Pre-silicon and post-silicon testing of SIWA, a low-power RISC-V microcontroller.

Thesis submitted for consideration of an evaluating tribunal
of the Interuniversity Doctorate of Engineering Program to opt for the degree of Doctor of Engineering

Roberto Molina Robles
Post-graduate Studies Unit-Costa Rica Institute of Technology-University of Costa Rica

Cartago
Tuesday 11th February, 2025

In compliance with the regulations of the Doctoral Program in Engineering at the Costa Rica Institute of Technology, the thesis presented by the candidate to doctor is accepted. The Doctoral Examination was presented on December 2nd of 2024. The Evaluating Court of this thesis was integrated by the following members:

Dr.-Ing. Alfonso Chacón Rodríguez	Director—Advisor
Dr.-Ing. Ronny García Ramírez	Co-advisor 1
Dr.-Ing. Renato Rimolo Donadio	Co-advisor 2
Dr.-Ing. Carmen Madriz Quirós	Post Graduate Program Director
Dr.-Ing. Alfredo Arnaud	External Researcher/Professor

TEC | Tecnológico de Costa Rica

UNIVERSIDAD DE COSTA RICA

ACTA DE EXAMEN DE DOCTORADO

Cartago, Costa Rica

Programa Interuniversitario de Doctorado en Ingeniería

Título de Proyecto Doctoral: **Pre-silicon and post-silicon testing of SIWA, a low-power RISC-V microcontroller.**

Estudiante: **Roberto Molina Robles**

Tribunal evaluador:

Dr. Alfonso Chacón Rodríguez

Dr.-Ing. Ronny García Ramírez

Dr. Renato Rimolo Donadio

Dr. Alfredo Arnaud

Dra. Carmen Elena Madriz Quirós, (preside, representante de la Dirección de Posgrado)

Lugar, fecha y hora de presentación del Examen:

Tecnológico de Costa Rica, Campus Central, Cartago, 02/12/2024, 9:00 am

Los firmantes damos fe de que el Examen Doctoral de Roberto Molina Robles se ha realizado con transparencia y sin ánimos de perjuicio o favorecimiento del estudiante. Se han evaluado profundamente los temas que el estudiante ha desarrollado en su investigación de tesis. El examen se ha desarrollado manteniendo los más altos estándares de calidad.

Por esta razón y tomando en cuenta las rúbricas de evaluación de tesis escrita, defensa y examen oral, el dictamen del tribunal es:

[✓ **Aprobar**] ☐ **No Aprobar**, el examen doctoral del estudiante Roberto Molina Robles.

Nota final: 94.28

Observaciones:
Mención de Honor Cum Laude.

ACTA DE EXAMEN DOCTORAL

Estudiante: **Roberto Molina Robles**

TEC | Tecnológico de Costa Rica

UNIVERSIDAD DE COSTA RICA

Firmas:

[illegible]

Dr. Alfonso Chacón Rodríguez
Director de Proyecto Doctoral

[illegible]

Dr.-Ing. Ronny García Ramírez
Miembro del Comité Asesor 1

[illegible]

Dr. Renato Rimolo Donadio
Miembro del Comité Asesor 2

[illegible]

Dr. Alfredo Arnaud
Miembro del Comité Asesor 3

[illegible]

Dra. Carmen Elena Madriz Quirós
Representante de la Dirección de Posgrado

ACTA DE EXAMEN DOCTORAL

Estudiante: **Roberto Molina Robles**

Dedication

This thesis is dedicated to all the people who have supported me in the last 6 years. First, I would like to express my most profound gratitude to my thesis director, Alfonso, who has guided me for over a decade in all my university studies. I couldn't have been where I am right now, if not for him. I just hope I can repay him for all the things he has done for me over the years. Second, I would like to express my gratitude to my other colleague who also has worked hard on this big project, Ronny, his support has been invaluable to the point that I wouldn't have been able to finish this doctorate without his help, I am indeed lucky to be able to call you a true friend. Third, my beloved Jill with whom I have shared the last 9 years of my life, your unconditional support has been one of my most important pillars, I love you with all my being, I hope we can keep sharing our lives and keep supporting each other until the last of our breaths. Fourth, my elder parents Roberto and Elizabeth, and my brother, Ricardo, you always have been there for me and words will never express the gratitude I feel, but I'll always be there for you for whatever you need, I love you three. Fifth, I would like to thank people who I appreciate fondly, Alfredo, Matías and the uDIE crew, thank you form the bottom of my heart for your support and hospitality, this thesis also wouldn't have been possible without your help, you guys are truly awesome, I will always be in your debt and I hope I can visit you soon. Sixth, I want to give my thanks to four colleagues who have constantly supported my work and doctoral studies: Aníbal Coto, Renato, Francisco and Miguel, I am truly grateful for your promptness when I needed your help, thank you so much, without a doubt I will do the same for you, friends. Lastly, when writing this dedication I realized that many people have had a very important impact in my life, and the list is so big that it is very difficult for me to remember them all at the same time, but I will do my best. Sofía, thank you for being the bestest of friends, I dearly cherish you and hope we can keep being as close as we always have been. Kaled, Pablo, Juan José, José Miguel and Aníbal Ruiz, friends from my student days and now friends as fellow professors, I greatly enjoy my time with you all, I appreciate you and I hope our friendship lasts forever. Felipe, Luis and Bernardo, I am very grateful to be your friend, a friendship that has transcended time and distance, I eagerly await for our next DnD and gaming sessions. Roberto Pereira, it has been a long time since we spoke to each other, but I have not forgotten your support in the past, I wouldn't be where I am without your guide and I will do my utmost to support and repay you with whatever you may need. To my two deceased grandmothers, Aida and Rosa, thank you so much, and to the rest of my family, thank you!

Abstract

Making chips is a complicated and costly process that requires specialized knowledge. A single product of the semiconductor industry often involves many teams of engineers who work to make the chip as flawless as possible. The VLSI design flow is usually the compass needed to navigate these tricky seas, as it provides a detailed description of what is required to successfully build a chip.

With each passing year, chips have evolved ever more complex, drastically increasing research and development costs. A look into how the industry has changed is enough to understand the previous sentence. At the moment of this dissertation, there are far more companies in the semiconductor industry that specialize in specific aspects of the chip-making process than the decreasing number of Integrated-Device-Manufacturers (IDM) around the world. This trend is due to the increasing costs of chip manufacturing as designs and processes become more complex. Companies in this industry have learned to chain and intertwine their specialized business models to create chips at reduced costs.

However, small design teams and academic groups still struggle to design and create their chips as the effort required and costs are still considerable for them. As part of a group of academic research in microelectronics, this doctoral dissertation was built from the experiences and challenges we faced while developing a low-power RISC-V microcontroller for implantable medical devices, SIWA. In particular, this thesis focuses on the validation of our microcontroller, and the strategies discussed in this document were developed considering a low-budget and the scarce human resources at our disposal.

This thesis covers several aspects of the test process that SIWA went through over several years. In this document, the functional verification strategy to validate SIWA in the pre-silicon phases will be found. Then, a detailed description of the physical testing framework and how it was complemented with FPGA emulation is shown. Later, it is explained how software applications are built for SIWA and how I/O hardware emulation through software was achieved. Finally, this dissertation presents a benchmark proposal for classifying low-power RISC-V microcontrollers used in implantable medical devices. It is worth mentioning that part of the results of this thesis were four scientific publications indexed in the Scopus database, demonstrating that the effort done in my PhD studies made a contribution to the knowledge in the field of microelectronics.

Contents

List of Figures

List of Tables

Acronyms

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CPI	Cycles Per Instruction
CPT	Cycles Per Tasks
CSR	Control Status Register
DFT	Design For Testability
DRC	Design Rule Check
DUT	Design Under Test
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
GPIO	General Purpose Input Output
I2C	Inter-Integrated Circuit
IDM	Integrated Device Manufacturers
IC	Integrated Circuit
IoT	Internet of Things
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LIFO	Last-In First-Out
LVS	Layout Versus Schematic
MBC	Memory Bus Controller
MIPS	Million of Instructions Per Second
MISO	Master-In Slave-Out
MOSI	Master-Out Slave-In
MTPS	Million of Tasks Per Second
PCB	Printed Circuit Board
PEX	Parasitic Extraction
RAL	Register Abstraction Layer
RAM	Random Access Memory

RTL	Register Transfer Level
RV32I	RISC-V 32Integer
SCL	Serial Clock Line
SCLK	Serial Clock
SDA	Serial Data Line
SCS	Slave Chip Select
SPI	Serial Peripheral Interface
SV	SystemVerilog
UART	Universal Asynchronous Receiver-Transmitter
UVM	Universal Verification Methodology
VLSI	Very Large Scale Integration

Chapter 1

Introduction

In the last five decades, the semiconductor industry has become a pillar of society's daily life and future development. Many of us use electronic devices for our everyday needs. One of the easiest devices for us to understand the impact of this industry would be the cellphone. The cellphone is so important on our lives that we used it to communicate with other people we know, to connect with people we don't know, to stay informed of the latest news, as a mean of entertainment by consuming videos, music, books or videogames, as an useful tool in our respective works, to call on a transport to commute, and even as a means to pay and make monetary transactions, and many more things.

However, the importance of the semiconductor industry does not end there; consumer electronics is but one of many other markets that depend on the semiconductor industry. For example, aviation industry, automotive industry, medical devices industry, military industry, information technologies (IT) applications, artificial intelligence (AI) applications, space industry, drone applications, Internet of Things (IoT) industry, and many more. Each one of these sectors requires electronic chips to power their products and devices, and with each passing day more and more elements in our society incorporate chips to become intelligent and solve a necessity.

The semiconductor industry has been creating electronic chips since decades ago, and the process of creating these chips has evolved with time. Very-Large-Scale Integration (VLSI) design has become the cornerstone of modern electronics, enabling the creation of highly sophisticated integrated circuits by embedding millions of transistors on a single silicon chip. As a matter of fact, the industry has always tried to fulfill Moore's Law for more than half a century, and so far it has been successful. This continuous increase in integration over the years has led to an exponential increase in the processing capabilities of chips and a decrease in the cost per transistor, driving rapid advancements in technology and the electronics industry.

This integration requires sophisticated design methodologies to ensure that all components work seamlessly together. The design process involves not only the design and layout of the circuit, but also the management of power consumption, signal integrity, and interconnection, which must be meticulously optimized.

Verification and testing are equally intricate. Ensuring that a chip performs correctly under a range of conditions involves extensive simulation and emulation. This process is resource-intensive but also requires a deep understanding of both hardware and software interactions. Rigorous testing protocols are essential to confirm that chips operate reliably over their intended lifespan.

Now, as the integration of transistors becomes more dense with the passing of the years, electronic devices become increasingly more complex and ubiquitous. As a result, the design and verification processes of system-on-chip (SoC) and application-specific integrated circuit (ASIC) solutions have become very time-consuming, requiring a higher number of talented engineers to complete the design cycle.

Transistor miniaturization and chip manufacturing are not immune to this effect. The fabrication of chips at a nanoscale demands state-of-the-art equipment and extreme precision. The lithography techniques used to imprint intricate patterns onto silicon wafers are immensely complex and costly. Furthermore, maintaining high yield rates is crucial, as the proportion of functional chips per wafer can significantly impact production costs and timelines.

Electronic Design Automation (EDA) tools, required at every step of the electronic chip development cycle, have also seen a major increase in complexity. Many of these tools have heavily invested in AI to analyze and simulate a myriad of complex chips designs. However, these tools require a high degree of expertise to be properly used.

When we take all these points into consideration, we can understand why the semiconductor industry has naturally evolved into an ecosystem around the chip-making process. No wonder we see fewer Integrated Device Manufacturers (IDM) with each passing decade and an increasing trend of companies focusing their resources on a specific role inside this ecosystem.

Looking the complete picture and seeing the big names that come around when doing a quick search online about the semiconductor industry (Nvidia, Apple, Intel, TSMC, AMD, jut to name a few), many would think that, in the current state, this industry is only for big companies with large teams of electronics engineers. And to a certain degree, it may be, but it is also very possible for smaller groups to complete the VLSI design flow and develop their own semiconductor products.

This PhD thesis is a compilation of works fulfilling a role or step inside a bigger project for a developed silicon chip. What we aim with this thesis is to give insight about how the chip development cycle looks like from the perspective of a small team of engineers and how it is possible to develop semiconductor chips inside an ecosystem filled with tech giants and complex processes.

1.0.1 General background

Tecnologico de Costa Rica, as a university focused on teaching and researching high-technology topics, is no stranger to semiconductor development. The Electronics Engineering School has at least three research laboratories and more than 25 years of experience developing ASICs, digital and analog integrated circuits, and FPGA-related applications, among others. This thesis dissertation is a product of one of these types of projects.

Specifically, the DCILab (Laboratorio de Investigación de Circuitos Integrados Digitales) had been working on a big project for several years, about creating a low-power RISC-V microcontroller aimed for implantable medical devices. This thesis is a compilation of works done to resolve particular testing problems inside the development cycle of this microcontroller.

Semiconductors play a crucial role in the functionality of implantable medical devices, enabling the miniaturization, power efficiency, and advanced capabilities necessary for life-saving technologies. These devices, such as pacemakers, insulin pumps, and neurostimulators, rely on semiconductor components to process and transmit data, manage power consumption, and perform precise control of therapeutic functions. The ability of semiconductors to integrate complex circuits on a small scale allows for the development of compact, reliable, and responsive devices that can monitor, diagnose, and treat various medical conditions directly within the body. This advancement not only improves patient outcomes, but also expands the possibilities for innovative treatments and continuous monitoring, significantly improving the quality of healthcare.

1.0.2 Definition and justification of the problem

This doctoral dissertation is framed within a long-term development project with the goal of building a low-power general-purpose RISC-V microcontroller. As the application involving said microcontroller is inside the human healthcare dome, testing this microcontroller thoroughly is paramount. Any medical device that may use this microcontroller has to be safe, functionally correct and reliable. Therefore, the microcontroller needs to pass through a series of testing stages along its design to properly validate its intended functionality.

Now, creating a chip is no simple feat and is even more so when the team involved in chip design and test is small with limited resources. Creating a chip under these circumstances presents significant challenges, requiring ingenuity and strategic trade-offs. With constraints on funding, expertise, and advanced equipment, the team must find innovative ways to optimize the chip's architecture, often simplifying designs to reduce complexity and cost. In addition, limited access to state-of-the-art fabrication facilities forces reliance on older, less efficient technologies, potentially impacting performance and scalability. However, for applications where chip size and performance are not that important, like in medical applications, using old technologies may prove beneficial, as they are more reliable.

Additionally, the lack of a large, specialized team can slow down the development process, as engineers must wear multiple hats, handling everything from design to testing. Despite these hurdles, resourcefulness and creativity can lead to successful outcomes, although the path is fraught with obstacles and demands a high degree of perseverance and problem solving.

Encapsulating the issue within the scope of this doctoral dissertation, there were four main problems inside the larger project that were addressed. As such, one can summarize the definition of the problem for this doctoral dissertation with the following list.

- Assurance that the microcontroller was functionally operated according to the RISC-V standard prior to its fabrication.
- Once fabricated, functionally testing the physical RISC-V microcontroller with limited equipment and resources.

- Testing the communication capabilities of the RISC-V microcontroller with other devices.
- Measuring the performance and power consumption of the RISC-V microcontroller and comparing the results versus other microcontrollers used in medical devices.

1.0.3 General objective of the doctoral dissertation

Taking into account the previous considerations, the general objective of this dissertation was defined as:

To develop a thorough verification framework for a RISC-V microcontroller that provides pre-silicon and post-silicon validation, testing and benchmarking against other microcontrollers used in implantable medical devices, while ensuring compliance with RISC-V standard.

The engineering design process was used as the methodological procedure to complete this objective and each partial goal involved. This design method generally uses the following steps: definition of the problem, search for pertinent information, generation of multiple solutions, selection of the best alternatives, and finally implementation and verification of the solution.

Specific objectives

The general objective was achieved by fulfilling a series of smaller specific objectives. Each one of them was defined based on the problems we found along the way in the testing process, and they will be explained shortly. Four of them were needed to complete the general objective.

In the early stages of development, the RISC-V microcontroller needed a functional verification environment to properly test its functionality before fabricating. Fabricating without a verification environment would just increase the chances of failed fabrications, incrementing costs. Therefore, the first specific objective can be expressed as:

To design a functional verification environment for a RISC-V microcontroller capable of running hundreds of automated tests based on its specification to catch errors before its fabrication.

The next step was to prepare a post-silicon validation platform to test the physical chip once its fabrication was realized. One of the main challenges was to create this platform at low cost. Hence, the second objective was written as follows:

To create a post-silicon validation platform for a RISC-V microcontroller to properly test its basic functionality at low cost and ensuring compliance with the RISC-V standard.

Once the chip was fabricated and tested, we wanted to exploit and show the capabilities of the fabricated RISC-V microcontroller. To do that we needed to communicate the microcontroller with an OLED display, to ensure the microcontroller was capable of communicating with other devices and emulating I/O interfaces. With that, the third specific objective was set as follows:

To develop and program an application inside the RISC-V microcontroller to establish communication with an OLED Display using I2C protocol as the communication mean between both devices to demonstrate the microcontroller's operability.

Finally, the last challenge was the necessity to compare the manufactured microcontroller with other commercial microcontrollers used in implantable medical devices. As there were not many clear ways to compare two microcontrollers in this field, it was decided to create a method to compare microcontrollers for implantable medical devices. Having said that, the final specific objective was defined as:

To design a benchmark for microcontrollers used in implantable medical devices to be able to compare and rank them under similar conditions.

1.0.4 Methodology

To create a functional microchip the semiconductor industry follows a series of processes around designing, manufacturing, and testing the microchips. Semiconductors are materials that have a variable electrical conductivity between conductors (like metals) and insulators (like glass) depending on how impurities are injected on them. They are fundamental to modern electronics because they can be precisely engineered to control electrical currents through the transistor, making them essential for creating integrated circuits and microchips.

These processes encompass a wide range of activities, yet the main three steps that summarize how chips are created inside the semiconductor industry appear in figure 1.1.

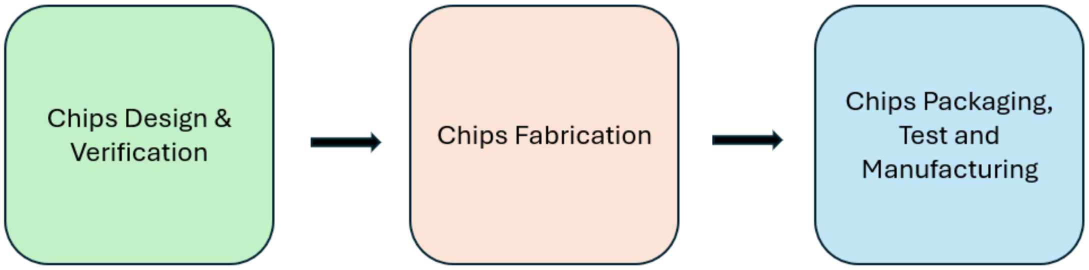

Chips Design & Verification
Chips Fabrication
Chips Packaging, Test and Manufacturing

Figure 1.1. General steps to create a chip in the semiconductor industry.

In the design and verification phase, companies create complex microchips and integrated circuits that form the brains of various electronic devices. This involves developing new architectures, optimizing performance, validating the designs, and defining the physical characteristics of the chip. Semiconductor fabrication occurs as a second step, involving the production of actual chips through processes like photolithography, etching, and doping. This stage requires highly specialized facilities known as foundries or fabs. As a last step, chips undergo rigorous testing to ensure they meet performance standards. They are then packaged for integration into electronic devices.

Now, this is a very minimalistic approach, as many important details are omitted. Therefore, a more deep analysis is in order, and for that purpose we should look at the VLSI design flow, a process our microcontroller went through. This will lay the methodological foundation to understand the aspects on which this thesis was focused.

1.0.5 Scientific Contributions and Dissertation Structure

The remainder of this doctoral dissertation is structured as follows. Chapter 2 briefly explains the state-of-the-art for the current thesis, and contains relevant information that helps to understand the starting point for these projects. Chapters 3, 4, 5 and 6 describe the work carried out to fulfill the specific objectives mentioned in this chapter, respectively. Hence, Chapter 3 explains a verification environment built for the RISC-V microcontroller. Chapter 4 details the post-silicon validation framework for physically testing the microcontroller. Chapter 5 has information about an I2C application and the software engineering behind it. And finally, Chapter 6 presents a designed benchmark devised in order to compare microcontrollers employed in implantable medical devices. With that in mind, the aforementioned chapters can be considered as successful contributions to the scientific community, and more specifically to the electronics engineering field inside the semiconductor space, as each one of them fills a niche gap inside the state-of-the-art. Chapter 7 is an addendum that presents additional technical work that was critical to the fulfillment of the project, and it helps to visualize the roadmaps for the future of the project.

Chapter 2

State of the Art

The "state-of-the-art" refers to the highest level of development, knowledge, or technology available in a particular field at any given time. It represents the most recent advances, techniques, methods, or innovations that are regarded as best practices in a specific area. In the context of project development, understanding the state-of-the-art is essential because it provides insight into the most effective tools, strategies, and approaches currently available. By being aware of cutting-edge developments, project teams can make informed decisions, remain competitive, and avoid reinventing the wheel.

Incorporating state-of-the-art technology or methods into a project improves its quality, efficiency, and relevance. It allows teams to take advantage of the latest innovations, which can lead to improved performance, faster development cycles, and reduced costs. For example, utilizing state-of-the-art software or frameworks can streamline processes and optimize results. In the case of this thesis, it certainly helped to create reliable test methods for a RISC-V microcontroller. Furthermore, understanding these advances enables teams to anticipate trends and ensure that their projects remain futureproof in the rapidly evolving technology industries.

The importance of knowing the state of the art also extends to identifying potential limitations or challenges associated with the latest technologies. It provides a clearer understanding of what has been accomplished and where gaps remain for scientific contributions, and guiding further research and development. A project that integrates state-of-the-art approaches is more likely to be innovative and robust, defining it in a competitive market. In this sense, keeping up with the state-of-the-art is not just beneficial but critical for sustained success and growth in any field.

In this chapter, a general state-of-the-art review of the chip making process will be presented. However, each chapter has the detailed state-of-the-art for the particular solution it addresses.

2.1 VLSI Design Flow

The VLSI design flow is a detailed and systematic approach used to develop chips, which encompasses several stages from concept to production. VLSI design is not a new concept; the reality of integrating transistors into small circuits has been around since 1960s-1970s, just in the decades when Moore's Law was created. However, rapid advancements in electronics products, computers, and design tools have allowed the VLSI design flow to continuously evolve and adapt to new technologies and trends. As of 2024, the VLSI design flow is well established and has been stable for several years, although the big leaps made by AI (which, to a degree, is also a product of semiconductor development) in EDA tools may change its approach in the coming years. Nevertheless, I allow myself to take a snapshot in figure 2.1 of how the VLSI design flow looks to this day.

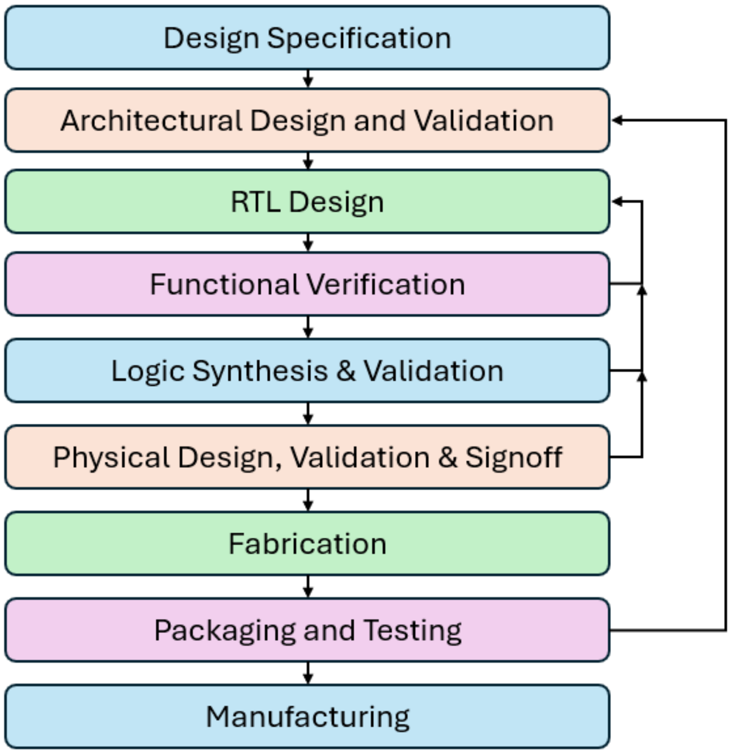

Design Specification
Architectural Design and Validation
RTL Design
Functional Verification
Logic Synthesis & Validation
Physical Design, Validation & Signoff
Fabrication
Packaging and Testing
Manufacturing

Figure 2.1. VLSI Design Flow.

Next, a brief summary for each of the steps described in Figure 2.1 VLSI design flow is provided below as they set up the basis of the thesis.

2.1.1 Design Specification

The VLSI design flow begins with the initial concept and specification phase. This stage involves defining the requirements and functionalities of the intended integrated circuit. The specifications outline the chip's performance goals, such as speed, power consumption, and functionality. This phase usually requires collaboration between architects, designers, and engineers to ensure that the final product meets the needs of the intended application. These needs are established after doing a throughout analysis of the market in which the electronic product will be part of and the gaps inside said market that are intended to be filled. In this first step, creating comprehensive documentation and clear communication is crucial to translate high-level requirements into actionable design goals.

2.1.2 Architectural Design and Validation

Once the specifications are set, the architectural design phase begins. This stage involves creating a high-level architectural model of the chip that outlines the overall structure and organization of the system. Architects design functional blocks and their interactions, such as processors, memory units, and input/output interfaces. They also determine the data flow and control mechanisms within the chip. The goal is to establish a blueprint that will guide subsequent design stages. Tools such as block diagrams and functional specifications are used to visualize and plan the architecture accordingly.

2.1.3 RTL Design

The Register-Transfer-Level (RTL) design phase follows the architectural design as a guide to construct blocks that conform to the developed chip. In this phase, designers convert the architectural model into a detailed RTL description using Hardware-Description-Languages (HDLs) such as SystemVerilog. RTL describes the operation of the chip at an abstraction level that focuses on data transfer between registers and the logic operations performed on these data. Therefore, this phase involves writing RTL code to define the behavior and functionality of the chip. The RTL code should undergo a series of simulations to ensure that the design will meet the specified requirements. Afterwards, the RTL code should be synthesized to analyze a series of physical characteristics and determine size estimates for the chip.

2.1.4 Functional Verification

Functional verification is a critical phase that ensures that the RTL design behaves as intended. This stage involves simulating the RTL code to verify its functionality against the specifications defined by the architects. Various verification techniques, such as designing sophisticated testbenches and assertion-based verification, are used to validate the design. Simulation tools run extensive test cases to verify logical errors, functional discrepancies, and performance issues. The goal is to identify and rectify any issues before moving onto the next stages of design, reducing the risk of costly errors in later phases.

2.1.5 Logic Synthesis and Validation

The logic synthesis phase involves translating the RTL code into a gate-level netlist, which represents the design using basic logic gates and flip-flops. This process is carried out using synthesis tools that optimize the design for various metrics, such as area, speed, and power consumption. The synthesis tool maps the RTL description to a library of standard cells, which are predefined building blocks with known characteristics. This library is facilitated by the company that is in charge of fabricating the chip according to the node and technology chosen for the developed chip. The gate-level netlist serves as the basis for the physical design stage. Synthesis is a crucial step in ensuring that the design is feasible for fabrication.

2.1.6 Physical Design, Validation and Signoff

The physical design phase, also known as the layout, or place-and-route phase, involves transforming the gate-level netlist into a physical layout that defines the geometry of the chip. This stage includes several substeps:

- Placement: The placement step involves positioning the standard cells on the layout of the chip. The goal is to optimize placement to minimize wire length and improve performance while adhering to the design rules defined by the node and technology.
- Routing: Routing connects the cells using metal layers to create the necessary electrical connections. This step ensures that the interconnections meet the design specifications and do not violate any manufacturing constraints.
- Design Rule Checking (DRC): DRC verifies that the physical layout adheres to manufacturing design rules, such as spacing and width requirements for various layers, among many others.
- Layout Versus Schematic (LVS) Checking: LVS ensures that the physical layout matches the gate-level netlist and that there are no discrepancies between the design intent and the actual layout.
- Parasitic Extraction (PEX): This step involves extracting parasitic effects, such as capacitance and resistance, from the layout to analyze their impact on the chip's performance and signal integrity.

After the layout has been constructed, a timing analysis is performed to ensure that the chip meets its timing constraints and operates correctly at the desired clock speeds. This involves analyzing the timing of various paths within the design to ensure that signals propagate through the chip within the required time constraints. Static Timing Analysis (STA) tools are used to check for timing violations, such as setup and hold time violations, and to optimize the design if necessary, ensuring proper timing is crucial for the reliable operation of the chip.

At the same time, power analysis is also performed to evaluate and optimize chip power consumption. This involves estimating dynamic and static power dissipation and identifying areas where power consumption can be reduced. Power analysis tools help in assessing the impact of various design choices on power consumption and in implementing power saving techniques, such as clock gating and power gating. Efficient power management is essential for battery-operated devices and for reducing heat dissipation in high-performance systems.

Optionally, design-for-testability (DFT) structures may be incorporated into the designed chip to facilitate manufacturing testing and quality assurance. This includes adding scan chains, built-in self-test circuits

(BIST), and other mechanisms to enable effective testing of the functionality of the chip. DFT ensures that the final product can be thoroughly tested for defects and reliability before it reaches the market. The bigger and more complex the chip, the more important DFT structures become to detect manufacturing failures.

2.1.7 Fabrication

The final phases of the VLSI design flow are fabrication, testing, and manufacturing. The chip design is sent to a semiconductor foundry for manufacturing, where it is processed through a series of photolithography, etching, and deposition steps to create the final silicon wafers. The chips are then cut off from the wafer and sent back to their designers or testing facility.

2.1.8 Packaging and Testing

Once fabricated, the chips are packaged according to the specification and then undergo extensive testing to ensure that they meet the performance, functionality, durability, and reliability requirements. Testing at this level includes functional tests, performance evaluations, and reliability assessments. Many tests are also created based on the DFT structures added to the design. If the chips are not working properly or the result is not satisfactory for mass production, then the process is repeated until the chip works as intended.

2.1.9 Volume Manufacturing

The last stage is mass-production of the chips if the prototype chips passed all tests and worked as defined by the architects. Although the design phases are over and the functionality of the chips has been validated, all units should undergo a series of tests to detect defective products and discard them. The products that surpass these tests are ready to be shipped and sold. At this point, metrics such as good yield are important for economic success.

Chapter 3

A compact functional verification flow for a RISC-V 32I based core[1]

Functional verification is hardly a new topic. There is a solid standard [2], and many teaching resources are available (for example, [3] for a comprehensive site with plenty of free resources). And there exists some recent literature with examples that can provide guidance to nonexperienced verification teams. Some works, for example, show how to implement verification environments with or without UVM, such as [4], where a custom environment was proposed to improve coverage, or [5] and [6], where UVM was applied to different RTL blocks. Yet, most of the documentation available points towards the use of massive, highly integrated frameworks attached to a particular methodology—typically derived from the Universal Verification Methodology (UVM)—, where the access to commercial tools and personnel is not a limitation, and the departure specifications for the expected results are readily defined (or at least do not depend directly on the verification team itself).

