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 29, 2022

Abstract

The recent appearance of the *RISC-V* ISA opened many exciting possibilities for building processor-based systems without the need to license the base architecture from providers like ARM. Running applications on bare metal *RISC-V* machines is a good starting point, but an OS is required to ease the developers' efforts for more complex applications. Linux has been around for over three decades and is a well-polished OS. The problem is that open-source SoC platforms that run Linux and simultaneously are modular and configurable do not exist. This work aims to create an SoC capable of executing a Linux OS. The author bases the work on *IOb-SoC*, a modular and configurable open-source SoC platform that only runs bare-metal applications. The author achieves this work's goal by changing the *IOb-SoC* CPU and adding three hardware peripherals. Additionally, the author develops software that improves the *IOb-SoC* platform, complements the hardware components created and allows the execution of a complete OS in the new SoC. Throughout this work, the thesis might refer to the SoC developed as *IOb-SoC-Linux*. The *IOb-SoC-Linux* uses less than 10% of the FPGA resources on the supported development boards. Moreover, the Linux OS boots in five seconds in the Kintex Ultrascale and seven seconds in the Cyclone V.

Keywords: RISC-V, Linux, Systems on-Chip (SoC), Verilog, IOb-SoC

Contents

List of Figures

List of Tables

List of Code

List of Acronyms

A Standard Extension for Atomic Instructions.

ACK Acknowledgement.

ALU Arithmetic Logic Unit.

AMO Atomic Memory Operations.

ASCII American Standard Code for Information Interchange.

ASIC Application-Specific Integrated Circuit.

BRAM Block Random Access Memory.

C Standard Extension for Compressed Instructions.

CGRAS Coarse-Grained Reconfigurable Arrays.

Chisel Constructing Hardware in a Scala Embedded Language.

CISC Complex Instruction Set Computer.

CLINT Core-local Interrupt Controller.

CPU Central Processing Unit.

CSR Control and Status Register.

DC1 Device Control 1.

DDR Double Data Rate.

DEMUX desmultiplexer.

DMA Direct Access Media.

DRAM Dynamic Random Access Memory.

DTB Device Tree Blob.

DTC Device Tree Compiler.

DTS Device Tree Source.

ENQ Enquiry.

EOT End of Transmission.

FCR FIFO Control Register.

FIFO First In, First Out.

List of Acronyms

FPGA field-programmable gate array.

FRX Send a file request.

FTX Receive a file request.

HDL Hardware Description Language.

IER Interrupt Enable Register.

IIR Interrupt Identification Register.

IoT Internet of Things.

ISA Instruction set architecture.

LCR Line Control Register.

LR Load-Reserved.

LSB Less Significant Byte.

lsb Less Significant bit.

LSR Line Status Register.

M Machine.

M Standard Extension for Integer Multiplication and Division.

MCR Modem Control Register.

MMU Memory Management Unit.

MRO Machine Read Only.

MRW Machine Read and Write.

MSB Most Significant Byte.

msb Most Significant bit.

MSR Modem Status Register.

MUX multiplexer.

OoO Out-of-Order.

OS Operating System.

PC Program counter.

PCI Peripheral Component Interconnect.

PLIC Platform-Level Interrupt Controller.

RBR Receiver Buffer Register.

RISC Reduced Instruction Set Computer.

rootfs root file system.

rtc real time clock.

RTL register-transfer level.

RTOS Real-Time Operating Systems.

RV32I 32-bit Base Integer Instruction Set.

S Supervisor.

SBI Supervisor Binary Interface.

SC Store-Conditional.

SEE Supervisor Execution Environment.

SoC System on a chip.

SPI Serial Peripheral Interface.

SRW Supervisor Read and Write.

THR Transmitter Holding Register.

U User/Application.

UART Universal asynchronous receiver/transmitter.

UUT Unit Under Test.

Zicsr Control and Status Register (CSR) Instructions.

1 | Introduction

This chapter talks about what motivates the development of this thesis project and refers to the project's objectives and results delivered. Furthermore, the chapter also covers which hardware and software components the author had to develop to achieve the thesis goal. Lastly, the author will explain the overall composition of this thesis report.

1.1 Motivation

The availability of fully open-source systems capable of executing an Operating System (OS) is limited. For a long time, the Linux kernel [45] and the open-source software built around it allowed developers to implement a fully open-source Linux OS on their closed-source hardware devices. However, the scarcity of open-source hardware complicated the development of fully open-source systems. With the appearance of *RISC-V* [3], open-source hardware availability started growing. Developing a *RISC-V* System on a chip (SoC) capable of executing a Linux OS would allow researchers access a fully open-source system executing an OS. Having a Linux OS running in an SoC would allow developers to create new applications to run in that SoC without worrying about its hardware components. The Linux community is significant, and researchers are used to working with the Linux kernel. Therefore, the requirement for an SoC capable of running Linux is high.

A Linux OS allows using many features unavailable in bare-metal applications. When developers create a bare-metal application, they must be aware of the SoC hardware and are limited in terms of functionalities. Similarly, if developers were to create an application using Real-Time Operating Systems (RTOS), for example, freeRTOS [6], they would have access to features such as a scheduler, events, threads, semaphores and message boxes. However, a Linux OS provides those and more functionalities. A Linux OS implements memory management and protection mechanisms, allows the execution of multiple applications simultaneously, supports multiple network adapters, and can interact with the user through a terminal. A Linux OS is also more secure than bare-metal or RTOS applications since it limits the user application's access to the machine resources, preventing misuse or damage.

What most motivates the development of a *RISC-V* SoC capable of running a Linux OS is its advantages for future development. Such as creating hardware accelerators which work with a Linux OS and integrating them with the SoC the author developed to test in a real-world application.

1.2 Objectives and Deliverables

This dissertation work aims to develop an open-source SoC and execute a minimal Linux OS on it. The author will adapt the existing *IOb-SoC* [18] to create an SoC that supports a Linux OS. *IOb-SoC* is a modular open-source *RISC-V* SoC that allows researchers to develop their own SoC. The *IObundle* developers use *Verilog* [43] to describe *IOb-SoC* and peripherals hardware.

The author had first to swap the current CPU used in *IOb-SoC*. The problem with the current CPU is that it cannot run an OS, only bare-metal applications. Therefore, *IOb-SoC-Linux* contains a 32-bit *RISC-V*

CPU capable of running Linux. Then, since the *IOb-SoC* does not support interrupts, the author had to create and integrate into *IOb-SoC-Linux* the hardware needed to generate interrupts in a *RISC-V* SoC. Lastly, the author had to ensure the Linux OS supported the UART used in the *IOb-SoC-Linux*. Since Linux does not support the *IOb-SoC* UART, the author integrated a *UART16550* in *IOb-SoC-Linux*.

Four major software components comprise a Linux OS. Those software components are the Linux kernel, the bootloader firmware, the root file system (rootfs) and a Device Tree Blob (DTB). The author had to build those software components to run a Linux OS on the *IOb-SoC-Linux*. On power-on, the *IOb-SoC-Linux* transfers the Linux OS software binary files onto the board where it runs, and the Linux OS will boot. After the OS boots, the user can run custom applications and take advantage of the Linux OS. The author also automated and documented the process of generating and deploying the Linux image to *IOb-SoC-Linux* so that, after this work, creating new images with different characteristics will be straightforward.

Finally, the system must be fully verified both on simulation and running on an FPGA board. The *IOb-SoC* needed a fast *Verilog* simulator to verify the Linux OS execution. Therefore, the author developed a simulation testbench using the free-of-charge and open-source *Verilator* [37] simulator.

1.3 Author's Work

The author had to develop four hardware modules to develop a System on a chip (SoC) capable of executing a Linux OS. Those hardware modules allowed the integration of a new CPU, a new UART and the hardware needed to support interrupts in the *IOb-SoC*. Besides integrating new hardware in the *IOb-SoC*, the author made minor changes to the *IOb-SoC* core. The new CPU used generates from the *SpinalHDL* [25] *VexRiscv* [26] platform. The *VexRiscv* platform allowed the author to generate a *VexRiscv* CPU core that meets the requirements to execute an Operating System. Furthermore, the author still had to create a CPU wrapper to adapt the *VexRiscv* CPU to the *IOb-SoC*. Another hardware component the author had to develop was a wrapper for a UART compatible with Linux. The author adapted the wishbone interface of an industry-standard *UART16550* to the *IOb-SoC*. Additionally, the author fully developed the CLINT hardware. The CLINT is a hardware component that generates timer and software-related interrupts for a *RISC-V* system. Another hardware component which manages interrupts generated by other peripherals in a *RISC-V* system is the PLIC. The author adapted an existing PLIC module to the system by creating an interface that worked with *IOb-SoC*.

During this thesis, the author also developed many software components. Those software components were essential to run a Linux OS in *IOb-SoC* or enhance the *IOb-SoC* platform. First, the author translated a program written in *C* programming language, called *Console*, to *Python*. The *IOb-SoC* platform uses the *Console* program to communicate through Serial with the board. Furthermore, the author added features to the *Console* that made it capable of working with the simulator testbench and communicating with a Linux OS running in *IOb-SoC*. Secondly, the author developed a new simulation testbench, based on the previous *IOb-SoC* verification software, capable of communicating with the *Console* program. Moreover, the author integrated the *Verilator* [37] simulation software in *IOb-SoC* and created a *Verilator C++* testbench. Thirdly, the author created a simulation testbench for the CLINT hardware and an interrupt routine firmware that took advantage of the CLINT. The firmware created demonstrates how to use interrupts in *IOb-SoC*. Finally, the author adapted, built and deployed the

software needed to execute a Linux OS in the SoC. The author had to adapt the *IOb-SoC* bootloader firmware. He created a device tree file describing the hardware components of the SoC. He compiled a Linux kernel version compatible with the *VexRiscv* CPU and developed a rootfs adequate for a minimal Linux OS. While developing the hardware and software components, the author also developed Makefile scripts that help integrate the components in *IOb-SoC* and automatise the building and deployment process.

1.4 Thesis Outline

This thesis has five significant chapters. Chapters 2 and 3 address the theme's state-of-the-art. Chapters 4 and 5 discuss the components the author developed to achieve the thesis goal, and chapter 6 shows the results. The thesis's first chapter (i.e. this chapter) is an introduction to the thesis, and the last chapter (7) is the author's conclusions after completing the project.

In chapter 2, the author discusses the tools, concepts and standards the reader must understand to comprehend the following chapters and the decisions the author made while developing the thesis project. Chapter 3 presents the existing technologies capable of executing a full feature Operating System and discusses the possibility of integrating them in *IOb-SoC*. The author described the hardware he had to develop to successfully create an SoC capable of executing a Linux OS in chapter 4. Chapter 5 presents the software components the author had to develop. The software presented in chapter 5 is not only needed to execute the Linux OS in the developed SoC but also improves the *IOb-SoC* platform and tests the hardware the author developed.

Finally, chapter 6 shows the products of this expedition. The author achieved many breakthroughs in various steps of this thesis development. For every breakthrough, the author obtained results that allowed analysing and discussing the hardware and software developed up until then.

2 | Must Have Concepts

During this chapter, the author will discuss topics that help understand the technological developments along this 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 following chapters.

The candidate will develop a SoC in this project. However, he will not create it from scratch. The candidate will use the *IOb-SoC* as a starting point. Consequently, it is important to understand how the *IOb-SoC* works beforehand. It is also indispensable 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 *RISC-V* foundation has created hardware specifications for hardware compatible with *RISC-V* systems which are essential to know. Furthermore, a necessary concept for this project is the boot flow of an Operating System (OS) on a *RISC-V* platform. Lastly, a crucial part when developing a system is its testing and simulation before implementation. Therefore, the author will review the available methods for simulating the developed components.

2.1 The *IOb-SoC* platform

The *IOb-SoC* [18] is a System on a chip (SoC) template that eases the creation of a new SoC. The *IOb-SoC* provides a base Verilog [43] 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, the *IOb-SoC* will include an instruction L1 cache, a data L1 cache and a shared L2 cache. The L2 cache communicates with a third-party memory controller IP (typically a DDR controller) using an AXI4 [44] 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 take time and effort. The *IObundle* developers created the *IOb-SoC* to facilitate this process. This work develops a variant of the existing *IOb-SoC* capable of running a Linux Operating System. *IOb-SoC* currently supports two FPGA board models: the Xilinx Kintex UltraScale KU040 Development Board and the Cyclone V GT FPGA Development Kit.

2.1.1 *IOb-SoC Makefiles*

Makefiles are essential because they allow automating processes. This way, instead of executing multiple 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. For

Figure 2.1: *IOb-SoC* sketch.

example, to run a simulation, a developer using this project SoC would only have to write in the terminal the commands in listing 2.1.

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

Listing 2.1: Run a simulation.

The first command in listing 2.1 cleans all the files related to a previous simulation execution. Then the second command runs a new simulation. This simulation will use the default configurations in the "config.mk" file. A *Makefile* target follows each of the "make" commands. A target is a section of the *Makefile* that may or may not depend on other targets and executes a sequence of commands. In the example 2.1 there are two targets, "sim-clean" and "sim-run".

The main *Makefile* in *IOb-SoC* is located at the *IOb-SoC* root directory. The main *Makefile* contains targets that call other *Makefiles* and sets the values for the default frequency, baud rate, FPGA board used and simulator used. The *Makefiles* the main one can call are at the *IOb-SoC* FPGA boards, simulators, firmware, "PC" emulation or documentation directory. Each directory in *IOb-SoC* contains a "*.mk" file which holds "make" variables and targets that complement the *Makefiles*. The *IOb-SoC Makefiles* can include only the "*.mk" they need.

When executing the command `make sim-run` the computer will run the "run" target of the *Makefile* in the default simulator directory. The simulator *Makefile* will include the "simulator.mk", "hardware.mk" and "config.mk" files. The "simulator.mk" file is common to all simulators. Both FPGAs and simulators *Makefiles* include the "hardware.mk" file. The "hardware.mk" file includes all the hardware modules that the SoC uses. Lastly, the "config.mk" is located at the *IOb-SoC* root directory and all *Makefiles* use it. The "config.mk" defines the "make" variables that are important for hardware and software, for example which peripherals the SoC contains.

