# Linux-capable RISC-V CPU for IOb-SoC

Pedro Nuno de Melo Antunes
pedronmantunes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 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 Holdings (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 is a well-polished OS since people have been using it for over three decades. The problem is that open-source SoC platform solutions 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 based the work on *IOb-SoC*, a modular and configurable open-source SoC platform that only runs bare-metal applications. This project achieves its goals by changing the *IOb-SoC* CPU and adding three hardware peripherals. Additionally, the author develops software solutions that improve the *IOb-SoC* platform, complement the hardware components created and enhance the hardware to allow the execution of a complete OS in a new SoC called *IOb-SoC-Linux*. The size of *IOb-SoC-Linux* is only marginally above that of the original *IOb-SoC* and can run in most low-cost FPGAs. The Linux OS takes four minutes and thirty seconds to build. The kernel boots in a *Kintex Ultrascale* device in five seconds and in seven seconds in a *Cyclone V* device. The work developed in this thesis met all the project's goals and went beyond them.

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

## 1. Introduction

### 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 [15] 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 lack of open-source hardware makes it difficult to develop fully open-source systems. With the appearance of *RISC-V* [1], open-source hardware availability started growing. Developing a *RISC-V* System on a chip (SoC) capable of running a Linux OS allows researchers to execute an OS in a fully open-source system. Having a Linux OS running in an SoC enables developers to create new applications for 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 are limited on software functionalities and must be aware of the SoC hardware characteristics. If developers were to build an application using Real-Time Operating Systems (RTOS), for example, *freeRTOS* [2], they would only have access to features such as a scheduler, events, threads, semaphores and message boxes. 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 various 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.

The development of a *RISC-V* SoC capable of running a Linux OS allows future open-source developments. Such as producing hardware accelerators which work with a Linux OS and integrating them with *IOb-SoC-Linux*. These, and the possibilities to test in a real-world application, were the main reasons and motivations for developing this thesis.

### 1.2. Objectives and Deliveries

This study aims to develop an open-source SoC and execute a minimal Linux OS. The SoC developed must derive from the existing *IOb-SoC* [6]. *IOb-SoC* is a modular open-source *RISC-V* SoC that allows researchers to develop their own SoC. The *IObundle* developers use *Verilog* [12] to describe *IOb-SoC* and its peripheral's hardware.

An SoC compatible with a Linux OS must contain a compatible CPU, support for interrupts and an appropriate UART. The *IOb-SoC* CPU has a problem: it cannot run an OS, only bare-metal applications. Therefore, *IOb-SoC-Linux* contains a 32-bit *RISC-V* CPU capable of running Linux. Since the *IOb-SoC* does not support interrupts, *IOb-SoC-Linux* requires the integration of the respective hardware created for that purpose. Lastly, the Linux kernel does have drivers for the *IOb-SoC* UART. Consequently, *IOb-SoC-Linux* must incorporate an industry-standard *UART16550*.

Four major software components make up a Linux OS. Those software components are the Linux kernel, the bootloader firmware, the root file system (rootfs) and a Device Tree Blob (DTB). 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 process of generating and deploying the Linux OS to *IOb-SoC-Linux* has to be automated and documented. So, after this work, creating new OSs with different characteristics will be straightforward.

Finally, the system is verified both on simulation and running on an FPGA board. *IOb-SoC* needs a fast *Verilog* simulator to verify the Linux OS execution. Therefore, the project must develop a simulation testbench using the free-of-charge and open-source *Verilator* [11] simulator.

## 2. Must-Have Concepts

This section discusses topics that help understand the technological developments along this thesis project. The developments involve both hardware and software components. As such, there are hardware and software concepts that are important to have before discussing the following chapters.

### 2.1. The *IOb-SoC* platform

The *IOb-SoC* [6] is a System on a chip (SoC) template that eases the creation of a new SoC. The IOb-SoC provides a base *Verilog* [12] 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* [13] master bus.

Figure 1 represents a sketch of the SoC design. This design is valid at the start of this project. During the hardware development the *IOb-SoC* original template suffered a few alterations.