Small design teams usually do not have the budget to tackle the verification problem using such an approach, particularly when the design and verification process must be handled by the same people. This means finding ways to optimize hardware and software resources, through the use of open source tools whenever possible, while providing a flexible environment that can easily be migrated from a project to another. Now, some examples of small teams using functional verification for their chip approach can be found. Yet, to our knowledge, most of the goal specifications in those papers were already defined by the use of a standard given architecture (SiFive's Rocket-chip), written in Chisel. This was not the case here, being Siwa an architecture written from scratch, with several modifications from the RISC-V 32I standard mandated by the ultra-low power specifications of its intended application.

The main purpose of this chapter is to document the implementation of a functional verification environment used for the pre-silicon verification of a RISC-V 32I processor, called Siwa, developed by a small team of only six people as the main controller core of a medical implantable tissue stimulator (see [7] for details on the processor and the medical device system itself). The present work gathers strategies as those used in the previous references, and incorporates them not only for the verification of architectural blocks, but also adds

[1] This chapter contents have been published verbatim in [1]

up the use of reference models, the integration of custom architectures and the automation of regressions for random verification, topics often missing in the literature.

As such, the structure presented here allows for functional verification at different hierarchical levels, using oriented and random-constrained tests, based on custom reference models, or requiring the incorporation of code simulators serving as golden vector generators, while using regression systems for extending code coverage. All of this in an environment compact and versatile enough for a verification team of only three people.

This chapter is organized as follows: Section 3.1 briefly describes the tools and resources selected for the verification of the DUT and the reasons for that selection. Section 3.2 summarizes the most important details of the DUT, as it is the basis for the construction of the verification plan and helps build the context for the remainder of the chapter. Section 3.3 describes the strategy developed for the DUT's verification efforts. It explains the structure of the verification platform, approaches, and executed tests. Section 3.4 shows some coverage results and how the platform looks when a regression is run on the DUT. Section 3.5 refers to an updated version of the developed environment, specifically how chip-level verification was migrated to UVM and the changes made to reuse as much code as possible. Lastly, section 3.6 highlights the main conclusions of the chapter and discusses some future work.

3.1 Definition of Tools and Resources

Small IC design teams with modest financial resources typically require affordable yet competitive low-maintenance tools. However, one cannot always resort to the open source community for such tools, although there is a strong movement pushing in that direction of open hardware design, for example, the Linux Foundation CHIPS Alliance. And that is because there is a lack of technology kits for open design tools, as foundries base their production environments on the three major EDA providers: Mentor Graphics, Synopsys, and Cadence. Particularly, our group has a Synopsys license available.

In our case, the selection of SystemVerilog (SV) as the specification language and UVM as the verification methodology was straightforward. SystemC was discarded due to its lack of support from the Synopsys synthesis tools and the library flow provided by the foundry for the project, with the added extra of SV already having a UVM library incorporated and being supported by most of compilers/simulators. This meant using Synopsys VCS for the compilation, simulation, and coverage processes and Synopsys DVE as the Waveform Viewer and Coverage Analyzer of choice. This is mainly because of the tools' easy interfacing with the Design Compiler and IC Compiler flows. However, alternative simulators (such as Verilator) and the wave analysis tool (GTKwave) may be used.

Concerning the construction of the regression platform, the Linux Cron utility and basic Bash scripting were selected, although there are other options such as Jenkins and Bamboo that are typically more favored by industry due to their wide array of features and support for large development teams. However, Cron is easier to setup and less demanding in terms of computing resources.

3.2 DUT Summary: RISC-V microcontroller

To grasp a better understanding of this chapter, it is convenient to start with a brief summary of the DUT (Design Under Test), a RISC-V 32I. Figure 3.1 depicts the microarchitecture of the DUT.

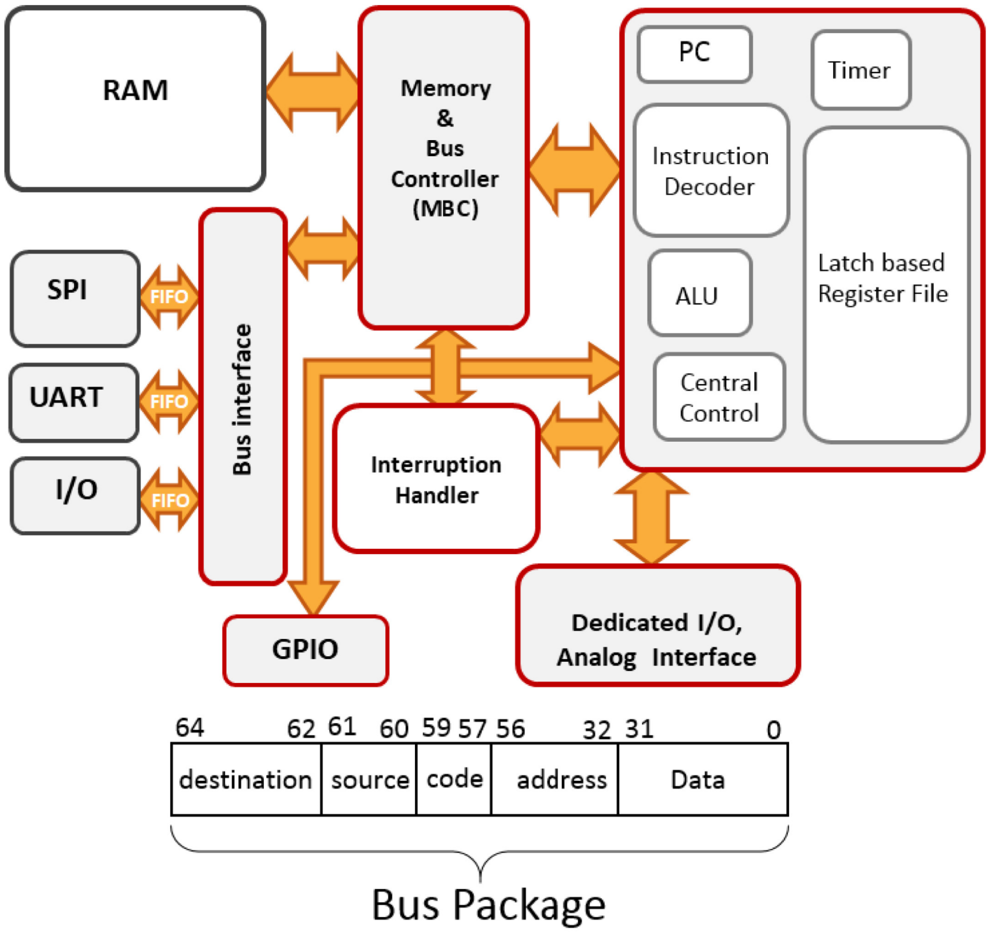


Figure 3.1. Microarchitecture of the verified DUT, SIWA, a RISC-V 32I microcontroller [7].

The elements of this microcontroller can be listed as follows:

- A simple non-pipelined escalar core that functions as the main processing unit. This block contains basic elements such as a PC, ALU, Timer, Decoder, and a Control FSM.
- A RAM memory where the Instruction Memory, Data Memory and Stack are placed.
- A Memory-Bus Controller (MBC) acting as the smart interface between the RAM memory, the bus, and the core.
- A typical Interruption Handler for special cases where the core has to interrupt its normal flow.
- A bus that works as the interface between I/O modules (SPI, UART, etc.) and the MBC.

• GPIO and analog I/Os are connected directly to the core.

For more information on how its microarchitecture works, please refer to [7], focused on medical applications [7]. RISC-V architecture specifics can be found at [8], although, chapter 5 will expand a little bit more on the ISA when explaning a developed application on SIWA.

3.2.1 The RISC-V architecture

We have addressed a brief introduction to the makings of a chip so far. However, we are missing one very important aspect, which is referring to the application of the created microchip. This thesis has been written around the development of a RISC-V microcontroller, planned to be used in low-power medical applications. Thus, an introductory section about the RISC-V architecture had to be included to have the complete background picture of the work described on this thesis.

RISC-V is an open source instruction set architecture (ISA) that has garnered significant attention for its potential to mold to different types of applications. RISC-V architecture is based on Reduced-Instruction-Set-Computing (RISC) whose paradigm revolves around a simple and modular computer architecture. Unlike proprietary ISAs, RISC-V is freely available for modification and implementation, allowing hardware designers and researchers to innovate without the constraints of licensing fees or restrictive patents. Designed with simplicity, modularity, and extensibility in mind, RISC-V is designed to support a wide range of applications, from low-power embedded systems to high-performance computing. Its open nature fosters collaboration and rapid development, positioning RISC-V as an alternative to established architectures on computing technology.

Computer architecture is the conceptual design and fundamental operational structure of a computer system. It encompasses its organization (or microarchitecture) and interrelationship of the computer's key components, such as the data processing units, flow control units, memory, and input/output (IO) devices. These components work together according to the specification of a particular computer architecture to execute instructions, process data, and perform tasks. RISC-V, like most computer architectures, falls under this definition. Figure 3.2 illustrates the interaction mentioned above.

As any other ISA, the RISC-V specification is quite lengthy and covers many aspects that directly influence its instruction set, such as instruction length, instruction formats, data types, data register bank structure, control status register structures, variations of its instruction sets, etc. This thesis will not cover all this information as it transcends the contents of this document. Please refer to [8] for more information on the RISC-V ISA. However, one thing worth mentioning is the specific variant of RISC-V ISA of the microcontroller we developed, RISC-V 32I (RV32I).

RISC-V was designed to have a minimalist instruction set to reduce the hardware required for implementation. RV32I is the RISC-V instruction set in its most base form, focused on integer operations (hence the 'I' in its name), and it contains 47 unique instructions. Just for reference, figure 3.3 and figure 3.4 show the register bank and the instruction set for the RV32I variant, both pictures acquired from [8]. To view the list of CSRs, refer to [9].

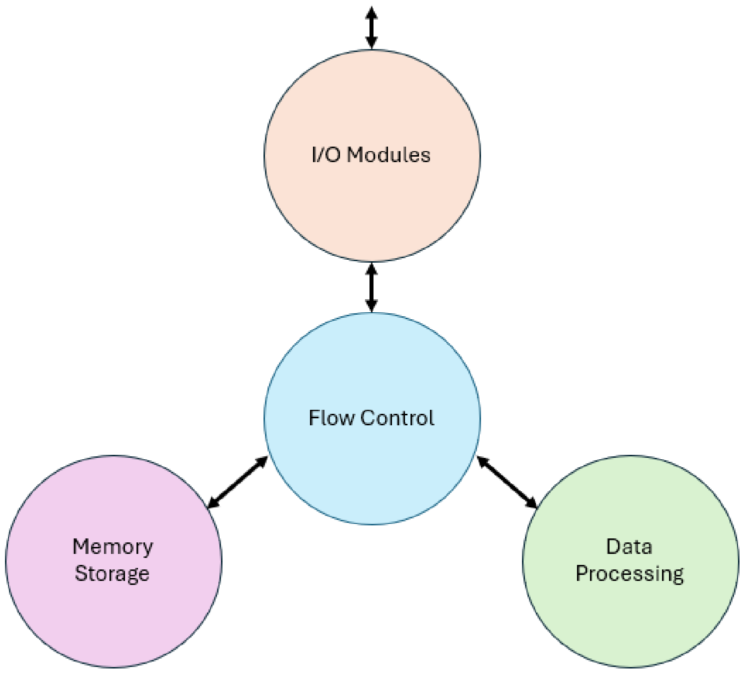


Figure 3.2. Base functionality of a computer.

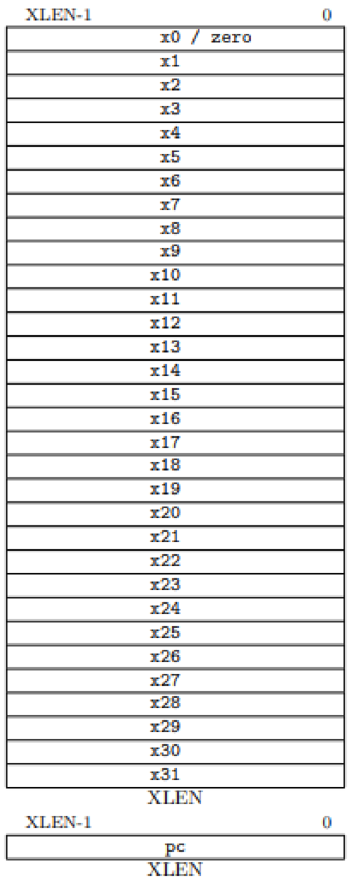
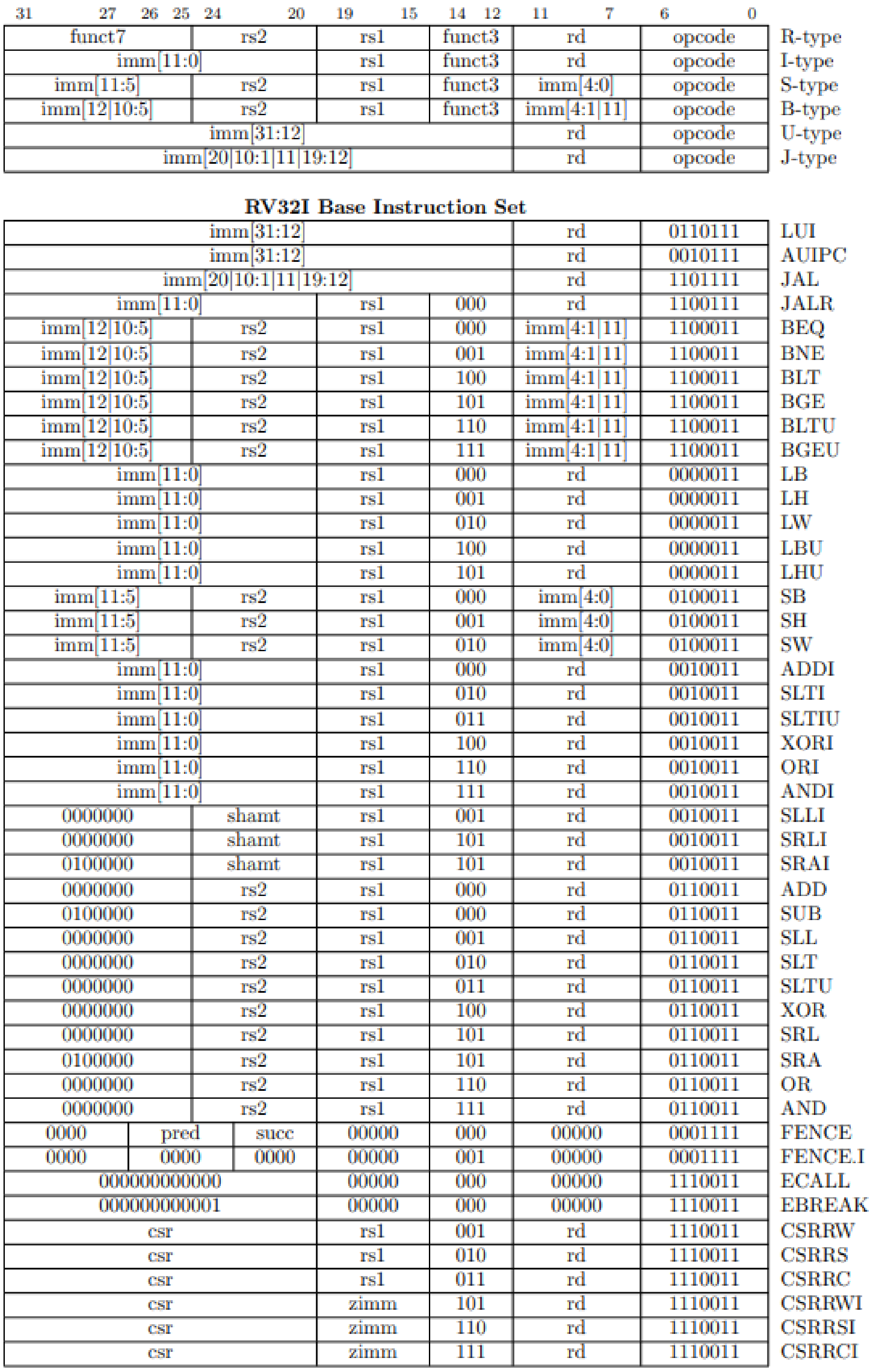


Figure 3.3. Register bank used for data processing on the RV32I variant. [8]



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

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2		rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2		rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2		rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2		rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2		rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2		rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]	rs2		rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2		rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2		rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000	shamt		rs1	001	rd	0010011	SLLI
0000000	shamt		rs1	101	rd	0010011	SRLI
0100000	shamt		rs1	101	rd	0010011	SRAI
0000000	rs2		rs1	000	rd	0110011	ADD
0100000	rs2		rs1	000	rd	0110011	SUB
0000000	rs2		rs1	001	rd	0110011	SLL
0000000	rs2		rs1	010	rd	0110011	SLT
0000000	rs2		rs1	011	rd	0110011	SLTU
0000000	rs2		rs1	100	rd	0110011	XOR
0000000	rs2		rs1	101	rd	0110011	SRL
0100000	rs2		rs1	101	rd	0110011	SRA
0000000	rs2		rs1	110	rd	0110011	OR
0000000	rs2		rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Figure 3.4. Instruction format and set for the RV32I variant. [8]

3.3 Functional Verification of SIWA

Before addressing the verification architecture, a roadmap with a testplan was first created. This means studying the design through specification documents and industry standards, and planning tests with architects and designers. Having adequate knowledge about the functionality of the chip is critical in the verification process. Coding can be tackled once the verification plan is ready and a strategy has been established. Methodology, verification architecture, testplans, resources and tools, time-lined efforts, coverage points, and other aspects are expected sections of the verification plan, and should be reviewed several times with architects, designers, and other verification engineers.

The environments were planned to be custom built for our microcontroller, some of them were UVM based, and others were constructed using basic SystemVerilog. In the end, all non-UVM environments were converted to UVM. Now, since the processor was a small RTL design, the verification efforts were focused on only two hierarchical levels. The lower level for block verification and the higher level for chip verification. The general strategy was to use coverage-driven simulation-based verification. It is recommended to check [10] to study the characteristics needed in a simulation-based environment.

3.3.1 Block-Level Verification

The selected blocks submitted for verification were the Arithmetic Logic Unit (ALU), the Memory Bus Controller (MBC), the System Bus, the Universal Asynchronous Receiver-Transmitter Port (UART) and the Bus. The ALU and UART blocks were designed using a standard register-based RTL approach, with minor custom modifications. The MBC and the bus used a latch-based microarchitecture for area and power reduction. For block-level verification, UVM was used to implement verification architectures. The SPI block, GPIOs, and the main core block were stimulated at the chip level when bootstrapped or when running specific programs. The RAM memory block was a third-party IP, hence, it was assumed as verified. A gray box approach was selected for the block-level verification efforts, depending on the characteristics of each one of them. In [11], there is a clear explanation of the advantages of this type of verification. Figure 3.5 shows the generalized block diagram for the block-level architecture, based on [2].

There were small variations in the architecture's implementation for each individual block, but the general idea remains the same, using agents inside an UVM environment to stimulate a block and monitor the interfaces, then predict and compare the output. For example, the TX Agent and RX Agent from Fig. 3.5 were fused together into a single agent for the ALU and UART RTL blocks, as their specifications were simple enough with few interfaces. In contrast, the MBC and Bus needed more than two agents, as they had many interfaces. Of course, this type of modification is supported inside the UVM standard, but one must remember that the less agents one has, the easier its environment construction band, the harder its maintenance.

The ALU and the UART are blocks with standard interfacing, not prone to change in future microarchitecture updates, and thus, agents may be considered stable and not in need of much maintenance on future chip versions. Meanwhile, the MBC and Bus are blocks that may change depending on features that might be added or removed in further spins. If we separate the interfaces in features/devices and connect an individual agent to it, then we can only adjust the associated agent instead of modifying a more complex and bigger

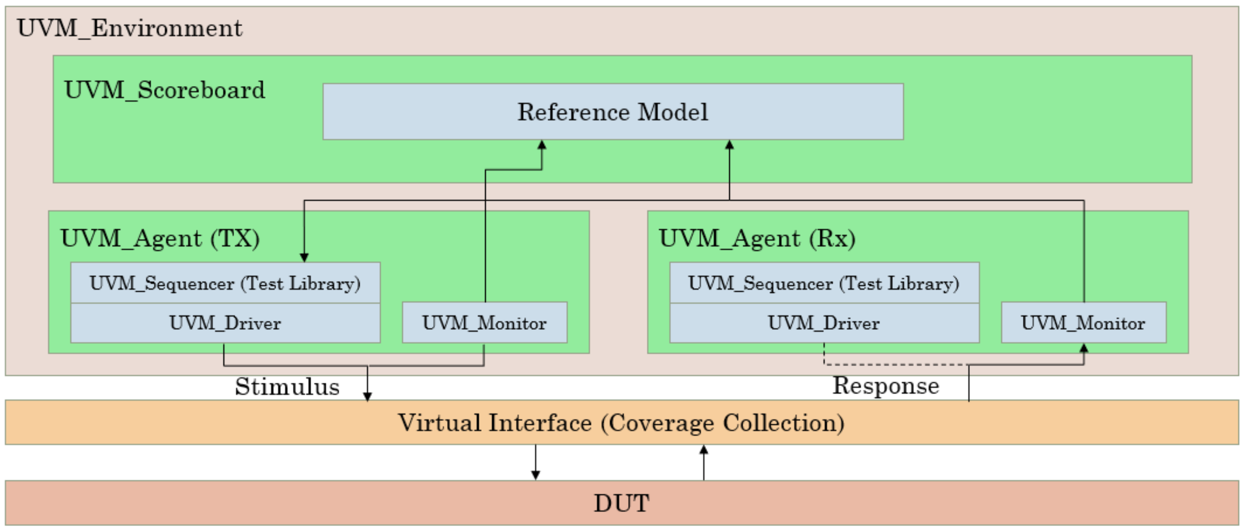


Figure 3.5. A testbench implemented using a UVM architecture for block-level verification. The quantity of agents varies depending on the modularity of the design block interfaces. The sequencer calls sequences built with sequence items to form a test. The desired test is called from the command line.

agent connected to all interfaces at once.

At the block-level, the reference model used was a transaction-based model [11]. That is, at every transaction between the verification environment and the DUT, a checking is being made between the custom reference model's prediction and the actual results of the DUT. This reference model was implemented in SV inside the scoreboard, based on a written specification to avoid similarities with the RTL implementation as much as possible. Finally, code coverage and functional coverage were implemented in each environment to guide the stimulus as any other coverage-based verification strategy.

Table 3.1 lists some of the major tests implemented to verify the blocks inside SIWA:

3.3.2 Chip-Level Verification

At chip level verification the DUT is the complete RISC-V microcontroller and different challenges arise versus its block counterparts. First, a major consideration at top-level verification of microcontrollers is that the core runs programs that cannot be totally randomized: for instance, memory addressing instructions have limited valid ranges, and also, certain registers have pre-defined functions. Secondly, the core has concurrent interfaces like the UART, SPI and GPIO, which may or may not have to respond back. Third, one of best ways to debug a processor is looking into its register bank when running instructions, and this is specially true when you are running and debugging a program in an emulator. However, none of these registers can be externally accessed, except by using indirect methods like communicating via UART or SPI ports, making simulations extremely slow and heavy. Finally, timing is really important when checking the actual data of the registers as the time needed to execute instructions varies depending on the core structure (for example, simple scalar multicycle versus pipelined scalar versus pipelined superscalar) and the microarchitecture could change in newer versions. Taking all these considerations into account could save a lot of time and effort in

Table 3.1. Test list for each verified block inside SIWA.

Block Name	Test Name	Description
ALU	Signed Add Test	Signed add test with random operands.
	Unsigned Add Test	Unsigned add test with random operands.
	Unsigned Subtract Test	Unsigned subtrack test with random operands.
	Random Logic Op Test	Random OR|XOR|AND with random operands.
	Comparator Op Test	GreaterThan|LessThan|Equal with random operands.
	Shift Op Test	Random Shift Right|Left with random operands.
	Arith. Shift Op Test	Arithmetic Shift Right with random operands.
MBC	Mem. to Core Read Test	Read transaction from the RAM memory to the core.
	Mem. to Bus Read Test	Read transaction from the RAM memory to the bus.
	Core to Mem. Write Test	Write transaction from the core to the RAM memory.
	Bus to Mem. Write Test	Write transaction from the bus to the RAM memory.
	Misaligned Read/Write Test	Read/Write misaligned operations.
	Interrupt Push Test	Data push interruption.
	Interrupt Request Test	Data request interruption.
	Interrupt Clean Test	Clean signal interruption.
Bus	MBC to SPI Test	Packet transaction from MBC to SPI.
	MBC to UART Test	Packet transaction from MBC to UART.
	SPI to MBC Test	Packet transaction from SPI to MBC.
	SPI to UART Test	Packet transaction from SPI to UART.
	UART to MBC Test	Packet transaction from UART to MBC.
	UART to SPI Test	Packet transaction from UART to SPI.
UART	TX Test	Send packet on the UART port.
	RX Test	Receive packet from the UART port.
	Bad transfer Test	Introduce errors during transfer.

the environment development, test creation, and maintenance.

These restrictions imposed a Gray-Box approach too for the chip-level verification approach, where the register bank is better accessed via a backdoor. A "Golden Reference" methodology was selected, as recommended by [11], although it is possible to use a transaction-based reference model if the checking occurs at the end of each instruction cycle. The generation of test programs was handcrafted at this level as the programs had to make sense and be as close as possible to the programs it is going to run in a physical setup. Figure 3.6 shows the custom architecture of the chip-level verification environment.

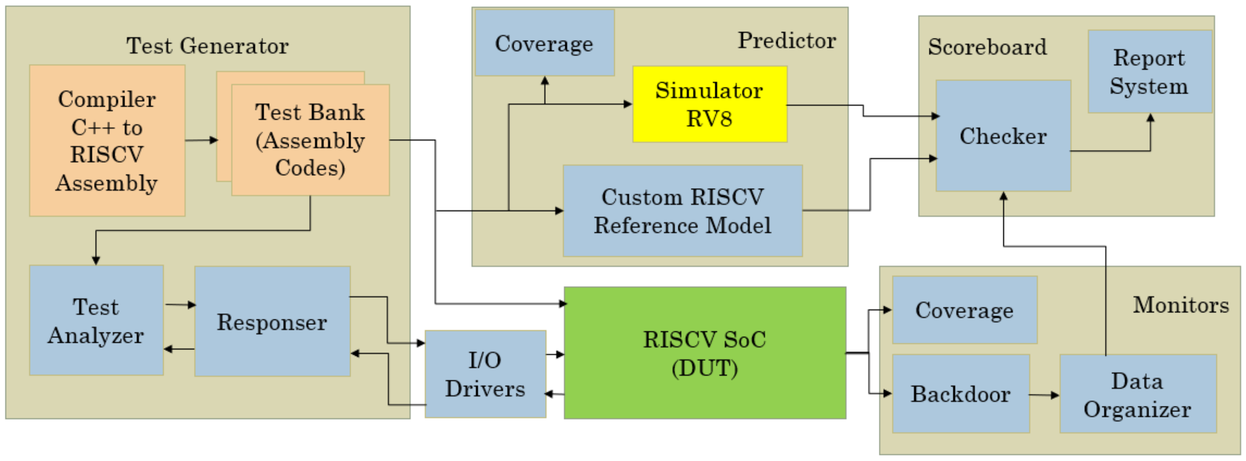


Figure 3.6. Custom architecture implemented for pre-silicon testing of a RISC-V core. The test generator calls a program created with a RISC-V compiler and loads it inside the DUT. The responser communicates with the DUT if it is instructed in the program. Monitors are connected via backdoors to registers and specific control signals. Blue blocks and the DUT were written in SystemVerilog.

The main idea behind this environment is: a program is loaded and executed in Siwa's RTL model, then information from the data and control-status registers (CSRs) is stored in an array at the end of each instruction cycle. Simultaneously, a reference model predicts the correct result that ought to be stored in each register for every instruction, and stores it as well in another array. The expected and actual data arrays are compared after the testing program is over.

3.3.3 Generation of the Reference Model for Chip-Level Verification

In contrast to block-level reference models that were custom built using a specification document as a guideline, a standardized reference model based on an emulator was used for chip-level verification. In order to build such a reference model, the logic described in Fig. 3.7 was followed. The RV8 RISC-V simulator [12] is used in conjunction with a SV-written custom reference model, validating itself against the simulation results of RV8. The goal was to construct a model capable of predicting results stored into the register bank for each instruction validated on a widely RISC-V emulator, but with the particularities of this DUT, for example, Siwa's custom operations that do not follow the RISC-V standard; specifically, instructions for a smaller set of CSR and a restricted memory map (8kB).

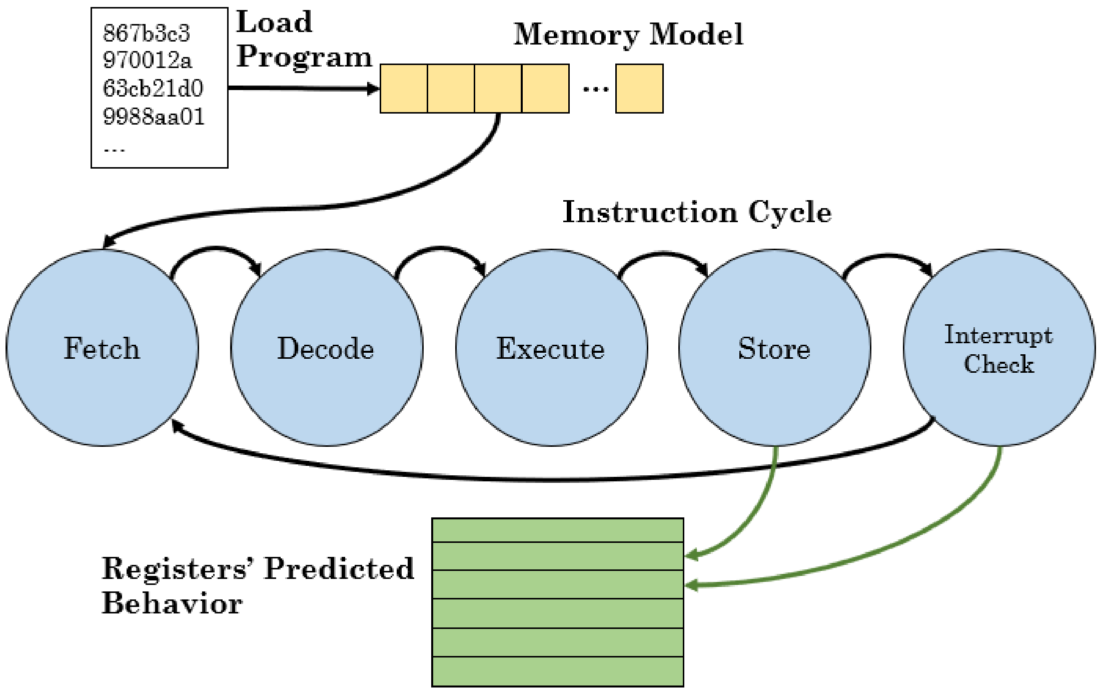


Figure 3.7. RISC-V reference model flow diagram. This model is called before loading the Flash memory. The predictor also helps to determine when a test should end, if the RTL does not reach this point, it means that something went wrong. Each of these steps conforms a typical instruction cycle.

3.3.4 Test Generation

The test generation was handled differently depending on the hierarchical level. All tests for block-level DUTs were randomly generated and then constrained. The verification plan previously designed specifies the necessary tests. For instance, a test for each arithmetic-logic functionality of ALU was implemented for each type of transaction through the UART at different speeds and configurations. Read and write tests for the MBC and tests that emulate data flow traffic through the bus were also created.

