TÉCNICO LISBOA

Development Environment for a RISC-V Processor

António Pedro Charana e Silva

Introduction to Research in

Electrical and Computer Engineering

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

Examination Committee

Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Marcelino Bicho dos Santos

January 2019

Abstract

CPUs are extremely complex systems which take several years to develop and need extraordinary amounts of capital investment. For a long time, only large companies could afford creating their own CPUs, which they could either integrate in other more complex systems they build or license to other companies who use them to develop their own systems. However, thanks to a large open source community, it is also becoming possible for smaller companies to build their own systems using free and high quality hardware and software components that are typically available in repositories hosted in web-based platforms like GitHub and Bitbucket. The largest initiative so far to develop an open source processor and respective ecosystem is arguably the RISC-V Instruction Set Architecture (ISA), whose ambition is to become the standard ISA for all computing devices, from microcontrollers to supercomputers. This paper presents a development environment to create open source Systems on Chip (SoCs) that use the PicoRV32 RISC-V core architecture, while providing a base SoC called IObSoC to exemplify the development process and to serve as a starting point for future development of SoCs of the same kind.

Keywords: RISC-V, Development Environment, Open Source, SoC

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the advent of portable devices such as smartphones, tablets and others, there has been a high demand for low power CPUs, in order to meet stringent battery life specifications. As the world market of portables devices grows and the Internet of Things (IoT) emerges, hardware developers need to focus more and more on efficient low power processor cores to be integrated in SoCs that have very narrow specifications for power, area and performance.

Having just one or two providers of CPU cores, such as ARM, cannot satisfy the enormous appetite of many smaller businesses and consultancies who may and effectively can provide services in this domain. Therefore, open source processor cores are the way to go since the CPU is one of the most complex parts of the system but is also a commodity which, by itself, adds little value to the final product. In this context, after previous attempts such as the OpenRISC experience, the RISC-V architecture finally emerged and promises to become for hardware what the Linux operating system is for software. The Internet of Things is a promising market for the future and is one of the main targets of many innovative startups that use open source hardware and software components for development. SoCs are typically embedded in larger systems, so companies nowadays tend to specialize their development in very particular systems and/or applications, which are then licensed to other companies as intellectual property (IP).

1.2 Problem

Having access to a complete enough set of hardware and software building blocks to build useful systems is either too expensive or time consuming, which is a huge barrier for startups to initiate their business. In the prequel of the IoT era, the semiconductor IP market must change in order to allow smaller startups to have some share of it. The entry of such new companies in this market is imperative because the size of the future IoT market will simply be too big for only a few large companies in the world to satisfy the brutal global demand on extremely dedicated and complex IoT systems.

However, this also means that the IoT market will be extremely competitive. For a company to survive in it, it will need to be able to produce high quality hardware and software systems in a short window of time and budget, which may seem a difficult task for small startups.

1.3 Solution

Therefore, for the IoT revolution to finally trigger off, it is indispensable that an Instruction Set Architecture (ISA) free of intellectual property and licensing fees is made available to companies and their engineers, so that they can fully unleash their creativity in designing innovative systems without having to worry about such legal aspects of the semiconductor IP market and consuming the scarce funds they have available [1]. Such an ISA is provided by the RISC-V initiative.

Hence, the creation of a development environment for SoCs using RISC-V cores is an asset in a such a company, as it saves time wasted in developing these extremely dedicated systems which would otherwise take too much time to develop if an open source CPU was not made available in the first place. This way, the majority of the company's resources can be focused on developing higher level and innovative systems for the particular application they commit to work on, without wasting time and money creating a processor core from scratch or paying companies like ARM to license them their CPUs.

Previous attempts to create SoCs along with their respective development environments have been made by the same research team that currently is hosting the author and that has spun off company IObundle, Lda, also a stakeholder in this project. The first approach was made in [2], using an OpenRISC open source processor to build an SoC called Blackbird and a programming environment with the purpose of building reconfigurable systems based on it. However, this OpenRISC approach was eventually put aside due to the lack of some essential parts of the system, which made the team realize that it was not worth the effort.

Recently, a new attempt was made in [3], which introduced Warpbird, whose ambition was to become one of the first untethered SoCs built with open source components. Realizing the limitations of the OpenRISC initiative, the team opted to use the open source RISC-V toolchain. Concretely, they used the Chisel3 Hardware Description Language (HDL) [4], allowing a higher-level description of complex systems, and the Rocket Chip generator [5], which converts the Chisel3 description of a system to Verilog code, which in turn is converted to a C++ or SystemC equivalent cycle-accurate behavioral model using Verilator [6], useful for simulation purposes. The use of the Rocket (Chip) framework reduced the effort of building a RISC-V based SoC significantly. However, the project budget ran out before its conclusion because the fundamental components of the SoC, after being synthesized and implemented down to bitstream, faced some unexpected problems in the FPGA emulation phase. There were also some peripherals that were still in development which ultimately remained unfinished.

The work to be done and described in this document can be considered a continuation of what was done in [2] and [3]. A new solution using a different RISC-V core architecture will be considered, as well as an Ethernet interface instead of a JTAG module in order to speed up program data transfers from the host computer to the base SoC, which will be named IObSoC for future reference in this document.

1.4 Objectives

This project's main objective is to create an environment which allows a company to fully develop SoCs using RISC-V processor cores and other open source components in a reasonable time frame, such that it does not have to license these IP from third parties, and is more able to develop their systems in time.

In order to achieve this goal, the first step to take is to build the IObSoC System on Chip using the size-optimized RISC-V CPU known as PicoRV32 [7]. Afterwards, it will be necessary to understand how to add, remove and/or modify hardware and software components and develop an environment which allows this process to be made with the maximum ease possible, so to assure that developers can work fast. Then, a test environment is to be built so that the components added to the SoC can be properly tested with RTL simulation and FPGA emulation. Finally, both these environments should be integrated in a single development environment that comprises the entire flow of the SoC's building process.

1.5 Author's work

The work to be done during the master thesis will be the continuation of the works presented in [2, 3], which were carried out at IObundle, Lda, a Portuguese company based in Lisbon and spun off by the INESC-ID research institute, and that designs semiconductor IP. The author started a professional internship at IObundle in October 2018, where he has done research on the RISC-V ISA and open source RISC-V processor cores that will be useful to build the IObSoC System on Chip and the development environment during the dissertation project. The author aided the Versat [8] team at IObundle to attach a UART core to the Versat architecture, managed to install the toolchain necessary to use the PicoRV32 RISC-V processor architecture [7] and synthesized a simple CPU + memory system using a PicoRV32 core to have an early estimate of its size, based on the FPGA resources usage report.

1.6 Outline

This document has four more chapters besides this introductory one. A brief description of the topics addressed in each of these chapters follows.

- Chapter 2 consists on a research regarding processor architectures and SoCs. A comparison between Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) architectures is presented, highlighting the main differences between the two within the scope of SoC development and showing why RISC architectures are preferred for the goals of this project. A description of the RISC-V ISA and its main extensions is provided, while explaining which ones should be used for SoC development and why. Next, some state-of-the-art methods for building SoCs are referred and the IObSoC System on Chip is described. Its components' roles in the SoC are explained.