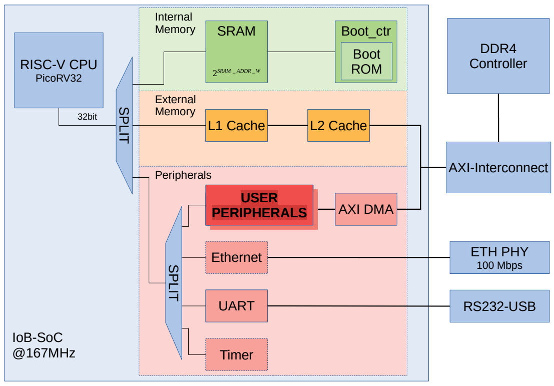


Figure 1: *IOb-SoC* sketch.

*IOb-SoC* currently supports two FPGA board models: the *Xilinx Kintex UltraScale KU040 Development Board* and the *Cyclone V GT FPGA Development Kit*.

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* FPA 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.

A *IOb-SoC* **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 *IOb-SoC* **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 shows how the *IOb-SoC* distributes the signals in the request bus. Furthermore, figure 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.

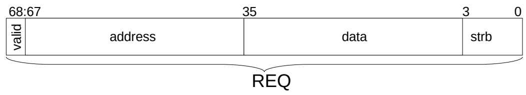


Figure 2: Request bus with address and data width equal to 32 bits.

The *IOb-SoC* **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 3 shows how the request signal is composed when the address and data width are 32 bits.

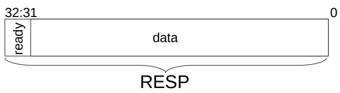


Figure 3: Response bus with address and data width equal to 32 bits.

The ***iob-split*** is simply a configurable demultiplexer (DEMUX). The developer can configure it when he instantiates the *iob-split* hardware module. The developer can change the size of the DEMUX 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 DEMUX selection bits. Equation 1 calculates the number of the selection bits.

$$Nb = log_2(N\_SLAVES) + (log_2(N\_SLAVES) == 0) \tag{1}$$

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.

The *IOb-SoC* **bootloader** is the first firmware to run on the SoC. Figure 4 represents a flow chart of the bootloader firmware behaviour.

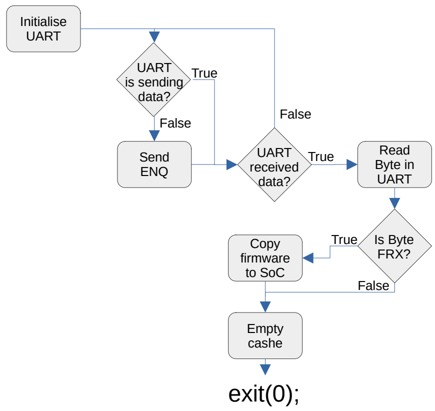


Figure 4: Bootloader firmware flow chart.

## 2.2. *RISC-V*

*RISC-V* [1] is a free-to-use, open-source RISC Instruction set architecture (ISA). The *RISC-V* ISA 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.

The *RISC-V* ISA is divided in two main volumes. The "RISC-V Instruction Set Manual Volume I" [18] 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" [17] defines the *RISC-V* **privilege** levels and the instructions that take advantage of them. Table 1 shows the privilege levels currently defined in the *RISC-V* specification. Developers must implement all three privilege levels to run a Unix-like OS.

| Level | Name | Abbreviation |
|---|---|---|
| 0 | User/Application | U |
| 1 | Supervisor | S |
| 2 | Reserved | |
| 3 | Machine | M |

Table 1: *RISC-V* privilege levels.

The *RISC-V* **CLINT** specification [16] 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.

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 [4].

In the *RISC-V* Platform Specification [10] it is defined that every embedded 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 and developers often use it to connect to an RS-232 interface.

The Supervisor Binary Interface (**SBI**) specification [5] defines an abstraction for platform-specific functionalities. Figure 5 illustrates the purpose of the SBI in a system executing an OS like the one the author is going to develop in this thesis.