However, at the chip-level, the generation of random instructions was more complicated. Instructions must follow the ISA standard, and the program needs a coherent intention to be able to run a proper application. As a result, the tests were completely oriented, and hence the random factor was taken out. The test selection was created based on the verification plan, as such, tests were implemented with the goal of validating each instruction within the RISC-V 32I ISA [13] specification, each I/O port and its intended functionality, each type of interruption, and coverage over the register bank and memory space.

To create suitable programs that may stimulate SIWA properly, we needed an easy way to build the hexadecimal files containing the instructions, and the fastest way is to use the RISC-V toolchain compiler. With it, we can write some test algorithms in assembly code (in C++ language as well) and compile it to generate the hexadecimal file.

Then, to compile an assembly program with the RISC-V toolchain, you typically start by writing your assembly code in a file with a .s extension. The RISC-V toolchain includes tools like riscv64-unknown-elf-gcc or riscv32-unknown-elf-gcc, which can be used to compile the code. You can compile your assembly file by running a command like riscv64-unknown-elf-gcc -o output.elf input.s. This command assembles the .s file into an object file and then links it to produce an executable, typically in ELF format. The output executable can be run directly on a RISC-V processor or used with a simulator like Spike or QEMU. If you need to inspect the compiled binary, you can use tools like riscv64-unknown-elf-objdump to disassemble the executable or riscv64-unknown-elf-objcopy to convert it into different formats. This process enables you to take your RISC-V assembly code from source to runnable binary, making it ready for execution on RISC-V hardware or simulators. For more information, refer to the Sifive toolchain [14, 15].

Fig. 3.8 shows a diagram depicting how tests are handled before handing them over to the verification environment.

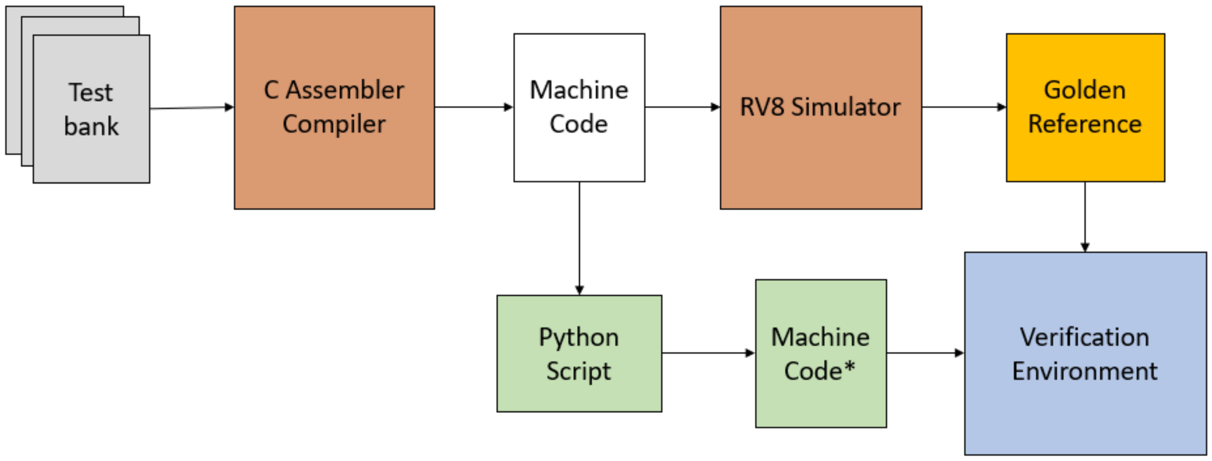


Figure 3.8. Diagram that depicts how tests are handled prior the beginning of the simulation. Hexadecimal machine code marked with a * has been edited with a Python script, in order to match the Flash memory model requirements.

The resulting text file with the program in hexadecimal code was post-processed afterwards via a script to adjust it to how the commercial Flash Memory SystemVerilog model could read it. Then, a simulation is run where the core boots from that Flash memory connected to an SPI (Serial Peripheral Interface) port. And for the rest of the simulation the checking occurs as explained before.

Table 3.2 lists some of the major tests implemented to verify SIWA at chip-level:

3.3.5 Regression System

Regressions in functional verification are similar to test farms, where multiple tests are run one after another in a pseudorandom style using seeds associated to each simulation run. The idea was to quickly create random stimulus for each simulation so that multiple scenarios can be exercised to look for errors. The proposed flow integrated a free regression platform, depicted in Fig. 3.9, based on simple scripting.

Table 3.2. Test list at chip-level. (*)There is one test for each operation on the ISA (add, sub, srl, and, or, xor, etc). (**)There is one test for each data size on the ISA (byte, halfword and word).

Category	Test Name	Description
Logic-Arithmetic	Immediate Operation(*) Test	Immediate operation with random operands.
	Operation(*) Test	Operation with random operands.
Memory	Load Data(**) Memory Test	Load random data into the register file.
	Store Data(**) Memory Test	Store random data from the register file.
	Misaligned Store|Load Test	Load|Store a misaligned transaction.
Control	Jump Test	Execute jumps in the program.
	Branch Test	Execute branches in the program.
	SPI to MBC Test	Packet transaction from SPI to MBC.
CSR	CSR Test	Tests all CSRs on a single program.
	Timer Test	Test interruption timers.
	GPIO Test	Write and Read GPIO CSRs.

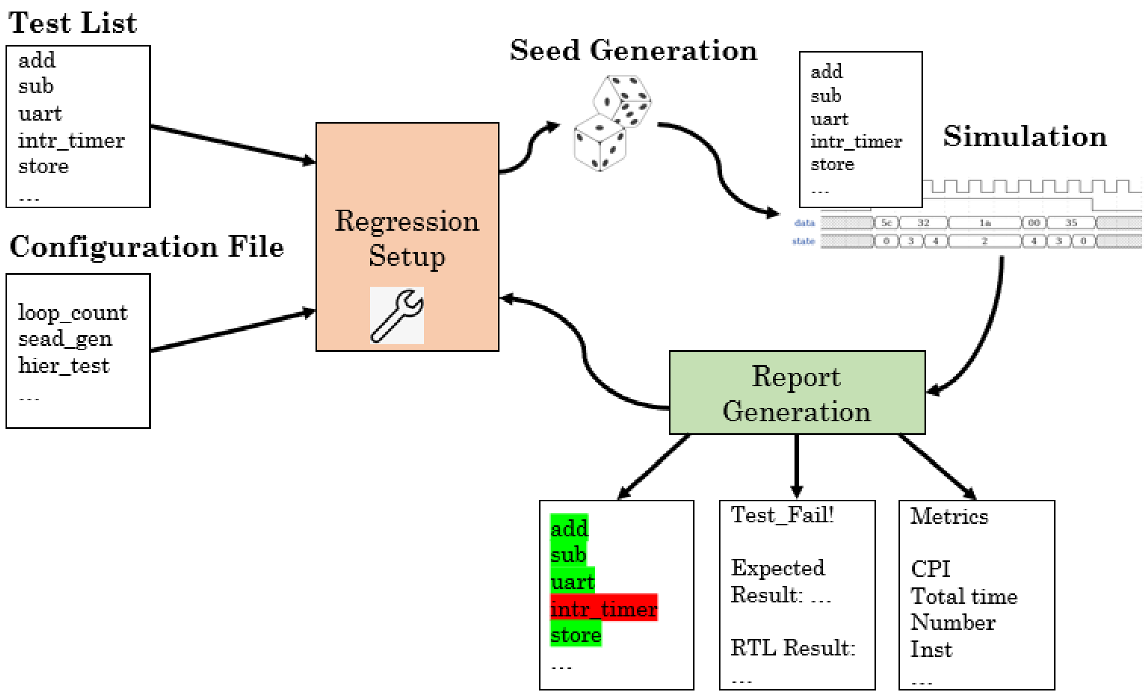


Figure 3.9. General structure of the regression system. Each test is run according to a list of tasks, with a configuration file defining specific aspects of the regression, such as loop quantity and type of coverage collected.

Basic shell scripting was used to code the regression platform following the general structure shown in Fig. 3.9, where the seeds were generated using Python. For oriented tests at top-level, the RISC-V compiler is invoked for test generation using RV8 to create the golden reference construction prior to invoking VCS for simulation and coverage collection. A report is created or updated with the collective resulting status for each individual test. Once a batch of tests is finished, a new batch is prepared and executed using a different seed. The required number of iterations is specified by the user, according to the goal in terms of coverage.

3.4 Overall Results

The proposed verification structure is fully functional and can be replicated to equivalent designs. More than tens of tests were implemented, some of them were run with multiple seeds to cover as much as possible of the space of possibilities, others were directed tests as their randomizations were much more complicated to run. At least five different environments were developed and integrated in the regression platform.

Fig. 3.10 shows how the regression console platform interacts with the user. The user can choose what type of simulations they want to run. An external text file with the list of tests can be modified to include or exclude tests as the user sees fit. The regression can also be adjusted via the configuration file before running the tests as mentioned before. This platform will generate reports to help the user quickly collect information about the verification of the different DUTs. These reports contain information such as: test result status with its respective seed, performance parameters such as execution time or Clock Per Cycle (CPI), comparisons between the predicted and the RTL register bank for each individual test, and coverage data collected from the test battery.

Finally, figure 3.11 shows an example of some coverage results extracted from the Synopsys Verdi coverage tool.

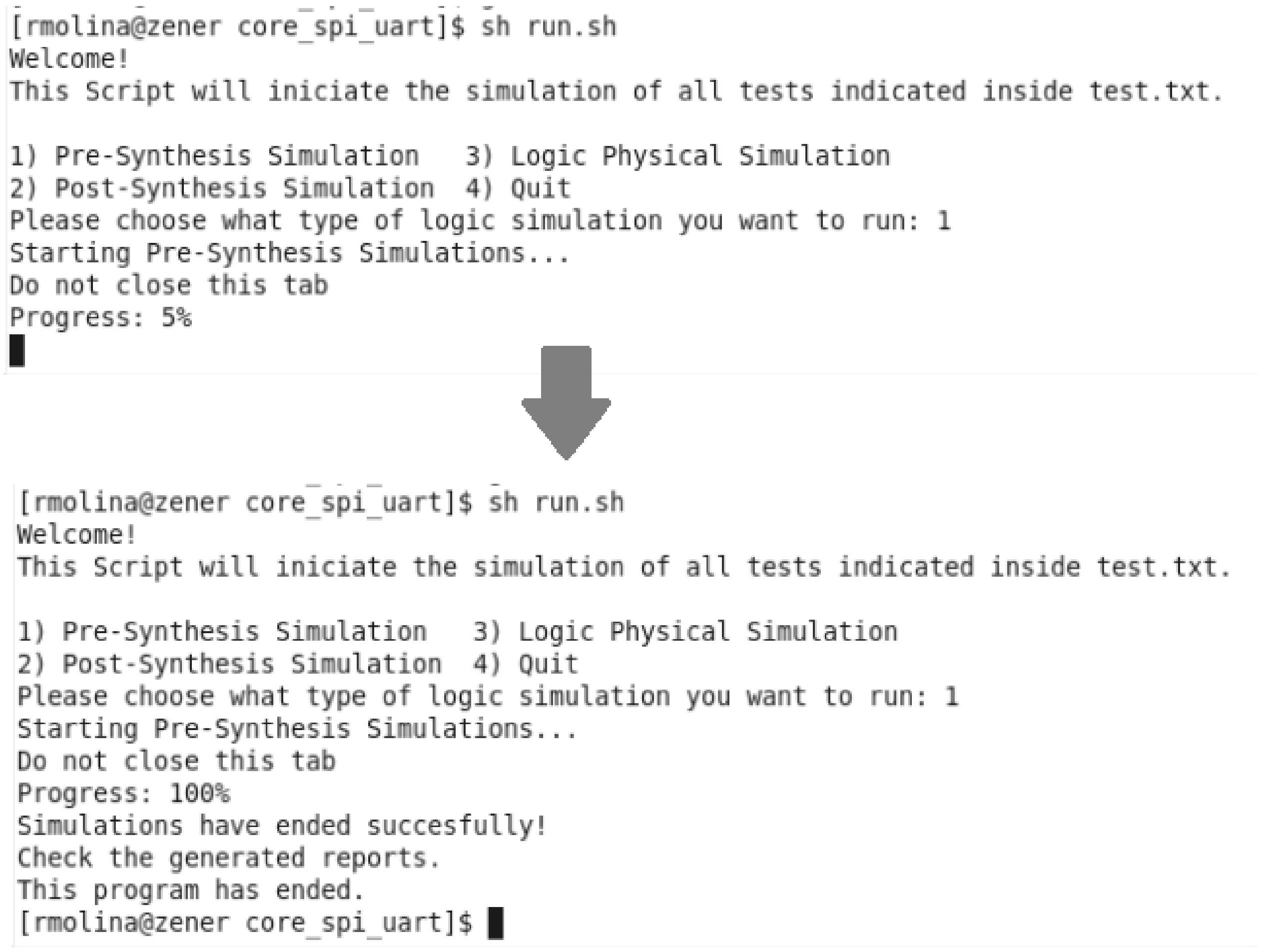
3.5 Upgrading the environment to UVM

The last section of this chapter is related to the update of the chip-level environment to UVM, as it is more used on the design verification mainstream, not to mention, easier to maintain and adapt to new features.

As a standard, Fig. 3.5 serves rightly to illustrate the UVM structure for the new chip-level environment. However, there are a couple of caveats worth mentioning in the migration of the old environment into the new one: what could be reused and what could new features could we add.

Reusing existing code, old tests, and previous systems is always a challenge when updating an existing environment. Fortunately, as the previous environment was structured using a modular approach on SystemVerilog, all the features and tests that we had implemented could, in fact, be ported over, and we did. Fig. 3.12 shows how we integrated UVM in our environment.

We basically rewrote the basic UVM code for each block, converting them into UVM elements. The basic



```
[rmolina@zener core_spi_uart]$ sh run.sh
Welcome!
This Script will iniciate the simulation of all tests indicated inside test.txt.

1) Pre-Synthesis Simulation   3) Logic Physical Simulation
2) Post-Synthesis Simulation  4) Quit
Please choose what type of logic simulation you want to run: 1
Starting Pre-Synthesis Simulations...
Do not close this tab
Progress: 5%
█
```

```
[rmolina@zener core_spi_uart]$ sh run.sh
Welcome!
This Script will iniciate the simulation of all tests indicated inside test.txt.

1) Pre-Synthesis Simulation   3) Logic Physical Simulation
2) Post-Synthesis Simulation  4) Quit
Please choose what type of logic simulation you want to run: 1
Starting Pre-Synthesis Simulations...
Do not close this tab
Progress: 100%
Simulations have ended succesfully!
Check the generated reports.
This program has ended.
[rmolina@zener core_spi_uart]$ █
```

Figure 3.10. Screenshot of the regression console. Several reports are generated, and the execution is scheduled via Linux's Cron.

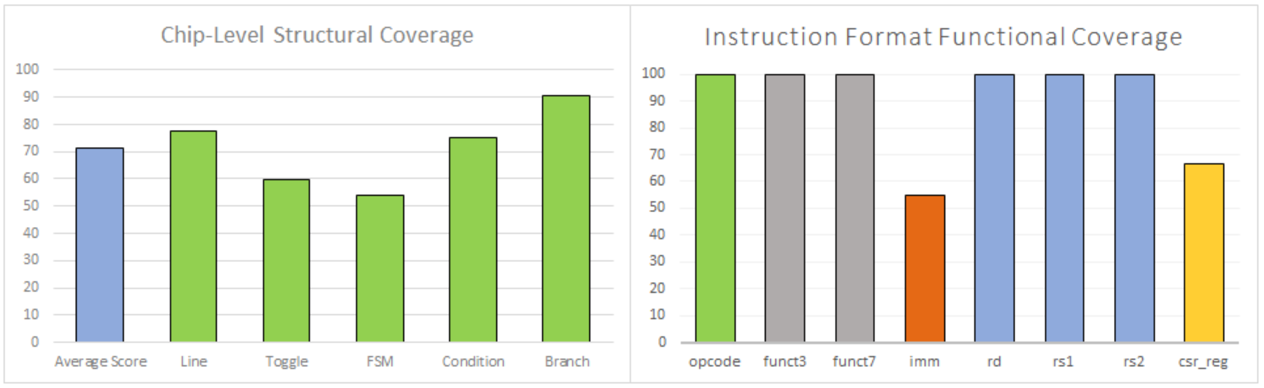


Figure 3.11. Accumulative coverage results obtained with the custom verification environment. The graph to the right shows the functional coverage of the RISC-V instruction formats [13]. The graph to the left depicts the average structural coverage obtained for chip level verification.

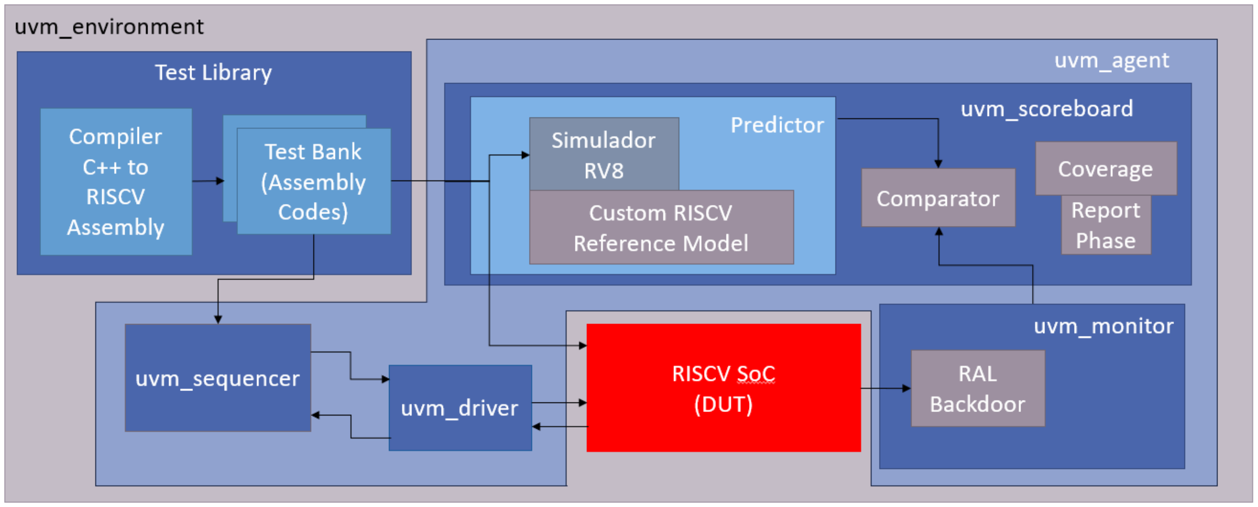


Figure 3.12. Updated chip-level environment to UVM. All elements were rewritten to UVM components and a RAL was integrated as a substitute for the previous backdoor access.

test just generates the clock and a uvm sequence for the reset, since the last one is how the microcontroller knows when to begin booting from the external FLASH memory. The rest of the elements were converted to their UVM equivalents using the standard framework provided by the UVM library.

One big change was how the backdoor was handled. In the original code, we directly access the RTL to monitor the register file each time the control FSM reached the end of the instruction cycle. This made the environment difficult to maintain in future updates, as introducing the complete path at a specific point is detrimental when the RTL faces big internal changes. This was especially true when the environment had to access the register file data in several places throughout the simulation. To alleviate this issue in future updates, we decided to introduce an UVM Register Abstract Layer (RAL) on our environment for the register file, control/status registers and the control FSM. Using the RAL on this scenario allows us to access any of the mentioned elements anywhere in the environment at any given moment, facilitating maintenance on the long run.

3.6 Chapter Conclusions

A compact, affordable functional verification flow has been generated and used for the verification of a small RISC-V 32I based microcontroller. The flow follows a functional verification strategy that includes the development of a test methodology and verification architecture, and is capable of carrying out hierarchical verification, reporting coverage metrics, and performing intensive, randomized regressions.

The flow can be extended to other digital designs and can incorporate alternative tools if required. Future works include adding the option for formal verification tests to the flow, and the possibility of generating random RISC-V coherent programs. This methodology also shows how it is possible to incorporate external elements, such as the RV8 compiler and the C Assembler compiler, as useful components in the verification effort.

Five different environments were developed for five different DUTs with independent functional and structural coverage metrics. These environments are easy to integrate into each other but also limited to the specification of our design. Block-level environments were UVM

A regression console platform was constructed as shown in Fig. 3.10. This platform is capable of generating useful reports to help steer the verification strategy in the right direction. These reports contain information such as: test result status with its respective seed, performance parameters such as execution time or Cycles Per Instruction (CPI), comparisons between the predicted and the RTL register bank for each individual test, and coverage data collected from the test battery.

Finally, this chapter also showed how an old verification environment can be upgraded into UVM standard reusing code, approaches, and techniques without the need to rebuild the environment from scratch.

Chapter 4

An affordable post-silicon testing framework applied to a RISC-V based microcontroller[1]

RISC-V architecture is a very attractive option for developing application-specific systems that require an affordable and efficient central processing unit. The popularity of the RISC-V architecture might be due to its open source nature and its high configurable structure, which allow it to meet the specifications of a wide range of applications. Such features are particularly appealing to a growing number of small design groups using the RISC-V architecture as a stepping stone for their particular goals. However, developing a RISC-V core can still be challenging when trying to ensure proper testing and validation on a budget.

In the semiconductor industry, chips complexity is ever increasing, continuously adding new challenges at every step of the VLSI design flow. For example, RTL design is harder to converge, power consumption is harder to optimize, functional verification is harder to complete, the risk of a buggy product after fabrication is higher, etc. Due to this constant complexity increase, functional verification, as we know it, may not be enough to guarantee an error-free RTL. Therefore, adding new steps to the chip development process could be helpful before fabricating a chip, and in this case, RTL emulation raises its hand.

It is well-known that post-silicon testing is a required step in the chip development process. Post-silicon validation on RISC-V applications has been done in the industry for a while, yet documentation is scarce. This chapter covers a practical low-cost post-silicon testing framework applied to a RISC-V RV32I based microcontroller. The framework uses FPGA-based emulation as a cornerstone to test the microcontroller before and after its fabrication. The platform only requires a handful of elements such as the FPGA, a PC, the emulated or fabricated chip, and some discrete components. The main idea was to validate the design under test and save development testing time in different steps by using a reuse philosophy.

As mentioned above, increasing complexity of digital systems and their tight development schedules place strict demands on proper design validation. Moreover, simulating extracted post-place-and-route models can not always detect timing-related structural errors. Industry's post-silicon validation environments are often custom-based to the characteristics of their already closed design-under-test (DUT), and most of them are

[1]This chapter contents have been published verbatim in [16]

very expensive to replicate. FPGA verification of RTL designs provides a faster avenue for the debugging of architectural problems, particularly in microprocessor-based designs, when thorough testing implies the execution of software tests on them as well. Without much financial and human resources, we decided to focus on the post-silicon validation efforts of our RISC-V core, employing FPGA as the testing platform. Additionally, we decided to re-utilize what we could of the functional verification environments' elements used on previous steps of the design flow to save time and resources.

There are several publications related to the design and use of RISC-V based processors for specific applications. For example, [17], [18], [19], [20], [21] and [22], describe their designs or how they used their respective RISC-V-based processors or microcontrollers to address a particular problem. Information about pre-silicon functional verification also exists, yet it is scarcer. Publications like [23], [1], [24] and [25] share some of their functional verification flows and techniques applied to RISC-V based processors. The search for publications on specific post-silicon testing strategies of any RISC-V-based microcontroller was more difficult. For example, the authors of [26], focus their paper on the integration and evaluation of DFT modules on a RISC-V SOC, but without addressing the general post-silicon testing framework.

Regarding FPGA for emulation and post-silicon validation, there is plenty of literature on alternative approaches, such as [27], where a test generator for microcontroller on-silicon validation was proposed. In [28], a post-silicon method was presented to validate memory consistency in multiprocessor systems. The authors of [29] implemented an interesting framework on a FPGA to try to achieve observability similar to that in functional verification phases by adding some scan cells to a specific design. Other works, such as [30], [31] and [32], emulated different types of hardware piece on FPGAs for testing purposes, too, while [33] and [34] proposed using an FPGA as a low-cost ATE tool. In [35], an FPGA was used as a mean to capture data from a DUT, and in [36] an automated test methodology was proposed to test an FPGA was proposed.

But none of the former publications directly address the particularities of RISC-V post-silicon validation; this may be because of the still ongoing maturation process of the RISC-V toolchains and the standard itself. On the RISC-V community websites you can find some information for pre-silicon verification; however, RISC-V post-silicon validation documentation is rare. The objective of this chapter is to propose a post-silicon validation framework amenable to other small design groups with limited resources, using a RISC-V core as a case study. A secondary objective, perhaps not so technical but still critical for the authors, is to promote a larger discussion on the issue of post-silicon validation in the RISC-V community. Especially as more and more RISC-V designs by small and not so small teams move into the commercial arena, with the consequential need for stronger, standardized post-silicon validation flows.

In this chapter, Section 4.1 briefly describes the interfaces incorporated on SIWA just to illustrate what was implemented and what we had at our disposal for the post-silicon tests. Section 4.2 explains the strategy for emulating SIWA on an FPGA and how we developed a physical framework that could help us test SIWA once it has been fabricated. Section 4.3 contains information on the tests we exercised on SIWA using the described framework. Finally, Section 4.4 discusses the conclusions of the chapter and discusses future improvements.

4.1 Summary: The RISC-V core interfaces overview

Figure 4.1 describes the interfaces of the RISC-V based microcontroller that is to be tested.

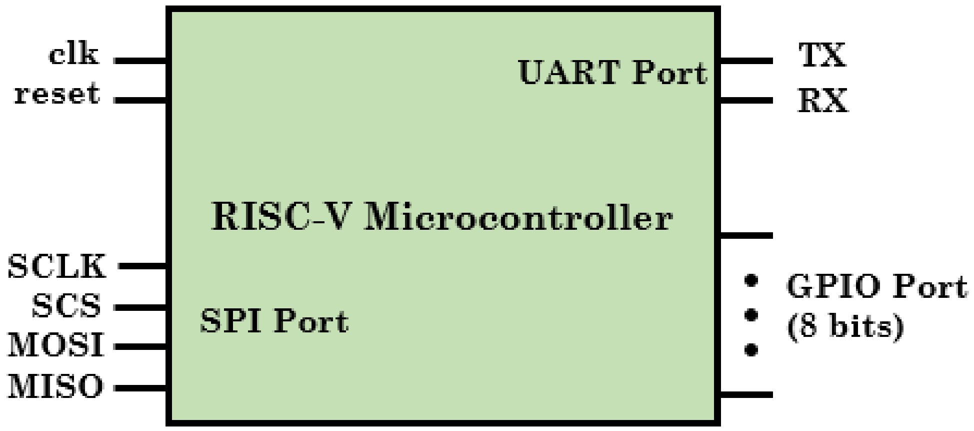


Figure 4.1. Chip-level description of the RISC-V microcontroller's interfaces.

As illustrated, the microcontroller has three main interfaces: a Serial Peripheral Interface (SPI) port [37], a Universal Asynchronous Receiver-Transmitter (UART) port and a General Purpose Input/Output (GPIO) port. The design intent is to connect the microcontroller to a flash memory through the SPI port.

From those three, the SPI port is in charge of handling an external flash memory, where the microcontroller's bootstrap routine is stored. After booting up, the flash memory serves as a secondary data memory. The UART interface is a simple port that uses a standard 19200 baud rate, with 8 data bits, 1 start bit, 1 stop bit, and even parity. The main purpose of the UART is to be able to communicate with the processor from a PC. GPIO ports are used to control displays connected to the processor or to receive data from sensors.

4.2 Emulation on FPGA: bootloading test programs

The microcontroller testing process was divided into phases. Each phase represents a specific moment inside the project development schedule. Figure 4.2 shows the order in which the proposed phases were executed.

The "Functional/Formal Verification" and the "Post-Layout Validation" phases are considered pre-silicon testing, and as such, they will not be covered in this article.

Now, because testing time is one of the bottlenecks inside the VLSI design flow, we strove to re-utilize code and implementations as much as possible between steps, while trying to debug one feature at a time to try to isolate hidden bugs.

The first step on our post-silicon testing was to define "how" were we going to test the microcontroller.

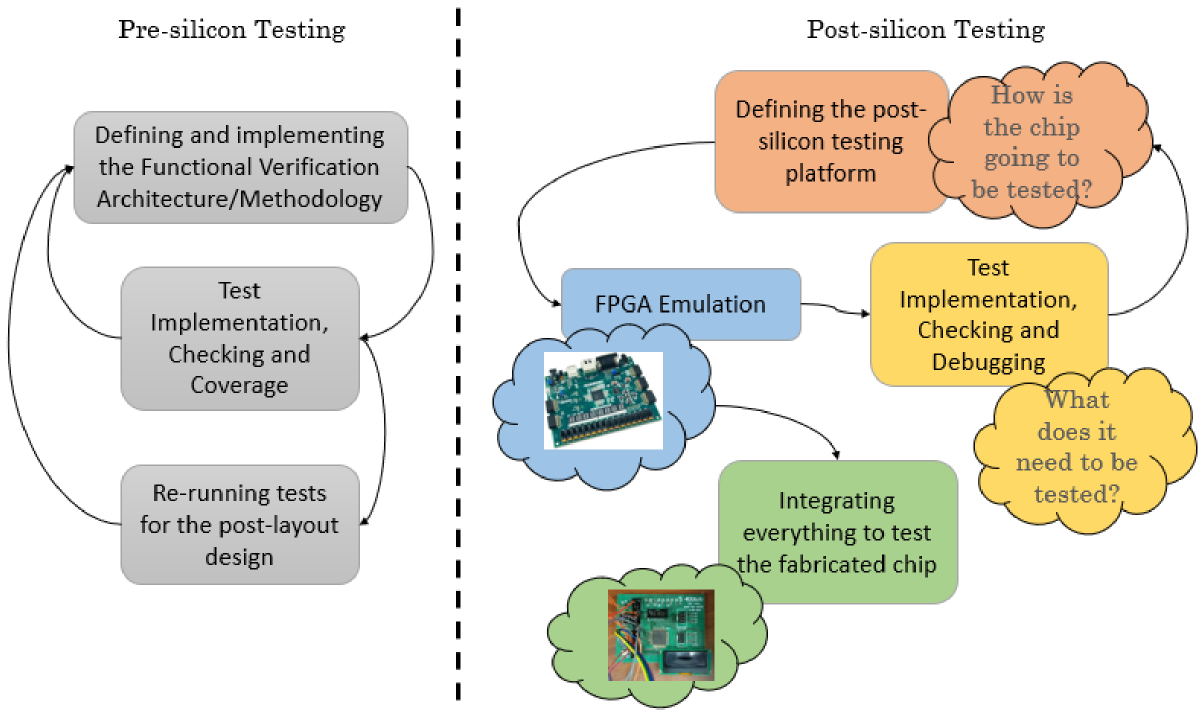

Pre-silicon Testing
Defining and implementing the Functional Verification Architecture/Methodology
Test Implementation, Checking and Coverage
Re-running tests for the post-layout design
Post-silicon Testing
Defining the post-silicon testing platform
How is the chip going to be tested?
FPGA Emulation
Test Implementation, Checking and Debugging
What does it need to be tested?
Integrating everything to test the fabricated chip

Figure 4.2. Testing methodology used to test the RISC-V microcontroller.

The following phases can vary depending on the needs of the project. For our case, we decided to emulate the microcontroller and flash memory on a FPGA, using it as a support to test the final chip that was going to be integrated along with some displays over a PCB.

