TÉCNICO LISBOA

Linux capable RISC-V CPU for IOb-SoC

Pedro Nuno de Melo Antunes

Thesis to obtain the Master of Science Degree 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. Horácio Cláudio De Campos Neto

October 12, 2022

Abstract

Blablabla

Honor Declaration

“Declaro por minha honra de que ”

Local, Date Signature

Contents

List of Figures

List of Tables

List of Code

List of Acronyms

ACK Acknowledgement.

ALU arithmetic logic unit.

AMO Atomic Memory Operations.

ASCII American Standard Code for Information Interchange.

ASIC Application-Specific Integrated Circuit.

Chisel Constructing Hardware in a Scala Embedded Language.

CLINT Core-local Interrupt Controller.

CPU Central Processing Unit.

CSR Control and status register.

DC1 Device Control 1.

DDR Double Data Rate.

DEMUX desmultiplexer.

DRAM Dynamic Random Access Memory.

DTB Device Tree Blob.

DTC Device Tree Compiler.

DTS Device Tree Source.

ENQ Enquiry.

EOT End of Transmission.

FPGA Field-programmable gate array.

FRX Send a file request.

FTX Receive a file request.

HDL Hardware Description Language.

ISA Instruction set architecture.

LR Load-Reserved.

lsb Less Significant bit.

M Machine.

MMU Memory Management Unit.

msb Most Significant bit.

MUX multiplexer.

OoO Out-of-Order.

OS Operating System.

PC Program counter.

PLIC Platform-Level Interrupt Controller.

rootfs root file system.

rtc arithmetic logic unit.

RTL register-transfer level.

S Supervisor.

SC Store-Conditional.

SoC System on a chip.

U User.

UART Universal asynchronous receiver/transmitter.

UUT Unit Under Test.

1 Introduction

1.1 Motivation

1.2 Objectives

2 Must Have Concepts

During this chapter, it is going to be discuss topics that help understand the technology developments made along the thesis project. The developments will involve both hardware and software components. As such there are hardware and software concepts that are important to have before discussing the next chapters.

In this project a SoC is going to be developed. But it is not going to be created from scratch. The *IOb-SoC* is going to be used as a starting point. Consequently it is important to understand how it works beforehand. It is also important to study the *RISC-V* Instruction set architecture (ISA). Since the hardware developed in this project will be compatible with the *RISC-V* ISA. Additionally, the software and firmware used will be compiled with the *RISC-V* cross-compiler toolchain. An important part when developing a system is its testing and simulation before implementation. Therefor, an overview of available methods for simulation of the developed components is going to be seen. Finally, the last concept that is important to this project is the boot flow of an Operating System (OS) on a *RISC-V* platform.

2.1 The *IOb-SoC* template

The *IOb-SoC* [10] is a System-on-Chip (SoC) template that eases the creation of a new SoC. The *IOb-SoC* provides a base Verilog hardware design equipped with an open-source *RISC-V* processor, an internal SRAM memory subsystem, a UART, and an optional interface to external memory. If the external memory interface is selected, an instruction L1 cache, a data L1 cache and a shared L2 cache are added. The L2 cache communicates with a third party memory controller IP (typically a DDR controller) using an AXI4 master bus. Users can add IP cores and software to build their own SoCs quickly. This way, hardware accelerators can be easily created and tested with the developed firmware.

In figure 2.1 it is represented a sketch of the SoC design. This design is valid at the start of this project. During the hardware developed chapter 4 some alterations were made to the *IOb-SoC* original template.

Building a new processor-based system from scratch can be difficult. *IOb-SoC* has been created to facilitate this process. In this work, a variant of the existing *IOb-SoC* capable of running a Linux Operating System is developed. *IOb-SoC* currently supports three FPGA board models: the Xilinx Kintex UltraScale KU040 Development Board, the Basys 3 Artix-7 FPGA Trainer Board and the Cyclone V GT FPGA Development Kit.

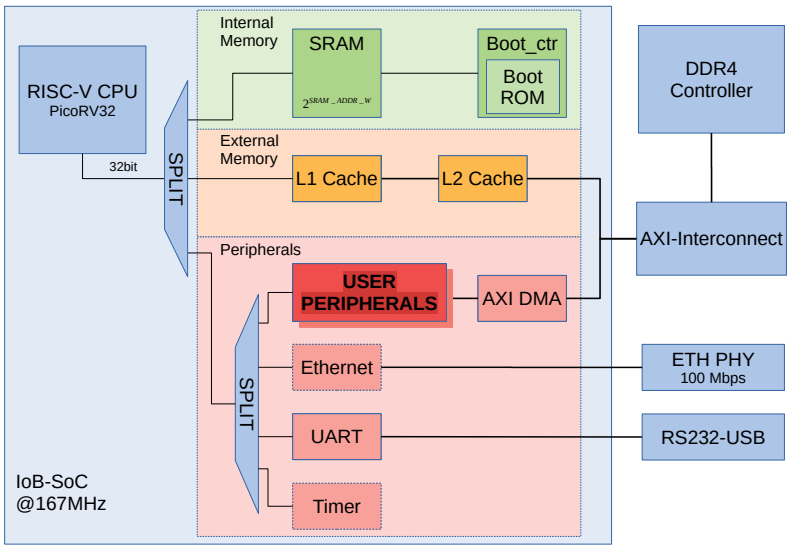

Figure 2.1: *IOb-SoC* sketch.

2.1.1 Adding peripherals

2.1.2 Internal Buses

Review the “cpu_i_resp”, “cpu_d_resp”, “cpu_i_req” and “cpu_d_req” signals.

2.1.3 *iob-split* and *iob-merge*

The ***iob-split*** is simply a configurable desmultiplexer (DEMUX). Meaning that when the *iob-split* hardware module is instantiated the developer can configure it. The developer is able to change the size of the desmultiplexer and the selection bits, through N_SLAVES and P_SLAVES respectively. N_SLAVES corresponds to the number of slaves, witch can also be seen as the number of the DEMUX outputs. P_SLAVES indicates the slave select word Most Significant bit (msb) position, in other words it is the position of the msb of the desmultiplexer selection bits. The number of the selection bits is given by equation 2.1.

$$Nb = log_2(N\_SLAVES) + (log_2(N\_SLAVES) == 0) \tag{2.1}$$

The ***iob-merge*** works similar to the *iob-split* but instead of being a DEMUX it is a configurable multiplexer (MUX). Meaning that instead of having multiple outputs and one input it has multiple inputs and one output. The number of inputs is indicated by N_SLAVES and the selection bits are chosen by P_SLAVES.

2.1.4 Bootloader and bare-metal firmware

2.1.5 Makefile scripts

Makefiles are important because they allow to automatize processes. This way instead of having to execute multiples lines to achieve a goal the user can run a single command. That command will execute a Makefile script that runs the multiple processes needed to achieve a specific goal without specifying them one by one. For example to run a simulation a developer using this projects SoC would only have to write in the terminal the line in listing 2.1.

```
1    make sim-clean
2    make sim-run
```

Listing 2.1: Run a simulation.

the lines in listing 2.1 would first clean all the files related to a previous simulation execution. Then it will run a new simulation. This simulation will use the default configurations in the "config.mk" file. And will execute the run target on the default simulator directory Makefile. The simulator software will be compiled and then executed.

2.2 *RISC-V*

Talk about the 32 registers in the register file

The instruction each ISA extension contains...

Control and status register (CSR) needed to run a full feature OS... (core_id, misa, mcause, ...)

2.2.1 CLINT Specification

The *RISC-V* **CLINT** is described ... Platform must support an ACLINT MTIME counter resolution of 100ns or less (corresponding to a clock tick frequency of at least 10 MHz).

2.2.2 PLIC Specification

The *RISC-V* **PLIC** was first described in the privilege instructions documentation, but since version 1.10 it was moved to its own document.

2.2.3 UART/Serial Console

In the *RISC-V* Platform Specification [18] it is defined that every embedded Operating System (OS) is required to have a UART port implementation that is register-compatible with the industry standard *UART 16550*, which was studied in chapter **??**. The *UART 16550* already exists for a long time, it was released by *National Semiconductor* in 1987, and is present and supported by a large number of software and hardware. The *UART 16550* is often used connected to an RS-232 interface and in this project the development boards used are connected through RS-232 to the computer.

The *UART 16550* registers are...

2.3 The Linux Boot Flow on *RISC-V*

2.3.1 Bootloader firmware

OpenSBI

2.3.2 What is a device tree?

2.4 Open Source Verification tools

Verification tools are ....

https://www.sifive.com/blog/risc-v-qemu-part-1-privileged-isa-hifive1-virtio

2.4.1 Hardware logic simulators

For testing purposes, it is important to have a good hardware simulation environment. For that, we take advantage of already existing and well-developed tools. There exist a number of simulation tools, most of them are proprietary, as for example *xcelium* from *Candence*. Its utilization can increase the cost of a project significantly. In this Thesis we will make an effort of using open-source, free to use, verification tools. In specific, we will take advantage of *Icarus Verilog* and *Verilator*. Although both tools are for verification, they serve different purposes, due to their characteristics.

- ***Icarus Verilog*** is a Logic Simulator that uses verilog or system-verilog testbenchs to test the UUT (Unit Under Test). Unfortunately, its support for system-verilog is limited and some designs might not run in this simulator. *Icarus Verilog* is also known as *IVerilog*.

  After compiling the hardware design, *IVerilog* outputs a file which can be run line by line to simulate designed logic.

- ***Verilator*** transforms the *Verilog* HDL designs into a *C++* program that can be executed after being compiled. Using *C++* to create a testbench allows to execute the converted hardware program as a normal program. This way simulating the hardware initially described in *Verilog*. While also allowing to easily make use of system calls. The testbench needed to run with Verilator is similar to the testbench in *Verilog* used with *Icarus*.

The biggest differences are: *Verilator* only represents logic signal as 1's or 0's, contrary to *IVerilog* which also represents unknown values as X's; Since *Verilator* ends up being a C++ program it is much faster to run the simulation than with *IVerilog*; On another perspective *Verilator* is slower than *IVerilog* to interpret the hardware logic design. As such, it is easier to use *IVerilog* to detect errors on the design, but it is better to use *Verilator* for more complexed simulations.

2.4.2 Qemu Simulation

QEMU is an open-source machine emulator and virtualizer, allowing running software and firmware, like operating systems and Linux distributions on many different devices and architecture using a personal computer. It is similar to VMware or VirtualBox. QEMU is a functional emulator; it translates the instructions that were supposed to run on the target architecture to instructions that run on the host CPU. Other types of emulators are Trace-accurate, like Spike, and Cycle-accurate, for example, Verilator. The advantage of using a functional emulator like QEMU is that it is way faster than the other emulation types. A functional emulator runs 100 million to > 1 billion instructions per second, while trace-accurate or cycle-accurate run 10 to 100 thousand instructions per second.

Emulating Linux in *RISC-V* with QEMU is helpful to create and test software for *RISC-V* platforms. QEMU can be used to emulate both 32-bit and 64-bit *RISC-V* CPUs. During this work, an implementation of the Linux kernel will be tested on QEMU running on a *RISC-V* board. The platform that is going to be used is the *virt* board. Virt is a board that does not contain any real hardware but is designed to be used in virtual machines. To define which board the emulation is supposed to run in when calling QEMU, the following argument should be passed: *–machine virt*.

3 Existing Embedded Technologies

There already exists embedded microcontrollers capable of running Linux. Big companies for example ARM, Qualcomm, MediaTek, Intel and AMD have created microcontrollers capable of running Linux. But the processor architecture of those microcontrollers is not open-source, much less the microcontroller itself.

As an example, the *Raspberry Pi 4* is a very capable and cheap board where a developer can test and implement new software running in Linux. The Raspberry CPU is an *Cortex-A72* [1] witch is a System on Chip (SoC) developed by ARM on their ARMv8 64-bit CPU architecture. But if someone wanted to use the Raspberry as a base for his costume hardware design, that would be impossible. And thus appears the need for open-source hardware that allows creating something new without having to start from scratch every time. This led to the appearance of *RISC-V* the open-source CPU architecture.

3.1 Closed source *RISC-V* Embedded Systems

Since then, a few companies using *RISC-V* have appeared. *RISC-V* CPUs are already present in the automotive and IoT markets, besides AI chips in data centres. Due to the *RISC-V* ISA royalty-free license, new StartUps tend to look at *RISC-V* CPUs as a solution for their cores. Even if the CPU Core isn't free to use it ends up being a cheaper solution.

While creating new products companies proved how advantageous the *RISC-V* architecture was. Furthermore, they have contributed to open-source software, hardware and documentation. Some companies with big recognition involved with *RISC-V* technology are:

- *Western Digital* who now uses *RISC-V* in its external storage disks.
- *Microchip* as launched the first *RISC-V*-Based System-on-Chip (SoC) FPGA, *PolarFire*.
- *Antmicro/Microsemi* [1] have built a software called Renode that is used to develop, debug and test multi-node *RISC-V* device systems.
- *BeagleBoard.org*, *Seeed Studio* and *StarFive* worked together to build the first affordable *RISC-V* computer designed to run Linux, *BeagleV* [5]. The board is priced around 150€.

These companies have all helped pave the way for a full-feature Operating System based on the Linux kernel to be compatible with the *RISC-V* architecture. However, two companies have a bigger impact on *RISC-V* CPU design, those are Andes Technology and SiFive.

[1] Microchip has acquired Microsemi Corporation in May 2018.

3.1.1 Andes Technology

Andes Technology is one of the founding members of *RISC-V* International. Since it is highly involved with *RISC-V* it ended up being one of the major contributors (and maintainers) of the *RISC-V* toolchain. This is important because the *RISC-V* ISA is merely an instruction set architecture, there needs to exist complementing software, such as compiler and development tools.