2.1.2 *IOb-SoC* peripherals

Developers add the *IOb-SoC* peripherals under the submodules directory in the *IOb-SoC* folder. Inside the submodules directory, there exists a folder for each peripheral. Furthermore, the CPU, the "iob_cache" module, the "iob_mem" hardware, the "iob_axi" interface and the *IOb-Lib* [17] repository can also be found in the submodules directory. The *IObundle* engineers developed the *IOb-Lib* submodule for it to contain small generic hardware modules and software script used in *IOb-SoC*.

All the submodules may contain hardware and software components. For each hardware peripheral, the *IOb-SoC* engineers recommend developing a set of bare-metal firmware functions that allow using the peripheral with the SoC without an OS. Since the *IOb-SoC* platform also allows emulating the developed firmware in the user's personal computer, the peripherals should have software functions that emulate its firmware drivers.

The peripheral should have the following "*.mk" files to integrate it into *IOb-SoC*:

- the "PERIPHERAL_REPO/hardware/hardware.mk" so the user can add the peripheral hardware modules to the SoC.

- the "PERIPHERAL_REPO/software/embedded/embedded.mk" allows the user to use the peripheral firmware drivers.

- the "PERIPHERAL_REPO/software/pc-emul/pc-emul.mk" permits emulating the peripheral behaviour in the user's computer.

The developer has to include the "PERIPHERAL_REPO/hardware/hardware.mk" file in the *IOb-SoC* "hardware.mk" file. He can include it by adding `include $(PERIPHERAL_DIR)/hardware/hardware.mk` to the *IOb-SoC* "hardware.mk". In the firmware *Makefile* the develops has to include the peripherals "embedded.mk" file by adding `include $(PERIPHERAL_DIR)/software/embedded/embedded.mk` to use the peripheral drivers. Lastly in the "pc-emul" *Makefile* he has to add `include $(PERIPHERAL_DIR)/software/pc-emul/pc-emul.mk` to allow the software to emulate the peripheral.

In the "config.mk" file, located in the *IOb-SoC* repository root, the developer needs to add the "PERIPHERAL_REPO" to the "PERIPHERALS" "make" variable and create a variable that indicates the peripheral directory. The user should define the variable that indicates the peripheral directory similarly to `PERIPHERAL_DIR=$(ROOT_DIR)/submodules/PERIPHERAL_REPO`.

The peripheral also needs the following files to be automatically instantiated in the SoC hardware: "inst_tb.vh", "inst.vh" and "pio.vh". The *Makefiles* use the "inst_tb.vh" file to add the needed Verilog to the testbench core for the system to simulate. It uses the "inst.vh" to instantiate the peripheral hardware module in the *IOb-SoC* core. Finally, the "pio.vh" file contains input and output signals that the *Makefiles* must add to the system core hardware. Those files should be in the "PERIPHERAL_REPO/hardware/include" directory. Listing 2.2 presents an example of the "inst.vh" contents.

```
1  //
2  // TIMER
3  //
4  iob_timer timer
5    (
6      .clk      (clk),
7      .rst      (reset),
8
```

```
9     //cpu interface
10    .valid(slaves_req['valid('TIMER)]),
11    .address(slaves_req['address('TIMER,'TIMER_ADDR_W+2)-2]),
12    .wdata(slaves_req['wdata('TIMER)]),
13    .wstrb(slaves_req['wstrb('TIMER)]),
14    .rdata(slaves_resp['rdata('TIMER)]),
15    .ready(slaves_resp['ready('TIMER)])
16    );
```

Listing 2.2: Example of the "inst.vh" file.

2.1.3 Internal Buses

The *IObundle* developers designed the *IOb-SoC* in a way that one master hardware component and multiple slave hardware components exist. To Interconnect the hardware components *IOb-SoC* defines two types of bus, the request bus and the response bus. In *IOb-SoC* the master component is the CPU. The CPU sends requests to the internal or external memory and the peripherals through the request buses. After making a request, the CPU will receive the response through the respective response bus. The CPU instantiated in the *IOb-SoC* core has the "cpu_i_req", "cpu_d_req", "cpu_i_resp" and "cpu_d_resp" signals. The "cpu_i_req" signal serves to fetch instructions from memory, and the "cpu_i_resp" will contain the instruction fetched after a few clock cycles. The CPU uses the "cpu_d_req" to make data requests to either the memory or a SoC peripheral.

The request bus comprises a valid bit, an address signal, a data signal and a strobe signal. The hardware sets the valid bit to '1' when it wants to execute a request and has already defined the other signals. The address signal indicates the register that the request is targeting. Figure 2.2 shows how the *IOb-SoC* distributes the signals in the request bus. Furthermore, figure 2.2 also represents the bits equivalent to each signal when the address width and data width are 32 bits. The address and data width in *IOb-SoC* are 32-bit by default.
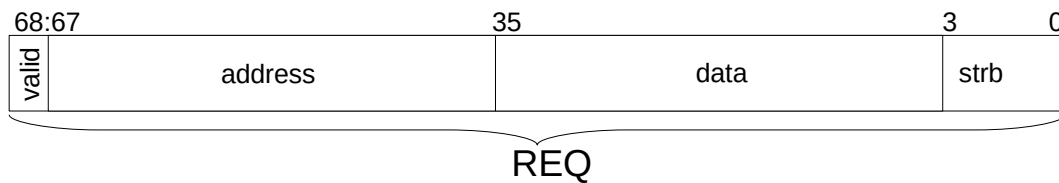
Figure 2.2: Request bus with address and data width equal to 32 bits.

The response bus contains a ready bit and a data signal. The hardware sets the ready signal to high when the component that made the request can receive the response. The data signal is the response data to the request made. For example, if the CPU wants to read the value in a register at address "x", the data in the response bus will be the data on register "x". Figure 2.3 shows how the request signal is composed when the address and data width are 32 bits.

Figure 2.3: Response bus with address and data width equal to 32 bits.

In the *IOb-Lib* submodule exists a file, called "iob_intercon.vh", that defines a set of macros that can be used in *Verilog*. Developers can use those macros to access the specific bits in the *IOb-SoC* buses. Table 2.1 shows the defined macros, how the pre-compiler calculates their value and the respective values when the address and data width are 32-bit.

Name	Defined expression	Description	Value with: ADDR_W=32 DATA_W=32
READY_W	1	Response ready width	1
VALID_W	1	Request valid width	1
WSTRB_W	DATA_W/8	Request data width	4
WRITE_W	DATA_W+DATA_W/8	Most significant bit of request data	36
READ_W	DATA_W+READY_W	Most significant bit of response data	33
WDATA_P	DATA_W/8	Less significant bit of request data	4
ADDR_P	DATA_W+DATA_W/8	Less significant bit of request data	36
VALID_P	ADDR_W+DATA_W+DATA_W/8	Request bus valid bit position	67
REQ_W	VALID_W+ADDR_W+WRITE_W	Request bus width	69
RESP_W	DATA_W+READY_W	Response bus width	33
req(I)	I*REQ_W +: REQ_W	Request bus bits boundary	I*69 +: 69
resp(I)	I*RESP_W +: RESP_W	Response bus bits boundary	I*33 +: 33
valid(I)	I*REQ_W+VALID_P	Request valid bit position	I*69+1
address(I,W)	I*REQ_W+ADDR_P+W-1 -: W	Request address bits boundary	I*69+W+35 -: W
wdata(I)	I*REQ_W+WDATA_P +: DATA_W	Request data bits boundary	I*69+4 +: 32
wstrb(I)	I*REQ_W +: WSTRB_W	Request strb bits boundary	I*69 +: 4
write(I)	I*REQ_W +: WRITE_W	Request write bits boundary	I*69 +: 36
rdata(I)	I*RESP_W+READY_W +: DATA_W	Response data bits boundary	I*33+1 +: 32
ready(I)	I*RESP_W	Response ready bit position	I*33

Table 2.1: Bus interconnect macros.

Understanding the interconnect macros is essential when developing the interface with peripherals and hardware components for the *IOb-SoC*. Some macros depend on an "I" value given to them when using the macro. The hardware uses the "I" value to distinguish which request or response signal the developers want to access when there is a bus (i.e. wire) with multiple requests or responses. In the "address(I,W)" macro, the "W" value corresponds to the number of bits of the address in the request signal the developer wants to select.

2.1.4 *iob-split* and *iob-merge*

The *iob-split* and the *iob-merge* hardware modules can both be found in the *IOb-Lib* submodule hardware directory. The *IOb-SoC* uses the *iob-split* in the systems core and in the internal memory hardware modules. The *iob-split* in *IOb-SoC* separates one request and one response bus into multiple buses and only enables the needed one. For example, there are multiple peripherals; each has an input request bus and an output response bus. However, there is only one "cpu_d_req". The *iob-split* only sends the "cpu_d_req" signal to the selected peripheral and sets the other peripherals request bus to zeros. The *IOb-SoC* instantiates the *iob-merge* in the external and internal memory hardware modules. The *iob-merge* unifies multiple request and response buses into one. For example, the CPU can execute both instructions and data requests to the memory. Nevertheless, the memory can only process one request at a time. The *iob-merge* merges the "cpu_i_req" and the "cpu_i_req" and sends to the memory the request with higher priority or the one that was first set has valid.

The **iob-split** is simply a configurable desmultiplexer (DEMUX). The developer can configure it when he instantiates the *iob-split* hardware module. The developer can 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. Developers can also interpret N_SLAVES as the number of the DEMUX outputs. P_SLAVES indicates the slave select word Most Significant bit (msb) position. In other words, P_SLAVES is the position of the msb of the desmultiplexer selection bits. Equation 2.1 calculates the number of the selection bits.

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

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. N_SLAVES indicates the number of inputs, and P_SLAVES chooses the selection bits.

2.1.5 Bootloader

The *IObundle* engineers developed a bootloader for *IOb-SoC* that is the first firmware to run on the System on a chip. The *IOb-SoC* always saves the bootloader firmware in the SoC internal memory in a boot control hardware unit. The boot control hardware unit defines the boot signal. The boot signal is one bit that can be set to high ('1') or low ('0') and indicates whether the SoC should execute the bootloader or other firmware. Moreover, the boot control unit sends a reset signal to the CPU when the bootloader ends before starting the users' firmware.

Figure 2.4 represents a flow chart of the bootloader firmware behaviour. The bootloader starts by initialising the UART hardware which the SoC uses to communicate with the user's computer. Then it will send an Enquiry (ENQ) if it still has not sent one. The ENQ byte has the value of 0x05 in ASCII and is sent to the user's computer to indicate that the *IOb-SoC* is waiting for the user's computer. The bootloader will ensure the UART is sending the ENQ byte while the user's computer does not send a response. After receiving a response byte, the bootloader firmware checks if the received byte is an

Send a file request (FRX). The FRX Byte has the value 0x07. If the bootloader receives a FRX, it has to transfer the firmware that runs after it.
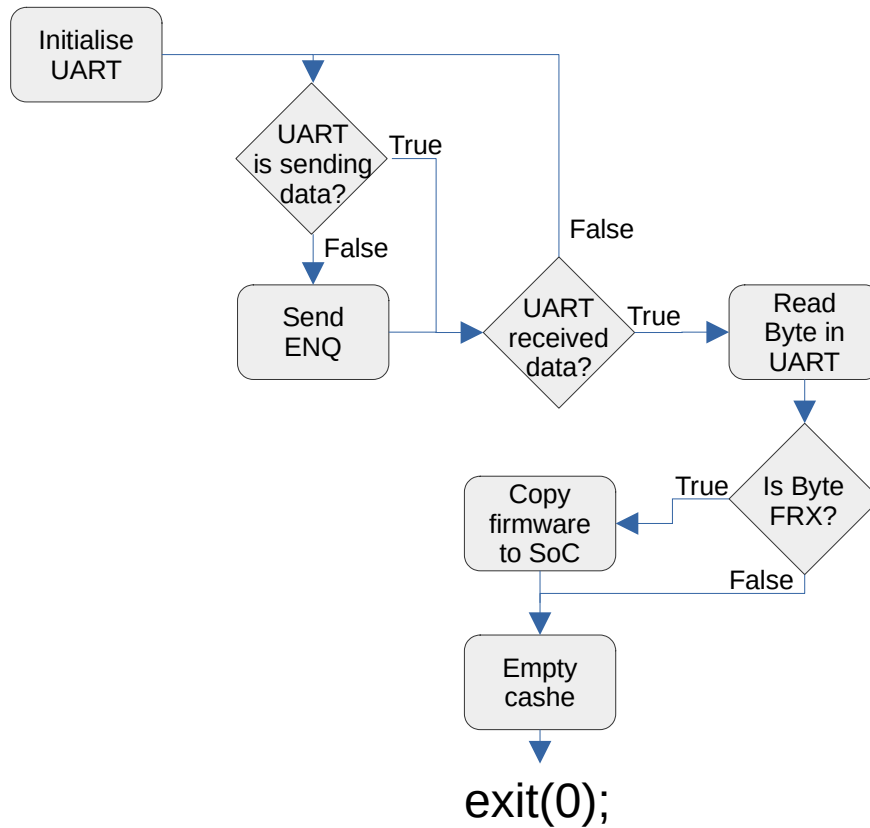
Figure 2.4: Bootloader firmware flow chart.

When transferring the firmware, the bootloader receives the binary data through the UART and copies it to the SoC memory. The bootloader can copy the firmware to either the internal or external memory depending on where the user defined the program would be stored. After copying the firmware to the *IOb-SoC*, the bootloader will send the data saved in the memory back to the user's computer. The user's computer can then check if the *IOb-SoC* successfully transferred the firmware by comparing the original firmware binary with the received copy. Before exiting, the bootloader will clean the cache so that the next firmware does not fetch data previously in the cache. Finally, when exiting, the bootloader asks the boot control unit to set the boot signal to low and reset the CPU.

2.2 *RISC-V*

RISC-V [3] is a free-to-use, open-source RISC Instruction set architecture (ISA). The *RISC-V* Instruction set architecture defines the instructions which a *RISC-V* compatible CPU can interpret. Those instructions represent the software written in C, Python, or any other programming language to be executed by the CPU.

