC_hpmcounter30_idx,      //PMC
65      PFC_hpmcounter31_idx,      //PMC
66      PFC_END_idx               //This should always be last!
67 };
68
69 #define UPFC_CYCLE             (PFC_CYCLE_idx          |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged Cycle counter for RDCYCLE
       Instruction
70 #define UPFC_TIME              (PFC_TIME_idx           |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged Timer for RDTIME
       instruction
71 #define UPFC_INSTRET           (PFC_INSTRET_idx        |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged Instructions-retired
       counter for RDINSTRET instruction
72 #define UPFC_hpmcounter3       (PFC_hpmcounter3_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
73 #define UPFC_hpmcounter4       (PFC_hpmcounter4_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
74 #define UPFC_hpmcounter5       (PFC_hpmcounter5_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
75 #define UPFC_hpmcounter6       (PFC_hpmcounter6_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
```

```
76 #define UPFC_hpmcounter7      (PFC_hpmcounter7_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
77 #define UPFC_hpmcounter8      (PFC_hpmcounter8_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
78 #define UPFC_hpmcounter9      (PFC_hpmcounter9_idx    |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
79 #define UPFC_hpmcounter10     (PFC_hpmcounter10_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
80 #define UPFC_hpmcounter11     (PFC_hpmcounter11_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
81 #define UPFC_hpmcounter12     (PFC_hpmcounter12_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
82 #define UPFC_hpmcounter13     (PFC_hpmcounter13_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
83 #define UPFC_hpmcounter14     (PFC_hpmcounter14_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
84 #define UPFC_hpmcounter15     (PFC_hpmcounter15_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
85 #define UPFC_hpmcounter16     (PFC_hpmcounter16_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
86 #define UPFC_hpmcounter17     (PFC_hpmcounter17_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
87 #define UPFC_hpmcounter18     (PFC_hpmcounter18_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
88 #define UPFC_hpmcounter19     (PFC_hpmcounter19_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
89 #define UPFC_hpmcounter20     (PFC_hpmcounter20_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
90 #define UPFC_hpmcounter21     (PFC_hpmcounter21_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
91 #define UPFC_hpmcounter22     (PFC_hpmcounter22_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
92 #define UPFC_hpmcounter23     (PFC_hpmcounter23_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
93 #define UPFC_hpmcounter24     (PFC_hpmcounter24_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
94 #define UPFC_hpmcounter25     (PFC_hpmcounter25_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
95 #define UPFC_hpmcounter26     (PFC_hpmcounter26_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
96 #define UPFC_hpmcounter27     (PFC_hpmcounter27_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
97 #define UPFC_hpmcounter28     (PFC_hpmcounter28_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
98 #define UPFC_hpmcounter29     (PFC_hpmcounter29_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
99 #define UPFC_hpmcounter30     (PFC_hpmcounter30_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
100 #define UPFC_hpmcounter31     (PFC_hpmcounter31_idx   |
       UNPRIV_COUNTERS_TIMER_MASK) //Unprivileged PMC
101
102 #define UPFC_END              (PFC_END_idx            |
       UNPRIV_COUNTERS_TIMER_MASK) //This should always be last!
103
104 #define UPFC_CYCLEH           (PFC_CYCLE_idx          |
       UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged
       Cycle counter for RDCYCLE Instruction
```