Nowadays Andes CPUs are applied nearly everywhere, from telecommunications, storage controllers, and touch screen sensors to data centres, etc. Andes Technologies has had incredible success using *RISC-V* technology, as proof they have shipped billions of embedded SoC with *RISC-V* processors based on their *RISC-V* ISA variant, AndeStar™ V5.

Andes CPUs which are capable of running Linux are the *A25* [29] and *AX25* [30]. Both support single and double precision floating points, the *RISC-V* P-extension (draft) DSP/SIMD ISA and an MMU (Memory Management Unit) for Linux applications. Besides that both enable the use of Machine (M), User (U) and Supervisor (S) Privilege levels that allow running Linux and other advanced operating systems with protection between kernel and user programs. Furthermore, both have L1 instructions and data cache. The difference between them is that *A25* is based on 32-bit architecture and the *AX25* is 64-bit. This leads to the *AX25* being ideal for embedded applications that need to access address space over 4GB, and the *A25* being smaller in gate count. Both CPUs can be implemented on the AE350 [28] SoC allowing to use these CPUs on developer boards, for example in the *ADP-XC7K160/410* [27].

3.1.2 SiFive

SiFive is a company that was born from the *RISC-V* ISA. SiFive was founded by three researchers from the University of California Berkeley, Krste Asanović, Yunsup Lee, and Andrew Waterman. Those researchers were deeply involved with the development of the *RISC-V* ISA, from working on the base ISA to working on the floating point numbers and compressed instructions ISA extensions. It is no surprise that the first company to release a chip and development board that implemented the *RISC-V* ISA was SiFive. This happened in 2016 one year after the company was founded.

In 2017 SiFive launched *U54* [22] which was the first *RISC-V* CPU capable of running a full fledge Operating System like Linux. With it they launched the *U54-MC* [23] SoC that had four *U54* 64-bit cores. Furthermore, the *U54-MC* implemented the initial CLINT and PLIC unit. The development of the CLINT and the PLIC made by SiFive would eventually lead to the documentation and specification of the respective hardware components with which *RISC-V* systems must be compliant if they proclaim to use either one. One year after, in 2018, they launched *HiFive Unleashed* [20] which was the first board that implemented the *U54* CPU and

run a Linux OS with a desktop environment (DE). The *HiFive Unleashed* has been discontinued and better hardware has been made available.

SiFive has since then extended its *U Cores* product lineup. All *U* cores are 64-bit application processors capable of running Linux. The highest performance core is the *U74* [24]. The core architecture is RV64GBC which means it supports the *RISC-V* I, M, A, F, D, B and C ISA extensions (explained in ***ref section 2.3.x***). This CPU has already been applied to multiple boards, for example, the *BeagleV* has a SoC with dual-core SiFive *U74* CPU. SiFive as also launched its own development board, *HiFive Unmatched* [21], with four *U74* cores on the *U74-MC* [25] SoC. Furthermore, in 2021, Canonical the developers behind Ubuntu announced the OS support for both the HiFive Unmatched and HiFive Unleashed.

3.2 Open-Source Solutions

Built upon the *RISC-V* open-source Instruction set architecture, various CPU designs have emerged. Some of them are fully open-source and might be implemented in other projects. Those CPUs were mostly developed by Universities research groups or by individuals with a grant.

RISC-V CPUs are most popular in embedded systems and IoT devices. Consequently there exists a wide variety of open-source CPUs which are implemented on multiple embedded microcontrollers. A few examples of those CPUs would be the *PicoRV32* [32], *NEORV32* [26], *DarkRISCV* [7] and *Ibex* [12] from lowRISC. But those will not be discussed in detail in this paper since they do not meet the requirements to run the Linux Kernel. This CPUs either only support Machine (M) level privilege mode or support Machine (M)+Supervisor (S) mode. Moreover, none of the given examples supports the Atomic *RISC-V* ISA extension. This extension is essential to run Linux. Since the kernel explicitly executes instructions from the Atomic extension.

To run a Linux-based Operating System an application processor is needed. A CPU is considered an *application processor* if it has the hardware required to run a full-feature Operating System (OS) and user applications. This means that the processor should have the required Control and status register (CSR), support M+S+U privilege modes and support atomic instructions. An open-source solution would be either the *CVA6* [33] (previously known as Ariane), *BOOM* [34] or *VexRiscv* [16].

3.2.1 *CVA6*

The CVA6 is a 6-stage, single issue, in-order CPU which can execute either the 32-bit or 64-bit *RISC-V* instruction set. CVA6 has support for the I, M, A and C *RISC-V* ISA extensions. The original design was initiated in a research group by a PhD student at ETH Zurich (where

they called the core Ariane). Since then the development and maintenance of CVA6 were incorporated in the *OpenHW Group* as part of their CORE-V processor lineup. The support for RV32IMAC was only developed recently by Thales and is also open-source. The CPU design is illustrated in figure 3.1 that was obtained from: `https://github.com/openhwgroup/cva6/`.

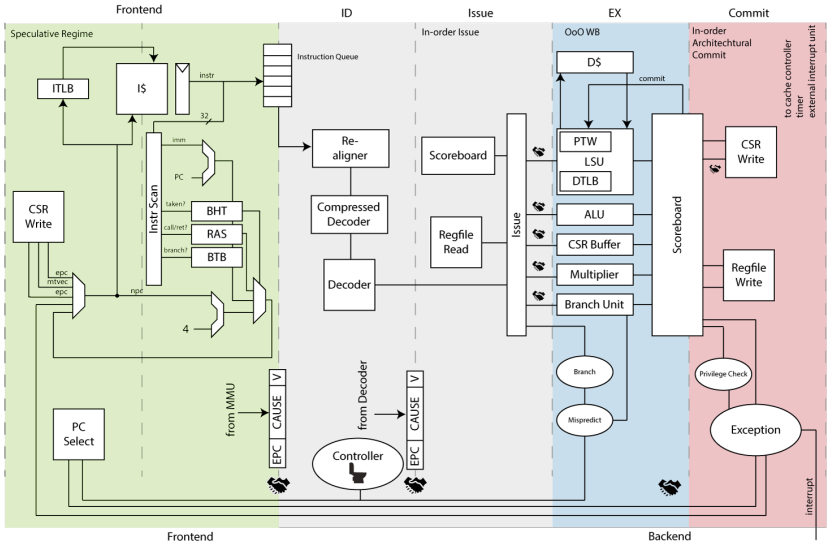

Figure 3.1: CVA6 core design architecture.

The CVA6 supports any operating system based on Unix since it implements the three needed privilege levels M, S and U. The core is written in SystemVerilog, and its micro-architecture is designed to reduce the critical path length. Since it is written in SystemVerilog, it is easier for someone knowledgeable in the classic Verilog and VHDL languages to understand and create a customized CPU based on the CVA6, than if it was written in a high-level hardware description language. However, although the CVA6 is an open-source project it is hard to take advantage of isolated hardware components. This is a consequence of how it was developed. Every SystemVerilog module of the CVA6 depends on other files from the project and the CPU itself is very little customizable. To illustrate the problem if we wanted to remove the L1 cache present in the CVA6 to use the L1 cache used on IOb-SoC, it would be very difficult and time-consuming to create a CPU core without that component.

The CVA6 can be found implemented on *OpenPiton* [4]. *OpenPiton* is an open-source project developed by the Princeton Parallel Group. With it, one can easily create an SoC that has multiple CV6 cores and run a full-feature Operating System (OS) on a development board with an FPGA.

3.2.2 The Berkeley Out-of-Order *RISC-V* Processor

The Berkeley Out-of-Order *RISC-V* Processor (*BOOM* [34]) is a superscalar Out-of-Order (OoO) processor executing the RV64GC variant of the *RISC-V* ISA. BOOM was created at

the University of California, Berkeley in the Berkeley Architecture Research group. The CPU design is optimized to run on ASICs, although it can also be implemented on FPGAs. Its priority is to be a high-performance, synthesizable, and parametrizable core for architecture research. The current release, named *SonicBOOM*, has one of the best performances from the publicly available open-source *RISC-V* cores.

BOOM is a 10-stage CPU with the following stages: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback, and Commit. However, in most practical implementations, many of those stages are merged, generating seven stages altogether: Fetch, Decode/Rename, Rename/Dispatch, Issue/Register Read, Execute, Memory and Writeback. Since committing happens asynchronously, it is not counted as part of the "pipeline". The load-store unit is optimized for the superscalar out-of-order architecture, and the data cache is organized into two dual-ported banks. At the front end, it is possible to customize the size of the L1 Instruction cache, the TLB, and the decode stage. Similarly to the CVA6, it is difficult or impossible to remove the cache from the core design and use the IOb-Cache instead.

This CPU design is written in Chisel [3] Hardware Description Language (HDL). The Constructing Hardware in a Scala Embedded Language (Chisel) allows for the production of synthesizable Verilog designs while using a high-level language to describe the hardware. Chisel is an adaptation of Scala [14] programing language, adding hardware construction primitives.

To build a System on a chip (SoC) with BOOM we would have to utilize the *Rocket Chip* [2] SoC generator from CHIPS Alliance. Since BOOM uses micro-architecture structures (TLBs, PTWs, etc) from that tool.

3.2.3 *VexRiscv*

The *VexRiscv* [16] CPU is a 32-bit Linux Capable *RISC-V* CPU written in the *SpinalHDL* [15]. The hardware description of this CPU is accomplished by utilizing a software-oriented approach. Similarly to Chisel, *SpinalHDL* is based on the Scala programing language.

VexRiscv is an in-order CPU with five "pipeline" stages. Many CPU plugins are optional, which add many functionalities to build a custom *RISC-V* CPU. The architecture design approach in this processor is unconventional, but it has its benefits: there are remarkably few fixed hardware components; Parts of the CPU can be swapped, turned on and turned off via the plugin system; without modifying any of the CPU sources, it is possible to add new functionalities/instructions easily; It permits the CPU arrangement to cover a significantly large spectrum of implementations, allowing the construction of an entirely parametrized CPU design. When the CPU is configured without plugins, it only includes the description of the five "pipeline" stages and their basic functionalities and nothing else. Everything else needs to be added to the CPU via plugins, including the program counter. VexRiscv can either be an application processor capable of

running a full-feature Operating System (OS) or a super simple microprocessor ideal for bare-bone applications depending on the way it is configured. Contrary to *BOOM*, *VexRiscv* does not need any external library. This makes it very easy to generate the synthesizable Verilog file from a *SpinalHDL* design.

There exists an open-source project that runs Linux with *VexRiscv*, *linux-on-litex-vexriscv* [11]. *LiteX* is used to create a System on a chip (SoC) around the *VexRiscv* core. *LiteX* SoC design and peripherals are written in *Migen* [6] another high level HDL. *Migen* unlike *SpinalHDL* and Chisel is based on Python 3.5. On account of the language describing its hardware and the way, the *linux-on-litex-vexriscv* project is structured it is very hard to understand how the system works, where the generated RTL is and how to add custom hardware. Furthermore, *linux-on-litex-vexriscv* uses FPGA specific hardware, making it impossible to port the system to ASIC.

Recently the developer behind *SpinalHDL* has also made public the *NaxRiscv* CPU. *NaxRiscv* is a CPU designed specifically to run a full-feature Operating System, like Linux. And just like *VexRiscv*, *NaxRiscv* uses *SpinalHDL* to describe its hardware. Although *NaxRiscv* seems like a very promising CPU it is still on its early stages. Consequently, it has a primitive interface which makes it complicated to implement on costume System on a chip (SoC).

3.3 Overall CPU comparison

In table 3.1 we can see a comparison of the CPUs that were presented in the previous sections, capable of running a full-feature Operating System (OS). All of the CPUs on the table are considered application processors. It can be observed that every CPU has a Memory Management Unit (MMU) and they all support U+S+M privilege mode. Furthermore, all of the CPUs hardware design have L1 Instruction Cache and L1 Data cache system integrated. This happens because to support atomic instructions it is easier to have direct access to the L1 Cache.

GNU/Linux is the combinations of *GNU* with the Linux kernel. The GNU Project developed a large part of the software that forms a complete Operating System (OS). Many “Linux” distributions make use of that software, a few examples would be *Debian*, *Ubuntu*, *openSUSE*, *Fedora*, and the list could go on. So a processor that supports the GNU/Linux feature is a CPU that is capable of running a distribution like *Ubuntu* or *Debian*. From the table, we can see that 32-bit *RISC-V* CPUs are the only ones not capable of running a *GNU/Linux* Operating System (OS).

	ARM	Andes Technology		SiFive		PULP platform	UC Berkeley	SpinalHDL	
	Cortex-A72	A25	AX25	U54	U74	CVA6	BOOM	VexRiscv	NaxRiscv
Architecture bit widths	64-bit	32-bit	64-bit	64-bit	64-bit	32/64-bit	64-bit	32-bit	64-bit
MMU	Y	Y	Y	Y	Y	Y	Y	Y	Y
FPU	Y	Y	Y	Y	Y	X	Y	X	Y
16-bit instructions	X	Y	Y	Y	Y	Y	Y	Y	X
Cache L1(I+D)	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interrupt Controller	X	Y	Y	Y	Y	X	X	X	X
U+S+M Mode	N/A	Y	Y	Y	Y	Y	Y	Y	Y
GNU/Linux	Y	X	Y	Y	Y	Y	Y	X	Y
Open-Source	X	X	X	X	X	Y	Y	Y	Y

Table 3.1: CPU comparison table: Y means the CPU supports the feature; X means the CPU does not support the feature; N/A means the feature does not apply to the respective CPU.

4 Hardware Developed

During the development of this thesis, there was both hardware and software developed. In this chapter, we will go through the hardware designed to build an appropriate System on a chip (SoC) capable of running a full-fledged Operating System (OS).