Developers need the *RISC-V* toolchain to cross-compile software for *RISC-V* platforms. The *RISC-V* toolchain has two main types of cross-compiler options, the Newlib and the Linux version. Furthermore, developers can install the toolchain to work with 32-bit, 64-bit or both system architectures. The author installed the Newlib, and the Linux *RISC-V* toolchain with "multilib" support (i.e. support for both 32-bit and 64-bit architecture). The *IOb-SoC* uses the Newlib cross-compiler to compile firmware to run in bare-metal. Developers must use the Linux cross-compiler to compile the Linux kernel and software related to the OS.

The *RISC-V* architecture is considered a RISC architecture. The Reduced Instruction Set Computer (RISC) architecture is one possible classification of the computer ISAs, and the essence of this approach is the simplicity of the instructions computed by the CPU. Another type of ISA is the Complex Instruction Set Computer (CISC) architecture. The RISC architecture instructions are simpler then the CISC architecture. Therefore, RISC CPUs need to execute more instruction then the CISC CPUs to achieve the same result. Nevertheless, the RISC architecture compensates by occupying less memory and allowing systems based on it to run at a higher frequency.

The organisation that maintains the *RISC-V* specifications is the *RISC-V* International. With the adoption of the *RISC-V* Instruction set architecture in hardware, *RISC-V* has growed. The growth of *RISC-V* leads to the appearance of standard compatible hardware specifications, for example, the hardware components that generate interrupts in a SoC. Furthermore, there are multiple platform requirements described in the *RISC-V* Platform Specification [29] for *RISC-V* compatible systems. Additionally, *RISC-V* has also set requirements for OSs that call themself *RISC-V* compatible.

In *RISC-V* a Hart is a CPU hardware thread. In simple language, a Hart is a *RISC-V* execution context that contains a complete set of *RISC-V* architectural registers and that executes its program independently from other Harts in a *RISC-V* system. A complete set of *RISC-V* architectural registers consists of thirty-one general-purpose registers (x1-x31), which hold integer values and an additional register (x0) that the system hardwires to the constant 0. Those registers' width equals the architecture bit width, which could be 32-bit, 64-bit or 128-bit. The table A.1 in annex A.1 presents the 32 registers a *RISC-V* Hart has. Additionally, a user-visible program counter register holds the address of the current instruction.

2.2.1 Instruction Set Architecture

A group of researchers initially designed the *RISC-V* ISA at the University of California, Berkeley [3], defining instructions for 32, 64, and 128-bit systems architecture. In this thesis, the author focuses on 32-bit architecture since the hardware the author used as a starting point follows a 32-bit architecture. One advantage of the *RISC-V* ISA is that it is flexible, and engineers designed it to be modular. Therefore, developers can use only certain parts of the *RISC-V* ISA in their applications. When a developer needs to use an instruction that is not on the standard *RISC-V* ISA, he can add the instruction through an ISA extension. Additionally, developers can create custom instructions and add them to their system.

The *RISC-V* ISA is divided in two main volumes. The "RISC-V Instruction Set Manual Volume I" [48] contains the specification for the **unprivileged** instructions. The unprivileged instructions are instructions

that do not need any special permission to execute. The "RISC-V Instruction Set Manual Volume II" [49] defines the *RISC-V* **privilege** levels and the instructions that take advantage of them.

The **unprivileged** instructions manual explains the *RISC-V* 32-bit Base Integer Instruction Set (RV32I), the ISA extensions which do not require special privileges, the *RISC-V* terminologies and other features that a *RISC-V* system should follow. This thesis project, besides the RV32I, takes advantage of the Standard Extension for Integer Multiplication and Division (M), the Standard Extension for Atomic Instructions (A), the Standard Extension for Compressed Instructions (C) and the Control and Status Register (CSR) Instructions (Zicsr). The RV32I defines forty basic instructions and is capable of emulating almost any ISA extension except the "A" extension. The "M" extension contains four instructions, two that multiply and the other two divide the values held in two integer registers. The "A" extension contains instructions that atomically read-modify-write memory to support synchronisation between multiple RISC-V harts running in the same memory space. In the "A" extension there are a total of nine instructions. There is the Load-Reserved (LR) instruction, the Store-Conditional (SC) instruction and seven Atomic Memory Operations (AMO) instructions. The "C" reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The "C" extension is essential for embedded systems. When using the "C" extension, normally 50%-60% of the RISC-V instructions in a program can be replaced with compressed instructions, which reduces the program's memory footprint by 25%-30%. Lastly, *RISC-V* defines a separate address space of 4096 Control and Status Register (CSR) associated with each Hart. The "Zicsr" extension defines the 6 CSR instructions that operate on these CSRs.

The **privileged** instructions manual covers all aspects of *RISC-V* systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running Operating Systems and attaching external devices. Table 2.2 shows the privilege levels currently defined in the *RISC-V* specification. Privilege levels protect different software stack components, and attempts to perform operations not permitted by the current privilege mode will raise an exception. These exceptions normally cause the system to trap and move to an underlying execution environment. The Machine (M) privilege level is the only mandatory level for all *RISC-V* systems. Software that runs in M-mode has access to the entire system and must be trusted by the developer. The Supervisor (S) privilege level (S-mode) is intended to run Operating Systems. Developers should use the User/Application (U) privilege level to execute common user applications.

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

Table 2.2: *RISC-V* privilege levels.

Developers must implement all three privilege levels to run a Unix-like Operating System. Each privilege level brings additional instructions and Control and Status Registers to the system. Most of the CSRs used by the system are irrelevant to this thesis project. Table A.2 and A.3 in annex A.2 show the CSRs the author thinks are essential to know of. From those registers it is crucial to understand how the "mstatus", "misa", "mip", "mie", "mtvec" and "mcause" work to debug the SoC the author developed in this thesis.

The "mstatus" register keeps track of and controls Hart's current operating state. Through the "mstatus" register, developers can enable or disable global interrupts in their system. The "misa" register contains information about the *RISC-V* ISA and extension the system uses. The two Most Significant bit (msb) of the "misa" register represent the architecture width (in case of 32-bit they should be '01') and the sixteen Less Significant bit (lsb) represent the ISAextensions implemented. The "mip" and "mie" registers represents which interrupts are pending and enabled respectively on a given *RISC-V* Hart. The "mtvec" register holds trap vector configuration (i.e. how the system should handle traps). The two lsb of the "mtvec" register set the trap vector mode which could be either direct ('00') or vectored ('01'). The rest of the "mtvec" register set the trap vector base address. If the trap vector mode is direct when a trap occurs, the Program counter (PC) will jump to the trap vector base address. However, if the trap vector mode is vectored, the Program counter (PC) will jump to the trap vector base address plus 4*cause. When a trap occurs in machine mode, the "mcause" register will hold a code indicating the event that caused the trap.

2.2.2 Compatible CLINT

The *RISC-V* CLINT specification [47] describes the hardware registers of a Core-local Interrupt Controller (CLINT) compatible with *RISC-V* platforms. The hardware uses the CLINT to generate the inter-processor software and timer interrupts. *SiFive* developed their Core-Local Interrupt (CLINT) device before the *RISC-V* foundation made the CLINT specification. Therefore, the RISC-V community has widely adopted it in their projects.

Table 2.3 shows the CLINT registers compatible with the *SiFive* CLINT and the *RISC-V* specification. The CLINT implemented in this thesis project only uses machine-level interrupts. However, the *RISC-V* CLINT specification also specifies the supervisor inter-processor software interrupts, which the author does not approach here.

Name	Address	Width (bit)	Access	Description
MSIP0	0x0	32	RW	HART index 0 machine-level IPI register
MSIP1	0x4	32	RW	HART index 1 machine-level IPI register
...	...	...	...	...
RESERVED	0x3FFC	32	-	Reserved for future use.
MTIMECMP0	0x4000	64	RW	HART index 0 machine-level time compare
MTIMECMP1	0x4000	64	RW	HART index 1 machine-level time compare
...	...	...	...	...
MTIMECMP4094	0x7FF0	64	RW	HART index 4094 machine-level time compare
MTIME	0xbff8	64	RW	Machine-level time counter

Table 2.3: CLINT registers compatible with *SiFive* and the *RISC-V* specification.

Each *RISC-V* Hart should be connected to an "msip" and "mtip" pin in the CLINT hardware unit. Through those pins, the CLINT will notify the CPU when software or timer interrupt occurs. Each "msip" pin is hardwired to the lsb of the corresponding "MSIP" register. The value stored in the "MSIP" register must be '0' after the hardware resets the CLINT. The only way the "MSIP" register value changes is if the

CPU writes to it. The CPU is only able to write either '0' or '1' to the "MSIP" register. The 31 msb of the "MSIP" can be hardwired to '0' and only the lsb changes value.

The "mtip" pins are connected to the output of a digital comparator with the respective "MTIMECMP" register and the "MTIME" register as input. A digital comparator is a hardware component that takes two numbers as input in binary form and determines whether one number is greater than, less than or equal to the other. The "mtip" triggers when the "MTIMECMP" register value is lower then the "MTIME" register. On CLINT reset, the hardware has to set the "MTIME" register value to '0', and the "MTIMECMP" register to a value that does not trigger the timer interrupt, for example, all bits set to '1'. Moreover, the CLINT timer counter resolution must be 100ns or less (corresponding to a clock tick frequency of at least 10 MHz).

2.2.3 Compatible PLIC

The *RISC-V* systems use the Platform-Level Interrupt Controller (PLIC) hardware to gather various device interrupts and have only one external interrupt line per *RISC-V* Hart context. A PLIC that claims to be a PLIC-Compliant standard PLIC has to follow the *RISC-V* PLIC specification [11]. The *RISC-V* PLIC was first described in the privilege instructions documentation. Since version 1.10, the *RISC-V* foundation has moved the PLIC specification to its documentation. However, nothing major has changed, and PLIC cores compatible with the PLIC described in the privilege documentation are still compatible with the current standard.

The PLIC hardware has many features that the author will not discuss here. In this thesis project, the author adapted an already developed PLIC core to his work. To integrate an existing PLIC core, what is essential to understand is how the system handles interrupt notifications and the PLIC memory-mapped registers. Furthermore, each interrupts pin of a hardware peripheral connected to the PLIC unit corresponds to a source. Additionally, a *RISC-V* Hart connected to the PLIC has to have at least one external interrupt pin. The PLIC considers each external interrupt pin as a target. The PLIC notifies the CPU of an existing interrupt through the target signal connected to the CPU external interrupt pin. A *RISC-V* Hart can have one external interrupt input for Machine level interrupts and another for Supervisor level interrupts.

Figure 2.5 was obtained from the *RISC-V* PLIC specification and shows a sketch of the interrupt flow between the PLIC hardware and the CPU. When a peripheral generates an interrupt, the corresponding source input in the PLIC is enabled, and the PLIC forwards it to the interrupt gateway that processes the interrupt signal from each source. Then the interrupt gateway sends a single interrupt request to the PLIC core, which latches these in the core interrupt pending bits (IP). Afterwards, the PLIC core will send a notification to the targets to whom the interrupt interests. When a target receives an interrupt notification, it sends a claim request to the PLIC core and waits for the response, which contains the interrupt id. After sending the identifier of the highest priority global interrupt source pending for that target, the PLIC core clears the corresponding interrupt source pending bit. The target will handle the interrupt through firmware, and when the firmware disables the interrupt, the target sends an interrupt completion message to the associated interrupt gateway. Finally, the interrupt gateway can forward another interrupt request for the same source to the PLIC core.

Figure 2.5: PLIC Interrupt Flow.

Table 2.4 shows the memory mapped registers that the *RISC-V* PLIC specification defines. The PLIC unit maps most registers into consecutive memory positions depending on their type. Every register in the PLIC hardware has a width of 32-bit. Nevertheless, each claim/complete and priority threshold register has an address offset of 0x1000 from the previous register, as shown in table 2.4. The PLIC core can have up to 1023 interrupt sources and 15871 targets. Consequently, there are a maximum of 1023 interrupt source priority registers; 32 interrupt pending bit registers; 507872 enable bits registers, and 15871 claim/complete and priority threshold registers. For each context, there are 32 enable bits registers. The interrupt source priority registers store the priority value of each PLIC source. The interrupt pending bit registers indicates if there exists a pending interrupt request from the respective PLIC sources. The hardware uses the enable bits registers to know for which targets an interrupt from a certain source is enabled. If a context has a priority threshold, an interrupt with a lower priority than the threshold will not be sent to the respective target. The claim/complete registers are used in the interrupt flow seen in figure 2.5.

Lastly, it is essential to note that the PLIC hardware only needs to generate the needed registers. The needed registers depend on the number of sources and the number of targets the PLIC hardware has to support. Those values can be passed as parameters to the PLIC unit when the developer instantiates it in the Hardware Description Language. Consequently, if the number of sources is 32, the PLIC hardware should only create 32 interrupt source priority registers, not 1023. The PLIC parameters can influence

---

[1] write-any-read-legal fields store only legal values written by CSR writes.

Address	Width (bit)	Access	Description
0x0	32	-	Reserved (interrupt source 0 does not exist)
0x4	32	WARL [1]	Interrupt source 1 priority
...	...	...	...
0xFFC	32	WARL	Interrupt source 1023 priority
0x1000	32	R	Interrupt Pending bit 0-31
...	...	...	...
0x107C	32	R	Interrupt Pending bit 992-1023
0x2000	32	RW	Enable bits for sources 0-31 on context 0
...	...	...	...
0x207C	32	RW	Enable bits for sources 992-1023 on context 0
...	...	...	...
0x1F1FFC	32	RW	Enable bits for sources 992-1023 on context 15871
0x200000	32	WARL	Priority threshold for context 0
0x200004	32	RW	Claim/complete for context 0
...	...	...	...
0x201000	32	WARL	Priority threshold for context 1
0x201004	32	RW	Claim/complete for context 1
...	...	...	...
0x3FFF000	32	WARL	Priority threshold for context 15871
0x3FFF004	32	RW	Claim/complete for context 15871

Table 2.4: PLIC registers compatible *RISC-V* specification.

the number of resources the hardware utilises.