- Chapter 3 will cover the development environment to be built. First, the verification environment is presented and its novelties in respect to [3] are highlighted. Next, the several parts of the SoC development flow are identified and described. Throughout this description, it is referred how the toolchain is installed on the host machine, which of its parts are used in each development phase and how they are used. Next, the SoC test and verification environments developed in the previous iterations of the project at hand [2, 3] are analyzed. A base concept of the development environment to be built is presented, based on IObSoC 's structure and interfaces.
- Chapter 4 demonstrates some preliminary results of the synthesis of a simple CPU + memory system in PicoRV32's GitHub repository [7].
- Chapter 5 contains some conclusions of the research described throughout the remaining chapters of this document. A summary of the achievements made with this research is made and some future work is outlined. The planning of the tasks to be carried out in the dissertation project is also presented.

Chapter 2

SoC Design

2.1 RISC vs CISC

There are two types of ISAs: reduced and complex instruction sets. The desktop PC market has been dominated by Intel's x86 ISA, a complex instruction set architecture that was first introduced in 1978 as a 16-bit ISA. In 1985, the x86 ISA was expanded to 32-bit and in 2003 to 64-bit. However, in the embedded systems market, companies like ARM gained the upper hand with reduced instruction set processor core architectures in the 21st century.

2.1.1 RISC and CISC definitions

On the one hand, a reduced instruction set architecture is an ISA that provides a small set of simple instructions that allow the machines they are implemented in to have a reduced CPI (Cycles Per Instruction). These machines are called Reduced Instruction Set Computers (RISC). On the other hand, as opposed to reduced instruction sets, a complex instruction set architecture is an ISA were each instruction sequence of several lower level operations (or micro-operations) (for example, a memory load of one or more operands, an arithmetic operation on these and a subsequent memory store of the result). The machines that implement a complex instruction set are called Complex Instruction Set Computers (CISC[1]).

2.1.2 Performance comparison between RISC and CISC ISAs

CISC and RISC are two different approaches to tackle the performance problem in CPUs. The performance of a benchmark program can be expressed as the inverse of the time t taken for the program's execution to complete, which in turn is usually given by the Performance Equation in (2.1):

$$t = N_I \cdot CPI \cdot T_{CLK} = \frac{N_I \cdot CPI}{f_{CLK}}, \tag{2.1}$$

[1]The term CISC is usually used to refer to any non-RISC computer system. For example, a microcontroller that does not separate memory loads/stores from arithmetic operations can be labeled as CISC.

where:

- N_I is the number of instructions of the benchmark program.
- CPI is the average number of clock cycles per instruction;
- $T_{CLK} = \frac{1}{f_{CLK}}$ is the clock period (i.e. the duration of one clock cycle);
- $f_{CLK} = \frac{1}{T_{CLK}}$ is clock frequency;

While the CISC approach, in order to increase performance, tries to reduce the number of instructions N_I sacrificing the number of clocks per instruction (CPI), the RISC approach does the opposite: it reduces the CPI sacrificing N_I.

CISC performance

CISC architectures often have higher CPI values because the instructions are usually multi-step sequences of simpler operations which take one clock cycle each (or a few more), resulting in instructions that take several clock cycles to execute. Because the instructions are built this way, they can be much more specific and customizable. For this reason, CISC based ISAs have a huge number of instructions, with many different sets of similar ones that differ only in very specific details. This allows for an ISA to have a very complete set of instructions, which results in extremely small and efficient assembly codes and, therefore, the reduction of the number of instructions N_I in programs.

RISC performance

RISC ISAs, as it was said in Subsection 2.1.1, are built so that its instructions are simple and executable in the minimal number of clock cycles targeting low CPI values, as opposed to CISC ISAs. A consequence of this feature is the increase of the programs' code size, as more instructions are needed to perform the same operations done by CISC instructions. This explains why N_I in RISC ISAs are higher than in CISC ISAs.

RISC vs CISC performance conclusions

As referred in [9], the differences between the CISC and RISC ISAs in performance were more important in the 1980's than in modern times, because the key constraints back then were the chip area and the processor design complexity. Nowadays, the primary constraints are low energy and power consumption, where using RISC or CISC seems irrelevant; ARM's RISC cores have penetrated the high-performance servers market (previously dominated by Intel's x86 CISC cores) while Intel's x86 CISC cores have penetrated the mobile low power devices market (previously dominated by ARM's RISC cores).

Also in [10] it is stated that the RISC vs CISC debate is of no more interest nowadays, as it mentions that today's CISC CPUs (i.e. x86 CPUs) are not true CISC cores anymore, but instead are CRISC (Complex-Reduced Instruction Set Computer) cores, which means that they still execute the x86 CISC

instruction set, but their internal implementation is based in RISC principles. This means that the internal behavior of both types of ISAs may not be so different, which may justify why this choice does not affect performance.

According to [9], the differences in performance between the ARM Cortex-A8 and Cortex-A9 and Intel Atom and Sandybridge i7 microprocessors are due to the microarchitecture features of the various cores such as:

- out-of-order execution;
- instruction-level parallelism (i.e. superscalar processors);
- cache hierarchies and policies;
- speculative execution (i.e. branch predictors);
- data-level parallelism (i.e. vector processors);
- register renaming;

The microarchitecture seems to have a significant impact on performance. In particular, in the x86 cores manage to maintain high performance chiefly due to the highly accurate branch predictor and large caches and not because of its ISA. So, the conclusion is that the RISC vs CISC performance debate is irrelevant nowadays.

2.1.3 Advantages and disadvantages of RISC and CISC cores

Although performance is not an issue in the RISC vs CISC debate, there are other factors that influence companies to opt for RISC cores instead of CISC cores in their designs.

CISC

Because there are no open source CISC cores available, a company would either have to get an x86 core license from Intel, AMD or VIA Technologies (which are the only companies in the world that manufacture x86 cores) or develop their own x86 core. Both options are too expensive for small companies and developing an x86 core has the extra cost of paying Intel a licensing fee, since they own the x86 ISA.

But even if that licensing cost was covered, developing and maintaining a CISC core and its toolchain is an overkill for smaller companies because both grow much in complexity over time, which means that adding new features and optimizing hardware and software will need more time and people. However, the degree of complexity achieved in both the CISC core hardware and the compiler (let alone the remaining toolchain components) is so high that the resources needed at a certain point would be simply too much for a small company to continue development and/or maintenance.

x86 is the only CISC ISA that survived along the years, and the only reason for that to happen is the compatibility with older legacy systems. One can even argue that pure CISC systems do not exist anymore, because like mentioned before, today's x86 systems are CISC only on the outside, while their internal implementation is RISC-based [10].

RISC

A company has much better reasons for using a RISC core and much more options available. The most obvious one is getting a licence from ARM to use one of their proprietary RISC cores. It can use an open source RISC core available in code sharing platforms like GitHub or BitBucket or it can even develop their own, as there are many open and free RISC ISAs, being RISC-V the most prominent one. From a resources point of view, a RISC core is the only option to even consider, as the downsides of CISC are too much for even considering using it.

Besides, a RISC core has much ok foless hardware (and therefore less area and power consumption) than a CISC one in small systems, because CISC needs much more instruction decode logic due to the ISA's large number of instructions. The same cannot be said for very high performance systems with much additional hardware like caches, branch predictor, register renaming and others, as these will occupy most of the chip's hardware and physical area and, therefore, will dissipate much more power. However, these are not the kind of systems that are targeted in this project, but instead the focus are low power and small area SoCs and embedded systems for IoT applications.

Also, even if a more complex instruction set is needed, one can build it on top of the open-source RISC-V ISA, as it support custom-made ISA extensions. With the due skills and knowledge, one can even tweak the compiler's source code (as it is open source) to account for pseudo-instructions that in truth consist of sequences of real instructions that are actually contemplated in the RISC-V ISA, mimicking CISC's instruction microcoding.

