Design and implementation of a bootrom in a Linux capable RISC-V processor

Jordi Garcia Aguilar

Supervised by:

Miquel Moretó Planas

Max Doblas Font

Victor Soria Pardos

In partial fulfillment of the requirements for the degree in:

Bachelor's degree in Informatics Engineering

January 2022

Abstract

The open-source movement promises to revolutionize the hardware world just as it has revolutionized software. Thanks to the open-source RISC-V instruction set architecture (ISA), many projects are making their way to offer an alternative in the hermetic and proprietary world of computer architecture.

The DRAC project, whose acronym refers to Designing RISC-V-based Accelerators for next-generation Computers, was created in this context. This project, led by the Barcelona Supercomputing Center (BSC), develops processors and accelerators based on RISC-V technology, and their purpose is to accelerate security tasks, personalized medicine and autonomous navigation.

This thesis aims to design, implement and verify a bootrom for the 64-bit *DRAC 22 nm* System on Chip (SoC). This SoC integrates an in-order 7-stage RISC-V core called Sargantana. The SoC design, based on the previous tape-out *PreDRAC*, is divided into two parts. One part contains all the ASIC-oriented components; the other contains the FPGA-oriented components. One of the main reasons for this division is that there was no ASIC-oriented bootrom, and therefore, it was necessary to use the FPGA to boot the chip. With the integration of the bootrom developed in this thesis, the SoC will be able to boot by itself, eliminating the FPGA-oriented part of the SoC design.

Keywords: Bootrom, Bootloader, DRAC, RTL design, RISC-V, SoC

Resum

El moviment de codi obert promet revolucionar el món del maquinari igual que el programari ha revolucionat. Gràcies a l'arquitectura de conjunt d'instruccions o ISA (de l'anglès *Instruction Set Architecture*) RISC-V de codi obert, molts projectes s'estan obrint camí per oferir una alternativa a l'hermètic i privatiu món de l'arquitectura de computadors.

En aquest context neix el projecte DRAC, les sigles del qual fan referència, de l'anglès, a *Designing RISC-V-based Accelerators for next-generation Computers.* Aquest projecte, liderat pel Barcelona Supercomputing Center (BSC), desenvolupa processadors i acceleradors basats en la tecnologia RISC-V, amb l'objectiu d'accelerar tasques de seguretat, medicina personalitzada i navegació autònoma.

Aquesta tesi té com a propòsit dissenyar, implementar i verificar una bootrom pel SoC (de l'anglès *System on Chip*) de 64 bits *DRAC 22 nm.* Aquest SoC integra un processador RISC-V de 7 etapes anomenat Sargantana. El disseny del SoC, basat en l'anterior *tape-out* anomenat *PreDRAC*, es divideix en dues parts. Una part conté tots els components orientats a l'ASIC; l'altra conté els elements orientats a la FPGA. Una de les raons principals d'aquesta divisió és que no existia una bootrom orientada a ASIC i, per tant, calia utilitzar la FPGA per arrencar el xip. Amb la integració de la bootrom desenvolupada en aquesta tesi, el SoC serà capaç d'arrencar per ell mateix, eliminant la part orientada a la FPGA del disseny del SoC.

Paraules clau: Bootrom, Bootloader, DRAC, RTL design, RISC-V, SoC

Resumen

El movimiento de código abierto promete revolucionar el mundo del hardware igual que ha revolucionado el software. Gracias a la arquitectura de conjunto de instrucciones o ISA (del inglés *Instruction Set Architecture*) RISC-V de código abierto, muchos proyectos se están abriendo camino para ofrecer una alternativa en el hermético y privativo mundo de la arquitectura de computadores.

En este contexto nace el proyecto DRAC, cuyas siglas hacen referencia, del inglés, a *Designing RISC-V-based Accelerators for next-generation Computers.* Este proyecto, liderado por el Barcelona Supercomputing Center (BSC), desarrolla procesadores y aceleradores basados en la tecnología RISC-V, y su objetivo es acelerar tareas de seguridad, medicina personalizada y navegación autónoma.

Esta tesis tiene como objetivo diseñar, implementar y verificar una bootrom para el SoC (del inglés *System on Chip*) de 64 bits *DRAC 22 nm.* Este SoC integra un procesador RISC-V de 7 etapas llamado Sargantana. El diseño del SoC, basado en el anterior *tape-out* denominado *PreDRAC*, se divide en dos partes. Una parte contiene todos los componentes orientados al ASIC; la otra contiene los elementos orientados a la FPGA. Una de las principales razones de esta división es que no existía una bootrom orientada a ASIC y, por tanto, era necesario utilizar la FPGA para arrancar el chip. Con la integración de la bootrom desarrollada en esta tesis, el SoC será capaz de arrancar por sí mismo, eliminando la parte orientada a la FPGA del diseño del SoC.

Palabras clave: Bootrom, Bootloader, DRAC, RTL design, RISC-V, SoC

Contents

List of Figures

List of Tables

Acronyms

ASIC	Application-Specific Integrated Circuit
CSR	Control and Status Register
PCR	Processor Control Register
FPGA	Field-Programmable Gate Array
ISA	Instruction Set Architecture
RTL	Register-Transfer Level
SoC	System on Chip
BHT	Branch History Table
BTB	Branch Target Buffer
PC	Program Counter
VIPT	Virtually Indexed, Physically Tagged
TLB	Translation Lookaside Buffer
ABI	Application Binary Interface
SBI	Supervisor Binary Interface
AEE	Application Execution Environment
SEE	Supervisor Execution Environment
OS	Operating System
GLS	Gate-Level Simulation
EEPROM	Electrically Erasable Programmable Read-Only Memory
CSU	Clock Selection Unit
PLL	Phased-Locked Loop
FMC	FPGA Mezzanine Card

1 Introduction

1.1 Context

This work is part of the Barcelona Supercomputing Center DRAC (Designing RISC-V-based Accelerators for next-generation Computers) project [1]. In the DRAC project collaborates the Institute of Microelectronics of Barcelona, the Barcelona Supercomputing Center, the Instituto Politécnico Nacional (Mexico) and UPC, among others. DRAC's principal aim is to develop processors based on the RISC-V ISA.

In order to achieve that goal, BSC is developing System on Chip (SoC) that includes RISC-V processors. The first SoC of these series, PreDRAC [1], was built at the end of 2019 as the first open-source chip developed in Spain. So, at this point, the BSC team is improving the initial design to manufacture the next SoC in early 2022. The work detailed on these thesis is designing and implementing the hardware and software components required to boot a Linux kernel using a custom bootrom controller.

1.2 Terms and concepts

As we want a self-contained document, we must introduce some terms and concepts.

1.2.1 Hardware Description Language

In the computer engineering field, a hardware description language (HDL) is used to describe the structure and behavior of digital logic circuits. HDL gives a precise, formal description of a digital circuit that enables an automated analysis and simulation. Also, a circuit described with HDL can be synthesized into a netlist (a specification of physical, electronic components, and the connecting rooting of the components), which can then be placed and routed to produce the set of masks used to manufacture the integrated circuit.

[1]The DRAC project[7] (with file number 001-P-001723) has been 50% co-financed with €2,000,000.00 by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020, with the support of Generalitat of Catalonia.

1.2.2 SystemVerilog

SystemVerilog [32], standardized as IEEE 1800, aims to provide a well-defined and official IEEE unified hardware design, specification, and verification standard language. It is commonly used in the semiconductor and electronic design industry as an evolution of Verilog. The language is designed to coexist and enhance the hardware description and verification languages (HDVLs) currently used by designers while providing the capabilities lacking in those languages. SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, specification of assertions, coverage, and testbench verification based on manual or automatic methodologies.

1.2.3 Chisel

Chisel [5] is an HDL that allows advanced circuit design and generation. The same design can be reused for both Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) implementations.

Chisel adds hardware construction primitives to the Scala programming language. It gives the power of a modern programming language to the computer architects that enables designing complex, parameterizable circuit generators that produce synthesizable Verilog. This generator methodology allows the conception of re-usable components and libraries, increasing abstraction in design while maintaining fine-grained control.

Chisel is powered by FIRRTL (Flexible Intermediate Representation for RTL), a hardware compiler framework that performs optimizations of Chisel-generated circuits and supports custom user-defined circuit transformations.

1.2.4 RTL

In digital circuit design, Register-Transfer Level (RTL) is a design abstraction that models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers and the logical operations performed on those signals.

Register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is typical practice in modern digital design [39].

1.2.5 FPGA

A FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term field-programmable. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an ASIC. Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects, allowing blocks to be wired together. Logic blocks can be configured to perform complex combinational functions or act as simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete memory blocks. Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible, reconfigurable computing as performed in computer software [37].

1.2.6 Bootrom

A bootrom (or bootROM) is a small Read-Only Memory (ROM) that contains the very first code executed by a processor on a reset or power-on. Hence this code contains instructions to configure the SoC to allow the execution of applications. The configurations performed by bootrom include initialization of the core's register and stack pointer, enablement of caches and line buffers, programming of interrupt service routine, and clock configuration.

1.2.7 ISA

An Instruction Set Architecture (ISA) is an abstract model of a computer. It defines the instructions, registers, data types, the hardware support for managing memory or input/output devices, and specifies the behavior of machine code running on that ISA [38].

1.2.8 ASIC

An application-specific integrated circuit is an integrated circuit (IC) chip customized for a particular use rather than intended for general-purpose use. For example, a chip designed to run in a digital voice recorder or a high-efficiency Bitcoin miner is an ASIC [36].

1.2.9 SoC

A system on a chip, also known as an SoC, is essentially an integrated circuit or an IC that takes a single platform and integrates an entire electronic or computer system onto it. It is, precisely as its name suggests, an entire system on a single chip. The components that an SoC generally looks to incorporate within itself include a central processing unit, input and output ports, internal memory, and analog input and output blocks. Depending on the kind of system that has been reduced to the size of a chip, it can perform a variety of functions, including signal processing, wireless communication, artificial intelligence, and more [2].

1.3 Problem to be resolved

In recent months, it has not gone unnoticed that the technology industry is currently suffering from a severe shortage of chips. This fact has seriously affected many European countries that in many cases have had to stop production, raise prices or take long delays in the distribution of new products. The leading cause of this situation is that, until now, Europe could not create new chips, as the vast majority of ISAs and designs were closed and privative. In recent years, the

Open Source movement has reached the hardware design, which so far has resisted, promising a revolution similar to the one that Open Source brought to the software.

Thanks to the open RISC-V ISA, nowadays, Europe is allowed to design and manufacture its chips instead of buying them abroad [43]. Thus, acquiring technological sovereignty is necessary to decide on the future of the technological industry, making it more competitive and secure.

1.4 Stakeholders

The project has many direct or indirect involved parties detailed below.

1.4.1 Barcelona Supercomputing Center

The primary stakeholder that will make the most out of this work is the DRAC team at the BSC. DRAC is an experimental vehicle for academic purposes that can be used as a base platform to perform research based on RISC-V processors or accelerators

1.4.2 Scientific community

The community of European Supercomputation is interested in the project results to evaluate the viability of the RISC-V European Processor Initiative.

1.4.3 University education

The DRAC project aims to be provided under an open-source license. Therefore, academic institutions could leverage the work done to aid students in getting familiar with RTL design and RISC-V or as a base platform for computer architecture research.

1.5 Justification

The current SoC design is divided into two major parts: the ASIC and the FPGA implementation. There are specific components essential for running an operating system like Linux on the SoC that cannot be implemented for ASIC due to economic or time constraints, for example, the DDR3 controller. Figure 1.1 shows a diagram of the SoC to start making improvements for the next tape-out.

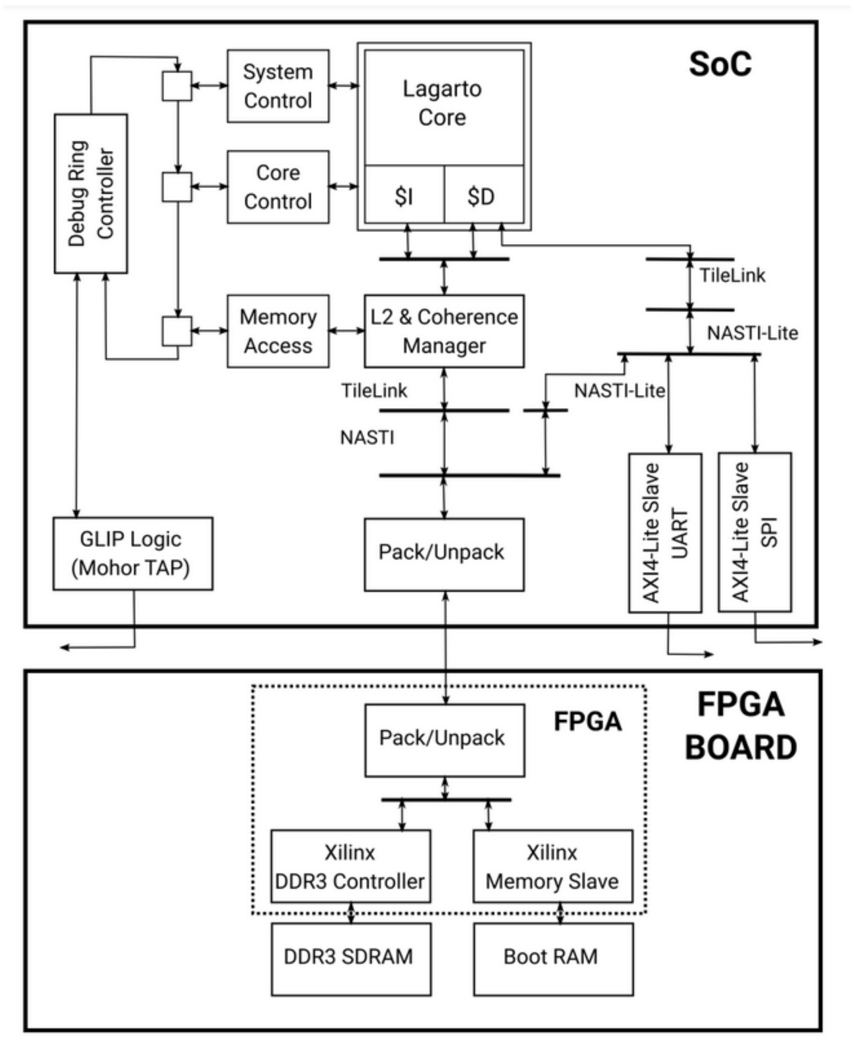


Figure 1.1: PreDRAC block diagram. Source: [1]

The main objective of this work is to implement a custom bootrom that allows the boot process not to go through the FPGA. Making the SoCs entirely work alone is of significant importance to the project. It would release, depending on the configuration, up to 68 pins used by the ASIC-to-FPGA interface, called *packetizer* or the *Pack/Unpack* blocks in Figure 1.1, which would be available for future improvements and functionalities of the chip.

1.6 Project scope

This section will define the global objectives and sub-objectives, the functional requirements, and the possible obstacles and risks of the project.

1.6.1 Objectives

Next, the objectives that must be followed to accomplish the desired task.

- **Understand the SoC implementation:** Understanding in detail the basis is essential to know all the options or limitations when developing and integrating the bootrom.

- **Analyze similar existing projects:** Analyzing what decisions have been made and what criteria have been taken into account in other similar projects will serve as a guide to define the parameters to be taken into account in the implementation of this project.
- **Evaluate different bootrom technologies and sizes:** Analyzing the advantages and disadvantages of using a specific type of memory is crucial to avoid unwanted restrictions in the future.
- **Evaluate different implementation strategies:** Analyze the different options for implementing the bootrom. The implementation and integration of the bootrom should be modular to allow suitable modifications in future phases of the project if needed.
- **Be on time for the tape-out:** Have a functional and verified bootrom for the SoC tape-out, expected in early 2022. This way, it will be possible to verify that the proposed design is correct definitively.
- **Provide a backup mechanism in case of failure:** The bootrom is a crucial element of the SoC. Suppose, after chip fabrication, the boot process fails for some reason. In that case, other SoC components and aspects developed in parallel with this project, such as the hyperRAM controller, or core enhancements, cannot be evaluated. Therefore, it is necessary to have an alternative plan.
- **Ensure a boot-ready environment:** We want to add an ASIC-oriented bootrom controller capable of booting Linux for end-user usage. This implies possible changes in the bootloader or other aspects of the SoC.
- **Evaluate and verify the implementation:** Once we have the bootrom controller integrated, it is necessary to analyze what advantages and disadvantages it has brought to the whole chip. If there are possible improvements, define how costly it would be to implement them.

1.6.2 Requirements

There are only two fixed requirements, namely:

- The implementation of the bootrom should minimize the pins used and it must not exceed the available pins of the chip.
- The implementation of the bootrom must be completed before the RTL freeze, estimated in January 2022.
- The SoC after the bootrom integration must be able to boot the Linux operating system.

1.6.3 Obstacles and risks

During the execution of the project, we can classify the obstacles and risks we may encounter into two categories: technical problems and synchronization problems with the team. Possibles obstacles in the first category are:

- Implementation errors
- Hardware failures
- Difficulties in tools environment set-ups
- Debug or verification environment limitations

Regarding the synchronization obstacles, some of them that we can expect are:

- Misunderstandings with other team parts
- Delays of depending tasks of the rest of the team
- Lack of documentation

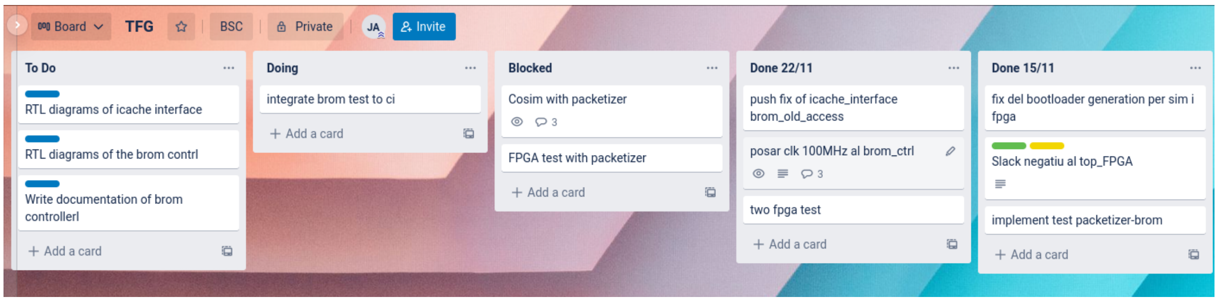
It is worth mentioning that there is a risk that the current health situation may make it more challenging to coordinate with the rest of the group, as it is very likely that either because of recommendations from the authorities or because of Covid-19, we will have to stay at home. We can also expect a lack of stock or delays in the shipment of material needed for the project.

1.7 Methodology and rigour

This section lists the different tools used to develop the project and coordinate with the team.

1.7.1 Trello

Trello [33] is a collaboration tool that organizes projects into boards. It is intended to be used by team members, although it is also handy for only one person, as it allows to keep track of the tasks carried out.



To Do
- RTL diagrams of icache interface
- RTL diagrams of the brom contrl
- Write documentation of brom controllerl

Doing
- integrate brom test to ci

Blocked
- Cosim with packetizer 3
- FPGA test with packetizer

Done 22/11
- push fix of icache_interface brom_old_access
- posar clk 100MHz al brom_ctrl 3
- two fpga test

Done 15/11
- fix del bootloader generation per sim i fpga
- Slack negatiu al top_FPGA
- implement test packetizer-brom

Figure 1.2: Trello organization boards

In our case, we will follow a variant of the Kanban methodology consisting of the boards shown in Figure 1.2:

- **To Do:** In this board, we will create a card for each unstarted task that remains to be done.
- **Doing:** In this board, we will place the tasks of the previous group with which we have started to work or are currently working.
- **Blocked:** This board is used to classify those tasks with which we were working, but for some reason, we can not continue their development, and we must leave them for later. In this way, this board will avoid us that the group of tasks in the Doing board is always updated with the tasks we work on in a concrete moment.
- **Done:** In this board, we will put the finished tasks. We have decided to create a table for each week. This way, we can keep the chronology of the project development.

1.7.2 Slack

Slack [30] is a communication tool that gives us access to real-time conversations with any team or teammate. It is common for team members to work from home, which makes Slack a valuable tool for coordinating and contacting someone when there is an emergency, and we expect a quick response.

1.7.3 Git

Git is a code version manager. The DRAC team has a private Git server to see other teammates' code and update the code as we advance in the development process.

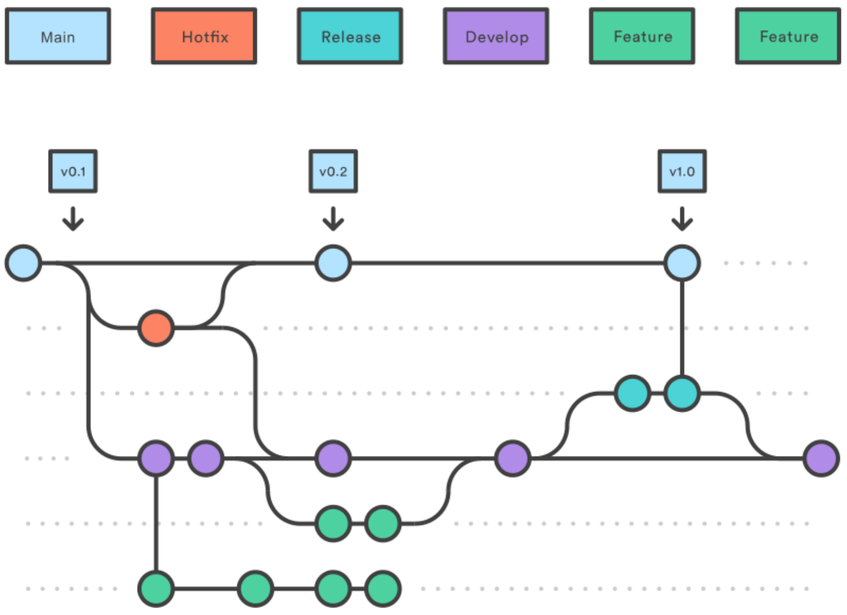


Figure 1.3: GitFlow workflow branch diagram. Source: [4]

Working with Git, we follow the GitFlow workflow (see Figure 1.3), a strategy for organizing the different aspects of code development into five types of git branches: Main, Hotfix, Release, Develop, Feature.

1.7.4 Team meetings

Finally, Miquel Moretó, the team manager, organizes two weekly meetings. The first one is focused on the physical and design group. We discuss the materials and resources that we need to build our processor in this meeting. Also, we talk about the progress of each design project. The other meeting intends each subgroup of the DRAC team to share its project progress.

1.8 Description of tasks

In this section, we will specify the definition of the tasks related to the realization of the project. The project will last 670 hours, starting on October 1, and will end in late January. The tasks are grouped into three categories: Project Management, Project Development, and Project Conclusion. The tasks will be accompanied by a brief description, their dependencies, when available, the resources needed, and an estimate of their duration.

1.8.1 Project Management

In this category, we can find those tasks before the start of project development, namely:

- **PM1 - Project scope and context:** This task will take 20 hours and a computer with a text editor to write and detail the context and scope of the project.
- **PM2 - Project planning:** Defining the project planning will require 20 hours and a computer with a text editor.
- **PM3 - Budget and sustainability:** Defining the budget and sustainability requires 20 hours and a computer with a text editor.
- **PM4 - integration into a final document:** This task requires the completion of the previous tasks (PM1, PM2, and PM3). Writing a final document with all the project information will demand 15 hours and a text editor.
- **PM5 - Weekly meetings:** One-hour weekly meetings will be held, a total of 15 hours. This task requires the participation of the entire DRAC team and a computer to attend the meetings.

1.8.2 Project Development

The tasks related to the development of the project are defined below. Most of these tasks have detailed subtasks in the same way. Most of the tasks listed require a computer as the main material resource.

1.8.2.1 PD1 - Research and state of the art

- **PD1.1 - Familiarize PreDRAC architecture:** Understand the architecture limitations and reevaluate the different options and strategies for the implementation of the bootrom. This task will require 20 hours.
- **PD1.2 - Learn about System-Verilog syntaxis:** To read the project's source code is essential to understand how System-Verilog works and its syntaxis. This task will require 20 hours.
- **PD1.3 - Research in bootrom implementations:** Search information about how bootroms are implemented in different projects. This task will require the completion of task PM4 and will take 15 hours.

1.8.2.2 PD2 - Choose and analyze a bootrom

Choose a memory device that adapts to the project requirements. This task requires 10 hours and the completion of tasks PD1.1 and PD1.3.

1.8.2.3 PD3 - Implement the bootrom controller

- **PD3.1 - defining the controller:** Define the module to perform a read operation to the botroom and its interface to communicate with the core. This task will require 40 hours and the completion of task PD2.
- **PD3.2 - Implement the RTL module:** Implement the RTL module defined in the previous task (PD3.1). This task will require 30 hours and a computer with a code editor.

1.8.2.4 PD4 - Test and verify the controller

- **PD4.1 - Implement a standalone testbench:** Implement a testbench to verify the controller behavior. This task will take 5 hours and requires the completion of task PD3.2 and a code editor.
- **PD4.2 - Test the controller with an RTL simulation:** Perform an RTL simulation that stimulates the controller and assures the correct behavior. This task requires the completion of the previous task (PD4.1) and some RTL simulation software. It can take a maximum of 25 hours.

1.8.2.5 PD5 - Integrate the controller into the SoC

- **PD5.1 - Place and connect:** Place and connect the bootrom controller with the core. This task requires the completion of task PD4.2 and a code editor. The task will take 30 hours.

- **PD5.2 - Test the integration:** Test the correct behavior of the SoC using some tests and benchmarks in an RTL simulation. This requires the completion of the previous task (PD5.1) and an RTL simulation software. The task will take 40 hours.
- **PD5.3 - Verify the integration with GLS:** Perform a Gate-Level Simulation (GLS) on the SoC with the bootrom integration. This task requires the completion of the previous task (PD5.2) and some GLS software. The task will take 50 hours.

1.8.2.6 PD6 - Develop a bootloader

- **PD6.1 - Write the bootrom code:** Write a bootloader considering the current architecture support. This task will take 20 hours and requires the completion of PD5.3 and a code editor.
- **PD6.2 - Test the bootloader:** Perform an RTL of the SoC with the bootloader. This task requires the execution of the previous task (PD6.1) and some RTL simulation software. The task will take 40 hours.

1.8.2.7 PD7 - Test on FPGA

- **PD7.1 - Order the bootrom device:** Order the bootrom device devkit and solder a PMOD interface to connect to the FPGA. This task requires the completion of task PD2. The task will take 5 hours.
- **PD7.2 - Synthesize the whole SoC for the FPGA:** Configure and monitor the bitstream generation process for the FPGA. This task requires the completion of task PD6.2 and the Xilinx Vivado software. The task will take 10 hours.
- **PD7.3 - the SoC on the FPGA:** Verify the correct behavior of the SoC on the FPGA. This task requires the completion of the two previous tasks (PD7.1 and PD7.2), an FPGA device, and the Xilinx Vivado software. The task will take 40 hours.

1.8.3 Project Conclusion

Next, we detail the tasks after the technical development of the project until the presentation of the thesis.

- **PC1 - Write Documentation:** Document the bootrom controller implementation and integration and its tests. This task requires the completion of task PD7.2 and a text editor. The task will take 50 hours.
- **PC2 - Confection of support material:** Confection of support material for the thesis or oral defense. This task requires the task PC1 to be started. The task will take 15 hours.
- **PC3 - Prepare the oral defense:** Prepare the oral defense of the project. This task requires the execution of the previous task (PC2). The task will take 30 hours.