The *IOb-SoC* was used as a System on a chip (SoC) template. *IOb-SoC* has some features that make it ideal for developing this project SoC. Firstly, it is open-source hardware. Open-source means there are no royalties, and the source code is publicly available. Secondly, adding new peripherals is very easy and intuitive, as was previously seen in section 2.1. Thirdly, the *IOb-SoC* implements the interface with an internal (SRAM) and an external (DRAM) memory. When using external memory, the *IOb-SoC* instantiates an *iob-cache* system. Finally, the *IOb-SoC* implements a boot hardware unit that controls the first boot stage (also known as stage zero) executed after powering/resetting the system.

The hardware components that needed to be changed from *IOb-SoC* were the Central Processing Unit (CPU) and the Universal asynchronous receiver/transmitter (UART) peripheral. The CPU had to be changed because the previous CPU (*PicoRV32*) could not run a full-featured Operating System. The UART had to be swapped since no compatible Linux drivers worked with *iob-UART*. Besides changing a few components from the chip, the new hardware had to be added. The new hardware is the CLINT and the PLIC, both compatible with *RISC-V* specifications. The CLINT was added to implement timer and software interrupts on the SoC. The PLIC was added to manage interrupts generated by other peripherals. In this project, the UART is the only additional peripheral that causes an external interrupt. A sketch of the SoC developed can be seen in figure 4.1.

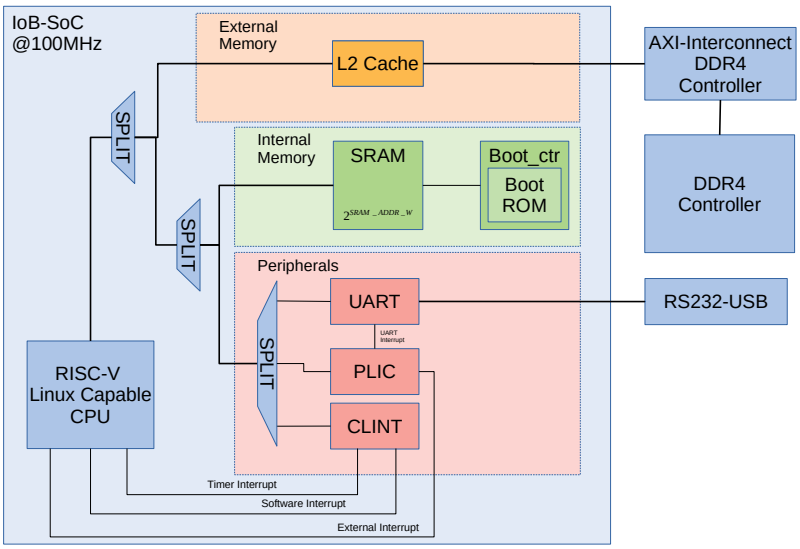

Figure 4.1: Developed SoC sketch.

Comparing figure 4.1 with the original design of *IOb-SoC* (figure 2.1) we can see that there were a few additional modifications. In the first place, it can be seen that the L1 Cache was removed. Since every application processor studied had an L1 cache built in, there was no need for the L1 *iob-cache*. Next, a *iob-split* was added to the *IOb-SoC*. Previously, there was a single *iob-split* for the data bus with three branches (the internal memory, the external memory and the peripheral bus). This meant that there were 2 selection bits, when '00' then the internal memory bus was active, when '01' it was the peripheral bus and when '10' it was the external memory. This caused a problem because when addressing the external memory if its size is bigger than 1GB the selection bits would be '11'. The desmultiplexer (DEMUX) output selected by '11' is not connected anywhere, so this caused an internal hardware error. The solution was to include two *iob-split* modules each with two branches. The first would choose between the external memory and either the internal memory or peripheral bus. The second would choose between the internal memory and the peripheral bus. Another advantage of using this method is that now the selection bit's position does not vary depending on if we are using the DDR or not. This makes it easier to use external software that does not use the *iob-soc* Makefiles. Before the peripheral addressing on external software had to be changed every time the developer wanted to test with or without the external memory.

During this project, there was also an improvement on the *IOb-SoC* verification. This led to the creation of a top hardware module for the developed System on a chip.

4.1 Central Processing Unit

The CPU chosen to use in this project was *VexRiscv* [16]. The performance of the CPU is not a significant issue for this project. However, how the core was designed and developed highly influenced the CPU decision. The flexibility of the CPU design, meaning how easily the CPU can be adapted to take advantage of the other components in *IOb-SoC*, is an essential factor. Since the hardware and software developed in this project are open-source, the CPU implemented had to be open-source hardware. Moreover, knowing that the *IOb-SoC* signals are 32-bit wide, ideally, the selected CPU should support RV32IMAC to facilitate its integration with IOb-SoC. From the CPUs studied in chapter 3 *VexRiscv* looked like the more indicated.

Generating the RTL *verilog* file from the *SpinalHDL* hardware description is very simple. After cloning the *VexRiscv* GitHub repository the developer only has to run one command. As can be seen below in listing 4.1. On the *VexRiscv* repository there exist a couple of demo CPU configurations. The configurations can be directly used or configured to generate a custom *VexRiscv* core. There even already exists a demo configuration to generate a Linux-compatible core. Although in the developed hardware a custom core was implemented, the Linux demo configuration was used as a starting point. Unfortunately, the Linux Demo design is outdated and the instructions, commented on the hardware configuration file, to run a Linux simulation and test the core does not work.

```
git clone https://github.com/SpinalHDL/VexRiscv.git && \
  cd VexRiscv && sbt "runMain vexriscv.demo.LinuxGen"
```

Listing 4.1: Generate *verilog* from *SpinalHDL*

The *VexRiscv* can be configured by adding and removing plugins. Plugins are hardware components described in *SpinalHDL* that can be reused in different designs by simply adding "new Plugin_Name(...)," to the plugins list in the top CPU description file. The existing plugins are described in the *VexRiscv* repository on the "src/main/scala/vexriscv/plugin" directory. Looking at the available plugins it can be seen that there are two different plugins for the instruction bus and data bus. They are "IBusSimplePlugin", "IBusCachedPlugin", "DBusSimplePlugin" and "DBusCachedPlugin". The difference is that the "cached" plugins have the L1 Cache integrated, while the simple plugins do not. An additional difference between the data cached and simple plugin is that, although the "DBusCachedPlugin" fully supports both, the "DBusSimplePlugin" supports only Load-Reserved (LR)/Store-Conditional (SC) but not Atomic Memory Operations (AMO) instructions. The "DBusSimplePlugin" could also be adapted to enable the full "A" extension. To do that the it would be needed to have knowledge in *SpinalHDL*. Learning how to code in *SpinalHDL* would be very time-consuming and not in the scope of this project.

The first step on implementing the *VexRiscv* core on the *IOb-SoC* was making sure that it worked on "bare metal" applications. Meaning it had to be working with the application accessing the silicon chip directly without any intermediary like an Operating System (OS). This was done using the instruction and data "simple" plugins. The next step was to run the Linux kernel. To do so the instruction and data "simple" plugins had to be changed to the "cached" plugins. The missing support for Atomic Memory Operations (AMO) instructions was noticeable because the software would stop executing and enable an unknown instruction signal. It was possible to identify which instruction was causing the problem through the signal waves created during simulation.

The final *VexRiscv* core configuration file contained the needed plugins to run a minimal Operating System (OS) based on Linux. The plugins present were:

- The "IBusCachedPlugin" was added. With it, the address of the first instruction the CPU had to fetch was defined by setting the reset value of the Program counter (PC). Also, it was specified that the CPU had no branch predictor and that it supported compressed instructions. The decision to not use any branch predictor was because there seemed to be a compatibility problem between the most recent *RISC-V* toolchain and the branch predictor that are available in the *VexRiscv*. Since performance was not a concern in this project I choose to not use a branch predictor.
- The "DBusCachedPlugin" was added for the reason that it fully supported the atomic instructions.
- The "DecoderSimplePlugin" is used to decode the instructions.

- The “RegFilePlugin” implements the register file. These are the registers inside the CPU.
- The “IntAluPlugin” is used to calculate arithmetic and logic operations.
- The “SrcPlugin” is an auxiliary plugin for the plugins that contain arithmetic logic unit (ALU), Branch related hardware and Load/Store hardware logic.
- The “FullBarrelShifterPlugin” implements the shift instructions present in the *RISC-V* base Instruction set architecture (ISA).
- The “HazardSimplePlugin” is used by the core to determine where it needs to stall.
- The “MulPlugin” allows the core to execute multiplication instructions.
- The “MulDivIterativePlugin” could be used to add multiplication and division support to the core (*RISC-V* M ISA extension). In this case, it was used to add only division since the multiplication support was added by another plugin.
- The “CsrPlugin” is configured to fully support Linux. This plugins adds the needed Control and status register (CSR) to run a full feature OS.
- The “DebugPlugin” was deactivated in the used core. But it could be used to debug the CPU core if there existed a JTAG interface. Currently the *IOb-SoC* does not support it.
- The “BranchPlugin” allows the core to execute and make decisions on the jump instructions. This is part of the base Instruction set architecture (ISA)
- The “MmuPlugin” added support for the Memory Management Unit (MMU). Which is required to run a full feature OS.
- The “FpuPlugin” can add support for both the floats and doubles instruction extensions. In the core used this plugin was deactivated since to run a minimal OS there is little to no advantage of using this extension. Causing the FPU to only be adding unnecessary hardware logic.

After generating the Verilog file that describes a *VexRiscv* core I had to create a wrapper hardware module that adapted the *VexRiscv* core interface to the *IOb-SoC* internal bus.

4.2 VexRiscv Wrapper

The Verilog wrapper, witch is called *iob_VexRiscv*, is instantiated by the *IOb-SoC* top SoC hardware module as the CPU component and instantiates the *VexRiscv* core Verilog module. The interface between the *IOb-SoC* hardware and the *VexRiscv* core is created by establishing a connection between the inputs and outputs from both sides.

The input signals of *iob_vexriscv* are the clock signal which is the system clock derivative from the development board where the SoC is implemented; the reset signal which is set to high

('1') when the system reboots and when the stage 0 bootloader finishes; the boot signal that has the value '1' while the stage 0 bootloader is executing, after it finishes the boot signal value drops to '0' at the same time the reset signal is set to high; the instruction bus response signal that is connected to "cpu_i_resp"; the data bus response signal that is connected to "cpu_d_resp"; the timer interrupt and software interrupt signals which are set to '1' or '0' by the CLINT unit; the external interrupt signal which is controlled by the PLIC unit. The output signals are the instruction bus request signal and the data bus request signal, which connect to the "cpu_i_req" and "cpu_d_req" respectably. The "cpu_i_resp", "cpu_d_resp", "cpu_i_req" and "cpu_d_req" signals were reviewed in section 2.1.

The input and output signals of the *VexRiscv* core can be seen in table 4.1. It can also be seen the signal's width and their equivalent signal in the *IOb-SoC* top hardware.

After understanding the inputs and outputs of each module it is easy to see which wires should be connected. But after connecting all the wires there were three problems. The first was the "strb" signal needed by *IOb-SoC* when writing data to memory which did not exist in the *VexRiscv* signals. The "strb" signal could be obtained in two different ways. One way would be through the "dbus_req_size" signal, the two less significant bits of the "dbus_req_address" and the "dbus_req_wr" signal. The other way was through the "dBus_cmd_payload_mask" signal and the "dBus_cmd_payload_wr" signal. The "DBusSimplePlugin", contrary to the "DBusCachedPlugin", had no "dBus_cmd_payload_mask" signal that is why the first method was created. Accordingly for the first method, the "mask" signal had to be generated by the hardware logic expressed in equation 4.1.

$$\begin{cases} dbus\_req\_mask\_aux = dbus\_req\_size[1]?4'hF : (dbus\_req\_size[0]?4'h3 : 4'h1) \\ dbus\_req\_mask = dbus\_req\_mask\_aux << dbus\_req\_address[1:0] \end{cases} \tag{4.1}$$

Moreover, the "mask" signal indicated the active bytes when both read or write operations were occurring. On the other hand, the "strb" signal should only be active when a write operation is happening. Both methods logic expressions can be seen in equation 4.2. This implements a MUX where "dbus_req_wr" is the selection bit.

$$\begin{aligned} strb &= dbus\_req\_wr?dbus\_req\_mask : 4'h0 \\ &= dbus\_req\_wr?dBus\_cmd\_payload\_mask : 4'h0 \end{aligned} \tag{4.2}$$

The second is that the *IOb-SoC* internal bus did not contain all the signals that were needed by the *VexRiscv* core. To successfully make the interface handshake with the *VexRiscv* core an instruction and data request "ready" signal had to be generated. The "ready" signal indicated that the SoC was ready to receive and accept a request from the CPU. To solve this problem a register that saved the value of the "cmd_valid", called "valid_reg", was created. This register

Port	Width	Direction	Description	IOb-SoC Port
dBus_cmd_valid	1	output	Indicates that the CPU is ready to make a data request.	cpu_d_req[‘valid(0)]
dBus_cmd_ready	1	input	Indicates that the SoC is ready to receive a data request.	N/A
dBus_cmd_payload_wr	1	output	Indicates that the CPU wants to write data to memory.	N/A
dBus_cmd_payload_uncached	1	output	Indicates if data is on L1 cache	Not used
dBus_cmd_payload_address	32	output	Used to address memory.	cpu_d_req[‘address(0,32)]
dBus_cmd_payload_data	32	output	Used to send data to memory.	cpu_d_req[‘wdata(0)]
dBus_cmd_payload_mask	4	output	Indicates which bytes in a word are accessed.	N/A
dBus_cmd_payload_size	2	output	log_2(number of bytes in the burst)	Not used
dBus_cmd_payload_last	1	output	Indicates when the last byte is transferred.	Not used
dBus_rsp_valid	1	input	Indicates that the SoC is ready to send a response.	cpu_d_resp[‘valid(0)]
dBus_rsp_payload_last	1	input	Indicates when the last byte is transferred.	Not used
dBus_rsp_payload_data	32	input	Receive data from memory.	cpu_d_resp[‘rdata(0)]
dBus_rsp_payload_error	1	input	Indicates existence of an error.	Not used
timerInterrupt	1	input	Indicate a Timer Interrupt.	timerInterrupt
externalInterrupt	1	input	Indicate an External Interrupt.	externalInterrupt
softwareInterrupt	1	input	Indicate a Software Interrupt.	softwareInterrupt
externalInterruptS	1	input	Indicate an External Interrupt at the Supervisor level.	Not used
iBus_cmd_valid	1	output	Indicates that the CPU is ready to make an instruction request.	cpu_i_req[‘valid(0)]
iBus_cmd_ready	1	input	Indicates that the SoC is ready to receive an instruction request.	N/A
iBus_cmd_payload_address	32	output	Used to address memory.	cpu_i_req[‘address(0,32)]
iBus_cmd_payload_size	2	output	log_2(number of bytes in the burst)	Not used
iBus_rsp_valid	1	input	Indicates that the SoC is ready to send a response.	cpu_i_resp[‘valid(0)]
iBus_rsp_payload_data	32	input	Receive an instruction from memory.	cpu_i_resp[‘rdata(0)]
iBus_rsp_payload_error	1	input	Indicates existence of an error.	Not used
clk	1	input	System clock signal.	clk
reset	1	input	CPU reset signal.	cpu_reset

Table 4.1: *VexRiscv* core inputs and outputs.

would be updated when either the “cmd_valid” or the “rsp_valid” signal were active. The “ready” signal should be high (’1’) before accepting a request, and after it should be low (’0’) while the response is not available. The initial approach to the values that the “cmd_ready” signal should assume can be seen in the truth table 4.2. This truth table was obtained by analyzing the simulation signals wave. The “N/A” values in the table mean that those situations never occurred.