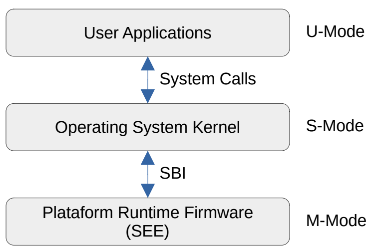


Figure 5: *RISC-V* system running an OS.

***OpenSBI*** is the recommended interface between a platform-specific firmware running in M-mode and a general-purpose OS executing in S-mode.

Figure 6 shows the various stages a *RISC-V* system has to pass through to fully **boot a Linux OS**.

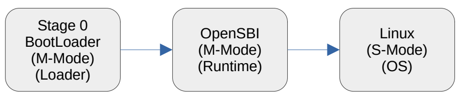


Figure 6: Stages of the Linux boot on *RISC-V* on a minimal system.

### 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.

Developers can use cycle-accurate and event-driven simulators to simulate the hardware logic designs. **Cycle-accurate** simulators are suitable for complex hardware designs. An example of a cycle-accurate simulator would be *Verilator*. **Event-driven** simulators are adequate for small hardware designs. An example of an event-driven simulator would be *Icarus Verilog*.

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. An example of a functional emulator would be ***QEMU*** [3].

## 3. Existing Embedded Technologies

There already exists embedded microcontrollers capable of running Linux. However, most of them are closed source. For example from Arm Holdings (Arm ®), Andes Technology and SiFive. Andes Technology and SiFive are members of the *RISC-V* community and have contributed with open-source components.

Built upon the *RISC-V* open-source ISA, various open-source CPU designs have emerged. An *application processor* is needed to run a Linux OS. *Application processors* have the necessary CSR, support M+S+U privilege modes, and support atomic instructions.

An open-source CPU solution would be either the *CVA6* [20] (previously known as Ariane), *BOOM* [21] or *VexRiscv* [8]. 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. The Berkeley Out-of-Order *RISC-V* Processor (*BOOM*) is a superscalar Out-of-Order processor executing the RV64GC variant of the *RISC-V* ISA. The *VexRiscv* CPU is a 32-bit Linux Capable *RISC-V* CPU written in the *SpinalHDL* [9].

## 4. Hardware Developed

The author had to develop four hardware modules to build a 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*, minor changes to the *IOb-SoC* core were made. The newly used CPU core was generated based on the *SpinalHDL* [9] *VexRiscv* [8] platform. The *VexRiscv* platform enabled the development of a *VexRiscv* CPU core that meets the requirements of an OS. The *VexRiscv* CPU still needed a CPU wrapper to integrate with the *IOb-SoC* interface. The Linux OS also requires a compatible UART to communicate with the user. Linux has drivers that support an existing *UART16550*. A hardware wrapper allows the integration of the *UART16550* on *IOb-SoC-Linux*. Additionally, the SoC has to support timer and software interrupts to run an OS. The CLINT hardware module developed generates timer and software-related interrupts for a *RISC-V* system. Another hardware component which manages interrupts in a *RISC-V* system is the PLIC. A developed hardware component creates an interface with the *IOb-SoC* and instantiates an existing PLIC core and register modules enabling external interrupts on *IOb-SoC-Linux*.

A sketch of the SoC developed can be seen in figure 7.

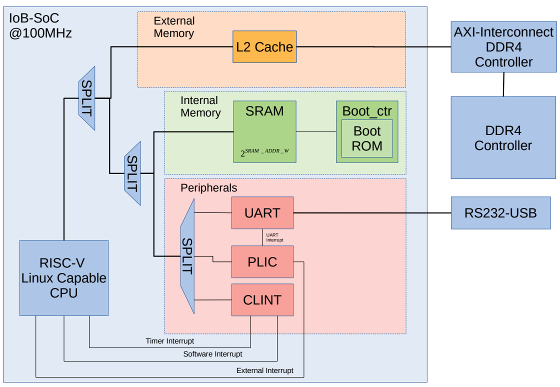


Figure 7: Developed SoC sketch.