2.2.4 Compatible UART/Serial Console

In the *RISC-V* Platform Specification [29] 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 *UART16550*. The *UART16550* already existed for a long time. *National Semiconductor* released it in 1987, and the *UART16550* is present and supported by many software and hardware components. Developers often use the *UART16550* connected to an RS-232 interface. The *IOb-SoC* supported development boards use an RS-232 connector to communicate with the user's computer.

Table 2.5 presents the *UART16550* registers that are always accessible after the SoC resets. The *UART16550* contains more registers which the author discusses further ahead. The registers in table 2.5 may have the same address if the CPU can only read to one register and only write to the other. The *UART16550* only uses the 5 Less Significant bit of a 32-bit address to address its registers.

The RBR and the THR are the *UART16550* registers the SoC uses to send and receive data through the serial interface. The hardware uses the IER and the IIR to enable and identify the different types of interrupts the *UART16550* hardware can generate. The interrupts generated by the *UART16550* are

Name	Address	Width (bit)	Access	Description
Receiver Buffer Register	0	8	R	Receiver FIFO output
Transmitter Holding Register	0	8	W	Transmit FIFO input
Interrupt Enable Register	1	8	RW	Enable/Mask interrupts generated by the UART
Interrupt Identification Register	2	8	R	Get interrupt information
FIFO Control Register	2	8	W	Control FIFO options
Line Control Register	3	8	RW	Control connection
Modem Control Register	4	8	W	Controls modem
Line Status Register	5	8	R	Status information
Modem Status Register	6	8	R	Modem Status

Table 2.5: Assessable *UART16550* registers by default.

forward to a PLIC unit. The hardware can use the FCR to control the receiver and transmitter FIFOs. The LCR is able alter the *UART16550* serial interface configuration and the LSR indicates the current status of the serial connection. Lastly, the MCR allows transferring control signals to a modem connected to the UART and the MSR displays the current state of the modem control lines.

Additionally, there exist two clock divisor registers that together form a 16-bit register. Table 2.6 shows their properties. The *UART16550* uses the divisor latches register to adapt the reading and writing cycles depending on the system clock frequency and the system baud rate. When initialising the UART in firmware the value stored in the registers should be $\frac{systemFrequency}{16*BaudRate}$. On the SoC reset, both registers have all bits set to '0', and the IO interface is disabled. After CPU writes to the divisor latch, LSB, the *UART16550* enables the IO interface. Therefore developers must set the MSB first. Since the divisor latches address is the same as the receiver and transmitter registers to access the divisor latches, developers have to set the 7th bit of LCR to '1'. After setting the divisor latches, the bit should be restored to '0' to restore access to the receiver and transmitter registers.

Name	Address	Width (bit)	Access	Description
Divisor Latch Byte 1 (LSB)	0	8	RW	The Less Significant Byte of the divisor latch
Divisor Latch Byte 2 (MSB)	1	8	RW	The Most Significant Byte of the divisor latch

Table 2.6: Divisor latch *UART16550* registers.

Finally, when using a 32-bit data bus interface, two additional debugging registers exist. The first register address is 0x8, and the second is 0xC. Both registers are 8 bits and read-only. Some of the registers presented in table 2.5 are not relevant to the development of this thesis project. Nevertheless, the reader can study more about them in the documentation of the *UART16550* [16].

2.2.5 Supervisor Binary Interface

The Supervisor Binary Interface (SBI) interface permits supervisor-mode (S-mode or VS-mode) software to be portable across all RISC-V implementations. The Supervisor Binary Interface (SBI) speci-

fication [12] defines an abstraction for platform-specific functionalities. A software application that implements the Supervisor Binary Interface (SBI) specification is known as an SBI implementation or Supervisor Execution Environment (SEE). The SBI creates an interface between the Kernel running in S-Mode and the platform-specific firmware running in M-Mode. The platform-specific firmware can be an Supervisor Execution Environment. Figure 2.6 illustrates the purpose of the SBI in a system executing an OS like the one the author is going to develop in this thesis.

Figure 2.6: *RISC-V* system running an Operating System.

OpenSBI, also known as the *RISC-V* Open Source Supervisor Binary Interface, is the recommended interface between a platform-specific firmware running in M-mode and a general-purpose OS executing in S-mode. The main component of *OpenSBI* is provided in the form of a platform-independent static library "libsbi.a" implementing the SBI interface. *OpenSBI* can be used as a bootloader that is platform-dependent and uses the "libsbi.a" to ensure conformance with the SBI interface specifications. The *OpenSBI* platform-dependent firmware can be configured to suit a specific System on a chip hardware. The SBI specification version supported by the *OpenSBI* firmware is v0.2.

2.2.6 The Linux Boot Flow

Figure 2.7 shows the various stages a *RISC-V* system has to pass through to fully boot a Linux Operating System. The SoC executes the stage 0 bootloader write after the system resets. The stage 0 bootloader will load the runtime firmware and the kernel software to the Double Data Rate (DDR) memory. The runtime firmware in a *RISC-V* system implements the SBI interface. Therefore, developers can use the *OpenSBI* firmware in this stage. In a minimal operating system, the Linux kernel can be loaded right after *OpenSBI*. Otherwise, when building a more complex system where the SoC can load the Linux kernel from flash memory or a network device, developers should use a bootloader like *U-Boot* between the *OpenSBI* firmware and the Linux kernel.

The *OpenSBI* firmware has limited knowledge of the SoC specific hardware. Furthermore, researchers developed the Linux kernel software so that it is platform-independent. Consequently, there needs to exist a way of passing the SoC hardware configuration to the software. For that purpose, developers have to create a Device Tree Source (DTS) file and compile the *OpenSBI* software so that it knows the
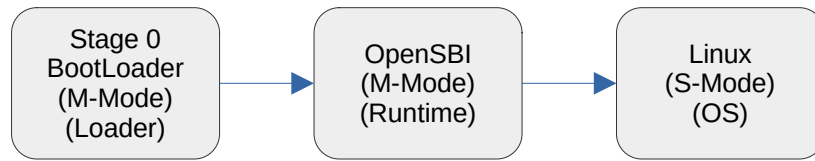
Figure 2.7: Stages of the Linux boot on *RISC-V* on a minimal system.

DTS memory location. The Device Tree Source is a file that describes the hardware in a SoC through a device tree. A device tree is a tree data structure with nodes describing a system's hardware. Each node has property/value pairs that describe the hardware's characteristics. Each node has exactly one parent except for the root node, which has no parent. There are various nodes available to describe the hardware in the device tree. Developers can find all information about the device tree on its specification documentation [14]. The device tree describes information about the components in a SoC that the system can not dynamically discover. For example, the device tree should not describe the devices attached to a Peripheral Component Interconnect (PCI) since the PCI architecture allows the OS to detect and identify the devices. Nevertheless, the device tree should include a device node describing the PCI interface controller in the SoC.

2.3 Open Source Verification tools

Verification tools are essential when developing hardware or software components. Verification tools allow developers to simulate their work before implementing it in real hardware and test new features in a safe environment where the SoC implementation does not use hardware components. In this thesis project, the author has to simulate hardware logic components and platform-independent software. For that purpose there are three types of verification software that the author is going to use: a **functional** emulator, a **cycle-accurate** simulator and an **event-driven** simulator.

All components in this thesis project are open-source. The tools the author uses should be open-source, and the simulator software is no exception. The advantage of open-source technologies is that they are free to use, and researchers can create new things based on them.

2.3.1 Hardware Logic Simulators

It is important to have a good hardware simulation environment for testing purposes. Researchers take advantage of already existing and well-developed tools. Several hardware simulation tools exist, most of which are proprietary, for example, *xcelium* from *Candence*. The utilisation of proprietary software can increase the cost of a project significantly. In this thesis the author will take advantage of ***Icarus Verilog*** [50] and ***Verilator*** [37]. Although both tools are for verification, they serve different purposes due to their characteristics.

Developers can use cycle-accurate and event-driven simulators to simulate the hardware logic designs. A **cycle-accurate** simulator evaluates the state of the logic components in a hardware design every cycle. Cycle-accurate simulators only have two states for the logic signal, either '1' or '0'. Furthermore, cycle-accurate simulators do not consider the propagation times and delays of the hardware logic gates. This feature makes the cycle-accurate simulators fast but not very accurate. When an uninitialised signal exists in the cycle-accurate simulation, the simulator assumes the values are '1' or '0'. Letting an uninitialised signal propagate in the SoC could lead to bugs that pass unnoticed in the cycle-accurate simulation. Cycle-accurate simulators are suitable for complex hardware designs. An example of a cycle-accurate simulator would be *Verilator*.

An **event-driven** simulator propagates a signal through the hardware logic gates until the signal reaches a steady state. Event-driven simulators are more accurate than cycle-accurate simulators and permit estimation if the circuit obeys the timing requirements since it evaluates the state of the logic components each time an event causes an alteration to that component. Moreover, event-driven simulators allow signals to have more than two states, and they can evaluate the signals as '0', '1', undefined, unknown or high-impedance. The disadvantage of event-driven simulators is that they are much slower than cycle-accurate simulators. Event-driven simulators are adequate for small hardware designs. An example of an event-driven simulator would be *Icarus Verilog*.

- *Icarus Verilog* is a Logic Simulator that uses Verilog or System-Verilog testbench 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 translating the hardware logic design to software, *IVerilog* outputs a script which can be run line by line to simulate the designed logic.

- *Verilator* transforms the *Verilog* HDL designs into a *C++* program that can be included in a test-bench. Using *C++* to create a testbench allows calling the converted hardware program as a function of a *C++* library. After compiling the *C++* testbench, developers can execute it and simulate the hardware initially described in *Verilog*. While also allowing to make use of system calls easily.

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 in the design, but it is better to use *Verilator* for more complex simulations.

2.3.2 Software Simulator

A **functional** emulator translates the instructions that were supposed to run on the target architecture to instructions that run on the host CPU. The advantage of using a functional emulator is that it is way faster than the other emulation types. A functional emulator runs 100 million to > 1 billion instructions per second, while a cycle-accurate run 10 to 100 thousand instructions per second. An example of a functional emulator would be **QEMU** [8].

QEMU is an open-source machine emulator and virtualiser that allows running software and firmware, like operating systems, built for different devices and architecture on a personal computer. *QEMU* is

similar to *VMware* [10] or *VirtualBox* [24]. *VMware* and *VirtualBox* allow the user to create virtual machines that simulate real hardware and execute an operating system. Developers can use *QEMU* to run an operating system on virtual hardware.

QEMU can be used to emulate both 32-bit and 64-bit *RISC-V* CPUs. *QEMU* is ideal for testing user applications written to execute on a *RISC-V* embedded system. User applications, contrary to the initial bootloaders, are platform-independent. In case of an application not running on a certain SoC, developers can use *QEMU* to check if the problem is the application or the SoC. A user can emulate an operating system by compiling the firmware to be compatible with the *QEMU* "virt" board. The "virt" is a board that does not replicate any actual hardware. However, the *QEMU* developers designed this virtual hardware so the users could test the software even if there existed no hardware that could run it. To define which board the emulation is supposed to run in when calling *QEMU*, the user should pass *–machine virt* as an argument.

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. However, the processor architecture of those microcontrollers is not open-source, much less the microcontroller itself.

For example, the *Raspberry Pi 4* is a competent 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. Nevertheless, if someone wanted to use the Raspberry as a base for his costume hardware design, that would be impossible. Thus, the need for open-source hardware appears, allowing the creation of something new without starting from scratch every time. The need for open-source hardware led to the appearance of the *RISC-V* 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 is not free to use, it is 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 great 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 developers use 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* [7]. The company priced the board 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 more significant impact on *RISC-V* CPU design: Andes Technology and SiFive.

---

[1] Microchip 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. Maintaining and contributing to *RISC-V* is essential 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 everywhere, from telecommunications, storage controllers, and touchscreen sensors to data centres and more advanced computing. 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* [41] and *AX25* [42]. 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. The different architecture 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 [40] SoC allowing to use these CPUs on developer boards, for example in the *ADP-XC7K160/410* [39].

3.1.2 SiFive

SiFive is a company that was born from the *RISC-V* ISA. Three researchers from the University of California Berkeley founded SiFive, Krste Asanović, Yunsup Lee, and Andrew Waterman. Those researchers were deeply involved in developing the *RISC-V* ISA, from working on the base ISA to 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. The chip release happened in 2016, one year after the researchers founded the company.

In 2017 SiFive launched *U54* [33], which was the first *RISC-V* CPU capable of running a full fledge Operating System like Linux. With it they launched the *U54-MC* [34] 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* [31], which was the first board that implemented the *U54* CPU and ran a Linux OS with a desktop environment (DE). SiFive has discontinued *HiFive Unleashed*, and better hardware has been made available.

SiFive has 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* [35]. 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***). Developers have already applied this CPU to multiple boards. For example, the *BeagleV*

has a SoC with dual-core SiFive *U74* CPU. SiFive as also launched its own development board, *Hi-Five Unmatched* [32], with four *U74* cores on the *U74-MC* [36] SoC. Furthermore, in 2021, Canonical, the developers behind Ubuntu, announced the OS support for 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 are fully open-source, and developers might implement them in other projects. University research groups or individuals with a grant developed most of the open-source CPUs.

RISC-V CPUs are most popular in embedded systems and IoT devices. Consequently, engineers developed many open-source CPUs that other developers can implement in embedded microcontrollers. A few examples of those CPUs would be the *PicoRV32* [51], *NEORV32* [38], *DarkRISCV* [13] and *Ibex* [21] from *lowRISC*. However, this paper will not discuss those in detail 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.

An application processor is needed to run a Linux-based Operating System. A CPU is considered an *application processor* if it has the hardware required to run a full-feature Operating System (OS) and user applications. Having the required hardware means that the processor should have the necessary Control and Status Register (CSR), support M+S+U privilege modes, and support atomic instructions. An open-source solution would be either the *CVA6* [52] (previously known as Ariane), *BOOM* [53] or *VexRiscv* [26].

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 *OpenHW Group* has incorporated the development and maintenance of CVA6 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/`.