ID	Task	Duration	Predecessors	Ressources
PM1	Project scope and context	20		PC, ed
PM2	Project planning	20		PC, ed
PM3	Budget and sustainability	20		PC, ed
PM4	Integration into a final document	15	PM1,PM2,PM3	PC, ed
PM5	Weekly meetings	15		PC, ed
PD1.1	Familiarize PreDRAC architecture	20		PC
PD1.2	Learn about System-Verilog syntax	20		PC
PD1.3	Research in bootrom implementations	15	PM4	PC
PD2	Choose and analyze a bootrom	10	PD1.1,PD1.2	PC
PD3.1	Defining the controller	40	PD2	PC, Ced
PD3.2	Implement the RTL module	30	PD3.1	PC
PD4.1	Implement a standalone testbench	5	PD3.2	PC, Ced
PD4.2	Test the controller with an RTL simulation	25	PD4.1	PC, RTL
PD5.1	Place and connect	30	PD4.2	PC, Ced
PD5.2	Test the integration	40	PD5.1	PC, RTL
PD5.3	Verify the integration with GLS	50	PD5.2	PC, GLS
PD6.1	Write the bootrom code	20	PD5.3	PC, Ced
PD6.2	Test the bootloader	40	PD6.1	PC, RTL
PD7.1	Order the bootrom device	5	PD2	PC
PD7.2	Synthesize the whole SoC for the FPGA	10	PD6.2	PC, Xil
PD7.3	Test the SoC on the FPGA	40	PD7.1,PD7.2	PC, Xil, FPGA
PC1	Write Documentation	50	PD7.2	PC, ed
PC2	Confection of support material	15	PC1	PC
PC3	Prepare the oral defense	30	PC2	PC, ed

Table 1.1: Estimate table

1.9 Estimates and Gantt

In this section, we have an overview of the estimation time in Table 1.1, and the Gantt diagram in Figure 1.4

Resources legend:

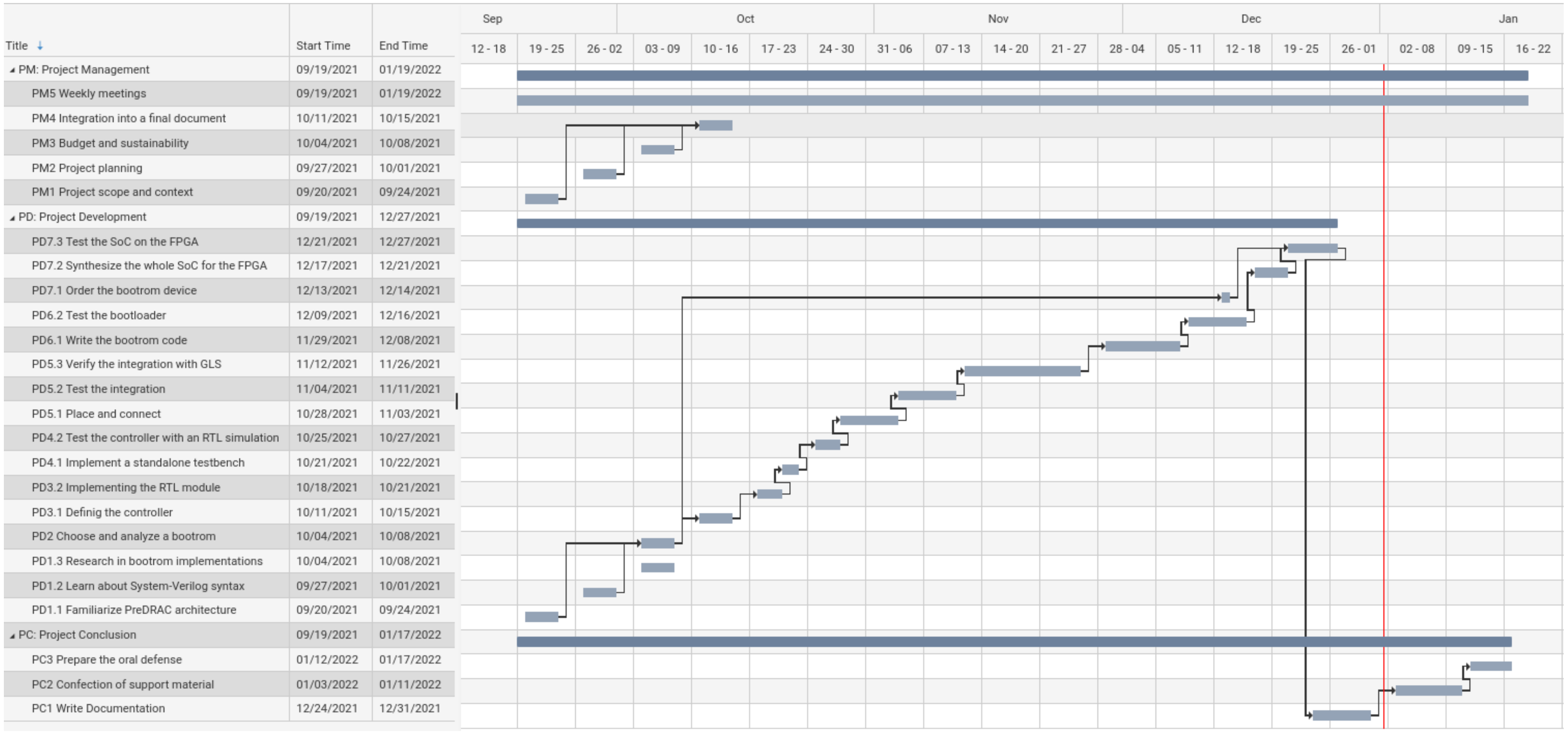
PC: Computer

ed: Text editor software

Ced: Code editor software

RTL: RTL simulation software

GLS: Gate-Level simulation software



Title	Start Time	End Time
PM: Project Management	09/19/2021	01/19/2022
PM5 Weekly meetings	09/19/2021	01/19/2022
PM4 Integration into a final document	10/11/2021	10/15/2021
PM3 Budget and sustainability	10/04/2021	10/08/2021
PM2 Project planning	09/27/2021	10/01/2021
PM1 Project scope and context	09/20/2021	09/24/2021
PD: Project Development	09/19/2021	12/27/2021
PD7.3 Test the SoC on the FPGA	12/21/2021	12/27/2021
PD7.2 Synthesize the whole SoC for the FPGA	12/17/2021	12/21/2021
PD7.1 Order the bootrom device	12/13/2021	12/14/2021
PD6.2 Test the bootloader	12/09/2021	12/16/2021
PD6.1 Write the bootrom code	11/29/2021	12/08/2021
PD5.3 Verify the integration with GLS	11/12/2021	11/26/2021
PD5.2 Test the integration	11/04/2021	11/11/2021
PD5.1 Place and connect	10/28/2021	11/03/2021
PD4.2 Test the controller with an RTL simulation	10/25/2021	10/27/2021
PD4.1 Implement a standalone testbench	10/21/2021	10/22/2021
PD3.2 Implementing the RTL module	10/18/2021	10/21/2021
PD3.1 Definig the controller	10/11/2021	10/15/2021
PD2 Choose and analyze a bootrom	10/04/2021	10/08/2021
PD1.3 Research in bootrom implementations	10/04/2021	10/08/2021
PD1.2 Learn about System-Verilog syntax	09/27/2021	10/01/2021
PD1.1 Familiarize PreDRAC architecture	09/20/2021	09/24/2021
PC: Project Conclusion	09/19/2021	01/17/2022
PC3 Prepare the oral defense	01/12/2022	01/17/2022
PC2 Confection of support material	01/03/2022	01/11/2022
PC1 Write Documentation	12/24/2021	12/31/2021

Figure 1.4: Gantt diagram

1.10 Risk management: alternative plans and obstacles

As already mentioned in the previous delivery, during the execution of the project, we may encounter different obstacles that may impede the correct completion of the project. Some tasks can take longer than estimated with ease, for example, the bootrom integration task (PD5.2). In order to reduce the chances of this happening, it is necessary to do simple tests at the beginning and gradually increase the complexity.

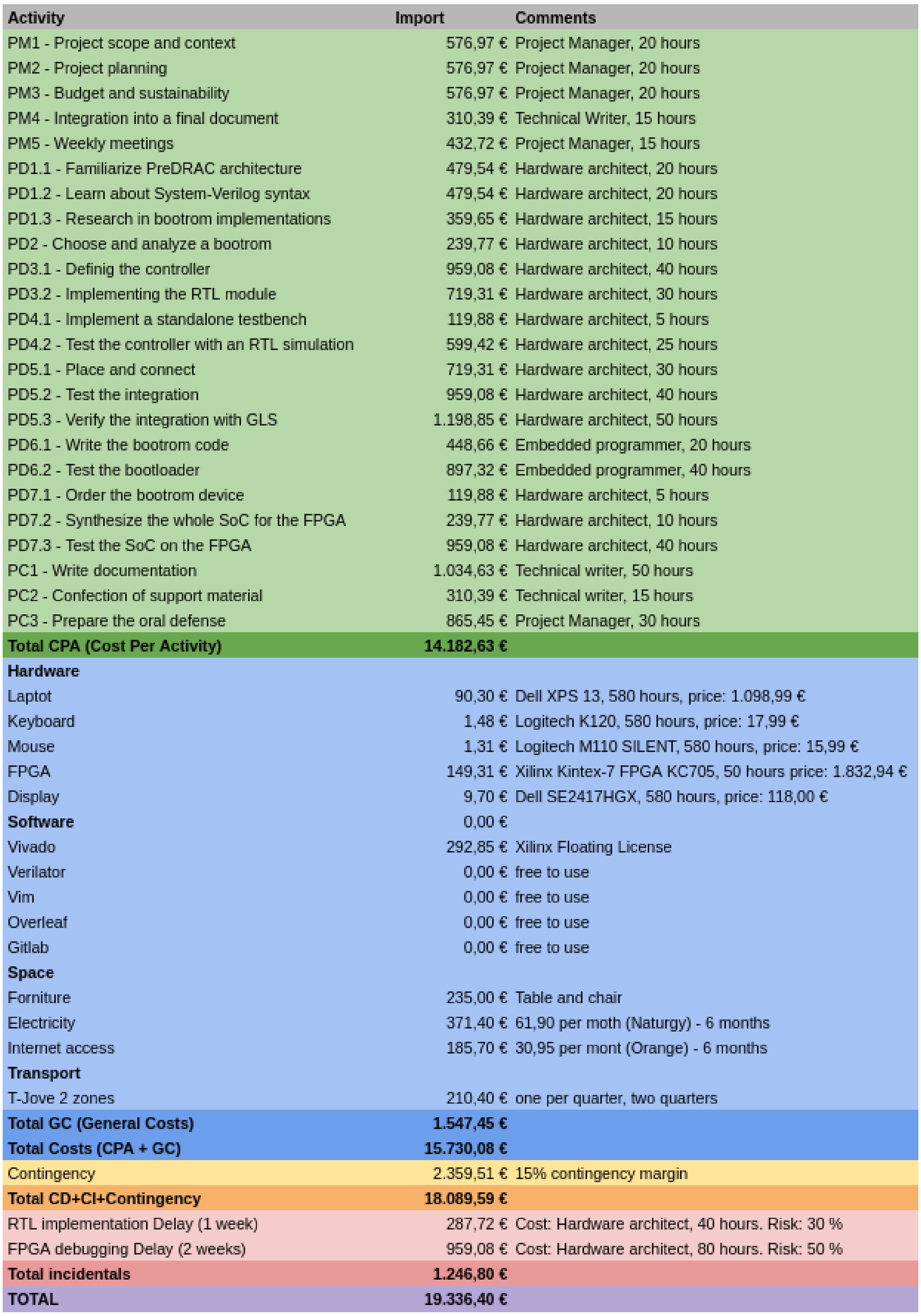
On the other hand, the task that requires the manipulation of an FPGA, can also involve delays, as they are difficult devices to debug. In order to manage this risk, and others that may arise, the project planning has been done taking into account that we should leave few weeks of margin. In addition, we plan an alternative task to add in the SoC a mechanism to boot without the bootrom, as does the original version, in case the botroom does not work correctly before the deadline.

1.11 Budget

1.11.1 Identification of costs

In this section, the costs of the project are identified and detailed. We must first determine what the resources are, their value, and their duration. First, we will refer to human resources associated with each of the project's tasks. We can differentiate the following roles:

- **Project manager:** It is the person in charge of proposing and safeguarding the correct execution of the project. In this case, the BSC tutor, the directors, and the student will play this role. The project manager role will be very relevant in all Project Management tasks (PM1-PM3, PM5, and PC3).
- **Technical writer:** It is the person responsible for writing all the technical reports and general documentation. The student will play this role, which will be present in the project tasks PM4 and PC1-PC2.
- **Hardware architect:** It is the person responsible for designing the entire computer architecture environment. The student will play this role, which essentially covers the whole development of the project, specifically in tasks PD1-PD5, PD7, and all its subtasks.
- **Embedded programmer:** It is the person responsible for developing code considering the different restrictions that may involve the hardware on which it will run. The student will play this role, which is indispensable for the PD6 task and subtasks.



Activity	Import	Comments
PM1 - Project scope and context	576,97 €	Project Manager, 20 hours
PM2 - Project planning	576,97 €	Project Manager, 20 hours
PM3 - Budget and sustainability	576,97 €	Project Manager, 20 hours
PM4 - Integration into a final document	310,39 €	Technical Writer, 15 hours
PM5 - Weekly meetings	432,72 €	Project Manager, 15 hours
PD1.1 - Familiarize PreDRAC architecture	479,54 €	Hardware architect, 20 hours
PD1.2 - Learn about System-Verilog syntax	479,54 €	Hardware architect, 20 hours
PD1.3 - Research in bootrom implementations	359,65 €	Hardware architect, 15 hours
PD2 - Choose and analyze a bootrom	239,77 €	Hardware architect, 10 hours
PD3.1 - Definig the controller	959,08 €	Hardware architect, 40 hours
PD3.2 - Implementing the RTL module	719,31 €	Hardware architect, 30 hours
PD4.1 - Implement a standalone testbench	119,88 €	Hardware architect, 5 hours
PD4.2 - Test the controller with an RTL simulation	599,42 €	Hardware architect, 25 hours
PD5.1 - Place and connect	719,31 €	Hardware architect, 30 hours
PD5.2 - Test the integration	959,08 €	Hardware architect, 40 hours
PD5.3 - Verify the integration with GLS	1.198,85 €	Hardware architect, 50 hours
PD6.1 - Write the bootrom code	448,66 €	Embedded programmer, 20 hours
PD6.2 - Test the bootloader	897,32 €	Embedded programmer, 40 hours
PD7.1 - Order the bootrom device	119,88 €	Hardware architect, 5 hours
PD7.2 - Synthesize the whole SoC for the FPGA	239,77 €	Hardware architect, 10 hours
PD7.3 - Test the SoC on the FPGA	959,08 €	Hardware architect, 40 hours
PC1 - Write documentation	1.034,63 €	Technical writer, 50 hours
PC2 - Confection of support material	310,39 €	Technical writer, 15 hours
PC3 - Prepare the oral defense	865,45 €	Project Manager, 30 hours
Total CPA (Cost Per Activity)	**14.182,63 €**	
Hardware		
Laptot	90,30 €	Dell XPS 13, 580 hours, price: 1.098,99 €
Keyboard	1,48 €	Logitech K120, 580 hours, price: 17,99 €
Mouse	1,31 €	Logitech M110 SILENT, 580 hours, price: 15,99 €
FPGA	149,31 €	Xilinx Kintex-7 FPGA KC705, 50 hours price: 1.832,94 €
Display	9,70 €	Dell SE2417HGX, 580 hours, price: 118,00 €
Software	0,00 €	
Vivado	292,85 €	Xilinx Floating License
Verilator	0,00 €	free to use
Vim	0,00 €	free to use
Overleaf	0,00 €	free to use
Gitlab	0,00 €	free to use
Space		
Forniture	235,00 €	Table and chair
Electricity	371,40 €	61,90 per moth (Naturgy) - 6 months
Internet access	185,70 €	30,95 per mont (Orange) - 6 months
Transport		
T-Jove 2 zones	210,40 €	one per quarter, two quarters
Total GC (General Costs)	**1.547,45 €**	
Total Costs (CPA + GC)	**15.730,08 €**	
Contingency	2.359,51 €	15% contingency margin
Total CD+CI+Contingency	**18.089,59 €**	
RTL implementation Delay (1 week)	287,72 €	Cost: Hardware architect, 40 hours. Risk: 30 %
FPGA debugging Delay (2 weeks)	959,08 €	Cost: Hardware architect, 80 hours. Risk: 50 %
Total incidentals	**1.246,80 €**	
TOTAL	**19.336,40 €**	

Table 1.2: Budget summary

Next, it is necessary to define the costs of the material resources. That is all the necessary hardware and software for the project execution and the material to assure a suitable space for the correct development. Finally, we will include transportation costs to travel to the workplace within this category.

Table 1.2 details all the project costs, divided by Costs per Activity (CPA), General Costs (GC), Contingency, and Incident Costs.

1.11.2 Cost estimates

We have estimated a consistent annual gross salary for each of the roles defined above to compute personnel costs. The estimations are based on GlassDoor [12], a web portal that provides a substantial role's average salary. The price per hour column has been calculated assuming an agreement of about 1780 hours per year. Table 1.3 shows the estimated annual gross salary, the price per hour, the total hours, and the total cost of each role.

Role	Annual salary	Total salary including SS	Price per hour	Total hours	Total cost
Project Manager	39.500,00 €	51.350,00 €	28,85 €	105,00	3.029,07 €
Technical Writer	28.333,00 €	36.832,90 €	20,69 €	80,00	1.655,41 €
Hardware Architect	32.830,00 €	42.679,00 €	23,98 €	340,00	8.152,17 €
Embedded Programmer	30.716,00 €	39.930,80 €	22,43 €	60,00	1.345,98 €

Table 1.3: Roles salary

Table 1.2 shows the cost for each task (CPA), which is the result of multiplying the hours allocated for that task by the price per hour of the role that performs it. The total personnel cost is 14.182,63 €.

As for the material costs, in particular, those relating to software and hardware must be amortized. The formula we used to make this calculation is as follows:

*(Purchase price * Total project hours) / (1780 hours * 4 years)*

The cost of transport to the workspace and the essential elements, such as internet access or electricity, are also part of the general costs (GC). Altogether they add up to a total of 1.547,45 €.

Once the two values have been calculated, we apply a contingency margin of 15%. Thus, the sum of CPA and GC with the contingency margin is equivalent to 18,089.59 €.

Finally, we will add the costs that may result from the anticipated incidents. The two incidents contemplated consist of delays, either in the implementation task or in the testing. In both cases, it involves an expense corresponding to overtime, which the hardware architect must dedicate. The values shown in the Table 1.2 are weighted to the risk associated with each incident. The cost of the expected incidents is 1,246.80 €.

1.11.3 Management control

In order to be able to evaluate the budget planned for the project, it is necessary to have a control mechanism for it. We will define a series of indicators that will allow us to advertise potential budget deviations. The project manager must check the indicators at the end of each task. Following, we detail the indicators and how they are calculated:

- **Human resources (CPA) deviation:** When the personnel does less or more hours than expected. We compute this deviation as shown below.

*Human resources deviation = (Estimated cost per hour - Real cost per hour) * Total hours consumed*

- **Amortization deviation:** In case we use the hardware resource less or more time than expected, the amortization cost will vary. We compute this deviation as shown below.

*Amortization Deviation = (Estimated usage hours - Real usage hours) * Price per hour*

- **Electricity deviation:** The electricity supply cost could change significantly, affecting the budget. We compute this deviation as shown below.

*Electricity cost deviation = (Estimated usage hours - Real usage hours) * Price per hour*

- **GC deviation:** Given that the workplace will always be the same, no place or transport deviations are contemplated. We compute this deviation as shown. below.

GC deviation = Electricity deviation + Amortization Deviation

- **Incidental cost deviation:** There could be more incidents than expected or none. We compute this deviation as shown below.

*Incidental cost deviation = (Estimated incidental hours - Real incidental hours) * Total incidental hours*

Finally, we need to add the CPA, GC, and incidental deviations to get an idea of the general cost deviation:

General cost deviation = Human resources deviation + GC deviation + Incidental cost deviation

1.12 Sustainability report

Project sustainability is a fundamental element to consider. However, we do not know firsthand what criteria and indicators we should use to assess the project's sustainability, allowing us to modify the initial planning to improve in this aspect. For this reason, we are grateful for the survey provided to help the student to reflect on this matter. Below, we present the student's reflections on the project's sustainability divided into environmental, social and economic dimensions.

1.12.1 Environmental dimension

1.12.1.1 PPP

Although we have not explicitly quantified the project's environmental impact, we have determined the different aspects that affect the environment to reduce them. One of the most evident aspects is the student's transportation to the workplace. Although it has not always been possible, it has been preferable to use public transport. One of the actions taken to mitigate the impact of transportation to the workspace has been to enable remote connections to the material necessary to develop the project allowing the student to work from home. The material is another aspect to consider. All possible material from other projects has been reused. Taking this into account, we have reduced to the maximum, within our capabilities, the environmental impact of the project's development.

1.12.1.2 Exploitation

The project's specific objective is to implement a bootrom to dispense using an FPGA to boot the SoC. This directly affects the power consumption of the final chip since the power consumption of the FPGA is much higher than that of the bootrom that will replace it. Furthermore, in general terms, the research and development of an open-source chip that will one day match the private alternatives on the market and, thus facilitating the local production of these chips, have clear positive consequences for the environment, if only by reducing emissions to transportation.

1.12.1.3 Risks

We contemplate some situations that could affect the sustainability of the project. During the execution of the project, the material may break and need to be replaced. However, this risk is minor since most of the tools used during development are software. Therefore, their replacement, updating or searching for alternatives would not impact the project's sustainability. The most significant risk is to discover, once the chip is manufactured, that the implemented bootrom fails. Implementing a fallback boot mechanism using the FPGA again is envisaged to avoid rendering the entire chip unusable. However, we would lose the power reduction of not needing it.

1.12.2 Economic dimension

1.12.2.1 PPP

As detailed in Section 1.11, an exhaustive analysis of the budget for the project has been carried out. Human and material resources have been taken into account. We have also adapted to reuse material from other projects to reduce the budget. The total cost is 18.089,59€ without the 15% contingency margin. The only expense that has not been contemplated in the initial budget is the purchase of the development boards for the EEPROM memory used as a bootrom. We had considered this purchase because we did not know which memory we would use as a bootrom when calculating the budget. In addition, initially, we did not include testing with the physical device in the project's scope. However, the costs related to this purchase do not exceed 40€ and are, therefore, within the planned contingency margin.

1.12.2.2 Exploitation

The project operation costs are directly related to the energy consumption of the final SoC and its maintenance. If everything works correctly, we will considerably reduce the power consumption since we will eliminate an FPGA from the equation. The power consumption introduced by the bootrom controller and the bootrom itself is minimal in relation to the total power consumption of the chip and much lower than that of the FPGA. In the worst case, the FPGA will still be needed. Therefore, the final power consumption will increase significantly because the chip design will include the bootrom controller even if it is not functional.

1.12.2.3 Risks

As we have repeated before, there is a critical risk that after the chip is manufactured, the bootrom controller implemented and integrated during the realization of the project will not work. As the bootrom is a crucial element to use the chip, and the manufacturing of the chip is costly, it has been decided to implement an alternative option in case of failure. In this way, we mitigate the cost due to the possible error having to assume only the increase of power consumption due to the use of the FPGA.

1.12.3 Social dimension

1.12.3.1 PPP

The realization of this project has allowed the student to learn about the world of academic research. In addition, the project covers different disciplines: from computer architecture to the development of code for embedded systems or the different stages involved in the manufacture of a processor. The student has been able to expand the knowledge acquired during his bachelor by discovering an exciting sector within the field of computer engineering.

1.12.3.2 Exploitation

The research and development of open-source chips have a very positive impact on society. Opening the designs of these components that have been part of people's daily lives for many years offers security and the guarantee that the product that the user consumes is at the end what it claims to be. The implementation and integration of the bootrom in the DRAC 22 nm SoC is another step in this direction.

1.12.3.3 Risks

If the bootrom controller does not work, it would directly affect the end-user, complicating the use of the chip since it would require an FPGA with the extra costs that it implies at an economic and technological level.

2 Background

This chapter aims to provide the reader with the appropriate background to understand the project better. First, we will discuss some of the open-source Hardware projects related to the one detailed on this thesis. Then, we will review the RISC-V privilege architecture and a typical boot process.

2.1 Open-source Hardware

Below we list a few open-source projects that have contributed to the development of Drac 22 nm SoC in one way or another. Some of these projects are widely known and used by the entire RISCV-based hardware development community. These are just a few examples since the community continuously grows and publishes new projects.

- **Rocket:** Rocket [3] is a 5-stage in-order scalar processor core generator, originally developed at UC Berkeley and currently supported by SiFive. The Rocket core supports the open-source RV64GC RISC-V instruction set and is written in the Chisel hardware construction language. It has an MMU that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction. Rocket also supports the RISC-V machine, supervisor, and user privilege levels. A number of parameters are exposed, including the optional support of some ISA extensions (M, A, F, D), the number of floating-point pipeline stages, and the cache and TLB sizes. Rocket is one of the most widely used cores recently and is also usually used as a component library for more complex processors such as the BOOM core [9].
- **LowRISC:** LowRISC chip is a Rocket-based 64-bit SoC design developed by lowRISC [20]. They offer an FPGA-ready SoC distribution, with open-source peripherals such as SD and Ethernet.
- **Ariane:** Ariane [41] is a 64-bit RISC-V in-order 6-stage core from ETH Zurich. It is one of the world's best implementations of a RISC-V core. There are two tape-outs based on the Ariane design: Kosmodron [42] and Poseidon [21].
- **Lagarto:** Lagarto [1] is a single-issue in-order 5-stage core developed by CIC-IPN, BSC, IMB-CNM (CSIC), and UPC, which implements the 64bit RISC-V ISA with the M and A extensions. It also implements the privilege ISA 1.7 with the machine, supervisor, and user modes.

- **Spike:** Spike [26] is a RISCV-V ISA simulator that suports RV32I and RV64I base ISAs along with MAFDQCVP and hypervisor v1.0 extensions, machine, supervisor ans user modes v.1.11 and much more.
- **Verilator:** Verilator [34] is a tool that converts Verilog to a cycle-accurate behavioral model in C++ or SystemC. It is restricted to modeling the synthesizable subset of Verilog, and the generated models are cycle-accurate, 2-state, with synthesis (zero delays) semantics. Consequently, the models typically offer higher performance than the more widely used event-driven simulators, which can process the entire Verilog language and model behavior within the clock cycle.
- **OpenSBI:** The goal of the OpenSBI project [25] is to provide an open-source reference implementation of the RISC-V SBI specifications for platform-specific firmwares executing in M-mode. An OpenSBI implementation can be easily extended by RISC-V platform and system-on-chip vendors to fit a particular hardware configuration.
- **OpenPiton:** OpenPiton [6, 18] is the world's first open-source, general purpose, multithreaded manycore processor. It is a tiled manycore framework scalable from one to 1/2 billion cores. It is a 64-bit architecture using SPARC v9 ISA [31] with a distributed directory-based cache coherence protocol across on-chip networks. It is highly configurable in both core and uncore components.
- **JuxtaPiton:** JuxtaPiton [19] is the first open-source, general-purpose, heterogeneous-ISA processor. It is the first time a new RISC-V core has been integrated with the OpenPiton framework

2.2 RISC-V privilege levels

This section will introduce some basic components and concepts of the RISC-V privileged architecture. Refer to the RISC-V privileged architecture specification for more information [35]. We will first review privileged software stacks. Then, we will detail the different privilege levels of a RISC-V-based system.

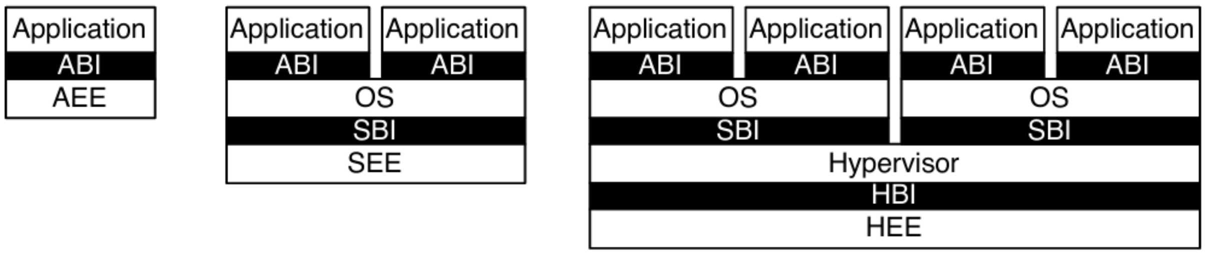


Figure 2.1: Different implementation stacks supporting various forms of privileged execution. Source: [35]

Figure 2.1 shows three possible software stacks. We will focus only on the middle one since it is applied in the PreDRAC chip boot process. This is the typical configuration with an Operating

System (OS) capable of running multiple applications. These applications communicate with the OS through the Application Binary Interface (ABI), providing an Application Execution Environment (AEE). The ABI includes the supported user-level ISA, and a set of ABI calls to interact with the AEE. The ABI hides the details of the AEE from the application to allow greater flexibility in the implementation of the AEE.

Similarly, the OS communicates with an Supervisor Execution Environment (SEE) through an Supervisor Binary Interface (SBI). An SBI comprises the user-level and supervisor-level ISA along with a set of SBI function calls. The use of a single SBI in all SEE implementations allows a single binary image of the OS to run on any SEE. The SEE can be a simple boot loader and BIOS-like I/O system on a low-end hardware platform, or a virtual machine provided by a hypervisor on a high-end server, or a thin translation layer on top of a host OS in an architectural simulation environment.

It is important to note that at any time, a RISC-V hardware thread (a core) runs at some privilege level encoded as a mode in a Control and Status Register (CSR). Table 2.1 shows the privilege levels defined by the ISA.

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

Table 2.1: RISC-V privilege levels. Source: [35]

Privilege levels are used to provide protection between different software stack components, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and OS usage, respectively.

2.3 Boot process

All the steps that a hardware platform has to perform on a power-on to provide an environment ready to run applications are part of the boot process. We will first briefly define this process and the elements involved in it. Then we will talk about the specific boot process on the PreDRAC SoC. Figure 2.2 shows the main software components to be considered during the boot process.

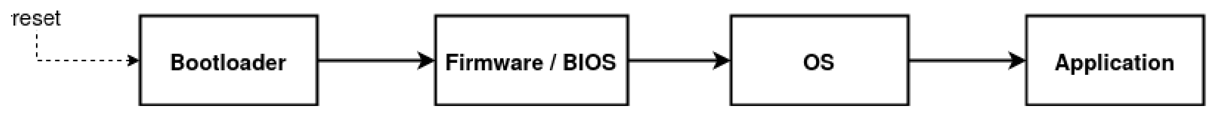


Figure 2.2: Boot-realated software components

The bootloader is the first code to be executed on the system. Any designed chip will require several hardware components to support the processor, such as memory devices or peripheral controllers like the keyboard and display. The bootloader is usually stored in a ROM memory and runs in machine mode. Its primary purpose is to initialize the hardware components of the chip to prepare the environment needed to load the firmware (or BIOS) into the processor's memory hierarchy and run it.

The firmware is usually located in some on-chip storage device, such as flash or DDR memory. It is responsible for configuring and preparing all the chip components and loading the OS.

The OS typically runs in supervisor mode. It prepares the environment for the execution of applications and is responsible for programming them. Before switching to the context of an application, it drops privileges by entering user mode.

2.3.1 PreDRAC bootloader

The PreDRAC chip bootloader is very straightforward. The Figure 2.3 shows a sequence diagram of the most relevant actions performed by the bootloader. As mentioned before, the bootrom that hosts the bootloader is a perfect memory implementation for an FPGA communicated directly with the SoC.

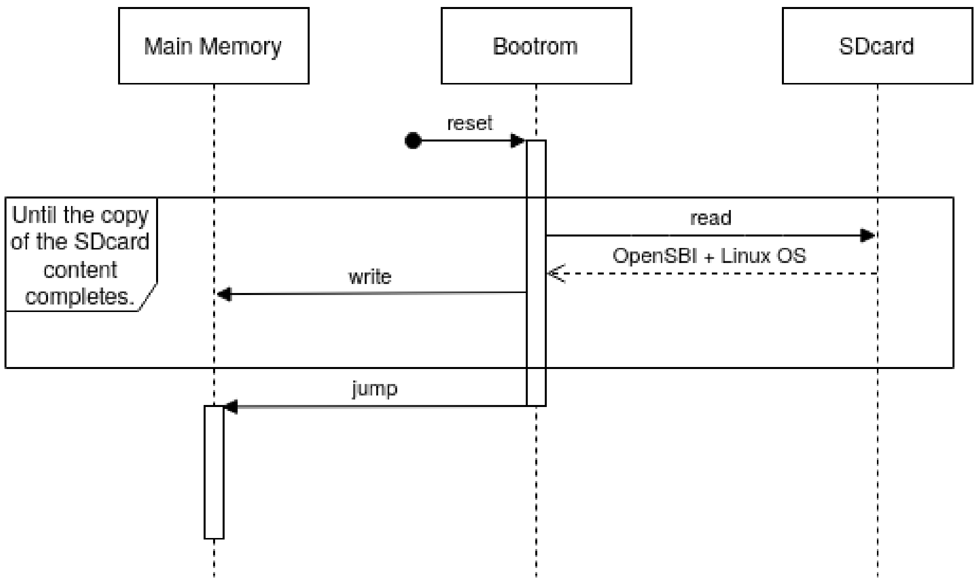


Figure 2.3: PreDrac boot sequence diagram

In the diagram, we can recognize five distinct steps of the bootloader:

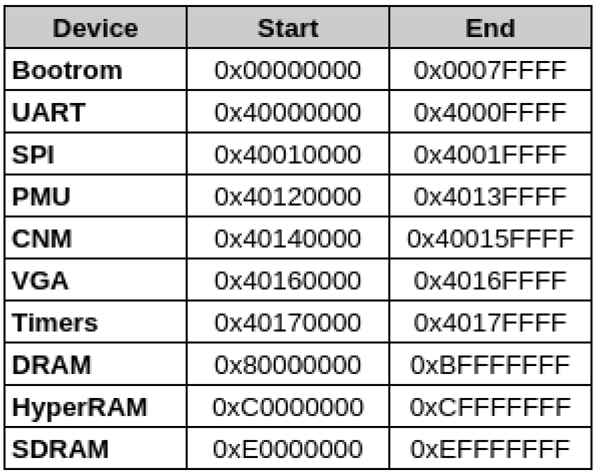
1. Initializes the UART, thus allowing communication with the outside of the chip. The bootloader implements a modified UART version of the classic *printf()* function. In this way, we can comfortably obtain information for the rest of the bootloader execution.