## 5. Software Developed

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 new *Console* program written in *Python* allows the *IOb-SoC* platform to communicate through serial with the board. Previously, the *Console* program was written in *C* and had fewer features than the new *Console*. The new *Console* can work with the simulator testbench and communicate with a Linux OS running in *IOb-SoC-Linux*. Secondly, based on the previous *IOb-SoC* verification software, a new hardware simulation testbench can test the SoC and communicate with the *Console* program. Moreover, the *Verilator* [11] simulation software allowed the creation of a *Verilator C++* testbench to test the SoC faster. Thirdly, a hardware simulation testbench created for the CLINT verifies its behaviour, and a bare-metal interrupt routine firmware developed shows how to use interrupts in *IOb-SoC-Linux*. Finally, the author adapted, built and deployed the software needed to execute a Linux OS in the SoC. The adapted *IOb-SoC* bootloader firmware allows loading the software to the *IOb-SoC-VexRiscv* memory. A device tree file describes the hardware components of the SoC to the Linux kernel. The compiled Linux kernel version must be compatible with the *VexRiscv* CPU, and the root file system developed must be adequate for a minimal Linux OS. While developing the hardware and software components, Makefile scripts helped integrate the components in *IOb-SoC* and automatise the building and deployment process.

Figure 8 shows the *Console* program flowchart.

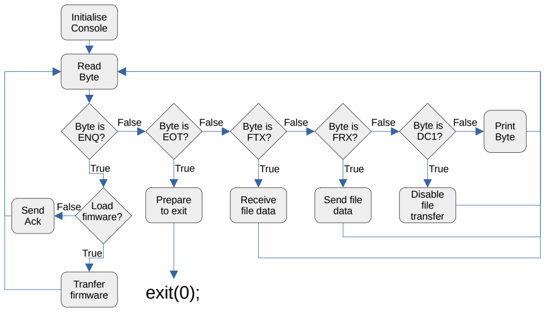


Figure 8: *Console* program flowchart.

The new verification software interacts with the *Console* through files. Figure 9 represents a sketch of the verification software and its interaction with the *Console*.

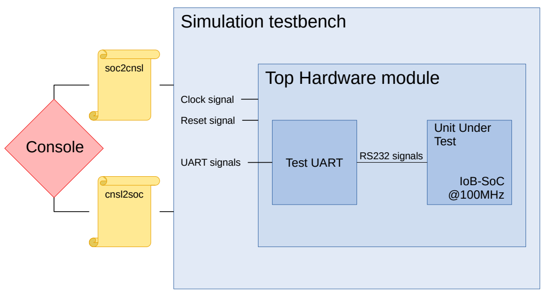


Figure 9: Simulated hardware interfaces.

## 6. Project Results

The following section analyses the results obtained from the hardware and software developed in this project. The candidate successfully executes 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 depending on the software needs.

The objective of this thesis project was to run an Operating System in the *IOb-SoC-Linux*. Table 2 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 | 4m29,570s |
| user | 8m12,039s |
| sys | 0m56,887s |

Table 2: Time it takes to build the OS.

The OS size is to big to run in the FPGA internal memory. Consequently, the author had to implement the *IOb-SoC-Linux* on the FPGA with access to the external memory. Figures 10 and 11 show the start of the OS simulation with *Verilator*.

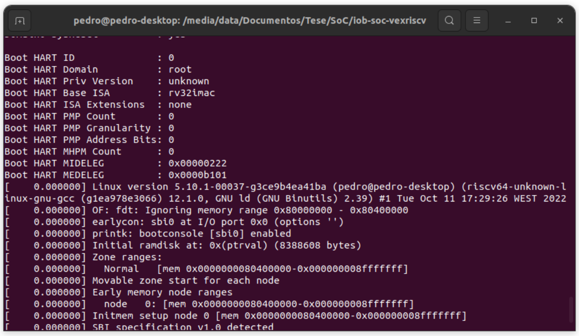


Figure 10: *iob-UART16550* and *iob-PLIC* properties.


Figure 11: *IOb-SoC* bootloader and *OpenSBI* firmware.

Figure 10 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 11 shows the *iob-bootloader* and the start of the *OpenSBI* bootloader. The *iob-bootloader* in figure 11 does not transfer the software to the memory because the author executed the simulation considering that the software was already in the memory.