The CVA6 supports any operating system based on Unix since it implements the three needed privilege levels M, S and U. The researchers wrote the core in SystemVerilog and designed its micro-architecture to reduce the critical path length. Since the developers wrote it in SystemVerilog, it is easier for someone knowledgeable in the classic Verilog and VHDL languages to understand and create a customised CPU based on the CVA6 than if they had written it 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 difficulty is a consequence of how the engineers developed it. Every SystemVerilog
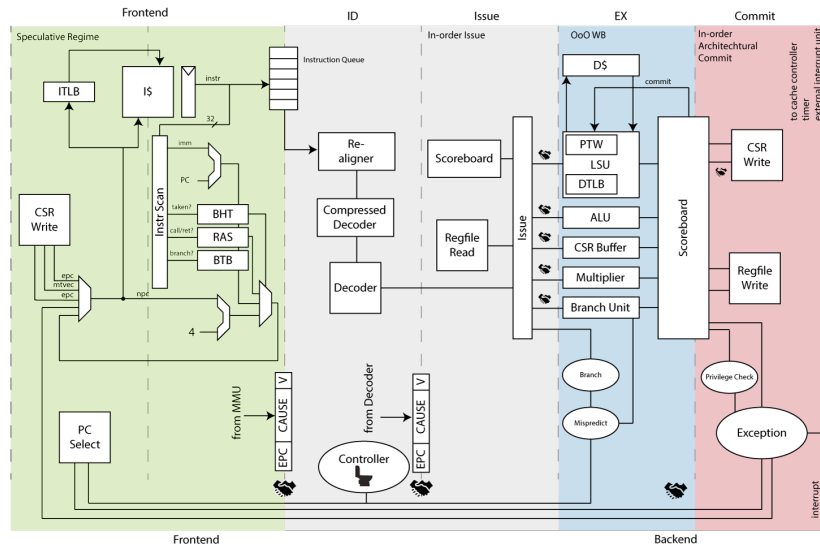
Figure 3.1: CVA6 core design architecture.

module of the CVA6 depends on other files from the project, and the CPU itself is very little customisable. 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 challenging and time-consuming to create a CPU core without that component.

The CVA6 can be found implemented on *OpenPiton* [5]. *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* [53]) is a superscalar Out-of-Order (OoO) processor executing the RV64GC variant of the *RISC-V* ISA. Researchers created BOOM in the Berkeley Architecture Research group at the University of California, Berkeley. The CPU design is optimised to run on ASICs, although developers can also implement it on FPGAs. Its priority is to be a high-performance, synthesisable, and parameterisable 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: Fetch, Decode/Rename, Rename/Dispatch, Issue/Register Read, Execute, Memory and Writeback. Since committing happens asynchronously, it does not count as part of the "pipeline". The BOOM developers optimised the load-store unit for the superscalar out-of-order architecture and organised the data cache into two dual-ported banks. At the front end, it is possible to customise the size of the L1 Instruction cache, the TLB, and the decode stage. Similarly to the CVA6, removing the cache from the core design and using the IOb-Cache is difficult or impossible.

This CPU design is written in Chisel [4] Hardware Description Language (HDL). The Constructing Hardware in a Scala Embedded Language (Chisel) allows for the production of synthesisable Verilog designs while using a high-level language to describe the hardware. Chisel is an adaptation of Scala [23] programing language, adding hardware construction primitives.

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

3.2.3 *VexRiscv*

The *VexRiscv* [26] CPU is a 32-bit Linux Capable *RISC-V* CPU written in the *SpinalHDL* [25]. The *VexRiscv* author accomplished the hardware description of this CPU by utilising 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 parametrised CPU design. When the user configures the CPU without plugins, it only includes the description of the five "pipeline" stages, their basic functionalities, and nothing else. Developers must add everything else 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 how the user configures it. Contrary to *BOOM*, *VexRiscv* does not need any external library. Not using an external library makes it easier to generate the synthesisable Verilog file from a *SpinalHDL* design.

There exists an open-source project that runs Linux with *VexRiscv*, *linux-on-litex-vexriscv* [20]. *LiteX* is used to create a System on a chip (SoC) around the *VexRiscv* core. *LiteX* SoC design and peripherals are written in *Migen* [9] another high level HDL. *Migen* unlike *SpinalHDL* and Chisel is based on Python 3.5. Because of the language describing its hardware and the way the *linux-on-litex-vexriscv* project is structured, it is tough 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, making 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. The reader can observe 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. All CPUs have an L1 cache because it makes it easier to support atomic instructions.

GNU/Linux is the combinations of *GNU* with the Linux kernel. The GNU Project developed a large part of the software comprising 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 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, the reader 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 2.1.2. 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 author had to swap the UART since no compatible Linux drivers worked with *iob-UART*. Besides changing a few components from the chip, he had to add new hardware. The new hardware is the CLINT and the PLIC, both compatible with *RISC-V* specifications. The author created the CLINT to support timer and software interrupts on the SoC. He added the PLIC 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, the reader can see that the candidate removed the L1 Cach. 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). Having three branches meant there were two selection bits; when '00' then, the internal memory bus was active; when '01', it was the peripheral bus; when '10', it was the external memory. Using two bits for selection 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. Not changing the selection bit 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.

4.1 Central Processing Unit

The CPU chosen to use in this project was *VexRiscv* [26]. The performance of the CPU is not a significant issue for this project. However, how the developers designed the core 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 developers can use configurations directly or configure them to generate a custom *VexRiscv* core. There even already exists a demo configuration to generate a Linux-compatible core. Although the author implemented a custom core, he used the Linux demo configuration 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.

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

Listing 4.1: Generate *verilog* from *SpinalHDL*

Developers can configure the *VexRiscv* 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. The available plugins show developers two different plugins for the instruction bus and another two for the 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. However, knowing how to write in *SpinalHDL* would be necessary. Learning how to code in *SpinalHDL* would be very time-consuming and not in this project's scope.

The first step on implementing the *VexRiscv* core on the *IOb-SoC* was making sure that it worked on "bare metal" applications. The developed SoC had to work with the application accessing the silicon chip directly without any intermediary like an Operating System (OS). The author first developed the SoC for "bare metal" applications 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. The signal waves created during the simulation made it possible to identify which instruction was causing the problem.

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, developers could define the address of the first instruction the CPU had to fetch by setting the reset value of the Program counter (PC). In this plugin, developers could also specify if the CPU had a branch predictor and if it supported compressed instructions.

- 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" determines where the core 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 project, the author used it to add only division since another plugin added multiplication support.

- The "CsrPlugin" is configured to support Linux fully. This plugins adds the needed Control and Status Register (CSR) to run a full feature OS.

- The "BranchPlugin" allows the core to execute and make decisions on the jump instructions. Jump instructions are 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.

For this project, the author decided not to use any branch predictor since the performance of the SoC was not a concern. Furthermore, there seemed to be a compatibility problem between the most recent *RISC-V* toolchain and the branch predictors available in the *VexRiscv*. Additional, he decided that CPU should support compressed instructions because compressed instructions allowed to reduce the firmware size considerably.

The author could have used some additional plugins. The "DebugPlugin" could be used to debug the CPU core if a JTAG interface existed. However, the *IOb-SoC* does not support it. The "FpuPlugin" can add support for both the floats and doubles instructions. 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 only to be adding unnecessary hardware logic.

After generating the Verilog file that describes a *VexRiscv* core, the author 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 author created an interface between the *IOb-SoC* hardware and the *VexRiscv* core by establishing a connection between the inputs and outputs from both sides.

The *iob_vexriscv* has multiple input signals. The clock signal is the system clock derivative from the development board where the users implement the SoC. The reset signal is set to high ('1') when the system reboots or the stage 0 bootloader finishes. The boot signal has the value '1' while the stage 0 bootloader is executing. After it finishes, the boot signal value drops to '0' while the hardware sets the reset signal to high. The instruction bus response signal is connected to "cpu_i_resp". The data bus response signal is connected to "cpu_d_resp". The timer interrupt and software interrupt signals are '1' or '0' depending on the outputs from the CLINT unit. Lastly, the PLIC unit controls the external interrupt input signal. 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 correspond to the request and response buses reviewed in section 2.1.3.

The input and output signals of the *VexRiscv* core can be seen in table 4.1. The reader can also see 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. However, 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 author could obtain the "strb" signal 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

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.

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.

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

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.

$$strb = dbus\_req\_wr?dbus\_req\_mask : 4'h0$$
$$= dbus\_req\_wr?dBus\_cmd\_payload\_mask : 4'h0$$

(4.2)

The second is that the *IOb-SoC* internal bus did not contain all the signals needed by the *VexRiscv* core. To successfully make the interface handshake with the *VexRiscv* core, the candidate had to generate an instruction and data request "ready" signal. 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 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. The candidate obtained this truth table by analysing the simulation signals wave. The "N/A" values in the table mean that those situations never occurred.

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.

Engineers can transform the truth table into a logic gates expression, and the reader can see the expression in equation 4.3.

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

(4.3)

However, this approach had an issue. The logic expression depended on the "cmd_valid" signal which was generated inside the *VexRiscv* core. Depending on the "cmd_valid" signal 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.

Engineers can see the truth table 4.3 as a simple XOR logic gate. The equation 4.4 represents the hardware logic expression implement.

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

(4.4)

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.

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. The author solved this problem 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.

Finally, the CPU should be able to run firmware from both the internal and external memory. When the stage 0 bootloader is running, the hardware has to force the Most Significant bit (msb) of the instruction fetch address to '0'. Having the msb set to '0' forces the CPU to fetch instructions from the boot hardware unit. When the user defines the firmware to run from the external memory ("RUN_EXTMEM=1"), the first instruction fetched should be at address $0x80000000$. When "RUN_EXTMEM=1", the wrapper defined the Most Significant bit (msb) as the negated value of the boot signal. When "RUN_EXTMEM=0", the Verilog forced the msb to 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 the Linux kernel supports. The other option was to create a Linux driver compatible with *iob-UART* and compile it with the kernel. The chosen approach seemed an adequate and more straightforward 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* [16] that has been made available by *freecores* on *github*. The developers of the *UART16550* wrote it 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 the author did with the CPU, he had to create a wrapper to adapt the core to the *IOb-SoC*.

The *UART16550* developers defined the wishbone interface in the top hardware module of their UART 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*.

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.

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 signals that the wishbone interface requires 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 should exist during write and read operations. The hardware can obtain this signal through the address signal's two Less Significant bit (lsb). Furthermore, since registers are addressed byte by byte, only one bit at a time will be set to high on the select signal. When the CPU wants to write to the *UART16550* registers, the hardware has to set the write signal high. 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. the wrapper should set the "WB_STB_I" signal should '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. The FPGA connects those pins to the development board RS232 connector.

4.4 CLINT Unit

The CLINT was the only hardware component 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 that only needs a few registers and signals to work. The author would have to create an interface with *IOb-SoC* independently of the CLINT core used. The best solution was to entirely create the CLINT hardware unit.

Table 4.6 shows the memory mapped register and their respective address implemented on the developed CLINT hardware. The CLINT registers are compatible with both the *SiFive* and the *RISC-V* specification. Since the SoC developed only has one Hart (i.e. CPU core), the Verilog code only creates one register of each type, three registers in total. Nevertheless, the register's address must be separate because developers could use the CLINT created by the author in more complex systems. Furthermore, the logic behaviour of the CLINT hardware is identical to the behaviour described in the *RISC-V* specification reviewed in section 2.2.2.

Name	Address	Width (bit)	Access	Description
MSIP	0x0	32	RW	Software interrupt
MTIMECMP	0x4000	64	RW	Register to compare with timer register
MTIME	0x1bff8	64	RW	Timer register

Table 4.6: CLINT Registers.

The inputs and outputs of the CLINT unit can be seen in table 4.7. "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. Nevertheless, the author built the CLINT considering a possibility of a multi-core system. Each core has its timer and software interrupt. As such, the "mtip" and "msip" signals width is the number of core, "N_CORES". When the CLINT hardware receives a request from the CPU, the response is ready one clock cycle after. The response data will be the data stored in the register designated by the request address signal. If the request strobe signal is different from 0, then the CLINT writes to the register the request addresses. The value stored in that register will be the same as the request data signal.

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 real time clock (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 hardware detected the rising edge by taking samples of the "rt_clk" signal at the system clock frequency. Taking those samples is possible because the rtc is slower than the system clock. Suppose in the future, the CLINT unit developed is implemented in a system with a rtc. In that

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 4.7: CLINT interface with *IOb-SoC*.

case, developers can easily adapt the hardware design to support the rtc since the author developed the logic.

The development boards used in this thesis do not have an rtc, so the author had to find an alternative. The method implemented was more straightforward than the logic if a rtc existed. The system clock operates at 100 MHz, so the author added a counter to the design that increases the timer register when the counter reaches a defined threshold. The CLINT timer frequency the author defined is 100KHz and the Verilog code calculates the counter threshold with $\frac{SystemFrequency}{100000} - 1$. Therefore, if the system frequency is 100MHz, the hardware simulates a rtc working at 100 kHz by incrementing the timer when the counter reaches 999.

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 SoC uses the PLIC to drive interrupts generated by other peripherals to the CPU, in this project, the only peripheral connected to the PLIC is the *UART16550*. For the growth of the SoC, the PLIC unit is a handy requirement. Developers in the future can add some peripherals that 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* [22] 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* [28], 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* [30] 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 customise that influence the hardware. The number of interrupt sources defines how many peripheral interrupt signals the developer can connect to the PLIC. In this project, the number of interrupt sources is one (the *UART16550*). The number of interrupt targets normally equals 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 interrupts 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. A parameter also defines if a 'threshold' is implemented. The last configurable parameter determines whether the PLIC implements the 'configuration' register. 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 4.8. The value of "SOURCES" is 8. The author decided It should 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 one 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 4.8: 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 a 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

During this project, software applications were created simultaneously with the development of hardware logic components. 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 automate the 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. In figure 5.1 the reader can see the *Console* program flowchart.

Figure 5.1: *Console* program flowchart.