The FPGA emulation is not only useful to design the post-silicon tests for when the resulting fabricated chip comes back from the foundry, but it also helps as a complement to the functional verification phase. As a matter of fact, this was particularly helpful in identifying structural bugs that have been overlooked in previous phases of our project, and they were spotted when implementing the design on the FPGA.

Our goal with FPGA emulation was to try to replicate the whole testing environment as if the chip had already been fabricated and was ready for testing. Consequently, it was necessary to emulate the flash memory and find a way to ensure that the tests were working as intended. However, observability during post-silicon phases is minimal, even more so when no DFT modules are included inside the design. Hence, the only way to see whats happening inside was using peripherals connected to the UART and GPIO ports as indicators. Figure 4.3 shows the general idea of this strategy.

As depicted, a standard PC was enough for the testing platform. The idea was to employ the PC as a USB/COM port receptor for the UART after implementing the design inside the FPGA.

The microcontroller was an independent, easy-to-find module inside the top level of the testing architecture. In this way, the FPGA can easily be incorporated as a support driver/receiver into the final testing framework by removing only the microcontroller module from the FPGA implementation and setting the SPI Port as an FPGA I/O connected to the fabricated chip, thus saving testing development time.

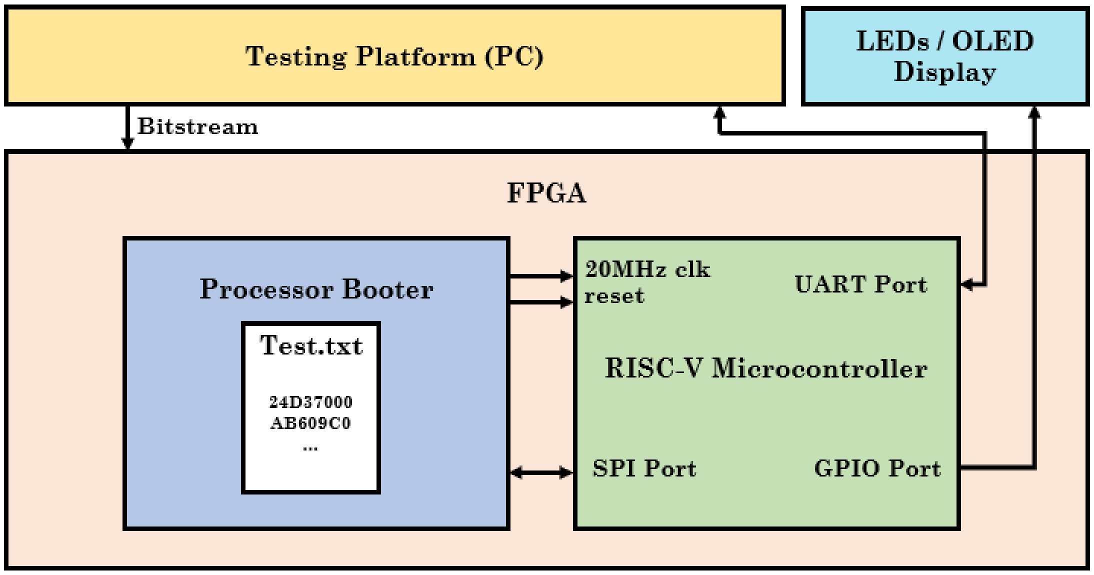


Figure 4.3. Testing concept for the developed RISC-V microcontroller synthesized inside a Nexys 4.

To emulate flash memory, two major RTL blocks were proposed to implement its functionality as shown in Figure 4.4. The described boot loader module must emulate the flash memory, generate the microcontroller's clock and control the bootstrap routine of the processor by clearing its reset input signal.

The test programs used on post-silicon phases were previously validated, compiled, and linked in the needed HEX format on the microcontroller's functional verification environment, saving more development time. These programs were loaded into the instantiated flash memory emulator during the FPGA synthesis stage.

The SPI interface controller is a synchronous block that acts as "slave" and receives commands from its "master", the microcontroller. After that, it performs an action according to the code received. This communication must match the functionality of the selected commercial flash memory. The commands we needed were those related to read and write operations, which can be found here [38]. Figure 4.5 illustrates a command sent by the RISC-V microcontroller as it starts to boot up.

The SPI interface module had to capture this information. If a read command came in, it should prepare the corresponding data to be sent to the core, as depicted in Fig. 4.6.

To achieve the SPI functionality, a finite state machine was implemented. This FSM was created under the intention that the SPI interface module could have two purposes: to monitor and decode received commands through the MOSI input, and to execute the captured command using the MISO output. Figure 4.7 shows the state diagram of this FSM. Inside the MOSI_mode state, the SPI controller waits for the microcontroller's command. Otherwise, inside the MISO_Mode state, the SPI controller should resolve the command previously received.

Auxiliary counters and shift registers to capture the command, a combinational module to decode the command, a counter to update the internal memory address, a clock generator using the FPGA 100 MHz

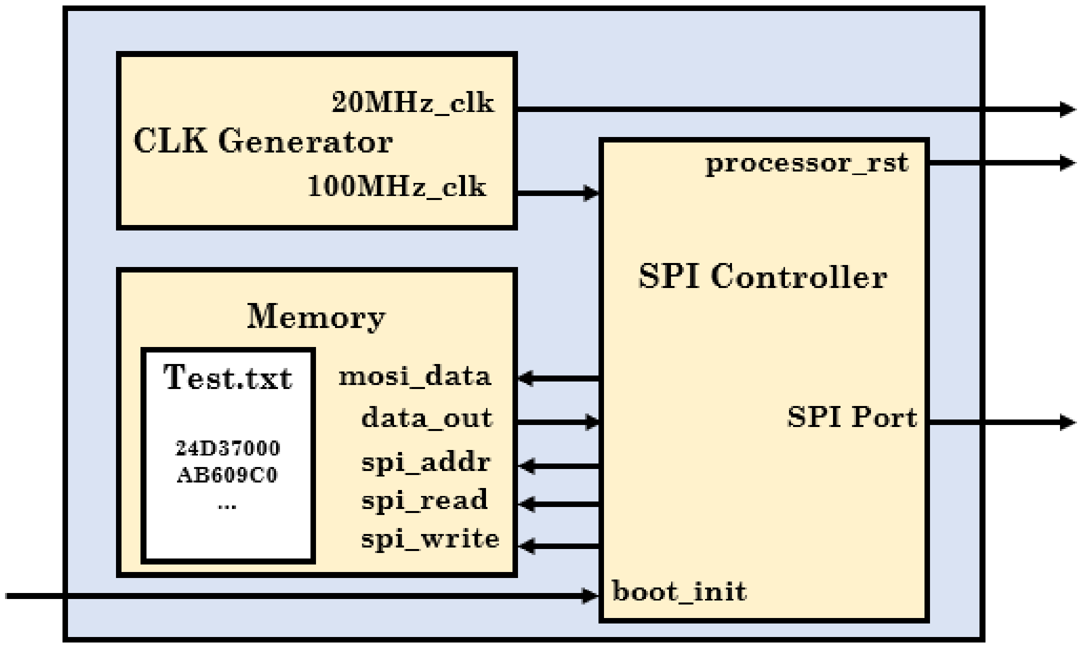


Figure 4.4. Block-level micro-architecture of the boot loader module.

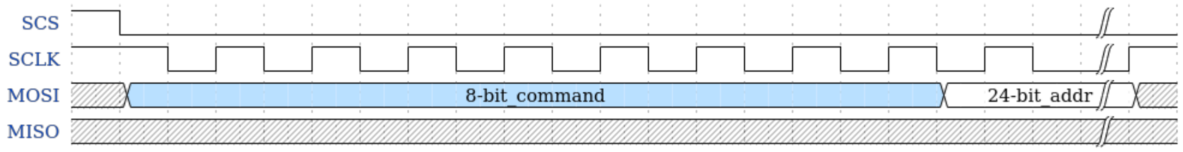


Figure 4.5. Timing diagram of a command sent by the microcontroller through the SPI port.

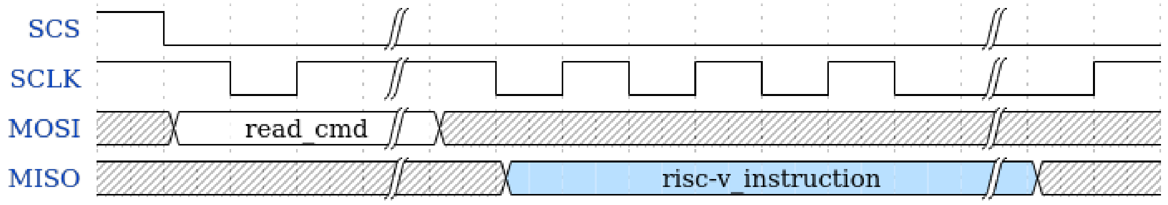


Figure 4.6. Timing diagram of the boot loader response to a read command.

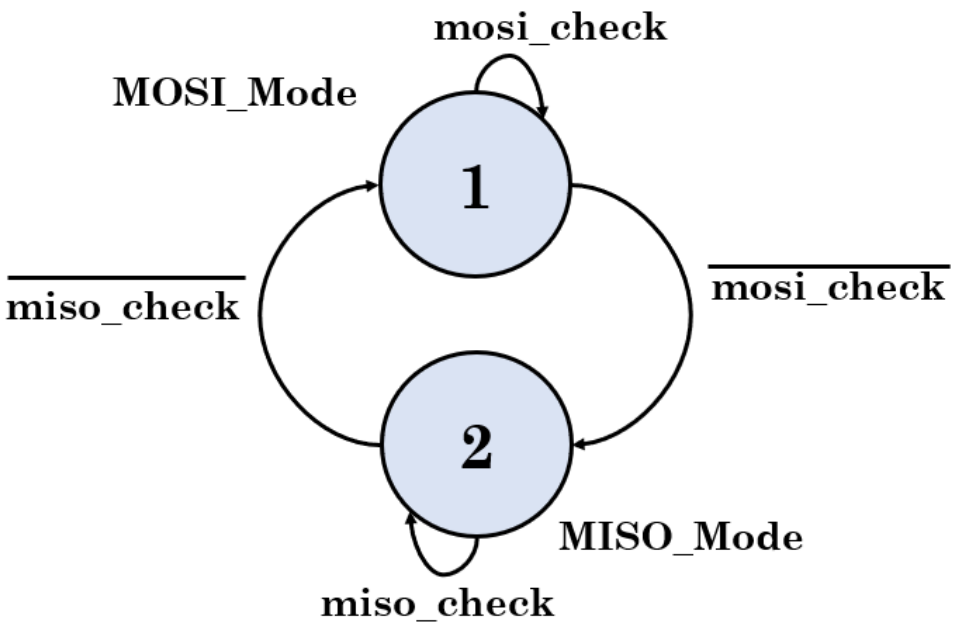

mosi_check
MOSI_Mode
1
$\overline{\text{miso\_check}}$
$\overline{\text{mosi\_check}}$
2
MISO_Mode
miso_check

Figure 4.7. SPI Controller FSM. State change evaluation is triggered with SCLK positive and negative edges.

clock as source, etc; were ther digital blocks needed to achieve the emulation. Other important note is that The SPI Controller's frequency was set at 100 MHz, while the microcontroller's frequency was set at 20 MHz by our specification. As a result, the FSM and counters are triggered with the slower SCLK line generated by the microcontroller. Because of this, both positive and negative edges were needed within the FSM functionality to avoid metastability. Figure 4.8 shows the way both edges of the SCLK were interpreted. During the positive edges, the master reads the current data on the MISO line and, at the same time, the following data bit was requested to the internal memory. On negative edges, the requested bit is set on the MISO line, ready for the next positive edge. Edge detectors where needed to identify the corresponding edge.

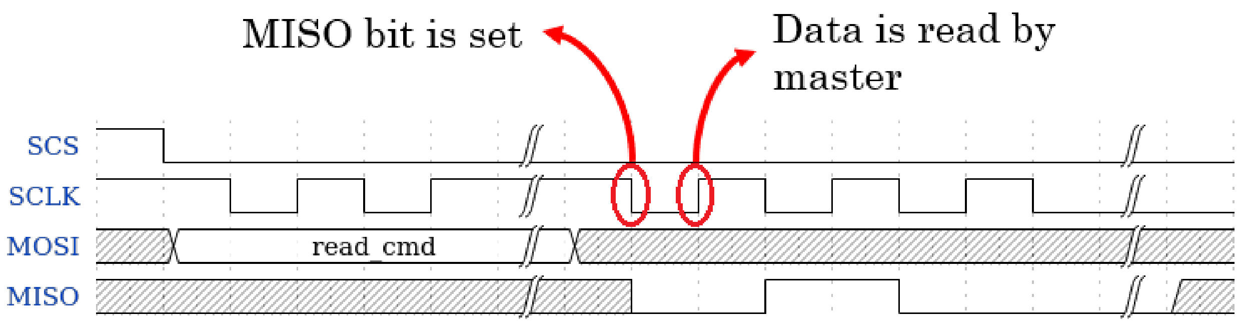


Figure 4.8. FSM SPI Controller functionality triggered by positive and negative edges.

4.3 Post-Silicon Test Elaboration and Chip Testing Phases

The most fundamental features we considered to be tested were: the characteristics related with the compliance of the microcontroller with the chosen RISC-V instruction subset (RISC-V RV32I), that interruptions behaved as intended by our specification, that all the I/O ports worked as expected by their respective protocols, that data could be stored and loaded from the internal memory and registers, and to ascertain how much power consumption did the chip need. Most of the required assembly code for these tests were already available from functional verification.

Because of observability, the first thing that had to be tested were the needed I/O ports working as specification checkers. For instance, developing programs that turn on and off LEDs and may use the internal timer to toggle, and "Hello World" programs that sent a message through the UART port to the PC. For UART monitoring, we used RealTerm or LabVIEW on the PC.

Once the I/O ports were tested to some degree, we were ready to validate features that without DFT were difficult to test, like, register bank checking, memory checking, interruptions checking, etc. We also converted some of the pre-silicon tests into post-silicon self-checking tests. The same test program checks its internal results and uses a LED or a generated message as indicators to the user that the test has passed or not. In this way, we tested that the microcontroller could run the complete RISC-V RV32I instruction subset and other internal features already mentioned.

Once the chip was fabricated, it was integrated into a PCB subsystem with LEDs, an OLED display, an oscillator, the flash memory, and circuitry for voltage supply. Separate supply rails allowed for the independent power measurement of each voltage domain inside the microcontroller: the processor core (1.8 V), memory (1.8 V) and pads (3.3 and 1.8 V). Multiplexers were added to allow the use of reset, clock, and bootstrap signals either from the FPGA or from the discrete components in the PCB.

The microcontroller was successfully tested on the PCB re-using the post-silicon tests developed so far on the FPGA emulation, with little adjustments on the testing environment (just removing the DUT from the FPGA implementation). Nevertheless, two more types of test were developed for this chapter: a test for the emulation of an I^2C interface via the GPIO ports and a bank of continuous tests to measure the microcontroller's power consumption. For the I^2C test, a "Hello World" assembly program is written on the OLED display. The program emulates the signaling required to handle the OLED's I^2C protocol, using around 4 KB of the internal memory (our internal memory size was 8 KB). Internal interrupts and timers were used to generate the I^2C clock, and its functionality was emulated with an algorithm following a designed state diagram. Figure 4.9 shows the microcontroller writing on the OLED display.

The last group of tests were designed to loop the microcontroller, while performing specific operations to measure the average consumed current. Estimates of power consumption per instruction or per type of instruction were generated, as a result, we obtained an average power consumption of 1,67 mW. The idle current consumption was also obtained by measuring the current while maintaining the boot signal high, so that the microcontroller could not boot up, and in this state it consumed 1,3 mW. Finally, static power consumption was estimated by measuring the current when the microcontroller was powered on, without an active clock, resulting in 36 nW power consumption. From simulations we have identified unexpected power consumption on certain modules inside the microcontroller, and that is why we strive to reduce significantly

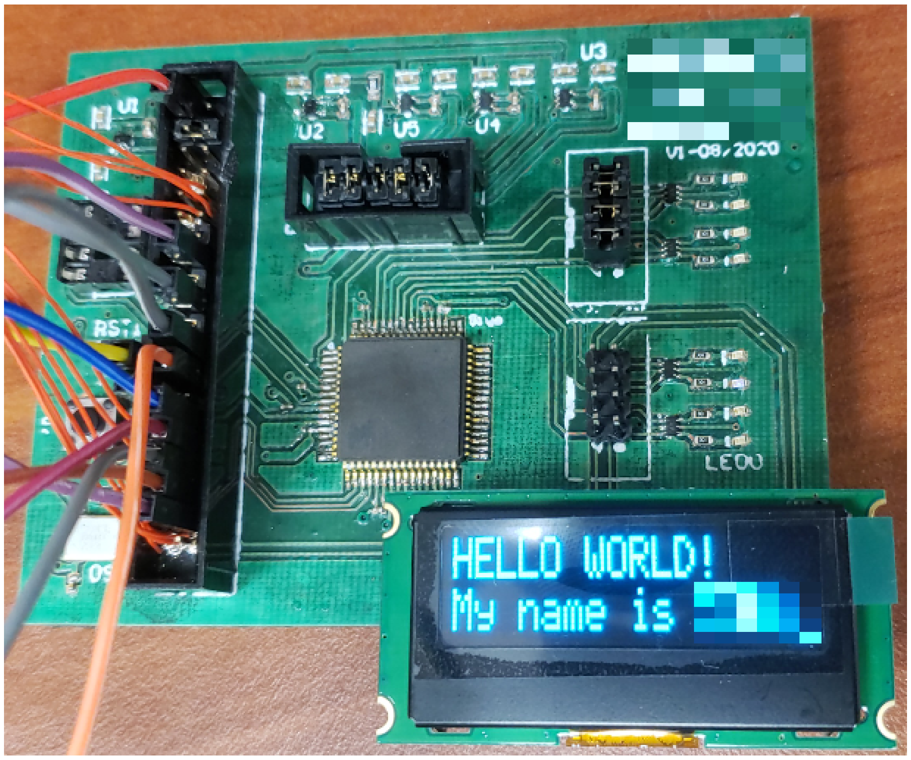


Figure 4.9. "Hello World" program on a OLED display, emulating the required I^{2}C protocol via software routines and the GPIO ports.

the power consumption when computing on subsequent versions of the microcontroller.

Summarizing the resources used in this framework, the only physical components needed to replicate it are a FPGA, a PC for monitoring, the fabricated chip (and PCB if wanted) and some lesser display/support components. On the software side, most of the software used is free. In addition, the time needed to develop tests can be greatly reduced if a recycling code approach is used, such as the one used in this article. All, without losing the capacity to test exhaustively the RISC-V microcontroller.

4.4 Incorporating a FLASH memory as the bootloader

FLASH memories are a type of nonvolatile storage that retains data even when the power supply is turned off, making it a vital component in various electronic devices, from smartphones to embedded systems. It works by trapping electrons in a floating gate structure within each memory cell, enabling the storage of binary data as electrical charges. Unlike traditional hard drives, FLASH memory has no moving parts, which not only makes it more durable but also allows for faster data access speeds. Its compact size and low power consumption make it ideal for use in portable devices, where energy efficiency and space are critical considerations. Additionally, FLASH memory can be erased and reprogrammed electrically, providing the flexibility to update or modify stored data without physically removing the memory module.

In embedded systems, FLASH memory plays a crucial role in the storage of firmware, bootloaders, and application code, ensuring that the system can initialize and operate correctly after a power cycle. Its in-system programmability allows developers to update the system software without needing to replace the hardware, which is particularly useful for devices deployed in the field. FLASH memory also supports a variety of use cases, from storing user data in consumer electronics to logging critical sensor data in industrial applications. However, while FLASH memory offers many advantages, it has limitations, such as a finite number of write/erase cycles and slower write speeds compared to other forms of memory like SRAM or DRAM. These limitations require careful management and optimization, particularly in applications that require frequent data updates or long-term reliability.

SIWA is no exception to this, and eliminating FPGA dependency in favor of using an actual FLASH memory as a bootloader will be one step closer to using SIWA as it is intended. The goal is to replace figure 4.4 to something more likely to be a real application as shown in figure 4.10.

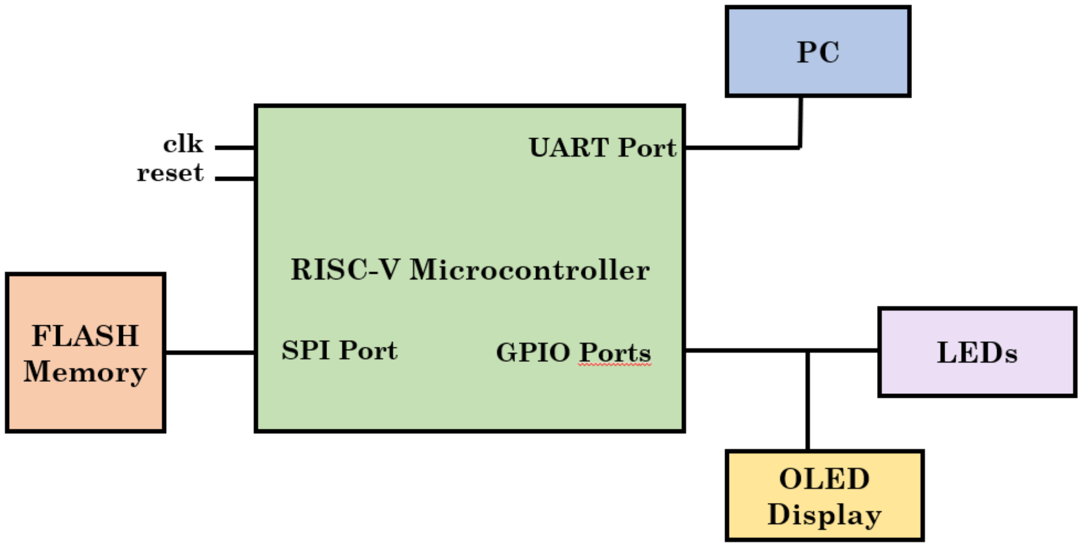


Figure 4.10. Block diagram depicting the connection between SIWA and other physical components.

By this point, SIWA has been proven to be capable of booting from an SPI device, as explained in figures 4.5, 4.6, 4.7 and 4.8. Therefore, only two considerations are needed to accomplish the integration of SIWA with a FLASH memory: to choose a real FLASH memory and to program it with the desired test code. The former consideration was easy, as the only hard requirement is for the FLASH memory to use SPI as the communication protocol between the master (SIWA) and the slave (memory). Winbond W25Q32JV was selected for its voltage level, communication protocol, and the availability of its SystemVerilog model. The latter consideration is also straightforward but requires more work as it needs a test program compiled in a hexadecimal file and a way to program the FLASH memory.

The test program creation and compilation process is a little more extensive, yet this step is similar to what was done in Chapter 3. The only difference is that, this time, the tests will use the microcontroller I/Os to communicate with the external measurement equipment to physically validate SIWA.

Let us first focus on how to program FLASH memory. It is possible to implement a FLASH programmer using an FPGA, but time is most valuable when operating with limited resource. As such, saving time on mundane tasks can have a big impact on the development of the larger project, as time can be used on other critical works. FLASH programmers can be found for a small price on the market, so, the CH341A FLASH Programmer was bought as the means to program our memory FLASH. Figure 4.11 shows a picture of the selected FLASH programmer.

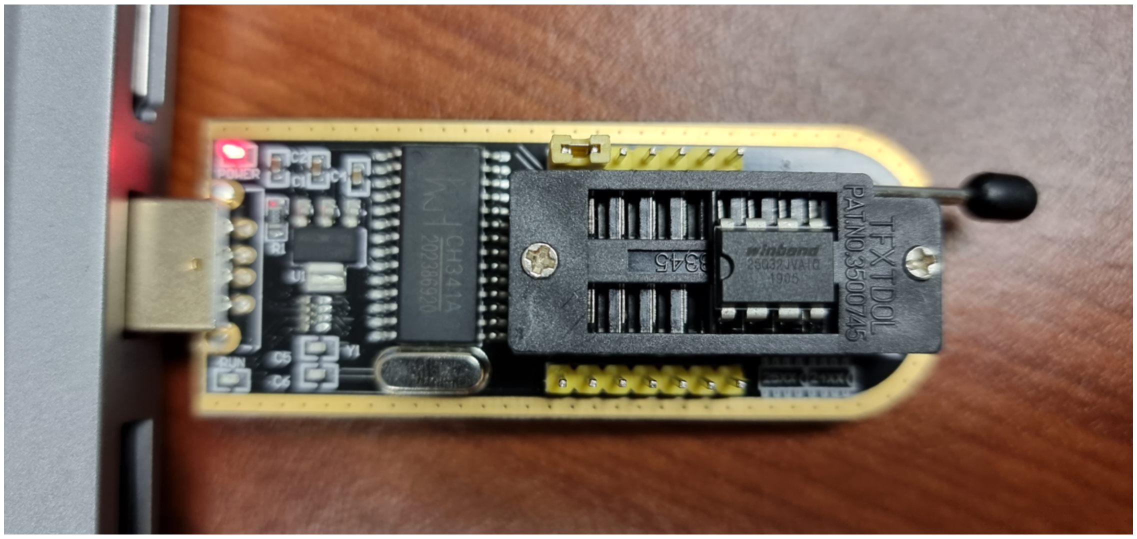


Figure 4.11. FLASH programmer.

After programming the FLASH memory, it is then connected to SIWA to begin its bootloading without using an FPGA. In this way, the transition from emulating SIWA and its surrounding elements on FPGA to using real physical components along with the fabricated SIWA is complete. Figure 4.12 illustrates the integration of SIWA with other discrete components.

4.5 Chapter Conclusions

This chapter proposed a low-cost post-silicon validation framework to test a RISC-V RV32I microcontroller using a FPGA for emulation and support as a study case. This framework shows that it is possible for small design teams in the RISC-V community to take advantage of modern, standardized verification and post-silicon techniques at a low cost. The chapter has included a detailed description of the tested methodology employed, some architectural information about design and implementation of the testing framework, and integration of all the components along with experimental results, using the process of testing this DUT as a concept example to validate the methodology. Our roadmap for future works includes: adaptation of pre-silicon and post-silicon testing environments to newer versions of the microcontroller, to develop software applications to command on-the-fly the microcontroller from a PC and polishing the current framework to improve delivery times.

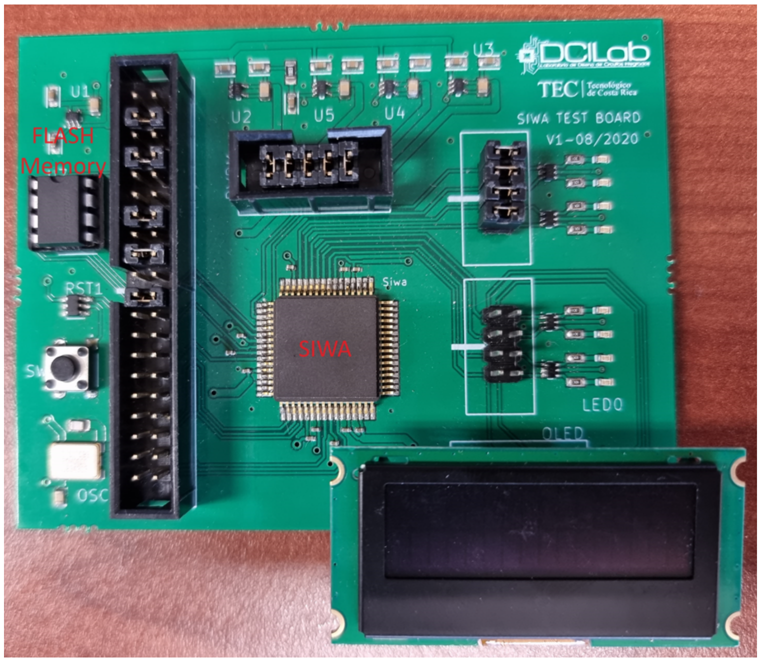


Figure 4.12. SIWA on PCB with other discrete components.

Chapter 5

Low-level algorithm for a software-emulated I^{2}C I/O module in general purpose RISC-V based microcontrollers[1]

The testing of microchips is not limited to exercising the RTL and running a basic stimulus on the fabricated chip. Running applications similar to what is expected under a real scenario is also paramount to guarantee correct functionality up to a certain degree. In our case, since we were developing a RISC-V microcontroller, creating a software application that could prove that our chip works as intended was a necessity at this point. We decided that manipulating a LED display using SIWA would be sufficient and visually attractive.

The aim of this chapter is to explain how it was technically possible to control a LED display, without having implemented an I^{2}C port on SIWA, through emulation using its GPIO ports. This way, not only we are showing the capabilities of SIWA to control other devices, but we can test thoroughly that SIWA is able to run a more complex piece of code akin to drivers used for protocols not implemented physically.

I^{2}C (Inter-Integrated Circuit) is a communication protocol [40] that has been used continuously throughout the years in digital applications since its conception. In today's applications, I^{2}C is widely used in communications, especially for microcontroller applications where several I/O devices need to be driven. Fields like the medical industry, space technology, public security, automation controls, heterogeneous networks and telecommunications are fine examples where I^{2}C is constantly employed. Hence, many microcontroller design teams considerate using I^{2}C to drive peripheral devices. However, developing a microcontroller from scratch is a daunting task, where many difficult decisions and trade-offs occur at every step. In VLSI design, power consumption, development time, cost and chip size are topics of interest that usually affects the quantity of features and components that can be added to a final product. As with any other hardware element, if an I^{2}C controller is added to a chip, all the aspects mentioned before are negatively impacted. In these kinds of challenges, industry and academic researchers are always on the lookout for alternative solutions that can save on chip size and other resources.

[1] This chapter contents have been published verbatim in [39]

Designed for low-speed, short-distance communication, I2C is particularly valuable in systems where multiple peripherals, such as sensors, EEPROMs, and microcontrollers, need to communicate with a central processor. Its two-wire interface, consisting of a data line (SDA) and a clock line (SCL), significantly reduces the complexity and pin count compared to other communication protocols, making it ideal for compact devices with limited pin availability. I2C also supports multiple devices on the same bus through unique addressing, allowing for scalable designs without the need for additional wiring. Furthermore, the protocol's ability to manage both master-slave and multimaster configurations adds flexibility, making it suitable for a wide range of applications, from consumer electronics to industrial systems. Its robust design and ease of implementation have cemented I2C as a go-to choice for inter-device communication in modern electronic designs.

Hardware emulation is a technique that has been around in electronics design for several decades, and we could say that it consists in using a hardware device to mimic the functionality of another, whatever the goal. For example, [41] uses hardware emulation in a traffic network for "Internet of Things" applications. [42] brings a hardware software co-designed solution for a decoder of an x86 emulated system. In [43] it is described that hardware emulation on FPGA can be used to improve testing methods, and in [44] hardware emulation is used to measure and characterize network architectures. We have pondered for some time about the idea of using software-emulated hardware solutions to reduce the final chip size and the development time of our project: [21], [22], [16] and [1]. As such, we decided to emulate by software an I^{2}C controller using GPIO (General Purpose Input-Output) ports to be able to manipulate I^{2}C external devices without implementing the I^{2}C controller hardware into the microcontroller.

Hardware emulation is crucial when testing new chips because it provides a highly accurate and scalable environment to verify and debug complex designs before fabrication. Emulators mimic the behavior of the chip hardware in real-time, allowing engineers to identify and resolve design flaws that might not be detectable through software simulations alone. This process is especially important for modern chips, which often feature billions of transistors and intricate interactions between various components. By using hardware emulation, developers can test the chip's performance, validate its functionality, and ensure compatibility with other system elements under real-world conditions. This early-stage validation helps prevent costly errors that could arise during manufacturing, reducing the risk of expensive rework or product recalls. Additionally, hardware emulation accelerates the development process by enabling parallel software development and hardware testing, ensuring that the final product meets the desired specifications and performs reliably in its intended applications.