2.2 The RISC-V ISA

In the previous chapter, the reasons why RISC processors are more suited for SoC development and embedded systems were described. Summarizing, RISC processors generally need less hardware, occupy less area, consume less power and there are free open source ISAs and respective cores available in GitHub and other alike platforms. Indeed, most of the embedded system and SoC market consists of RISC-based solutions.

The main free open source RISC ISA nowadays is RISC-V, a free reduced instruction set architecture and surrounding software ecosystem that allows standard and custom extensions of its base ISA. There are also other alternatives for open source RISC ISAs such as OpenRISC, although it has few commercial implementations.

2.2.1 Brief History

The RISC-V project started in 2010 at the University of California, Berkeley. Since then, many contributors such as volunteers and industry workers have joined the project, building a large RISC-V community. A startup called SiFive was created in 2015 to encourage RISC-V dissemination in the semiconductor and electronic design industries. On November 29, 2016, SiFive released the first integrated circuit (IC) that implements the RISC-V ISA and in October 2017 they released the first SoC that supports fully

featured operating systems (OSs) like Linux, which validates the RISC-V ISA's potential to become an industry standard in the future.

In 2018, the RISC-V ISA continues to gain importance in the semiconductor IP industry also for large companies like NVidia and Western Digital Corp., who *"have decided to use RISC-V in their own internally developed silicon. Western Digital's chief technology officer has said that in 2019 or 2020, the company will unveil a new RISC-V processor for the more than 1 billion cores the storage firm ships each year. Likewise, Nvidia is using RISC-V for a governing microcontroller that it places on the board to manage its massively multicore graphics processors"* [11].

2.2.2 Instruction set architecture

The RISC-V ISA supports four base ISA and several standard extensions, whose names and status can be consulted in Table 2.1. It is also possible to build custom extensions for the RISC-V ISA like, for example, GPU-based instruction sets.

User-level ISA

At the time of writing of this document, the RISC-V user-level architecture is in version 2.2. This current version of the RISC-V user-level architecture manual [12] contains the following versions of the RISC-V ISA modules:

Table 2.1: The RISC-V user-level architecture, version 2.2 (taken from [12], Preface, page i).

Base	Version	Frozen?
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	**Version**	**Frozen?**
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

The currently defined extensions to the base Integer (I) ISA are:

- M, which provides integer multiplication and division instructions;

- A, which provides atomic[2] instructions;
- F, D, and Q, which provide floating point operations in single (32-bit), double (64-bit) and quadruple (128-bit) precision, respectively;
- C, which provides compressed instructions.

The other expansions contemplated in Table 2.1 are still in development.

The standard ISA's instructions are all 32-bit wide. This results in a simple implementation but in large code sizes, which, as pointed out in Section 2.1.2, typically happens in RISC architectures. To mitigate this issue, the instructions were built to use, in reality, only 30 bits, which allows for 3/4 of the opcode space to be used to address a variable-length subset of instructions.

The two remaining bits of the instruction/opcode are used to select 3 subsets[3] of smaller instructions (typically 16-bit wide[4]), which are aliases of the larger base ISA's instructions. This compressed instruction alias scheme is also used in other compressed RISC ISAs like ARM's Thumb and the MIPS16.

However, RVC (RISC-V's compressed ISA) has an advantage over the previous two, because it was built so that each of its compressed instruction expands into a single 32-bit instruction in one of the base ISAs of the F and D standard expansions, which lets regular and compressed instruction be used in a single piece of code without having to worry about operation modes like in Thumb or MIPS16.

Another advantage of this is that RVC instructions can be directly expanded in the instruction decode stage, which accounts for less and simpler hardware. This can make a RISC-V processor core much more efficient, as the instruction decode hardware is usually one of the most (if not the most) costly energy consuming parts of the processor circuit.

Having a compressed ISA also helps to reduce code size, because typically two compressed instructions can fit in the same space as a regular instruction.

In conclusion, the compressed RISC-V ISA subset (C) is particularly useful for embedded and SoC designs, because it allows for smaller code sizes and more energy efficiency [13]. Using the RV32E base ISA is also quite useful in these kinds of systems, as it was made specially for them (the E stand for "Embedded"). This base ISA uses only 16 32-bit integer registers (instead of 32 registers, like RV32I) and does not support floating point instructions (although it is possible to use floating point software libraries). The RVC extension is usually used alongside the RV32E base ISA in embedded systems and SoCs.

Privileged level ISA

There is also a privileged level instruction set in the RISC-V ISA [14], which details privileged instructions and other functionalities required for OS support and attaching external devices. This part of the RISC-

[2]Instruction that read and write in the same memory position in order to prevent other CPU core or I/O device to access it before the instruction is completed.

[3]There are 4 subset in total, corresponding to the four possible combinations of this 2-bit word. When at least one bit is 0, a compressed subset of only 16-bit wide instructions is used. All 32-bit instructions have both of these bits set to 1.

[4]When the base ISA used is RV64I or RV128I, these instruction do not necessarily use 16 bits, but usually half of the size of the base ISA's instructions.

V ISA is sill in the development stage and has not yet been accepted by the RISC-V Foundation as standard.

2.3 SoC structure

2.3.1 SoC components

A System on Chip (SoC) is an electronic system completely incorporated in a single integrated circuit (or, as it is more simplistically often referred to, a chip). Some components that make up an SoC are the same as those found in a computer system such as CPU cores, memory units and several Input and Output (I/O) ports and peripheral interfaces. Other typical components are co-processors or peripheral modules that implement dedicated functions, such as audio encoders or decoders, or even low power programmable hardware cores like CGRAs (for example, Versat [8]).

The CPU is typically the key component of this kind of systems, because it is where programs can be run. If a more specific task needs to be done, the CPU can communicate with one of its dedicated peripherals or co-processors and feed them with operands for they to process or receive their results.

If the SoC needs to receive external data or send data to the outside world, the CPU can write or read to the I/O peripheral interfaces available in the SoC, such as JTAG, Ethernet, USB, HDMI, UART, SPI, PCI Express and others. It is also possible for an SoC to use interfaces for wireless communication protocols such as WiFi and Bluetooth, although one needs to keep in mind that wireless protocols may need support of lower layer devices, which may need to be external chips due to their specificities.

An SoC typical structure is the one considered for building the IObSoC System on Chip, which uses almost every type of component mentioned in the last paragraphs. This structure is detailed in Subsection 2.4.

2.3.2 SoC intermodule communication

Global bus infrastructure

Traditionally, the various components that make an SoC communicate with each other via a bus infrastructure that consists in a global shared bus. In order to guarantee a certain throughput of data exchanges between the memory and other components, Direct Memory Access (DMA) controllers can be used to bypass the CPU, which cannot do these data transfers as fast and may be busy with other things.

This kind of intermodule communication is not scalable, because when the number of cores and/or components in an SoC rises above a certain level, the bus infrastructure needed to ensure communication between all components grows to a point where silicon area and power consumption are too high to meet the tight specifications of IoT applications.

