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 allow 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

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 [10] 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 author will adapt the existing *IOb-SoC* [4] to make 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* [7] to describe *IOb-SoC* and peripherals hardware.

The author had first to swap the *IOb-SoC* CPU. 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 interruption, the author had to create and integrate into the *IOb-SoC-Linux* the hardware needed to generate interrupts in a *RISC-V* SoC. Lastly, the author had to ensure the Linux OS supports the SoC UART. Since Linux does not support the *IOb-SoC* UART, the author integrated a UART16550 in *IOb-SoC-Linux*.

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). The author built 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 OS to *IOb-SoC-Linux*. So, after this work, creating new OSs with different characteristics will be straightforward.

Finally, the system was 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* [6] 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.

The candidate will develop an 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 vital to understand how the *IOb-SoC* works beforehand. Studying the RISC-V Instruction set architecture (ISA) is also indispensable. 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 an Operating System (OS) boot flow 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* [4] is a System on a chip (SoC) template that eases the creation of a new SoC. The IObSoC provides a base *Verilog* [7] 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* [8] 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.

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.

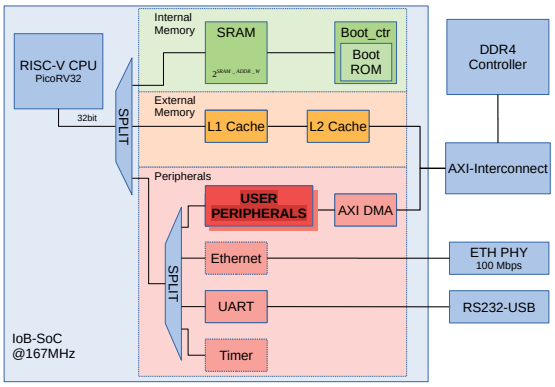

Figure 1: *IOb-SoC* sketch.

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.

2.2. Sub-section...

More text...

3. Implementation

Place text here...

3.1. Sub-section...
More text...

3.2. Sub-section...
More text...

4. Project Results

In the following chapter, the author analyses the results obtained from the hardware and software developed in this thesis 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 1 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 1: 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 (2^23) 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 figures 2 and 3 the reader can see the start of the OS simulation with *Verilator*.

Figure 2 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 3 shows

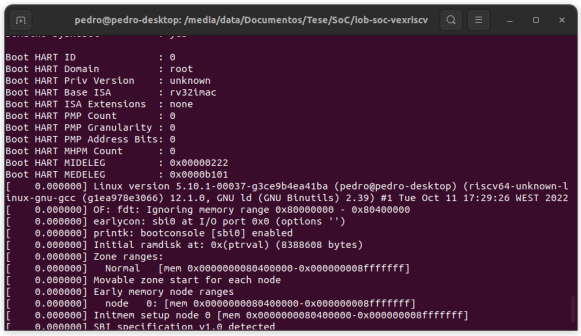

Figure 2: *iob-UART16550* and *iob-PLIC* properties.

Figure 3: *IOb-SoC* bootloader and *OpenSBI* firmware.

the *iob-bootloader* and the start of the *OpenSBI* bootloader. The *iob-bootloader* in figure 3 does not transfer the software to the memory because the author executed the simulation considering that the software was already in the memory.

Figure 4 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.

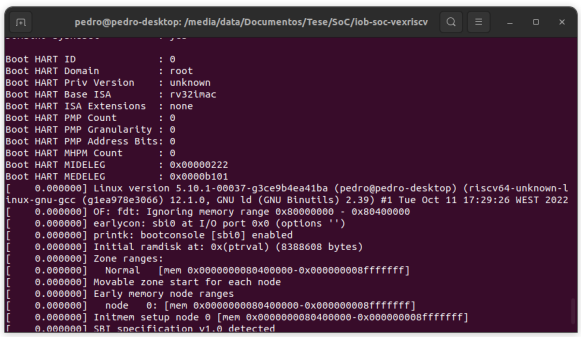

Figure 4: Start of the Linux kernel boot with *Verilator*.