Figure 12 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.

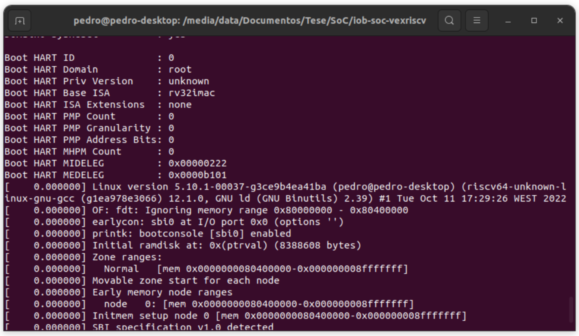


Figure 12: Start of the Linux kernel boot with *Verilator*.

While figure 12 shows the start of the Linux kernel, figure 13 shows the end of the Linux kernel booting process and the execution of the "init" script. The "init" script 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 12 and in 13. 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 14 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 time the Linux kernel takes to boot in real hardware, figure 14, is almost double what it takes to boot in simulation, figure 14. The time to boot

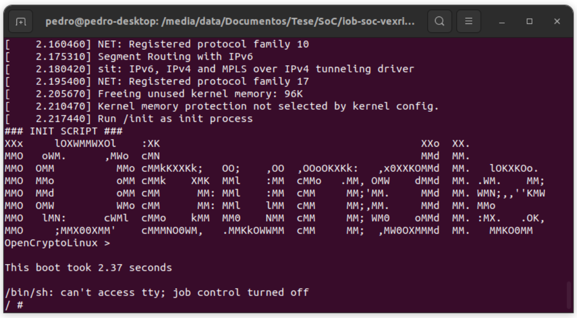


Figure 13: End of Linux kernel boot with *Verilator*.

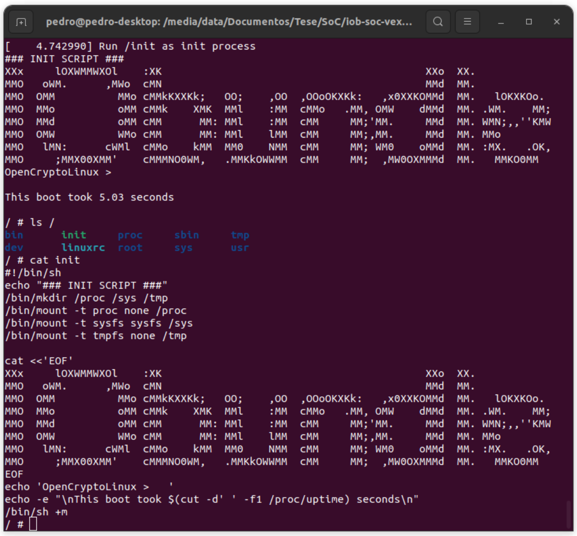


Figure 14: Linux kernel boot in the FPGA.

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

A more complex rootfs generated with *Buildroot* provides more features than the minimal rootfs developed. The *Buildroot* rootfs allows using *MicroPython* [14] in *IOb-SoC-Linux* and executing the *Dhrystones* [19] benchmarking software. The rootfs size is a little over 2MB. Figure 15 shows the final output of the *Dhrystones* benchmark and the execution of simple commands in *MicroPython*. With the *Buildroot* rootfs the Linux kernel takes 6.40 seconds to boot in the *Kintex Ultrascale* and 8.14 seconds in the *CYCLONE V*.

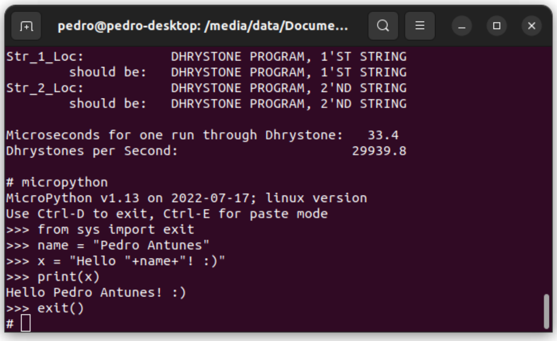


Figure 15: Linux OS with *Buildroot* rootfs.