```
105 #define UPFC_TIMEH            (PFC_TIME_idx          |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged
        Timer for RDTIME instruction
106 #define UPFC_INSTRETH         (PFC_INSTRET_idx       |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged
        Instructions-retired counter for RDINSTRET instruction
107 #define UPFC_hpmcounter3H     (PFC_hpmcounter3_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
108 #define UPFC_hpmcounter4H     (PFC_hpmcounter4_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
109 #define UPFC_hpmcounter5H     (PFC_hpmcounter5_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
110 #define UPFC_hpmcounter6H     (PFC_hpmcounter6_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
111 #define UPFC_hpmcounter7H     (PFC_hpmcounter7_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
112 #define UPFC_hpmcounter8H     (PFC_hpmcounter8_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
113 #define UPFC_hpmcounter9H     (PFC_hpmcounter9_idx   |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
114 #define UPFC_hpmcounter10H    (PFC_hpmcounter10_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
115 #define UPFC_hpmcounter11H    (PFC_hpmcounter11_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
116 #define UPFC_hpmcounter12H    (PFC_hpmcounter12_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
117 #define UPFC_hpmcounter13H    (PFC_hpmcounter13_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
118 #define UPFC_hpmcounter14H    (PFC_hpmcounter14_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
119 #define UPFC_hpmcounter15H    (PFC_hpmcounter15_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
120 #define UPFC_hpmcounter16H    (PFC_hpmcounter16_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
121 #define UPFC_hpmcounter17H    (PFC_hpmcounter17_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
122 #define UPFC_hpmcounter18H    (PFC_hpmcounter18_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
123 #define UPFC_hpmcounter19H    (PFC_hpmcounter19_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
124 #define UPFC_hpmcounter20H    (PFC_hpmcounter20_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
125 #define UPFC_hpmcounter21H    (PFC_hpmcounter21_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
126 #define UPFC_hpmcounter22H    (PFC_hpmcounter22_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
127 #define UPFC_hpmcounter23H    (PFC_hpmcounter23_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
128 #define UPFC_hpmcounter24H    (PFC_hpmcounter24_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
129 #define UPFC_hpmcounter25H    (PFC_hpmcounter25_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
130 #define UPFC_hpmcounter26H    (PFC_hpmcounter26_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
131 #define UPFC_hpmcounter27H    (PFC_hpmcounter27_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
132 #define UPFC_hpmcounter28H    (PFC_hpmcounter28_idx  |
        UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
```

```
133 #define UPFC_hpmcounter29H    (PFC_hpmcounter29_idx   |
    UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
134 #define UPFC_hpmcounter30H    (PFC_hpmcounter30_idx   |
    UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
135 #define UPFC_hpmcounter31H    (PFC_hpmcounter31_idx   |
    UNPRIV_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Unprivileged PMC
136
137 #define UPFC_ENDH             (PFC_END_idx            |
    UNPRIV_UPPER_COUNTERS_TIMER_MASK) //This should always be last!
138
139 #define MPFC_CYCLE            (PFC_CYCLE_idx          |
    MACHINE_COUNTERS_TIMER_MASK) //Machine Cycle counter for RDCYCLE
    Instruction
140 #define MPFC_TIME             (PFC_TIME_idx           |
    MACHINE_COUNTERS_TIMER_MASK) //Machine Timer for RDTIME instruction
141 #define MPFC_INSTRET          (PFC_INSTRET_idx        |
    MACHINE_COUNTERS_TIMER_MASK) //Machine Instructions-retired counter
    for RDINSTRET instruction
142 #define MPFC_hpmcounter3      (PFC_hpmcounter3_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
143 #define MPFC_hpmcounter4      (PFC_hpmcounter4_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
144 #define MPFC_hpmcounter5      (PFC_hpmcounter5_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
145 #define MPFC_hpmcounter6      (PFC_hpmcounter6_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
146 #define MPFC_hpmcounter7      (PFC_hpmcounter7_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
147 #define MPFC_hpmcounter8      (PFC_hpmcounter8_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
148 #define MPFC_hpmcounter9      (PFC_hpmcounter9_idx    |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
149 #define MPFC_hpmcounter10     (PFC_hpmcounter10_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
150 #define MPFC_hpmcounter11     (PFC_hpmcounter11_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
151 #define MPFC_hpmcounter12     (PFC_hpmcounter12_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
152 #define MPFC_hpmcounter13     (PFC_hpmcounter13_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
153 #define MPFC_hpmcounter14     (PFC_hpmcounter14_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
154 #define MPFC_hpmcounter15     (PFC_hpmcounter15_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
155 #define MPFC_hpmcounter16     (PFC_hpmcounter16_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
156 #define MPFC_hpmcounter17     (PFC_hpmcounter17_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
157 #define MPFC_hpmcounter18     (PFC_hpmcounter18_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
158 #define MPFC_hpmcounter19     (PFC_hpmcounter19_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
159 #define MPFC_hpmcounter20     (PFC_hpmcounter20_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
160 #define MPFC_hpmcounter21     (PFC_hpmcounter21_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
161 #define MPFC_hpmcounter22     (PFC_hpmcounter22_idx   |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
```