2. Mounts the SD card. The SD card contains a single binary with OpenSBI and the Linux Kernel together. The OpenSBI is firmware that offers an abstraction of the platform-specific functionalities.

3. Parses and copies the binary from the SD card to the DRAM, the chip's main memory.

4. Sets the mapping of the different memory and I/O devices. Table 2.2 shows the address ranges that correspond to each of the devices.

5. Finally, it jumps to the first address of the DRAM giving the execution to the OpenSBI that will later execute the Linux kernel.



Device	Start	End
Bootrom	0x00000000	0x0007FFFF
UART	0x40000000	0x4000FFFF
SPI	0x40010000	0x4001FFFF
PMU	0x40120000	0x4013FFFF
CNM	0x40140000	0x40015FFFF
VGA	0x40160000	0x4016FFFF
Timers	0x40170000	0x4017FFFF
DRAM	0x80000000	0xBFFFFFFF
HyperRAM	0xC0000000	0xCFFFFFFF
SDRAM	0xE0000000	0xEFFFFFFF

Table 2.2: Device physical addresses mapping

3 Design and implementation

This chapter covers the design and implementation of the bootrom controller. The final goal is to create a SystemVerilog module, the language used in most of the project, so that it can be instantiated in the SoC and, in this way, allow the processor to fetch and execute the instructions the bootloader. However, for now, we will leave the SoC aside and focus on implementing a module capable of processing read and write requests.

3.1 First steps

Since every device may differ in the way we should interface with it, the first step to designing a bootrom controller is to choose which device we will use as a bootrom. Then, we will talk about how we can interact with it; we will define its pinout and some of the instructions it supports.

3.1.1 Choose of the bootrom

When choosing which memory device to use, three key factors are, generally, the interface, technology, and size. To evaluate the different options, we must take into account what are the limitations we start with. In this case, the restriction is clear: we don't have too many free pins in the SoC pinout, and, therefore, we need to use as few pins as possible.

This requirement directly affects the type of interface the device has to have, thus discarding any parallel protocol due to the number of pins it would require. However, within the most common serial protocols, we have different options. The table 3.1 shows the number of pins needed to implement each of the serial protocols we consider.

Protocol	**Sync/Async**	**# Pins**	**Duplex**	**Max speed (Kbps)**
I2C	sync	2	half	3.400
SPI	sync	4	full	>1.000
Microwire	sync	4	full	>625
1-Wire	async	1	half	16

Table 3.1: Serial protocols comparision [Self compilation]

Although it may seem that 1-Wire is the protocol best suited to the requirement of using the minimum number of pins, this is not entirely true. The DRAC SoC already implements the SPI protocol to communicate with the SD card. Figure 3.1 shows a typical SPI connection with three independent slaves sharing the data pins and the clock (MISO, MOSI, and SCLK). Therefore, adding one more slave device, such as the bootrom in our case, would only cost a single pin (SS), which is responsible for activating the new device when needed. In this way, both SPI and 1-Wire only involve a 1-pin overhead, and in consequence we will choose the SPI protocol for its simplicity and higher transmission speed.

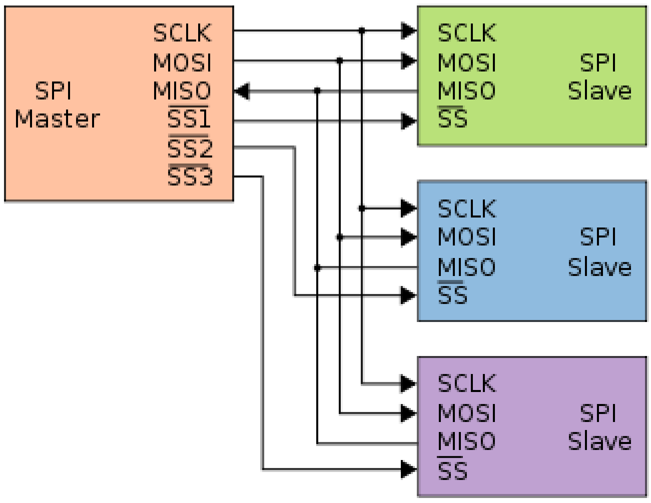


Figure 3.1: Typical SPI bus: master and three independent slaves [40]

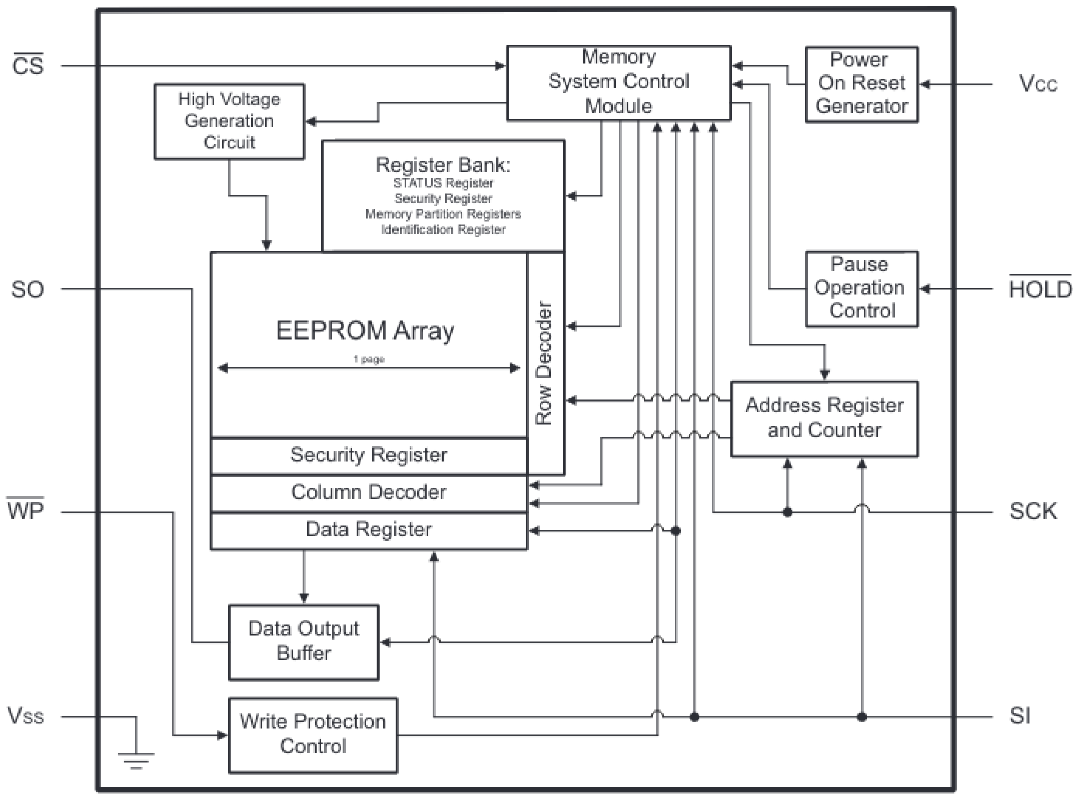
The technology of the memory device is not a critical factor in the development of the chip. The two most commonly used options are Electrically Erasable Programmable Read-Only Memory (EEPROM) and Flash memories. Both can be erased and reprogrammed. The main difference between them is that Flash memories allow block erases, while EEPROM memories erase byte by byte. This has an advantage in the erase speed of Flash over EEPROM, but, as it does not affect the specific use case of the bootrom, it is a negligible advantage. Another aspect to consider is that Flash memories have a data retention of approximately 40 years while the data retention of EEPROMs is over 100 years.

Finally, memory size is the last factor to consider. Since the bootrom is, in the end, a peripheral of the chip, we have no area or size limitation other than that the bootloader must fit in the memory. Therefore, we have considered that with memories of approximately 512 KB, we will have enough space.

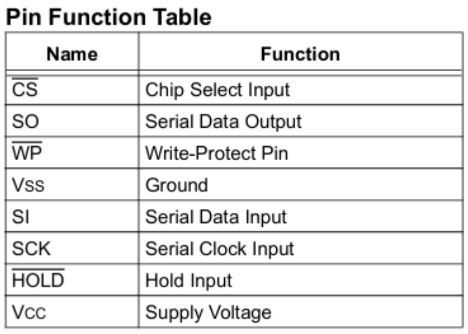
With all this in mind, the memory we will use as bootrom is the 4-Mbit SPI serial EEPROM 25CSM04 [14], which fits the project's technical needs and has a free SystemVerilog model that will allow us to test the implementation of the controller in simulation.

3.1.2 Understanding the 25CSM04 pinout

As shown in the Figure 3.2, the device has six pins, excluding ground and power supply pins. The ones involved in the SPI interface, as mentioned above, are only four, and we will talk about them later: SI, SO, SCLK and CS.



(a) 25CSM04 block diagram


Pin Function Table

Name	Function
$\overline{CS}$	Chip Select Input
SO	Serial Data Output
$\overline{WP}$	Write-Protect Pin
Vss	Ground
SI	Serial Data Input
SCK	Serial Clock Input
$\overline{HOLD}$	Hold Input
Vcc	Supply Voltage

(b) 25CSM04 pin function table

Figure 3.2: 25CSM04 EEPROM pin details. Source: [14]

The HOLD and WP pins serve to control two functionalities the device provides. The first, the hold function, allows us to pause an operation without having to stop or restart the serial clock sequence. This pause does not affect the write cycle, which means that if we trigger this function during a write operation, it will continue till the write completes.

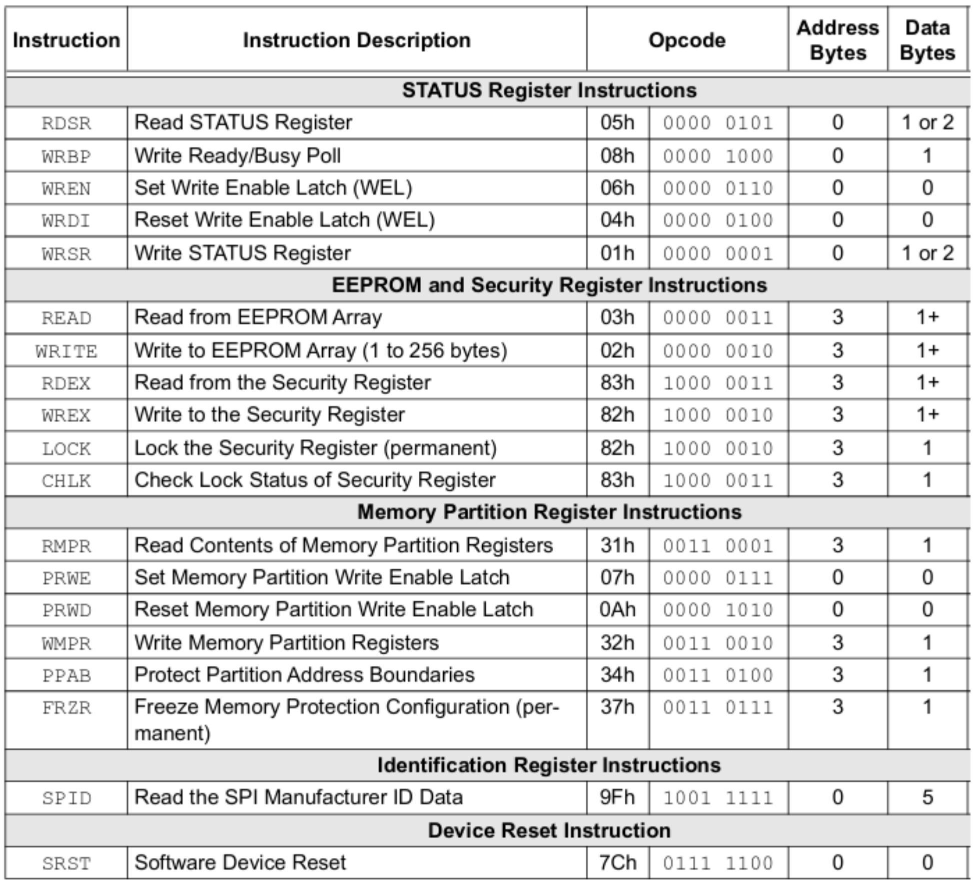
The WP pin activates the write-protect function. This function allows us to divide the memory into nine partitions and define them as read-only, blocking writes at hardware or software level as desired. We will not go into more detail about these two features since we will not use them. As we only want to be able to read instructions stored in the device, we can hardwire these pins high to deassert them.

Next, we will detail the pins related to SPI communication. The SO is the slave data output pin, so the memory contents at the requested address will be shifted through this pin. The SI is the slave input pin, and we will use it to specify the operation (see Table 3.2), addresses, and data when needed. The CS pin selects the device, which is required to be able to interact with it; otherwise, the slave will remain in Stand-By mode. In the Stand-By mode, the device ignores the SI pin and sets the SO pin in a high-impedance state.

Finally, the SCLK is the serial clock, this is an input of the device, so the controller must generate it. The maximum clock frequency is 8MHz if we power the device from 3.0V to 5.5V, or 5MHz if we power it from 2.5V to <3.0V. We must also consider that data on the SO pin is always clocked out on the falling edge of SCK, and the data on the SI pin is always latched on the rising edge.

3.1.3 25CSM04 basic operations

Once we select the device, driving the CS pin low, the first byte sent must be the opcode of the instruction we want the device to perform. The Table 3.2 shows the 19 instructions implemented by the 25CSM04 along with their 8-bit serial opcode. All instructions, addresses, or data must be transmitted starting with the most significant bit.



Instruction	Instruction Description	Opcode		Address Bytes	Data Bytes
STATUS Register Instructions					
RDSR	Read STATUS Register	05h	0000 0101	0	1 or 2
WRBP	Write Ready/Busy Poll	08h	0000 1000	0	1
WREN	Set Write Enable Latch (WEL)	06h	0000 0110	0	0
WRDI	Reset Write Enable Latch (WEL)	04h	0000 0100	0	0
WRSR	Write STATUS Register	01h	0000 0001	0	1 or 2
EEPROM and Security Register Instructions					
READ	Read from EEPROM Array	03h	0000 0011	3	1+
WRITE	Write to EEPROM Array (1 to 256 bytes)	02h	0000 0010	3	1+
RDEX	Read from the Security Register	83h	1000 0011	3	1+
WREX	Write to the Security Register	82h	1000 0010	3	1+
LOCK	Lock the Security Register (permanent)	82h	1000 0010	3	1
CHLK	Check Lock Status of Security Register	83h	1000 0011	3	1
Memory Partition Register Instructions					
RMPR	Read Contents of Memory Partition Registers	31h	0011 0001	3	1
PRWE	Set Memory Partition Write Enable Latch	07h	0000 0111	0	0
PRWD	Reset Memory Partition Write Enable Latch	0Ah	0000 1010	0	0
WMPR	Write Memory Partition Registers	32h	0011 0010	3	1
PPAB	Protect Partition Address Boundaries	34h	0011 0100	3	1
FRZR	Freeze Memory Protection Configuration (permanent)	37h	0011 0111	3	1
Identification Register Instructions					
SPID	Read the SPI Manufacturer ID Data	9Fh	1001 1111	0	5
Device Reset Instruction					
SRST	Software Device Reset	7Ch	0111 1100	0	0

Table 3.2: 25CSM04 EEPROM instructions with their operation opcode. Source: [14]

We will not discuss the sequence of all the instructions but only the ones we will use. We will start with the simplest one, the WREN (Write Enable) instruction. If the device is not write-enabled, it will ignore the write instructions. As seen in the Figure 3.3, the sequence is straightforward and only transfers the opcode.

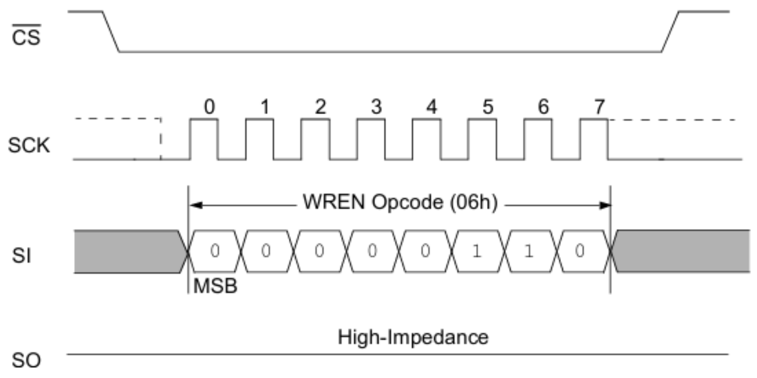


Figure 3.3: 25CSM04 write enable (WREN) instruction sequence. Source: [14]

The following is the WRITE instruction, which will allow us to write the contents of the memory. The writing sequence is a bit more complex. Once we transfer the opcode, the direction of the write is specified. The addresses are 3 bytes long, but the device will ignore the 5 most significant bits since only 19 are needed to address 512KB of memory. Finally, as shown in the Figure 3.4, we must transfer the data. The internal write cycle time is five milliseconds.

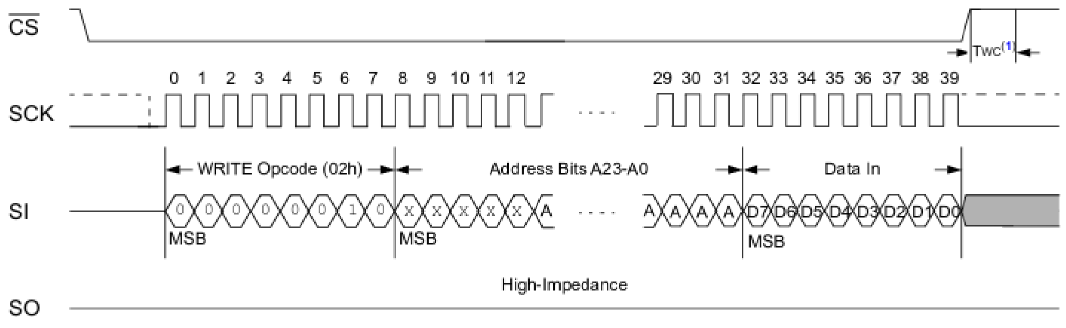


Figure 3.4: 25CSM04 byte write (WRITE) instruction sequence. Source: [14]

When a write finishes, we must consider that the device will disable the writes again. Therefore, it will be necessary to re-enable them using the WREN instruction before performing a new write.

The 25CSM04 also allows writes of up to 256 bytes, so it is unnecessary to enable the writes, specify the opcode and address for each byte to write. However, the writes must be of contiguous addresses and must belong to the same page of the memory array (where the bits from 18 through 8 of the address are the same). After writing each byte, the device increments only the lowest 8 bits of the address, keeping the rest of the address constant. In the case of providing more data than will fit in the page, the address counter will return to the beginning of the same page. The Figure 3.5 shows a page write sequence.

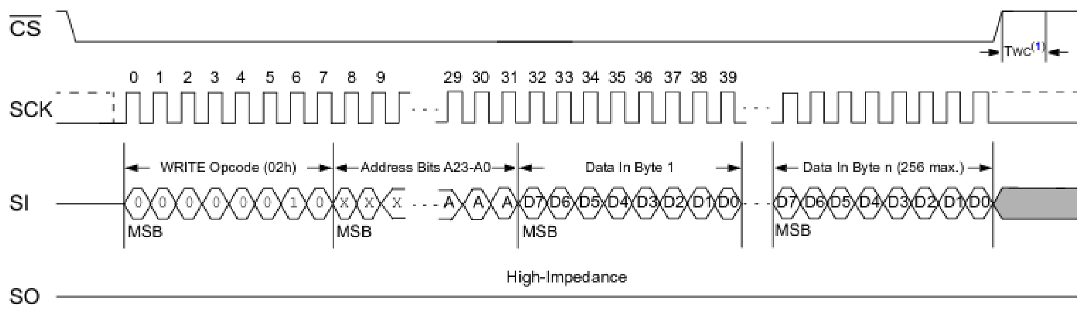


Figure 3.5: 25CSM04 page write (WRITE) instruction sequence. Source: [14]

The last we will discuss is the READ instruction. It is the most crucial instruction since it is the one that will allow us to read the EEPROM contents. The read sequence, shown in the Figure 3.6, is very similar to the write instruction sequence. In this case, however, once we transfer the address, the device will ignore the SI pin and shift out the contents of the specified address onto the SO pin. The address is automatically incremented, and the sequence continues, shifting the data onto the SO pin. To terminate the sequence, we have to drive de CS line high.

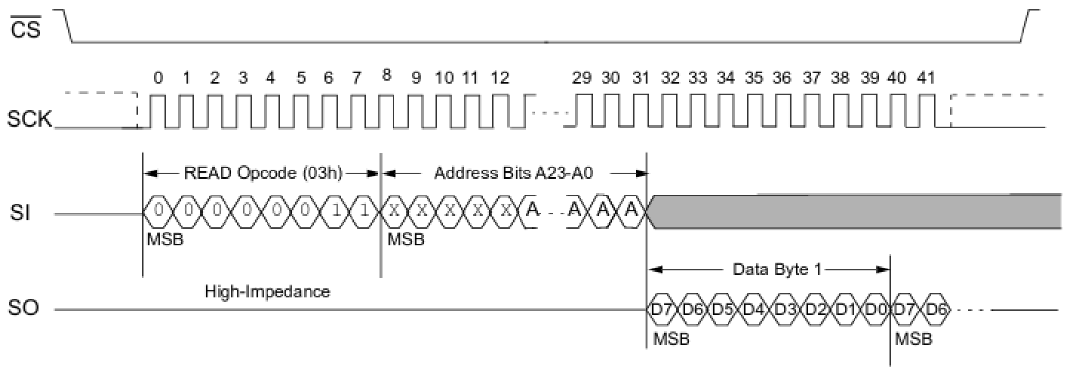


Figure 3.6: 25CSM04 read (READ) instruction sequence. Source: [14]

3.2 Bootrom controller implementation

Next, we will explain the implementation of the bootrom controller. To do this, we will first talk about the controller's interface and, then, some implementation details, such as the handshake or state machine.

3.2.1 25CSM04 controller design

Once we know how to interact with the memory, we can design and implement the controller. To do this, we will start from the outside in, defining the inputs and outputs of the component. Then, we will detail some relevant aspects of the implementation.

3.2.1.1 Controller interface details

For simplicity, we will separate the controller module interface into two groups. First, the Controller-Slave interface is the one that is established directly with the EEPROM, and then the Master-Controller interface, which in our case is the one that communicates with the SoC. Note that referring to the signals' names, we will use the suffix '*_i*' for the module inputs and '*_o*' for the outputs for better readability.

The Controller-Slave interface will consist of only four signals, all of them of one bit.

- **mo_o:** The master output, which we will use to serially transfer opcodes, addresses, and data in case of writes.
- **mi_i:** The master input, where we will receive the data after a read operation.
- **sclk_o:** The serial clock that the controller must provide for synchronization. The 25CSM04 supports SPI modes 0 and 3. These define the polarity and phase of the serial clock. We will implement mode 0, which states that the clock must be low when the device is not asserted.
- **sclk_en_o:** We will use this bit as chip select to assert or deassert the EEPROM.

The Master-Controller interface is a bit more complex. The module will be parameterized to be able to adapt to different needs easily without wasting resources. We will first detail the interface and then discuss how to use the parameters to modify it when instantiating the module.

- **clk_i:** Reference clock from which the serial clock will be generated.
- **rstn_i:** Active low asynchronous reset signal.
- **req_opcode_i:** 8-bit bus for the opcode of the operation that the controller will transfer to the serial device.
- **req_address_i:** 24-bit bus for the write or read address.
- **req_data_i:** Bus, of variable size, for the data in case of a write operation.
- **req_bytes_i:** Variable size bus to specify the name of the bytes to be written or read.
- **req_valid_i:** Signal to inform that the input fields are valid. This signal is synchronous to the reference clock. As long as the controller is not busy, for each rising clock edge where this signal is up, the controller will start the communication process with the data present at that moment.
- **ready_o:** Signal to indicate if the controller is free or not. When the controller is busy, it will ignore the *req_valid_i* signal.
- **resp_data_o:** 8-bits signal for the data in case of a read operation.

- **resp_valid_o:** Signal, synchronized with the reference clock, that indicates when the read data is valid. This signal will be driven high as many times as the number of bytes requested in the read operation.

In summary, Figure 3.7 shows a diagram with the controller interface and the connections to the 25CSM04 EEPROM.

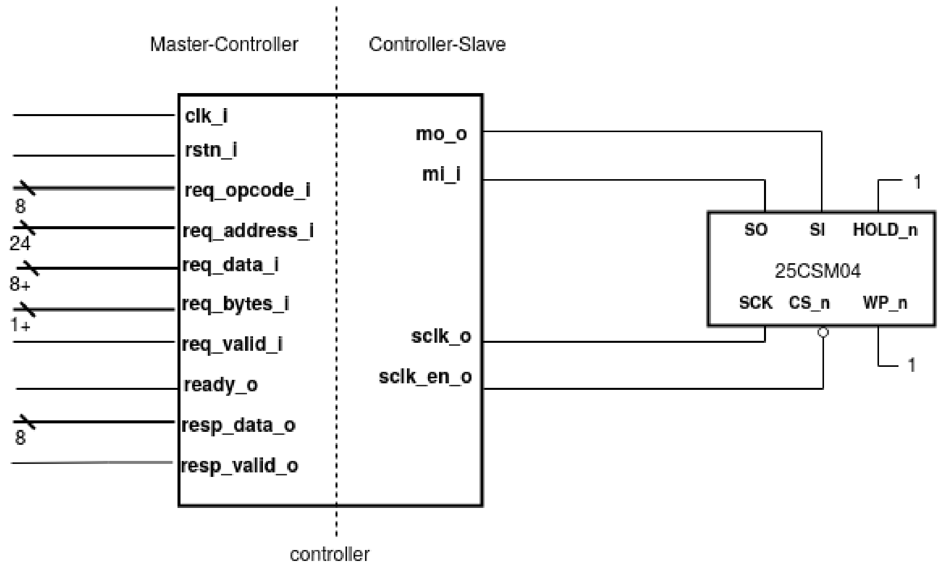


Figure 3.7: Controller interface diagram

As mentioned above, we are interested in doing a parameterized implementation. Keeping the interface and implementation simple, we want to take advantage that the memory supports page writes, which will be helpful when writing the bootloader. However, this is a rare situation since no writes will be performed for the most general use case; reading instructions from memory. That is why, to avoid wasting resources, we will use the *MAX_REQ_BYTES* parameter. This parameter indicates the size in bytes of the *req_data_i* bus and determines the size of the *req_bytes_i* bus that matches the number of bits needed to represent that value. We will discuss this and other parameters in more detail below.

3.2.1.2 Controller implementation details

Although you can find the entire implementation in the Appendix A.1, it is appropriate to point out some implementation details. First, however, we will see how we have defined the port map of the controller module. The Listing 3.1 shows the input and output signals mentioned above and two parameters.

```
module spi_eeprom_req
  #(
    parameter CLK_DIV_FACTOR = 32,
    parameter MAX_REQ_BYTES = 4)
  (
    // Master-Controller interface
    input  logic                             clk_i,
  input  logic                           rstn_i,
  input  logic [7:0]                     req_opcode_i,
  input  logic [23:0]                    req_address_i,
  input  logic [(MAX_REQ_BYTES*8)-1:0]   req_data_i,
  input  logic [$clog2(MAX_REQ_BYTES):0] req_bytes_i,
  input  logic                           req_valid_i,
  output logic                           ready_o,
  output logic [7:0]                     resp_data_o,
  output logic                           resp_valid_o,
  // Controller-Slave interface
  output logic                           sclk_o,
  output logic                           sclk_en_o,
  output logic                           mo_o,
  input  logic                           mi_i);
```

Listing 3.1: 25CSM04 custom controller port map declaration

The first parameter, *CLK_DIV_FACTOR*, is the reference clock division factor. By default, it takes the value of 32. The second one is the already mentioned *MAX_REQ_BYTES* parameter. Its purpose is to dimension the signals *req_data_i* and *req_bytes_i* to take better advantage of the resources. By default, it takes the value of 4, so the sizes are 32 bits and 3 bits, respectively. The default values of the parameters can be overwritten in the module instantiation.

The clock signal generation is an example of a detail to be further explored. Listing 3.2 shows the implementation of the clock frequency reduction. We have chosen to generate only clocks using powers of 2 as frequency divisors. We use a counter sized with the value of the *CLK_DIV_FACTOR* parameter, which increments at each rising edge of the reference clock. In this way, we get a more stable signal by taking the most significant bit of the counter. Since it is very simple logic, it involves fewer logic gates than other implementations that would allow us to choose the frequency accurately.

```
localparam DIV_CLK_COUNT_LEN = $clog2(CLK_DIV_FACTOR);
...
logic [DIV_CLK_COUNT_LEN-1:0] clk_div_cnt;

always_ff @(posedge clk_i, negedge rstn_i) begin
  if (~rstn_i) begin
    clk_div_cnt <= 0;
  end else begin
    clk_div_cnt <= clk_div_cnt + 1;
  end
end

assign clk = clk_div_cnt[DIV_CLK_COUNT_LEN-1];
...
assign sclk_o = (sclk_en_o)? clk: 0;
```

Listing 3.2: Serial clock frequency division

Another reason that leads us to make this design decision is that achieving the maximum frequency at which the 25CSM04 can operate is not crucial. This memory is accessed only to load and run the bootloader in our use case. Once the code is in the main memory, the access latency will be that of the instruction cache, and therefore, we can afford this latency increase since it is only at boot time.