As it stands, hardware emulation is definitely an excellent tool for teams with limited resources, as we usually are on academia. Just fabricating a chip is costly, let alone having to repeat a fabrication because a major bug was found in post-silicon phases. Hardware emulation helps reduce the probability that bugs escape functional verification efforts and make their way to the physical chip.

In looking at the state of the art of I^{2}C bus protocol, one can find solutions about applications of said standard. For example, in [45] and [46] an I^{2}C bus is used to interface I^{2}C peripheral devices to an FPGA. [47] presents an implementation on FPGA of an I^{2}C controller for secure data transmissions. In [48] an arbitrated multi-master multi-slave I^{2}C bus application is shown, and in [49] this communication protocol is used to form a network with actuators, sensors and microcontrollers. All these past papers show proper hardware implementations related to the I^{2}C communication standard. By adding the software component to the search formula, we can find articles like [50], where a software solution is included to build an I^{2}C bus analyzer. In other approaches, [51] uses I^{2}C to collect information from a sensor network using an embedded system with Ubuntu as its operating system. However, there are similar implementations to what we wanted to achieve. For example, Texas Instruments showed on [52] a description of a software-emulated I^2C controller using GPIO ports on a RISC family of microcontrollers. Microchip and Intel presented their software-emulated I^2C approaches in [53] and [54], respectively, in the corresponding assembly language for their chips. The implementation of our I2C controller emulated by software is similar to these last approaches, with the caveat that our application had to be written for our own RISC-V microcontroller, with a restriction on memory size since our main memory could hold up to 8 *KB* in total.

This chapter proposes an alternative software algorithm for microcontrollers to emulate the I^2C communication protocol using incorporated GPIO ports, as a way to replace the I^2C controller hardware implementation. The memory size cost of this algorithm is about 0.5 *KB* and its development time was moved to the post-silicon phases. The development of this application was done in RISC-V assembly code, using the open source toolchain available online for the said architecture [14]. Therefore, section 5.1 explains the software architecture created using RISC-V assembly. Section 5.2 describes how the application was characterized and the results were shown demonstrating a working I^2C protocol. Finally, Section **??** discusses the conclusions of the chapter.

5.1 Description of the Software Architecture and Methods

For our application, the emulation of the I^2C bus protocol had to be done in assembly language using GPIO ports and a limited memory space. Figure 5.1 illustrates the theory behind the concept for our application. If you desire, you can check [21], [22] and [16] for more information about our microcontroller. As mentioned before, in the absence of the hardware controller for an I^2C device, we opted to develop an algorithm, written in RISC-V assembly code, to load the instructions of an LED display from the main memory and send them over to the GPIO ports, effectively emulating the I^2C.

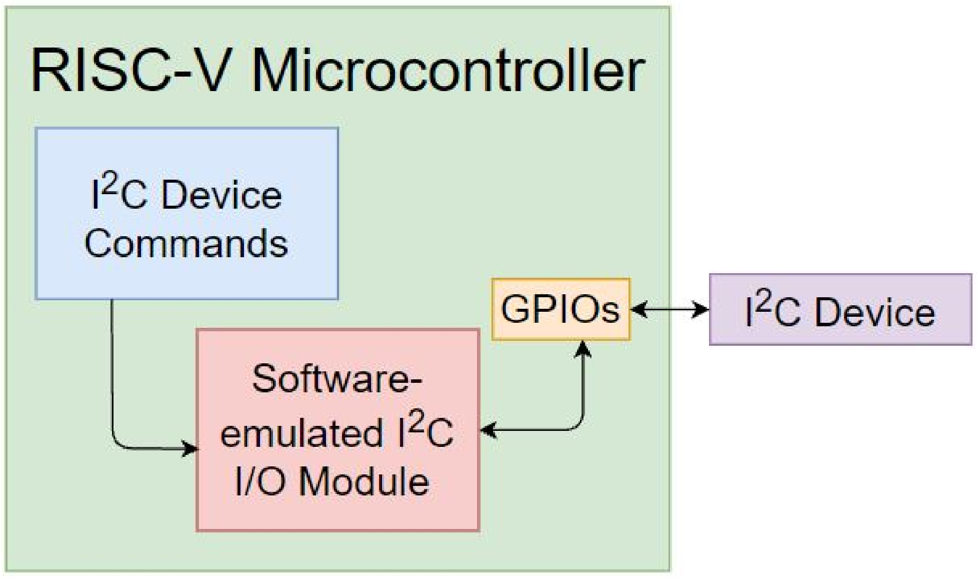


Figure 5.1. Concept idea and motivation for the development of the I^2C emulation application.

The assembly code must be carefully structured in memory to ensure efficient execution and proper functionality. The organization typically follows a defined memory layout that separates different sections of code and data. The instruction code segment, which contains the executable instructions, is placed at a specific starting address to ensure that the processor can fetch and execute the code correctly. The data segment, which includes global and static variables, is usually placed after the text segment, followed by an uninitialized data segment (BSS segment), which contains uninitialized variables. The stack, used for managing function calls and local variables, grows downward from a higher memory address, while the heap, used for dynamic memory allocation, grows upward from a lower address. This structured approach ensures that the code and data do not overlap, which could lead to unpredictable behavior or crashes. Proper alignment and padding may also be applied to meet the architecture's requirements, optimizing access speed, and reducing memory fragmentation. By organizing the assembly code in this manner, developers ensure that the program runs smoothly, efficiently, and within the constraints of the hardware.

A memory stack is a specialized data structure that is used to manage the execution flow, particularly in handling call functions, local variables, and control flow. The stack operates on the Last-In, First-Out (LIFO) principle, where the most recently added item is the first to be removed. When a function is called, its return address, along with any parameters and local variables, is pushed onto the stack as a "stack frame." As the function executes, additional data might be pushed onto the stack, and when the function completes, the stack frame is popped off, returning control to the calling function. This mechanism ensures that each function has its own isolated workspace in memory, preventing interference between different functions. The stack is essential for supporting recursion, nested function calls, and managing the program state in a predictable and organized manner. Due to its dynamic nature, the stack grows and shrinks as needed, but it is also limited in size, so careful management is required to avoid stack overflow, a condition where the stack exceeds its allocated memory space, leading to program crashes or erratic behavior.

In assembly code, interrupts are handled through a mechanism that allows the processor to temporarily stop its current operations and execute a specific set of instructions in response to an event or signal. When an interrupt occurs, the processor saves its current state, including the program counter and other crucial registers into the stack. Then it loads the address of the interrupt service routine (ISR), which is a block of code specifically designed to address the interrupt. The ISR executes, performing tasks such as handling input/output operations, updating data, or managing system events. After the ISR completes its task, the processor restores the saved state from the stack and resumes execution of the interrupted program, ensuring a seamless transition and minimal disruption to the ongoing processes. Each type of interruption needs its own ISR.

Taking all this information into account, the proposed algorithm has a code structure described in Figure 5.2. This picture shows a representation of the code distribution inside the microcontroller's main memory. The code was allocated following the idea of logically separated data and instruction memories. The data memory stores the commands of the I^2 C peripheral device, namely its driver, while the instruction memory holds the actual algorithm that emulates the I^{2}C standard. On the upper part of the main memory, the microcontroller's interrupt service routine is located, and some space can be reserved for a stack if needed. Since physical space and power consumption were important for our microcontroller, the main memory size was limited to 8 *KB*, hence, the totality of this application could not surpass that threshold.

Since this is a low-level software application and very close to the device hardware, a “time" concept was needed to a certain degree. Its algorithm was conceived in a similar way that a FSM (Finite State Machine)

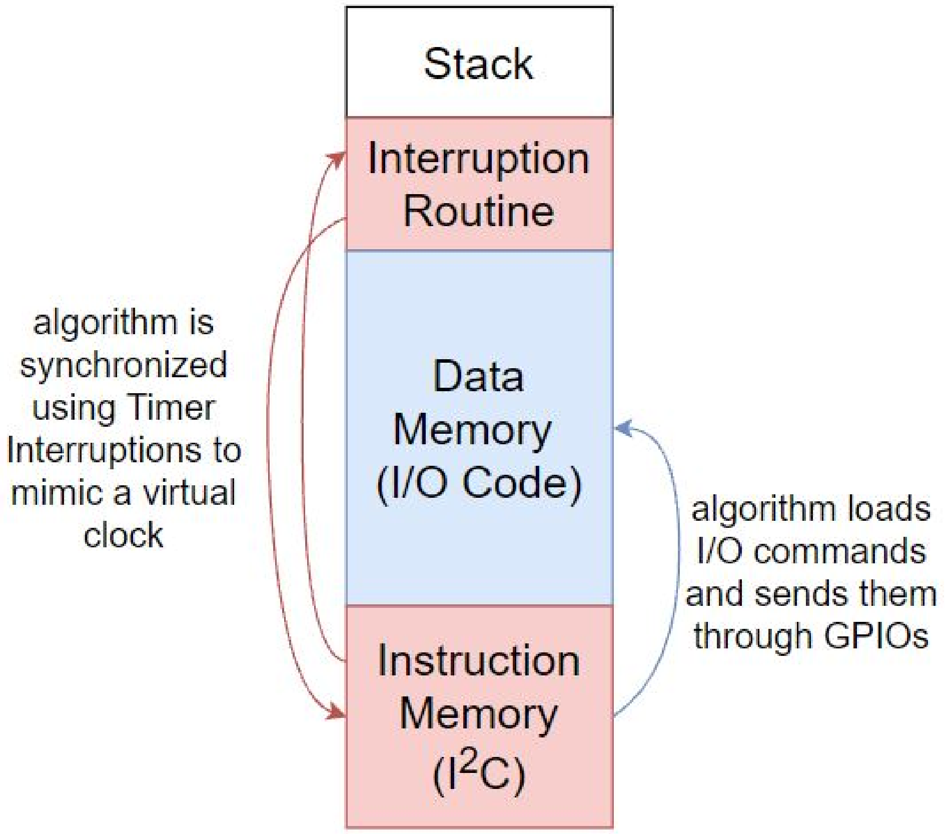

Stack
Interruption Routine
Data Memory (I/O Code)
Instruction Memory (I^{2}C)
algorithm is synchronized using Timer Interruptions to mimic a virtual clock
algorithm loads I/O commands and sends them through GPIOs

Figure 5.2. Code structure of the I^{2}C emulation application written in assembly code. The red arrows represent jumps in the code and the blue arrow represents reads on Data Memory to load the driver commands of the peripheral device to send them over the emulated I^{2}C.

is designed for digital control units. And, like an FSM implemented in hardware, a clock was used to move through the states, although this clock was virtually implemented by software and its frequency could be adjusted to match the different speeds the I^{2}C bus standard permits. This FSM had to replicate a typical I^{2}C transaction. Figure 5.3 shows its graphical representation and Table 5.1 describes briefly what each of the states does in this algorithm.

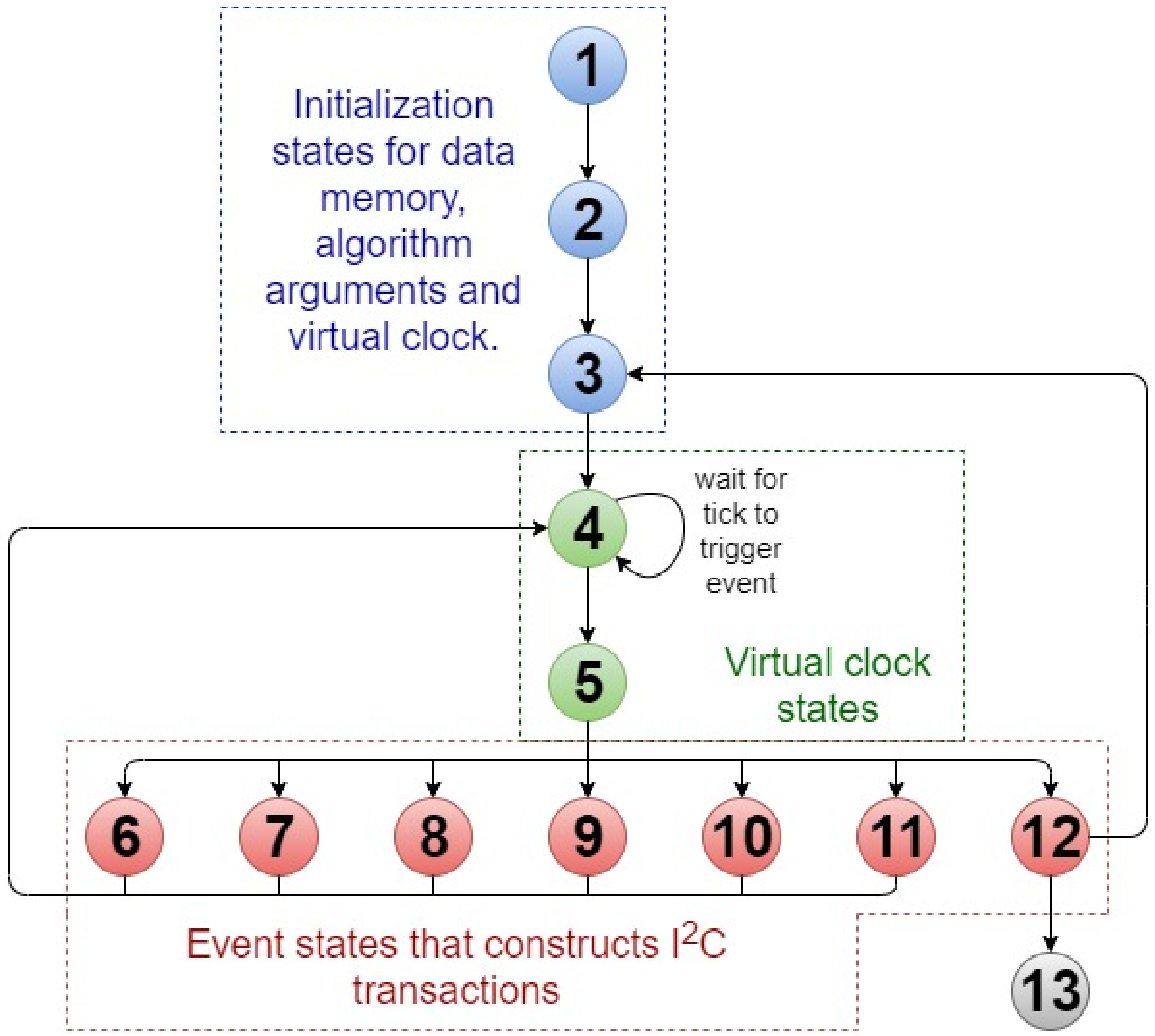

Initialization states for data memory, algorithm arguments and virtual clock.
1
2
3
4
wait for tick to trigger event
5
Virtual clock states
6
7
8
9
10
11
12
Event states that constructs I^{2}C transactions
13

Figure 5.3. State diagram representation of the I^{2}C emulation algorithm.

The states 1, 2 and 3 (blue) are used to setup the microcontroller before the algorithm can use its resources. This setup includes tasks such as configuring the CSRs (Control Status Registers) and the interruption unit. Also, these states are in charge of creating the local variables (managed at register-level) needed by the algorithm to adjust the virtual clock speed, internal counters, transfer packets, etc. States 4 and 5 (green) implement the virtual clock that marks the pace of the FSM and the evaluator that decides the route of the FSM after identifying the current phase of the transaction. These states take advantage of the microcontroller's internal timer to adjust the speed according to the application needs. States 6 to 12 (red) are the constructors of the I^{2}C transaction, each state representing a different phase in a single transfer.

Table 5.1. Figure 5.3 state description.

State Number	Description
1	Initial Setup (CSRs, interruptions, virtual clock freq.)
2	Load variables into register bank
3	Microcontroller setup and initialization.
4	Wait for timer interruption
5	Check I^{2}C transaction phase
6	I^{2}C device reset
7	Start I^{2}C transaction
8	Clear the SCL line
9	Set data bit or ACK on the SDA line
10	Set the SCL line
11	Hold and wait to complete a SCL cycle
12	Stop I^{2}C transaction
13	End-routine configuration

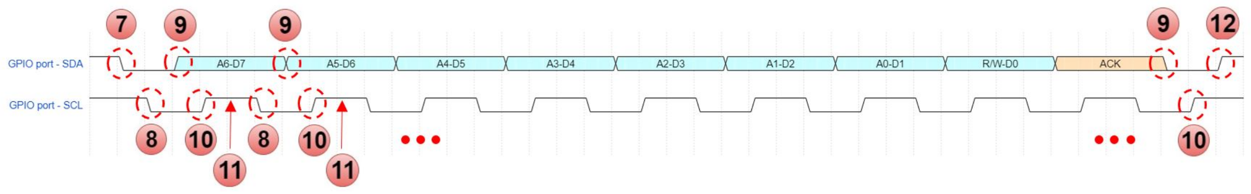


Figure 5.4. An I^{2}C transaction needed for peripheral addresses, commands and data. The numbers represent the states of Figure 5.3 diagram and Table 5.1.

Figure 5.4 illustrates a typical I^{2}C transaction, where commands or data bits are issued. Two lines are used for this transaction as specified by the I^{2}C standard: the SCL line created by the master (the microcontroller) to synchronize the transaction, and the SDA line where data bits or commands with address bits are sent back and forth. Both lines were implemented on GPIO ports, and a third GPIO port was used for the I/O device's reset signal. For each transfer, eight address-commands/data bits are required plus an acknowledge bit to notify the sender if the capture was successful. In Figure 5.4 there are several numbers that match the constructor states of Figure 5.3. Figure 5.4 helps us to understand the loops that the FSM takes to emulate a complete I^{2}C transfer cycle. After a transaction has finished, the algorithm evaluates on a stop condition at state 12 if there are more packets to be sent and a new transfer is prepared, or if there are no packets left, the algorithm continues to the finalization routine to end the program or return to its caller.

5.2 Characterization and Results

After its implementation, the algorithm was characterized to better understand its advantages and features. Also, it helps us mark a road map for future updates. Several characteristics were analyzed to measure the capabilities of the algorithm, such as code size, adaptability to different I^{2}C devices and completeness of the I^{2}C standard.

Figure 5.5 shows an experimental result after the implementation of the proposed algorithm, in which three I^{2}C transactions occur: one command is issued and two data bytes are sent afterward. As explained in the previous section, two GPIO ports were used to emulate the SCL line and the SDA line from the I^{2}C bus, and a third GPIO port was used just to generate the I/O device reset. The acknowledge bits for each transaction, the start condition and the stop conditions are marked on the picture for a better understanding of the illustration. In the results section of [16], a picture is shown in which an LED device was handled using this algorithm on a RISC-V microcontroller.

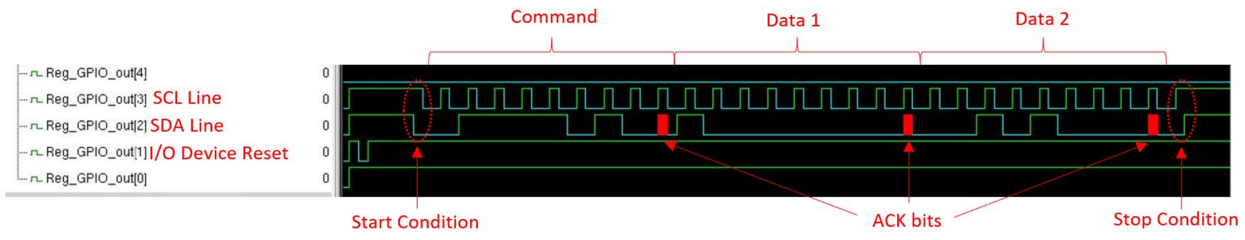


Figure 5.5. Resulting I^{2}C transaction after implementing the proposed algorithm.

The size of the code was measured after compiling the program following Figure 5.2 code structure. Thus, the interrupt service routine implementation used 24 bytes, and the instruction memory used 524 bytes. The data memory had a variable size since it contains the commands to drive the I^{2}C peripheral device. Considering that the entire main memory had a size of 8 *KB*, we could say that the I^{2}C emulation code only occupied around 1/16 segment of the available memory.

The application was built with the ability to easily adjust the driver code depending on the connected I^{2}C device. Since the code to drive the I/O device is contained inside the data memory and separate from the emulated controller, it can be easily swapped before compiling with another code to command a different peripheral device. Hence, it could be said that the emulation algorithm can adapt to different I/O devices that use the I^2C protocol. However, there are two important drawbacks with its current implementation. First, the driver code is still needed inside the main memory before compiling and programming the microcontroller. Second, the current algorithm cannot function as a slave and yet receive information from an external device. These two aspects mark future improvements that can be included in newer versions of this application.

Regarding the completeness of the I^2C protocol, most features of the standard can be handled by the algorithm. Data transactions can be managed with the four speed grades dictated by the I^2C standard. The application can be adjusted to achieve these speed grades with an input argument that modifies the frequency of the algorithm's virtual clock, given that the microcontroller's clock frequency permits such speeds. Equations 5.1, 5.2 and 5.3 describe this modifier.

$$1\ virtual\ clock\ tick = X \times \frac{1}{Freq.} \qquad (\mathbf{5.1})$$

, where *X* is the number in the microcontroller's timer needed for a tick of the virtual clock inside the algorithm and *Freq.* is the actual frequency of the microcontroller.

$$4\ virtual\ clock\ ticks = \frac{1}{SCL\ Freq.} \qquad (\mathbf{5.2})$$

$$Speed\ Grade = SCL\ Freq. \qquad (\mathbf{5.3})$$

, where *SCL Freq.* is the frequency of the SCL line and *Speed Grade* is the desired speed for the application. Let us assume that it is desired to use the I^2C bus standard-mode speed grade, whose transfer value is 100 *kbit/s*. If in one cycle of SCL line, 1 bit is transferred through the SDA line, then the SCL frequency should be 100 *kHz* to transfer 100 *kbit/s*. From there, knowing the frequency of the microcontroller, it is possible to calculate the timer count to adjust the algorithm to the desired speed grade. There are features from the I^2C standard that are not included, like “multi-master" capability or “clock stretching", features that could be added in the future if the necessity arises. Also, it should be noted that figure 3.9 is also a result of the I2C emulation effort, as the OLED display used with SIWA interacts with other devices using the I2C protocol, and SIWA was able to drive the OLEd Display using the emulation shown in this chapter.

As a final note on this application for I^2C emulation versus the proper hardware implementation of the I^2C controller. If the chip size is considered relevant in a project, this application can be incorporated at the expense of a small memory fraction, since the microcontroller has GPIO ports (or similar) that can reproduce the behavior of the I^2C. In addition, the project cost and risk of developing the I^2 C emulation are considerably lower than its hardware counterpart, since the implementation of the algorithm or the fix of a bug can be done even after fabricating the microcontroller.

5.3 Chapter Conclusions

This chapter proposed a low-level flexible algorithm for a software-emulated I^{2}C controller for microcontrollers to drive I^{2}C devices using GPIO ports as substitutes. Applications like this are alternatives for microcontroller design teams when hard decision-making trade-offs arise between chip size, cost, time development, and the necessity to include an I^{2}C bus controller in a project. The chapter described the logic behind this emulation scheme along with the motivation and background of its development. A detailed description of the construction of an I^{2}C transaction was also shown. This article also presented an experimental result appears after the I^{2}C emulation implementation, along with an analysis on the benefits of this approach. Some future works we might consider as a result of this project could be: improvement of the algorithm to include more features of the I^{2}C bus protocol, design of variations of the for other types of communication protocols, and the development of an user interface to modify the arguments of the algorithm without the need of reprogramming the microcontroller.

Chapter 6

A power consumption benchmark for microcontrollers employed in implantable medical devices[1]

Engineering applied to the medical field is a research topic that is highly valued for its possible impact on medical treatments, devices, and procedures, with electronic devices playing a significant role. Knowing how to characterize these electronic devices is required to close the “feedback loop” and identify what may or must be improved. Among such electronic devices, microcontrollers, which are a prevalent solution in many fields, are also a key part in the development of medical devices. As a result, there is a constant need for microcontroller evaluation and research for such applications.

However, the study of microcontrollers is a tricky topic, as the comparison between different microcontrollers is not straightforward: features, architecture, and organization can vary greatly depending on the intended application, making it hard to find a common ground. Logic dictates that it would not be very effective to compare a microcontroller whose design was conceived for high-performance, with a microcontroller that sacrifices performance for low-power consumption. Moreover, their final application environment will probably be different as well. Comparing two microcontrollers with similar applications and desired characteristics makes a little more sense, and even then, the results can lead to misguided conclusions if not done properly.

Classic microcontroller comparison approaches in the literature follow criteria such as measuring CPI (Cycles Per Instruction) and MIPS (Millions Instructions Per Second). However, this comparison is only valid when the subjects share a common trait. In the CPI case, the processors need to operate at the same frequency. In the MIPS case, albeit the equal frequency requirement is eliminated, the processors under study have to share the same architecture for transparency reasons. A more realistic comparison approach contemplates the use of benchmarks. In a few words, benchmarks are points of reference and, particularly in computer architecture, benchmarks are usually a set of programs, tables, and/or charts that allow engineers to assess certain characteristics of the processor, no matter its architecture.

[1] This chapter contents have been published verbatim in [55]

Implantable medical devices, such as pacemakers, require processing units with specific properties. On the one hand, the processing unit needs to communicate with different types of peripherals such as sensors and actuators, and with the continuous advancement of IoT (Internet of Things) and embedded systems, it also has to be able to communicate with other intelligent devices, too. However, the direct impact of power consumption on battery life makes low-power a much desired characteristic.

This paper proposes a benchmark to compare the microcontrollers employed in implantable medical devices. A general purpose RISC-V microcontroller, called SIWA, is used as a foundation to exercise the benchmark (see [21] for more details). This sets a reference point for other low-power microcontrollers that may also be used in medical applications. The benchmark evaluates power consumption when running different programs with particular instructions of an ISA, programs on sleep-mode and active-mode, and a program that emulates a typical application for implantable devices.

Section II gives a brief overview of the state-of-the-art in microcontroller benchmarking and sheds some light on related topics. Section III summarizes the architecture and organization of SIWA. Section IV describes the methodology followed for the experimental measurements and the test programs developed for this benchmark. Section V presents results and analysis to lay the ground for comparison to other microcontrollers. Finally, Section VI shares conclusions and proposes future work.

6.1 State of the Art overview

SIWA, the microcontroller used for the development of this benchmark, is thoroughly described in terms of its architecture in [21] (some additional references provide detailed aspects of SIWA in terms of physical design [20], its medical application [22], hardware emulation [39], hardware verification [1] and post-fab testing [16]). For the sake of clarity, Section III describes a simplified overview for SIWA.

Regarding the use of microcontrollers employed in the medical field, [56] proposed a 8-bit low-power microcontroller for wireless biomedical sensors. The work in [57] describes the design of a pacemaker using an Atmega328P from Microchip, and [58] proposes a two-phase pacemaker using a C8051F340 from Silicon Labs. There are several other works found on online repositories, however, none follows a consistent methodology used to present or compare information, especially in terms of power consumption. In reality, it is not easy to find a power consumption evaluation that at least attempts to provide equivalent measurement conditions, such as operating at the same frequency or performing similar tasks. This could lead to misleading results, which points to the relevance of the proposed benchmark.

Microcontrollers benchmarking is a continuously evolving field. There is a compilation of benchmarks used to compare microcontrollers given in [59], and some of them still are used today, while others have evolved into more modern benchmarks. Most of these are general purpose benchmarks, not suitable for generating relevant data for microcontrollers on implantable devices that are mostly very application-specific. Still, inspiration was derived from some of them, such as Dhrystone, for its idea and simplicity.

There are several publications about benchmarking for specific applications, for example, [60] proposes a benchmark for microcontrollers used on a specific wireless application, [61] explains a benchmark using

ATPGs to test microcontrollers under radiation effects, and [62] presents a study using different benchmarks to find an optimal microcontroller depending on some desired characteristics. Although any of them are exactly related to this paper's field of study, they indeed prove that developing benchmarks towards specific applications can yield more detailed results in research analysis and generate better conclusions.

This chapter explains the technical details of the benchmark created that helps compare microcontrollers for intended use on medical applications and low-power applications. Section 6.1 summarizes, yet again, important SIWA details necessary to understand the remainder of the chapter. Section 6.2 describes the methodology and creation of the benchmark, with the purpose of comparing existing microcontrollers with SIWA and future implementations. Section 6.3 shows some results and analyses their meaning in a medical application context. Lastly, section 6.4 concludes the most important findings of the chapter.

6.2 Microcontroller Architecture and Organization

As covered throughout this thesis, SIWA, the microcontroller used as a subject test for this benchmark, is a custom-made RISC-RV32I-based architecture. Figure 6.1 depicts the high-level hierarchical organization of SIWA.

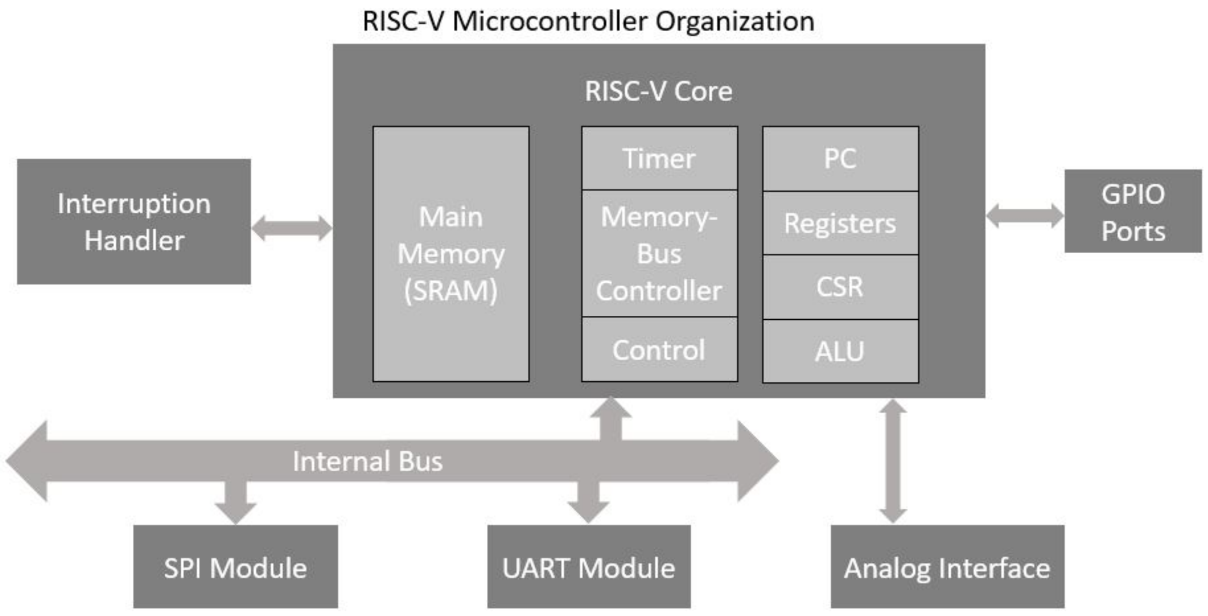


Figure 6.1. Diagram that summarizes the architecture and organization of SIWA, the microcontroller used to develop the proposed benchmark. More details may be found at [21], [20], [22], [1] and [16].