```
162 #define MPFC_hpmcounter23    (PFC_hpmcounter23_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
163 #define MPFC_hpmcounter24    (PFC_hpmcounter24_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
164 #define MPFC_hpmcounter25    (PFC_hpmcounter25_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
165 #define MPFC_hpmcounter26    (PFC_hpmcounter26_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
166 #define MPFC_hpmcounter27    (PFC_hpmcounter27_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
167 #define MPFC_hpmcounter28    (PFC_hpmcounter28_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
168 #define MPFC_hpmcounter29    (PFC_hpmcounter29_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
169 #define MPFC_hpmcounter30    (PFC_hpmcounter30_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
170 #define MPFC_hpmcounter31    (PFC_hpmcounter31_idx  |
    MACHINE_COUNTERS_TIMER_MASK) //Machine PMC
171
172 #define UPFC_END             (PFC_END_idx           |
    UNPRIV_COUNTERS_TIMER_MASK) //This should always be last!
173
174 #define MPFC_CYCLEH          (PFC_CYCLE_idx         |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine Cycle
    counter for RDCYCLE Instruction
175 #define MPFC_TIMEH           (PFC_TIME_idx          |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine Timer
    for RDTIME instruction
176 #define MPFC_INSTRETH        (PFC_INSTRET_idx       |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine
    Instructions-retired counter for RDINSTRET instruction
177 #define MPFC_hpmcounter3H    (PFC_hpmcounter3_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
178 #define MPFC_hpmcounter4H    (PFC_hpmcounter4_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
179 #define MPFC_hpmcounter5H    (PFC_hpmcounter5_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
180 #define MPFC_hpmcounter6H    (PFC_hpmcounter6_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
181 #define MPFC_hpmcounter7H    (PFC_hpmcounter7_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
182 #define MPFC_hpmcounter8H    (PFC_hpmcounter8_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
183 #define MPFC_hpmcounter9H    (PFC_hpmcounter9_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
184 #define MPFC_hpmcounter10H   (PFC_hpmcounter10_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
185 #define MPFC_hpmcounter11H   (PFC_hpmcounter11_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
186 #define MPFC_hpmcounter12H   (PFC_hpmcounter12_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
187 #define MPFC_hpmcounter13H   (PFC_hpmcounter13_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
188 #define MPFC_hpmcounter14H   (PFC_hpmcounter14_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
189 #define MPFC_hpmcounter15H   (PFC_hpmcounter15_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
190 #define MPFC_hpmcounter16H   (PFC_hpmcounter16_idx  |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
```

```
191 #define MPFC_hpmcounter17H    (PFC_hpmcounter17_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
192 #define MPFC_hpmcounter18H    (PFC_hpmcounter18_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
193 #define MPFC_hpmcounter19H    (PFC_hpmcounter19_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
194 #define MPFC_hpmcounter20H    (PFC_hpmcounter20_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
195 #define MPFC_hpmcounter21H    (PFC_hpmcounter21_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
196 #define MPFC_hpmcounter22H    (PFC_hpmcounter22_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
197 #define MPFC_hpmcounter23H    (PFC_hpmcounter23_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
198 #define MPFC_hpmcounter24H    (PFC_hpmcounter24_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
199 #define MPFC_hpmcounter25H    (PFC_hpmcounter25_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
200 #define MPFC_hpmcounter26H    (PFC_hpmcounter26_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
201 #define MPFC_hpmcounter27H    (PFC_hpmcounter27_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
202 #define MPFC_hpmcounter28H    (PFC_hpmcounter28_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
203 #define MPFC_hpmcounter29H    (PFC_hpmcounter29_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
204 #define MPFC_hpmcounter30H    (PFC_hpmcounter30_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
205 #define MPFC_hpmcounter31H    (PFC_hpmcounter31_idx   |
    MACHINE_UPPER_COUNTERS_TIMER_MASK) //Upper 32 bits of Machine PMC
206
207 #define UPFC_ENDH             (PFC_END_idx            |
    UNPRIV_UPPER_COUNTERS_TIMER_MASK) //This should always be last!
208
209 #define MPFC_EVENT_SEL3       (PFC_hpmcounter3_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 3
210 #define MPFC_EVENT_SEL4       (PFC_hpmcounter4_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 4
211 #define MPFC_EVENT_SEL5       (PFC_hpmcounter5_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 5
212 #define MPFC_EVENT_SEL6       (PFC_hpmcounter6_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 6
213 #define MPFC_EVENT_SEL7       (PFC_hpmcounter7_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 7
214 #define MPFC_EVENT_SEL8       (PFC_hpmcounter8_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 8
215 #define MPFC_EVENT_SEL9       (PFC_hpmcounter9_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 9
216 #define MPFC_EVENT_SEL10      (PFC_hpmcounter10_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 10
```