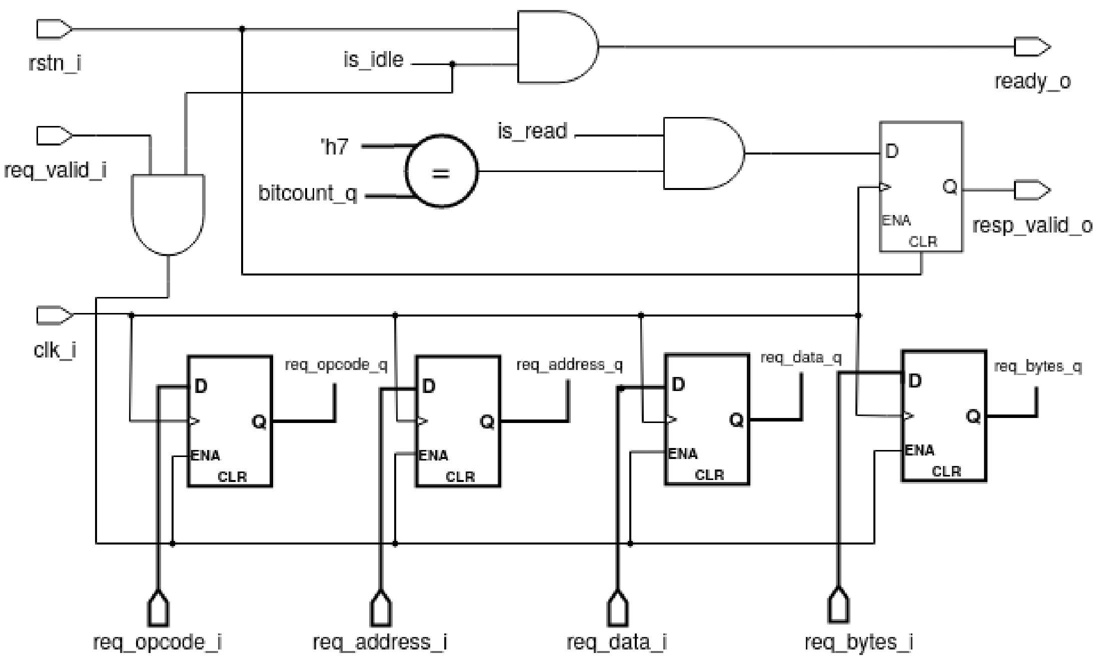


Figure 3.8: Handshake logic

The handshake is also an essential element. It is the mechanism through which, on the one hand, the controller indicates that it is ready to process a request or to deliver data, and the master, on the other hand, informs the controller that the data of the request is valid.

Figure 3.8 shows the combinational logic that implements the handshake. The signals shown in bold are composed of more than one bit. Although it is omitted in the diagram, we must consider that we have implemented a bit counter,which increments every cycle of the serial clock, and a byte counter. These will allow controlling the state machine, which we will talk about next, among other things. A change of the state clears these counters.

The controller is ready when it is in an idle state, and there is no reset. The *resp_valid_o* signal is only high when the controller has read a byte. At that time, there is valid data on the *resp_data_o port*. This implies two things; on the one hand, by depending on the bit counter, the *resp_valid_o* signal remains high one cycle of the serial clock, and on the other hand, the consumer must wait for this signal to go high as many times as bytes it has requested to read. Note that WRITE and WREN operations do not have a response from the controller. Finally, when the consumer raises the *req_valid_i* signal, as long as the controller is idle, the request's data is stored in registers to keep the information until the subsequent request.

The FSM of the controller, shown in Figure 3.9, is a Moore state machine. The initial state, which is reached asynchronously on reset, is the *STATE_IDLE* state. The remaining states have the purpose of either serially transferring opcode, address, and data or receiving the data bit by bit from a read operation. The three conditions that influence the change of states are:

1. The request is valid.
2. The byte counter matches the number of bytes expected to receive or transfer in that state.
3. The opcode of the instruction.

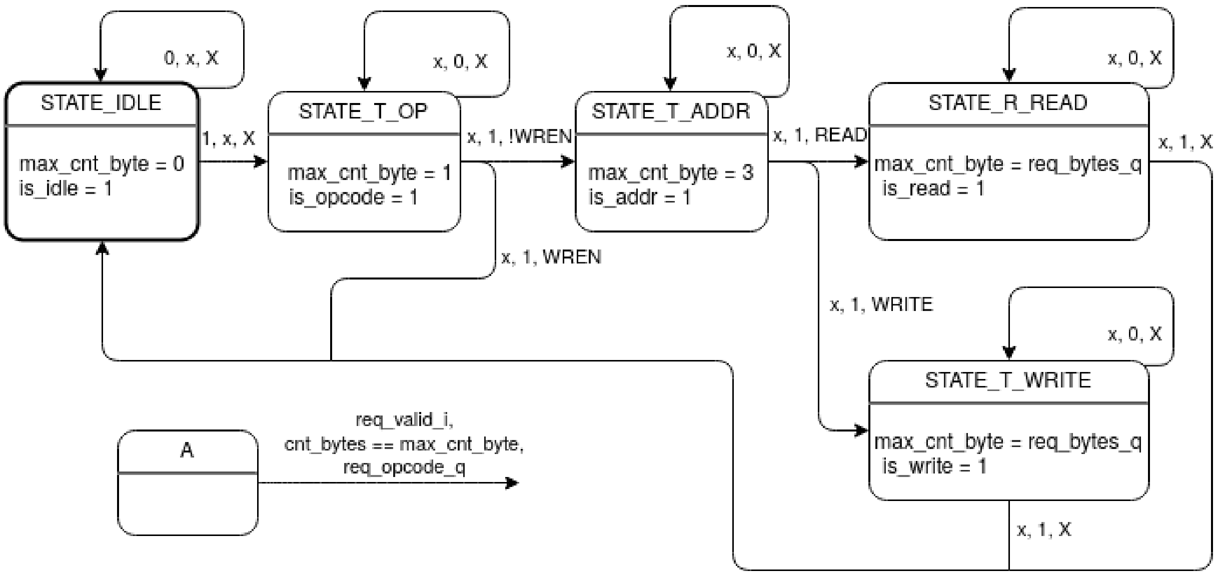


Figure 3.9: State machine

The controller will remain in *STATE_IDLE* until it receives a valid request, at which time it will change to the *STATE_T_OP* state. The controller enables the serial clock and starts transferring the opcode. After eight cycles of the serial clock, i.e., after transferring the opcode byte, it will return to the *STATE_IDLE* state in case of a WREN instruction. Otherwise, if it is a READ or WRITE instruction, it will switch to the *STATE_T_ADDR* state. After 24 cycles dedicated to the address, it will proceed to the *STATE_R_READ* or *STATE_T_WRITE* state, depending on whether the opcode corresponds to the READ or WRITE instruction, respectively. The cycles in which the controller will remain in these two states depend on the bytes requested. Finally, we return to the idle state at the end of the operation.

Shifter registers are fundamental in our implementation to perform serial communication. They are registers that shift the data in each cycle. As shown in the Listing 3.3, we use them to interact with the output and input signals of the SPI EEPROM.

```
always_ff @(posedge clk) begin: shift_regs
  dataShifterI <= {dataShifterI[6:0], mi_i};
  if (bitcount_q == 7) begin
    resp_data_o <= {dataShifterI[6:0], mi_i};
  end
end


always_ff @(negedge clk) begin
```

```
  if (bitcount_q == 0) begin
    if (is_opcode) begin
      dataShifterO <= req_opcode_q;
    end else if (is_addr) begin
      dataShifterO <= req_address_q[bytecount_q];
    end else if (is_write) begin
      dataShifterO <= req_data_q[bytecount_q];
    end
  end else if (sclk_en_o) begin
    dataShifterO <= dataShifterO[6:0] << 1;
  end
end

assign mo_o = dataShifterO[7];
```

Listing 3.3: Shiter registers

The data input register is shifted on the rising edges of the serial clock, and each shifted byte is propagated to the *resp_data_o* port. In contrast, the data output register is shifted on the falling edges. At the beginning of the byte, it takes value with the opcode, the corresponding byte of the address, or the data to be written depending on the controller's state.

4 Bootrom controller verification

In this chapter, we will detail the process followed to ensure that the behavior of the 25CSM04 controller complies with the intention of its implementation. This procedure is called verification, and for this purpose, we will create a module exclusively to test and verify the correct behavior of the controller detailed in the previous chapter. These modules are commonly known as *testbenches* and are generally used to verify designs through RTL simulations. Next, we will document the process of creating the testbench, and we will see how it can be handy to visualize and prove that the implemented design meets the requirements.

4.1 Writing the testbench

The testbench must provide the module to be verified with the necessary environment to perform the verification. On the one hand, that is generating the signals that stimulate the Device Under Test (or DUT) and, on the other hand, checking the results for these stimuli. In our case, the DUT is the bootrom controller.

4.1.1 25CSM04 simulation model

Generating the stimuli and checking logic for the Controller-Slave interface of the controller (shown in the Figure 3.7) is a costly and complex task. Instead, we will use a model of the 25CSM04 memory obtained from the resource bank of the Microchip website [16]. The model is an HDL module that behaves exactly like the device. This module uses non-synthesizable Verilog statements and is therefore only valid for simulations. In addition, in this case, the model includes a series of checks that indicate whether the input signals, such as input data and serial clock, have the latency and frequency supported by the physical device.

```
‘timescale 1ns/1ps
wire TimingCheckEnable = (RESET == 0) & (CS_N == 0);
specify
  specparam
    tHI  =  40,                                 // SCK pulse width - high
    tLO  =  40,                                 // SCK pulse width - low
    tSU  =  10,                                 // SI to SCK setup time
    tHD  =  10,                                 // SI to SCK hold time
    tHS  =  10,                                 // HOLD_N to SCK setup time
    tHH  =  10,                                 // HOLD_N to SCK hold time
    tCSD =  30,                                 // CS_N disable time
```

```
12      tCSS =  30,                               // CS_N to SCK setup time
13      tCSH =  30,                               // CS_N to SCK hold time
14      tCLD = 50,                                // Clock delay time
15      tCLE = 50;                                // Clock enable time
16
17
18    $width (posedge SCK,  tHI);
19    $width (negedge SCK,  tLO);
20    $width (posedge CS_N, tCSD);
21
22    $setup (SI, posedge SCK &&& TimingCheckEnable, tSU);
23    $setup (negedge CS_N, posedge SCK &&& TimingCheckEnable, tCSS);
24    $setup (negedge SCK, negedge HOLD_N &&& TimingCheckEnable, tHS);
25    $setup (posedge CS_N, posedge SCK &&& TimingCheckEnable, tCLD);
26
27    $hold  (posedge SCK    &&& TimingCheckEnable, SI,   tHD);
28    $hold  (posedge SCK    &&& TimingCheckEnable, posedge CS_N, tCSH);
29    $hold  (posedge HOLD_N &&& TimingCheckEnable, posedge SCK,  tHH);
30    $hold  (posedge SCK    &&& TimingCheckEnable, negedge CS_N, tCLE);
31  endspecify
```

Listing 4.1: 25CSM04 model timing checks

The Listing 4.1 shows the existing timing checks in the model. All specified parameters are expressed in nanoseconds and are defined in the 25CSM04 datasheet [14].

The *$width* task checks that the time elapsed between the transition specified in the first parameter and the subsequent transition of that signal is at least the time specified in the second parameter. This task checks that the signal pulses comply with the constraints. For example, lines 18 and 19 checks that the serial clock's positive and negative pulses are not less than 40 nanoseconds. The statement in line 20, on the other hand, ensures that the time between two device assertions is greater than 30 nanoseconds.

The *$setup* task verifies that at least the time defined in the third parameter elapses between the change in the signal specified in the first parameter and the reference time specified in the second parameter.

Finally, the *$hold* task works similarly to the previous one; the only difference is that the reference time is defined in the first parameter and the signal in the second. Therefore, line 22 verifies that the input data is stable 10 nanoseconds before the rising edge of the serial clock, while line 27 ensures no change in the signal 10 nanoseconds after the edge.

4.1.2 Test definition and implementation

Apart from the timing check provided by the model, it is necessary to define what other aspects of the controller implementation we want to check. Above all, we must ensure that the READ, WREN, and WRITE instructions work as expected. For this purpose, we will perform the following tests:

- Single-byte read operations.
- 4-byte read operations.

- One page write operations.

We are specifically interested in testing the 4-byte reads since, being the size of the instructions the core works with, it will be the most common and crucial use case. To generate the stimuli for the Master-Controller interface (shown in the Figure 3.7), we will create a task for each test we want to perform.

```
task do_read;
  input [23:0] addr;
  begin
    req_opcode = `READ;
    req_address = addr;
    req_bytes = 9'h1;
    #CLK_PERIOD;
    wait(ready);
    req_valid = 1;
    #CLK_PERIOD;
    req_valid = 0;
    wait(~ready);
    check_read(addr);
  end
endtask

```

Listing 4.2: Singly-byte read stimuli generation task

```
task do_read_4;
  input [23:0] addr;
  input [31:0] chk_val;
  begin
    req_opcode = `READ;
    req_address = addr;
    req_bytes = 9'h4;
    #CLK_PERIOD;
    wait(ready)
    req_valid = 1;
    #CLK_PERIOD;
    req_valid = 0;
    wait(~ready);
    check_read_4(addr, chk_val);
  end
endtask
```

Listing 4.3: 4-byte read stimuli generation task

Listings 4.2 and 4.3 show the stimulus generation tasks for the single-byte and 4-byte read operation, respectively. The first one has the address of the byte we want to read as a parameter. The task prepares the opcode, the address, and the number of bytes requested for at least one cycle before raising the valid request signal. This signal is driven high for only one cycle when the controller is ready, and then the results of the read operation are checked by calling the *check_read()* task, which we will examine later. The second one is very similar; the most relevant change is that the bytes requested are 4. The *chk_val* parameter is used to check that the read operation is correct. It contains the data expected to be read and is passed directly to the *check_read_4()* task, which verifies the 4 byte reads.

```
task check_read;
  input [24:0] addr;
  begin
    wait(resp_valid);
    if(resp_data === test_mem[addr]) begin
      $display("Test 1-byte read [%d] addr %h: OK (read: %h, exepected: %h)",
     cnt, addr, resp_data, test_mem[addr]);
    end else begin
      $display("Test 1-byte read [%d] addr %h: FAIL (read: %h, expected: %h)",
     cnt, addr, resp_data, test_mem[addr]);
      $stop;
    end
  end
```

```
endtask
```

Listing 4.4: Singly-byte read stimuli generation task

As we can see in Listing 4.4, the task *check_read()* waits until the controller sets the *resp_valid* signal high, at which time it checks that the data in the controller's *resp_data_o* output port is as expected. To determine which is the desired data, we use a perfect memory array of the same size as the EEPROM: *test_mem*. SystemVerilog provides a system function called *$readmemh()*, which allows us to dump the hexadecimal contents of a file in our system into the memory array. Since the internal memory of the 25CSM04 model is uninitialized, it is necessary to modify it by adding this system function so that the contents of the model memory and the testbench memory are the same at the start of the simulation. If the response data from the controller does not match the data in the memory array at the same address, the *$stop* function call will stop the simulation, helping the developer find the bug.

```
task check_read_4;
  input [24:0] addr;
  input [31:0] chk_val;
  begin
    wait(resp_valid);
    resp_data4[7:0] <= resp_data;
    wait(~resp_valid); wait(resp_valid);
    resp_data4[15:8] <= resp_data;
    wait(~resp_valid); wait(resp_valid);
    resp_data4[23:16] <= resp_data;
    wait(~resp_valid); wait(resp_valid);
    resp_data4[31:24] <= resp_data;
    wait(~resp_valid);
    if(resp_data4 === chk_val ) begin
      $display("Test 4-byte read [%d] addr %h: OK (read: %h, expected: %h)", cnt
    , addr, resp_data4, chk_val);
    end else begin
      $display("Test 4-byte read [%d] addr %h: FAIL (read: %h, expected: %h)",
    cnt, addr, resp_data4, chk_val);
      $stop;
    end
  end
endtask
```

Listing 4.5: 4-byte read stimuli generation task

Listing 4.5 shows the implementation of the *check_read_4()* task. When checking a four-byte read operation, the controller must drive high the *resp_valid* signal four times, and each one of them, the task saves the value so that it obtains the complete data at the end. Once we have the entire data, the comparison is made with the *chk_val* input, which must contain the expected value. In this case, the fact that the desired value comes as a parameter allows us, as we will see below, to reuse this task to check the correct behavior of the writes operations.

```
task do_write;
  begin
    req_opcode = `WREN;
    #CLK_PERIOD;
    wait(ready)
    req_valid = 1;
    #CLK_PERIOD;
    req_valid = 0;
    wait(~ready)
    req_opcode = `WRITE;
    req_address = 24'b0;
    req_bytes = 9'd256;
    req_data = {2048{1'b1}};
    wait(ready)
    #CLK_PERIOD;
    req_valid = 1;
    #CLK_PERIOD;
    req_valid = 0;
    wait(~ready); wait(ready);
    $display("Test page write [%d] write 0xff in page %h", cnt, req_address);
    #5000000
    do_read_4($urandom_range('h0, 'hfc), 32'hffffffff);
  end
endtask
```

Listing 4.6: Page write stimuli generation task

Listing 4.6 shows the *do_write()* task. This task generates the stimuli to perform a write of a page. To do so, it first generates a WREN request, necessary to enable writes. Then it prepares the WRITE request to write ones to all memory locations of the first page. Once the write operation has finished and the controller raises the ready signal, the task waits for five milliseconds corresponding to the EEPROM internal write cycle time. After this time, the write has already propagated to the device's internal memory. We perform the write check using the *do_read_4()* task described above. The first parameter, the address, is a random position within the first page, and the expected value, defined in the second parameter, is 4 bytes with all its bits set to 1. If the read result does not match, we know that either the WREN operation or the WRITE operation has not been performed correctly.

```
spi_eeprom_req # (.MAX_REQ_BYTES (256), .CLK_DIV_FACTOR (4))
DUT ( // Master-Controller interface
    .clk_i (clk), .rstn_i (~reset), .req_opcode_i (req_opcode), .req_address_i (
    req_address), .req_data_i (req_data), .req_bytes_i (req_bytes), .req_valid_i
     (req_valid), .ready_o (ready), .resp_data_o (resp_data), .resp_valid_o (
    resp_valid),
    // Controller-Slave interface
    .mo_o (mo), .sclk_o (spi_clk), .sclk_en_o (spi_clk_en), .mi_i (mi));

M25CSM04  mem0 (.CS_N (cs_n), .SO (mi), .WP_N (1'h1), .SI (mo), .SCK (spi_clk),
    .HOLD_N (1'h1), .RESET(reset));

always #CLK_PULSE clk = ~clk;
assign cs_n = ~spi_clk_en;

initial begin
  clk = 1;
  reset = 1;
  req_valid = 0;
  $readmemh("bootrom_content.hex", test_mem);
  #120
  reset = 0;
  cnt = 0;
  while (cnt < `N_TESTS) begin
    do_read($urandom_range(`ADDR_MIN, `ADDR_MAX));
    cnt++;
  end
...
end
```

Listing 4.7: Module instantiations and conections in the testbench

Finally, it is necessary to instantiate the bootrom controller module and the memory model and interconnect them (see Listing 4.7). The *initial* clause is the core of our testbench. First, we generate the reset signal and initialize the *test_mem* memory to perform the subsequent tests. Then we use the tasks to generate the different stimuli detailed above.

4.2 Visualization and behavioral testing

We can run the testbench and analyze the results using an HDL language simulator. In our case, we have used ModelSim [17], which allows us to perform RTL simulations and observe the changes of different signals through a waveform.

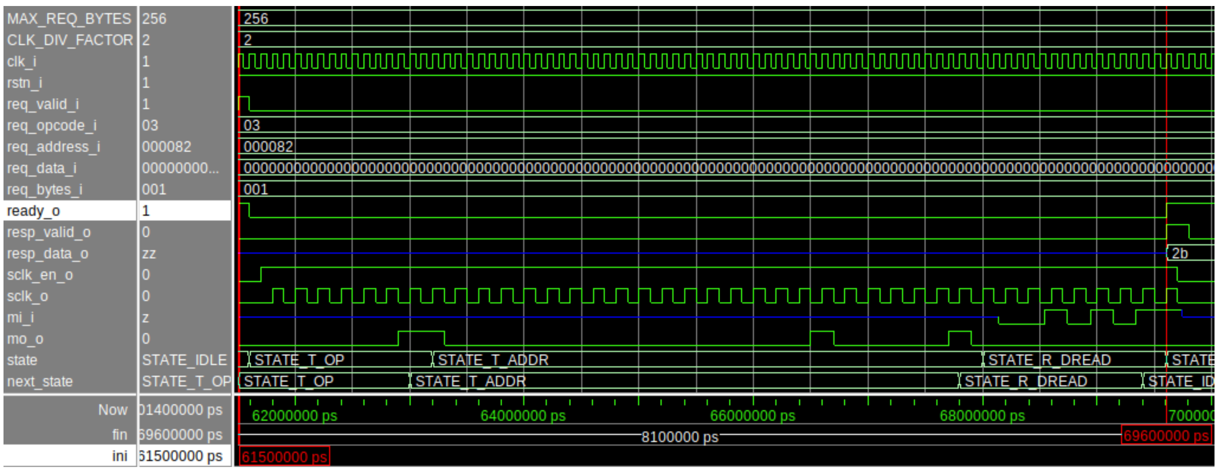


Figure 4.1: Single-byte read request waveform

Figure 4.1 shows the signals from the bootrom controller interface during a 1-byte read operation. The start and end of the operation are marked with markers. To better visualize the image, we have used a clock division factor of 2, defined by the parameter CLK_DIV_FACTOR. This means that the period of the serial clock is twice that of the reference clock. When the master raises the *req_valid_i* signal, the opcode at the controller input is 3, which corresponds to the READ operation, the specified address is 0x82, and only 1 byte is requested. When the controller sees the valid request, it drives the *ready_o* signal low, switches to the STATE_T_OP state, and enables the serial clock, thus activating the EEPROM. This state lasts for eight cycles of the serial clock, and in the *mo_o* signal, we can see that the bit corresponding to the opcode is transmitted at each cycle, starting with the most significant one. Immediately after, the controller switches to the address transmission state. This state, as expected, lasts for 24 serial clock cycles, in which we can see the corresponding bit of the address on the same signal. Then, the controller switches to the STATE_R_READ state, wherein the subsequent eight serial clock cycles, the input signal *mi_i* is fetched. Once the operation finishes, the data is at the output port *resp_data_o*, and the controller raises the *resp_valid_o* signal for one cycle. In addition, the controller is in an idle state and, accordingly, disables the serial clock and sets the *ready_o* port high again.

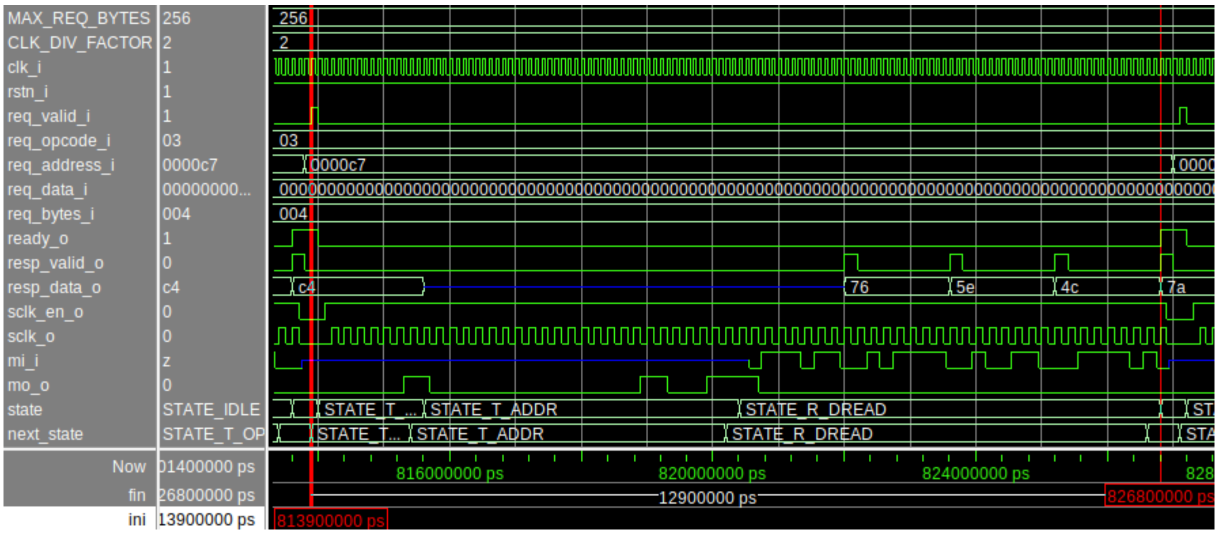


Figure 4.2: Four-byte read request waveform

Figure 4.2 shows the waveform resulting from a 4-byte read operation. Unlike the previous example, the STATE_R_READ state lasts for 32 serial clock cycles, since now we requested four bytes instead of only one. The data remains in the *resp_data_o* port for each byte read, and the valid response signal is raised for one cycle. In this case, assuming the memory is little-endian, the data stored at address 0xC7 is 0x7A4C5E76.

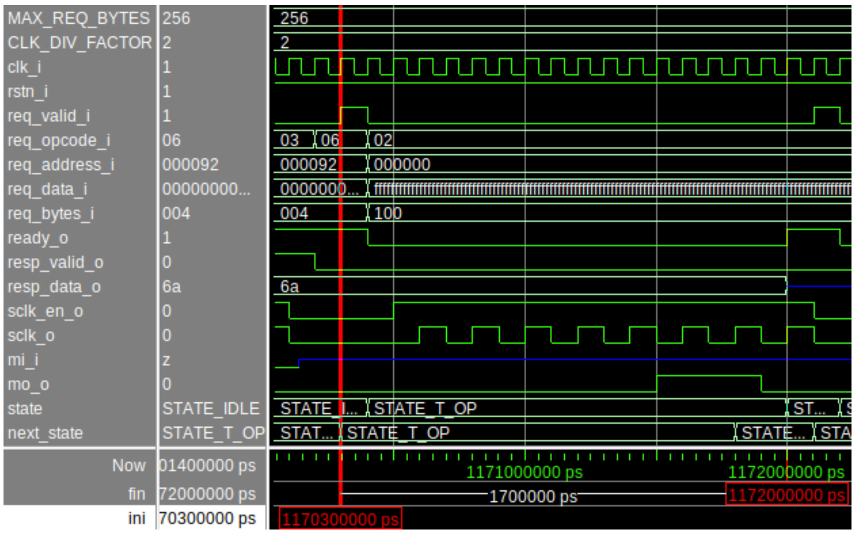


Figure 4.3: Write enable (WREN) request waveform

The WREN operation, whose opcode is 6, is required before writing. Figure 4.3 shows the waveform corresponding to this operation. In this case, the waveform allows us to observe that even if the inputs change once the request is valid, it does not affect the procedure. When the controller starts to process the WREN operation, driving the *ready_o* signal low, we observe how the input opcode value immediately changes to 2, which corresponds to the WRITE operation. However, we can observe that a six is transmitted during the opcode transmission state and that the next state transition is to STATE_IDLE, which, as shown in Figure 3.9 is to be expected for the WREN operation.

Figure 4.4 shows the waveform corresponding to the write operation. The *req_bytes_i* port when the request is valid is 256 bytes, and the address is 0x00 so that this request will write the entire first page of the EEPROM. As we see in the *req_data_i* port, the data to write is 256 bytes with all its bits at 1. The procedure is identical to the reads detailed above until the address is transmitted. Then, the controller enters the STATE_T_DWRITE state, where, during the subsequent 2048 cycles of the serial clock, it transmits bit by bit the data to be written by the *mo_o* signal. Once the controller finishes transmitting the 256 bytes of data, it changes to the idle state, driving the *ready_o* signal high again and disabling the serial clock.

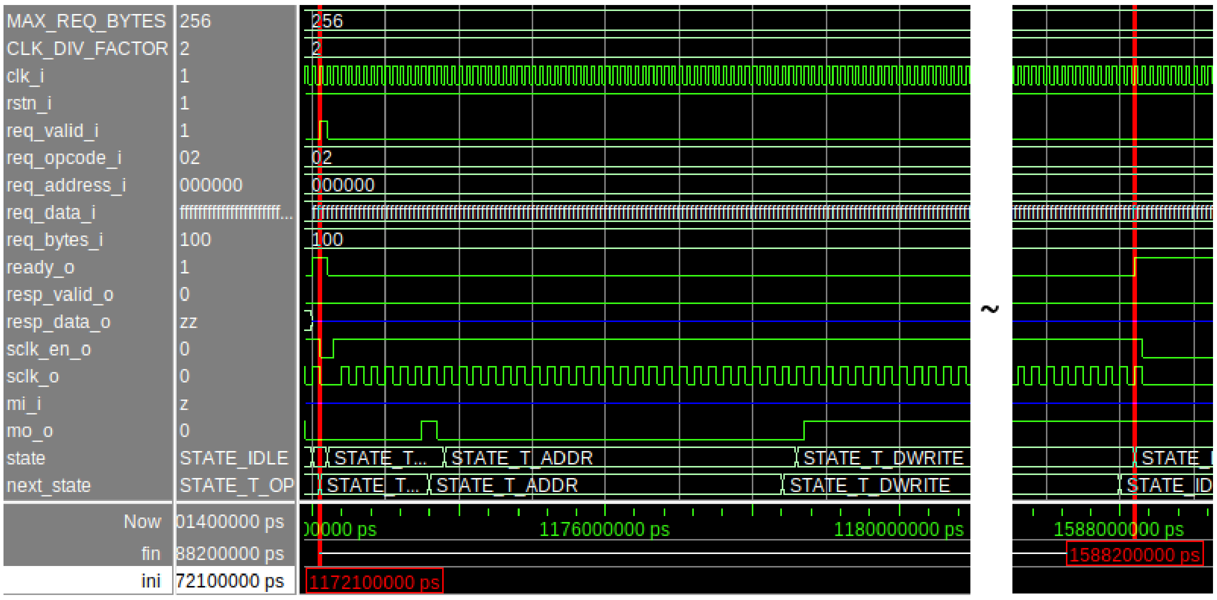
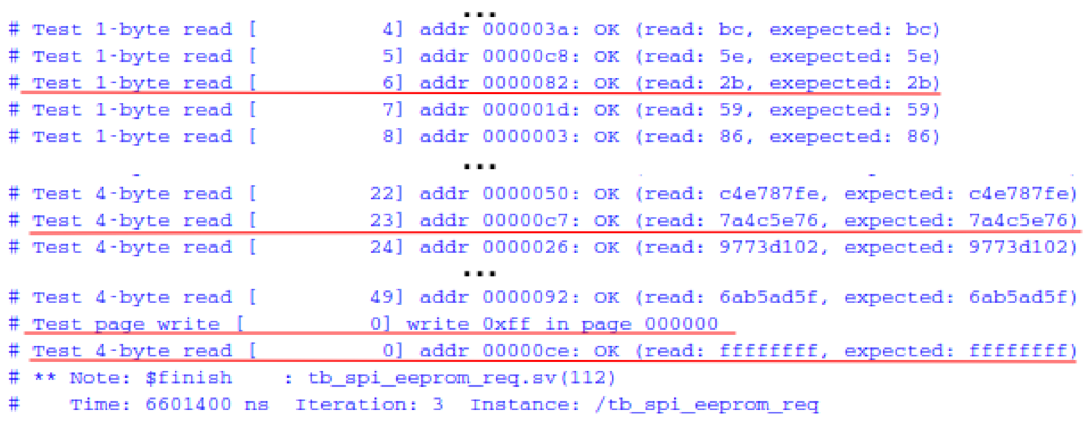


Figure 4.4: Page write (WRITE) request waveform

Finally, this process of verifying and checking the behavior does not need to be done manually by viewing the waveform; by adding the *$display* Verilog statements in the testbench, we can print information as the simulation runs. In Figure 4.5 we can identify, underlined in red, the messages corresponding to the waveforms shown in this chapter.



```
                                                ...
# Test 1-byte read [           4] addr 000003a: OK (read: bc, exepected: bc)
# Test 1-byte read [           5] addr 00000c8: OK (read: 5e, exepected: 5e)
# Test 1-byte read [           6] addr 0000082: OK (read: 2b, exepected: 2b)
# Test 1-byte read [           7] addr 000001d: OK (read: 59, exepected: 59)
# Test 1-byte read [           8] addr 0000003: OK (read: 86, exepected: 86)
                                                ...
# Test 4-byte read [          22] addr 0000050: OK (read: c4e787fe, expected: c4e787fe)
# Test 4-byte read [          23] addr 00000c7: OK (read: 7a4c5e76, expected: 7a4c5e76)
# Test 4-byte read [          24] addr 0000026: OK (read: 9773d102, expected: 9773d102)
                                                ...
# Test 4-byte read [          49] addr 0000092: OK (read: 6ab5ad5f, expected: 6ab5ad5f)
# Test page write [           0] write 0xff in page 000000
# Test 4-byte read [           0] addr 00000ce: OK (read: ffffffff, expected: ffffffff)
# ** Note: $finish    : tb_spi_eeprom_req.sv(112)
#    Time: 6601400 ns  Iteration: 3  Instance: /tb_spi_eeprom_req
```

Figure 4.5: Modelsim transcript window output

5 Bootrom controller integration

With the bootrom controller implemented and verified, we can proceed to the next step: integration into DRAC's 22 nm SoC. However, we will first analyze specific details of the SoC and processor to evaluate different integration strategies. Next, we will see what changes will need to be made to the current implementation.