*MicroPython* is a software project that aims to implement a *Python* version, highly compatible with *Python3*, in microcontrollers and small embedded systems. *Dhrystones* is a general-performance benchmarking software used in multiple embedded systems. A common representation of the *Dhrystones* benchmark is *DMIPS*. Table 3 represents a comparison between the *Dhrystones* benchmarking scores of both FPGA boards.

| | **Kintex Ultrascale** | **CYCLONE V** |
|---|---|---|
| DMIPS | 23.33 | 17.04 |

Table 3: *Dhrystones* benchmarking.

Tables 5 and 4 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 |

Table 4: Cyclone V GT

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

Table 5: Kintex Ultrascale

Tables 5 and 4 show that the resources utilization from the *IOb-SoC-Linux* is not much bigger than the *IOb-SoC*. The FPGA still has enough resources to implement hardware accelerators.

## 7. Conclusions

### 7.1. Achievements

The *Verilator* simulation testbench created in this thesis was much faster than the previous verification process, saving 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* 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 requires only a few more resources than the original. The *IOb-SoC-Linux* resource usage leaves plenty of space in the FPGA to implement new hardware accelerators.

The minimal Linux OS developed executes on the supported FPGA boards and in the 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. The work developed in this thesis successfully achieved the project's goals.

### 7.2. Future Work

After completing this thesis, there is still space for new features and optimisation. The author or others can submit new features to optimise *IOb-SoC-Linux*. The author is working on four optimisations. First, enhancing the L1 cache may optimise the performance of the SoC by integrating a *VexRiscv* CPU into *IOb-SoC-Linux*, which supports 32 bytes per cache line. The current CPU has an L1 data and instructions cache with 4 bytes per line. Secondly, *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, *IOb-SoC-Linux* has to transfer the Linux OS every time it starts working. Transmitting data through the UART is slow. Integrating a Serial Peripheral Interface (SPI) controller would allow *IOb-SoC-Linux* to load the software from a 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. 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 strengths of this thesis 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* [7] 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. *OpenCryptoHW* IObundle developments implement a reconfigurable open-source cryptographic hardware IP core. The hardware is reconfigurable because the CPU controls 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.


**Acknowledgements**
The author would like to thank his friends and professors who helped and accompanied him through his studies. Furthermore, above all, the author is thankful for his family that has been in his life since day 0, giving advice and guiding him, leading him to where he is today.


**References**


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

[2] R. Barry et al. Freertos. *Internet, Oct*, 2008.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[4] A. Chang, A. Waterman, R. Balas, D. Barbier, J. Scheel, J. Clarke, K. Asanovic, P. Dabbelt, and Yan. Risc-v platform-level interrupt controller specification, 2022.

[5] P. Dabbelt and A. Patra. Risc-v supervisor binary interface specification, 2022.

[6] I. Lda. Iob-soc.

[7] I. Lda. Iob-soc-opencryptohw.

[8] C. Papon. Vexriscv.

[9] C. Papon. Spinalhdl: An alternative hardware description language. In *FOSDEM*, 2017.

[10] riscv. Risc-v platform specification.

[11] W. Snyder. Verilator: Fast, free, but for me? *DVClub Presentation*, page 11, 2010.

[12] D. Thomas and P. Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[13] N. 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)*, pages 1647–1651. IEEE, 2018.

[14] N. H. Tollervey. *Programming with MicroPython: embedded programming with microcontrollers and Python.* " O'Reilly Media, Inc.", 2017.

[15] L. Torvalds. Linux: a portable operating system. *Master's thesis, University of Helsinki, dept. of Computing Science*, 1997.

[16] A. Waterman, G. Favor, J. Hauser, A. Patel, B. Meng, and W. Norris. Risc-v advanced core local interruptor specification.

[17] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual volume ii: Privileged architecture version 1.9. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.

[18] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA, version*, 2, 2014.

[19] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[20] 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. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 10 2019.

[21] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. 05 2020.