The truth table can be transformed in a logic gates expression, which can be seen in equation 4.3.

valid_reg	cmd_valid	rsp_valid	cmd_ready
0	0	0	0
0	0	1	N/A
0	1	0	1
0	1	1	N/A
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.2: First try at identifying the rules the cmd_ready should follow.

$$(valid\_reg \cdot rsp\_valid) + (valid \cdot \overline{valid\_reg} \cdot \overline{rsp\_valid}) \tag{4.3}$$

But this approach had an issue. The logic expression depended on the "cmd_valid" signal which was generated inside the *VexRiscv* core. This could generate a bigger complication since the combinatorial circuit that generates the "cmd_valid" is unknown and might generate an infinite hardware loop. From better analyzing the signal behavior it was noticed that when "valid_reg", "cmd_valid" and "rsp_valid" are low ('0') the value of "cmd_ready" is irrelevant. It could be concluded since when the "valid_reg", "cmd_valid" and "rsp_valid" are low happens the "cmd_ready" signal was not being used by the *IOb-SoC*. The truth table can them be simplified to table 4.3.

valid_reg	rsp_valid	req_ready
0	0	1
0	1	N/A
1	0	0
1	1	1

Table 4.3: Simplified truth table.

Which can be seen as a simple XOR logic gate. The equation 4.4 show the hardware logic expression implement.

$$(valid_reg \cdot rsp\_valid) + (\overline{valid_reg} \cdot \overline{rsp\_valid}) = valid_reg \odot rsp\_valid \tag{4.4}$$

The last problem was that after accepting an instruction or data request the values of the "address", "data" and "mask" signals could change inside the *VexRiscv* core. This changes would pass through *iob_VexRiscv* and reflect in the rest of *IOb-SoC* hardware. Which caused the *iob-cache* and peripherals to not function currently. This problem was solved by creating registers that saved the value of the "address", "data" and "strb" signals when the request

was accepted. The register values would then only change when the response was already received.

To finalize, the CPU should be able to run firmware from both the internal and external memory. When the stage 0 bootloader is running the Most Significant bit (msb) of the instruction fetched address had to be forced to '0'. This would force the CPU to fetch instructions from the boot hardware unit. When defined that the firmware had to run from the external memory (RUN_EXTMEM=1) the first instruction fetched should be at address $0x80000000$. To achieve this requirement the Most Significant bit (msb), when RUN_EXTMEM=1 was defined as the negated value of the boot signal. When RUN_EXTMEM=0 the msb was forced to always be '0' since there is no need to access the external memory. On the data request bus, it should also be taken into account that the msb had to be '0' when the CPU wanted to access the peripherals.

4.3 *UART 16550*

The approach taken in this project was to adapt an existing open-source Universal asynchronous receiver/transmitter (UART) core that is supported by the Linux kernel. The other option was to create a Linux driver compatible with *iob-UART* and compile the kernel with it. The chosen approach seemed more adequate and a simpler solution.

Since the developed chip is supposed to be open-source the UART core should also be open-source hardware. The core used was a *UART16550* [9] that has been made available by *freecores* on *github*. This UART was written in Verilog, although it was an older version of Verilog it is still synthesisable by modern tools and easy to understand. The *UART16550* core used implements a wishbone interface to interact with the System on a chip (SoC). Similarly to what was done with the CPU, I had to create a wrapper to adapt the core to the *IOb-SoC*.

The wishbone interface is established in the top hardware module from the used *UART16550* core. In table 4.4, which can be obtained from the open-source *UART16550* core documentation, it can be seen the wishbone interface signals. The wishbone specification determines that there needs to be a master and a slave. In this case when the CPU sends a request signal for the *UART16550* peripheral the master is the CPU and the slave is the *UART16550*.

The interface between the *IOb-SoC* and the *UART16550* top hardware is established by the verilog module that acts as a *UART16550* wrapper. The wrapper module is called "iob_uart16550". The "iob_uart16550" hardware component has to generate the missing signal that are needed by the wishbone interface but do not exist in *IOb-SoC*. In table 4.5 the connection between the *IOb-SoC* and the *UART16550* wishbone interface can be analyzed. The missing signal are the "WB_SEL_I", the "WB_WE_I" and the "WB_STB_I". The select signal is similar to the "strb" signal but it should exist during write and read operations. This signal can be obtained through the

Port	Width	Direction	Description
CLK	1	Input	Block's clock input
WB_RST_I	1	Input	Asynchronous Reset
WB_ADDR_I	5 or 3	Input	Used for register selection
WB_SEL_I	4	Input	Select signal
WB_DAT_I	32 or 8	Input	Data input
WB_DAT_O	32 or 8	Output	Data output
WB_WE_I	1	Input	Write or read cycle selection
WB_STB_I	1	Input	Specifies transfer cycle
WB_CYC_I	1	Input	A bus cycle is in progress
WB_ACK_O	1	Output	Acknowledge of a transfer

Table 4.4: WISHBONE interface signals.

two Less Significant bit (lsb) of the address signal. And since registers are addressed byte by byte only one bit at a time will be set to high on the select signal. The write signal is set to high whenever the CPU wants to write to the *UART16550* registers. This can be perceived through the *IOb-SoC* "strb" signal. If any of the "strb" bits are enabled the "WB_WE_I" signal should be high ('1'). The transfer cycle should happen (i.e. "WB_STB_I" signal should be set to '1') when the *UART16550* has accepted a request but still has not issued the response. Furthermore, the *UART16550* has an interrupt output pin that is connected to the SoC "uartInterrupt". The SoC "uartInterrupt" is then passed to the PLIC unit.

UART16550 Wishbone	IOb-UART16550	IOb-SoC
CLK	clk	clk
WB_RST_I	rst	reset
WB_ADDR_I	address[‘UART_ADDR_WIDTH-1:0]	slaves_req[‘address(‘UART16550,32)]
WB_SEL_I	1<<address[1:0]	N/A
WB_DAT_I	wdata	slaves_req[‘wdata(‘UART16550)]
WB_DAT_O	rdata	slaves_resp[‘rdata(‘UART16550)]
WB_WE_I	| wstrb	N/A
WB_STB_I	valid&(∼ready)	N/A
WB_CYC_I	valid	slaves_req[‘valid(‘UART16550)]
WB_ACK_O	ready	slaves_resp[‘ready(‘UART16550)]

Table 4.5: *UART16550* interface with *IOb-SoC*.

Finally, the "iob_uart16550" also has to pass the interface between the *UART16550* core and the RS232 connector. That is why it implements the: txd output to transmit data through serial; the rxd input to receive data through serial; the cts input which indicates that the destination is ready to receive a transmission sent by the UART; the rts output which indicates that the UART is ready to receive a transmission from the sender. Those pins are connected to the development board RS232 connector.

4.4 CLINT Unit

The CLINT was the only hardware component that was developed from scratch. Even though, there already exist open-source Core-local Interrupt Controller (CLINT) hardware modules. For example, the CLINT used with the *CVA6* core and developed by the *PULP platform*. The problem with this CLINT module is that it is written in system-Verilog and uses packages and definitions from the *CVA6* core. The CLINT core is a simple hardware component. Considering that it only needs a few registers and signals to work, as it was studied in section 2.2. The interface with *IOb-SoC* would have to be developed independently of the CLINT core used. The best solution was to fully create the CLINT hardware unit.

The inputs and outputs of the CLINT unit can be seen in table 4.6. “N_CORES” is the number of CPU core that are used in the SoC. In the System on a chip (SoC) developed there is only one core. But the CLINT is built with a multi-core system in mind. Each core has its timer and software interrupt. As such, the “mtip” and “msip” registers width is the number of core, “N_CORES”.

Port	Width	Direction	Description	IOb-SoC Port
clk	1	input	System clock	clk
rst	1	input	System reset	reset
rt_clk	1	input	Real-time clock	rtc
valid	1	input	Indicates that the CPU is ready to make a data request.	slaves_req[‘valid(‘CLINT)]
address	32	input	Register address.	slaves_req[‘address(‘CLINT,16)]
wdata	32	input	Data to write to register.	slaves_req[‘wdata(‘CLINT)]
wstrb	4	input	Used to generate a "write" signal.	slaves_req[‘wstrb(‘CLINT)]
rdata	32	output	Data read from register.	slaves_resp[‘rdata(‘CLINT)]
ready	1	output	Indicates that the CLINT is ready to send a response.	slaves_resp[‘ready(‘CLINT)]
mtip	N_CORES	output	Raise a timer interrupt in a core.	timerInterrupt
msip	N_CORES	output	Raise a software interrupt in a core.	softwareInterrupt

Table 4.6: CLINT interface with *IOb-SoC*.

The “rt_clk” signal, although it is connected to “rtc” wire, is not available in *IOb-SoC* since the development board does not have any arithmetic logic unit (rtc) crystal connected. The rtc frequency is commonly 32.768 kHz, because it is a power of 2 (2^15) value. To get a precise 1-second period (1 Hz frequency) would only be needed a 15 stage binary counter. With a rtc the CLINT unit had to detect the rising edge of the “rt_clk” signal. The rising edge was detected by taking samples of the “rt_clk” signal at the system clock frequency. This is possible because the rtc is slower than the system clock. If in the future the CLINT unit developed is implemented in a system with a rtc the hardware design can be easily adapted to it. Since the logic was already developed.

In the development boards used there is no rtc so an alternative had to be found. The method

implemented was simpler than the logic if there existed a rtc. The system clock operates at 100 MHz so a counter was added to the design and each time that counter reached 999 the timer register would increment. Like this, a rtc working at 100 kHz was simulated in the CLINT hardware.

Concluding, the needed registers to develop a CLINT unit in accordance with the *RISC-V* specifications were implemented. And the hardware logic needed to trigger the timer and software interrupts were described.

4.5 PLIC Unit

The PLIC is not essential to run a full feature Operating System (OS) on a System on a chip (SoC). Since the PLIC is used to drive interrupts generated by other peripherals to the core, in this project the only peripheral connected to the PLIC is the *UART16550*. For the growth of the SoC, the PLIC unit is handy and a requirement to have. Some peripherals that might be added in the future also use the PLIC hardware, for example, the *ethernet* controller can be used to wake a core from low power mode.

Since the PLIC hardware unit is more complex then the CLINT, it was decided that an open-source PLIC core would be adapted to the *IOb-SoC*. There were three available cores. The PLIC developed by *lowRISC* [13] is written in their own variation of *System Verilog*, consequently it is difficult to adapt and test in simulation. The PLIC used with the *CVA6* core, developed by the *PULP platform* [17], started as a fork of an older version of the *lowRISC*. This PLIC unit is written in traditional *System Verilog*. I tried to adapt the PLIC unit from *PULP platform* but there were many incompatibilities with the *IOb-SoC*. Finally, the PLIC developed by *RoaLogic* [19] was the hardware unit used as a starting point. The *RoaLogic* PLIC is also written in *System Verilog* and implements an *apb4* or *ahb3lite* interface with the SoC. The biggest advantage of the *RoaLogic* hardware is that the PLIC relevant components for the *IOb-SoC*, the PLIC registers and core, are well separated from the modules that create the interface with the SoC and instantiate them. To integrate the a PLIC unit on the *IOb-SoC* I had to develop a PLIC wrapper.

The PLIC wrapper had to create the interface with the *IOb-SoC* internal buses. Furthermore, it needed to instantiate the PLIC registers and core hardware modules.

The PLIC hardware module was named “iob_plic”. When instantiating the “iob_plic” there are various parameters the developer can customize that influence the hardware. The number of interrupt sources defines how many peripherals interrupt signals are connected to the PLIC. In this project, the number of interrupt sources is one (the *UART16550*). The number of interrupt targets is normally equal to the number of CPU cores. The PLIC will send an external interrupt signal to each target. The number of priority levels in the developed SoC is 8 and each interrupt source has a register that holds the value for its priority. The maximum number of ’pending’ events is the maximum number of peripheral interrupts the PLIC will register while waiting for

the CPU to solve the current interrupt. There is also a parameter that defines if 'threshold' is implemented. The last configurable parameter determines whether the 'configuration' register is implemented. The changes in the parameters influence the number of registers and wires the most. The hardware logic does not change.