```
217 #define MPFC_EVENT_SEL11      (PFC_hpmcounter11_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 11
218 #define MPFC_EVENT_SEL12      (PFC_hpmcounter12_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 12
219 #define MPFC_EVENT_SEL13      (PFC_hpmcounter13_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 13
220 #define MPFC_EVENT_SEL14      (PFC_hpmcounter14_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 14
221 #define MPFC_EVENT_SEL15      (PFC_hpmcounter15_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 15
222 #define MPFC_EVENT_SEL16      (PFC_hpmcounter16_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 16
223 #define MPFC_EVENT_SEL17      (PFC_hpmcounter17_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 17
224 #define MPFC_EVENT_SEL18      (PFC_hpmcounter18_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 18
225 #define MPFC_EVENT_SEL19      (PFC_hpmcounter19_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 19
226 #define MPFC_EVENT_SEL20      (PFC_hpmcounter20_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 20
227 #define MPFC_EVENT_SEL21      (PFC_hpmcounter21_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 21
228 #define MPFC_EVENT_SEL22      (PFC_hpmcounter22_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 22
229 #define MPFC_EVENT_SEL23      (PFC_hpmcounter23_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 23
230 #define MPFC_EVENT_SEL24      (PFC_hpmcounter24_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 24
231 #define MPFC_EVENT_SEL25      (PFC_hpmcounter25_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 25
232 #define MPFC_EVENT_SEL26      (PFC_hpmcounter26_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 26
233 #define MPFC_EVENT_SEL27      (PFC_hpmcounter27_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 27
234 #define MPFC_EVENT_SEL28      (PFC_hpmcounter28_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 28
235 #define MPFC_EVENT_SEL29      (PFC_hpmcounter29_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 29
```

```
236 #define MPFC_EVENT_SEL30      (PFC_hpmcounter30_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 30
237 #define MPFC_EVENT_SEL31      (PFC_hpmcounter31_idx    |
    MPM_EVENT_SELECTOR_START_MASK) // Machine performance-monitoring
    event selector 31
238
239 #define MPFC_EVENT_SEL_END    (PFC_END_idx             |
    MPM_EVENT_SELECTOR_START_MASK) //This should always be last!
240
241 #define MPFC_MCOUNINHIBIT (int) 0x320
242 #define MPFC_MCOUNTEREN (int) 0x306
243
244 #endif // _RISCV_CSR
```

Listing A.1: CSR-Address mapping in C for ISA CSR addresses

Appendix B

Register Configuration Code in C

```
#ifndef configSYSTEM_CFG
#define configSYSTEM_CFG  {                 \
    .hpm_counters = {                       \
        {                                   \
            .address = MPFC_CYCLE,          \
            .type = reg_hpmcounter,         \
            .id = csr_cycle                 \
        },                                  \
        {                                   \
            .address = 0x1808,              \
            .type = reg_hpmcounter,         \
            .id = csr_time                  \
        },                                  \
        {                                   \
            .address = MPFC_INSTRET,        \
            .type = reg_hpmcounter,         \
            .id = csr_instret               \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter3,    \
            .type = reg_hpmcounter,         \
            .id = csr_hpmcounter3           \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter4,    \
            .type = reg_hpmcounter,         \
            .id = csr_hpmcounter4           \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter5,    \
            .type = reg_hpmcounter,         \
            .id = csr_hpmcounter5           \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter6,    \
            .type = reg_hpmcounter,         \
            .id = csr_hpmcounter6           \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter7,    \
            .type = reg_hpmcounter,         \
            .id = csr_hpmcounter7           \
        },                                  \
        {                                   \
            .address = MPFC_hpmcounter8,    \
            .type = reg_hpmcounter,         \
```