5.1 DRAC SoC

The DRAC 22 nm is an SoC based on the lowRISC SoC project, developed by the lowRISC company. The lowRISC project is a 64-bit SoC design that integrates the Rocket core [13] to offer an FPGA-ready SoC distribution with open-source peripherals.

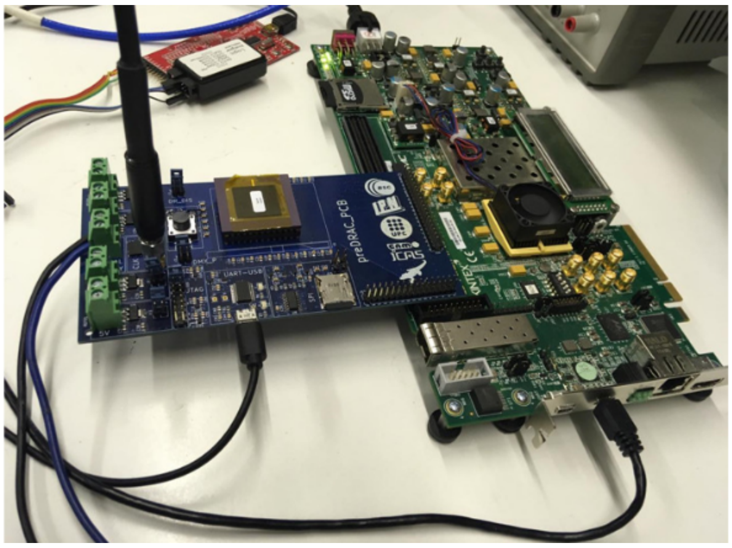


Figure 5.1: PreDRAC connected to an FPGA [1]

The DRAC SoC modifies the lowRISC project caches to enable a SIMD pipeline and incorporates Advanced eXtensible Interface (AXI) peripherals. In addition, the design of the SoC is divided into two differentiated parts: *fpga_top* and *asic_top*. As mentioned before, the *asic_top* is the ASIC-oriented implementation of the SoC. At the same time, the *fpga_top* contains all those modules that, either because it was not possible to implement it for ASIC or to keep a backup option in case the ASIC fails, will be burned into an FPGA that will be connected to the rest of the SoC. This environment, shown in Figure 5.1, is already used in preDRAC, the predecessor of DRAC 22 nm.

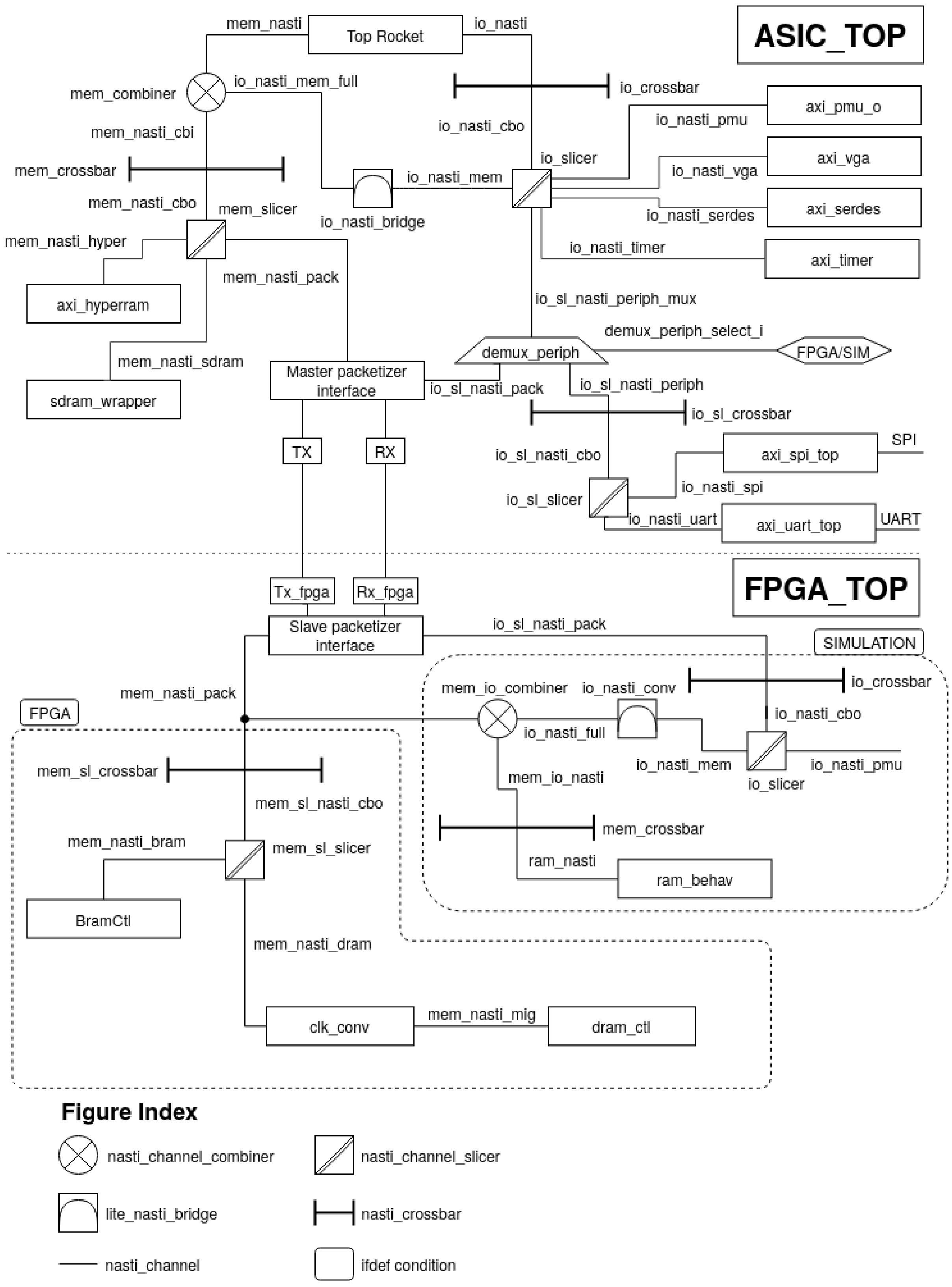


Figure 5.2: DRAC 22nm SoC simplified block diagram

This SoC has a modular design, allowing us to add different functionalities without significant structural changes. Figure 5.2 shows a simplified block diagram of the DRAC 22 nm SoC. Compared to the implementation of its predecessor, the current design includes the hyperRAM, SDRAM and SerDes controllers, and the timer module. In addition, it also has a CSU and two PLLs, along with other components that we have not added to the schematic for simplicity.

This design has not been manufactured yet; it has only been verified in simulation and FPGA emulation. The DRAC 22 nm tape-out is scheduled for January 2022. As mentioned above, one of the key objectives for this tape-out is to make the SoC fully functional without connecting it to an FPGA. However, we prefer to be conservative and keep in the design the components for connecting the *asic_top* to the *fpga_top*. For example, the hyperRAM and SDRAM controllers provide access to the chip's main memory. However, should it turn out during the bring-up process that these modules do not work correctly, in this case, the SoC would maintain a fallback option for this exact purpose: accessing the *dram_ctl* module present in the *fpga_top* through the *packetizer* interface, as was done in preDRAC, its predecessor. Therefore, it is worth mentioning that we must guarantee a backup option in case of failure of the bootrom controller.

5.2 Sargantana core

Sargantana [11], represented by the Top Rocket block in the Figure 5.2, is the next generation of the Lagarto core [1], present in the preDRAC SoC. Sargantana is a 7-stage single-stream RISC-V core that supports the base integer instruction set RV64I, the G and V extensions, and the privileged instruction set v.1.11. It also features a floating-point pipeline and SIMD. In the CoreMark benchmark [10], Sargantana gets 2 CoreMark/MHz. Next, as the integration of a bootrom should not affect the back-end of the kernel, we will detail only the front-end pipeline.

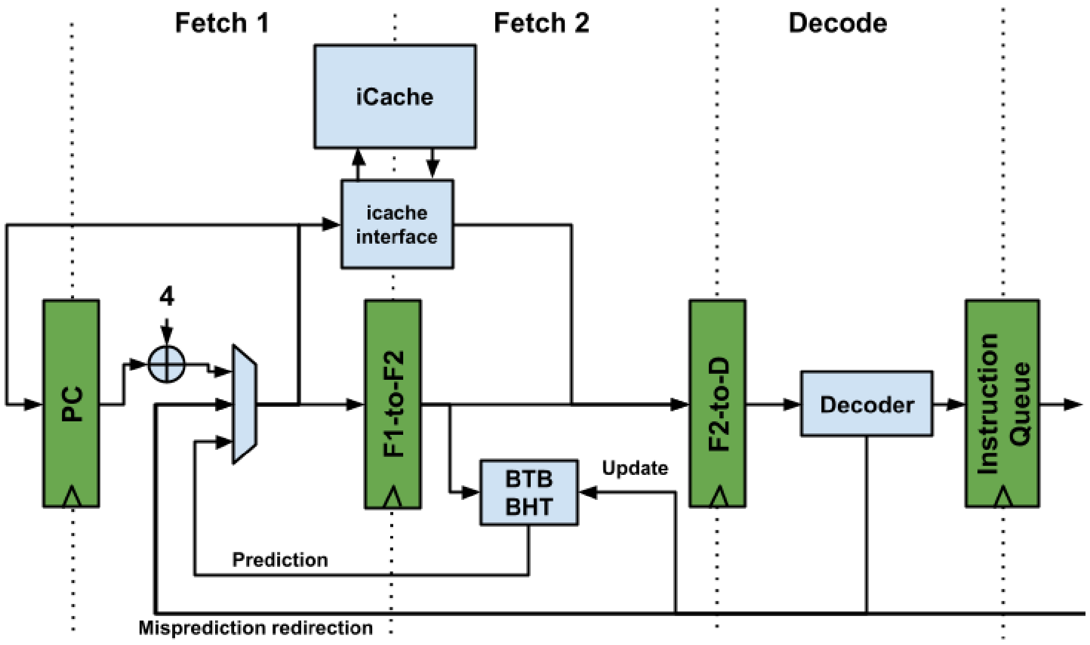


Figure 5.3: Sargantana Front-End pipeline

5.2.1 Sargantana Front-End

The Sargantana front-end, as shown in Figure 5.3, has three stages: two fetch stages and a decode stage. The front-end and back-end are decoupled by an instruction queue. The operations performed in each of the stages are as follows:

- **Fetch1:** Send a request with the current Program Counter (PC) to the instruction cache through the cache interface.
- **Fetch2:** Receive through the icache interface the response from the instruction cache. This response can be a page exception. In case of an instruction cache miss, the front-end will pause until the response from the cache arrives. Finally, the Branch Target Buffer (BTB) and Branch History Table (BHT) are accessed to predict the next PC.
- **Decode:** In this stage, the instruction obtained in the previous stage is decoded, and then it is pushed to the instruction queue. In addition, the JAL instructions update the PC with the destination address of the jump.

5.3 Approach evaluation

The objective of integrating the botroom controller detailed in the previous chapters is to provide the chip with a mechanism to bring the data read from the 25CSM04 memory to the Sargantana core pipeline. To do this, we had to decide which of the following strategies was the most appropriate.

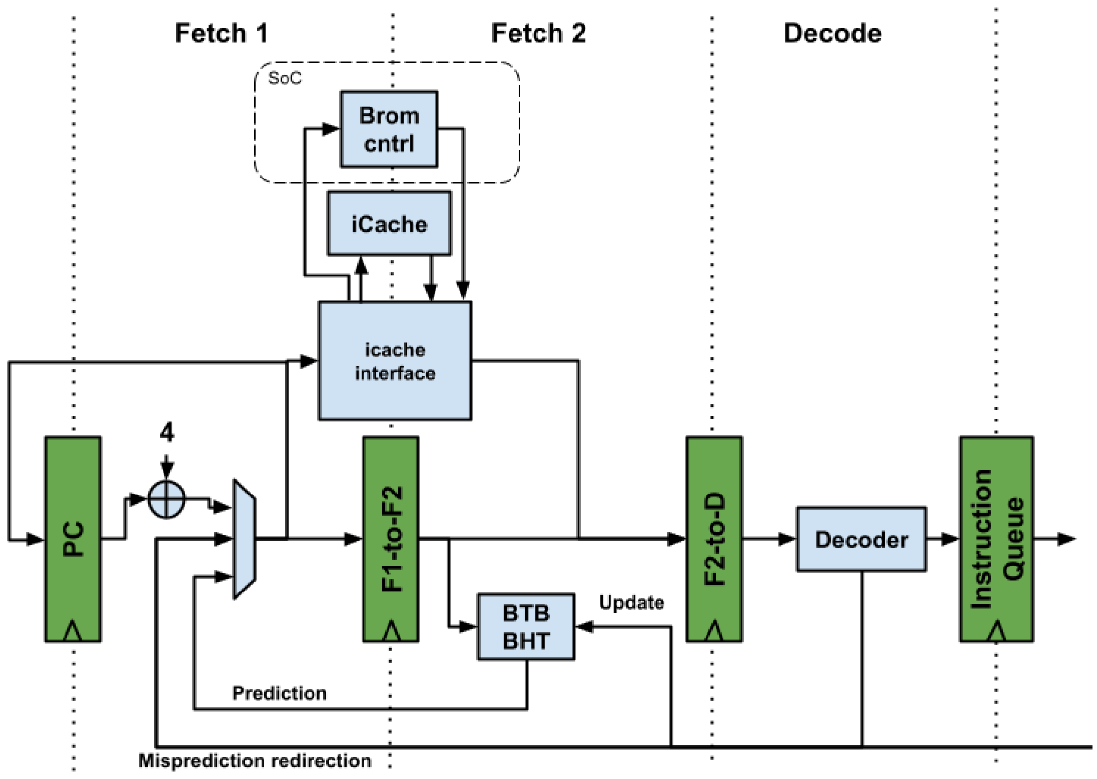


Figure 5.4: Sargantana Front-End pipeline after the Custom Path strategy integration

The AXI Wrapper strategy consists of providing the bootrom controller with an AXI-lite interface. We consider the Bootrom an I/O device due to its high access latency. The first-level data cache provides a mechanism for getting data from the I/O devices since they are not present in the memory hierarchy. This mechanism consists of requesting through an AXI-lite bus hierarchy. The request would be handled by the AXI-lite wrapper of the bootrom controller instead of by the memory hierarchy. However, the first-level instruction cache does not implement this mechanism.

The Custom Path strategy consists of creating a dedicated path for the data read through the bootrom controller to the Sargantana core front-end. Although this option is more straightforward, it has some limitations; for example, the bootrom content does not pass through the memory hierarchy. Therefore, no data read (nor write) operations should be performed on the address range reserved for the bootrom. This restriction will mainly affect the bootloader.

Finally, we have decided to implement the custom path strategy. We consider that the AXI Wrapper strategy would be extremely challenging to meet the project constraint of being on time for the RTL freeze date for the next tape-out. Furthermore, this would require replicating to the instruction cache the mechanism provided by the data cache for I/O accesses, and we believe that this is beyond the project's scope. Figure 5.4 shows what the front-end pipeline would look like after integrating the bootrom controller following the Custom Path strategy. Note that, in any case, the bootrom controller is not part of the core but of the SoC.

5.4 RTL design modifications

To integrate the bootrom controller, we need to modify the RTL design of both the Sargantana core and the SoC. In the core design, we need to adjust the front-end to make the bootrom address range requests not to the instruction cache but to the bootrom controller. On the SoC side, it will be necessary to instantiate the controller and connect it correctly with the core in the SoC RTL design.

5.4.1 Sargantana Front-End modifications

Figure 5.5 shows the name of the most relevant front-end modules of the RTL design part. In this way, the reader can situate the design changes that we will show next. The *top_drac* module is the top of the Sargantana core, where the *datapath* module, which contains the entire core pipeline, is instantiated. The first core stage, the Fetch 1, is located in the *if_stage_1* module and has all the necessary submodules to perform this stage's operations. In the same way, the modules *if_stage_2* and *id_stage* correspond to the Fetch 2 and Decode stages, respectively.

The *if_stage_1* module, once it knows the value of the new PC, requests the 32-bit instruction corresponding to the PC from the *icache_interface* module. This module manages the requests made to the instruction cache. For example, it selects the requested instruction from the 128 bits that the cache responds to offer it to the *if_stage_2* module. The dashed lines shown in the Figure 5.5 represent the data paths that we will add to integrate the bootrom. These paths will

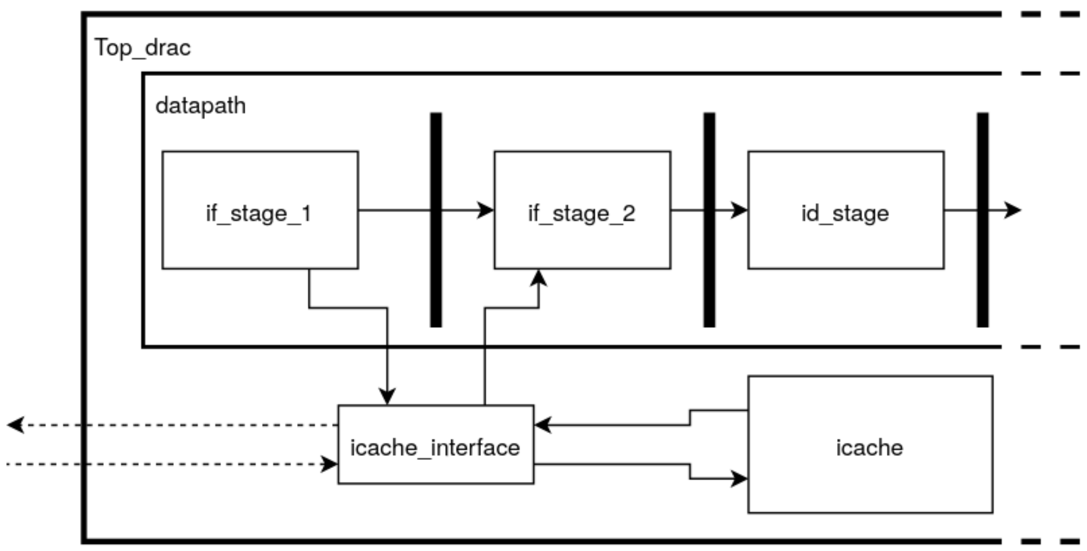


Figure 5.5: Sargantana front-end module names

go from the *icache_interface* module to the SoC and vice versa.

The instruction cache has a latency of 1 cycle on a hit since it follows a Virtually Indexed, Physically Tagged (VIPT) structure and accesses the Translation Lookaside Buffer (TLB) in parallel with the tag array. Note that this adds a limitation: in the *icache_interface* module, we do not know the physical address, only the virtual one. It could be the case that the virtual address corresponds to the bootrom range but once translated, the physical address does not.

Regarding the fallback option, we will implement a CSR dedicated to selecting the boot flow, as we will detail later. For now, it is only necessary to know that we will use a signal to indicate if we want to boot through the bootrom controller or if we prefer to boot as the preDRAC SoC did, with an FPGA that provides the data to the memory hierarchy through the *packetizer*. We will call the CSR containing this information CSR_SPI_CONFIG.

In summary, the additional data we will need in the *icache_interface* module are:

- If the address translation is enabled or not.
- The value of CSR_SPI_CONFIG.
- If the address belong to the bootrom address range.

Listing 5.1 shows all the inputs and outputs we have added to the *icache_interface* module. The *en_translation_i* signal is 1 when address translations are enabled. When the *csr_spi_config_i* signal equals 0, it will boot through the bootrom controller. Otherwise, the controller is disabled and the fallback option is chosen. We have also added all the necessary signals to interface to the bootrom controller: the address and the valid request signal, as outputs; and the data and the valid response signal together with the bootrom controller status (ready/busy), as inputs.

Below we will show fragments of the module's implementation before and after the integration, either in diagrams or listings. The goal is not to give the reader a perfect understanding of the original implementation but to understand how the introduced changes modify the module's behavior. Table 5.1 details some of the signals of the *icache_interface* module interface to facilitate the understanding of these fragments.

```

module icache_interface(
...
input logic              en_translation_i,
input logic              csr_spi_config_i,
...
// Response input signals from bootrom
input logic              brom_ready_i,
input logic [31:0]       brom_resp_data_i,
input logic              brom_resp_valid_i,

// Request output signals to bootrom
output logic [23:0]      brom_req_address_o,
output logic             brom_req_valid_o,
...
);
```

Listing 5.1: Inputs and outputs added to the icache_interface module

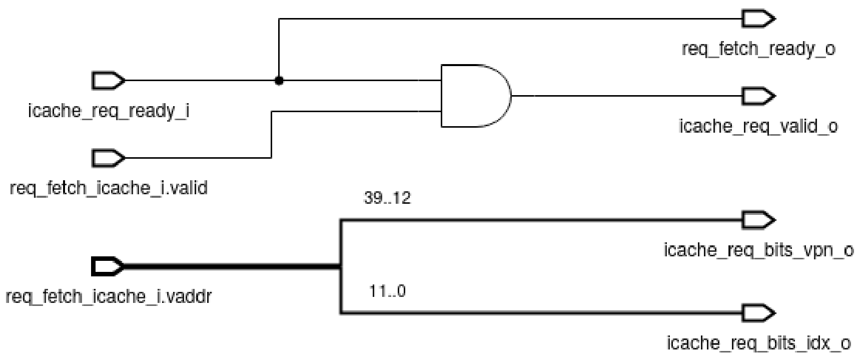
Name	Source	Destination	Description
icache_req_ready_i	icache	icache_interface	Instruction cache ready to handle new requests
icache_req_valid_o	icache_interface	icache	New petition to instruction cache is valid
icache_req_bits_vpn_o	icache_interface	icache	Virtual page number corresponding to address
icache_req_bits_idx_o	icache_interface	icache	Offset of the VPN of the corresponding address
req_fetch_icache_i.vaddr	if_stage_1	icache_interface	Virtual address from the Fetch 1 stage
req_fetch_icache_i.valid	if_stage_1	icache_interface	Address form the Fetch 1 stage is valid
req_fetch_ready_o	icache_interface	control_unit	The device that must handle the request is ready
icache_resp_datablock_i	icache	icache_interface	128-bit response data block
icache_resp_valid_i	icache	icache_interface	Response data from icache is valid
tlb_resp_xcp_if_i	icache	icache_interface	Instruction page fault exception from icache
resp_icache_fetch_o.data	icache_interface	if_stage_2	Fetched instruction to Fetch 2 stage
resp_icache_fetch_o.valid	icache_interface	if_stage_2	Fetched instruction is valid
resp_icache_fetch_o.instr_page_fault	icache_interface	if_stage_2	The icache access generated an intruction page fault

Table 5.1: Source and destination modules, and a brief description of the relevant signals of the *icache_interface* module interface.

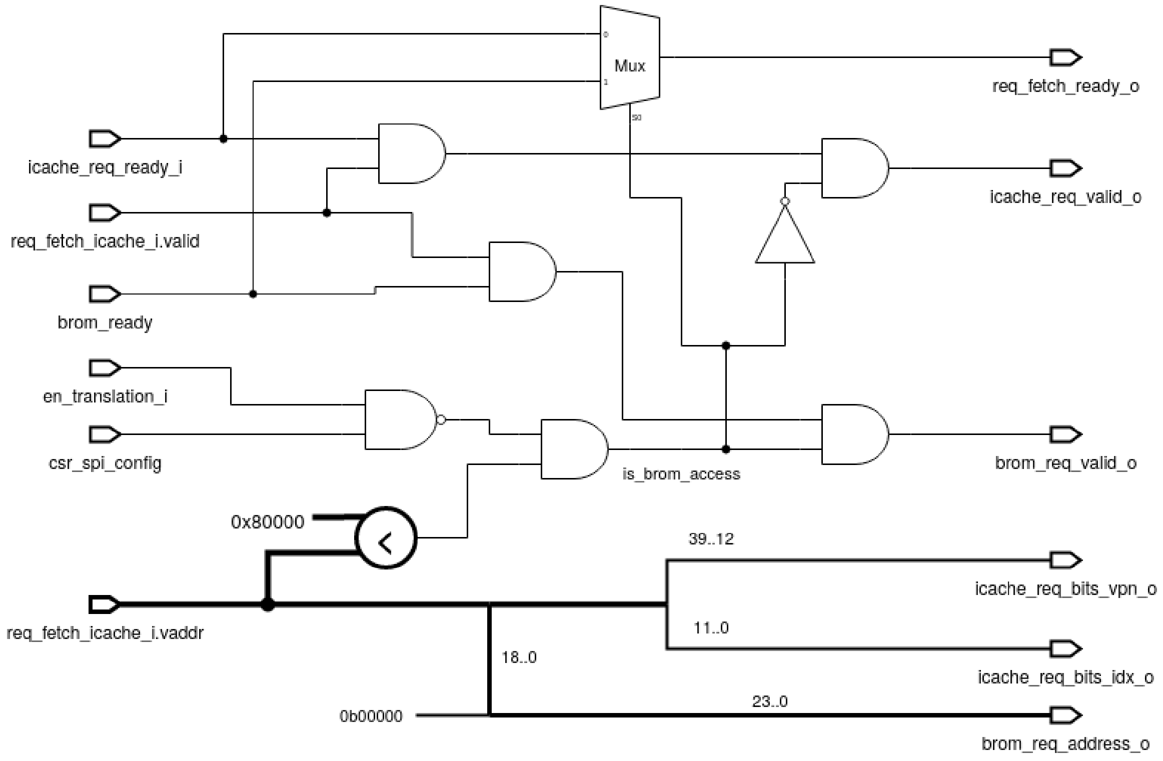
As we can see in Figure 5.6a, initially, the request is propagated to the instruction cache when it is ready to accept a new request and the address is valid. However, we must now discriminate the requests corresponding to the bootrom from those that must reach the instruction cache. We have defined the following instructions to consider that a request must access the bootrom:

- Address translations must be disabled.
- The value of the *csr_spi_config* must be 0.
- The access address, according to Table 2.2, must be less than 0x80000.