The "iob_plic" interface with the *IOb-SoC* can be seen in table 4.7. The value of "SOURCES" is 8. It was decided to be this value because although right now the SoC has only one source in the future it might have up to 8 sources. The value of "TARGETS" is 1 since there is only one core to inform of an existing interrupt.

Port	**Width**	**Direction**	**Description**	**IOb-SoC Port**
clk	1	input	System clock	clk
rst	1	input	System reset	reset
valid	1	input	CPU is sending a request.	slaves_req[‘valid(‘PLIC)]
address	32	input	Register address.	slaves_req[‘address(‘PLIC, 16)]
wdata	32	input	Data to write to register.	slaves_req[‘wdata(‘PLIC)]
wstrb	4	input	Used to generate a write and a read enable signal.	slaves_req[‘wstrb(‘PLIC)]
rdata	32	output	Data read from register.	slaves_resp[‘rdata(‘PLIC)]
ready	1	output	PLIC is ready to send a response.	slaves_resp[‘ready(‘PLIC)]
src	SOURCES	input	Peripherals Interrupts	{uartInterrupt, {7{1’b0}}}
irq	TARGETS	output	Informs targets of existing interrupt.	{externalInterrupt}

Table 4.7: PLIC interface with *IOb-SoC*.

Furthermore, the PLIC response signal is always ready one clock cycle after a request is received. Additionally, the PLIC has write enable and read enable signals. The write enable signal can be generated from equation 4.5. The read enable is obtained similarly but when all bits of the "strb" signal are '0', equation 4.6.

$$valid \cdot (wstrb[3] + wstrb[2] + wstrb[1] + wstrb[0]) \tag{4.5}$$

$$valid \cdot \overline{(wstrb[3] + wstrb[2] + wstrb[1] + wstrb[0])} \tag{4.6}$$

5 Software Developed

Simultaneously with developing the hardware logic components, software was developed during this project. The author first developed software for communication with the *IOb-SoC* through serial. Secondly, he programmed an application for hardware verification that used open-source logic simulators. Thirdly, he wrote firmware that could test and run interrupt routines. Fourth, he adapted and built the needed software/firmware to execute an Operating System on the SoC developed. Finally, the author wrote multiple Makefiles that facilitated user interaction and further development.

Some software developed was not mandatory to get a full-fledged Operating System to work with the System on a chip designed. The author worked on complementary software because it facilitates project development. Furthermore, the additional software allows the *IOb-SoC* platform to support more features.

In this chapter, the software developed will be analysed. The new *Python Console* and the new simulation system were already implemented on the upstream *IOb-SoC* template. Other developers are already using this software. Moreover, the *IObundle* developers using it have already been improving the software.

Developers write new Makefiles while developing new hardware and software tools. In this project, the author based some of the new Makefiles on existing *IOb-SoC* Makefiles. In contrast, the author wrote other Makefiles from scratch. The Makefiles help to automatise build processes and simplify the SoC usage.

5.1 *Python Console*

The *Console* is a program that runs on the user computer and communicates with the board where the *IOb-SoC* is implemented using an *RS-232* connection. Initially the *IOb-SoC* had a *Console* written in *C* programming language. One of the first tasks developed was translating the *Console* program to *Python*.

The *C Console* uses a set of open-source functions present on an external file that *IObundle* developers found to read/write to the serial port. The *Python* program uses the *PySerial* library, which provides ready-made communication functions like those in the original *C* code. Using *PySerial* is better because the community regularly maintains and updates *PySerial*. *PySerial* provides additional features, is less prone to have bugs, and communication is more trustworthy compared with the *C* functions.

One of the reasons for translating the *Console* program was to integrate an existing Ethernet controller already written in *Python*. *Python* can easily exploit feature like files, sockets and other Operating System (OS) functionalities.

Users can use the *Python Console* program in two different modes: locally working with simulators or communicating with a board running *IOb-SoC*. The program mode can be choose when calling the *Console* through adding “-L” or “–local” to the invoking arguments. Working in different modes is an additional feature to the original *Console* program. The *C Console* could only works with the FPGA board. When the *Console* is run in board mode a physical implementation of *IOb-SoC* runs on the board and communicates with *Console* through a *RS-232* serial connection. If the *Console* is called with the “-L” or “–local” augment it will communicate with the simulator. The communication with the hardware simulation is identical to the one with the board. They exchange the same messages. When communicating with the simulator, the *Console* uses files to send and receive data from the *IOb-SoC* hardware simulation. When stating, the *Console* program creates two empty files in the simulation directory. The “cnsl2soc” is used to send messages from the *Console* to the SoC. The “soc2cnsl” is used by the *Console* to receive messages from the SoC. Both files only contain one byte at a time. Whether the files are empty or not is used to synchronise the simulation with the *Console*. After reading from one of the files, the simulation or the *Console* program has to empty the respective file.

How the code is structured is very similar to how it was on the *C Console* program. It starts by defining the parameters that influence message identifiers and serial communication (for example, the number of bits per byte, the parity and the number of stop bits). When the program enters its primary function, it starts a loop where it waits for an available byte to read, either from the serial port or the file, depending on the *Console* execution mode. After receiving the byte from the SoC, it computes what type of message it is. The program exits successfully if the byte received is an End of Transmission (EOT = 04, ASCII value in hexadecimal). If the byte received is an Enquiry (ENQ = 05, ASCII value in hexadecimal), the program checks if it was the first time it received an enquiry. If it was, it could have one of either behavior: if the program was called with an argument equivalent to “-f”, meaning that there is a firmware that should be uploaded to the *IOb-SoC*, the *Console* sends a Send a file request message (FRX = 08, value in hexadecimal) to the *IOb-SoC*; if there is not a firmware file to send then the *Console* responds to the *IOb-SoC* with an Acknowledgement (ACK = 06, ASCII value in hexadecimal). If the *Console* receives a Receive a file request message (FTX = 07, value in hexadecimal), it will run a function that will receive any file sent from the *IOb-SoC* to the computer and save it under the directory where it is running. If the *Console* receives a *Send a file request* message, it will run a function that will send any file requested from the *IOb-SoC* to it. Any other byte received will be printed onto the stdout.

The FRX and the FTX bytes are specific to the *IOb-SoC* platform software. Being platform-specific could cause a problem when using external software that does not attribute the same meaning to their respective values. To solve this problem a meaning for the Device Control 1 (DC1 = 11, ASCII value in hexadecimal) byte was created. When receiving a DC1 byte the *Console* deactivates all platform-specific meanings for the respective bytes. This means that after receiving a DC1 byte stops associating the value 0x07, 0x08, 0x11 to FTX, FRX and DC1 respectively.

An example of how to call the *Console* to communicate with the simulation and send the firmware to the *IOb-SoC* when it starts would be 5.1. The “&” at the end means that the *Console* program will run in the background. Consequently, it allows other programs to run while the *Console* executes.

```
1    CONSOLE_CMD=$(CONSOLE_DIRECTORY)/console -L -f &
```

Listing 5.1: Call *Console* program

5.2 *IOb-SoC* Simulation

In order to support the new *Console* simulation mode a new *IOb-SoC* verification mechanism had to be developed. Verification is an important concern when developing hardware. As a result, it is unnecessary to synthesise and flash the hardware to an FPGA every time a developer wants to test a new feature. A correct and precise verification saves time.

The original simulation testbench was written in *Verilog* HDL. The simulation testbench in *Verilog* is a hardware module without any input or output signals. In this hardware module it was instantiated the Unit Under Test (UUT), a testing UART and the DDR memory. The UUT was equivalent to the SoC tested during the simulation. The testbench used the test UART to simulate an RS232 interface with the UUT. The DDR memory would only be instantiated when the SoC used an external DRAM memory.

Inside the initial block is where the developers described the simulation behaviour. The testbench runs the initial block in a *Verilog* hardware module only once when it instantiates the module. The original testbench starts by resetting the SoC. Resetting the SoC initialises the hardware registers to their reset values. The testbench configures the test UART to communicate with the SoC UART. Developers must configure both UARTs to use the same baud rate. After the initial setup, the simulation enters an “infinite” loop.

The testbench uses the “infinite” loop to send and receive bytes from the SoC until the simulation finishes. Thus the loop is not infinite, but the loop condition is always true. There exists a file called “cpu_tasks.v” that assists the testbench and contains multiple *Verilog* tasks. *Verilog* tasks are similar to *C* functions. The tasks in “cpu_tasks” are used to read and write to the test UART. The testbench uses a task to get a byte from the SoC that waits for the UART to have an available byte to read and reads it. Then, the testbench proceeds to process the received byte. If the received byte was a control byte, the testbench responds to the SoC by writing to the test UART. If not, the console prints the received byte to the stdout in the terminal. The control bytes could be an Enquiry (ENQ), End of Transmission (EOT), Receive a file request (FTX) or Send a file request (FRX).

The simulation testbench would successfully end when it received an EOT byte from SoC. The simulation would end abruptly when a trap notification was received. The “trap” signal was enabled (set to ’1’) when the CPU encountered an illegal instruction.

When the old testbench interacted with the test UART it emulated the *Console* program. Consequently, every time developers updated the *Console*, the simulation also had to be updated. Hence, the idea was to create a testbench that allowed the simulator to interact with the *Console* program. The new simulation now has the advantage of mostly using the same *Console* program as when the *IOb-SoC* is implemented in an FPGA.

The new verification software separates the previous simulation testbench into two parts. One of the parts is a hardware top module, and all hardware logic simulators use the hardware top module. The other is the simulation testbench, which is specific to each simulator. The simulation testbench interacts with the UUT through the hardware top module. The new testbench does not use the “trap” signal; only the old simulation uses it. Since the author swapped the CPU, the CPU no longer uses the trap signal to notify if something went wrong. Now the CPU handles trap exceptions on its hardware. Furthermore, they might not always mean the end of the execution or failure. The reader can see a sketch of the verification software in figure 5.1.

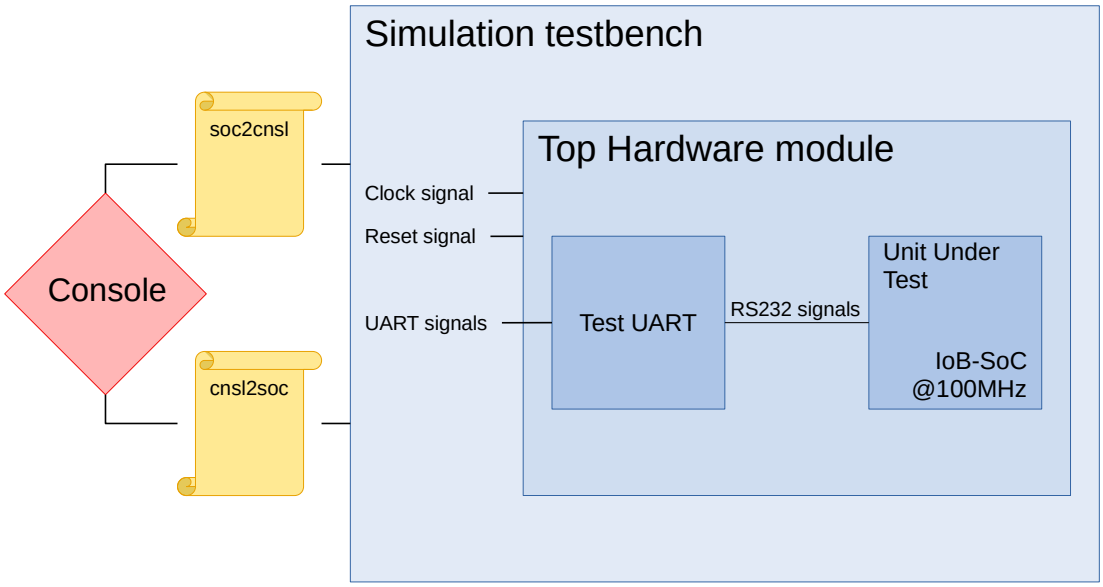

Figure 5.1: Simulated hardware interfaces.

5.2.1 Top Hardware module

This top module creates a *verilog* wrapper of the Unit Under Test (UUT). The UUT interacts with the different hardware logic simulators through this wrapper. Developers can never implement the top hardware module as real hardware. Developers only use this module in simulation as software.

The top module file adapts a part of the previous *verilog* simulation testbench. In the top module, the simulation only uses the initial block to obtain the system simulation signals waves. Developers can use these waves to debug the behaviour of the simulated hardware components. The user can define if the signal waves should be saved or not. The top hardware module instantiates the Unit Under Test (UUT), a testing UART and the DDR memory. The UUT shares an interface with the test UART and the DDR memory.

The test UART is connected to the SoC under test through the “rx” and “tx” pins. The “rx” pin in the SoC is connected to the “tx” pin in the test UART and is used to send information from the test UART to theSoC. Similarly, the “tx” pin in the SoC is connected to the “rx” pin in the test UART and is used to send information from the theSoC to the test UART. The interface between the SoC and the test UART simulates an RS232 connection.

The author described the input and output signals with which the top hardware module integrates with the simulation testbench in table 5.1.

Port	Width	Direction	Description
clk	1	input	The system clock signal generated in the simulation testbench.
rst	1	input	The system reset signal generated in the simulation testbench.
trap	1	output	Not used.
uart_valid	1	input	Used for the simulation testbench to make a write/read request to the test UART.
uart_addr	32	input	Indicates the test UART register to which the simulation testbench wants to write/read.
uart_wdata	32	input	Data that the simulation testbench wants to write in the test UART.
uart_wstrb	4	input	Select bytes from uart_wdat to write to the test UART register.
uart_rdata	32	output	Data sent from the test UART to the simulation testbench as a response to a read request.
uart_ready	1	output	Used by the test UART to indicate that the response to a write/read is ready.