SIWA was designed for low-power consumption. Therefore, its organization is concise and simple, yet, it has enough features to allow the programmer some degree of freedom for medical applications. For instance, from the architectural point of view, the microcontroller can interpret the complete RISC-V RV32I ISA (Instruction Set Architecture), allows for several types of interrupts, and has a built-in timer for real-time applications along with a 32-bit register bank and an 8 kB main memory.

Let us recall that SIWA also possesses several I/O modules to communicate with external devices: one SPI port, one UART port, eigth 1-bit GPIO ports and a custom digital port to allow interruptions from an analog component. In one of its versions, SIWA can also interact directly with several in-die high-voltage level-shifters and DC current sources which may be used for biological tissue stimulus (see [22]). The SPI port is an entry point from which the code is booted into the microcontroller, and said code resides inside a flash memory that communicates through the SPI interface. After reset, the microcontroller requests read operations to the flash memory using the SPI module until the program is fully loaded inside the main memory. The UART port is there to communicate to whatever peripheral the programmer needs to use. The eight 1-bit GPIO ports were included to drive digital indicators and actuators or monitor digital sensors. The SPI and UART are also capable of activating internal interrupts, and a pin is provided for external interrupts.

From the organizational point of view, the microcontroller has a minimalist design. Its instructions are executed in a linear and scalar way, in other words, it does not have a pipelined or superscalar implementation. However, processing performance is not the main concern for basic IMDs. Figure 6.2 shows a potential application of SIWA as a pacemaker and how it can interact with other elements within the medical device.

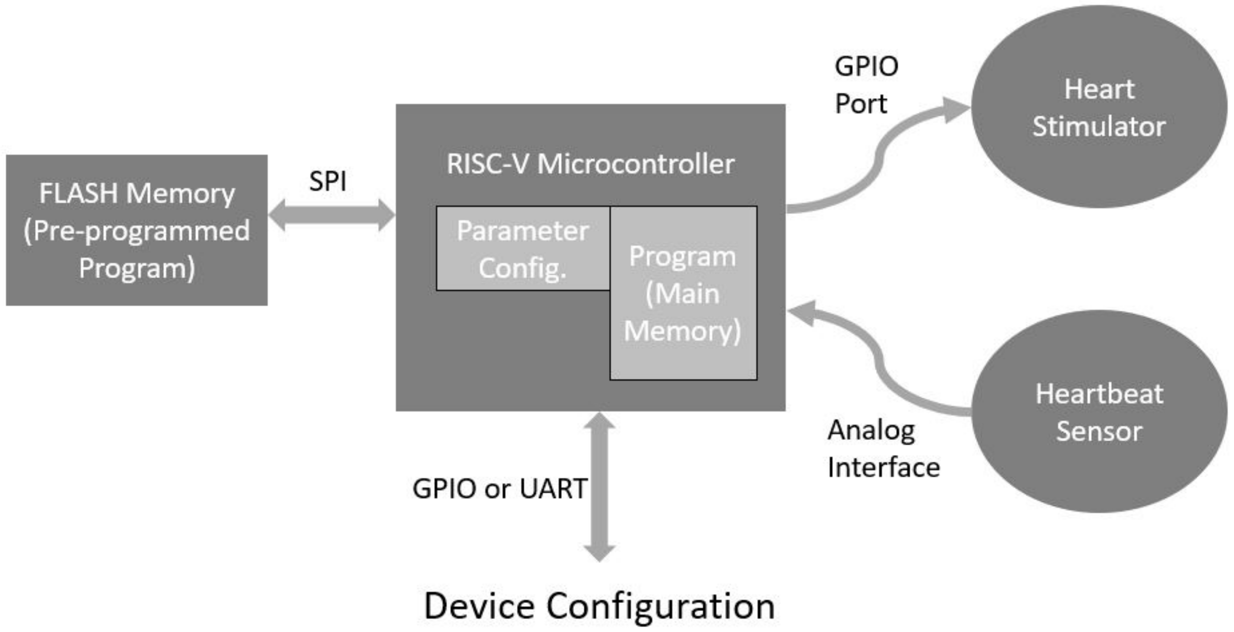


Figure 6.2. A potential application for SIWA as part of a pacemaker.

Inside a pacemaker, the SIWA interfaces would interact with its neighbors via the aforementioned interfaces. The SPI would be connected to a flash memory used first as a booting device, and after bootstrap as secondary storage. The GPIO ports would interact with the circuitry of a heart stimulator, according to the readings of a heartbeat sensor and the configurable timing variables of the program inside the microcontroller. The custom digital port of the analog interface would receive a signal from a heartbeat sensor each time the heart beats. The UART port could be used to configure the program from another device to adjust pacemaker timing according to the specific medical treatment of a patient. An alternative to the UART, in case another protocol is needed to communicate with the device, would be to use the GPIO ports as channels to emulate other communication protocols. For example, in [39], an I^2 C protocol was successfully implemented on SIWA using two GPIO ports to drive and display intelligible.

Subsection 6.3.2 shares some details about the development of the program for this application, as it was used as part of the proposed benchmark.

6.3 Methodology and Development

This section is subdivided into two subsections: the testbench description and the development of the benchmark programs. The former describes the framework employed to acquire data, and the latter details the logic behind the proposed benchmark programs.

6.3.1 Testbench description

The complete test framework required several components to be able to take measurements of SIWA. Figure 6.3 illustrates the testbench used to take power measurements.

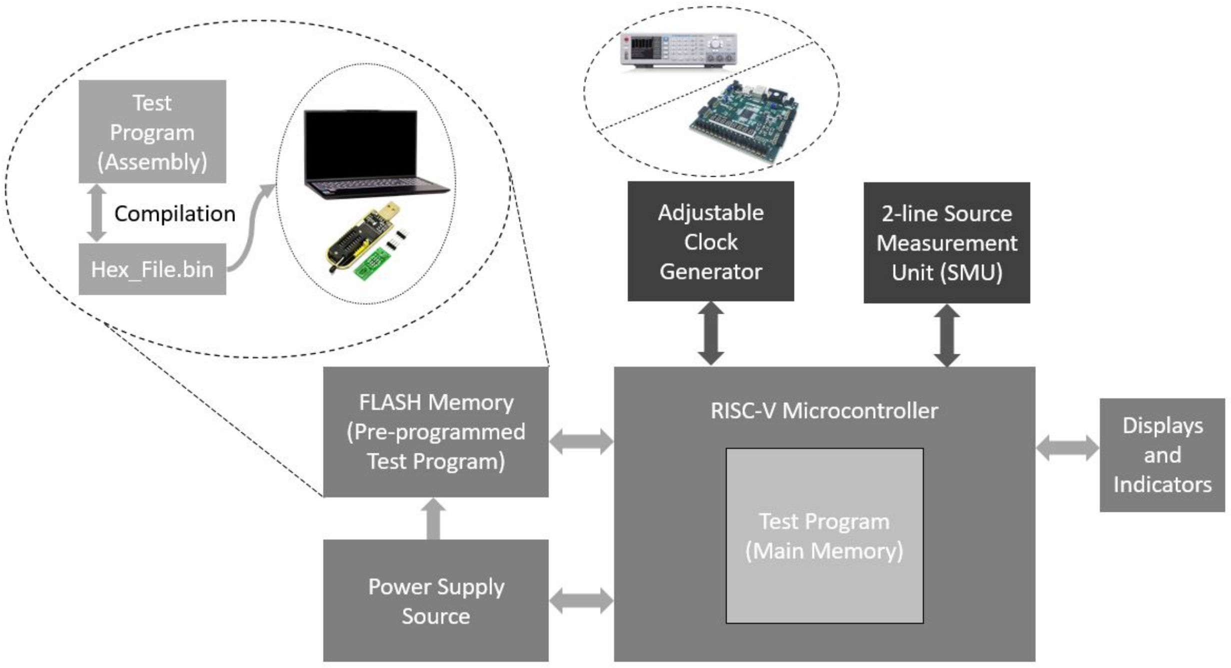

Test Program (Assembly)
Compilation
Hex_File.bin
Adjustable Clock Generator
2-line Source Measurement Unit (SMU)
FLASH Memory (Pre-programmed Test Program)
RISC-V Microcontroller
Test Program (Main Memory)
Displays and Indicators
Power Supply Source

Figure 6.3. Test framework for measurement reading.

Typically, when testing a device (DUT, Design/Device Under Test, from now on), efforts should be directed to two aspects: the stimuli that would put the DUT on a working state, and the monitors or measurement units that would capture the desired parameter data based on the generated stimuli. In this case, the stimuli are the benchmark programs, power supplies, and clock generators, while our measurement unit was a 2-line SMU (Source Measurement Unit) and LEDs or displays if required.

An external flash memory is used to boot SIWA. The test files encoded in the RISC-V 32I assembly language are downloaded to flash through a commercial programmer [63], using generic software. The test programs were already compiled into `*.txt` files using hexadecimal code taken from RTL simulations [1], using the RISC-V community open-source toolchain [14]. However, it was later discovered that the programmer software needed the test program to be loaded as a `*.bin` file instead of a `*.txt` format. This requires converting `*.txt` files to `*.bin` using the Linux system command: `xxd -r -p "<testfile_name>.txt" > "<testfile_name>.bin"`.

SIWA is a microcontroller designed to work at a 20 MHz or lower frequency clock, usually an external oscillator. However, our test plan included measurements at different frequencies; hence a function generator was used to create clock signals at lower frequencies (the equipment's maximum limit was 10 MHz) and a multiple clock generator synthesized inside an FPGA to create clock signals at higher frequencies (its maximum limit was 100 MHz). Also, SIWA's typical operational voltage is 1.8 V and the SMU was used to fill the role of voltage supplier to the core while measuring the current flow required. The rest of the components on the framework were supplied with other voltage sources. By dedicating the SMU to the microcontroller, one can guarantee that the current measurement reads correspond solely to the microcontroller.

In summary, assuming that all test programs are already designed and developed, the typical testbench workflow shown in Figure 6.3 contemplates the following steps:

1. Compile the assembly or C test programs using the RISC-V toolchain [14], then convert the hexadecimal output to a `*.bin file`.
2. Load a `*.bin file` file into the flash programmer, then clean and program the flash memory with said file.
3. Detach the flash memory from the programmer and connect it to the microcontroller.
4. Turn on all power supplies to bring the complete framework to a pre-reset state, then apply a reset.
5. Adjust the frequency to the desired value on the function generator/FPGA.
6. Clear the reset so that the microcontroller can boot from the flash memory, then wait (this may take several seconds for very low-frequency clocks).
7. Once the microcontroller is booted and stabilized according to the logic of the test program, take the desired measurements.
8. If finished, turn off all power supplies, disconnect the flash memory, and reattach it to the programmer.
9. Repeat from Step 2 onward to take more measurements with other test programs.

6.3.2 Benchmark programs description

When benchmarking, the microcontroller under test is generally commissioned to run a particular set of tasks to get a score. This score represents a reference point to compare with other devices. Dhrystone and Whetstone [59] are classic synthetic benchmarks that try to measure processor performance. Drhystone measures performance by submitting the subject to simple integer calculations while Whetstone does similar tasks using floating-point. Both are known as “synthetic” because their set of programs are not “real applications” but a compiled set of processes based on what is used most of the time.

In this paper, a set of synthetic programs are proposed, but with two important differences from the procedures, such as those followed by Drhystone and Whetstone. First, the proposed set of programs try to set a reference point based on power consumption and not performance. Second, the design of the programs was carried out taking into account medical applications for implantable devices. The designed test programs were used to take power measurements, based on the framework explained in the previous subsection.

The proposed set of programs are divided into two categories. The first category aims to measure power consumption based on “types of instruction”. Most instructions are used in typical medical applications, albeit not with the same occurrence. Therefore, this first category tries to set a baseline of power consumption measurements taking into account all instructions and condensing all data into classic functional groups: arithmetic/logic instructions, control instructions, memory instruction, etc. Figure 6.4 illustrates the algorithm and logic of the test programs that were developed and compiled into the microcontroller to take power consumption measurements for this category.

The intention was to program the microcontroller with a task that executes a recurring instruction indefinitely. Therefore, the microcontroller was set to continuously execute the same instruction several times and then restart the process anew to close the loop. Once the microcontroller was in this state, a power measurement was taken and logged for post-processing later on. Then a new program with the same algorithm but with a different instruction was programmed in the microcontroller to take different power consumption measurements. The process was repeated with several instructions.

For example, assume that one desires to measure the power consumption required by an `addi` instruction in a particular microcontroller. Then, using the algorithm of Figure 6.4, a set of hundreds of `addi`, instructions were programmed and looped; therefore, the microcontroller would execute these instructions most of the time to ensure that its power consumption is mostly due to the `addi` instructions.

For this benchmark, adjusting the microcontroller’s frequency is also proposed in order to study the behavior of the power consumption due to frequency changes. Since the power consumption of a microcontroller has, ideally, a linear ratio against its frequency of operation, power measurements were normalized to the MIPS (Millions of Instructions Per Second) parameter, helping thus to eliminate the frequency aspect from the comparisons if, effectively, the power consumption and frequency ratio is linear. To calculate the normalized power [64] the following equations were used:

$$\text{MIPS} = \frac{\text{CPU Freq.}}{\text{CPI} \times 10^6} \tag{6.1}$$

$$\text{Normalized Power} = \frac{\text{Avg. Power}}{\text{MIPS}} \tag{6.2}$$

The second category of programs is composed of those that emulate certain aspects of a typical medical application in pacemakers. Generally, this type of applications try to put the microcontroller on sleep-mode at 32.768 Hz [**empty citation**] most of the time while waiting for a heartbeat. If a heartbeat occurs in the

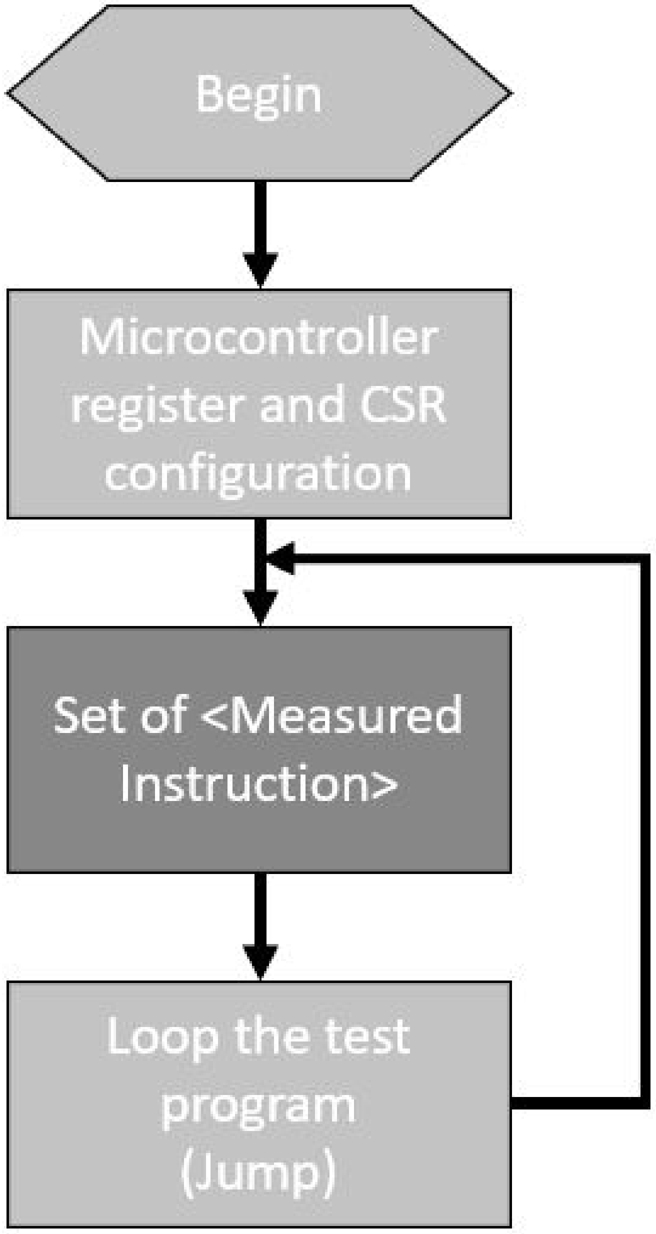

Begin
Microcontroller register and CSR configuration
Set of <Measured Instruction>
Loop the test program (Jump)

Figure 6.4. Generic algorithm for the first program set.

specified time range (as recommended by the cardiac electrophysiologist, based on the patient's conditions), then the device stays in sleep mode. If a heartbeat is not accounted for in the specified time, then the medical device must transition to a high-frequency mode to take action and stimulate the heart before going back to sleep-mode. This logic allows to reduce power consumption as much as possible, preserving battery life and thus reducing surgery interventions on the patient.

Figure 6.5 shows the algorithm of the proposed benchmark that emulates the behavior described. On the left side, the logic already described for the medical application for a pacemaker is shown; on the right side, there is an interrupt routine that complements the task on the left side. The microcontroller is configured through several CSRs (Control Status Registers) to set the internal timer and to have the microcontroller enter into sleep-mode at 32,768 Hz, simply waiting for an interrupt to occur. Once an interrupt occurs, the service routine reads the corresponding CSR to identify who triggered the interrupt. If the interrupt is triggered by a heartbeat flag, then the program continues in sleep-mode and waits for the next interrupt. But if the timer claims to be the owner of the interrupt, then the microcontroller speeds up to the maximum clock frequency (20 MHz) to stimulate the heart. An interrupt-based solution not only saves power but avoids synchronization issues with the cardiac rhythm, which render a port polling approach impractical.

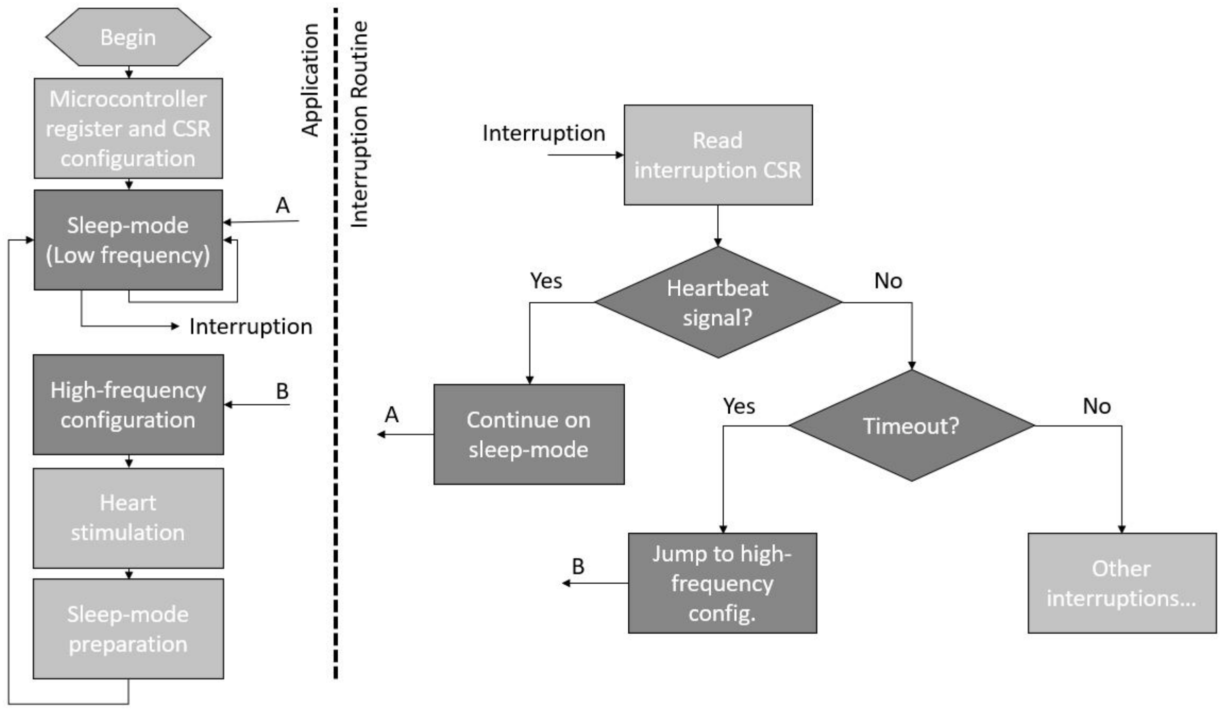


Figure 6.5. Pacemaker application algorithm used for the benchmark.

Going back to the benchmark details, the second category is conformed by particular aspects of this medical application implemented in smaller programs with specific purposes. As the primary goal here is to measure power consumption, two features standout from the logic of the medical application: a sleep-mode at low-frequency and and active-mode at high-frequency. Therefore, two measurement programs were conceived, one that keeps the microcontroller at a low-frequency clock constantly, and another that executes at a higher clock speed several instructions, similar to the ones contained inside the processes of the heart stimulation.

This first measurement program was implemented by slightly modifying the program of Figure 6.5. The only adjustment required was to avoid the program from entering the interruption routine, which was achieved by changing the jump destination in the corresponding CSR when an interrupt occurs, directing it to the part where the program puts the microcontroller into sleep-mode, which results in the logic described by Figure 6.6.

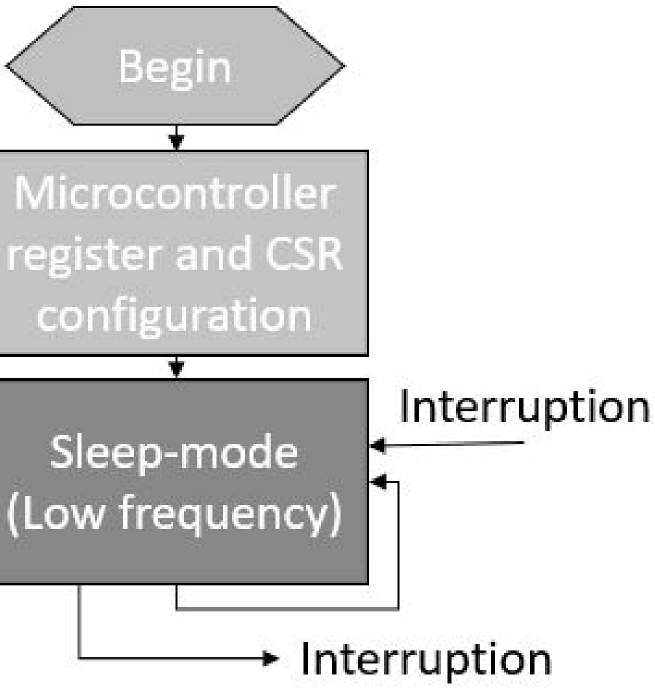

Begin
Microcontroller register and CSR configuration
Interruption
Sleep-mode (Low frequency)
Interruption

Figure 6.6. Forced sleep-mode used to measure power consumption.

The second measurement program focused on the stimulation processes in the heart, running on a high-frequency clock. This program is actually a set of three different routines. Using the algorithm shown in Figure 6.4, a new set of mixed instructions was written. This set is a combination of instructions typically used for heart stimulation, composed of readings and drivings from the different I/O modules, and several simple arithmetic calculations, memory operations, and jumps. Each routine has the same logic but runs with different data-size operations (8-bit, 16-bit, and 32-bit operands), in order to compare the respective impact on power performance.

All subject microcontrollers execute thus each program and power measurements are taken. Therefore, 8-bit, 16-bit and 32-bit microcontrollers will have to use their corresponding instructions to execute the different-sized operations for this program set. 8-bit microcontrollers will require more instructions than 32-bit microcontrollers to calculate some of these operations. Meanwhile, 32-bit microcontrollers will probably consume more power since there are more switching bits and the components are larger. This makes a comparison between the three types difficult. However, normalized power can be used to bring the comparison between 8-bit, 16-bit, and 32-bit closer. More meaningful results may be obtained by using a 'Cycles Per Task" (let us name it CPT) metric instead of using the typical CPI (Cycles Per Instruction) metric for the MIPS calculation. For instance, assume a subtraction operation with 32-bits operands is executed, a 32-bit microcontroller would require a single instruction to execute this operation while an 8-bit microcontroller would require several instructions. Then, after looping both the microcontrollers to do the exact same operation indefinitely, a power measurement is taken showing that the 8-bit microcontroller consumes less power. Afterwards, the power measurements are normalized, but instead of using the *sub* instruction CPI, the total number of cycles needed to complete the 32-bit subtraction operation (CPT) is used for the calculation to compensate that the 8-bit microcontroller takes longer to execute the same task. Thus, the power consumption due by hardware and time when executing the operation is balanced, bringing the microcontrollers to a similar floor level.

6.4 Results and Analysis

Several results are shown based on the framework and the test programs described earlier. Using the logic of Figure 6.4 as the baseline, the current was measured using an SMU, while executing several instructions from the RISC-V RV32I ISA. The power consumption was then calculated considering that SIWA uses a 1.8 V supply. This flow was repeated at different frequencies ranging from 1 kHz up to 30 MHz (maximum clock frequency operation for SIWA). Subsequently, power consumption data were grouped into the typical instruction types categories and the power consumption for each category was averaged to build the graph given in Figure 6.7.

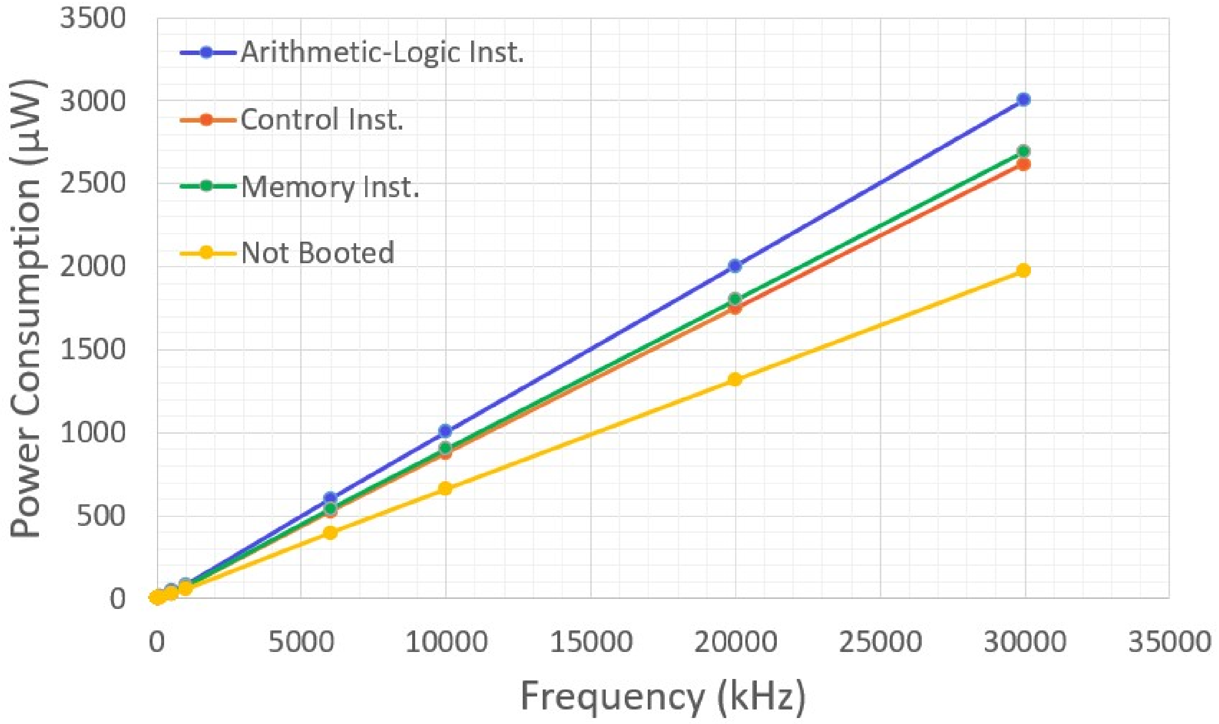


Figure 6.7. Average measured power consumption at multiple frequencies, classified by instruction types. V_{DD} =1.8 V

This chart illustrates four lines based on the power consumption calculations at different frequencies, each line corresponding to a different category. Arithmetic/logic instructions are those used byt the ALU for basic data processing like `add`, `sub`, `and`, or, `sll`, `sra`, etc; and their immediate equivalents. Control instructions are those that alter the linear sequence of a program by writing directly into the program counter, like jumps and branch instructions. Lastly, memory instructions are those related to memory operations, like `sw`, `lw`, `sh`, `lh`, etc. The fourth line gives the power consumption when the microcontroller has not yet booted up.

Because the frequency range for the measurements taken is so large, Figure 6.7 does not illustrate the power consumption behavior at frequencies lower than 1 MHz. Thus, Figure 6.8 zooms in on the same results but at lower frequencies.

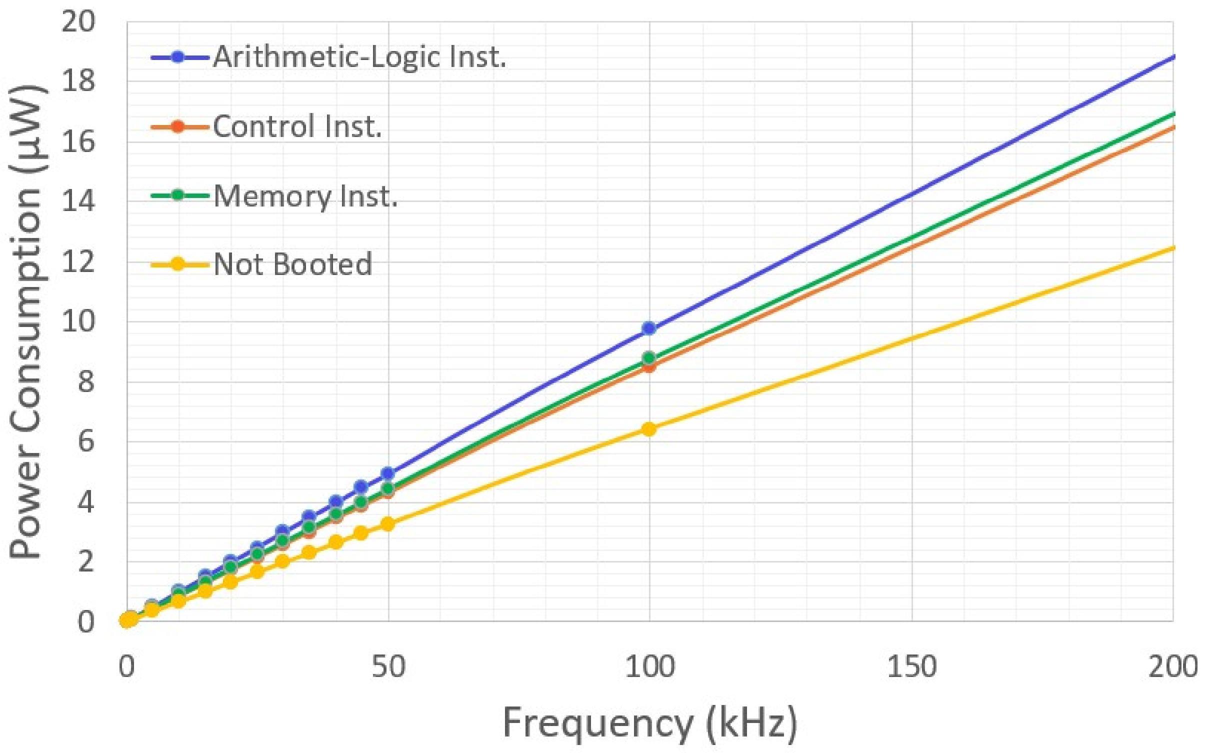


Figure 6.8. Average measured power consumption at lower frequencies, classified by instruction types. V_{DD} =1.8 V

The presented data gives a glimpse of the static power consumption. Figure 6.9 zooms in the data set even further to be able to visualize the Y-axis crossover to get an estimated value for the static power consumption. From there, the estimated static power consumption reading is around 20 nW.