The first of these conditions simplifies the paradigm: by not translating the addresses, we are working with physical addresses, avoiding the problems already mentioned of working with



(a) Request-related logic in *icache_interface* module before the integration

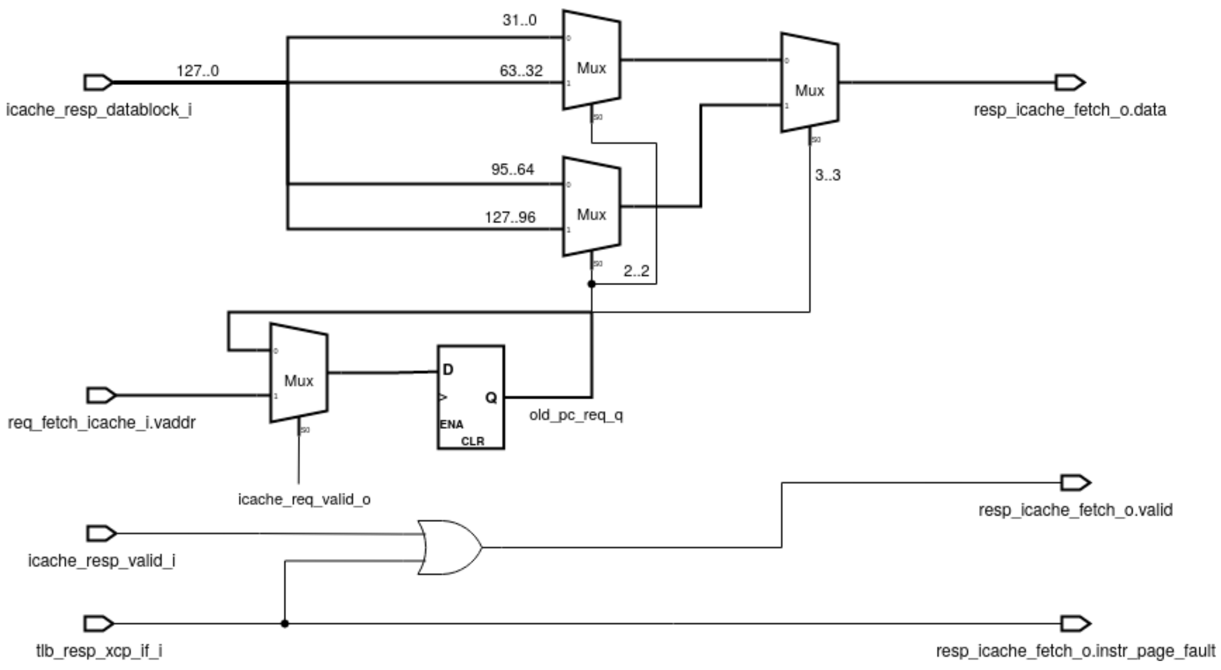


(b) Request-related logic in *icache_interface* module after the integration

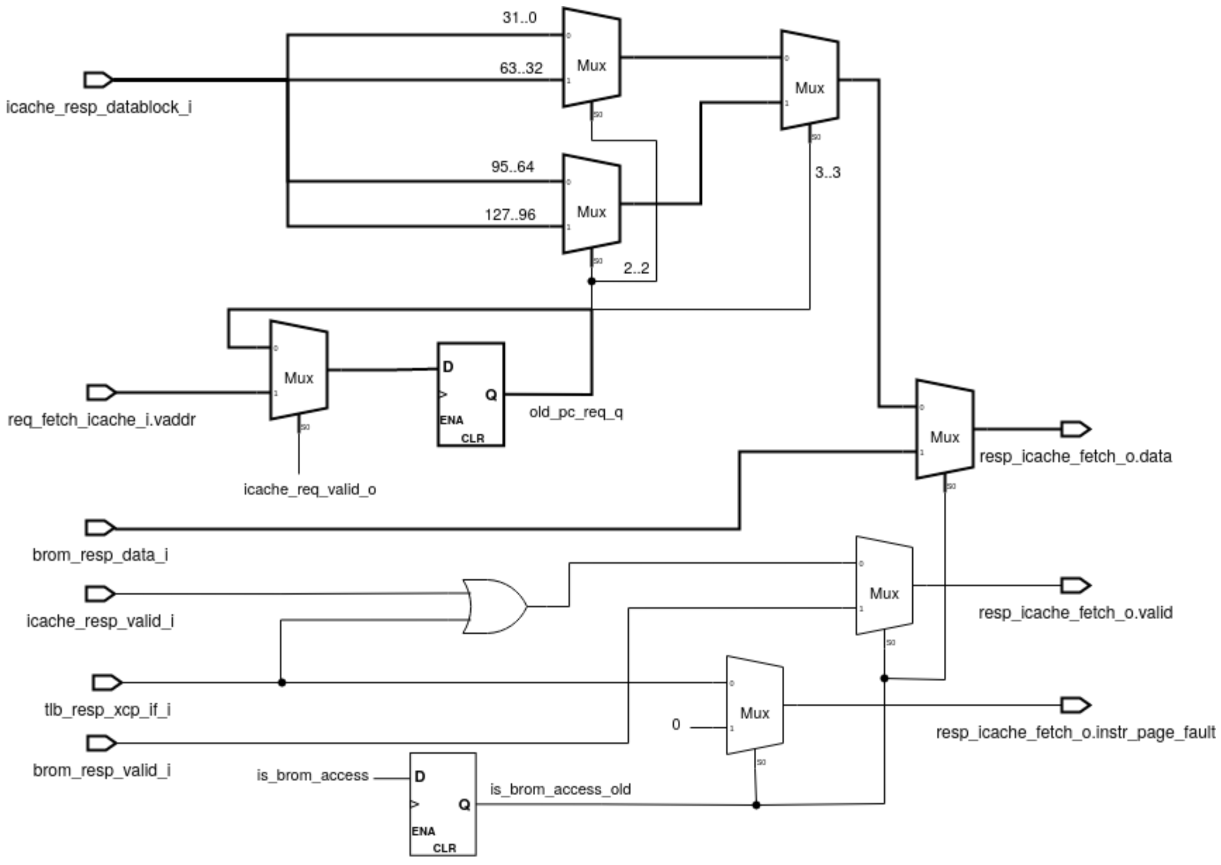
Figure 5.6: Comparison of request-related logic before and after integration

virtual addresses. Moreover, the management of address translations has much more to do with operating systems than with bootloaders. In this way, when an operating system or program activates the translation, it has already been loaded into the main memory of the chip. Therefore, it does not need to access the bootrom anymore, and consequently, there is no reason to work with virtual addresses when discriminating bootrom accesses.

The intermediate signal *is_brom_access* in Figure 5.6b indicates whether or not the current request meets the three conditions we have defined to determine if the access corresponds to the bootrom. However, to propagate the request, we must first determine that the address coming from the *if_stage_1* module is valid and that the bootrom controller is ready to handle the new request. If the petition fulfills all the conditions, the output *brom_req_valid_o* is driven high.



(a) Response-related logic in *icache_interface* module before the integration



(b) Response-related logic in *icache_interface* module after the integration

Figure 5.7: Comparison of response-related logic before and after integration

The bootrom access address, contained in the *brom_req_address_o* output, is the 19 bits of the address provided by the Fetch 1 stage padded with zeros to reach the 24 bits required by the bootrom controller. Recall that only 19 bits are needed to address all the EEPROM memory space, and therefore the rest are ignored. Finally, to the valid request signal for the instruction cache, *icache_req_valid_o*, we add the condition that the access does not correspond to the bootrom.

Figure 5.7a shows the pre-integration logic related to the responses. The third and fourth bits of the last instruction cache access address are used to choose the word from the 128-bit block of the cache response that the *icache_interfac*e module must propagate to the Fetch 2 stage. Note that an instruction page fault exception is also a valid response.

The changes made during integration are shown in Figure 5.7b. The modifications consist of adding three multiplexers, one for each of the original output ports. The multiplexers propagate to the Fetch 2 stage, either the bootrom controller response or the one coming from the instruction cache, depending on whether the last request access was a bootrom access or not.

```
module top_drac(
...
  input   logic                 brom_ready_i        ,
  input   logic [31:0]          brom_resp_data_i    ,
  input   logic                 brom_resp_valid_i   ,
  output  logic [23:0]          brom_req_address_o  ,
  output  logic                 brom_req_valid_o    ,
  input  logic                  csr_spi_config_i ,
  input  logic                  en_translation_i);
  ...
  icache_interface icache_interface_inst(
     ...
     .en_translation_i            ( en_translation_i ),
     .csr_spi_config_i            ( csr_spi_config_i  ),
     // Inputs Bootrom
     .brom_ready_i            ( brom_ready_i      ),
     .brom_resp_data_i        ( brom_resp_data_i  ),
     .brom_resp_valid_i       ( brom_resp_valid_i ),
     // Outputs Bootrom
     .brom_req_address_o      ( brom_req_address_o ),
     .brom_req_valid_o        ( brom_req_valid_o   ));
  ...
```

Listing 5.2: Ports added to the top_drac module and icache_interface instance

Finally, it is necessary to establish the paths that interconnect the new inputs and outputs of the *icache_interface* module to the inputs and outputs of the *top_drac* module, the core. To do so, we will add the signals to the port definition of the Sargantana core module and modify the instance of the *icache_interface* module to interconnect them, as shown in Listing 5.2.

5.4.2 DRAC SoC modifications

Figure 5.8 shows the diagram of the most relevant modules for the implementation with their respective names. The dashed lines represent those elements that do not exist in the original design. The gray boxes are part of the Lowrisc chip project, which allows generating SoCs from a modular design written in Chisel. However, this language supports black-boxing functionality, which allows defining the interface of a module but provides its implementation using another language. Thus, the *DRAC* and *csr_bsc* modules are black boxes, and SystemVerilog modules give their implementation: *top_drac* module, the Sargantana core mentioned above; and *csr_bsc* module, respectively.

The *TOP* module corresponds to the processor, while the *RocketTile* module corresponds to the tile: the core, the L1 caches, the CSRs, etc. The design is scalable and allows instantiating multiple tiles. A Processor Control Register (PCR) is a subset of CSRs shared among all the processor's cores and they are managed by the *PCRcontrol* module. Note that the CSR_SPI_CONFIG in charge of choosing the boot mechanism is actually a PCR since it is a configuration value of the SoC and not of a specific core. When the core tries to access a CSR, it sends the request to the *csr_bsc* module. In case that the access corresponds to a PCR, the request is redirected to the *PCRcontrol* module.

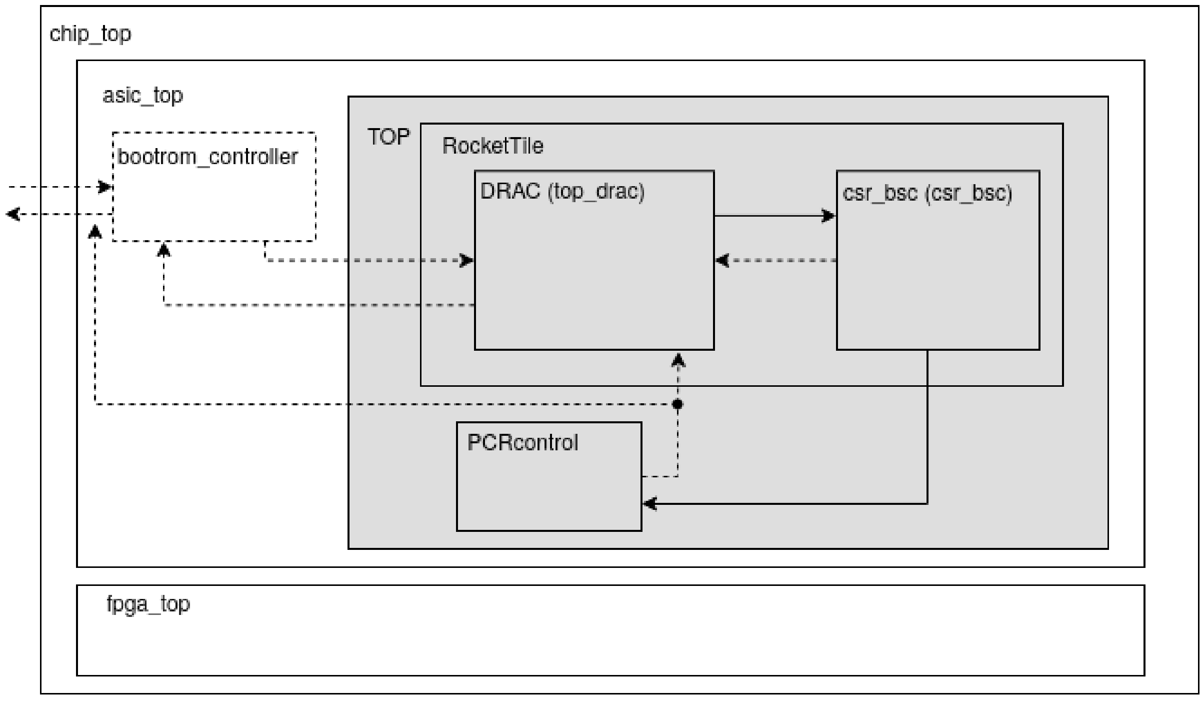


Figure 5.8: Modules related to the DRAC SoC bootrom controller integration

Next, we will detail the necessary changes to implement the bootrom controller module by module.

The *DRAC* module only requires changing the black box interface to match the changes made in the previous section to the *top_drac* module port definition. To do this, we will create an interface for the bootrom signals to reuse it in the interfaces of the *RocketTile* and *TOP* modules, which must also propagate those signals. Listing 5.3 shows this interface in Chisel.

```
class BootromReq extends CoreBundle {
  val address = Bits(width = 24)
}
class BootromResp extends CoreBundle {
  val data = Bits(width = 32)
}
class BootromIO extends Bundle{
  val req = Decoupled(new BootromReq)
  val resp = Valid(new BootromResp).flip
}

```

Listing 5.3: BootromIO interface Chisel definition

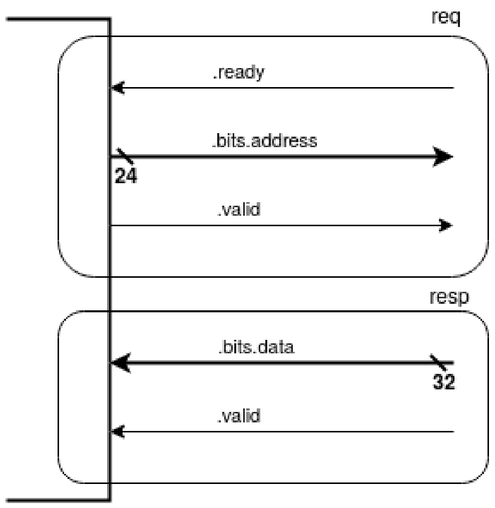


Figure 5.9: BootromIO interface diagram

Figure 5.9 shows a diagram of the resulting *BootromIO* interface. In addition, as shown in Listing 5.4, it is necessary to add the *en_translation* and *csr_spi_config* signals to the *DRAC* module interface. As this is a black box module, we define the corresponding names of the *top_drac* module for each of the added signals.

```
  class Drac (id:Int, resetSignal:Bool = null) extends BlackBox
  {
    val io = new Bundle {
    ...
      val brom = new BootromIO
      val en_translation = Bool(INPUT)
      val csr_spi_config = Bool(INPUT)

    }
    io.brom.resp.bits.data.setName("brom_resp_data_i")
    io.brom.resp.valid.setName("brom_resp_valid_i")
    io.brom.req.ready.setName("brom_ready_i")
    io.brom.req.bits.address.setName("brom_req_address_o")
    io.brom.req.valid.setName("brom_req_valid_o")
    io.en_translation.setName("en_translation_i")
    io.csr_spi_config.setName("csr_spi_config_i")
    ...
     moduleName = "top_drac"
}
```

Listing 5.4: Changes made to the Chisel definition of DRAC black box module

The *PCRcontrol* module, as shown in Figure 5.10, incorporates a new register called *reg_spi_config* corresponding to the CSR_SPI_CONFIG PCR. The CSR address reserved to access this register is 0x7F2. The lowest bit of the register is connected to the new *io.spi_config* output of the module.

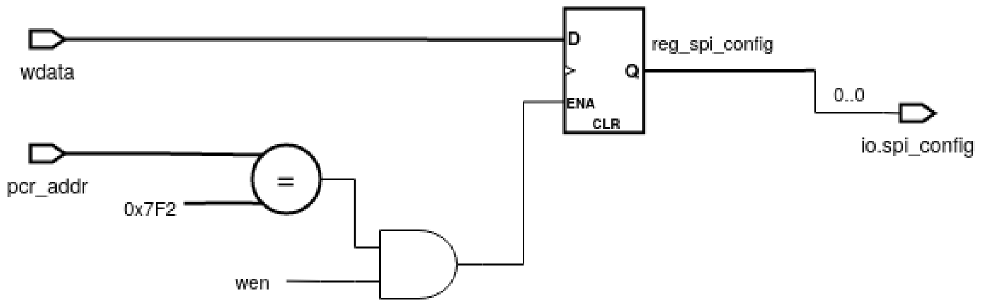


Figure 5.10: PCRcontrol module implementation modifications

The *RocketTile* module incorporates the *bootromIO* interface that interconnects to the *DRAC* module and an input for the *CSR_SPI_CONFIG* value obtained from the *PCRcontrol*, which interfaces with the corresponding input of the *DRAC* module. It also interfaces the output of the *csr_bsc* module that carries the information whether translations are enabled or not with the corresponding input of the *DRAC* module.

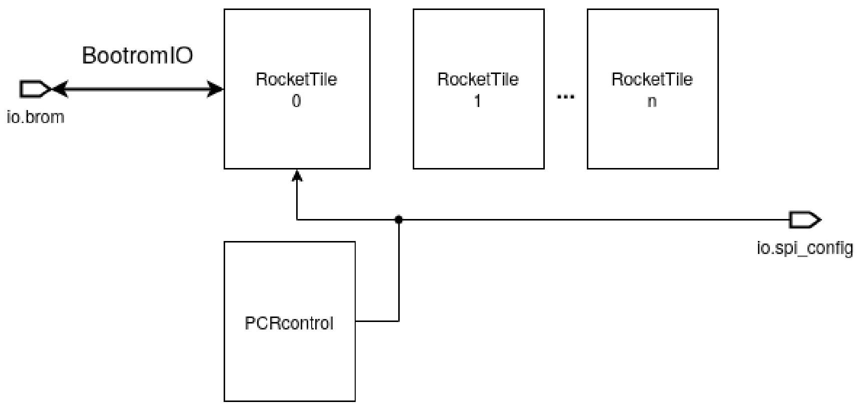
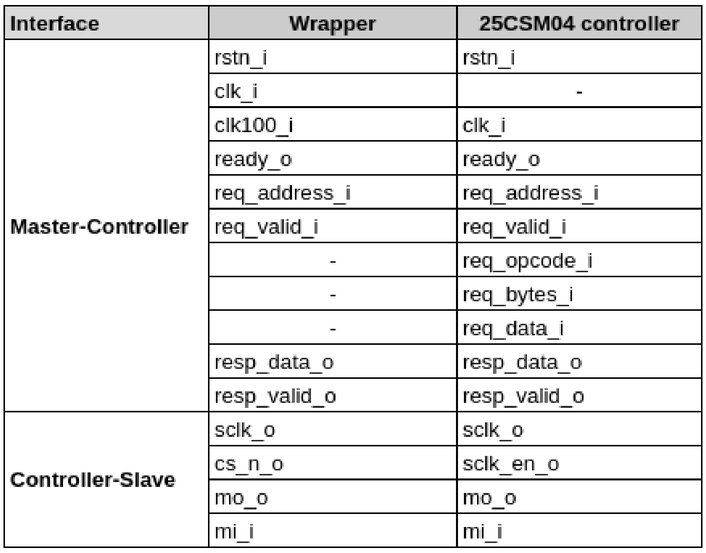


Figure 5.11: TOP module implementation modifications

The *TOP* module instantiates one or more tiles. However, the bootloader execution and SoC setup is a single-core task. We add the *BootromIO* interface and connect it only to the first instantiated tile, as shown in Figure 5.11. In the same way, the information contained in the PCR in charge of choosing the boot mechanism will only need to be connected to this tile. As we will see later, this information will also be required in the *asic_top* module, so we will create a new output on the TOP module interface to propagate this signal outwards.

The *asic_top* module is the module where all ASIC-oriented SoC components are instantiated. It is here where we instantiate the bootrom controller. However, we implement a wrapper to simplify its interface. The wrapper implementation is shown in Appendix A.2. The primary purpose of the wrapper module is to accumulate four bytes of responses from the bootrom controller (see Section3.2.1.1) to obtain the complete instruction. Table 5.2 shows the comparison of the wrapper and bootrom controller interfaces. Note that the Master-Controller interface of the wrapper, ignoring the clock and reset signals, is essentially the *BootromIO* interface shown in Figure 5.9.



Interface	Wrapper	25CSM04 controller
Master-Controller	rstn_i	rstn_i
	clk_i	-
	clk100_i	clk_i
	ready_o	ready_o
	req_address_i	req_address_i
	req_valid_i	req_valid_i
	-	req_opcode_i
	-	req_bytes_i
	-	req_data_i
	resp_data_o	resp_data_o
	resp_valid_o	resp_valid_o
Controller-Slave	sclk_o	sclk_o
	cs_n_o	sclk_en_o
	mo_o	mo_o
	mi_i	mi_i

Table 5.2: Comparison of wrapper and bootrom controller interfaces.

Figure 5.12 shows a diagram with the changes made in the design of the *top_asic* module. The *CSU* block is the Clock Selector Unit and offers clocks at different frequencies. We use the 100 MHz clock for the bootrom controller. We instantiate the controller with the CLK_DIV_FACTOR parameter value set to 32. Therefore the resulting serial clock *sclk_o* is 3.3MHz.

It is important to note that the SoC used the SPI ports to communicate with the SD card, where the system that the bootloader must load to the main memory of the processor is stored. We used the PCR value *csr_spi_config* to choose between the bootrom controller and SD card controller outputs. Note that the initialization value of the PCR is 0, so the SoC will be able to communicate with the bootrom after a reset. However, this implies that the bootloader will have to set this PCR before accessing the SD card, thus losing access to the bootrom.The only port added to the pinout of the chip is the *spi_cs_brom* output, which will allow us to assert the 25CSM04 EEPROM.

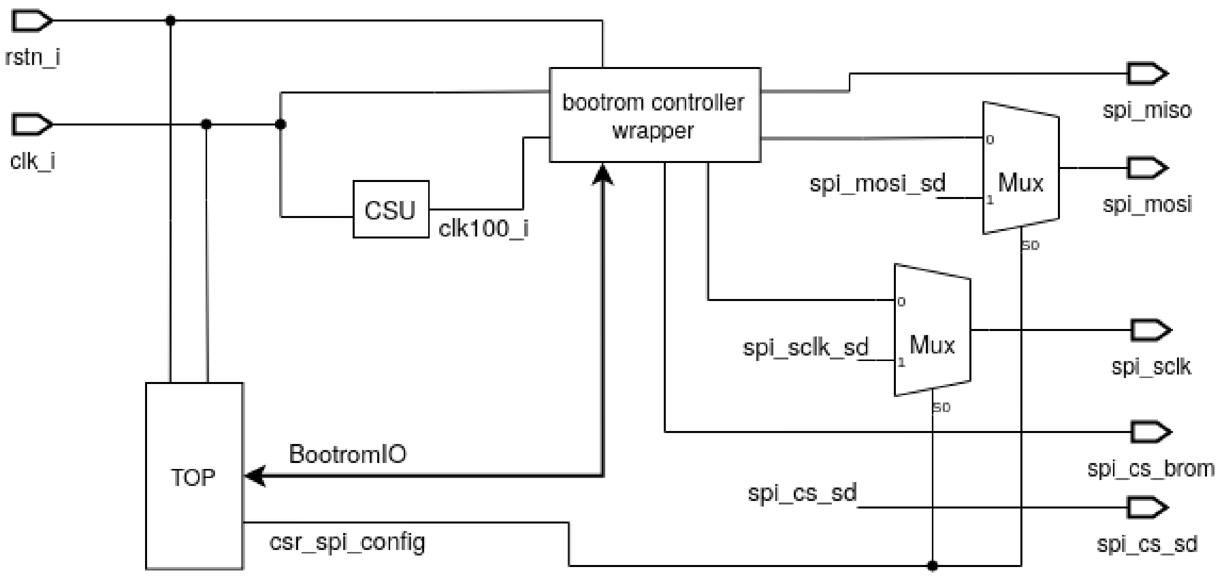


Figure 5.12: Diagram with the changes in the *asic_top* module implementation

5.5 Boot process modifications

The new bootrom, unlike the one used by the predecessor PreDRAC, is fetch-only. This forces us to modify the actual boot process, explained in section 2.3.1. We will need a loader to load the bootloader into the processor's memory hierarchy, thus allowing us to maintain read access to the bootloader. Figure 5.13 shows a diagram of the new boot paradigm.

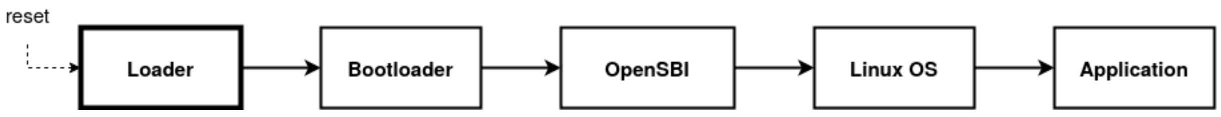


Figure 5.13: DRAC boot process

The loader code writes the bootloader to the end of the processor's main memory, the DRAM (see Table 2.2). Note that we use the term *writes* and not *copies* because there is no read paths to the bootrom. Therefore, the loader can only load the bootloader to the DRAM combining the *li* (load immediate) and *sw* (store word) RISC-V instructions. It is also important to note that in order to write the bootloader, the bootloader needs to have it in its code.

Listing 5.5 shows an example of loader code. In it, we can see that the second parameter of the *li* instructions is, in fact, the hexadecimal encoding of a bootloader instruction. Appendix B.1 shows a script to generate the loader code from the bootloader hex dump automatically.

```
.align 6
.globl _start
_start:
.equ WR_ADDR, 0xbff00000
li t5, WR_ADDR

```

```
# Starting the write
li t0, 0x00000093
sw t0, 0(t5)
li t0, 0x00000113
sw t0, 4(t5)
...
li t0, 0x00000000
sw t0, 140(t5)
# Write finished, jump to bootloader
li t5, WR_ADDR
jr t5
```

Listing 5.5: Loader sample code

Figure 5.14 shows a simplified sequence diagram of the new boot process. The white rectangles above the vertical lines represent the code execution flow.Apart from adding the loader, the bootloader needs to be modified to add the instruction `csrwi 0x7f2, 1`. This instruction writes a 1 to the *csr_spi_config* PCR (see Section 5.4.2), enabling access to the SD card and thus allowing the bootloader to copy the OpenSBI and Linux kernel.

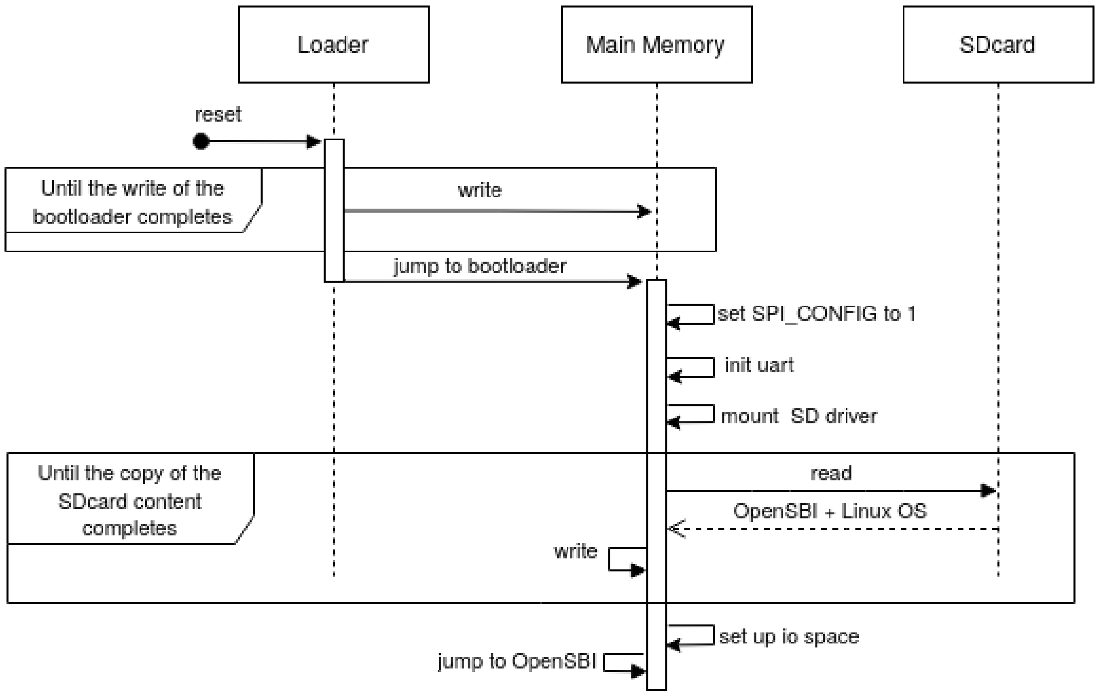


Figure 5.14: Simplified DRAC boot sequence diagram

6 Final verification

This chapter details the different verification process stages of the bootrom integration into the SoC. First, we will discuss the modifications made to the SoC design. Then, we will simulate the chip with the bootrom controller and the 25CSM04 simulation model, and finally, we will emulate it with an FPGA with a 25CSM04 EEPROM device connected to it.

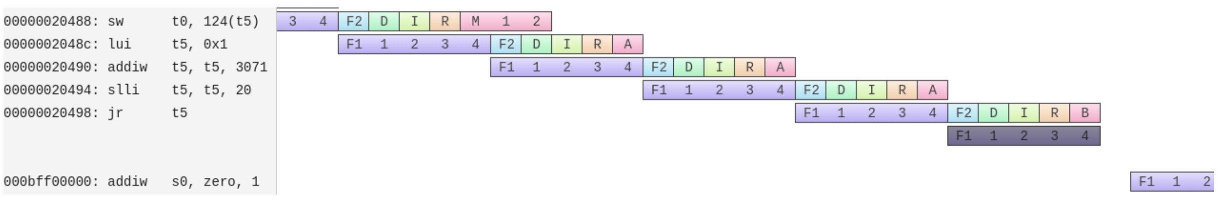
6.1 SoC design modifications verification

The objective of this verification stage is not to check the correct behavior of the SoC with the bootrom controller but to ensure that the different paths implemented during the integration process do not break the original design of the chip. For this reason, we will use a bootrom controller stub to mask possible errors introduced by the bootrom controller wrapper or even the controller itself. The stub is a perfect memory that receives the same inputs as the bootrom controller wrapper and returns the expected data. However, we have added four delay cycles that symbolize the latency of the bootrom controller. Note that the actual latency of the bootrom controller is much higher, but we have decided to reduce it to speed up the simulations.

The verification environment consists of Verilator [34], a tool for RTL simulations. This tool transforms modules written in Verilog or SystemVerilog to C++ models. This makes the user's control of design stimulation or behavior checking more flexible. To evaluate the correct operation of the SoC, we first use ISA RISC-V intensive unit tests [24]. If all tests pass, we run the EEMBC AutoBench Performance Benchmark Suite [23]. These benchmarks are more complex than the unit tests allowing us to observe errors in more specific cases. In addition, the benchmarks also allow us to evaluate whether the SoC design modifications to integrate the bootrom are in any way detrimental to the performance of the processor. Finally, we run the Linux kernel to observe that the chip can still boot Linux despite the design changes.

On the other hand, we can also visualize the processor behavior using Konata [29], which graphically displays the instruction pipeline. Thanks to the flexibility offered by the Verilator simulation environment, we can easily integrate this tool into the verification process to facilitate the analysis of the chip behavior.

Figure 6.1 shows an example of a Konata visualization of the execution of the last instructions of the loader. We can observe that the Fetch1 stage of the instructions belonging to the loader has 5-cycles latency. Note that the four extra cycles correspond to the symbolic delay added to the stub of the bootrom controller. The last instruction executed by the loader is the jump to the end of the main memory, where the bootloader is loaded: the address 0xbff00000.