Table 5.1: Inputs and outputs of the top hardware module used in the simulation.

Although in this project SoC the *IOb-UART* in the System on a chip (SoC) was swapped for the *UART16550* in the simulation testbench the *IOb-UART* is still used. If the CLINT unit used a arithmetic logic unit derived from the board, the arithmetic logic unit should be added in the

simulation testbench. When added to the testbench, the top hardware module would have an additional input for the rtc.

5.2.2 Simulation Testbench

The author developed two simulation testbench during this project. One testbench was written in *Verilog* and is used by some simulators as for example *Icarus Verilog* and *xcelium*. Another testbench was written in *C* programming language and is used by the *Verilator* hardware logic simulator.

The new *Verilog* simulation testbench developed instantiates only the top hardware module. At the beginning of the testbench module, the testbench generates the system clock. If the development board had a arithmetic logic unit, the testbench should also generate the rtc signal. The reader can see the creation of both these signals in the code snippet 5.2. The rtc signal does not apply to the SoC developed in the end, so the developer wrote the code that generated it as a comment.

```
1    parameter realtime clk_per = 1s/`FREQ;
2    //parameter realtime rtc_per = 1s/`RTC_FREQ;
3
4    //clocks
5    reg clk = 1;
6    always #(clk_per/2) clk = ~clk;
7    //reg rtc = 1;
8    //always #(rtc_per/2) rtc = ~rtc;
```

Listing 5.2: System clock and rtc generation in *Verilog*.

The new testbench initial block starts by executing the same procedure as the previous testbench. It first sends a reset signal to the system and initialises the test UART. The testbench initialises the test UART with a specified baud rate. Before entering an “infinite” loop, the new testbench will check if the file used to communicate with the *Console* exists. This file is called “soc2cnsl”. The file “soc2cnsl” should have been created by the *Console* program, executed before the testbench. If this file does not exist, the simulation will end abruptly. When the simulation ends unsuccessfully, it informs the user of its cause. The *Console* creates the “cnsl2oc” file simultaneously with the “soc2cnsl”, but the testbench will only check if that file exists inside the loop. The testbench uses the “cnsl2oc” file to receive messages from the console. While it uses the “soc2cnsl” file to send messages to the console.

The “infinite” loop is a while statement whose loop condition is always true, similar to the old testbench. Inside the loop the testbench will read the test UART “rx” ready register and the “tx” ready register until either one of them is enabled (set to ’1’). As can be seen in the code snippet 5.3. The “cpu_uartread” task is the same task used in the previous testbench

to read from the test UART. The first argument of the task is the address of the UART register, which is to read. The second argument is the variable that saves the value of the register.

```
1    while(!rxread_reg && !txread_reg) begin
2        cpu_uartread(`UART_RXREADY_ADDR, rxread_reg);
3        cpu_uartread(`UART_TXREADY_ADDR, txread_reg);
4    end
```

Listing 5.3: Read the test UART "rx" ready register and the "tx" ready register.

After exiting the while, the testbench executes code depending on the register's value. If the variable used to store the "rx" ready register value ("rxread_reg") is true the testbench will read a byte sent by the SoC under test and send it to the *Console*. To do that, it will first open the "soc2cnsl" file in reading mode and check if it is empty. If the file is empty, the testbench will close the file and read the value stored in the test UART "rx" data register. Then the testbench will reopen the "soc2cnsl" file in write mode and write the value read. After writing to the file the testbench will clear the "rxread_reg" and close the "soc2cnsl" file. If the file is not empty, the testbench will just close it and verify if it is empty again in the next loop. The program logic is seen in code snippet 5.4. The simulation clock signal only advances when the testbench reads or writes to the test UART. By not clearing the "rxread_reg" variable, the testbench will wait until the "soc2cnsl" file is empty before proceeding with the simulation.

```
1    if(rxread_reg) begin
2        soc2cnsl_fd = $fopen("soc2cnsl", "r");
3        n = $fgets(cpu_char, soc2cnsl_fd);
4        if(n == 0) begin
5            $fclose(soc2cnsl_fd);
6            cpu_uartread(`UART_RXDATA_ADDR, cpu_char);
7            soc2cnsl_fd = $fopen("soc2cnsl", "w");
8            $fwriteh(soc2cnsl_fd, "%c", cpu_char);
9            rxread_reg = 0;
10       end
11       $fclose(soc2cnsl_fd);
12   end
```

Listing 5.4: Write byte from SoC to *Console*.

If the variable used to store the "tx" ready register value ("txread_reg") is true the testbench will read a byte sent by the *Console* and send it to the SoC under test. In order to do that, the testbench will first try to open the "cnsl2soc" file in reading mode. If opened successfully, it would proceed, read the first byte in the file, and save it in the "cpu_char" variable. When the read is successful, the testbench will write the "cpu_char" to the test UART. Then the testbench will close the "cnsl2soc" file and reopen it in write mode. This way, the file will be truncated to have 0 bytes. If the read was unsuccessful, it means the *Console* does not want to send aa Byte to thee SoC. Then the testbench will just ignore that part of the code. Finally the "cnsl2soc" file is closed and the "txread_reg" variable is cleared. The simulation finishes if the testbench cannot open the "cnsl2soc" file. Normally this means that the simulation

was successful. The “cnsl2soc” file is deleted by the *Console* program when the SoC send a End of Transmission (EOT) Byte. The EOT Byte means the SoC has finished running the firmware.

```
1   if(txread_reg) begin
2       cnsl2soc_fd = $fopen("cnsl2soc", "r");
3       if (!cnsl2soc_fd) begin
4           $finish;
5       end
6       n = $fscanf(cnsl2soc_fd, "%c", cpu_char);
7       if (n > 0) begin
8           cpu_uartwrite(‘UART_TXDATA_ADDR, cpu_char, ‘UART_TXDATA_W/8);
9           $fclose(cnsl2soc_fd);
10          cnsl2soc_fd = $fopen("./cnsl2soc", "w");
11      end
12      $fclose(cnsl2soc_fd);
13      txread_reg = 0;
14  end
```

Listing 5.5: Write byte from *Console* to SoC.

The *Verilator* testbench is similarly structured to the *Verilog* testbench. The code in the initial block written for the *Verilog* testbench also applies to the *Verilator* testbench. The code only had to be translated from *Verilog* to *C* language. One of the main differences between the *Verilator* testbench and the *Verilog* testbench is how the simulators generate the clock signals. *Verilator* executes the simulation through cycles. Executing the simulation through cycles means that the testbench controls the time advancement. With that in mind, the author implemented a global variable that saved the current time and created a function that advances time. The reader can see the function in code snippet 5.6. The function, when executed, advances the number of nanoseconds passed as an argument. Every nanosecond passed, a cycle has passed, and *Verilog* verifies the state of the Unit Under Test (UUT). The system clock signal has to change value every half of the clock period. If the hardware used a rtc, the rtc value would have to change every half of the defined arithmetic logic unit period.

```
1   void Timer(unsigned int ns){
2       for(int i = 0; i<ns; i++){
3           if(!(main_time%(CLK_PERIOD/2))){
4               uut->clk = !(uut->clk);
5           }
6           //if(!(main_time%(RTC_PERIOD/2))){
7           //  uut->rtc_in = !(uut->rtc_in);
8           //}
9           uut->eval();
10          main_time += 1;
11      }
12  }
```

Listing 5.6: *Verilator* Timer function.

In the previous testbench, the *IObundle* developers had developed tasks that initialise the test UART and read/write to the registers. The new *Verilog* testbench uses the tasks developed by *IObundle*. For the *Verilator* testbench those functions had to be rewritten in *C*. The test UART initialization starts by resetting the test UART hardware by writing to the “softreset” register. Then it writes to the “div” register the value needed for the UART to work with the defined SoC frequency and baud rate. Finally it enables the “rx” and “tx” communication by writing ’1’ to the “rxen” and “txen” registers respectively. The initialization function can be seen in code snippet 5.7.

```
1   void inituart(){
2       //pulse reset uart
3       uartwrite(UART_SOFTRESET, 1, UART_SOFTRESET_W/8);
4       uartwrite(UART_SOFTRESET, 0, UART_SOFTRESET_W/8);
5       //config uart div factor
6       uartwrite(UART_DIV, int(FREQ/BAUD), UART_DIV_W/8);
7       //enable uart for receiving
8       uartwrite(UART_RXEN, 1, UART_RXEN_W/8);
9       uartwrite(UART_TXEN, 1, UART_TXEN_W/8);
10  }
```

Listing 5.7: Function to initialize the test UART.

The testbench takes two clock cycles to read a test UART register. In the first clock cycle, the testbench makes a read request to the UART. The testbench sets the UART address signal to the address of the register that the testbench wants to read, and the UART valid signal is set to ’1’ to make the request. In the second clock cycle, the testbench reads the response data in the “rdata” signal. The read function can be seen in code snippet 5.8.

```
1   void uartread(unsigned int cpu_address, char *read_reg){
2       dut->uart_addr = cpu_address >> 2; // 32 bit address (ignore 2 LSBs)
3       dut->uart_valid = 1;
4       Timer(CLK_PERIOD);
5       *read_reg = (dut->uart_rdata) >> ((cpu_address & 0b011)*8); // align to
    32 bits
6       dut->uart_valid = 0;
7   }
```

Listing 5.8: Read from the test UART.

The testbench has to send a write request signal to the test UART to write on a UART register. To make a write request the testbench as to set the UART address sign, the UART valid, the UART “wdata” signal and the UART “wstrb” signal. The testbench sets the address signal to the address of the register where it wants to write and valid to ’1’. The testbench also sets the “wdata” to the data the developer wants to write on the register. Additionally, it sets the “wstrb” to indicate which Bytes from the “wdata” the developer wants to write on the register. After one clock cycle, the UART completes the writing process, and the valid signal has to be set back to ’0’. The write function can be seen in code snippet 5.9.

```
1  void uartwrite(unsigned int cpu_address, unsigned int cpu_data, unsigned int
   nbytes){
2      dut->uart_addr = cpu_address >> 2; // 32 bit address (ignore 2 LSBs)
3      dut->uart_valid = 1;
4      dut->uart_wstrb = nbytes << (cpu_address & 0b011);
5      dut->uart_wdata = cpu_data << ((cpu_address & 0b011)*8); // align data
  to 32 bits
6      Timer(CLK_PERIOD);
7      dut->uart_wstrb = 0;
8      dut->uart_valid = 0;
9  }
```

Listing 5.9: Write to the test UART.

5.3 Interrupt Routine

During the development of the CLINT hardware unit, no firmware used the CLINT features. The CLINT enables the support for time or software-related interrupts. Therefore, the author needed to create a simulation testbench to test the CLINT hardware. Moreover, the author also created bare-metal firmware that uses interrupts to understand how interrupts are used in code and handled by the CPU.

The software has to write to the “MTIMECMP” register to generate a time-related interrupt. The “MTIMECMP” register address is the “MTIMECMP_BASE” address, which is 0x4000, plus 0x08 times the core id. The core id is the “TARGET”, CPU core, which is supposed to receive the interrupt notification. Since there is only one core in this project, the core id is 0. Furthermore, when writing firmware to run on the SoC, it is necessary to consider the CLINT peripheral base address and add it to the “MTIMECMP” register address. For a timer interrupt to trigger after 10 seconds, the firmware has to do more than just write to the “MTIMECMP”. First, the firmware has to read the current time. A developer can obtain the current time from the “MTIME” register. Although not directly since the “MTIME” register increments with the CLINT designed frequency, in this case 100MHz. To convert the value in “MTIME” register to seconds we know that $seconds = \frac{*(MTIME)}{frequency}$. The “MTIME” register address is obtained similarly to the “MTIMECMP” register but instead of the “MTIMECMP_BASE”, the “MTIME_BASE”, which is 0xbff8, is used. After reading the current time, the software calculates the value that CPU needs to store in “MTIMECMP” register. The software calculates the value by adding to the current time the time waited before the CLINT hardware triggers the interrupt. The value calculated can then be stored in the “MTIMECMP” register. When the “MTIME” register value is equal to or greater than the “MTIMECMP” register value, the timer interrupt is enabled. The pseudo-code to set up the timer interrupt is in the code snippet 5.10.

```
1  #define MTIMECMP_BASE 0x4000
2  #define MTIME_BASE 0xbff8
3  #define FREQ 100000000
4  void set_up_mtip(time_sec){
```

```
5         long long aux_value = 0; // 64-bit integer
6         int core_id = 0;
7         aux_value = *(MTIME_BASE+8*core_id)
8         aux_value = aux_value + time_sec*FREQ;
9         *(MTIMECMP_BASE+8*core_id) = aux_value;
10    }
```

Listing 5.10: Set Up Timer Interrupt.

The CPU could have readen the core id from the CSR, which saves its value. The *RISC-V* instruction that does so is “csrr %0, mhartid”. An example of a C code integration would be code snippet 5.11.

```
1     static inline uint_32_t csr_read_mhartid(void) {
2         uint_32_t value;
3         __asm__ volatile ("csrr    %0, mhartid"
4                           : "=r" (value)  /* output : register */
5                           : /* input : none */
6                           : /* clobbers: none */);
7         return value;
8     }
```

Listing 5.11: Read core id from CSR.