Network on Chip

A solution to this problem is implementing network structures based on Internet protocols such as TCP and IP and routing algorithms such as Dijkstra's. This approach is called Network on Chip (NoC). This not only greatly reduces the silicon area occupied by wires connecting SoC modules and power consumption, but also improves throughput and latency [15]. Although this approach is much more interesting than traditional bus communication infrastructures for IoT applications, it also requires network knowledge such as communication protocols and routing algorithms, as well as how these can be implemented in hardware (routers). Besides, the IObSoC System on Chip will not have a large number of components, so for now the traditional bus approach will be used. However, implementing a NoC infrastructure in IObSoC is a very interesting perspective for future work.

2.4 The IObSoC System on Chip

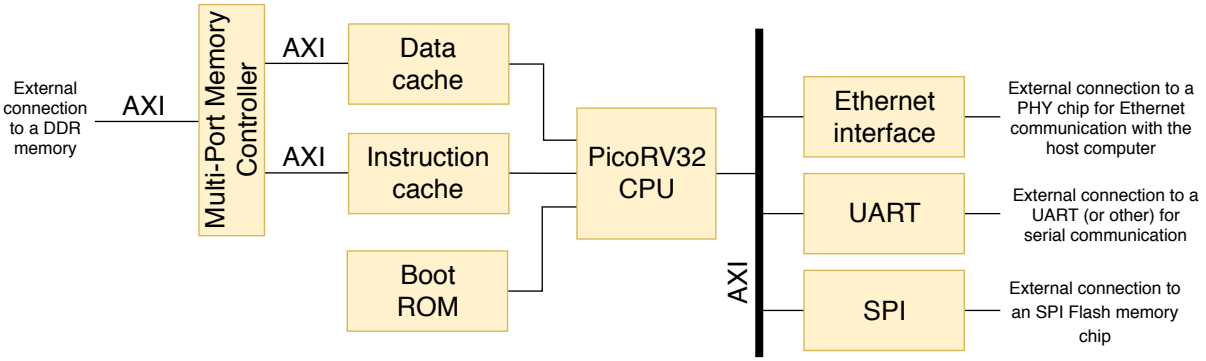

Figure 2.1: Top level block diagram of the IObSoC System on Chip.

The top-level topology of the IObSoC System on Chip is shown in Figure 2.1. IObSoC is intended to be build as a base SoC for future development of IoT hardware and software systems, which means that the key attributes of IObSoC must be small area, low power consumption and high performance. Another very important feature for the IObSoC System on Chip and for the development environment to be built during this project is configurability, so to allow developers to choose which of the specifications mentioned earlier are the most important to optimize, according to the application intended for the SoC being developed.

As such, several RISC-V CPU alternatives were taken into account when choosing which one of them was going to be used to build the IObSoC System on Chip. Those were Rocket Chip [5], Taiga [16], PULPino [17], and PicoV32 [7]. Brief descriptions of each architecture are given next, as well as the identified reasons to discard the ones that were not sufficiently fitting for the IObSoC System on Chip.

- **Rocket Chip** [5] is the de facto standard for SoC development using RISC-V processor cores, as it was made by the same team that introduced the RISC-V ISA. Rocket Chip is a synthesizable RTL generator that allows customization and parameterization of two RISC-V processor cores

architectures (one being in-order and the other out-of-order). However, due to the fact that Rocket Chip has a great amount of features, it has proved to be too hard to manipulate in the past, so this option is put aside.

- **Taiga** [16] is a high-performance RISC-V softcore written in SystemVerilog, developed mainly for use in FPGAs and for supporting multicore configurations using an OS. However, given the fact that the focus of this architecture is to solely optimize performance and resource allocation in FPGAs, it not suited for the applications envisioned for this project, as the IObSoC System on Chip is meant to reach ASIC implementation.

- **PULPino** [17] is an open source single core microcontroller system that can be configured to use either one of two different in-order and single-issue CPU architectures: RI5CY [18] or zero-riscy [19]. Both are designed to be used in ultra low power [5] designs [20].

  In particular, RI5CY has 4 pipeline stages, an IPC (Instructions Per Cycle) approximately equal to 1 [12] and full support of the RV32I base instruction set. It supports some of the RISC-V ISA extensions, such as the RV32C, RV32M and RV32F and it also supports other ISA extensions such as fixed-point operations, dot product and others. RI5CY implements a subset of the privileged specification of the 1.9 version of the RISC-V ISA [21]. As of the date of writing of this document, the latest version of the privileged architecture of the RISC-V instruction set is v1.10 [14].

  The zero-riscy architecture is derived from RI5CY CPU core. It features 2 pipeline stages and, like the RI5CY core, supports the RV32I base instruction set and the RV32C and RV32M extensions. It is also possible to reduce the number of registers to 16, configuring the zero-riscy core to adopt the RV32E RISC-V instruction set, which is useful for embedded systems. This architecture also implements a subset of the privileged specification of the 1.9 version of the RISC-V ISA, just like RI5CY.

  PULPino provides a set of peripherals such as I2S, I2C, SPI and UART for communication with external systems. PULPino, RI5CY and zero-riscy are were all written with the SystemVerilog HDL[6] (Hardware Description Language).

- **PicoRV32** [7] is a size-optimized RISC-V processor core that implements the RV32IMC instruction set. Although being small, it features a high maximum clock frequency (250-450 MHz on 7-Series Xilinx FPGAs, according to the README file in [7]). The RV32C and RV32M extensions are supported, so it is possible to configure this PicoRV32 as an RV32IM, RV32IC or RV32IMC core. It is also possible to configure it as a RV32E core. It also supports other optional useful features for SoCs and embedded systems, such as a built-in interruption controller and a co-processor interface.

So, the two contenders for being the CPU architecture in the IObSoC System on Chip are PULPino and PicoRV32. In the end, it was decided to use the PicoRV32 CPU architecture to build the IObSoC

[5] Ultra low power in eletronic circuits means that the transistors are operated with the minimal power supply possible. As such, the transistors are operated at sub-threshold regions, with ultra low threshold voltages.

[6] SystemVerilog is also a Hardware Verification Language (HVL).

System on Chip. The reasons for choosing this processor core over PULPino are the following:

1. PULPino's features, although interesting for SoC design and embedded systems, go beyond the scope of the project. Namely, it implements a subset of the RISC-V privileged ISA, which is a feature purposely left out of the project at hand because, for the time being, there is no interest in running an OS in the IObSoC System on Chip. Instead, a smaller set of simple custom instructions can be used for Interruption Request (IRQ) handling, which PicoRV32 already implements.

2. PicoRV32's GitHub repository is the most complete of the two, where there are made available some application examples of the PicoRV32 CPU, namely the PicoSoC example SoC and a simple system with just a CPU core and a memory unit connected by PicoRV32's native interface.

3. On the one hand, PicoRV32's provided testbenches in the respective GitHub repository delivered positive results. The simple CPU + memory system mentioned in the previous item was also synthesized in FPGA (details are in Chapter 4). On the other hand, there were efforts in the past to integrate PULPino in a system being developed at IObundle, but the team was incapable of detaching the PULPino core from the environment where it was demonstrated because there was not enough information available to do so.

2.4.1 CPU

The CPU is the most important component of the system. It runs programs stored in an external DRAM memory and connects to all other components[7].

The CPU architecture used in the IObSoC System of Chip is PicoRV32. It supports the RVC standard RISC-V ISA extension and the base ISA RV32E, which are important features for SoC development, as explained in Subsection 2.2.2.