While figure 4 shows the start of the Linux kernel, figure 5 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 4 and in 5. 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.

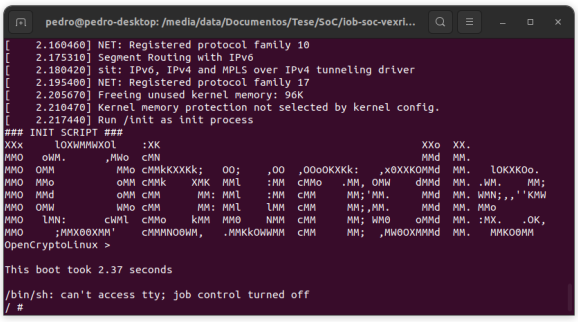

Figure 5: End of Linux kernel boot with *Verilator*.

Figure 6 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.

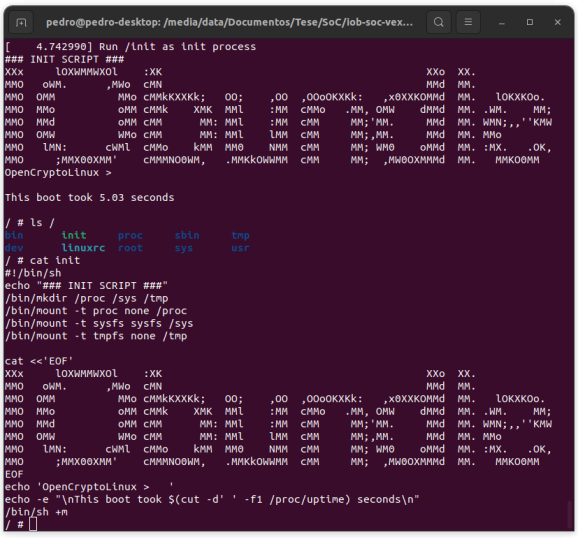

Figure 6: Linux kernel boot in the FPGA.

The time the Linux kernel takes to boot in real hardware, figure 6, is almost double what it takes to boot in simulation, figure 6. 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 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* [9] in *IOb-SoC-Linux* and executing the *Dhrystones* [11] benchmarking software. The rootfs size is a little over 2MB. Figure 7 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*.

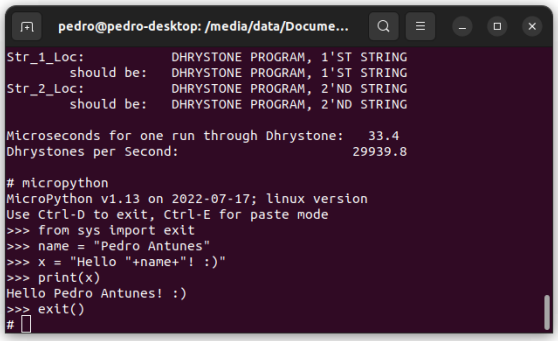

Figure 7: Linux OS with *Buildroot* rootfs.

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. With the *Dhrystones* benchmark, developers can compare the efficiency of different computers or compilers. A common representation of the *Dhrystones* benchmark is *DMIPS*. *DMIPS* is the number of *Dhrystones* per Second divided by 1757, the number of *Dhrystones* per second obtained on the *VAX 11/780* [3]. Table 2 represents a comparison between the *Dhrystones* benchmarking scores of both FPGA boards.

Tables 4 and 3 show the resources used by the

	Kintex Ultrascale	CYCLONE V
One run through Dhrystone (ms)	24.4	33.4
Dhrystones per Second	40983.2	29939.8
DMIPS	23.33	17.04

Table 2: *Dhrystones* benchmarking.

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 3: Cyclone V GT

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

Table 4: Kintex Ultrascale

Tables 4 and 3 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.

5. Conclusions

5.1. 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.

5.2. 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 *Con-*

sole 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* [5] 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] J. S. Emer and D. W. Clark. A characterization of processor performance in the vax-11/780. *ACM SIGARCH Computer Architecture News*, 12(3):301–310, 1984.

[4] I. Lda. Iob-soc.

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

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

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

[8] 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.

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

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

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