```
47              .id = csr_hpmcounter8           \
48          },                                  \
49          {                                   \
50              .address = MPFC_hpmcounter9,    \
51              .type = reg_hpmcounter,         \
52              .id = csr_hpmcounter9           \
53          },                                  \
54          {                                   \
55              .address = MPFC_hpmcounter10,   \
56              .type = reg_hpmcounter,         \
57              .id = csr_hpmcounter10          \
58          },                                  \
59          {                                   \
60              .address = MPFC_hpmcounter11,   \
61              .type = reg_hpmcounter,         \
62              .id = csr_hpmcounter11          \
63          },                                  \
64          {                                   \
65              .address = MPFC_hpmcounter12,   \
66              .type = reg_hpmcounter,         \
67              .id = csr_hpmcounter12          \
68          }                                   \
69      },                                       \
70      .upper_hpm_counters = {                 \
71          {                                   \
72              .address = MPFC_CYCLEH,         \
73              .type = reg_hpmcounter,         \
74              .id = csr_cycle                 \
75          },                                  \
76          {                                   \
77              .address = 0x1828,              \
78              .type = reg_hpmcounter,         \
79              .id = csr_time                  \
80          },                                  \
81          {                                   \
82              .address = MPFC_INSTRETH,       \
83              .type = reg_hpmcounter,         \
84              .id = csr_instret               \
85          },                                  \
86          {                                   \
87              .address = MPFC_hpmcounter3H,   \
88              .type = reg_hpmcounter,         \
89              .id = csr_hpmcounter3           \
90          },                                  \
91          {                                   \
92              .address = MPFC_hpmcounter4H,   \
93              .type = reg_hpmcounter,         \
94              .id = csr_hpmcounter4           \
95          },                                  \
96          {                                   \
97              .address = MPFC_hpmcounter5H,   \
98              .type = reg_hpmcounter,         \
99              .id = csr_hpmcounter5           \
100         },                                  \
101         {                                   \
102             .address = MPFC_hpmcounter6H,   \
103             .type = reg_hpmcounter,         \
104             .id = csr_hpmcounter6           \
105         },                                  \
```

```
106         {                                       \
107             .address = MPFC_hpmcounter7H,       \
108             .type = reg_hpmcounter,             \
109             .id = csr_hpmcounter7               \
110         },                                      \
111         {                                       \
112             .address = MPFC_hpmcounter8H,       \
113             .type = reg_hpmcounter,             \
114             .id = csr_hpmcounter8               \
115         },                                      \
116         {                                       \
117             .address = MPFC_hpmcounter9H,       \
118             .type = reg_hpmcounter,             \
119             .id = csr_hpmcounter9               \
120         },                                      \
121         {                                       \
122             .address = MPFC_hpmcounter10H,      \
123             .type = reg_hpmcounter,             \
124             .id = csr_hpmcounter10              \
125         },                                      \
126         {                                       \
127             .address = MPFC_hpmcounter11H,      \
128             .type = reg_hpmcounter,             \
129             .id = csr_hpmcounter11              \
130         },                                      \
131         {                                       \
132             .address = MPFC_hpmcounter12H,      \
133             .type = reg_hpmcounter,             \
134             .id = csr_hpmcounter12              \
135         }                                       \
136     },                                          \
137     .hpm_custom_csr = {                         \
138         {                                       \
139             .address = CSR_ADDRESS_mpcer,       \
140             .type = reg_hpmevent,               \
141             .id = mpcer                         \
142         },                                      \
143         {                                       \
144             .address = CSR_ADDRESS_mpcmr,       \
145             .type = reg_config_en,              \
146             .id = mpcmr                         \
147         },                                      \
148         {                                       \
149             .address = CSR_ADDRESS_mpccr,       \
150             .type = reg_hpmcounter,             \
151             .id = mpccr                         \
152         }                                       \
153     }                                           \
154 }
155 #endif
```

Listing B.1: esp32c6 system's registers configuration and mapping.