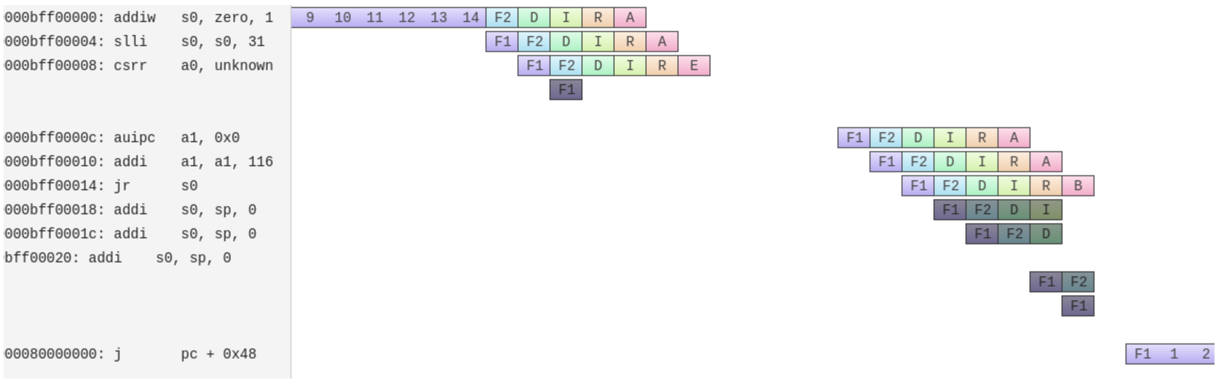


```
00000020488: sw      t0, 124(t5)
0000002048c: lui     t5, 0x1
00000020490: addiw   t5, t5, 3071
00000020494: slli    t5, t5, 20
00000020498: jr      t5

000bff00000: addiw   s0, zero, 1
```

Figure 6.1: Konata loader instructions visualization

Figure 6.2 shows the bootloader execution. We can observe that now the latency of the Fetch1 stage is generally one cycle since the bootloader instructions are fetched through the instruction cache. The Verilator verification environment loads the test or benchmark to the first DRAM address (see Table 2.2). Thus, the bootloader should simply cede execution to the already loaded test or benchmark.



```
000bff00000: addiw   s0, zero, 1
000bff00004: slli    s0, s0, 31
000bff00008: csrr    a0, unknown


000bff0000c: auipc   a1, 0x0
000bff00010: addi    a1, a1, 116
000bff00014: jr      s0
000bff00018: addi    s0, sp, 0
000bff0001c: addi    s0, sp, 0
bff00020: addi    s0, sp, 0



00080000000: j       pc + 0x48
```

Figure 6.2: Konata bootloader instructions visualization

Figure 6.3 shows the complete picture of a test run. We have added markers to identify the different steps of the boot process. On the left, identified by marker 1, is the execution of the loader. Marker 2, in the middle, is the bootloader execution. Finally, marker 3 is the execution of the test.

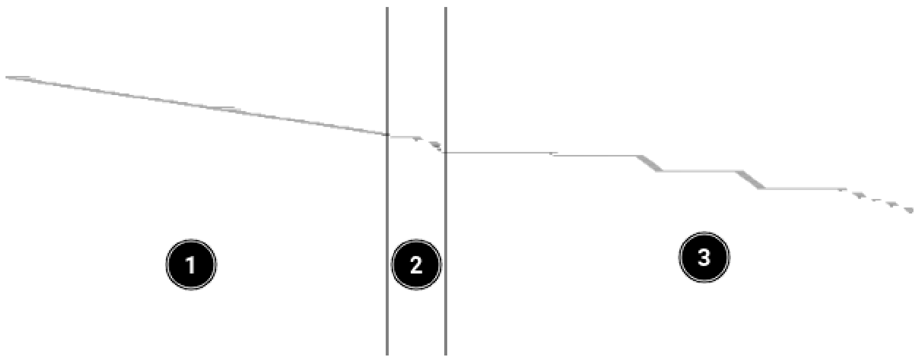


Figure 6.3: Konata zoomed-out visualization

Note that this representation gives us information about the chip's performance: the more vertical the line is, the higher the number of instructions per cycle; a relatively horizontal line indicates that the instruction loses cycles at some stage during its execution. Comparing the

Konata execution images with the SoC version before and after the bootrom integration lets us quickly see if we have introduced a performance-affecting change in the design.

6.1.1 Fallback option verification

We have written a specific test to check that the secondary boot mechanism works in case of bootrom controller failure. In Appendix C.1, you will find the test source code. The test changes the boot mechanism four times, writing a one or a zero alternatively to the PCR *csr_spi_config*. After each boot mechanism change, the test prints the contents of an array of characters at a given offset. When compiling the test, we generate two completely identical binaries except for one thing: the offset value is different.

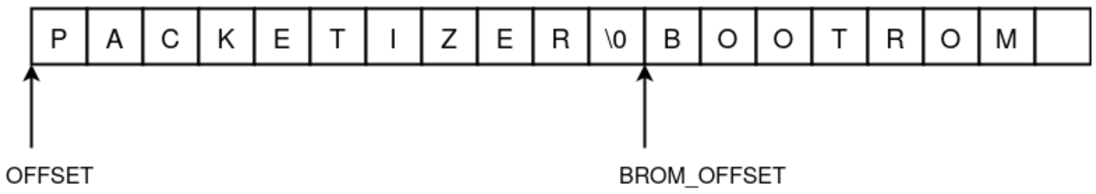
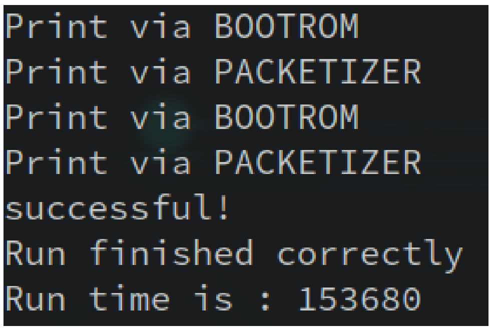


Figure 6.4: Test buffer with its two possible print offsets

One of the two almost identical binaries will be loaded to the bootrom and the other to the FPGA memory. Referring to Figure 6.4, the offset value in the bootrom binary is BROM_OFFSET, while in the binary loaded to the FPGA memory is OFFSET. In this way, when the PCR *csr_spi_config* is 0, the primary boot mechanism is active, the instructions will be fetched from the bootrom, and when printing the array of characters from the offset, we will be able to see the word *BOOTROM*. On the other hand, when the test sets the PCR *csr_spi_config* to 1, the fallback boot mechanism will be activated, obtaining the instructions through the packetizer, thus seeing the word *PACKETIZER* as a result of the print.



```
Print via BOOTROM
Print via PACKETIZER
Print via BOOTROM
Print via PACKETIZER
successful!
Run finished correctly
Run time is : 153680
```

Figure 6.5: Fallback boot test output

Figure 6.5 shows the test execution output using the Verilator verification environment, and Figure 6.6 the Konata trace of the same execution. In the Konata trace, we have identified the different boot mechanisms activated at each moment, easily recognizable by the slope of the instruction execution flow. Observing both figures, we can corroborate that the PCR that manages the selection of the boot mechanism works correctly, allowing the user to choose whether to fetch from the bootrom or the memory instantiated in the FPGA.

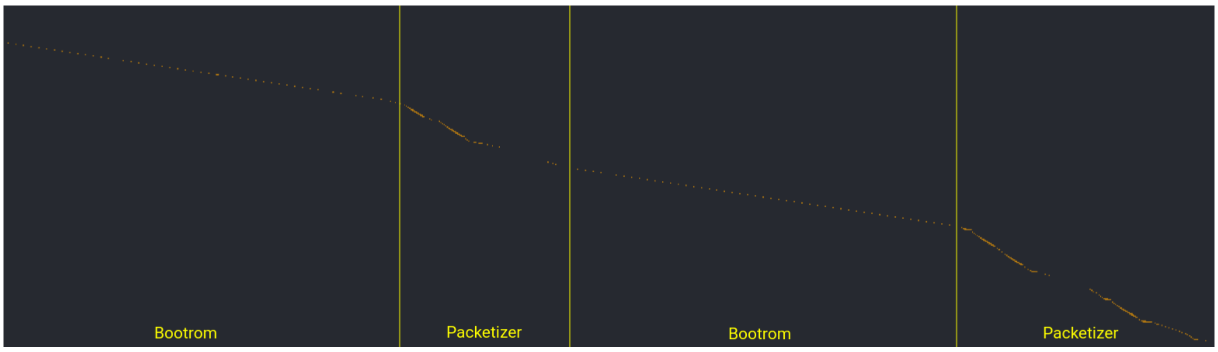
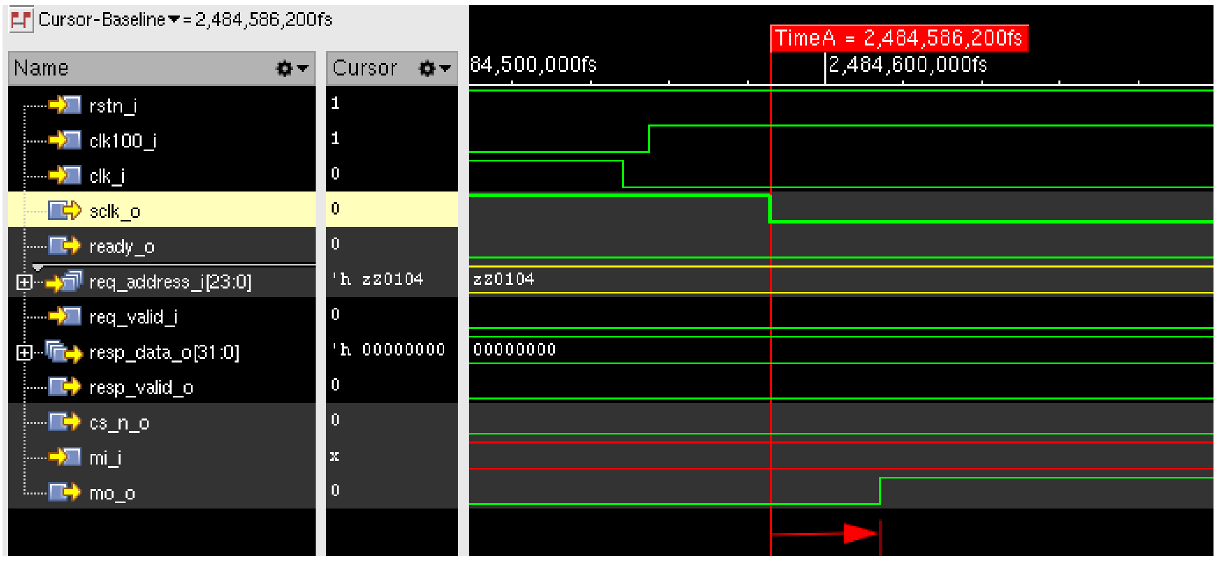


Figure 6.6: Selected boot mechanism in Konata trace of the fallback boot test execution

6.2 Bootrom controller integration verification

So far, we have only performed RTL simulations on the design. RTL simulations create a model of the RTL design and calculate cycle by cycle the value of each signal in the model. However, they are not able to provide information at the subcycle level. The objective of this verification stage is to perform a Gate-Level Simulation (GLS) on the new SoC design with the bootrom controller. These simulations are very accurate, as they include information about the technology node to be used for manufacturing. With this information, we can observe delays at the logic gate level, find glitches, propagations of X values (indeterminate values, for example, due to an uninitialized register), or timing constraint violations.

We execute the same tests and benchmarks again as in the previous stage using this verification environment. Figure 6.7 shows the waveform of a GLS of the new SoC design with the signals from the bootrom controller wrapper interface. Note that, although the output signal *mo_o* is clocked out on falling edges of the serial clock *sclk_o* (See Listing 3.3), there is a delay of approximately 0.02 ns from the falling edge of the clock until the signal is updated.



Name	Cursor
rstn_i	1
clk100_i	1
clk_i	0
sclk_o	0
ready_o	0
req_address_i[23:0]	'h zz0104
req_valid_i	0
resp_data_o[31:0]	'h 00000000
resp_valid_o	0
cs_n_o	0
mi_i	x
mo_o	0

Figure 6.7: Bootrom controller GLS waveform

6.3 FPGA emulation

This verification stage aims to check that the chip can boot through the bootrom. To do this, we will connect the EEPROM 25CSM04 to an FPGA, where we will emulate the new SoC design. However, to create this verification environment, it is necessary to perform some previous steps, such as programming the EEPROM or connecting it to the I/O ports of the FPGA.

6.3.1 Environment setup

Next, we will detail all the material used during this verification stage. We will use the Xilinx Kintex kc705 FPGA [15]. Figure 6.8 shows all the components included in the board. During the verification process, we will use only the following elements:

- The FPGA Mezzanine Card (FMC) connectors, identified by number 30 (HPC) and 31 (LPC), allow us to connect the FPGA with peripherals.
- The push-button in the south of the button group identified by number 23 acts as the reset.
- The LEDs identified by number 22 allow us to obtain debugging information.

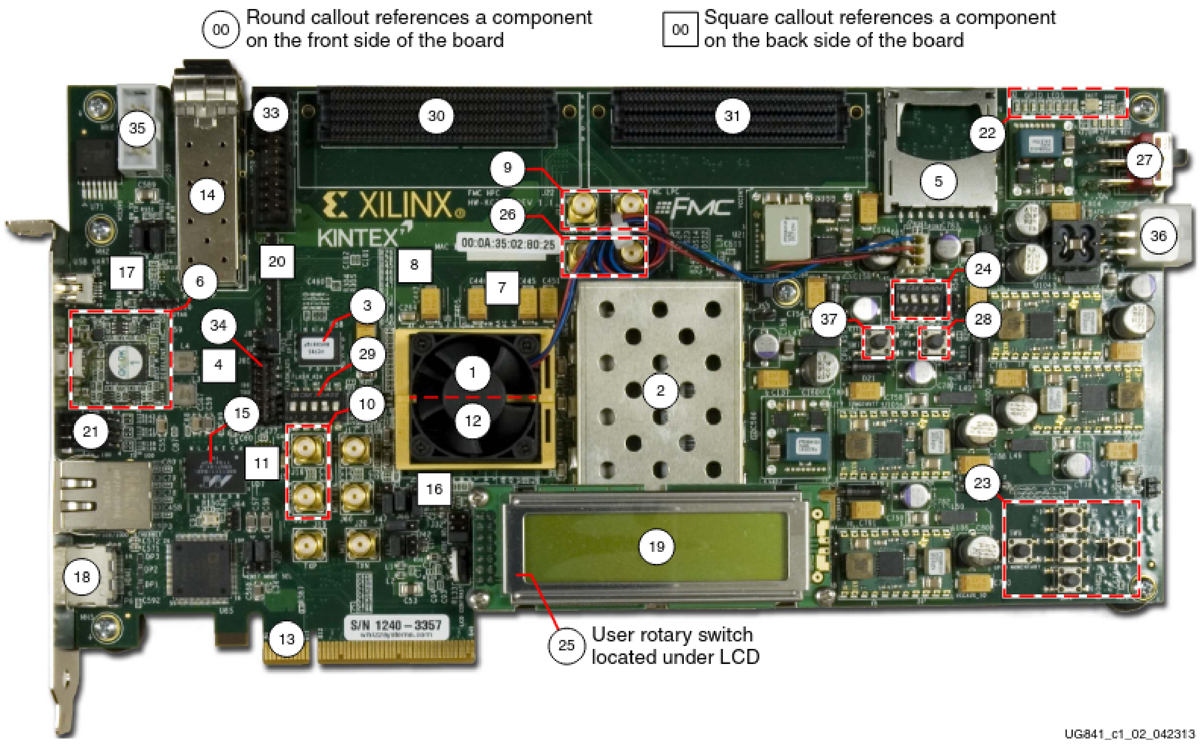
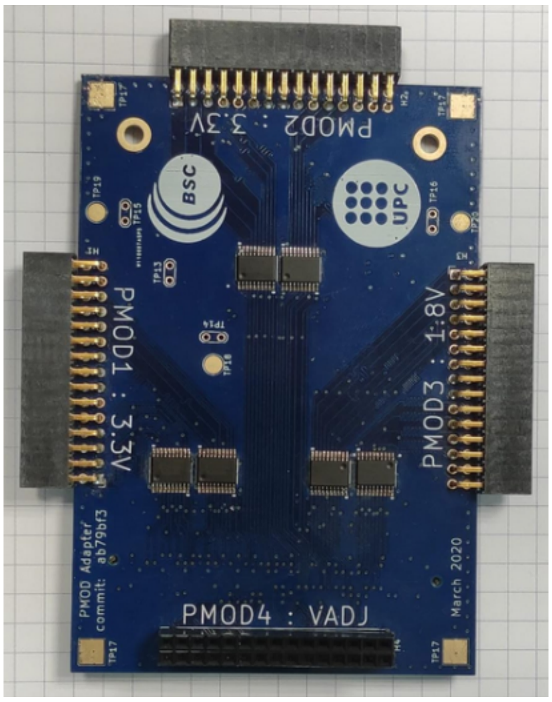
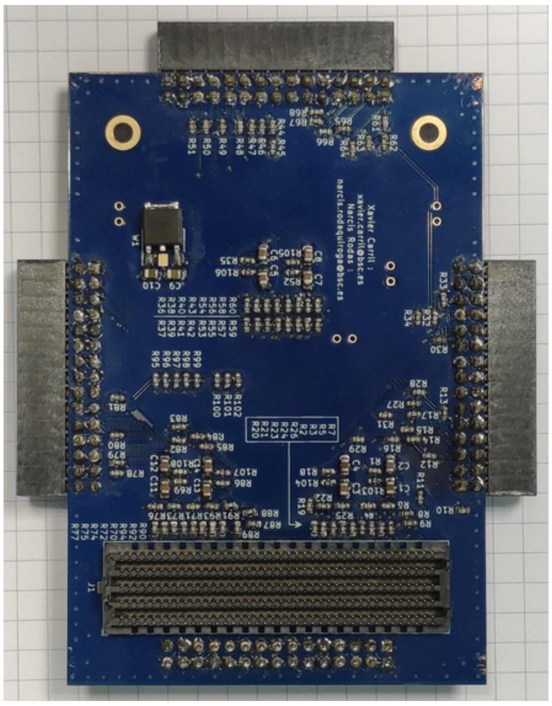

00 Round callout references a component on the front side of the board
00 Square callout references a component on the back side of the board
35
33
30
31
22
27
14
9
5
26
36
17
20
8
7
24
6
3
37
28
34
29
4
1
2
10
15
12
21
11
16
23
19
18
13
25 User rotary switch located under LCD
UG841_c1_02_042313

Figure 6.8: KC705 Board Components. Source: [15]

The two FMC connections are different. The FPC-HPC (High Pin Count) connection contains 400 pins, while the FMC-LPC (Low Pin Count) connection contains only 160 pins. We do not need many pins to connect peripherals, so we will reserve the FPC-HPC connection to connect the FPGA to another. This way, we can emulate the ASIC-oriented SoC design on one of them and the FPGA-oriented design on the other, creating an environment very similar to the real one once we have the SoC manufactured. Then, we will use the FMC-LPC connection to connect the rest of the peripherals, such as the EEPROM 25CSM04 in our case.



(a) PMOD to FMC adapter PCB front view



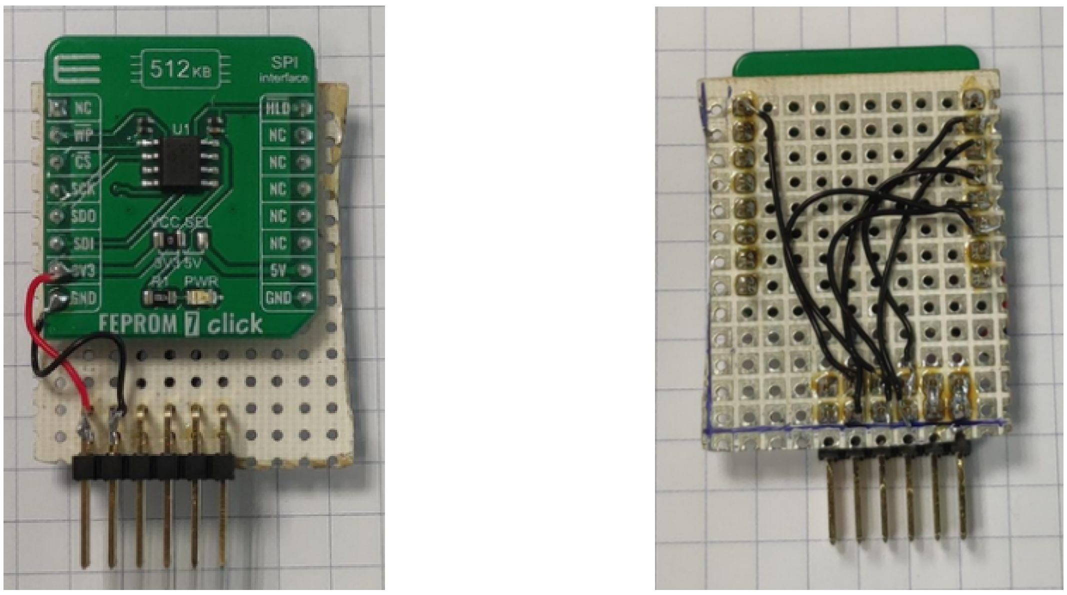
(b) PMOD to FMC adapter PCB back view.

Figure 6.9: PMOD to FMC adapter PCB

However, to connect the EEPROM, we will use an FMC to PMOD adapter developed in previous stages of the DRAC project [8, 27]. This adapter, shown in Figure 6.9, decouples the 160 pins of the FMC-LPC connection into four 2x15 PMOD sockets at different voltages.

To connect the bootrom to the adapter, we used the 25CSM04 EEPROM Platform Evaluation Expansion Board from MikroElektronika [22] soldered to a PMOD connector, as shown in Figure 6.10. We connect the bootrom to the PMOD2 socket of the adapter, powered at 3.3V since the voltage range supported by the bootrom is 2.5V to 5V.

Finally, we used the Saleae logic analyzer [28] to monitor the voltage levels received on each sampled pin, allowing us to verify its behaviour. Figure 6.11 shows the final assembly of all the elements involved in the verification process using the FPGA. Note that we have connected the MISO, MOSI, SCLK and CS_N pins of the EEPROM to the logic analyzer, the black box in Figure 6.11. This will allow us to capture 3 GB of SPI transactions at a frequency of up to 50 MS/s.



(a) 25CSM04 devkit front view (b) 25CSM04 devkit back view

Figure 6.10: 25CSM04 devkit soldered to a PMOD connector

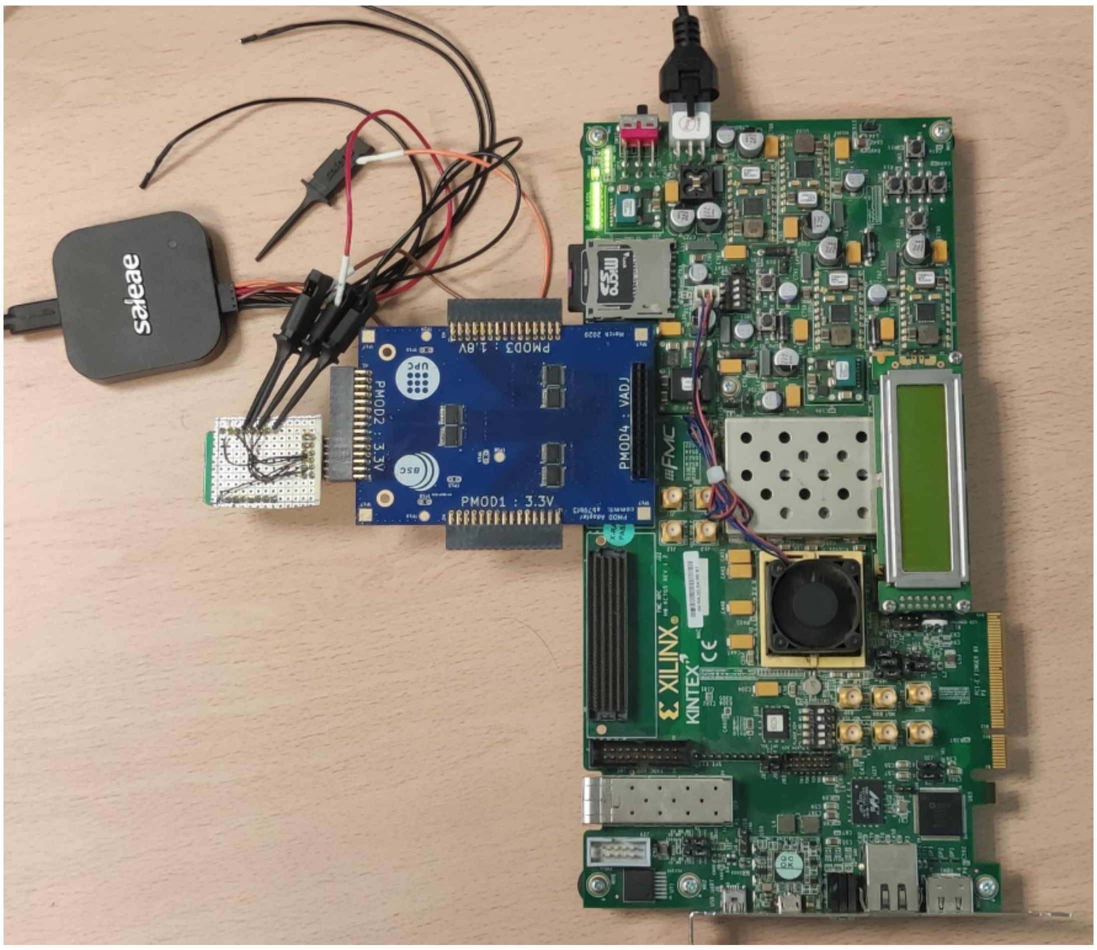


Figure 6.11: FPGA verification environment hardware setup

6.3.2 Program the 25CSM04 EEPROM

To emulate boot sequences using the environment described above, first, we need to write the loader to the EEPROM. To do this, we will create a module that writes page by page (256 bytes, the maximum write size supported by the 25CSM04 memory) the contents of the loader binary to the EEPROM. This module will instantiate the bootrom controller detailed in Section 3.2.1.1 and the state machine shown in Figure 6.12.

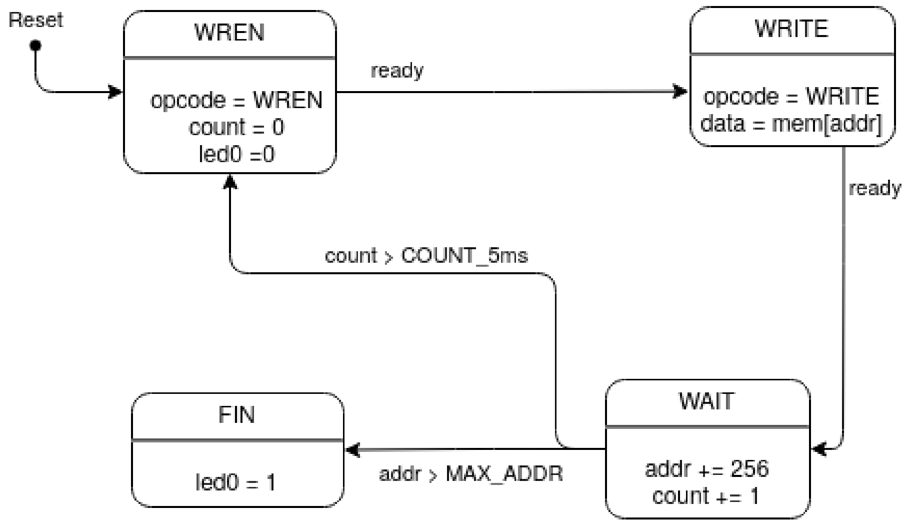


Figure 6.12: EEPROM programmer module's simplified state machine

Remember that it is necessary to enable writes with the WREN operation before performing any write on the device. It is important to note that the internal write cycle can take up to 5 ms. Any writes performed before completing the previous internal write cycle will be ignored. After each write, we reach the WAIT state, which increments a cycle counter and the address. The FPGA clock frequency is 200MHz. We can control the time between two write operations knowing de cycle period and the cycle counter value. After 5 ms, when it is sure that the last write has been propagated, the process is repeated if there is still data to write. Otherwise, we reach the FIN state. This state turns on an FPGA LED to inform the completion of the EEPROM programming.

Figure 6.13 shows a Saleae capture performed during the writing of the second EEPROM page using the module described above. From top to bottom, we can observe the logic levels of the SCLK, CS_N, MOSI and MISO signals. For each byte of the SPI transactions, there is a label above the data lines with the values transmitted. Thus we can see that, following the flow defined by the state machine in Figure 6.12, the byte transmitted in the first transaction values 0x06, which corresponds to the WREN opcode (see Table 3.2). The next SPI transaction starts with the value 0x02, corresponding to the WRITE opcode, then the three following bytes corresponds to the write address, followed by the 256 bytes to be written, starting with the lowest address.