One of the software interrupt usages is synchronising various cores in a system. When dividing the workload between cores, there might be a time when core 1 has to synchronise with core 0. Core 1 would wait until core 0 generates a software interrupt targeting core 1. This project has only one core, so this situation does not occur. Nevertheless, applications can run concurrently with multi-threading. One application could wait until a software interrupt is triggered. The CPU could trigger a software interrupt targeting core 0, using another application running in core 0. The software has to write to the “MSIP” register to generate a software-related interrupt. The “MSIP” register address is the “MSIP_BASE” address, which is 0x00, plus 0x04 times the core id. When executing the firmware, the developers must not overlook adding the CLINT peripheral base address to the “MSIP” register. Only hardware external to the CLINT unit can change the state of the “MSIP” register. The CLINT hardware cannot change it internally. The pseudo-code to set up the software interrupt is in the code snippet 5.12.

```
1     #define MSIP_BASE 0x00
2     void set_up_msip(){
3         int core_id = csr_read_mhartid();
4         *(MSIP_BASE+4*core_id) = 1;
5     }
```

Listing 5.12: Set Up Software Interrupt.

After enabling an interrupt, the CLINT sends a hardware notification. The interrupt notification has to be handled by the rest of the hardware. The CLINT testbench and the bare-metal firmware handle the interrupt notification differently.

5.3.1 CLINT simulation

The author built a simulation testbench in Verilog and another in C++ programming language to test the CLINT hardware unit. These simulations allow the developers to test the correctness of the hardware component without connecting it to the rest of the SoC.

Both the testbench in Verilog and the testbench in C++ had similar behaviours. First, they would set up a timer interrupt. The testbench set the timer interrupt to trigger in $0.2 * 10^{-6}$ seconds. Considering that the simulation was slow, this was a reasonable time. After the timer interrupt is triggered, the simulation receives its notification and proceeds to handle the interrupt. When receiving a timer interrupt, the simulation sets up the software interrupt. The CLINT unit notifies the testbench of an existing software interrupt in the next clock cycle. It then proceeds to disable the timer and the software interrupt. To disable the timer interrupt the "MTIMECMP" register has to be set to its maximum value, which is 0xFFFFFFFFFFFFFFFF (i.e. all 64 bits are '1'). The "MSIP" register had to be set to 0 to disable the software interrupt.

If the interrupts work correctly, there will be a message in the terminal indicating their correctness. After testing that both interrupts work as expected, the testbench can finish successfully. The simulation will always end $1 * 10^{-6}$ seconds after starting.

5.3.2 Bare-metal firmware

Once the author developed the CLINT unit, even though he knew that the CLINT generated the interrupts through the simulation testbench, he had to test it while integrated into the SoC. To test the CLINT in the SoC firmware that took advantage of the timer and software interrupts had to be developed.

Since the CLINT hardware developed is compatible with *RISC-V*, any firmware compatible with *RISC-V* that took advantage of interrupts should work. With this in mind, the open-source bare-metal firmware made available by *Five EmbedDev* [8], an embedded *RISC-V* blog, would be taken advantage of. The firmware could not all be used directly in *IOb-SoC*. Some functions that interact with the timer in the CLINT were adapted to the developed hardware and used. Furthermore, as a library, the firmware uses a file that implements functions that read/write to the Control and status register.

The developed firmware.c, similarly to the original *IOb-SoC* firmware, starts by initializing the UART and the CLINT hardware. Then it will disable all global interrupts. To disable the global interrupts the "mstatus" CSR, "mie" CSR and the "mcause" CSR are cleared. After the timer interrupt is set similarly to the code in 5.10. The program counter has to jump to a specific function when an interrupt occurs. The memory address of that function is saved in the "mtvec" CSR. Succeeding the timer interrupt set up, the respective interrupt bit can be set to '1' (i.e. enable the timer interrupt) in the "mie" CSR. Following this, the global interrupts can be enabled again by setting the needed bits in "mstatus" CSR to '1'. The bits that the firmware needs to set

to ’1’ correspond to the machine interrupts (MSTATUS_MIE_BIT_MASK). Finally, the program can wait for an interrupt to happen with the “wfi” instruction.

When an interrupt occurs, the CPU calls the interrupt handler function. The interrupt handler will read the cause of the generated interrupt from the “mcause” CSR. After knowing the cause, the interrupt handler will act accordingly to how it was programmed. The developed firmware informs the user that a timer interrupts occurred, and it sets the “MTIMECMP” register to a higher number. The developed firmware finishes after receiving the first interrupt.

It is important to note that besides the firmware.c some alterations had to be made in the *IOb-SoC* firmware.S file. In the firmware.S, the author had to set the global pointer register. The global pointer is similar to the stack pointer. The difference is that while the stack pointer points to the memory location where function variables will be stored, the global pointer points to the memory location where global variables are stored. When setting the global pointer, it is critical to write the “norelax” option in the Assembly code. Without “.option norelax”, the software will load the global pointer relative to the global pointer set in the assembly code.

5.4 IOb-SoC Linux OS integration

The last step in this thesis development was creating and integrating a minimal Operating System (OS) on the developed System on a chip (SoC). The minimal OS developed has the Linux software as its kernel. The Linux kernel allows the user applications run on the SoC to compile with the *riscv64-unknown-linux-gnu-** compiler. Applications compiled with *riscv64-unknown-linux-gnu-** can use the Linux system functions and drivers. Using Linux system functions and drivers allows the software developers to ignore the hardware platform where the software will run.

During the development of the OS to run on the SoC, the author had to adapt the bootloaders firmware and write the device tree describing the developed SoC. Additionally he had to compile the Linux kernel compatible with the *VexRiscv* CPU implemented on the SoC and create a root file system. Moreover, additional features had to be developed for the user to interact with the OS.

The author developed a Makefile target for each component that automates the build process. Furthermore, he created a general target called “build-OS”. “build-OS” calls all other targets and builds a full-featured minimal OS.

5.4.1 Bootloaders

Before running the Linux kernel software, the SoC has to run two Bootloaders firmware. The first firmware to execute is the stage 0 bootloader. The stage 0 bootloader is an adaptation of the bootloader firmware used in *IOb-SoC*. The second bootloader is a *RISC-V* specific software called *OpenSBI*.

The stage 0 bootloader has to copy the developed OS to the external memory of the SoC running on the FPGA board. The bootloader will send a request to the *Console* program asking for the binary data of the *OpenSBI*, the device tree, the Linux kernel and the root file system. Each file transferred will be stored in a specific memory location. The transfer bootloader code can be seen in 5.13.

```
1     prog_start_addr = (char *)(EXTRA_BASE + 0x00000000);
2     file_size = uart_recvfile(opensbi, prog_start_addr);
3     prog_start_addr = (char *)(EXTRA_BASE + 0x00400000);
4     file_size = uart_recvfile(kernel, prog_start_addr);
5     prog_start_addr = (char *)(EXTRA_BASE + 0x00F80000);
6     file_size = uart_recvfile(dtb, prog_start_addr);
7     prog_start_addr = (char *)(EXTRA_BASE + 0x01000000);
8     file_size = uart_recvfile(rootfs, prog_start_addr);
```

Listing 5.13: Transfer OS to the SoC external memmory.

When finishing, the bootloader has to ensure that it clears all of the CPU's function argument registers. The bootloader has to write '0' to a function argument register to clear it. An example of clearing the first function argument register is seen in listing 5.14. There are in total eight function argument registers in a *RISC-V* CPU core. If the bootloader does not clear the function argument registers, the CPU might pass unwanted arguments to the next software that executes.

```
1     asm volatile("and     a0,a0,zero");
```

Listing 5.14: Clear a function argument register.

The *OpenSBI* bootloader is platform-specific so the configuration files compatible with the developed SoC had to be created. The *OpenSBI* software provides a template for the platform-specific configuration files. In the "config.mk" file the author specifies that only the *OpenSBI* "fw_jump.bin" binary needs to be generated. The file that differs the most from the template example is the "platform.c". In the "platform.c", the author configures the functions the *OpenSBI* bootloader runs when executed. Moreover, in the "platform.c" it is also designated the PLIC hardware address, the CLINT register addresses, the CLINT timer frequency, the *UART16550* hardware address, the *UART16550* clock frequency and the *UART16550* baud rate.

The *OpenSBI* bootloader initializes the PLIC, the CLINT and the *UART16550* hardware units. When the PLIC and the CLINT initialization is successful the *OpenSBI* configures the Linux

interrupt handler. Furthermore, the *OpenSBI* checks if the device tree is compatible with the specified hardware. Finally, the *OpenSBI* bootloader will tell the CPU to start executing the Linux kernel. The device tree memory location is passed to the Linux kernel as an argument through the function argument registers.

The author created a Makefile target to build the *OpenSBI* software automatically. The Makefile target copies the custom platform configuration to the "OpenSBI/platform" directory and compiles the software for the developed hardware platform. The "build-opensbi" Makefile target can be seen in A.1.

```
1 build-opensbi: clean-opensbi os_dir
2     cp -r $(Custom_Platform_DIR)/opensbi_platform/* $(OpenSBI_DIR)/OpenSBI/
    platform/ && \
3         cd $(OpenSBI_DIR)/OpenSBI && $(MAKE) run PLATFORM=iob_soc
```

Listing 5.15: Makefile target to build OpenSBI.

In listing A.1 "PLATFORM=iob_soc" indicates that the configuration customized for the developed SoC is used.

5.4.2 Device Tree

A device tree file is needed to execute the Linux kernel on the developed SoC. The device tree file describes the hardware components in a specific SoC. Each component of the SoC is represented as a node in the Device Tree Source (DTS) file. The reader can see the complete device tree in annex A.1.

The most important nodes are the CPU and the memory. The CPU node describes the CPU architecture, the CPU cache and additional specific features. The CPU node description used by the author in the Device Tree Source (DTS) was based on the description suggested by the *VexRiscv* developer. The memory node indicates that the SoC DRAM start address is 0x80000000 and it has 512MB of available memory (i.e. memory length is 0x10000000). The DTS attribute "regs" indicates the respective hardware devices addresses in the SoC.

The "chosen" node is the only node that does not represent a real device. The "chosen" node indicates the arguments that should be passed to the Linux kernel when it boots. The "choosen" node also indicates where in the memory the root file system is located.

The SoC peripherals are inside a node which the author called "soc". The CLINT node indicates the CLINT addresses and the interrupts it can create. The author wrote the PLIC node on a comment. When a developer includes the PLIC node on the DTS, it causes the Linux kernel to panic. When the kennel panics, the OS stops working. The UART node indicated the systems baud rate, frequency, the UART address and how the registers should be accessed. All the peripherals nodes have an attribute called "compatible", which indicates the device drives with which the component is compatible.

Using the Device Tree Compiler (DTC) a developer is able to create a binary Device Tree Blob (DTB) from the Device Tree Source (DTS) files. The command needed to run to compile the DTS is 5.16.

```
dtc -O dtb -o $(DTB_DIR)/iob_soc.dtb $(DTS_DIR)/iob_soc.dts
```

Listing 5.16: Makefile target to build the device tree blob.

5.4.3 Linux kernel

To build a Linux kernel compatible with the developed SoC the kernel had to be compatible with the *VexRiscv* CPU in it. The *SpinalHDL* author has a Linux kernel fork compatible with the *VexRiscv* CPU. Consequently, a developer can use the *SpinalHDL* kernel fork to create the kernel to run with the SoC developed in this project.

The thesis author developed a Makefile target, called “build-linux-kernel”, to build the Linux kernel automatically. The author also created a configuration file adapted to the SoC. The Makefile target has to copy the customised configuration file to the configs directory related to the *RISC-V* ISA. Then the target has to configure the kernel build process with the copied file. Finally, the Makefile can compile the kernel and the resulting binary image copied to the OS directory.

```
build-linux-kernel: clean-linux-kernel os_dir
    cd $(Linux_DIR)/Linux && \
        cp $(Custom_Platform_DIR)/linux_config $(Linux_DIR)/Linux/arch/riscv/
    configs/iob_soc_defconfig && \
        $(MAKE) ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-
    iob_soc_defconfig && \
        $(MAKE) ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc)
    && \
        cp $(Linux_DIR)/Linux/arch/riscv/boot/Image $(VEX_OS_DIR)
```

Listing 5.17: Root file system Makefile target.

An alternative way to build the Linux kernel would be using *Buildroot*. *Buildroot* [**buildroot**] is a tool that simplifies and automates building a complete Linux system for an embedded system. Since it allows cross-compilation, anyone can build a root filesystem, a Linux kernel image and a bootloader for any existing CPU architecture using only their personal computer.

The author tested the build process using *Buildroot* and successfully created an executable Linux kernel. However, the process of creating the Linux kernel with *Buildroot* is slower than the previous method. For the development of the OS, *Buildroot* could be helpful in the future when creating a more complex system. A more complex system would be a SoC with more peripherals, for example, an Ethernet peripheral.

5.4.4 Root File System

The root file system (rootfs) stores the user applications that the users can execute while the OS runs. The kernel mounts the rootfs during the boot process. The rootfs contains the files and directories that the users can access within the OS.

The rootfs developed is populated with applications present in *Busybox* [31]. *Busybox* is a combination of essential programs commonly found on UNIX systems. Since the developers of *Busybox* developed it for size-optimization and limited resources, it is ideal for adding to a minimal Linux distribution that might run on an embedded SoC. However, its programs' functionality is limited compared to the equivalent programs in general Linux distributions. The *Busybox* applications will be able to be seen in the "/bin", "/sbin", "/usr/bin" and "/usr/sbin" directories of the rootfs.

The first program, called init, runs after the kernel boots. The init program has PID id 0. The Linux kernel uses the PID id to identify the different programs running. The init program is located at the root directory ("/") of the OS root file system. The init program is a shell script that starts by mounting files needed by the Linux kernel to work with user applications. Then it will greet the user with a welcoming message printed to the stdout. The greeting message indicates the time it took the system to boot. Finally, the init script will launch a Bourne shell or sh. The Bourne shell allows users to interact with the OS and launch other applications. The init script is seen in listing 5.18.

```
1  #!/bin/sh
2  echo "### INIT SCRIPT ###"
3  /bin/mkdir /proc /sys /tmp
4  /bin/mount -t proc none /proc
5  /bin/mount -t sysfs sysfs /sys
6  /bin/mount -t tmpfs none /tmp
7
8  echo 'IObundle-OpenCryptoLinux >   '
9  echo -e "\nThis boot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
10
11 /bin/sh
```

Listing 5.18: Init script.