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.

```
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 5.2.

Figure 5.2: 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 real time clock derived from the board, the real time clock 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 real time clock, 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 real time clock 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        char wstrb_int = 0;
3        switch (nbytes) {
4            case 1:  wstrb_int = 0b01; break;
5            case 2:  wstrb_int = 0b011; break;
6            default: wstrb_int = 0b01111; break;
7        }
8        dut->uart_addr = cpu_address >> 2; // 32 bit address (ignore 2 LSBs)
9        dut->uart_valid = 1;
10       dut->uart_wstrb = wstrb_int << (cpu_address & 0b011);
11       dut->uart_wdata = cpu_data << ((cpu_address & 0b011)*8); // align data to 32 bits
12       Timer(CLK_PERIOD);
13       dut->uart_wstrb = 0;
14       dut->uart_valid = 0;
15   }
```

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* [15], 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 bootloader firmware programs. The first firmware to execute is the stage 0 bootloader. The stage 0 bootloader , which the author refers to as *iob-bootloader*, is an adaptation of the bootloader firmware used in *IOb-SoC*. The second bootloader is a *RISC-V* specific software called *OpenSBI*.

The *iob-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 *iob-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 5.15.

```
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 5.15 "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 listing A.1 of annex A.3.

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 compile the Linux kernel automatically. Listing 5.17 shows the "build-linux-kernel" target. The author also created a configuration file adapted to the SoC. The configuration file allows the author to inform the Linux compilation of the *RISC-V* ISA supported by the SoC developed, the SoC support for external interrupts and the loading of a *RISC-V* SBI interface before the kernel. 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.

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

Listing 5.17: Linux Kernel Makefile target.

An alternative way to build the Linux kernel would be using *Buildroot*. *Buildroot* [27] 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* [46]. *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'
9   echo 'OpenCryptoLinux >   '
10  echo -e "\nThis boot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
11
12  /bin/sh
```

Listing 5.18: "init" script.

The author developed a build-root file system Makefile target to automate 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.

```
1   build-rootfs: clean-rootfs
2       cd $(busybox_DIR)/busybox && \
3           cp $(Custom_Platform_DIR)/rootfs_busybox/busybox_config $(busybox_DIR)/busybox/
        configs/iob_defconfig && \
4           $(MAKE) ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- iob_defconfig && \
5           CROSS_COMPILE=riscv64-unknown-linux-gnu- $(MAKE) -j$(nproc) && \
6           CROSS_COMPILE=riscv64-unknown-linux-gnu- $(MAKE) install && \
7           cd _install/ && cp $(Custom_Platform_DIR)/rootfs_busybox/init init && \
8           mkdir -p dev && sudo mknod dev/console c 5 1 && sudo mknod dev/ram0 b 1 0 && \
9           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.

```
1   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.

```
1    if select.select([stdin, ], [], [], 0.0)[0]:
2        user_str = stdin.read(1)
3        if(user_str!=''):
4            if SerialFlag:
5                ser.write(bytes(user_str, 'UTF-8'))
6            else:
7                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.

```
1    #!/usr/bin/python
2    import sys
3    import termios
4
5    stdin = sys.stdin
6    fd = stdin.fileno()
7
8    old = termios.tcgetattr(fd)
9    new = termios.tcgetattr(fd)
10   new[3] &= ~termios.ECHO
11   new[3] &= ~termios.ICANON
12
13   termios.tcsetattr(fd, termios.TCSAFLUSH, new)
14   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

In the following chapter, the author will analyse the results obtained from the hardware and software developed in this thesis project. The author's first objective was running the "Hello World!" firmware with the *VexRiscv* CPU. Secondly, he tested the implementation of the interrupt routine software with the developed CLINT hardware. Finally, the candidate successfully executed the minimal Linux OS in real hardware using the developed System on a chip.

All the results obtained in this thesis which communicate with the FPGA board or the SoC testbench, are executing the developed *Console* program. The hardware components comprising the SoC differ in each section of this chapter. The author customises the SoC hardware depending on the software needs. Along this chapter, the developed SoC will be referenced as the *IOb-SoC-Linux*.

In each step, the author studied the simulation with the different logic simulators and the memory resources needed to run the respective firmware. Furthermore, when running the SoC on the FPGA board he examined the required FPGA resources.

6.1 System Running "Hello World!"

The *IObundle* developers created the "Hello World!" firmware to test the functionality of the *IOb-SoC* template. After the author implemented the *VexRiscv* CPU on the developed SoC, he executed a regression test to verify the correctness of the SoC. The regression test was the execution of the "Hello World!", which was known to work correctly on the *IOb-SoC*.

The "Hello World!" firmware is a program that prints a "Hello World!" message to the user, prints the value of π, which is a floating number, and tests file transferring between the *Console* and *IOb-SoC*. The alterations the author made to the *IOb-SoC* hardware to obtain the results presented in this section was swapping the CPU and removing the *IOb-SoC* external memory L1 cache.

The firmware size dictates the minimal size of the memory on the SoC. The *IOb-SoC-Linux* memory is 32 KB because the "Hello World!" program size is 23964 Bytes. The memory size should always be the closest upper bound power of two.

6.1.1 Execute in simulation

The author simulated the "Hello World!" program using the *Icarus* simulator and *Verilator*. The "Hello World!" simulation allows the author to make a fair comparison between both logic simulators.

In figure 6.1, the reader can see the expected output when executing the "Hello World!" firmware. In this example, the simulator executed the firmware using the internal memory of the SoC and considered that the firmware was already on the memory. Considering the firmware was already on the memory, allow the simulator not to execute the firmware transfer between the *Console* and the testbench.

(a) Start of the "Hello World!" firmware.

(b) End of the "Hello World!" firmware.

Figure 6.1: Running the "Hello World!" firmware in simulation.

Both open-source logic simulators are capable of executing the "Hello World!" program. The simulation is slower when the baud rate decreases or the memory size increases. The baud rate used in the simulation was 5000000. The baud rate is the number of bits per second transferred to the UART. The author executed the simulations considering the system clock frequency 100MHz. The SoC always stores the bootloader in internal memory. The bootloader memory is 4KB (2^12). Since the bootloader binary is 1508 Bytes, 4KB is enough memory to store the bootloader program. The firmware, by default, is stored in the internal memory, and the memory size is 32KB (2^15). In table 6.1 the reader can see a timing comparison between the different logic simulators simulating the original *IOb-SoC* and the developed SoC. The "INIT_MEM" flag indicates whether the firmware is already loaded in the FPGA or if the *Console* needs to transfer the firmware to the SoC, the user can set the flag to '1' or '0' respectively. The users can execute the simulations with or without external memory. Furthermore, the firmware can run in internal or external memory. The "make sim-test" command tests the different possible simulations.

	IOb-SoC-Linux		*IOb-SoC*	
Command \Simulator	Icarus	Verilator	Icarus	Verilator
make sim-run INIT_MEM=1	2m 26s	0m 3s	0m 25s	0m 3s
make sim-run INIT_MEM=0	88m 19s	1m 1s	15m 18s	0m 27s
make sim-test	231m 3s	2m 27s	43m 34s	1m 34s

Table 6.1: Timing the "Hello World!" firmware simulation.

From table 6.1 engineers are able to conclude the advantage of using *Verilator*. For more complexed systems the *C++* testbench is much faster than the *Verilog* counterpart. The disadvantage of using *Verilator* is that signal values can only be either '0' or '1'. However, the speed-up in the simulation is also due to the signal value limitation. In *Icarus*, the simulation can evaluate the signal as unknown ('x') when they are uninitialised. The author noted that *Verilator* is slower to compile the testbench. However, it is much faster to execute the software. The *IOb-SoC* simulation is faster then the authors SoC simulation

because the *PicoRV32* is less complex then the *VexRiscv* CPU.

6.1.2 Execute in the FPGA Board

In figure 6.2 the readers can see the output of executing the "Hello World!" firmware in the FPGA. The author synthesised the SoC with the external memory. Furthermore, the firmware is running from external memory.

(a) Start of the "Hello World!" firmware.

(b) End of the "Hello World!" firmware.
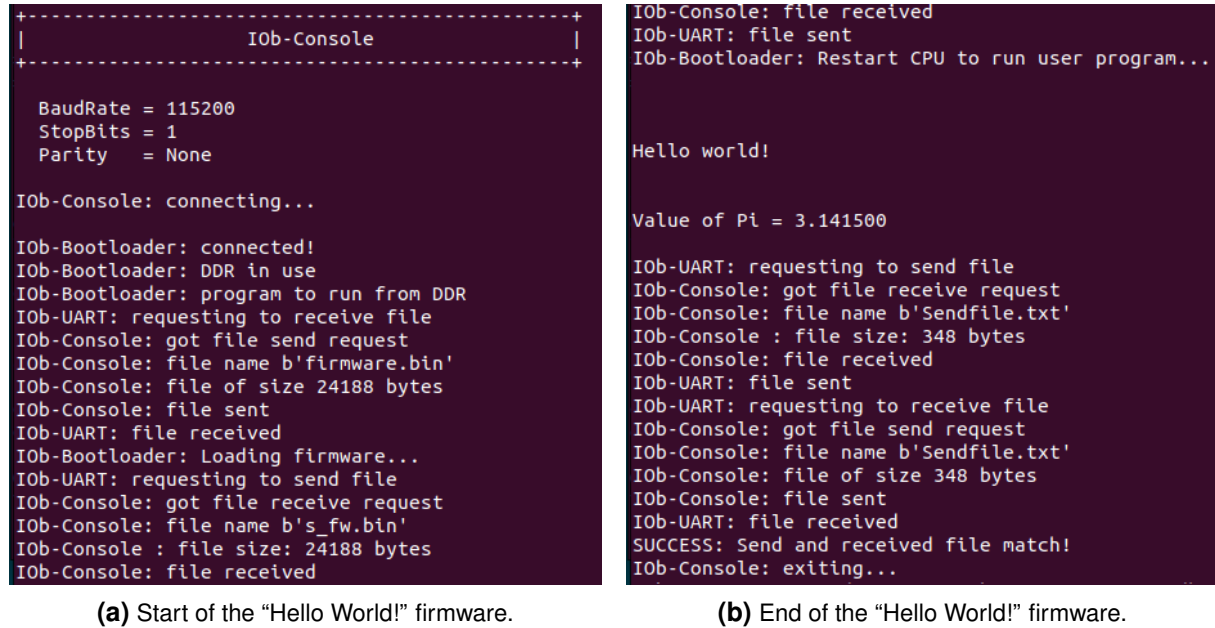
Figure 6.2: Running the "Hello World!" firmware in the FPGA Board.

The tables in 6.2 are the FPGA implementation results for two FPGA families. The author implemented the developed SoC on the kintex Ultrascale AES-KU040-DB-G board and in the CYCLONE V GT-DK. The kintex Ultrascale has an FPGA more capable than the CYCLONE V.

	IOb-SoC-Linux	*IOb-SoC*
ALM	10,062	9,280
FF	12150	10020
DSP	8	3
BRAM blocks	234	352
BRAM bits	753,248	779,744

(a) Cyclone V GT

	IOb-SoC-Linux	*IOb-SoC*
LUTs	21226	23003
Registers	23373	22588
DSPs	10	7
BRAM [1]	39.5	34.5

(b) Kintex Ultrascale

Table 6.2: FPGA results for "Hello World!" program using external memory.

The author obtained the values in table 6.2 while using the SoC with the external memory. When synthesising the SoC, the user can define whether he wants to use external memory with the "RUN_EXTMEM" flag. If "RUN_EXTMEM=1" then a memory controller will be synthesised alongside the developed SoC

---

[1]The exact number of BRAM bits used in the Kintex Ultrascale is unknown. However, each BRAM in it has 36K bits.

and loaded onto the FPGA. The memory controller is hardware logic written in Verilog and specific to the FPGA where the SoC is running. In order to better understand resource utilisation, the author decided to compare the resources used when running the "Hello World!" firmware from the internal memory. The resources without the memory controller are in tables 6.3.

	IOb-SoC-Linux	*IOb-SoC*
ALM	3,687	1,542
FF	4707	1214
DSP	8	3
BRAM blocks	56	38
BRAM bits	408,800	296,960

(a) Cyclone V GT

	IOb-SoC-Linux	*IOb-SoC*
LUTs	5457	2072
Registers	4405	1074
DSPs	7	4
BRAM	14	9

(b) Kintex Ultrascale

Table 6.3: FPGA results for "Hello World!" program.

From the tables in 6.3 the author can confirm the *VexRiscv* CPU requires more resources than the *PicoRV32*. Comparing tables 6.2 and 6.3 the reader can see that there is a big difference in resources due to the utilization of the external memory. Interestingly, in table 6.2a the *IOb-SoC-Linux* uses less BRAMs than the *IOb-SoC*. The *IOb-SoC-Linux* uses less BRAMs because the L1 cache integrated with the *VexRiscv* CPU is smaller than the L1 *iob-cache* hardware module used by the *IOb-SoC*. In table 6.3 the *IOb-SoC-Linux* clearly uses more resources because the *VexRiscv* CPU contains the L1 cache and is more complex than the *PicoRV32*. *VexRiscv* is more complex than *PicoRV32* because it supports more instruction extensions and has more Control and Status Register.