Other key features of PicoRV32 are:

- Small size. When deployed on a 7-Series Xilinx FPGA, it uses between 750 and 2000 LookUp Tables (LUTs);
- Maximum frequency between 250 MHz and 450 MHz on 7-Series Xilinx FPGAs;
- CPI approximately equal to 4, but depends on the mix of instructions in the code. For more details, the README file in [7] should be consulted;
- Selectable native memory interface or AXI4-Lite master and Verilog description of an interface to use between two cores with different memory interfaces (AXI4-Lite or native);
- Optional IRQ support using a simple custom ISA;
- Optional Co-Processor Interface, which can be used to implement non-branching instructions in external co-processors, such as the M standard RISC-V ISA extension;

[7] The CPU does not connect directly to the memory controller, but instead is connected through a cache interface.

- Option to choose single-port or dual-port register file implementation, which can be useful to configure reduce the core's size or to increase it is performance;

2.4.2 Ethernet interface

Ethernet is a widely used communication protocol that was first commercially introduced in 1980. Since then, it has evolved to a point where it is possible to transfer data through Ethernet cables with bandwidths from 10 Mbit/s to 100 Gbit/s[8]. An Ethernet interface is currently in development for a separate Versat [8] project at IObundle, which is intended to be used in the IObSoC System on Chip as well.

The Ethernet interface will be used instead of a JTAG tap for communication of the IObSoC System on Chip with the host computer, with the objective of reducing drastically the time needed to load programs into the SoC and to send bitstreams from the host computers to FPGA where IObSoC is being emulated.

In the IObSoC System on Chip, the Ethernet interface will be connected to the CPU via an AXI interface.

2.4.3 UART

An Universal Asynchronous Receiver-Transmitter (UART) is needed for debug and printing purposes. A UART is a hardware device that implements the RS232 serial protocol for data communication through two serial ports: one for transmitting (often labeled as TX) and another for receiving (often labeled as RX). When data transfer is taking place, the UART outputs a start byte in the TX port, followed by the 8 bits that compose the byte that needs to be transmitted and finally a stop bit. When receiving data, it checks if a start bit in the RX port in received and, when detected, it receives the 8 bits and the stop bit and converts the received serial data into a byte. So, the UART is usually used to send char strings from the SoC to a terminal on the host computer. The UART also has a clock divider scheme to adjust its BAUD rate.

The PicoRV32 GitHub repository already has an example SoC where a UART open source Verilog description is included and connected to the CPU via the PicoRV32's native memory interface. In the IObSoC System on Chip, the UART will be connected to the CPU via an AXI interface.

2.4.4 SPI

A Serial Peripheral Interface (SPI) is also included to allow communication with an SPI Flash memory unit, which can store program data destined to be transfered to the CPU. However, the SPI can also be used to communicate with other low speed peripherals. Similarly to the UART, the SPI is also a serial interface, but while the UART transfers data asynchronously, the SPI offers synchronous communication. The SPI is also much faster transferring data than the UART, because the first can usually send data through 2, 3 or 4 wires, while the second has only one wire available for this task.

[8]Ethernet using more than 100 Gbit/s of bandwidth is called Terabit Ethernet (TbE). TbE standards for 200 Gbit/s and 400 Gbit/s Ethernet were developed by the IEEE P802.3bs Task Force and were approved on December 6, 2017. However, the term Ethernet usually refers to the 10 Mbit/s to 100 Gbit/s range, while the above ranges are referred as TbE.

An alternative to the SPI would be an I2C, which is also a synchronous serial interface. However, there are numerous reasons why the SPI is preferable. The most obvious one is that the I2C is typically slower and is more complex than the SPI, because its hardware is based on an addressing scheme and the SPI simply uses a chip select line. Also, SPI flash memory is the easiest and cheapest kind of off-chip non-volatile memory available nowadays, so it is more pertinent to consider an SPI.

The PicoRV32 GitHub repository already has an example SoC where an SPI open source Verilog description is included and connected to the CPU via the PicoRV32's native memory interface. In the IObSoC System on Chip, the SPI will be connected to the CPU via an AXI interface.

2.4.5 Boot ROM

A Read-Only Memory (ROM) is typically used in computer systems to store a boot program that is executed after each machine reset. As such, boot ROMs are connected to the system's CPU. When the reset signal is received, it the boot ROM loads the program to the CPU's memory and is then executed. During the professional internship at IObundle, the author used the picoVersat controller (i.e. Versat's [8] controller), where the boot program stored in the boot ROM was a simple loop to check if a non-zero value was written to register R0 (signaling that a program was ready to be executed) and, if it did, a branch to the program memory's base address would be made, starting the program.

In the IObSoC System on Chip, the boot ROM is connected to the CPU via PicoRV32's native memory interface.

2.4.6 Multi-Port Memory Controller

A DRAM controller is used to control data transfer between the external DRAM memory and the CPU. The DRAM controller connects to the external DRAM memory and to the data and instruction caches mentioned below through AXI interfaces, as depicted in Figure 2.1.

2.4.7 Instruction and Data Caches

Between the DRAM controller and the CPU, a data cache and an instruction cache are present, in order to increase memory access performance. Caches are a kind of smaller but faster memory used to interface a CPU and a RAM memory in order to increase memory access speed. When the CPU access new data, it first checks if the data is in the cache. If not, then the data is loaded from the RAM memory to the cache so that the next time the processor needs to access it, it can read or write it faster.

The caches are connected to the CPU via PicoRV32's native memory interface and to the DRAM controller through an AXI interface.

Chapter 3

SoC development

In this chapter, the verification environment and development flow of the IObSoC System on Chip are presented and a description of how the first can be used in the several development stages is given.

3.1 SoC verification

Most of the time that takes for an IC to be developed is spent on verification i.e. running tests to see the design was correctly built. There are no systematic/general/standard methods for validating such designs that can vary much in complexity and size. Because of this, many different verification techniques are employed to test the SoC.

3.1.1 Warpbird verification environment

Two very similar SoC verification environments that use open source components are the ones developed in [2, 3]. Figure 3.1 illustrates the one built in the Warpbird project [3], as it is the most recent one.

Warpbird's verification environment is already described in [3], so this section will only refer and briefly detail some of its parts which contrast with the verification environment envisioned for IObSoC , so to emphasize the novelties introduced in respect to Warpbird.

Warpbird uses a JTAG tap which communicates with the host computer via an FTDI bridging device that translates JTAG to USB and vice-versa. The USB side of the FTDI devide then connect to OpecnOCD [22], an universal chip debug tool that provides a unified target access mechanism, facilitating the implementation of the communication structure between clients (such as the test program and GDB) and the FPGA.

OpenOCD allows debugging of programs running remotely on the FPGA using GDB, as if those programs were being ran locally. A test program is connected via a TCP/IP socket internal to the host machine to send OpenOCD commands such as program loading, execution halt and start, etc to the FPGA. OpenOCD is also connected via an internal TCP/IP socket to Verilator [6] (SystemC & Verilog simulator) to allow debugging on the RTL simulation target as well.

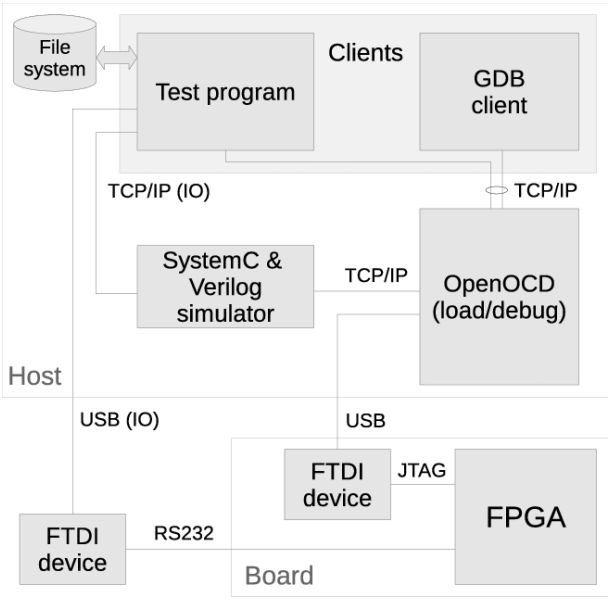

Figure 3.1: Warpbird's verification environment (figure taken from [3]).