The author developed a build-root file system Makefile target to automatise the process of creating a root file system. First, the Makefile script has to enter the directory where the *Busybox* software is. Then the Makefile script will compile the *Busybox* programs for the developed hardware platform and copy the binaries to an "./_install" directory. Succeeding the creation of the "./_install" directory, the Makefile target will copy the init program to it and create special files used by the Linux kernel during execution. Finally, the script will compress the "./_install" directory to a ".cpio.gz" format. The ".cpio.gz" format is a format that the Linux kernel can uncompress while booting. The Linux kernel will then use the uncompressed archive as the root file system. The build-root file system Makefile target can be seen in listing 5.19.

```
build-rootfs: clean-rootfs
    cd $(busybox_DIR)/busybox && \
        cp $(Custom_Platform_DIR)/rootfs_busybox/busybox_config $(busybox_DIR)/
    busybox/configs/iob_defconfig && \
        $(MAKE) ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-
    iob_defconfig && \
        CROSS_COMPILE=riscv64-unknown-linux-gnu- $(MAKE) -j$(nproc) && \
        CROSS_COMPILE=riscv64-unknown-linux-gnu- $(MAKE) install && \
        cd _install/ && cp $(Custom_Platform_DIR)/rootfs_busybox/init init && \
        mkdir -p dev && sudo mknod dev/console c 5 1 && sudo mknod dev/ram0 b 1
    0 && \
        find -print0 | cpio -0oH newc | gzip -9 > $(OS_DIR)/rootfs.cpio.gz
```

Listing 5.19: Root file system Makefile target.

The rootfs created can be tested with *QEMU* before implementing it on the developed SoC. The developer would have to compile the Linux kernel to be compatible with the *QEMU* virtual hardware. Then the rootfs can be simulated by running the bash command 5.20.

```
sudo qemu-system-riscv32 -nographic -machine virt -kernel linux/arch/riscv/boot/
    Image -append "root=/dev/ram init=/init ro console=ttyS0" -initrd rootfs.
    cpio.gz -bios default
```

Listing 5.20: root file system *QEMU* simulation.

5.4.5 User Interaction

After successfully creating the Operating System the interaction with the *Console* program was incomplete. The *Console* could receive and print messages sent by Linux to the stdout. However, the *Console* could not read user input and sent it to the Linux running on the SoC. When a user types a message on their computer, they will send the message to the stdin of the OS running in their personal computer. The stdin in the user's computer had to be redirected to the SoC serial input to send a message to the Linux running on the SoC.

The *Console* program had to check if there was data in the user's stdin before reading a Byte from it. After reading a Byte, the *Console* would write the Byte to the SoC serial input. In the code snippet 5.21 it can be seen the python code added to the *Console*. The author also had to ensure that the *Console* program would not block when reading from SoC serial output.

```
    if select.select([stdin, ], [], [], 0.0)[0]:
        user_str = stdin.read(1)
        if(user_str!=''):
            if SerialFlag:
                ser.write(bytes(user_str, 'UTF-8'))
            else:
                tb_write(bytes(user_str, 'UTF-8'))
```

Listing 5.21: Read user input and send to the SoC.

After receiving data from the serial input, the Linux running on the SoC would send that data to its stdin. Even though now the user could send information to the SoC, there was still a problem. The problem was that the terminal on his computer captured the user input. The terminal, by default, only sends data to the stdin after receiving a new line ('
n') character. Furthermore, by default, the terminal echoes the user input. Consequently, the user input would appear repeated.

A UNIX OS terminal has two modes of receiving user input. The default is the canonical mode, which waits until the user inserts a new line to send the data to the stdin. In the non-canonical mode, the terminal sends the data of a keyboard key to the stdin after the user presses it without editing. Developers can also configure the terminal to echo or not the character sent by the user. To configure the terminal, the author developed a python script that reads the current terminal configuration related to the stdin. Then the script will change the values corresponding to the canonical and the echo mode to the opposite of the original value. Before executing the script, the terminal, by default, is configured to echo the input in canonical mode. Changing the values to the opposite of the original value means that after executing the script, the terminal will be configured not to echo the input and be in non-canonical mode. Finally, the script will set the terminal configuration to the new configuration. The python script that changes the terminal mode is seen in the code snippet 5.22.

```
#!/usr/bin/python
import sys
import termios

stdin = sys.stdin
fd = stdin.fileno()

old = termios.tcgetattr(fd)
new = termios.tcgetattr(fd)
new[3] &= ~termios.ECHO
new[3] &= ~termios.ICANON

termios.tcsetattr(fd, termios.TCSAFLUSH, new)
print()
```

Listing 5.22: Enable or disable non-canonical mode.

The Makefile called by the user has to run the script in listing 5.22 in his computer before executing the console program and connecting to the SoC on the FPGA. However, the Makefile must only execute the script after it executes the file transfers. The file transferring copies the files in the user's computer to the computer connected to the FPGA board. The Makefiles executes the file transferring using the "rsync" command. The "rsync" command needs to function with the terminal in canonical mode. The current solution is to run the Python script right before the *Console* and right after the connection with the SoC ends. The first time the Makefile runs the script, the terminal mode changes to non-canonical and disables input echo. The second time the Makefile runs the script, the terminal is set to default again.

6 Project Results

6.1 System Running "Hello World!"

6.1.1 Simulation

6.1.2 FPGA Board

6.2 Interrupt Routines

6.3 Run/Boot Linux Performance

time that it takes to build a complete OS real 4m29,570s user 8m12,039s sys 0m56,887s

6.4 FPGA Resources Consumption

7 Conclusions

7.1 Achievements

7.2 Contributed Repositories

- **iob-soc**
- **iob-soc-OpenCryptoLinux**
- **iob-lib**
- **iob-vexriscv**
- **iob-uart16550**
- **iob-clint**
- **iob-plic**

7.3 Future Work

- ethernet
- development board with IOb-SoC

A | Annex

A.1 Annex 1 - Device Tree Source

```
/dts-v1/;

/ {
  #address-cells = <1>;
  #size-cells = <1>;
  model = "IOb-SoC, VexRiscv";
  compatible = "IOb-SoC, VexRiscv";
  cpus {
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    timebase-frequency = <100000>;
    CPU0: cpu@0 {
      clock-frequency = <100000000>;
            device_type = "cpu";
      reg = <0x0>;
      status = "okay";
            compatible = "riscv";
      riscv,isa = "rv32imac";
      mmu-type = "riscv,sv32";
      d-cache-block-size = <0x40>;
      d-cache-sets = <0x40>;
      d-cache-size = <0x8000>;
      d-tlb-sets = <0x1>;
      d-tlb-size = <0x20>;
      i-cache-block-size = <0x40>;
      i-cache-sets = <0x40>;
      i-cache-size = <0x8000>;
      i-tlb-sets = <0x1>;
      i-tlb-size = <0x20>;
      tlb-split;
      CPU0_intc: interrupt-controller {
        #interrupt-cells = <1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
      };
    };
  };
  memory@80000000 {
    device_type = "memory";
    reg = <0x80000000 0x10000000>;
  };
  chosen {
    bootargs = "rootwait console=hvc0 earlycon=sbi root=/dev/ram0 init=/sbin/
    init swiotlb=32";
    linux,initrd-start = <0x81000000>;
```

```
45      linux,initrd-end = <0x81800000>; // max 8MB ramdisk image
46    };
47    soc {
48      #address-cells = <1>;
49      #size-cells = <1>;
50      compatible = "iobundle,iob-soc", "simple-bus";
51      ranges;
52      clint@60000000 {
53        compatible = "riscv,clint0";
54        interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7>;
55        reg = <0x60000000 0xc0000>;
56        reg-names = "control";
57      };
58      // PLIC needs to be disabeld for tandem verification
59      //PLIC0: interrupt-controller@50000000 {
60      //  #address-cells = <0>;
61      //  #interrupt-cells = <1>;
62      //  compatible = "riscv,plic0";
63      //  interrupt-controller;
64      //  interrupts-extended = <&CPU0_intc 0xb>;
65      //  reg = <0x50000000 0x4000000>;
66      //  riscv,max-priority = <7>;
67      //  riscv,ndev = <0xa>;
68      //};
69      // Specifying the interrupt controller in the devicetree is not necessary.
70      // Furthermore, the IRQ 65535 will cause a ‘hwirq 0xffff is too large‘
      during
71      // Linux boot (occured with mainline linux 5.14.0).
72      uart@40000000 {
73        compatible = "ns16550a";
74        reg = <0x40000000 0x1000>;
75        clock-frequency = <100000000>;
76        current-speed = <115200>;
77        //interrupt-parent = < &PLIC0 >;
78        interrupts = <1>;
79        reg-shift = <2>; // regs are spaced on 32 bit boundary
80        reg-io-width = <4>; // only 32-bit access are supported
81      };
82    };
83  };
```

Listing A.1: Makefile target to build OpenSBI.

A.2 Annex 2

Annex 2

Bibliography

[1] Arm Holdings (arm). *The Cortex-A72 processor specifications*. <https://developer.arm.com/Processors/Cortex-A72>.

[2] Krste Asanovic et al. “The rocket chip generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[3] Jonathan Bachrach et al. “Chisel: constructing hardware in a scala embedded language”. In: *DAC Design automation conference 2012*. IEEE. 2012, pp. 1212–1221.

[4] Jonathan Balkind et al. “OpenPiton: An Open Source Manycore Research Framework”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414. <http://doi.acm.org/10.1145/2872362.2872414>.

[5] Seeed Studio BeagleBoard.org and StarFive. *BeagleV, The First Affordable RISC-V Computer Designed to Run Linux*. <https://beagleboard.org/static/beagleV/beagleV.html>.

[6] Sebastien Bourdeauducq. “Migen Manual”. In: *Release 0. X* 32 (2012).

[7] darklife. *DarkRISCV*. <https://github.com/darklife/darkriscv>.

[8] Five EmbedDev. *riscv-scratchpad - baremetal-startup-c*. <https://github.com/five-embeddev/riscv-scratchpad/tree/master/baremetal-startup-c/src>.

[9] Jacob Gorban. “Uart ip core specification”. In: *Architecture* 15 (2002), p. 1.

[10] IObundle Lda. *IOb-SoC*. <https://github.com/IObundle/iob-soc>.

[11] LiteX. *linux-on-litex-vexriscv*. <https://github.com/litex-hub/linux-on-litex-vexriscv>.

[12] lowRISC. *Ibex RISC-V Core*. <https://github.com/lowRISC/ibex>.

[13] lowRISC. *OpenTitan - PLIC*. <https://github.com/lowRISC/opentitan/tree/master/hw/ip_templates/rv_plic>.

[14] Martin Odersky et al. *The Scala language specification*. 2004.

[15] C Papon. “SpinalHDL: An alternative hardware description language”. In: *FOSDEM*. 2017.

[16] C Papon. *VexRiscv*. <https://github.com/SpinalHDL/VexRiscv>.

[17] pulp-platform. *RISC-V Platform-Level Interrupt Controller*. <https://github.com/pulp-platform/rv_plic/tree/master>.

[18] riscv. *RISC-V Platform Specification*. <https://github.com/riscv/riscv-platform-specs/blob/main/riscv-platform-spec.pdf>.

[19] RoaLogic. *AHB-Lite Platform-Level Interrupt Controller (PLIC)*. <https://github.com/RoaLogic/plic>.

[20] SiFive. *HiFive Unleashed*. <https://www.sifive.com/boards/hifive-unleashed>.

[21] SiFive. *HiFive Unmatched*. <https://www.sifive.com/boards/hifive-unmatched>.

[22] SiFive. *U54*. <https://www.sifive.com/cores/u54>.

[23] SiFive. *U54-MC*. <https://www.sifive.com/cores/u54-mc>.

[24] SiFive. *U74*. <https://www.sifive.com/cores/u74>.

[25] SiFive. *U74-MC*. <https://www.sifive.com/cores/u74-mc>.

[26] stnolting. *The NEORV32 RISC-V Processor*. <https://github.com/stnolting/neorv32>.

[27] Andes Technology. *ADP-XC7K160/410, FPGA Based Development Platform*. <http://www.andestech.com/en/products-solutions/andeshape-platforms/adp-xc7k160-410/>.

[28] Andes Technology. *AE350 Platform*. <http://www.andestech.com/en/products-solutions/andeshape-platforms/ae350-axi-based-platform-pre-integrated-with-n25f-nx25f-a25-ax25/>.

[29] Andes Technology. *AndesCore™ A25, Compact High-Speed 32-bit CPU for Real-time and Linux Applications*. <http://www.andestech.com/tw/%E7%94%A2%E5%93%81%E8%88%87%E8%A7%A3%E6%B1%BA%E6%96%B9%E6%A1%88/andescore-processors/riscv-a25/>.

[30] Andes Technology. *AndesCore™ AX25, Compact High-Speed 64-bit CPU for Real-time and Linux Applications*. <http://www.andestech.com/tw/%E7%94%A2%E5%93%81%E8%88%87%E8%A7%A3%E6%B1%BA%E6%96%B9%E6%A1%88/andescore-processors/riscv-ax25/>.

[31] Denys Vlasenko. *BusyBox*. <https://git.busybox.net/busybox>.

[32] YosysHQ. *PicoRV32 - A Size-Optimized RISC-V CPU*. 2015. <https://github.com/YosysHQ/picorv32>.

[33] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Oct. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[34] Jerry Zhao et al. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: (May 2020).