Normalized power consumption was calculated using equations (6.1) and (6.2) for each frequency reading, with the CPI as shown in Table 6.1. For this case, the behavior of the average power consumption related to frequency shows, as expected, a linear tendency. The result was that every normalized power calculation was almost identical for every frequency reading. Hence, if the behavior of the power and frequency ratio for the subject microcontroller is linear, equations 6.1 and 6.2 can be simplified to:

$$\text{Norm. Power} = (\text{CPI} \times 10^6)(\text{Power vs. Freq. Slope}) \quad \textbf{(6.3)}$$

Table 6.1 shows the results for the normalized power consumption according to (6.3).

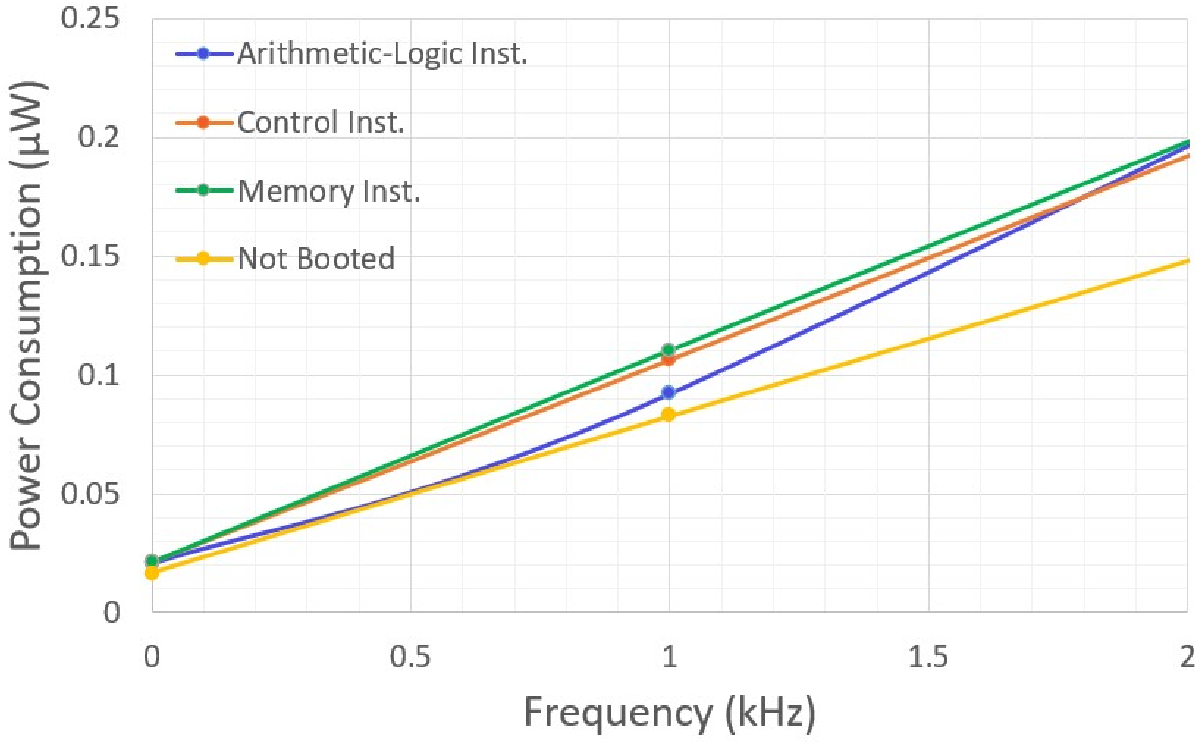


Figure 6.9. Y-axis crossover of the averaged power consumption data set. V_{DD} =1.8 V

Table 6.1. Normalized power consumption for each type of instruction with a slope of 1.003x10^{-7} mW/Hz from Figure 6.7.

Data Set	Avr. CPI	Normalized Power
Arithmetic-Logic Inst.	13.85	1.3896 mW/MIPS
Control Inst.	13.125	1.3169 mW/MIPS
Memory Inst.	10	1.003 mW/MIPS

Table 6.2. Average power consumption of a program emulating the tasks needed in a pacemaker device.

Data Set	Frequency	Avr. Power Consumption
Sleep Mode	32,768.00 kHz	1.139706 μW
8-bit Operations	20 MHz	2.02068 mW
16-bit Operations	20 MHz	2.07054 mW
32-bit Operations	20 MHz	2.12292 mW

The data presented so far is related to the first set of programs used to characterize instructions. The following charts and tables are created based on the data measured from the programs that emulate a pacemaker medical application based on the algorithm given in Figure 6.5. Table 6.2 shows the power consumption measurements for the second data set.

Table 6.2 presents four measurements. The first measure was taken when the microcontroller was in a permanent forced sleep-mode, reached by implementing the algorithm of Figure 6.6. A pacemaker spends most of its operational time in sleep-mode while waiting for a missed heartbeat, hence the importance of this measurement. The other three measurements were taken while executing the algorithm of Figure 6.5 using 8-bit, 16-bit, and 32-bit operations, respectively. The microcontroller is forced to stay in a high-frequency clock state for measurement reliability. These three measurements also have great relevance for the proposed benchmark study, since they represent the peak of power consumption during a typical low-power medical application.

One can see that running the Figure 6.5 benchmark program with different data sizes can provide valuable information when deciding on a microcontroller architecture word size (8-bit, 16-bit, or 32-bit) to use in an implantable device. The idea is that whatever the microcontroller word size is under study, the microcontroller should run all different sized operations. Afterwards, the power measurements should be normalized so that the comparison between microcontrollers is balanced. If not, the comparison could end up being biased, since 32-bit microcontrollers will consume more power while executing smaller-sized operations than their smaller counterparts. And under the same logic, 8-bit microcontrollers will consume power over a longer period of time while executing bigger sized operations than the bigger microcontrollers. As mentioned before, in this case, normalization to the CPI (or MIPS) can no longer be used for the same reason; 8-bit, 16-bit, or 32-bit microcontrollers may need a different number of cycles to complete a similar task. Thus, those who require more cycles have to stay in active-mode for longer periods of time.

For this purpose, we propose using the cycles required to complete the task instead of the average CPI. If the average CPI is used, then the fact that some microcontrollers may need to stay in active-mode longer will be omitted. A task would be a set of instructions that are used to execute a specific job. In a pacemaker case, if the microcontroller is on sleep-mode, then its task would be to sleep and wait for an interruption; hence, its CPT (Cycles Per Task) would be the cycles needed for the loop, a jump instruction jumping to itself. If the microcontroller is in active-mode, then its task would be the required instructions to read heart signals, do some calculations, use memories, and generate a stimulus. Table 6.3 presents the normalized power consumption to the “Million of Tasks Per Second” (MTPS), derived from the quantity of CPT (we treat tasks like individual instructions for the normalization process), using equations 6.1 and 6.2.

Since SIWA is a 32-bit microcontroller, it took the same number of cycles to execute the tasks with 8-bit, 16-bit, and 32-bit operations. If we would like it to be compared to an 8-bit microcontroller, we would

Table 6.3. Normalized power consumption to Millions of Tasks Per Second for each data set measured

Data Set	Cycles Per Task	Normalized Power
Sleep Mode	14	0.6 mW/MTPS
8-bit Operations	209	21.1161 mW/MTPS
16-bit Operations	209	21.6371 mW/MTPS
32-bit Operations	209	22.1845 mW/MTPS

implement the tasks on that microcontroller, take measurements, estimate the cycles needed to complete the tasks, and normalize it to the MTPS. In that case, the 8-bit microcontroller would consume less power when taking the equivalent measurements of Table 6.2, but would also require more instructions to execute 16-bit and 32-bit operations, rendering the CPT higher.

Another important characteristic that deserves some study is the sleep-mode feature in a microcontroller. Measurements shown in Figures 6.7, 6.8, 6.9 and Table 6.2 were taken when the microcontroller was "awake", meaning that all features were fully operational, resulting in a higher power consumption. The only measurement presented that was taken in sleep-mode are the ones shown in Tables 6.2 and 6.3 under the category of "Sleep-Mode". While in sleep-mode, SIWA turns off the internal bus, SPI and UART peripheral modules by clock gating them. Figures 6.10, 6.11 and 6.12 depict the power consumption of SIWA in sleep mode at high, mid, and low operation frequencies, respectively.

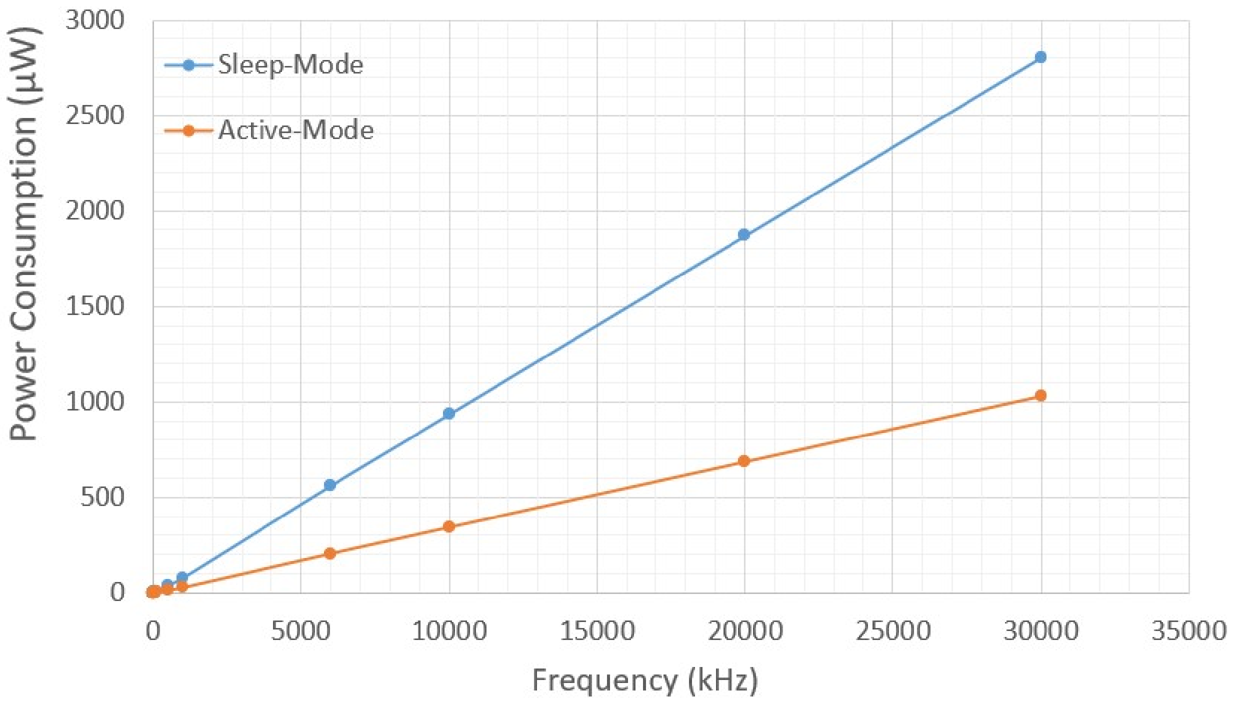


Figure 6.10. Sleep-mode power consumption comparison versus active-mode while running loops of NOP instructions. V_{DD} =1.8 V

The measurements used for these charts were acquired by running NOP instruction loops with the algorithm described in Figure 6.4. For SIWA, the power consumption was reduced by 63-64% while in sleep-mode versus its normal operation. If more sleep-mode characteristics are needed and if the microcontroller behav-

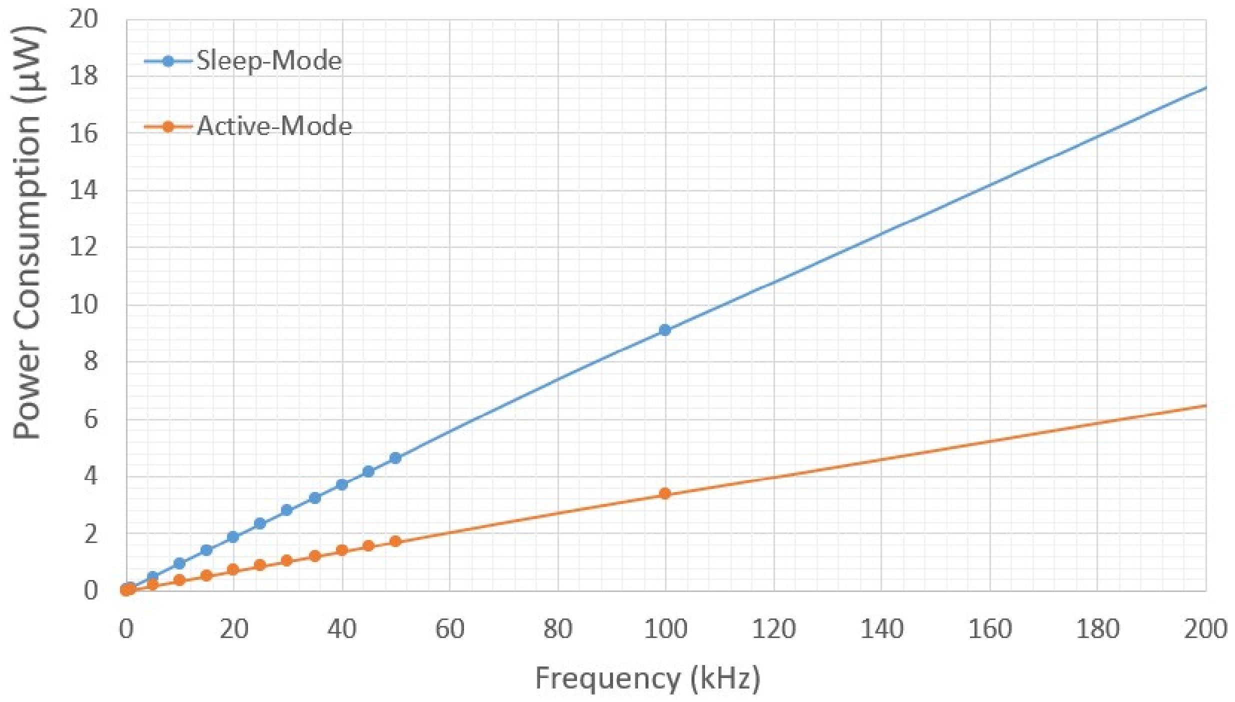


Figure 6.11. Sleep-mode power consumption comparison versus active-mode while running loops of NOP instructions at mid-range operation frequencies. VDD = 1.8 *V*

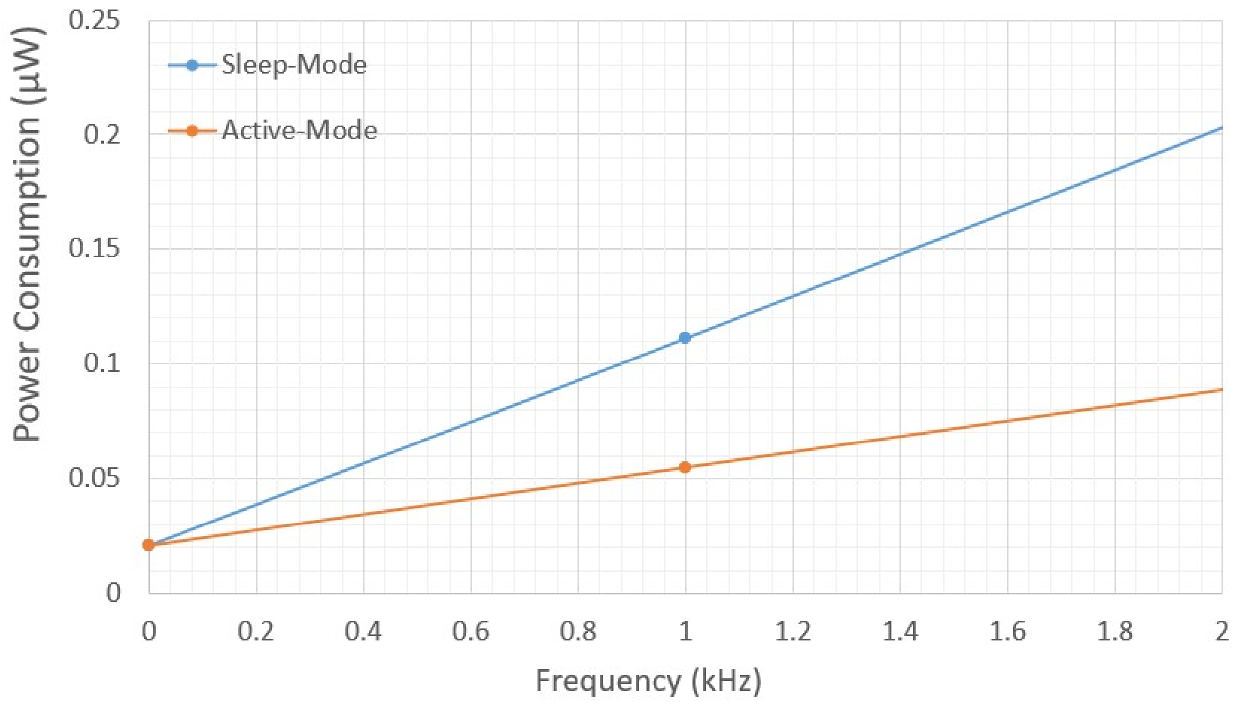


Figure 6.12. Sleep-mode power consumption comparison versus active-mode while running loops of NOP instructions at low operation frequencies. VDD = 1.8 *V*

ior shows something akin to a linear relationship with the frequency, an estimation can be made with the data presented so far by extrapolation.

One final result is a projection of power consumption over a particular time lapse, which is shown in Figure 6.13. The red pulses represent heartbeats in the form of step pulses of 5 ms, assuming a Tokyo signal [65] has been processed by an operating transconductance amplifier and a comparator to deliver a 1.8 V pulse to the microcontroller. The blue line corresponds to the power consumption of the microcontroller along the 10 second lapse. The blue baseline represents the microcontroller power consumption of 1.1 μW while on sleep-mode, the blue peaks represent the moments when the microcontroller switches to active-mode and stimulates the heart when a heartbeat is missing after 0.9 seconds (configurable according to the patient's needs).

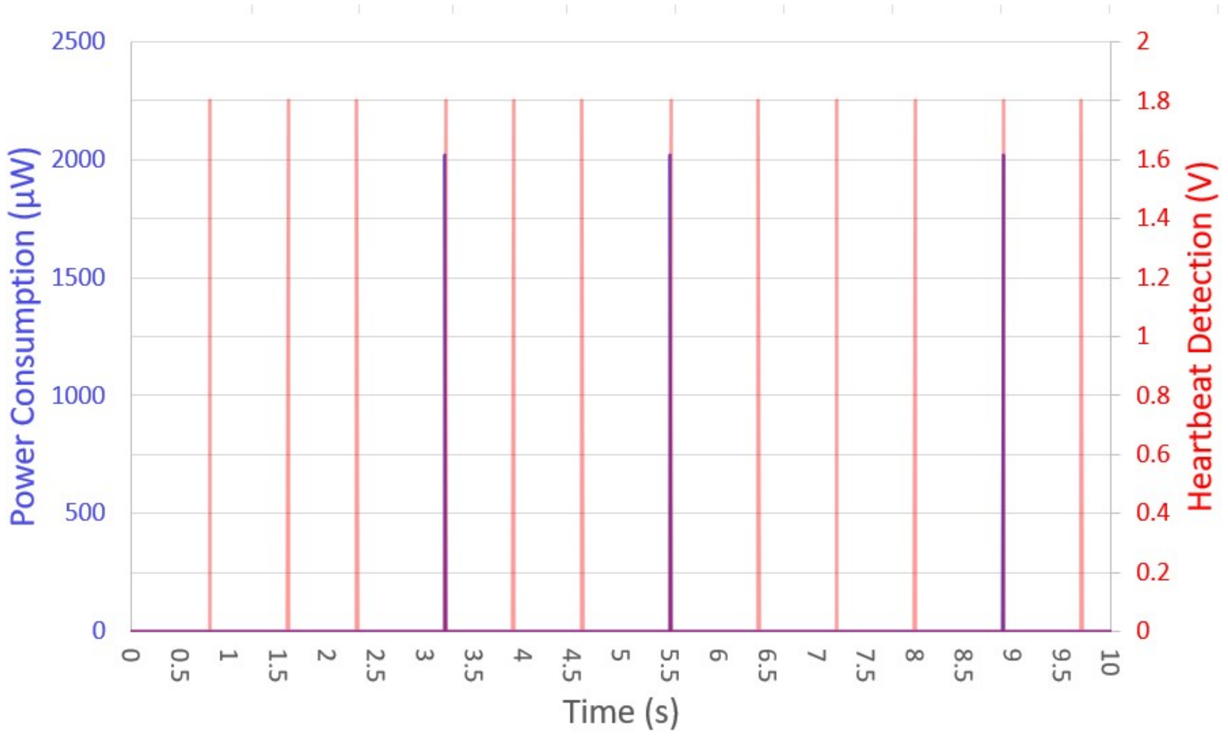


Figure 6.13. Power consumption projection of SIWA when running the pacemaker app over a period of 10 seconds.

The proposed benchmark allows us to compile a lot of data on power consumption in microcontrollers intended for implantable medical devices. This data is helpful in characterizing the power consumption of the microcontroller. Specifically, normalized power consumption for typical instruction types through all the frequencies, normalized power consumption when running an implantable medical-like application on sleep and active modes, power consumption in sleep-mode versus active-mode, and a projection of power consumption over a time lapse. Normalized measurements allow one to easily compare results with other subjects.

6.5 Chapter Conclusions

This paper proposed a benchmark for microcontrollers targeted for implantable medical devices. In the Introduction, the main motivation for the work is presented and how we believe that this benchmark fills a void about benchmarks on this specific field. Later, a description of the whole framework is detailed, from a brief explanation of the subject microcontroller to the equipment used to program and measure it. Afterwards, several benchmark program sets were proposed with their respective algorithms and intention for the decisions taken. Finally, results are shown in the form of charts and tables with accompanying analyses to explain how a comparison can be made through normalization and the importance of the benchmark.

The proposed algorithms are synthetic and are based on the behavior of real life implantable device applications. The algorithms were implemented in the RISC-V 32I assembly to be tested on SIWA, but they are small and easy to implement on other platforms. The proposed benchmark results permit the subject microcontroller power characterization to be studied at different levels. For example, how much power consumes on average a particular instruction type; how power consumption behaves against frequency changes; an estimation of static power leakage; how much does the microcontroller consume when running an implantable device application under standard conditions; the relationship between power consumption in sleep-mode and active-mode; and a method, based on [64], to compare results versus other microcontrollers by normalizing power consumption to the MIPS and MTPS parameters.

Chapter 7

Thesis Conclusions

This chapter summarizes the conclusions of all the work done in this thesis and provides a small overview of future work. The main conclusions are drawn from the four specific objectives defined in the introductory chapter along with more specific results based on the results obtained in each individual chapter.

With regard to the first objective of this dissertation, a functional verification framework for SIWA was designed and created. This verification framework was compact with enough components to warrant a solid verification of SIWA. The environment was able to drive a pseudorandom stimulus to SIWA, generate a gold reference complaint with the RISC-V standard for each individual test to verify results, collect coverage data for decision making, test multiple blocks at different levels of the hierarchical structure, and implement a regression system to run multiple tests at once on a daily basis. The environment has naturally evolved into UVM for simplicity.

The environment verified four major blocks of SIWA, in addition to the top level: the memory bus controller unit, the ALU, the bus, and the UART; each one of them with their own verification environment. The environments exercised every possible instruction of the RISC-V 32I ISA. The test programs were generated using the RISC-V toolchain, RV8 compiler, and C Assembler compiler, ensuring RISC-V compliance. A RISC-V emulator was designed and created to evaluate SIWA's behavior; this model was validated against other emulators before its integration to the environment. The environment was also capable of acquiring performance-related information such as CPI and power per instruction. Finally, the environment was designed to generate reports for each regression and each individual test to facilitate user debugging.

For this verification framework, there are many areas of improvement; for example, the exploration and inclusion of formal verification techniques will help find bugs faster and improve the quality of the verification effort, especially the use of assertions. Also, creating specific VIPs for every I/O will certainly help reuse code between iterations of SIWA and other projects. Another aspect to consider is that SIWA is evolving, new features are getting added that require testing, the verification environment must evolve with it, and create new tests for this new realm of possibilities.

The second specific objective was to physically validate SIWA. Chapter 4 details the solution to this problem and we can conclude that a low-cost post-silicon validation framework was created to physically test

SIWA using a FPGA for emulation and support. This platform was designed and created with FPGA emulation as the foundation. An emulator was created to boot the FLASH memory, including its SPI communication protocol from the point of view of the memory as a slave. The testing platform was validated using a copy of SIWA, also emulated on FPGA, warranting its functionality before incorporating SIWA as a physical chip. Communication with a PC was established between the platform and the emulated SIWA on FPGA using other ports available in SIWA's specification. A series of software programs were created with the compiler to stimulate SIWA and visualize displays in an environment with less observability.

This post-silicon platform was indeed low-cost as it only needed an FPGA, a PC, a voltage source, and an oscilloscope, showing that teams with few members and limited resources can implement physical tests and post-silicon environment techniques. This is an interesting statement because one of the main appeals of the RISC-V architecture is its open source nature, which aligns very well with design teams with limited funding. Summing both facts together, it is possible for small teams to develop microcontrollers in both pre-silicon and post-silicon phases on a low budget.

As next steps, one can expand on the emulation of other protocols on FPGA that might become useful to properly test new features in newer versions of SIWA, such as I2C, PCIe on AXI4. It is also important to consider that with the improvements of SIWA its complexity may increase, leading to incorporating Design-For-Testability (DFT) structures inside to improve physical observability or controlability. This change would imply that the validation platform need to provide interfaces to use these features such as JTAG. Finally, it is also possible that in future projects a change to a more robust tester will be a necessity; under that circumstances the platform may need to be refashioned.

The next objective was to provide a way to exploit SIWA's capacities as a RISC-V microcontroller. For this, a low-level algorithm of a software-emulated I^{2}C controller was developed for SIWA given its lack of an I^{2}C port to communicate with I^{2}C devices. Emulation was made possible using the GPIO ports as substitutes, using 4 of the 8 available GPIO ports as signals of the I^{2}C protocol. The application followed an FSM approach, adjusting the I^{2}C master clock using SIWA's timer. This application followed a favorable software structure, using stacks and interruption routines while maintaining order between the data and the code sections in the memory map. Communication with an I^{2}C device was successfully established, in this case it was an OLED display. Other demos such as playful lights on LED displays and an UART two-way communication app were developed.

Several new features can be incorporated and expended here, such as optimizing the emulation program to reduce the states needed. In addition, the I2C protocol can be expanded to work as a slave, as it only supports working as a master now. Other types of protocols can be emulated using GPIO and following a similar FSM approach as the one proposed.

The last specific objective was related to the challenge of comparing SIWA with other microcontrollers used for implantable medical devices. For this, a benchmark for microcontrollers for implantable medical devices was created, something that according to the literature has not been done. A detailed methodology for this benchmark was explained so that other teams could reproduce it for their own comparisons. The benchmark proposes a series of program tests to measure a key aspect of implantable medical devices, power consumption. The benchmark also proposes the use of a normalized power based on MIPS to avoid errors compared to when two microcontrollers work at different frequencies; this aspect is only valid if the ratio between power consumption and frequency is linear.

The set of programs for this benchmark was designed to compare the power consumption of microcontrollers when running specific instructions based on what is typically used in implantable applications, when running a pacemaker algorithm in active mode, and when the microcontroller runs in sleep-mode. Several charts with the results for SIWA were generated. New metrics were proposed when comparing 8-bit, 16-bit, and 32-bit microcontrollers as their MIPS metric vary and no longer was a reliable common ground. These new metrics were called Cycles Per Task (CPT) and Million of Tasks Per Second (MTPS), based on their CPI and MIPS counterparts. These two new metrics helped normalize the power consumption comparison by making it independent of the different working frequencies and architectures of the subject microcontrollers. Finally, a representation of power consumption was shown when SIWA was under the influence of a simulated heartbeat.

8.1 Scientific Contributions

In the form of scientific contributions, it was already mentioned that chapters 3, 4, 5 and 6 concluded in the form of 4 publications, as the main author, in IEEE related events such as IEEE Latin American Symposium on Circuits and Systems (LASCAS), IEEE Latin American Electron Devices Conference (LAEDC), URUCON del IEEE Cono Sur Council and a journal such as IEEE Embedded System Letters (ESL). The following list contains the publications information, all of them available at IEEEXplore:

- “A compact functional verification flow for a RISC-V 32I based core" [1].
- “An affordable post-silicon testing framework applied to a RISC-V-based microcontroller" [16].
- “Low-level algorithm for a software-emulated I2C I/O module in general purpose RISC-V based microcontrollers" [39].
- “An Energy Consumption Benchmark for a Low-Power RISC-V Core Aimed at Implantable Medical Devices" [55].

There were other publications as indirect results of this thesis, publications in which I participated as a secondary author [20], [21], [22], [66].

8.2 Final Remarks

In conclusion, this thesis has demonstrated the successful testing and evaluation of SIWA, providing a comprehensive analysis of its performance, reliability, and applicability in various embedded system applications. Through the design and execution of the detailed testing framework, the study validated the core functionalities of the RISC-V architecture, highlighting its efficiency, flexibility, and low power consumption as significant advantages over other alternatives. The results confirm that RISC-V can meet the performance and power requirements for implantable medical devices, making it a viable option for use in both academic research and industrial applications after creating some important features, with the possibility of expansion to other areas such as consumer electronics and energy.

This work not only contributes to the growing body of knowledge surrounding RISC-V technology, but also establishes a foundation for future research on optimizing and enhancing testing for RISC-V microcontrollers. The insights gained from the testing processes are valuable for further innovation in the field. Even though the work continues and many things are yet to be done, the findings of this thesis serve as a stepping stone toward the broader adoption and evolution of RISC-V microcontrollers in embedded systems and the possibility to create and test impactful chips with low funds and limited resources.

Appendix A

Complementary Works

There were many complementary activities during this thesis that were not included in the previous chapters but were pivotal components in some of the results obtained, or that point to new development avenues. This chapter delves into the intricate details of these auxiliary efforts that contribute to the project's success, providing a comprehensive overview of the methodologies, tools, and results that interlink with the core initiative.

A.1 Test PCB Structure

A Printed Circuit Board (PCB) was designed to physically test SIWA. It integrates various essential components to evaluate the microcontroller's functionality comprehensively. This PCB includes a chip socket for the FLASH memory that has been mentioned across this document, which serves as non-volatile storage, allowing the microcontroller to boot from it and then store and retrieve programs or data. The FLASH memory is crucial for loading different firmware and testing various operational scenarios, which is necessary for debugging and verifying the microcontroller's performance in real time.

To ensure accurate timing and synchronization of operations, the PCB is equipped with a clock generator as SIWA needs an external clock to operate. The clock speed was designed to be 20Mhz at most for the current version of SIWA. A reset circuit is also included on the PCB, allowing developers to easily restart the microcontroller. This feature is important because it enables SIWA to boot directly from the FLASH memory.

To provide visual feedback and facilitate interaction with the microcontroller, the PCB features several Light Emitting Diodes (LEDs) and an OLED display. The LEDs can be used to indicate the status of various operations, such as power, execution status, and error states, offering a simple yet effective means of monitoring the microcontroller's behavior. The OLED display, on the other hand, can show more detailed information, such as real-time data, diagnostic messages, or user interface elements. This combination of LEDs and an OLED display enhances the user experience during testing and allows for a more intuitive understanding of the microcontroller's operational state.