6.2 Interrupt Routines
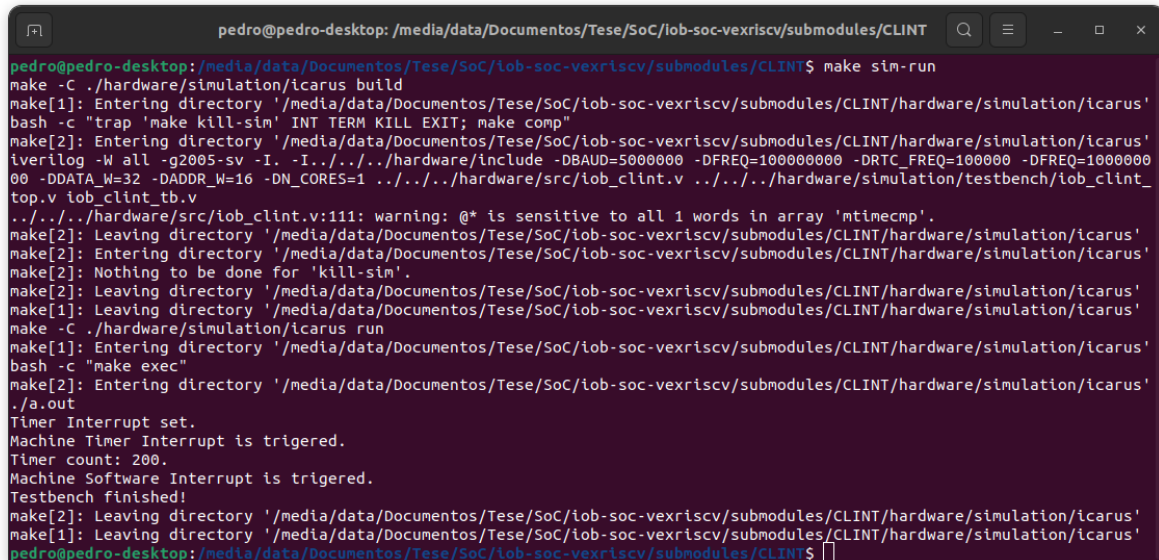
To test the correct functionality of the interrupts in the SoC, the author executed the developed CLINT testbench. Moreover, to test the complete SoC, he ran the bare-metal firmware created to handle interrupts. The firmware was executed in simulation and implemented in the FPGA Board.

The size of the firmware that tests the interrupt routine is 24364 Bytes. Consequently, the only difference in the SoC used on this section tests is the addition of the CLINT hardware. The memory size is the same since the "Hello World!" program and this firmware have similar sizes, under 32 KB.

6.2.1 Execute CLINT simulation

After developing the CLINT unit, the author executed its testbench, testing the timer and software interrupts. Figure 6.3 shows the complete process when running a simulation. The author executed the simulation with *Icarus Verilog*. However, the process is similar with *Verilator*. First, the Makefile script has to compile the testbench software. The `make sim-run` calls the "build" target of the Makefile in the simulator directory. The Makefile calls the "build" target in figure 6.3 with the command `make -C ./ hardware/simulator/icarus build`. The build target executes the "iverilog" application which compiles the

hardware sources and the testbench into an executable program. Secondly the Makefile calls the "run" target. The "run" target will call the software application that runs the testbench. Figure 6.3 shows the executable being called with `./a.out`. After that, the testbench will run until it finishes or the user kills the process.

Figure 6.3: CLINT timer and software interrupt simulation.

The author previously discussed the testbench executed in figure 6.3 in subsection 5.3.1. The testbench first sets the timer interrupt to trigger when the "MTIMER" register counter hits 200. The testbench disables the timer interrupt after the CLINT notifies its occurrence and then triggers the software interrupt. In the clock cycle, right after triggering the software interrupt, the testbench will receive the notification from the CLINT unit and disable the interrupt. The testbench finishes after enough time have passed to allow the interrupts to happen if there is nothing wrong with the hardware. Since the testbench and hardware are , *Icarus* is better then *Verilator* to simulate the CLINT.

6.2.2 Execute in simulation

The author executed the firmware discussed in subsection 5.3.2 on simulation with both *Icarus* and *Verilator*. The figures in 6.4 show the execution of the firmware. The *IOb-SoC-Linux* implemented does not use external memory.

The author wrote the firmware to set the timer interrupt differently for the simulation with *Icarus* and the simulation with *Verilator*. Since the simulation with *Icarus* is slower than with *Verilator*, the time the firmware is waiting for the interrupt to trigger does not have to be as big for the user to notice it while simulating. The *Icarus* simulation set the timer interrupt to trigger after 0.001 seconds, while the *Verilator* simulation set to 0.01 seconds. When the interrupt handler executes, the firmware prints the current time and the value in the "MTIMER" register. If 0.001 seconds had passed the value of the "MTIMER" register should be $100KHz * 0.001s = 100$. The "MTIMER" register value is not the exact time the interrupt takes to trigger because it also counts the time it takes to execute the rest of the firmware.

(a) Interrupt routine firmware with *Icarus*.

(b) Interrupt routine firmware with *Verilator*.

Figure 6.4: Running the interrupt routine firmware in the FPGA Board.

6.2.3 Execute in the FPGA Board

Lastly, the author implemented the *IOb-SoC-Linux* in the FPGA boards. Figure 6.5 shows the correct execution of the firmware from the *IOb-SoC-Linux* internal memory. The firmware programs the timer interrupt to trigger one second after the firmware starts.

Figure 6.5: Executing the interrupt routine program on the FPGA.

In figure 6.5, the interrupt handler prints the current time that has passed since the firmware started. The hardware consumed approximately one second waiting for the interrupt to trigger. The firmware uses the extra 0.004 seconds to print the "Hello World" message at the start of the firmware and to execute the interrupt handler. The extra time consumed when executing the *IOb-SoC-Linux* in hardware differs from the simulation's extra time. This difference is due to the baud rate used. Since the hardware baud rate is lower than the simulation baud rate, the messages theoretically take more time to print to the

terminal.

Tables 6.4a and 6.4b represent how much FPGA resources are consumed by the *IOb-SoC-Linux*. The author executed the firmware from the external memory and the internal memory. The tables in 6.4 show the resources when the SoC contains an external memory (w/ DDR) and when it does not (w/o DDR).

	w/o DDR	w/ DDR
ALM	3,883	10,257
FF	4940	12300
DSP	8	8
BRAM blocks	56	234
BRAM bits	408,800	753,248

(a) Cyclone V GT

	w/o DDR	w/ DDR
LUTs	5729	21478
Registers	4580	23545
DSPs	7	10
BRAM	14	39.5

(b) Kintex Ultrascale

Table 6.4: FPGA results for interrupt routine program.

Comparing the table in 6.4 with the tables in 6.3 and 6.2 engineers can see that the CLINT hardware does not use much resources.

6.3 Boot and use the Linux Operating System

The objective of this thesis project was to run an Operating System in the *IOb-SoC-Linux*. The *IOb-SoC-Linux* used in this section adds the PLIC hardware and substitutes the *iob-UART* for the *iob-UART16550*. The software that comprises the complete OS is the *OpenSBI* bootloader, the Device Tree Blob, the Linux kernel and the root file system.

Table 6.5 presents how much time it takes to build the complete OS with the command `make build-OS`. The "real" time is the time that passes since the user executes the command until it finishes. The "user" time is the time the CPU takes while executing operations in the user space. The "user" time is bigger than the "real" time because it counts the time passed in each CPU core. Part of the compilation of the rootfs and the kernel is done in parallel using two cores.

real	4m30s
user	8m12s
sys	0m57s

Table 6.5: Time it takes to build the OS.

The OS size is to big to run in the FPGA internal memory. The *OpenSBI* bootloader is 90896 Bytes. The Device Tree Blob is 1669 Bytes. The Linux kernel is 4426152 Bytes. Lastly the root file system is 1142733 Bytes. The memory has to have at least 8 MB ($2^2 3$) to store all this software. However, the Linux kernel needs a bigger memory where it can store virtual memory pages and execute the different application processes. The Device Tree Source describes the system had 512 MB of available memory. Consequently, the author had to implement the *IOb-SoC-Linux* on the FPGA with access to the external memory. The internal memory could never be as big as 512 MB.

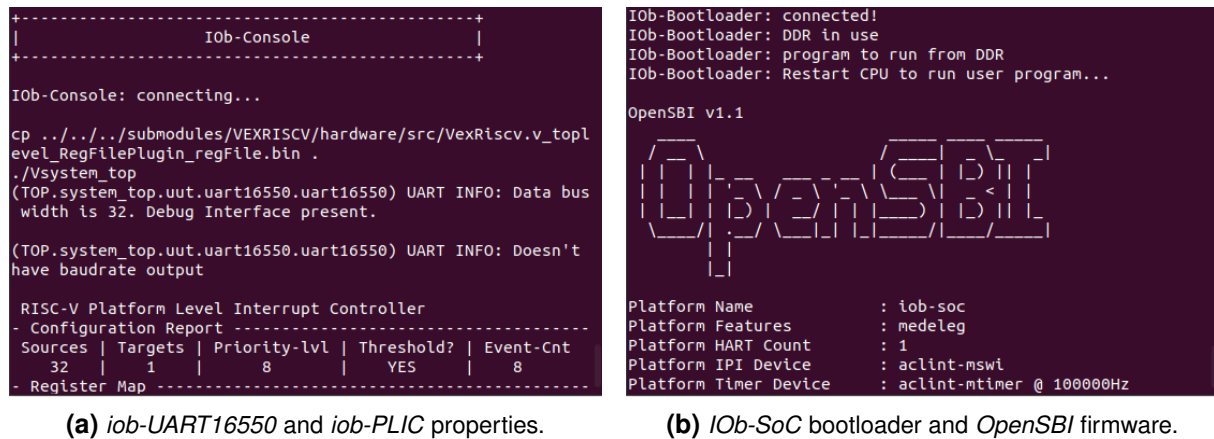In figure 6.6 the reader can see the start of the OS simulation with *Verilator*.

(a) *iob-UART16550* and *iob-PLIC* properties.

(b) *IOb-SoC* bootloader and *OpenSBI* firmware.

Figure 6.6: Start of the OS simulation with *Verilator*.

Figure 6.6a shows the initialization of the *Console* program. Furthermore, it shows the instantiation of the *iob-UART16550* and the *iob-PLIC*. The *iob-UART16550* and the PLIC core have an initial block that prints their properties. The synthesis tools do not synthesise the initial block to real hardware, but the simulator executes it. Figure 6.6b shows the *iob-bootloader* and the start of the *OpenSBI* bootloader. The *iob-bootloader* in figure 6.6b does not transfer the software to the memory because the author executed the simulation considering that the software was already in the memory.

Figure 6.7 shows the end of the *OpenSBI* bootloader and the start of the Linux kernel. The first line printed by the Linux kernel indicates the author built the kernel executing, the kernel version and which toolchain he used to compile it.
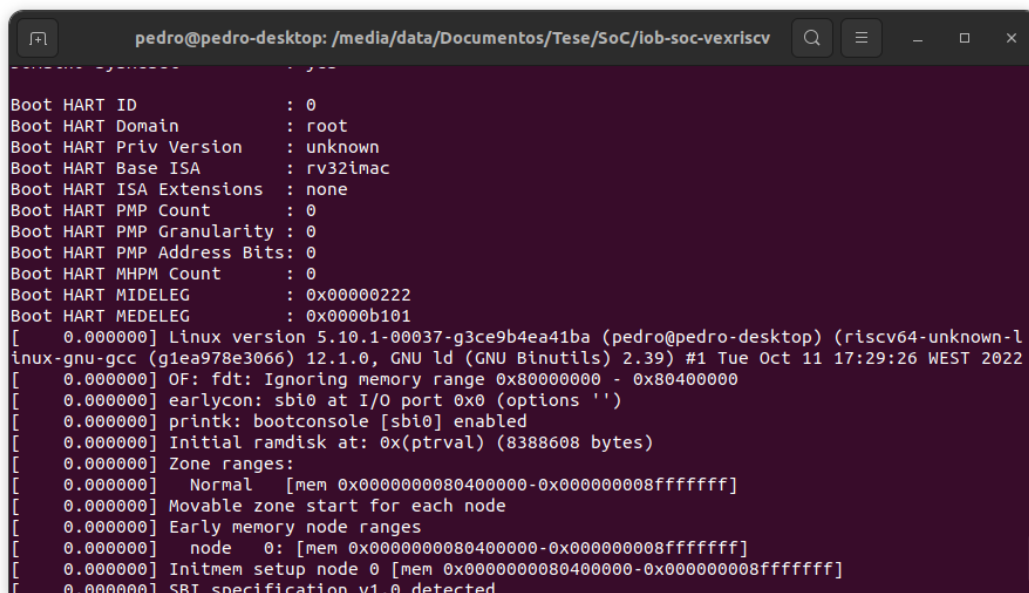

Figure 6.7: Start of the Linux kernel boot with *Verilator*.

While figure 6.7 shows the start of the Linux kernel, figure 6.8 shows the end of the Linux kernel booting process and the execution of the "init" script. The "init" script , as seen in subsection 5.4.4, is the first program the OS executes after the Linux kernel mounts the rootfs and finishes booting. There exist multiple

messages printed to the terminal between the output shown in figure 6.7 and in 6.8. Those messages show the progress while the Linux kernel boots. The Linux kernel boot process's last message is `Run / init as init process`. After that message the SoC executes the "init" program.

Figure 6.8: End of Linux kernel boot with *Verilator*.

The warning `/bin/sh: can't access tty; job control turned off` that appears at the end of the Linux boot in figure 6.8 means the shell program is not writing to a *tty*, but a socket. Advanced commands such as Ctrl+Z and Ctrl+C are unavailable when writing to a socket. Furthermore, sh will not support background processes (command &) and the associated bg/fg/disown/jobs commands. However, processes forking themselves and closing their inputs will still work. This way, the system is protected from a race condition that could occur if both the shell and the background process were waiting for user input. This problem happens because the author developed the rootfs to be lightweight for embedded systems. Consequently, it does not implement some Linux files and programs that would enable such features. The "init" script could call the shell program could with `sh +m`.

Figure 6.9 shows the developed minimal OS running on an FPGA. The reader can see that the author has suppressed the shell warning. The initial part of the figure shows the final stage of the Linux kernel booting. After booting, the author tested the `ls /` command that showed the files and directories in the systems' root. Lastly the author executed the `cat init` command for the OS to print the contents of the "init" script to the terminal. The difference between the "init" script printed and the one presented in listing 5.18 is that "IObundle" is printed as a design with ASCII characters and the shell program is called with the "+m" argument.

The time the Linux kernel takes to boot in real hardware, figure 6.9, is almost double what it takes to boot in simulation, figure 6.9. The time to boot is almost double because the memory module used in the simulation does not have any latency. When the L2 cache fetches data from memory in real hardware, it must wait before receiving the data burst. Using the *CYCLONE V* FPGA board the Linux kernel takes 7.01 seconds to boot. The author expected the boot to take longer since the system clock frequency used with the *CYCLONE V* is 50 MHz. The Kintex Ultrascale was able to run with a frequency of 100 MHz. The *OpenSBI* bootloader and the Device Tree Blob had to be recompiled with the system

Figure 6.9: Linux kernel boot in the FPGA.

frequency defined to 50 MHz to run in the *CYCLONE V*.