3.1.2 IObSoC verification environment

The verification environment for IObSoC will be based on Warpbird's. However, some adaptations must be made because the toolchains are different in the two and the two SoCs do not have the same interfaces to communicate with the outside world. For example, unlike Warpbird, IObSoC does not use the Rocket Chip generator and Chisel3 scala embedded language tools and has an Ethernet interface instead of a JTAG tap, which Warpbird has.

Because the JTAG tap is not used in IObSoC , we no longer need to use OpenOCD and the FT4232H Mini Module, which was used to bridge communications from the host machine to the JTAG pins in the FPGA board, which in turn were connected to Warpbird's JTAG tap, inside the FPGA itself. The FT2232H device[1] is no longer necessary as well, because the Ethernet interface in IObSoC allows the clients (i.e. the test program and GDB) can communicate with the FPGA via Ethernet instead of converting USB streams from the host computer into RS232 streams to the FPGA and vice-versa.

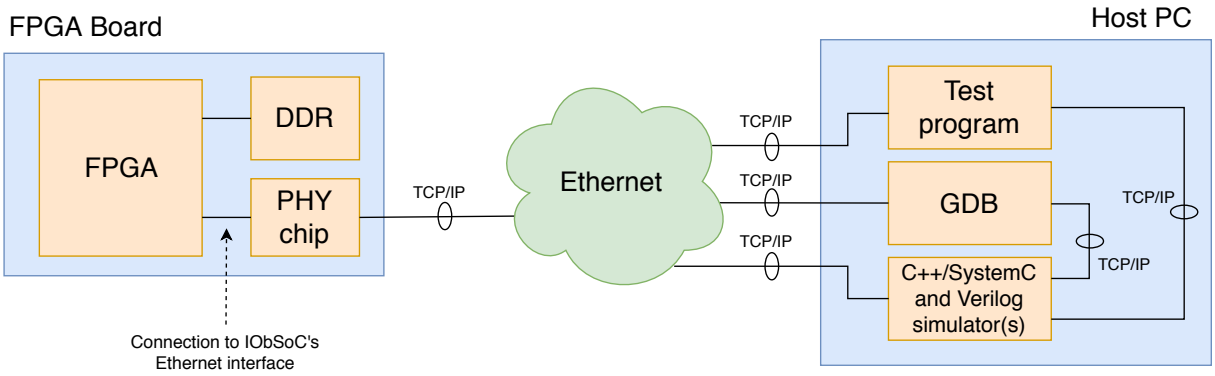

Figure 3.2: Verification environment based on IObSoC .

Therefore, all communications between the host computer and the FPGA are now done using Eth-

[1] i.e. the FTDI device in the lower left corner in Figure 3.1

ernet, which simplifies the verification environment structure, as it can be seen in Figure 3.2 compared to Figure 3.1. On top of that, Ethernet is the most popular link-layer protocol on which TCP/IP is carried out, so this Ethernet communication infrastructure is an appealing choice. Because OpenOCD no longer exists in the verification environment, the C++/SystemC and Verilog simulator(s), the test program and the GDB client communicate directly with the FPGA via a TPC/IP socket which runs on top of an Ethernet connection. Also, GDB's communication with the simulator(s) is now done via a direct internal TCP/IP socket in the host machine, just like the one that already exists between the test program and the simulator(s).

The use of this new Ethernet infrastructure not only simplifies the verification environment but also makes communications between the host machine and the FPGA (or the SoC) much faster, because Ethernet bandwidth ranges between 10 Mbit/s and 100 Gbit/s, when JTAG's typical maximum bandwidth is several tens of Mhz. Besides this new way of communication between the host PC and the FPGA, most of the remaining structure of the verification environment will be similar to Warpbird's.

After peripherals are added to IObSoC , SoC verification begins with RTL simulation, where dedicated testbenches and test programs are produced to test and evaluate it (these topics are better described in Section 3.2). The simulators considered to integrate this environment are:

- **Verilator** [6], which converts the Verilog description the SoC design to a C++/SystemC equivalent cycle-accurate behavioral model. This allows fast simulation of complex and large systems, but limits the truth values to 0 and 1, whereas other simulators use Z (high impedance), U (undefined) and others. Verilator is free and open source;
- **Icarus Verilog**, which is an open source simulation and synthesis tool more suited for small or medium-small sized projects. For medium to large-sized projects, a proprietary simulator or Verilator are better options, as Icarus becomes too slow, which may lead to project delays;
- **NCsim**, a commercial proprietary simulation engine contained in Cadence's design and verification tools suite. Although proprietary and very expensive, such a simulation tool may sometimes be necessary for large projects to be completed in a reasonable time window. Instituto Superior Técnico makes NCSim available in machines that can be remotely accessed by students, professors and researchers at the university. The licenses for NCSim are payed with the help on European Union (EU) funds.

The waveforms generated by the simulations are then viewed in a waveform viewer such as GTKWave (a free waveform viewer). If errors are encountered, the SoC design must be rectified and new RTL simulations need to be made. After obtaining the expected results, the second step of the verification process is to target an FPGA and implement the SoC. When a bitstream is successfully generated, deployment of the SoC in the FPGA is made and tests are made in the FPGA emulated SoC. Every time unexpected problems occur, the verification needs to take a step back to the previous phases until all problems are solved. In each development stage that uses the verification environment[2], after successfully deploying the SoC in the FPGA, development advances to the next stage.

[2]As it will be described in Section 3.2, all development stages except in the first one, which is the toolchain installation.

Another detail to account for is that because IObSoC does not have a debug module (unlike Warpbird), it is necessary to develop software to implement useful (and some indispensable) debug commands such as break, run, halt, continue, peek and poke. This debug program will be stored in the external DDR memory.

3.2 Development flow

The development flow in which the SoC development environment will be based is the same as the one used in [2, 3], which is illustrated in Figure 3.3.

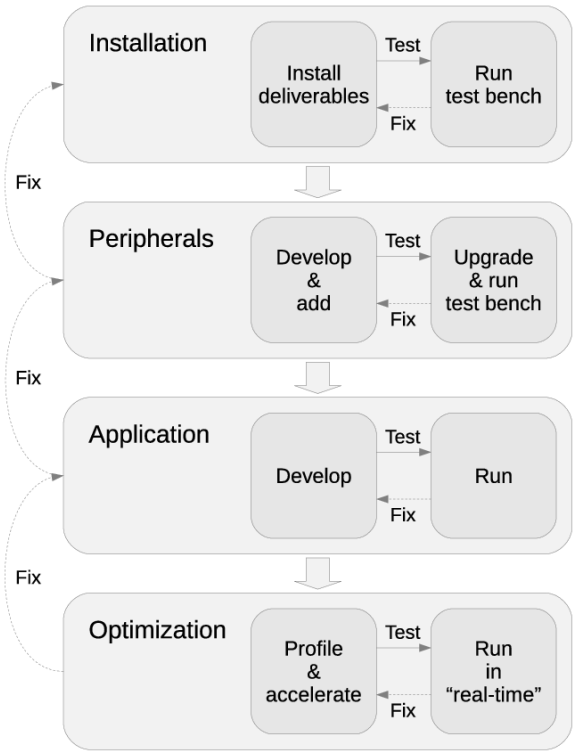

Figure 3.3: Development flow (figure taken from [2]).