Additionally, the PCB incorporates level-shifters to ensure proper communication between the microcontroller and other components that operate at different voltage levels. These level-shifters are essential when interfacing the RISC-V microcontroller with the Flash memory, OLED display, or any external peripherals that require specific voltage levels to function correctly, more specifically an FPGA. By managing voltage differences, the level-shifters protect the components from potential damage due to voltage mismatches, ensuring reliable operation and robust testing. In general, this PCB provides a comprehensive platform for evaluating and validating the performance and reliability of the developed RISC-V microcontroller in various scenarios. It was not mentioned in detail in previous chapters as its structure was not enough to create a publication, and yet it was critical to be able to test SIWA's interaction with other components.

Figure A.1 shows an image of the fabricated PCB.

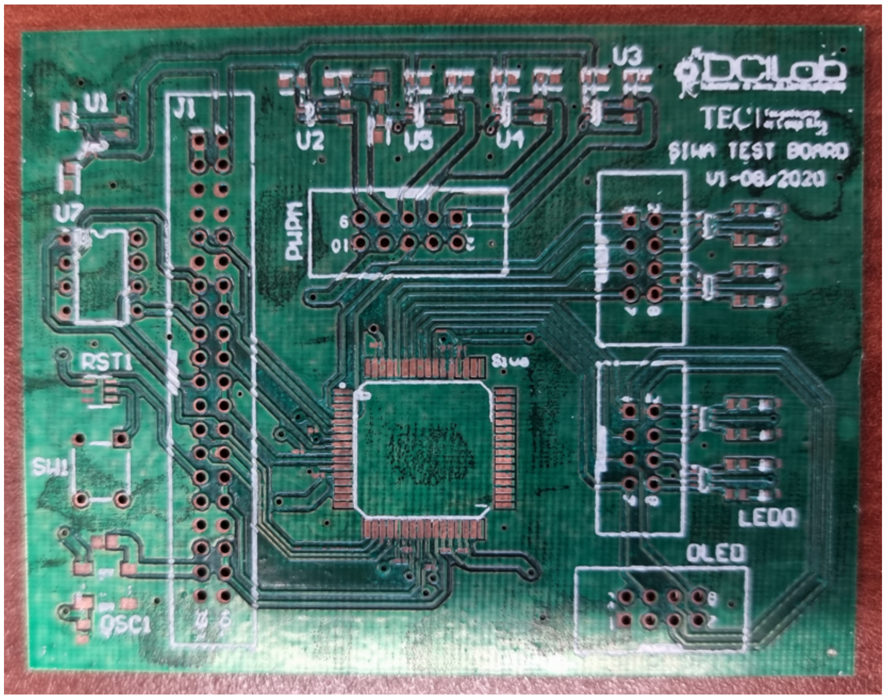


Figure A.1. Picture of the fabricated PCB.

A.2 SIWA on a dimmer application

A light dimmer is an electrical device that allows users to adjust the brightness level of a light source. By varying the amount of electrical power delivered to the light fixture, a dimmer can provide more or less illumination as needed. Traditional dimmers work by cutting a portion of the alternating-current (AC) waveform, effectively reducing the energy supplied to the light bulb. Modern dimmers often use semiconductor devices, such as triacs or transistors, to achieve more efficient and precise control over the light intensity. Light dimmers can be compatible with various types of lighting technologies, including incandescent, halogen, LED, and CFL, although compatibility varies and may require specific types of dimmers for certain bulbs.

Light dimmers are commonly used to create ambiance and mood in residential, commercial, and hospitality settings. By adjusting the lighting levels, users can tailor the environment to suit different activities, such as dimming the lights for a cozy, relaxing atmosphere or brightening them for tasks that require more visibility. Dimmers also contribute to energy savings and extend the life of light bulbs by reducing power consumption when full brightness is not necessary. In addition, in some cases, dimming can reduce the overall heat generated by lighting, which can be beneficial for maintaining a comfortable room temperature.

Using a microcontroller to create a light dimmer involves leveraging the microcontroller's ability to generate Pulse Width Modulation (PWM) signals. PWM is a technique where the microcontroller rapidly switches the power to the light source on and off at a high frequency, varying the width of the "on" pulses to control the average power delivered. By adjusting the duty cycle of these pulses (the ratio of on-time to off-time), the microcontroller can control the brightness of the light. For example, a high duty cycle results in brighter light, while a low duty cycle results in dimmer light. The microcontroller can be programmed to adjust the PWM signal based on user input, such as a knob or slider, or even automatically based on ambient light sensors or pre-defined settings.

To implement this setup, the microcontroller is connected to the light source via a suitable electronic switch, such as a transistor or a MOSFET, capable of handling the light's power requirements. The microcontroller outputs the PWM signal to the transistor gate, which then modulates the current flowing to the light source accordingly. For dimming mains-powered lights, an additional step-down circuit or isolation mechanism may be needed for safety and to match the microcontroller's low-voltage operation with the high-voltage requirements of the light source. In the case of LEDs, the microcontroller can directly drive the PWM signal to the LED driver circuit. This approach allows for precise control over lighting, enabling features such as smooth dimming transitions, programmable lighting schedules, and integration with smart home systems.

Although SIWA was created with implantable medical applications in mind, it is still a low-power general purpose microcontroller on its own. Therefore, we would like to expand SIWA area of work, and a dimmer seems like an interesting application for low-power devices. Currently, we are working on a dimmer circuit using SIWA as the microcontroller in charge of controlling the PWM. Figure A.2 shows a diagram with the idea behind this application.

A.3 SIWA as a data logger for anemometer

Anemometers are crucial for wind energy generation because they provide accurate measurements of wind speed and direction, which are essential to optimize the performance and efficiency of wind turbines. By continuously monitoring wind conditions, anemometers help select optimal sites for wind farms, ensuring that turbines are placed where they can capture the most wind energy. In addition, they allow real-time adjustments to turbine operations, such as blade pitch and yaw, to maximize energy output and prevent potential damage during extreme wind conditions. These precise wind data are also vital to predict power production, schedule maintenance, and effectively integrate wind energy into the power grid, ultimately contributing to the reliability and sustainability of wind energy as a renewable resource.

Digital data loggers are devices that record and store data over time, providing essential information for various applications, including meteorology, environmental studies, and wind energy assessment. These loggers are typically equipped with sensors that capture wind-related data at regular intervals, and store this

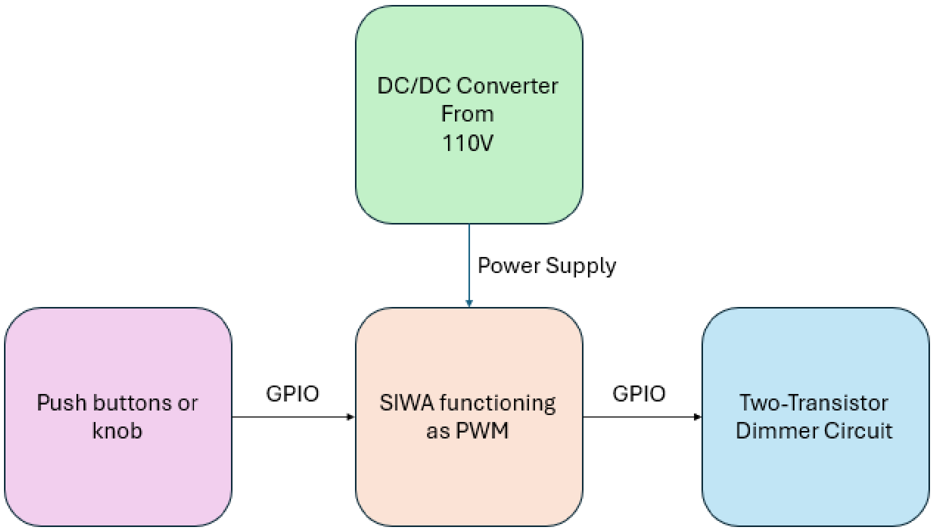


Figure A.2. Block diagram of the idea behind the dimmer application using SIWA as the main microcontroller.

information in internal memory for later retrieval and analysis. By offering precise, time-stamped records, digital data loggers allow for continuous monitoring of wind patterns, helping to identify trends, extreme weather events, and potential changes in climate conditions. In the context of wind energy, data loggers play a vital role in evaluating the feasibility of wind farm locations by providing accurate real-time data that can be used to make informed operational decisions. Their reliability, ease of use, and ability to function in harsh weather conditions make them indispensable tools in the field of wind measurement and analysis.

Microcontrollers can be used effectively to create digital data loggers by leveraging their processing capabilities, input/output interfaces, and onboard memory to collect, store, and manage data from various sensors. In a typical setup, a microcontroller is connected to sensors that monitor environmental parameters such as temperature, humidity, or wind speed. The microcontroller reads sensor data at predefined intervals and processes them, possibly applying filtering or averaging techniques to enhance accuracy. This data is then time-stamped using a real-time clock module and stored in the microcontroller's internal memory or on external storage media like SD cards for long-term logging. The compact size, low power consumption, and flexibility of microcontrollers make them ideal for data logging applications, especially in remote or battery-operated settings. In addition, microcontrollers can be programmed to communicate the logged data to a central system via wired or wireless interfaces, facilitating remote monitoring and analysis. This versatility makes them valuable tools in numerous fields, from scientific research and environmental monitoring to industrial automation and control.

Once again, we want SIWA, as a RISCV microcontroller, to be used in other low-power applications, in addition to medical ones. Wind farms in remote areas are in need of low-power data loggers to capture anemometer information, and SIWA might just be a possible solution for this. Therefore, as of the date of this document, we are currently on this application using SIWA as the main core, and Figure A.3 shows the diagram for the application.

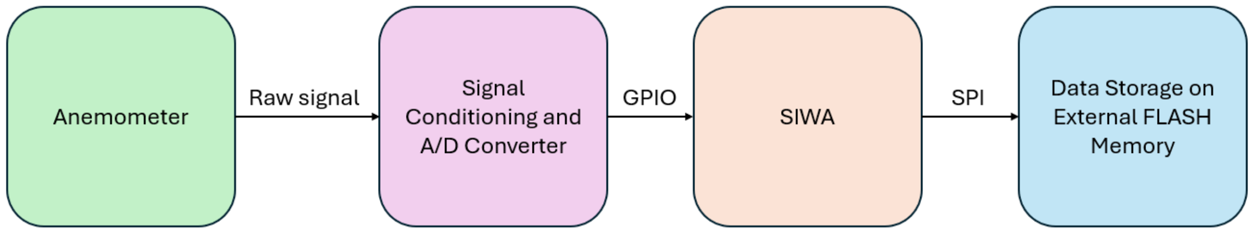

Anemometer
Raw signal
Signal Conditioning and A/D Converter
GPIO
SIWA
SPI
Data Storage on External FLASH Memory

Figure A.3. Block diagram of a data logger for wind farms in remote locations using SIWA as the micro-controller.

A.4 Exploring Formal Verification on parts of SIWA

Microcontroller applications aside, further improving our capabilities in terms on pre-silicon validation is also critical for us to develop reliable new devices in the future. Formal verification is one of those areas and we are using different components of SIWA to examine the possibilities of using formal verification in future iterations of SIWA or other new IPs.

Formal verification is a process that is used to rigorously prove the correctness of systems, particularly hardware and software, using mathematical methods. Unlike traditional testing, which checks a system's behavior under a set of predefined conditions, formal verification exhaustively examines all possible states and behaviors of a system against its specifications. This is achieved by modeling the system and using logical reasoning techniques, such as theorem proving and model checking, to ensure that the system adheres to its intended properties, such as safety, security, and functionality. Formal verification is especially important in critical applications where failures can have severe consequences. However, there are limitations to which components can be subject to formal verification as it requires large quantities of computing memory.

The importance of formal verification lies in its ability to detect and eliminate errors that might not be uncovered through conventional verification methods, especially in complex systems with a large number of possible states. By providing a mathematical guarantee of correctness, formal verification improves the reliability and robustness of the systems, reducing the risk of failures and the associated costs of fixing errors after the deployment.

SystemVerilog Assertions (SVA) are a powerful feature of the SystemVerilog language used to specify and verify the behavior of digital designs. Assertions are statements that describe expected conditions or properties in a hardware design that must always be true at specific points during simulation or in formal verification testbenches. They can be used to check for correct behavior, identify violations, and flag unexpected conditions in the design. SVAs are typically categorized into two types: immediate assertions, which are evaluated instantly and check the state of the design at a specific time, and concurrent assertions, which monitor sequences of events over time. By embedding these assertions directly into the design code, engineers can create self-checking mechanisms that continuously validate the functionality of the design, ensuring that it adheres to its specifications.

In formal verification, SVAs are utilized to define the properties that a design must satisfy, serving as the basis for formal analysis tools to verify the correctness of the design. These tools use assertions to explore all possible states and input combinations of the design systematically, checking that the asserted properties hold for every scenario. By doing so, formal verification with SVAs can uncover subtle design bugs that might not surface during conventional simulation-based testing, especially in complex designs with a vast state space. Assertions also facilitate early bug detection and can help identify the exact conditions under which a failure occurs, thereby streamlining the debugging process. The integration of SVAs in both simulation and formal verification environments provides a unified approach to design validation, improving the reliability and robustness of digital systems.

Equivalence checking is a formal verification technique used to ensure that two representations of a system, such as a high-level specification and its implementation, exhibit the same behavior and thus are functionally equivalent. This technique involves comparing the output of the high-level model with the output of the low-level implementation across all possible inputs and states. Equivalence checking is typically applied to verify that optimizations or transformations performed on the design (such as synthesis from RTL to gate level) have not altered its intended functionality. By using mathematical algorithms to exhaustively check that both models satisfy the same set of properties, equivalence checking provides a rigorous assurance that the design modifications have preserved correctness. This method is crucial in hardware verification, where even minor discrepancies can lead to critical failures, ensuring that the final product matches the initial specifications and performs reliably in its intended application.

SAT (Satisfiability) solving and SMT (Satisfiability Modulo Theories) solving are techniques used in formal verification and automated reasoning to determine whether logical formulas can be satisfied under given constraints. SAT solving focuses on Boolean formulas, determining if there exists an assignment of truth values to variables that makes the entire formula true. SMT solving extends SAT solving by incorporating additional theories, such as arithmetic, arrays, or bit vectors, allowing it to handle more complex logical formulas that involve these theories. SMT solvers provide a richer framework for expressing and solving problems that involve a combination of Boolean logic and mathematical constraints. Both SAT and SMT solvers are important tools in formal verification, enabling the rigorous verification of properties, detecting inconsistencies, and ensuring the correctness of designs and systems across a broad range of applications.

We have been using these techniques on various internal blocks of SIWA, such as the bus, the memory, and the memory bus-controller, depending on what is available by the EDA vendors and the characteristics of the blocks. We have yet to cover a lot of ground in this field, but we will without a doubt help to make our designs more robust.

A.5 Fabricanting SIWA on a smaller technologies

Another line of work on this big SIWA project is improving its performance, power consumption and size by making adjustments on its microarchitecture and fabricating new versions on smaller technologies.

Improving a microcontroller's microarchitecture can significantly enhance its performance and reduce power consumption by optimizing how it processes instructions and manages resources. Enhancements such as more efficient instruction pipelines, improved branch prediction, and advanced caching techniques allow the microcontroller to execute instructions more quickly and with fewer delays. For instance, by implementing out-of-order execution or increasing the depth of pipelines, a microcontroller can better utilize its computational resources, reducing idle times and increasing overall throughput. However, all these performance features might induce a higher power consumption, and one of SIWA's appeals is to be a low-power device.

Additionally, microarchitectural improvements can lead to more efficient power management by incorporating features such as dynamic voltage and frequency scaling (DVFS) or power gating, which adjust the power consumption based on the current workload. These optimizations help minimize energy usage during low activity periods, extending battery life, and reducing thermal output. Overall, refining a microcontroller's microarchitecture results in a balance between enhanced processing capabilities and energy efficiency, making it more suitable for a wide range of applications.

Fabricating a microcontroller using smaller semiconductor process technologies generally leads to better performance and reduced power consumption due to several factors. Smaller process nodes allow for greater transistor density, which means more transistors can be integrated onto a single chip, enabling higher performance and more complex functionalities within the same physical footprint. These smaller transistors also switch faster and consume less power because they require lower voltages to operate, thus reducing overall power consumption and heat generation. However, there are trade-offs to consider. As process technologies shrink, the challenges associated with manufacturing increase, leading to higher production costs and complexity. In addition, smaller transistors are more susceptible to leakage currents and variability, which can affect reliability and yield. Balancing these benefits and drawbacks is crucial for optimizing SIWA's design.

The first and second iterations of SIWA were manufactured with XFAB 180nm technology. The information found in this dissertation was built using these versions of SIWA. We are currently working on rerunning some of the tests and benchmarks we have made on a third iteration of SIWA with new features.

We hope this appendix helps the reader to have more context about how the work explained through this dissertation was done, as this section gives more insight on the tools we used to develop our project and what is the direction we are aiming from here onward.

Bibliography

[1] R. Molina-Robles et al. “A compact functional verification flow for a RISC-V 32I based core”. In: *2020 IEEE 3rd Conference on PhD Research in Microelectronics and Electronics in Latin America (PRIME-LA)*. 2020, pp. 1–4. DOI: `10.1109/PRIME-LA47693.2020.9062717`.

[2] *Universal Verification Methodology (UVM) 1.2 User's Guide*. Accellera. 2015.

[3] Doulos. *UVM KnowHow*. 2019. URL: `https://www.doulos.com/knowhow/sysverilog/uvm/`.

[4] R. Yang et al. “The research and implement of an advanced function coverage based verification environment”. In: *2007 7th International Conference on ASIC*. Oct. 2007, pp. 1253–1256. DOI: `10.1109/ICASIC.2007.4415863`.

[5] V. B and B. Bala Tripura Sundari. “UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol”. In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Sept. 2018, pp. 307–310. DOI: `10.1109/ICACCI.2018.8554919`.

[6] T. M. Pavithran and R. Bhakthavatchalu. “UVM based testbench architecture for logic sub-system verification”. In: *2017 International Conference on Technological Advancements in Power and Energy (TAP Energy)*. Dec. 2017, pp. 1–5. DOI: `10.1109/TAPENERGY.2017.8397323`.

[7] R. García et al. “Siwa: a RISC-V Platform in a 0.18μm HV CMOS Process for Implantable Medical Devices”. submitted.

[8] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Specification. SiFive Inc. URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`.

[9] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Specification. SiFive Inc. URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf`.

[10] “IEEE Standard for the Functional Verification Language e”. In: *IEEE Std 1647-2016 (Revision of IEEE Std 1647-2011)* (Jan. 2017), pp. 1–558. DOI: `10.1109/IEEESTD.2017.7805158`.

[11] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 0127518037.

[12] *RISC-V Simulator for x86-64*. URL: `https://rv8.io/` (visited on 10/26/2019).

[13] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. An optional note. SiFive Inc. CS Division, EECS Department, University of California, Berkeley, May 2017.

[14] *RISC-V GNU Toolchain*. URL: `https://github.com/sifive/riscv-gnu-toolchain` (visited on 10/26/2019).

[15] *RISC-V ELF to HEX Converter*. URL: `https://github.com/sifive/elf2hex` (visited on 10/26/2019).

[16] Roberto Molina-Robles et al. "An affordable post-silicon testing framework applied to a RISC-V based microcontroller". In: *2021 IEEE Latin America Electron Devices Conference (LAEDC)*. 2021, pp. 1–5. DOI: `10.1109/LAEDC51812.2021.9437939`.

[17] D. K. Dennis et al. "Single cycle RISC-V micro architecture processor and its FPGA prototype". In: *2017 7th International Symposium on Embedded Computing and System Design (ISED)*. Dec. 2017, pp. 1–5. DOI: `10.1109/ISED.2017.8303926`.

[18] R. Höller et al. "Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation". In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. June 2019, pp. 1–6. DOI: `10.1109/MECO.2019.8760205`.

[19] G. Zhang et al. "A RISC-V based hardware accelerator designed for Yolo object detection system". In: *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*. 2019, pp. 9–11. DOI: `10.1109/ICIASE45644.2019.9074051`.

[20] R. Garcia-Ramirez et al. "Siwa: a RISC-V RV32I based Micro-Controller for Implantable Medical Applications". In: *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*. 2020, pp. 1–4. DOI: `10.1109/LASCAS45839.2020.9068952`.

[21] Ronny Garcia-Ramirez et al. "Siwa: A custom RISC-V based system on chip (SOC) for low power medical applications". In: *Microelectronics Journal* 98 (2020), p. 104753. ISSN: 0026-2692. DOI: `https://doi.org/10.1016/j.mejo.2020.104753`. URL: `http://www.sciencedirect.com/science/article/pii/S0026269219303787`.

[22] A. Arnaud et al. "A RISC-V Based Medical Implantable SoC for High Voltage and Current Tissue Stimulus". In: *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*. 2020, pp. 1–4. DOI: `10.1109/LASCAS45839.2020.9068969`.

[23] M. Chupilko, A. Kamkin, and A. Protsenko. "Open-Source Validation Suite for RISC-V". In: *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*. 2019, pp. 7–12. DOI: `10.1109/MTV48867.2019.00010`.

[24] A. Munir et al. "Fast Reliable Verification Methodology for RISC-V Without a Reference Model". In: *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2018, pp. 12–17. DOI: `10.1109/MTV.2018.00012`.

[25] A. Oleksiak et al. "Design and Verification Environment for RISC-V Processor Cores". In: *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*. 2019, pp. 206–209. DOI: `10.23919/MIXDES.2019.8787108`.

[26] W. Ramirez, M. Sarmiento, and E. Roa. "A Flexible Debugger for a RISC-V Based 32-bit System-on-Chip". In: *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*. 2020, pp. 1–4. DOI: `10.1109/LASCAS45839.2020.9068995`.

[27] P. Moharikar and J. Guddeti. “Automated test generation for post silicon microcontroller validation”. In: *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 2017, pp. 45–52. DOI: 10.1109/HLDVT.2017.8167462.

[28] B. W. Mammo et al. “Post-Silicon Validation of Multiprocessor Memory Consistency”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.6 (2015), pp. 1027–1037. DOI: 10.1109/TCAD.2015.2402171.

[29] F. Moraes et al. “A generic FPGA emulation framework”. In: *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*. 2012, pp. 233–236. DOI: 10.1109/ICECS.2012.6463758.

[30] A. Koczor et al. “Verification approach based on emulation technology”. In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2016, pp. 1–6. DOI: 10.1109/DDECS.2016.7482447.

[31] K. Hayama, T. Matsumoto, and Y. Shimada. “Design and implementation of emulator on FPGA”. In: *2007 IEEE International Conference on Mechatronics*. 2007, pp. 1–5. DOI: 10.1109/ICMECH.2007.4279986.

[32] W. Y. Lo, C. S. Choy, and C. F. Chan. “Hardware emulation board based on FPGAs and programmable interconnections”. In: *Proceedings of IEEE 5th International Workshop on Rapid System Prototyping*. 1994, pp. 126–130. DOI: 10.1109/IWRSP.1994.315903.

[33] A. A. Bayrakci. “ELATE: Embedded low cost automatic test equipment for FPGA based testing of digital circuits”. In: *2017 10th International Conference on Electrical and Electronics Engineering (ELECO)*. 2017, pp. 1281–1285.

[34] D. Jadaan, K. Gonsholt, and A. Skavhaug. “Low-Cost Platform-Independent FPGA Based Real-Time Systems Tester”. In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, pp. 686–689. DOI: 10.1109/DSD.2016.67.

[35] W. Huang et al. “An FPGA-Based Data Receiver for Digital IC Testing”. In: *2019 IEEE International Test Conference in Asia (ITC-Asia)*. 2019, pp. 25–30. DOI: 10.1109/ITC-Asia.2019.00018.

[36] A. W. Ruan et al. “An automatic test approach for field programmable gate array (FPGA)”. In: *Proceedings of the 2009 12th International Symposium on Integrated Circuits*. Dec. 2009, pp. 474–477.

[37] F. Leens. “An introduction to I2C and SPI protocols”. In: *IEEE Instrumentation Measurement Magazine* 12.1 (Feb. 2009), pp. 8–13. DOI: 10.1109/MIM.2009.4762946.

[38] Integrated Silicon Solution Inc. *32MB SERIAL FLASH MEMORY WITH 133MHZ MULTI I/O SPI I& QUAD I/O QPI DTR INTERFACE*. Datasheet. 2019. URL: http://www.issi.com/WW/pdf/25LP-WP032D.pdf.

[39] Roberto Molina-Robles et al. “Low-level algorithm for a software-emulated I2C I/O module in general purpose RISC-V based microcontrollers”. In: *2021 IEEE URUCON*. 2021, pp. 90–94. DOI: 10.1109/URUCON53396.2021.9647309.

[40] NXP Semiconductors. *The I2C-Bus Specification and User Manual*. Standard. 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[41] Yoshiki Kuwabara, Tetsuya Yokotani, and Hiroaki Mukai. “Hardware emulation of IoT devices and verification of application behavior”. In: *2017 23rd Asia-Pacific Conference on Communications (APCC)*. 2017, pp. 1–6. DOI: 10.23919/APCC.2017.8304040.

[42] Sichun Zhang et al. “Hardware software co-design of pipelined instruction decoder in system emulation”. In: *2013 IEEE 4th International Conference on Software Engineering and Service Science*. 2013, pp. 149–153. DOI: 10.1109/ICSESS.2013.6615276.

[43] Tao Li and Qiang Liu. “Cost Effective Partial Scan for Hardware Emulation”. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 131–134. DOI: 10.1109/FCCM.2016.39.

[44] Mostafa Khamis et al. “A Configurable RISC-V for NoC-Based MPSoCs: A Framework for Hardware Emulation”. In: *2018 11th International Workshop on Network on Chip Architectures (NoCArc)*. 2018, pp. 1–6. DOI: 10.1109/NOCARC.2018.8541158.

[45] R. Shantha Selva Kumari and C. Gayathri. “Interfacing of MEMS motion sensor with FPGA using I2C protocol”. In: *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*. 2017, pp. 1–5. DOI: 10.1109/ICIIECS.2017.8275932.

[46] Prasanna Bagdalkar and Layak Ali. “Interfacing of light sensor with FPGA using I2C bus”. In: *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. 2020, pp. 843–846. DOI: 10.1109/ICACCS48705.2020.9074372.

[47] Vrushali S. Katkar, Divya K. Shah, and Shweta S. Ashtekar. “FPGA Implementation of I2C Based Networking System for Secure Data Transmission”. In: *2019 International Conference on Advances in Computing, Communication and Control (ICAC3)*. 2019, pp. 1–5. DOI: 10.1109/ICAC347590.2019.9036785.

[48] K. B. Bharath, K. V. Kumaraswamy, and Roopa K Swamy. “Design of arbitrated I2C protocol with DO-254 compliance”. In: *2016 International Conference on Emerging Technological Trends (ICETT)*. 2016, pp. 1–5. DOI: 10.1109/ICETT.2016.7873672.

[49] Tommaso Addabbo et al. “Using the I2C bus to set up Long Range Wired Sensor and Actuator Networks in Smart Buildings”. In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. 2019, pp. 1–8. DOI: 10.1109/CCCS.2019.8888085.

[50] J.W. Bruce, M.A. Gray, and R.F. Follett. “Personal digital assistant (PDA) based I2C bus analysis”. In: *IEEE Transactions on Consumer Electronics* 49.4 (2003), pp. 1482–1487. DOI: 10.1109/TCE.2003.1261257.

[51] Juraj Ďuďák et al. “Application of open source software on arm platform for data collection and processing”. In: *14th International Conference Mechatronika*. 2011, pp. 76–78. DOI: 10.1109/MECHATRON.2011.5961090.

[52] Texas Instruments. *Software I2C on MSP430™ MCUs*. 2018. URL: https://www.ti.com/lit/pdf/slaa703.

[53] Microchip. *Using a PIC16C5X as a Smart I2C™ Peripheral*. 1997. URL: http://ww1.microchip.com/downloads/en/AppNotes/00541e.pdf.

[54] Intel. *How to Implement I2C Serial Communication Using Intel MCS-51 Microcontrollers*. 1993. URL: http://ww1.microchip.com/downloads/en/AppNotes/00541e.pdf.

[55] Roberto Molina-Robles et al. “An Energy Consumption Benchmark for a Low-Power RISC-V Core Aimed at Implantable Medical Devices”. In: *IEEE Embedded Systems Letters* 15.2 (2023), pp. 57–60. DOI: 10.1109/LES.2022.3190063.

[56] Yongfu Li, Yong Lian, and Valerio Perez. “Design optimization for an 8-bit microcontroller in wireless biomédical sensors”. In: *2009 IEEE Biomedical Circuits and Systems Conference*. 2009, pp. 33–36. DOI: 10.1109/BIOCAS.2009.5372090.

[57] Mostafa Sayahkarajy et al. “Design of a microcontroller-based artificial pacemaker: An internal pacing device”. In: *2017 International Conference on Robotics, Automation and Sciences (ICORAS)*. 2017, pp. 1–5. DOI: 10.1109/ICORAS.2017.8308062.

[58] Longjian Xu, Houwu Zhang, and Kaixue Yao. “The analysis and design of diphasic pacemaker pulse system based on microcontroller”. In: *Proceedings of the 10th World Congress on Intelligent Control and Automation*. 2012, pp. 1192–1195. DOI: 10.1109/WCICA.2012.6358062.

[59] W.J. Price. “A benchmark tutorial”. In: *IEEE Micro* 9.5 (1989), pp. 28–43. DOI: 10.1109/40.45825.

[60] Soeharwinto et al. “Benchmarking of Wireless Microcontroller-based Three Phase Multi Meter With Industrial Standard Instrument”. In: *2019 3rd International Conference on Electrical, Telecommunication and Computer Engineering (ELTICOM)*. 2019, pp. 98–101. DOI: 10.1109/ELTICOM47379.2019.8943844.

[61] Krishna P. Gnawali, Heather M. Quinn, and Spyros Tragoudas. “Developing Benchmarks for Radiation Testing of Microcontroller Arithmetic Units Using ATPG”. In: *IEEE Transactions on Nuclear Science* 68.5 (2021), pp. 857–864. DOI: 10.1109/TNS.2021.3072861.

[62] Klaus-Dietrich Kramer, Thomas Stolze, and Thomas Banse. “Benchmarks to Find the Optimal Microcontroller-Architecture”. In: *2009 WRI World Congress on Computer Science and Information Engineering*. Vol. 2. 2009, pp. 102–105. DOI: 10.1109/CSIE.2009.928.

[63] *HiLetgo CH341A STC Flash 24-25 EEPROM BIOS Writer USB Programmer SPI USB to TTL*. Checked on January 10th, 2022. URL: https://www.amazon.com/HiLetgo-Programador-CH341A-Burner-EEPROM/dp/B014VSGH4Y.

[64] Martin Götz et al. “Benchmarking-Based Investigation on Energy Efficiency of Low-Power Microcontrollers”. In: *IEEE Transactions on Instrumentation and Measurement* 69.10 (2020), pp. 7505–7512. DOI: 10.1109/TIM.2020.2982810.

[65] NXP Semiconductors. *CENELEC - EN 45502-2-1: Active implantable medical devices: Particular requirements for active implantable medical devices intended to treat bradyarrhythmia (cardiac pacemakers)*. Standard. 2014.

[66] Leonardo Agis et al. “A SoC platform in CMOS-HV technology aimed at implantable medical devices.” In: *2023 IEEE 3rd Colombian BioCAS Workshop*. 2023, pp. 1–6. DOI: 10.1109/ColBioCAS59270.2023.10280999.