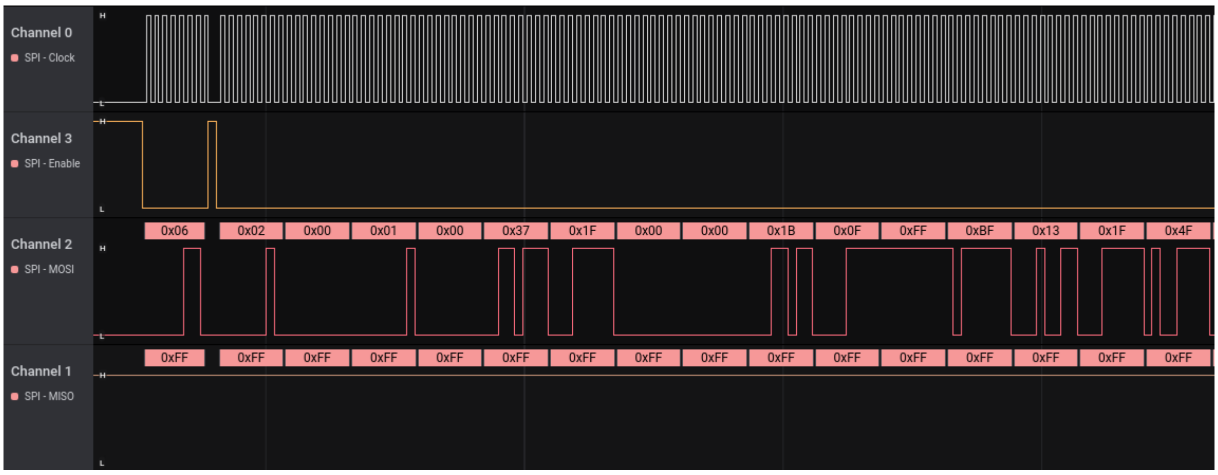
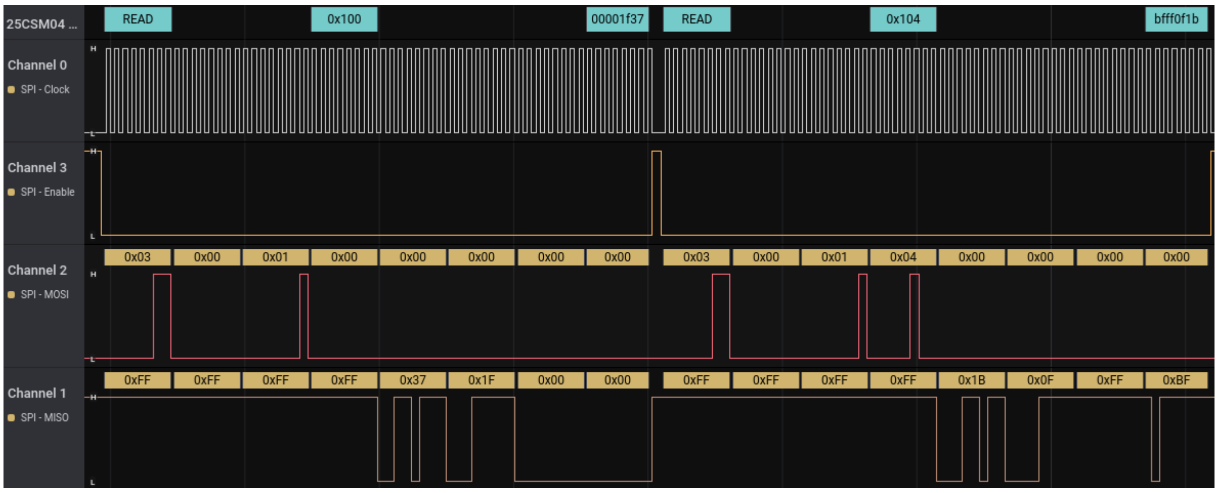
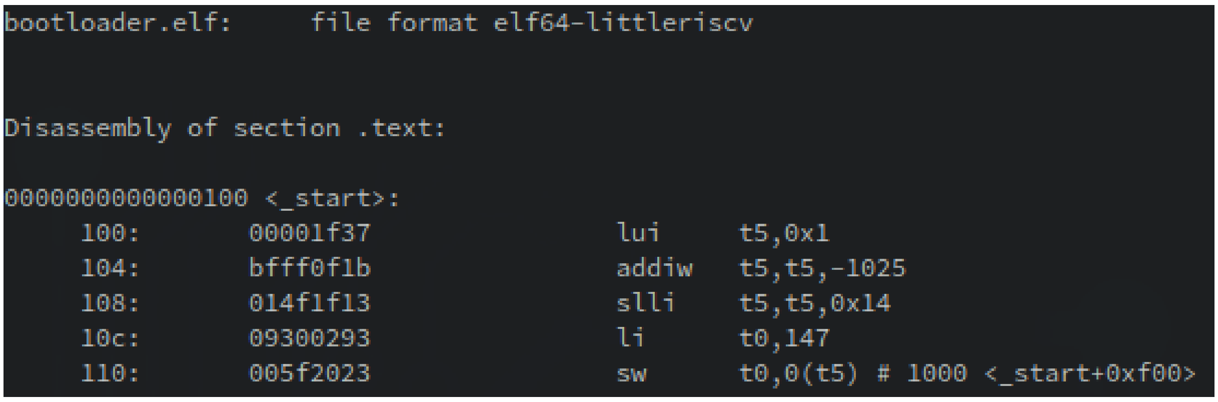


Figure 6.13: Saleae capture of the EPEPROM pins during the second page programming



(a) Saleae capture of the EPEPROM pins during the boot process



```
bootloader.elf:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000100 <_start>:
     100:       00001f37                lui     t5,0x1
     104:       bfff0f1b                addiw   t5,t5,-1025
     108:       014f1f13                slli    t5,t5,0x14
     10c:       09300293                li      t0,147
     110:       005f2023                sw      t0,0(t5) # 1000 <_start+0xf00>
```

(b) Loader ELF binary dump

Figure 6.14: Comparison of the instructions read during boot and those present in the original binary of the loader

Figure 6.14a shows a Saleae capture during the boot process. In this case, we have synthesized the SoC design for the FPGA and captured the logic levels on the EEPROM pins when triggering the reset. We have added labels at the top of the capture indicating each of the elements involved in the transaction, from left to right: the opcode, the access address (PC), and the 32-bit instruction obtained.

Figure 6.14b shows a dump of the loader executable. The first column corresponds to the PC, the second to the instruction encoding in hexadecimal and the third to the decoded instruction. We can observe that the instructions observed in the capture of Figure 6.14a correspond to the one that appears in the dump of the executable.

7 Conclusions

The main objective of this thesis was to implement an ASIC-oriented boot mechanism that would allow the *DRAC 22 nm* System on Chip (SoC) to boot standalone without requiring the originally used Field-Programmable Gate Array (FPGA). We consider that the objective has been successfully achieved, considering the constraints of using the minimum number of pins available on the chip, reaching the tape-out on time, and maintaining the ability to boot the Linux operating system.

In order to carry out this work, it has been necessary to comprehend the design of the DRAC SoC and the Sargantana RISC-V core to choose the right strategy for the integration. This process has been valuable not only for acquiring knowledge in the field of computer architecture and Register-Transfer Level (RTL) design but also for learning about the international semiconductor research scene and the different stages existing in the processor manufacturing process.

During this thesis, we have evaluated the different parameters to be taken into account to choose the most suitable bootrom for the project: the SPI 25CSM04 EEPROM. We have designed, implemented and verified a parameterizable SPI controller able to perform read and write operations. We have implemented a testbench to verify the controller by performing RTL simulations. Then, we have integrated the controller into the SoC design. The strategy followed for the integration has been to create a dedicated path from the front-end of the Sargantana core to the bootrom controller instantiated in the ASIC design of the SoC. This decision has consequences: there is no longer a data path to the bootrom; therefore, the processor will not be able to read the bootrom contents but only fetch instructions. This new paradigm forces us to adapt the boot process adding the loader: the code in charge of writing the bootloader to the processor's main memory.

We cannot forget that this processor has an academic purpose and includes in its design several improvements, compared to its predecessor *PreDRAC*, implemented by other members of the DRAC team. For this reason, as the bootrom is a critical element to evaluate the work done by the rest of the team, we have implemented a backup boot mechanism in case of failure. This mechanism allows using the original boot mode, i.e. through an FPGA, by writing a Control and Status Register (CSR).

Finally, we have performed Gate-Level Simulations (GLS) on the new SoC design to ensure that the changes introduced behaves as expected and do not add any critical path to the design nor affect the processor's performance. Then, we emulated the SoC using the Xilinx Kintex kc705 FPGA. Currently, the final DRAC SoC design is finishing the verification process. This is an exhaustive and iterative process, especially now, close to the tape-out deadline.

7.1 Future work

The next steps in the context of this project are as follows. First of all, after the chip manufacturing, we will have to perform the board bring-up, debugging and verifying that the chip can boot Linux using the new boot process implemented in this thesis.

Secondly, we could improve the verification process on the EEPROM controller write logic. Since it is not involved in the boot process and is not triggered by the SoC, all the exhaustive simulations performed on the chip at RTL, GLS or FPGA-level, did not cover this part of the controller. Performing further verification on the write logic of the controller will avoid possible undesired behaviour in some corner cases.

Finally, for the next tape-out, it would be desirable to re-evaluate the integration strategy of the AXI Wrapper, so we could use smaller bootloaders since the processor could read the bootrom content and copy it to the main memory more efficiently. However, the changes needed to implement this strategy are not trivial. It would involve modifying the implementation of the instruction cache to allow, in some cases, the cache to request data from the AXI bus and not from the memory hierarchy.

Bibliography

[1] Jaume Abella, Calvin Bulla, Guillem Cabo, Francisco J. Cazorla, Adrián Cristal, Max Doblas, Roger Figueras, Alberto González, Carles Hernández, César Hernández, Víctor Jiménez, Leonidas Kosmidis, Vatistas Kostalabros, Rubén Langarita, Neiel Leyva, Guillem López-Paradís, Joan Marimon, Ricardo Martínez, Jonnatan Mendoza, Francesc Moll, Miquel Moretó, Julián Pavón, Cristóbal Ramírez, Marco Antonio Ramírez, Carlos Rojas Morales, Antonio Rubio, Abraham Ruiz, Nehir Sönmez, Víctor Soria, Lluís Terés, Osman S. Unsal, Mateo Valero, Iván Vargas Valdivieso, and Luis Villa. An academic RISC-V silicon implementation based on open-source components. In *XXXV Conference on Design of Circuits and Integrated Systems (DCIS), 2020, Segovia, Spain, November 18-20, 2020*, pages 1–6. IEEE, 2020. doi: 10.1109/DCIS51330.2020.9268664. URL `https://doi.org/10.1109/DCIS51330.2020.9268664`.

[2] anysilicon.com. What is a system on chip (soc)? URL `https://anysilicon.com/what-is-a-system-on-chip-soc/`.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[4] atlassian.com. Gitflow workflow — atlassian git tutorial. URL `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi: 10.1145/2228360.2228584.

[6] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York,

NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872414. URL http://doi.acm.org/10.1145/2872362.2872414.

[7] BSC. Home — DRAC project. URL https://drac.bsc.es/en/home.

[8] Xavier Carril. Design and implementation of hyperram controller ip. *UPCommons*, 2020. URL https://upcommons.upc.edu/bitstream/handle/2117/340101/152624.pdf?sequence=1&isAllowed=y.

[9] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[10] EEMBC. EEMBC CoreMark. URL https://www.eembc.org/autobench/.

[11] Max Font. Risc-v core optimization in 22nm fd-soi technology. *UPCommons*, 2021. URL https://upcommons.upc.edu/bitstream/handle/2117/359952/TFM_doblas.pdf?sequence=2&isAllowed=y.

[12] glassdor.es. Glassdoor. URL https://www.glassdoor.es.

[13] LowRISC Inc. Rocket core overview, . URL https://www.cl.cam.ac.uk/~jrrk2/docs/tagged-memory-v0.1/rocket-core/.

[14] Microchip Technology Inc. 4-mbit spi serial eeprom with 128-bit serial number and enhanced write protection, . URL https://ww1.microchip.com/downloads/en/DeviceDoc/25CSM04-4-Mbit-SPI-Serial-EEPROM-With-128-Bit-Serial-Number-and-Enhanced-Write-Protect
pdf.

[15] Microchip Technology Inc. Kc705 evaluation board for the kintex-7 fpga, . URL https://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf.

[16] Microchip Technology Inc. Verilog models, . URL https://www.microchip.com/doclisting/TechDoc.aspx?type=Verilog.

[17] Intel. Intel® fpga simulation - modelsim*-intel® fpga. URL https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.

[18] Neiel I. Leyva-Santes, Ivan Perez, César-Alejandro Hernández-Calderón, Enrique Vallejo, Miquel Moretó, Ramón Beivide, Marco Antonio Ramírez Salinas, and Luis A. Villa-Vargas. Lagarto I RISC-V multi-core: Research challenges to build and integrate a network-on-chip. In *International Conference on Supercomputing in Mexico (ISUM)*, volume 1151 of *Communications in Computer and Information Science*, pages 237–248. Springer, 2019. doi: 10.1007/978-3-030-38043-4_20. URL https://doi.org/10.1007/978-3-030-38043-4_20.

[19] Katie Lim, Jonathan Balkind, and David Wentzlaff. Juxtapiton: Enabling heterogeneous-isa research with RISC-V and SPARC FPGA soft-cores. *CoRR*, abs/1811.08091, 2018. URL http://arxiv.org/abs/1811.08091.

[20] lowRISC 64-bit SoC. lowRISC . URL https://www.lowrisc.org/.

[21] Alfio Di Mauro, Francesco Conti, Pasquale Davide Schiavone, Davide Rossi, and Luca Benini. Always-on 674u 4gop/s error resilient binary neural networks with aggressive sram voltage scaling on a 22-nm iot end-node. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(11):3905–3918, 2020. doi: 10.1109/TCSI.2020.3012576.

[22] MIKROE. Hardware and software tools for the embedded world - mikroelektronika. URL https://www.mikroe.com/eeprom-7-click.

[23] Jason Poovey, Thomas Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, 29:18 – 29, 11 2009. doi: 10.1109/MM.2009.74.

[24] riscv.org. Github - riscv-tests, . URL https://github.com/riscv-software-src/riscv-tests.

[25] riscv.org. Github - opensbi, . URL https://github.com/riscv-software-src/opensbi.

[26] riscv.org. Github - spike, . URL https://github.com/riscv-software-src/riscv-isa-sim.

[27] Narcís Rodas. Rtl design and implementation of a framebuffer for a risc-v processor. *UPCommons*, 2020. URL https://upcommons.upc.edu/bitstream/handle/2117/340101/152624.pdf?sequence=1&isAllowed=y.

[28] Saleae. Saleae usb logic analyzers. URL https://www.saleae.com/es/.

[29] Ryota Shioya. Github - konata. URL https://github.com/shioyadan/Konata.

[30] slack.com. Where work happens — slack. URL https://slack.com/.

[31] Inc. Sun Microsystems. OpenSPARC™ T1 Microarchitecture Specification. Part No. 819-6650-11, April 2008.

[32] SystemVerilog. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[33] trello.com. Boards — trello. URL https://trello.com/.

[34] Veripool. Verilator Verilog/SystemVerilog simulator. URL https://www.veripool.org/wiki/verilator.

[35] Andrew Waterman and Krste Asanović. The risc-v instruction set manualvolume ii: Privileged architecture. Technical report, SiFive Inc. and EECS Department, University of California, Berkeley, Dec 2021. URL https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[36] Wikipedia. Application-specific integrated circuit, . URL https://en.wikipedia.org/wiki/Application-specific_integrated_circuit.

[37] Wikipedia. Field-programmable gate array, . URL `https://en.wikipedia.org/wiki/Field-programmable_gate_array`.

[38] Wikipedia. Instruction set architecture, . URL `https://en.wikipedia.org/wiki/Instruction_set_architecture`.

[39] Wikipedia. Register-transfer level, . URL `https://en.wikipedia.org/wiki/Register-transfer_level`.

[40] Wikipedia. Serial peripheral interface, . URL `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface`.

[41] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2926114.

[42] Florian Zaruba, Fabian Schuiki, Stefan Mach, and Luca Benini. The floating point trinity: A multi-modal approach to extreme energy-efficiency and performance. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 767–770, 2019. doi: 10.1109/ICECS46596.2019.8964820.

[43] José Ángel Plaza López. Europa se abraza al hardware libre para librarse de su dependencia de ee uu y china. URL `https://elpais.com/retina/2020/02/18/tendencias/1582023556_258635.html?outputType=amp`.

A Bootrom controller SystemVerilog implementation

A.1 25CSM04 controller implementation

```
/*-----------------------------------------------
* Project Name   : DRAC
* File           : spi_eeprom_req.sv
* Organization   : Barcelona Supercomputing Center
* Author(s)      : Jordi Garcia
* Email(s)       : jordi.garcia@bsc.es
* -----------------------------------------------
*/

module spi_eeprom_req
  #(
  parameter CLK_DIV_FACTOR = 32,
  parameter MAX_REQ_BYTES = 4)
  (
  input logic           clk_i,             //   serial data clock
  input logic           rstn_i,            //   reset
  input logic  [7:0]    req_opcode_i,      //   request opcode
  input logic  [23:0]   req_address_i,     //   request address, for read and writes
  input logic  [(MAX_REQ_BYTES*8)-1:0] req_data_i,        //   request data
  input logic  [$clog2(MAX_REQ_BYTES):0]    req_bytes_i,       //   request data size in bytes - max 256
  input logic           req_valid_i,       //   request inputs valid state
  output logic          mo_o,              //   SPI master output
  output logic          ready_o,           //   eeprom ready
  output logic          sclk_o,            //   spi clock - mode 0
  output logic          sclk_en_o,         //   spi clock - mode 0
  output logic [7:0]    resp_data_o,       //   response data bus
  output logic          resp_valid_o,      //   response valid
  input  logic          mi_i);             //   SPI master input

/////////////////////////////////////////////////////////////////////////
// DECLARATIONS
/////////////////////////////////////////////////////////////////////////
  localparam DIV_CLK_COUNT_LEN = $clog2(CLK_DIV_FACTOR);

```

```
typedef enum logic [2:0] {STATE_IDLE, STATE_T_OP, STATE_T_ADDR, STATE_R_DREAD,
  STATE_T_DWRITE} e_state;

e_state state;// = STATE_IDLE;
e_state next_state;// = STATE_IDLE;

logic cnt_finish;
logic is_idle;
logic is_opcode;
logic is_addr;
logic is_read;
logic is_write;
logic [9:0] max_cnt_byte;

logic [7:0] req_opcode_q;
logic [0:2][7:0] req_address_q;
logic [8:0] req_bytes_q;
logic [MAX_REQ_BYTES-1:0][7:0] req_data_q;
logic [2:0] bitcount_q;
logic [2:0] bitcount_d;
logic [8:0] bytecount_q;
logic [8:0] bytecount_d;
logic       resp_valid_d;
logic       resp_valid_q;
logic [7:0] resp_data_q;
logic [DIV_CLK_COUNT_LEN-1:0] clk_div_cnt;
logic clk;
logic [7:0] dataShifterO;
logic [7:0] dataShifterI;

/* 25CSM04 EEPROM opcodes */
`define OP_RDSR  8'b0000_0101 // Read Status Register instruction
`define OP_WRBP  8'b0000_1000 // Write Ready/Busy Poll instruction
`define OP_WREN  8'b0000_0110 // Set Write Enable Latch instruction
`define OP_WRDI  8'b0000_0100 // Reset Write Enable Latch instruction
`define OP_WRSR  8'b0000_0001 // Write Status Register instruction
`define OP_READ  8'b0000_0011 // Read EEPROM Array instruction
`define OP_WRITE 8'b0000_0010 // Write EEPROM Array instruction
`define OP_RDEX  8'b1000_0011 // Read Security Register instruction
`define OP_WREX  8'b1000_0010 // Write Security Register instruction
`define OP_LOCK  8'b1000_0010 // Lock Security Register instruction
`define OP_CHLK  8'b1000_0011 // Check Security Register Lock Status
 instruction
`define OP_RMPR  8'b0011_0001 // Read Memory Partition Registers instruction
`define OP_PRWE  8'b0000_0111 // Set MPR Write Enable Latch instruction
`define OP_PRWD  8'b0000_1010 // Reset MPR Write Enable Latch instruction
`define OP_WMPR  8'b0011_0010 // Write Memory Partition Registers instruction
`define OP_PPAB  8'b0011_0100 // Protect Partition Address Boundaries
 instruction
`define OP_FRZR  8'b0011_0111 // Freeze Memory Protection Configuration
 instruction
`define OP_SPID  8'b1001_1111 // Read Manufacturer ID instruction
`define OP_SRST  8'b0111_1100 // Software Device Reset instruction

```

```
//////////////////////////////////////////////////////////////////////////
// STATE MACHINE LOGIC
//////////////////////////////////////////////////////////////////////////
  /* next state logic */
  always_comb begin: next_state_logic
    if (~rstn_i) begin
      next_state = STATE_IDLE;
    end else begin
      case (state)
        STATE_IDLE: begin
          //$display("TIME %t:   READY: %b   VALID: %b", $time(), ready_o,
   req_valid_i);
          if (ready_o & req_valid_i) begin
            next_state = STATE_T_OP;
          end
        end
        STATE_T_OP: begin
          if (cnt_finish)   // opcode transmitted
          begin
            case (req_opcode_q)
              `OP_READ,
              `OP_WRITE : next_state = STATE_T_ADDR;
              default: next_state = STATE_IDLE;
            endcase
          end
        end
        STATE_T_ADDR: begin
          if (cnt_finish)
            case (req_opcode_q)
              `OP_READ  : next_state = STATE_R_DREAD;
              `OP_WRITE : next_state = STATE_T_DWRITE;
              default: next_state = STATE_IDLE;
            endcase
        end
        STATE_R_DREAD: begin
          if (cnt_finish)
            next_state = STATE_IDLE;
        end
        STATE_T_DWRITE: begin
          if (cnt_finish)
            next_state = STATE_IDLE;
        end
        default : next_state = STATE_IDLE;
      endcase
    end
  end

  assign cnt_finish = (bytecount_d == max_cnt_byte && bitcount_q == 7);

  /* state output logic */
  always_comb begin: state_output
    max_cnt_byte = 0;
    is_idle = 0;
    is_opcode = 0;
    is_addr = 0;
    is_read = 0;
    is_write = 0;
    case (state)
      STATE_IDLE: begin
        max_cnt_byte = 0;
        is_idle = 1;
      end
      STATE_T_OP: begin
        max_cnt_byte = 1;
        is_opcode = 1;
      end
      STATE_T_ADDR: begin
        max_cnt_byte = 3;
        is_addr = 1;
      end
      STATE_R_DREAD: begin
        max_cnt_byte = req_bytes_q;
        is_read = 1;
      end
      STATE_T_DWRITE: begin
        max_cnt_byte = req_bytes_q;
        is_write = 1;
      end
      default: ;
    endcase
  end

  /* slck enable*/
  always_ff @(negedge clk, negedge rstn_i) begin
    if(~rstn_i) begin
      sclk_en_o <= 0;
    end else begin
      sclk_en_o <= ~is_idle;
    end
  end

  /* state logic */
  always_ff @(posedge clk, negedge rstn_i) begin: state_reg
    if(~rstn_i) begin
      state <= STATE_IDLE;
    end else begin
      state <= next_state;
    end
  end
/////////////////////////////////////////////////////////////////////////
// Counters
/////////////////////////////////////////////////////////////////////////
  always_ff @(posedge clk, negedge rstn_i) begin: update_counters
    if(~rstn_i) begin
      bytecount_q <= 0;
      bitcount_q <= 0;
    end else begin
      if (next_state != state) begin
```

```
        bytecount_q <= 0;
        bitcount_q <= 0;
      end else begin
        if (bitcount_q == 7)
          bytecount_q <= bytecount_d;
        bitcount_q <= bitcount_d;
      end
    end
  end

  assign bitcount_d = bitcount_q + 1;
  assign bytecount_d = bytecount_q + 1;
//////////////////////////////////////////////////////////////////////////
// Shifters
//////////////////////////////////////////////////////////////////////////
  always_ff @(posedge clk) begin: shift_regs
    dataShifterI <= {dataShifterI[6:0], mi_i};
    if (bitcount_q == 7) begin
      resp_data_q <= {dataShifterI[6:0], mi_i};
    end
  end

  always_ff @(posedge clk_i) begin
    if ( is_idle & req_valid_i ) begin
      req_opcode_q <= req_opcode_i;
      req_address_q <= req_address_i;
      req_data_q <= req_data_i;
      req_bytes_q <= req_bytes_i;
    end
  end

  always_ff @(negedge clk) begin
    if (bitcount_q == 0) begin
      if (is_opcode) begin
          dataShifterO <= req_opcode_q;
      end else if (is_addr) begin
          dataShifterO <= req_address_q[bytecount_q];
      end else if (is_write) begin
          dataShifterO <= req_data_q[bytecount_q];
      end
    end
    else if (sclk_en_o) begin
      dataShifterO <= dataShifterO[6:0] << 1;
    end
  end
//////////////////////////////////////////////////////////////////////////
// Ready/valid logic
//////////////////////////////////////////////////////////////////////////
  assign ready_o = rstn_i & is_idle;
  assign resp_valid_d = is_read & (bitcount_q == 7);
  always_ff @(posedge clk, negedge rstn_i) begin
    if (~rstn_i)
      resp_valid_q <= 0;
    else begin
```

```
      resp_valid_q <= resp_valid_d;
    end
  end
///////////////////////////////////////////////////////////////////////////
// Clock divider
///////////////////////////////////////////////////////////////////////////
  always_ff @(posedge clk_i, negedge rstn_i) begin
    if (~rstn_i) begin
      clk_div_cnt <= '0;
    end else begin
      clk_div_cnt <= clk_div_cnt + 1;
    end
  end

  assign clk = clk_div_cnt[DIV_CLK_COUNT_LEN-1];
///////////////////////////////////////////////////////////////////////////
// Basic output assignments
///////////////////////////////////////////////////////////////////////////
  assign sclk_o = (sclk_en_o)? clk: 0;
  assign resp_valid_o = resp_valid_q;
  assign resp_data_o = resp_data_q;
  assign mo_o = dataShifter0[7];

endmodule
```

A.2 Bootrom controller wrapper implementation

```
/* ----------------------------------------------
 * Project Name   : DRAC
 * File           : bootrom_ctrl.sv
 * Organization   : Barcelona Supercomputing Center
 * Author(s)      : Jordi Garcia
 * Email(s)       : jordi.garcia@bsc.es
 * ----------------------------------------------
 */

module bootrom_ctrl (
  // Core interface
  input logic               clk_i,           //   core clock
  input logic               clk100_i,        //   100MHz clock
  input logic               rstn_i,          //   reset
  input logic   [23:0]      req_address_i,   //   request address, for read and
   writes
  input logic               req_valid_i,     //   request inputs valid state
  output logic              ready_o,         //   eeprom ready
  output logic  [31:0]      resp_data_o,     //   response data bus
  output logic              resp_valid_o,    //   response valid
  // Memory interface
  output logic              sclk_o,          //   SPI clock - mode 0
  output logic              cs_n_o,          //   SPI chip select
  output logic              mo_o,            //   SPI master output
  input  logic              mi_i);           //   SPI master input
```

```



//////////////////////////////////////////////////////////////////////////
// DECLARATIONS
//////////////////////////////////////////////////////////////////////////

  logic ready_ser;
  logic last_byte;
  logic resp_valid_byte_d;
  logic resp_valid_byte_q;
  logic resp_valid_q;
  logic resp_valid_d;
  logic [2:0] bytecount_d;
  logic [7:0] resp_data_ser;
  logic [2:0] bytecount_q;
  logic [31:0] resp_data_q;
  logic [31:0] resp_data_d;
  logic cs;
  logic [23:0] req_address_q;
  logic req_valid_q;
  logic sclk;

  `define OP_READ            8'b0000_0011                    // Read EEPROM
    Array

spi_eeprom_req ser (
  .clk_i           (clk100_i),       //  serial data clock
  .rstn_i          (rstn_i),         //  reset
  .req_opcode_i    (`OP_READ),       //  request opcode
  .req_address_i   (req_address_q),  //  request address, for read and writes
  .req_bytes_i     (9'h4),           //  request data size in bytes - max 256
  .req_data_i      (8'h0),
  .req_valid_i     (req_valid_q),    //  request inputs valid state
  .mo_o            (mo_o),           //  SPI master output
  .ready_o         (ready_ser),      //  eeprom ready
  .sclk_o          (sclk),           //  spi clock - modei 0
  .sclk_en_o       (cs),             //  spi clock enable
  .resp_data_o     (resp_data_ser),  //  response data bus
  .resp_valid_o    (resp_valid_d),   //  response valid
  .mi_i            (mi_i));          //  SPI master input

//////////////////////////////////////////////////////////////////////////
// CORE LOGIC
//////////////////////////////////////////////////////////////////////////

  always_ff @(posedge clk_i, negedge rstn_i) begin
    if(~rstn_i) begin
      req_address_q <= 0;
    end else if (req_valid_i) begin
      req_address_q <= req_address_i;
    end
  end

  always_ff @(posedge clk_i, negedge rstn_i) begin
```

```
    if(~rstn_i) begin
      req_valid_q <= 0;
    end else begin
      req_valid_q <= (req_valid_i | req_valid_q) & ready_ser;
    end
  end

  always_ff @(posedge clk_i, negedge rstn_i) begin
    if(~rstn_i) begin
      ready_o <= 1;
    end else if (req_valid_i) begin
      ready_o <= 0;
    end else if (resp_valid_byte_d & last_byte) begin
      ready_o <= 1;
    end
  end

  always_ff @(posedge clk_i, negedge rstn_i) begin
    if(~rstn_i ) begin
      bytecount_q <= 0;
    end else if (req_valid_i) begin
      bytecount_q <= 0;
    end else if (resp_valid_byte_q) begin
      bytecount_q <= bytecount_d;
    end
  end

  always_ff @(posedge clk_i, negedge rstn_i) begin
    if(~rstn_i) begin
      resp_valid_q      <= 0;
      resp_valid_byte_q <= 0;
      resp_data_q       <= 0;
    end else begin
      resp_valid_q      <= resp_valid_d ;
      resp_valid_byte_q <= resp_valid_byte_d;
      if (resp_valid_byte_d) begin
        resp_data_q <= resp_data_d;
      end
    end
  end

  assign last_byte         = bytecount_q == 3;
  assign resp_data_d       = {resp_data_ser, resp_data_q[31:8]};
  assign bytecount_d       = bytecount_q + 1;
  assign resp_valid_byte_d = resp_valid_d      & ~resp_valid_q; // positive edge
     detection
  assign resp_valid_o      = resp_valid_byte_q & last_byte;
  assign cs_n_o            = ~cs;
  assign sclk_o            = sclk;
  assign resp_data_o       = resp_data_q;

endmodule
```

B Loader generation script

B.1 Loader generation python script

```python
#! /usr/bin/python
import sys
import os
import re

def usage():
print("Usage: {} binari.hex".format(sys.argv[0]))
exit()

def exitOnError(error):
print("Error: {}".format(error))
exit()

if len(sys.argv) != 2:
usage()


COPY_HEX_FILE = sys.argv[1]
if not os.path.isfile(COPY_HEX_FILE):
exitOnError("File {} not found".format(COPY_HEX_FILE))

MAX_OFFSET=2047

with open(COPY_HEX_FILE, "r") as fd:
lines = fd.read().split("\n")

#while lines[-1] == "00000000" or lines[-1] ==  "":
#    lines.pop()

lines.append("00000000")
lines.append("00000000")
lines.append("00000000")
lines.append("00000000")

offset = 0
instr_count = 0
print("""
.align 6
.globl _start
_start:
.equ MAIN_MEM, 0x80000000
li t5, MAIN_MEM
""")
print("# Starting the copy")
for line in lines:
if len(line) == 8:
#         if instr_count > Ninstr:
#             break
instr_count+=1
print("li t0, 0x"+line)
if (offset <= MAX_OFFSET):
print("sw t0, {}(t5)".format(offset))
offset += 4
else:
print("addi t5, t5, {}".format(offset-4))
print("sw t0, 4(t5)")
offset = 8

#print("j MAIN_MEM")
print("# Copy finished: {}".format(instr_count))
print("li t5, MAIN_MEM")
print("jr t5")
```

C Verification tests

C.1 Fallback boot option test implementation

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define BROM_OFFSET 11
#define OFFSET 0

uint32_t __attribute__ ((noinline))  checkBuff(uint32_t spi_brom_csr,uint32_t offset, char* buff) {
  uint32_t ret;
  printf("Print via %s\n",buff+offset);
  if(spi_brom_csr) {
    ret = (offset == OFFSET);
  }
  else {
    ret = (offset == BROM_OFFSET);
  }
  return ret;
}

void printResult(uint32_t ok) {
if(ok) {
  printf("successful!\n");
}
  else {
    printf("FAIL!\n");
  }
}

int main(void){
  #ifdef BROM
  uint32_t offset = BROM_OFFSET;
  #else
  uint32_t offset = OFFSET;
  #endif
  uint32_t result = 1;
  char buff[20] = "PACKETIZER\0BOOTROM  ";

  __asm__("csrwi 0x7f2, 0");
```

```
    result &= checkBuff(0,offset, buff);
    __asm__("csrwi 0x7f2, 1");
    result &= checkBuff(1,offset, buff);
    __asm__("csrwi 0x7f2, 0");
    result &= checkBuff(0,offset, buff);
    __asm__("csrwi 0x7f2, 1");
    result &= checkBuff(1,offset, buff);

    printResult(result);

    return !result;
}
```