3.2.1 Toolchain installation

The first stage of the development flow is the installation of the tools necessary to develop the SoC, such as compiler, assemblers, linkers, simulators and others. The PicoRV32's GitHub repository [7] provides a Makefile script to install the complete RISC-V toolchain. However, it is not necessary to install all of its tools, as some are not used by PicoRV32.

The default settings of the build scripts available in the riscv-tools repository [23] build a compiler, assembler and linker that can target any RISC-V ISA. However, the libraries are built for RV32G and RV64G targets. For this reason, the README file in PicoRV32's GitHub repository [7] also provides instructions with Linux shell and make commands to install a pure RV32I, RV32IC, RV32IM or RV32IMC CPU (including libraries). These commands include:

1. the installation of various software packages[3];

2. the creation of the installation directory inside the /opt folder;

3. the download of the RISC-V GNU toolchain from [24] using git commands;

4. building the toolchain in the host machine with a configure script.

After installing the toolchain, a testbench is ran with standard configurations to check if the former was correctly installed. The testbench uses the Icarus Verilog simulation and synthesis tool [25]. There is also a simpler testbench that does not require an external firmware .hex file, which can be useful in systems that do not have the RISC-V compiler toolchain installed.

3.2.2 Adding peripherals to IObSoC

In the second stage of development, SoC components are added to IObSoC by describing them with the Verilog HDL. The example projects' files provided in PicoRV32's GitHub repository [7] can provide helpful guidelines for this task. The peripherals to be added can be used as co-processors using PicoRV32's Pico Co-Processor Interface (PCPI) or can simply be connected to a PicoRV32 core through the AXI4-Lite memory interface provided. More PicoRV32 cores can be instantiated and interconnected using its native memory interface.

The standard RISC-V ISA extensions supported by PicoV32 can be added by turning on Verilog *'define* flags in PicoRV32's Verilog description file (picorv32.v). When used, the multiply core provided in the repository in connected to PicoRV32 through the PCPI interface. One must remember to active the PCPI interface enable flag for the multiply core to work.

The verification environment described in Section 3.1 is used to test the new SoC components. After being sure the peripherals are correctly connected and working, the SoC development may advance to the next stage.

3.2.3 Application development

Next, the software application to be ran in the SoC's CPU is developed using an IDE. Since no IDE integration is provided, the IDE can be chosen by the company or developer. The application can be ran in the verilated model of the SoC (i.e., the equivalent C++/SystemC model obtained with Verilator), generating cycle-accurate results, and in the FPGA emulated SoC, which is faster and more accurate. Debugging the application in both targets using GDB is also possible. These features are already detailed in Section 3.1.

The verification environment is once again used in this development stage in order to test the software application. After assuring the software implements the desired functionality, development advances to the final stage.

[3]The list of packages installed can be seen in the shell commands provided in [7], in the *Building a pure RV32I Toolchain* topic.

3.2.4 Optimization

The final development stage consists on optimizing the application developed in the previous stage. In order optimize software applications, profilers are used. A profiler is a dynamic program analysis tool that measures, for example, the frequency and duration of each function call in a program, the usage of instructions (i.e., the total number of times an instruction is used) and the memory space and time complexities of a program. This allow the software developer to understand which aspects or code blocks are affecting performance the most, identifying more easily the software components that should be priorly optimized.

The RISC-V toolchain does not have a profiler available. A way to work around this issue in software development using C or C++ is to use timing libraries such as time.h or other more appropriate ones to a targeted FPGA board to measure the time duration of each function call. Furthermore, for easing the use of this feature, one can use the pre-build compiler directives to turn the time measure feature on and off by simply uncommenting or commenting a compiler flag definition. An example is provided in the following piece of C code:

```
#define COUNT_TIME // comment/uncomment this line to shut off/on time measure feature

// more code

#ifdef COUNT_TIME
start_time_count_routine();  // assume time duration value's mem. space is selected here
#endif
some_function_call();
#ifdef COUNT_TIME
finish_time_count_routine(); // assume time duration value is stored here
#endif

// more code
```

Chapter 4

Results

4.1 Toolchain installation

The toolchain installation was simple because the PicoRV32' GitHub repository [7] provides a Makefile script for that effect. Different make commands are available so that it is possible to install just the RV32I, RV32IC, RV32IM and RV32IMC ISAs. Some adjustments to the Linux shell commands were made in order for them to work in CentOS 7 (the operating system running in the host machine), such as using *yum* instead of *apt-get*, which is used in some Linux distributions such as Debian and Ubuntu but not in CentOS, which uses *yum* for software package management[1].

The provided testbenches were ran using simple make test commands (check the README file in [7] for details) and the results were correct. As such, the RV32IMC toolchain was successfully installed in the host computer running the CentOS 7 distribution of Linux.

4.2 Synthesis of a simple PicoRV32 CPU + memory system

The simple CPU + memory system in folder scripts/quartus of the PicoRV32's GitHub repository was successfully synthesized in an Intel's Cyclone V GT FPGA device. The clock period used was 10 ns, which means the clock frequency was 100 MHz. The README file states that the maximum clock frequency is between 250 MHz and 400 MHz on 7-Series Xilinx FPGAs.

The PicoRV32 core used was a pure RV32I core. The FPGA's resource usage summary stated that this PicoRV32 core used:

- 1103 combinational ALUTs;
- 648 dedicated logic registers;
- 2048 block memory bits.

[1] CentOS uses *yum* because it is derived from the Red Hat Enterprise Linux distribution.

The README file in the PicoRV32' GitHub repository states that a regular-sized[2] PicoRV32 core (which is the type of core used in the system synthesized) for a Xilinx 7-series Kintex device uses:

- 917 slice LUTs;
- 583 slice registers;
- 48 LUTs as memory.

Although the comparison between Intel and Xilinx resources is not straightforward, these results show that the number of LUTs and registers is close enough, which validates the information provided in the repository. PicoRV32 is, indeed, a small RISC-V CPU core that can be used in a SoC.

[2]See the bottom of the README file in [7] for the definition of small, regular and large-sized PicoRV32 core.

Chapter 5

Conclusions

This document presents a research made to understand the state-of-the-art methods for developing SoCs using open source hardware and software components.

First, a revision of the evolution of RISC ISAs and processors was made, comparing it to its CISC counterpart. The advantages of using open source RISC processors to build SoCs were highlighted and carefully explained. Then followed a description of the RISC-V ISA and some of its main advantages for SoC development. Next, a brief survey of SoC trends and features was made and the most pertinent ones were detailed.

With this, four RISC-V processor cores were identified and their features were discussed in order to select which one would be chosen to start building the IObSoC System On Chip. In the end, the size-optimized PicoRV32 core was selected because it seems to provide the most adequate features to fulfill the goals in mind for this dissertation project. Afterwards, a description of the top-level design intended for the IObSoC System on Chip was made.