Table 6.6 show the resources used by the *IOb-SoC-Linux* in the different FPGAs.

	Resources	FPGA usage %
ALM	11,227	10
DSP	8	3
FF	13725	2
BRAM blocks	234	19
BRAM bits	755,424	9

(a) Cyclone V GT

	Resources	FPGA usage %
LUTs	23126	9.54
Registers	24505	5.05
DSPs	10	0.52
BRAM	39.5	6.58

(b) Kintex Ultrascale

Table 6.6: FPGA results for interrupt routine program.

The tables in 6.6 show that the resources utilization from the *IOb-SoC-Linux* is less than 10% in the supported FPGA boards. Comparing the tables in 6.6 with the tables in 6.2 it is clear the *IOb-SoC-Linux* does not use many more resources than the *IOb-SoC*. The FPGA still has enough resources to implement hardware accelerators.

7 | Conclusions

This chapter goes through the thesis achievements, contributions to open-source repositories during this project, future optimisations for the work developed, and projects using *IOb-SoC-Linux* as its core.

7.1 Achievements

The work developed in this thesis project successfully achieves the project's goals. The *Verilator* simulation testbench created is much faster than the previous verification process, and using *Verilator* saves time when verifying an SoC based on the *IOb-SoC*. Furthermore, the *Python Console* program developed works correctly with the simulation testbench and the FPGA boards.

The author successfully integrated a CPU that meets the requirements to run an OS and verified that what worked with the previous CPU still worked in the new SoC. The CPU integrated is the *VexRiscv* CPU generated using the *SpinalHDL VexRiscv* platform. Additionally, the author successfully created the CLINT component for timer and software interrupts, and the simulation testbench developed for the CLINT shows it works as expected. Moreover, the interrupt routine firmware developed, which takes advantage of the CLINT, shows how interrupts work in bare-metal with the *IOb-SoC-Linux*. The PLIC integrated into *IOb-SoC-Linux* allows the SoC to support interrupts from its peripheral hardware components. Furthermore, since the Linux OS does not support the *IOb-SoC* current UART, in this thesis, the author adapts an industry-standard *UART16550* to the *IOb-SoC-Linux*. The number of resources the complete *IOb-SoC-Linux* uses is less than 10% of the supported FPGA boards. Comparing the *IOb-SoC* resource consumption with the resources used by the *IOb-SoC-Linux*, which can execute a Linux OS, the author can conclude that the developed SoC does only a few more resources than the original. There is still plenty of space in the FPGA o implement new hardware accelerators.

The minimal Linux OS developed executes on both supported FPGA boards and simulation with the *Verilator* testbench. The *OpenSBI* bootloader, the Device Tree Blob, the Linux kernel and the root file system constitute the Linux OS. The *OpenSBI* bootloader implements the *RISC-V* SBI functions, which the supervisor mode software uses to communicate with the machine privilege level. The Device Tree Blob describes the *IOb-SoC-Linux* hardware, which the Linux Kernel uses to know what drivers to use. The Linux kernel implements the system calls that the user applications can use. Lastly, the root file system uses the *Busybox* software package and allows users to interact with the Linux OS. The minimal Linux OS developed takes five seconds to boot in the *Kintex Ultrascale* board and seven seconds in the *Cyclone V*.

Finally, the *Makefiles* written in this thesis allow researchers to use the developed components easily. Building a complete Linux OS with the created *Makefiles* takes the user four minutes and thirty seconds.

7.2 Contributed Repositories

The author contributed to various existing and new open-source repositories during the thesis development. The engineers working with *IObundle* manage those repositories and make them available on the company's *GitHub* page. The *IObundle GitHub* URL is `https://github.com/IObundle`. The following list shows each repository's respective names and the authors' contributions.

- **iob-soc**: The *IOb-SoC* platform currently uses the verification software developed in this thesis.

- **iob-soc-OpenCryptoLinux**: This repository holds the *IOb-SoC-Linux* capable of executing the minimal Linux OS developed. Furthermore, the author will continue to work in this repository as he is currently working on the *OpenCryptoLinux* project.

- **iob-lib**: The SoC uses this repository as the *LIB* submodule. The *Python Console* program is part of the python scripts of the *LIB* submodule. The author also contributed minor optimisations to this repository that he did not present in this thesis.

- **iob-vexriscv**: This repository holds the *IOb-SoC-Linux VexRiscv* CPU and the software needed to build the minimal Linux OS.

- **iob-uart16550**: The *UART16550* adapted for the *IOb-SoC-Linux* is here.

- **iob-clint**: This repository contains the CLINT hardware design, the CLINT simulation testbench and the CLINT bare-metal firmware functions. The interrupt routine firmware example, which developers can use in *IOb-SoC-Linux*, is also in this repository.

- **iob-plic**: The PLIC hardware unit adapted for the *IOb-SoC-Linux* is here.

7.3 Future Work

After completing the thesis objectives, there is still room for new features and optimisation. The author or other developer/researcher can submit new features to optimise *IOb-SoC-Linux*. The author is working on four optimisations. First, the candidate could optimise the performance of the SoC if the *VexRiscv* CPU integrated into *IOb-SoC-Linux* supported 32 bytes per cache line. Currently, the CPU has an L1 data and instructions cache with 4 bytes per line. Secondly, right now, *IOb-SoC-Linux* does not have support for internet connections. Therefore, the author will adapt an existing Ethernet controller to the *IOb-SoC-Linux* by creating a hardware wrapper. Thirdly, currently, *IOb-SoC-Linux* has to transfer the Linux OS every time it starts working and transmitting data through the UART is slow. Integrating a Serial Peripheral Interface (SPI) controller would allow *IOb-SoC-Linux* to load the software from flash memory. An alternative solution would be to implement a PCI interface and transfer the data through it. Lastly, the author will optimise the *Console* program. With the existing program, the user input is not fluid since the *Console* software does the input processing sequentially after the program waits a short period for data to be read from the serial connection. The optimised *Console* program should receive the user input and read from the serial interface concurrently in two different threads.

One of the best advantages of this thesis project is the opportunities it creates. Many possible projects could use *IOb-SoC-Linux*. The author is currently involved in a project called *OpenCryptoLinux*, which the *NLnet Foundation* has funded through the *NGI Assure Fund* with financial support from the *European*

Commission's Next Generation Internet programme. *OpenCryptoLinux* aims to adapt the *OpenCryptoHW* [19] project to *IOb-SoC-Linux* therefore creating a secure and user-friendly open-source SoC template with cryptography functions running a Linux OS on a *RISC-V* system. A colleague working at *IObundle* develops *OpenCryptoHW*, which implements a reconfigurable open-source cryptographic hardware IP core. The hardware is reconfigurable because the CPU controls low-cost Coarse-Grained Reconfigurable Arrays (CGRAS). *OpenCryptoLinux* can enhance the security, privacy, performance, and energy efficiency of future Internet of Things (IoT) devices. The *OpenCryptoLinux* project will be fully open-source, guaranteeing public scrutiny and quality. The author has to develop Linux drivers that can control the *openCryptoHW* hardware and possibly integrate a DMA controller in the *IOb-SoC-Linux* to integrate *openCryptoHW* features in the Linux OS. Finally, it would also be interesting to implement the *IOb-SoC-Linux* as an ASIC and create a development board with it at its core.

A | Annexes

A.1 *RISC-V* CPU registers

5-bit Encoding (rx)	3-bit Compressed Encoding (rx')	Register	ABI Name	Description	Saved by Calle-
0	-	x0	zero	hardwired zero	-
1	-	x1	ra	return address	-R
2	-	x2	sp	stack pointer	-E
3	-	x3	gp	global pointer	-
4	-	x4	tp	thread pointer	-
5	-	x5	t0	temporary register 0	-R
6	-	x6	t1	temporary register 1	-R
7	-	x7	t2	temporary register 2	-R
8	0	x8	s0 / fp	saved register 0 / frame pointer	-E
9	1	x9	s1	saved register 1	-E
10	2	x10	a0	function argument 0 / return value 0	-R
11	3	x11	a1	function argument 1 / return value 1	-R
12	4	x12	a2	function argument 2	-R
13	5	x13	a3	function argument 3	-R
14	6	x14	a4	function argument 4	-R
15	7	x15	a5	function argument 5	-R
16	-	x16	a6	function argument 6	-R
17	-	x17	a7	function argument 7	-R
18	-	x18	s2	saved register 2	-E
19	-	x19	s3	saved register 3	-E
20	-	x20	s4	saved register 4	-E
21	-	x21	s5	saved register 5	-E
22	-	x22	s6	saved register 6	-E
23	-	x23	s7	saved register 7	-E
24	-	x24	s8	saved register 8	-E
25	-	x25	s9	saved register 9	-E
26	-	x26	s10	saved register 10	-E
27	-	x27	s11	saved register 11	-E
28	-	x28	t3	temporary register 3	-R
29	-	x29	t4	temporary register 4	-R
30	-	x30	t5	temporary register 5	-R
31	-	x31	t6	temporary register 6	-R

Table A.1: *RISC-V* CPU registers.

A.2 Some of the *RISC-V* Control and Status Registers

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.

Table A.2: *RISC-V* supervisor-level CSR.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine bad guest physical address.

Table A.3: Some of the *RISC-V* machine-level CSR.

A.3 Device Tree Source

```
1  /dts-v1/;
2
3  / {
4    #address-cells = <1>;
5    #size-cells = <1>;
6    model = "IOb-SoC, VexRiscv";
7    compatible = "IOb-SoC, VexRiscv";
8    cpus {
9      #address-cells = <0x1>;
10     #size-cells = <0x0>;
11     timebase-frequency = <100000>;
12     CPU0: cpu@0 {
13       clock-frequency = <100000000>;
14             device_type = "cpu";
15       reg = <0x0>;
16       status = "okay";
17             compatible = "riscv";
18       riscv,isa = "rv32imac";
19       mmu-type = "riscv,sv32";
20       d-cache-block-size = <0x40>;
21       d-cache-sets = <0x40>;
22       d-cache-size = <0x8000>;
23       d-tlb-sets = <0x1>;
```

```
24        d-tlb-size = <0x20>;
25        i-cache-block-size = <0x40>;
26        i-cache-sets = <0x40>;
27        i-cache-size = <0x8000>;
28        i-tlb-sets = <0x1>;
29        i-tlb-size = <0x20>;
30        tlb-split;
31        CPU0_intc: interrupt-controller {
32          #interrupt-cells = <1>;
33          interrupt-controller;
34          compatible = "riscv,cpu-intc";
35        };
36      };
37    };
38    memory@80000000 {
39      device_type = "memory";
40      reg = <0x80000000 0x10000000>;
41    };
42    chosen {
43      bootargs = "rootwait console=hvc0 earlycon=sbi root=/dev/ram0 init=/sbin/init swiotlb
          =32";
44      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: *IOb-SoC-Linux* Device Tree Source (DTS).

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] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

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

[5] 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`>.

[6] Richard Barry et al. "FreeRTOS". In: *Internet, Oct* (2008).

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

[8] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555.

[9] Sebastien Bourdeauducq. "Migen Manual". In: *Release 0. X* 32 (2012).

[10] Edouard Bugnion et al. "Bringing virtualization to the x86 architecture with the original vmware workstation". In: *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), pp. 1–51.

[11] Abner Chang et al. *RISC-V Platform-Level Interrupt Controller Specification*. 2022. <`https://github.com/riscv/riscv-plic-spec/releases/download/1.0.0_rc4/riscv-plic-1.0.0_rc4.pdf`>.

[12] Palmer Dabbelt and Atish Patra. *RISC-V Supervisor Binary Interface Specification*. 2022. <`https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v1.0.0/riscv-sbi.pdf`>.

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

[14] devicetree-org. *Devicetree Specification*. <`https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf`>.

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

[16] Jacob Gorban. "UART IP Core Specification". In: (2002). <`https://raw.githubusercontent.com/freecores/uart16550/master/doc/UART_spec.pdf`>.

[17] IObundle Lda. *IOb-Lib*. <`https://github.com/IObundle/iob-lib`>.

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

[19] IObundle Lda. *IOb-SoC-openCryptoHW*. `<https://github.com/IObundle/iob-soc-opencrypt ohw>`.

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

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

[22] lowRISC. *OpenTitan - PLIC*. `<https://github.com/lowRISC/opentitan/tree/master/hw/ip_ templates/rv_plic>`.

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

[24] VM Oracle. *VirtualBox*. 2015.

[25] C Papon. "SpinalHDL: An alternative hardware description language". In: *FOSDEM*. 2017.

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

[27] Thomas Petazzoni and Free Electrons. "Buildroot: a nice, simple and efficient embedded Linux build system". In: *Embedded Linux System Conference*. Vol. 2012. 2012.

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

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

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

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

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

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

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

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

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

[37] Wilson Snyder. "Verilator: Fast, free, but for me?" In: *DVClub Presentation* (2010), p. 11.

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

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

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

[41] 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/>`.

[42] 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/>`.

[43] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[44] Neethika Tidala. "High performance network on chip using AXI4 protocol interface on an FPGA". In: *2018 second international conference on electronics, communication and aerospace technology (ICECA)*. IEEE. 2018, pp. 1647–1651.

[45] Linus Torvalds. "Linux: a portable operating system". In: *Master's thesis, University of Helsinki, dept. of Computing Science* (1997).

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

[47] Andrew Waterman et al. *RISC-V Advanced Core Local Interruptor Specification*. <`https://github.com/riscv/riscv-aclint/releases/download/v1.0-rc4/riscv-aclint-1.0-rc4.pdf`>.

[48] Andrew Waterman et al. "The RISC-V instruction set manual". In: *Volume I: User-Level ISA, version* 2 (2014).

[49] Andrew Waterman et al. "The RISC-V instruction set manual volume II: Privileged architecture version 1.9". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129* (2016).

[50] Stephen Williams. *Icarus verilog*. 2006.

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

[52] 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`.

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