The state-of-the-art methods for SoC verification using open source toolchains are described, as well as the usual development flow of the entire SoC development process and how the toolchain is used throughout it. The development environment to be made in this dissertation project is then introduced, emphasizing the use of an Ethernet interface with the roles usually played by a JTAG tap in this kind of systems: loading programs and sending bitstreams from the host computer to the SoC (which may reside in an FPGA), and remote debugging of programs running on these.

Some preliminary results regarding PicoRV32's deployment on FPGA are shown and commented.

5.1 Achievements

The research made so far has showed that the development of RISC-based systems is more alive than ever, and it will continue to grow in the upcoming IoT era. The RISC-V open ISA is a promising initiative; for example, startup SiFive Inc. has already released the world's first RISC-V based 64-bit quad-core CPU to support fully featured operating systems like Linux.

The main features of SoCs for IoT purposes have been identified and an early approach for the top-

level design of the IObSoC System on Chip has been defined. A preliminary simplistic diagram for the Test program and software debug has been produced, although some details have yet to be reviewed and better analyzed. The previous iterations of the project at hand proved to be useful for understanding how a robust SoC development environment can be built using open source components, and will serve as an inspiration for the work to be carried out during the dissertation project.

Some preliminary experiments have been made to assess the resources needed for FPGA implementation of the PicoRV32 core. The results show that the PicoRV32 core is indeed small and that this CPU architecture is a good contender for the development of the IObSoC System on Chip.

5.2 Future Work

5.2.1 Dissertation Project

During the upcoming dissertation project, one of the first main focuses will be to understand how the PicoRV32 architecture is built and how it implements the RISC-V ISA. It is necessary to understand how peripheral components and interfaces such as the UART, SPI and Ethernet interface and the memory controller can be connected to the CPU's pins such that all components in the IObSoC System On Chip commuicate well with each other.

Another important aspect that still needs some further study and research is the GNU RISC-V toolchain, because the are many details regarding it that are not still well understood, such as how can Newlib and the GCC and clang/LLVM cross-compilers be used to make the compilation aware of new hardware units, their clock frequencies, etc. Also, it will be necessary to study better how GBD's commands work in order to write the debug software to use with the new Ethernet communication infrastructure.

The makefiles for implementing the development environment will need to be written as IObSoC is built, so to follow its progress. Fortunately, the makefiles provided in PicoRV32's GitHub repository for this matter seem to have been well structured and may prove useful for the development environment.

The planning of the tasks for the dissertation project is presented in Table 5.1.

5.2.2 Future perspectives

In case the IObSoC System on Chip is successfully built during the dissertation project, one of the next future projects will be to integrate in it one or several Versat [8] CGRA cores. This opens new reconfiguration capabilities for SoCs and development possibilities for IoT applications. CGRAs offer hardware reconfigurability like FPGAs but are less power hungry, because the infrastructure that connects the hardware blocks that make up the reconfigurable array in a CGRA is much smaller than in an FPGA. This is a consequence of CGRA's operation being done a the word level, while the FPGA's operation is made at the bit level.

Another interesting future work perspective is to convert the IObSoC System on Chip on a Network on Chip (NoC), i.e., replace the bus communication infrastructure and interfaces in the SoC by a network

Table 5.1: Planning of the dissertation project.

Work Planning	**Scheduling**
Study PicoRV32 RISC-V documentation and PicoRV32's architecture and examples	18/02 - 25/02
Build the IObSoC System on Chip writing Verilog code	26/02 - 26/03
Study PicoRV32's Makefiles and toolchain documentation	27/03 - 04/04
Write Makefiles to implement the verification environment	05/04 - 12/04
Create testbench to verify IObSoC and its components	13/04 - 20/04
Develop debug program	21/04 - 18/05
Verify IObSoC with RTL simulations	19/05 - 26/05
Verify IObSoC with FPGA emulation	26/05 - 07/06
Create/use existing software application and debug it	08/06 - 15/05
Write the dissertation report	16/06 - 01/07

infrastructure based on Internet communication protocols such as TCP and IP and routing algorithms. Components will communicate with each other through routers attached to them inside the chip. It has already been demonstrated in academia (namely, in [15]) that NoC approaches confer great communication upgrades to SoCs with conventional bus-based infrastructures in nearly all relevant parameters such as static and dynamic power, area, throughput, latency and maximum frequency, which validates this future work perspective.

Bibliography

[1] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[2] J. Sousa, C. Rodrigues, N. Barreiro, and J. Fernandes. Building reconfigurable systems using open source components. 04 2014. doi: 10.13140/2.1.3133.2483.

[3] L. Fiolhais and J. Sousa. Warpbird: an untethered system on chip using risc-v cores and the rocket chip infrastructure. 01 2018.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. 06 2012. doi: 10.1145/2228360.2228584.

[5] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[6] Introduction to verilator, 2018. URL `https://www.veripool.org/wiki/verilator`.

[7] C. Wolf and et. al. Picorv32, 2018. URL `https://github.com/cliffordwolf/picorv32`. GIT code repository.

[8] J. Lopes and J. Sousa. Versat, a minimal coarse-grain reconfigurable array. pages 174–187, 07 2017. ISBN 978-3-319-61981-1. doi: 10.1007/978-3-319-61982-8_17.

[9] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013. doi: 10.1109/HPCA.2013.6522302.

[10] P. P. R. Lakhe. A technology in most recent processor is complex reduced instruction set computers (crisc): A survey. *International Journal of Innovative Research & Studies*, Volume 2(Issue

6), 2013. URL https://web.archive.org/web/20150714180129/http://www.ijirs.com/vol2_issue-6/59.pdf. ISSN 2319-9725.

[11] The rise of risc - [opinion]. *IEEE Spectrum*, 55(8):18–18, Aug 2018. ISSN 0018-9235. doi: 10.1109/MSPEC.2018.8423577.

[12] E. A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.

[13] A. Waterman. Improving energy efficiency and reducing code size with risc-v compressed. Master's thesis, EECS Department, University of California, Berkeley, May 2011. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-63.html.

[14] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.10. Technical report, May 2017. URL https://riscv.org/specifications/privileged-isa.

[15] R. Kamal and N. Yadav. Noc and bus architecture: A comparison. *International Journal of Engineering Science and Technology*, 4, 04 2012.

[16] E. Matthews and L. Shannon. Taiga: a configurable risc-v soft-processor framework for heterogeneous computing systems research. 2017.

[17] PULP-Platform. Pulpino, 2018. URL https://github.com/pulp-platform/pulpino. GIT code repository.

[18] PULP-Platform. Ri5cy, 2018. URL https://github.com/pulp-platform/riscv. GIT code repository.

[19] PULP-Platform. zero-riscy, 2018. URL https://github.com/pulp-platform/zero-riscy. GIT code repository.

[20] D. Markovic, C. C. Wang, L. Alarcon, T.-T. Liu, and J. Rabaey. Ultralow-power design in near-threshold region. *Proceedings of the IEEE*, 98:237 – 252, 03 2010. doi: 10.1109/JPROC.2009.2035453.

[21] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.9. Technical Report UCB/EECS-2016-129, EECS Department, University of California, Berkeley, Jul 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html.

[22] OpenOCD — Open On-Chip Debugger, 2013. URL http://openocd.sourceforge.net.

[23] Risc-v toolchain, 2018. URL https://github.com/riscv/riscv-tools. GIT code repository.

[24] Risc-v gnu toolchain, 2018. URL https://github.com/riscv/riscv-gnu-toolchain. GIT code repository.

[25] Icarus verilog, 2018. URL http://iverilog.icarus.com/.