TÉCNICO LISBOA

Linux capable RISC-V CPU for IOb-SoC

Pedro Nuno de Melo Antunes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

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

Examination Committee

Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Horácio Cláudio De Campos Neto

September 26, 2022

Abstract

Blablabla

Honor Declaration

“Declaro por minha honra de que ”

Local, Date Signature

Contents

List of Figures

List of Tables

Code Listing

Acronyms

ALU arithmetic logic unit.

AMO Atomic Memory Operations.

ASIC Application-Specific Integrated Circuit.

Chisel Constructing Hardware in a Scala Embedded Language.

CLINT Core-local Interrupt Controller.

CPU Central Processing Unit.

CSR Control and status register.

DEMUX desmultiplexer.

FPGA Field-programmable gate array.

HDL Hardware Description Language.

ISA Instruction set architecture.

LR Load-Reserved.

M Machine.

MMU Memory Management Unit.

msb Most Significant bit.

MUX multiplexer.

OoO Out-of-Order.

OS Operating System.

PC Program counter.

PLIC Platform-Level Interrupt Controller.

RTL register-transfer level.

S Supervisor.

SC Store-Conditional.

SoC System on a chip.

U User.

UART Universal asynchronous receiver/transmitter.

UUT Unit Under Test.

1 Introduction

1.1 Motivation

1.2 Objectives

2 Must Have Concepts

During this chapter, we're going to discuss topics that help understand the technology developments made along the thesis project.

2.1 The IOb-SoC Template

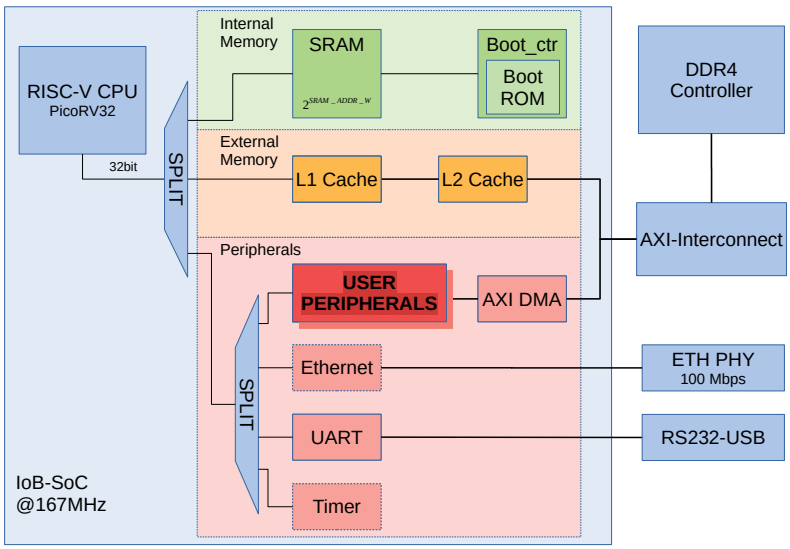

Figure 2.1: *IOb-SoC* sketch.

2.1.1 Adding peripherals

2.1.2 Internal Buses

Review the "cpu_i_resp", "cpu_d_resp", "cpu_i_req" and "cpu_d_req" signals.

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

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

(msb) position, in other words it is the position of the msb of the desmultiplexer selection bits. The number of the selection bits is given by equation 2.1.

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

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

2.1.4 Bootloader and bare-metal firmware

2.2 Open Source Verification tools

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

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

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

- ***'Verilator'***

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

2.2.1 Qemu Simulation

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

2.3 RISC-V architecture

Talk about the 32 registers in the register file

The instruction each ISA extension contains...

Control and status register (CSR) needed to run a full feature OS...

The RISC-V **CLINT** is described

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

2.4 The Linux Boot Flow

2.4.1 Bootloader firmware

2.4.2 What is a device tree?

3 Existing Embedded Technologies

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

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

3.1 Closed source RISC-V Embedded Systems

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

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

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

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

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

3.1.1 Andes Technology

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

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

Andes CPUs witch are capable of running Linux are the *A25* [24] and *AX25* [25]. Both support single and double precision floating point, 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 instruction and data cache. The difference between them is that *A25* is based on 32-bit architecture and the *AX25* is 64-bit. This leads to the *AX25* being ideal for embedded applications that need to access address space over 4GB, and the *A25* being smaller in gate count. Both CPUs can be implemented on the AE350 [23] SoC allowing to use these CPUs on developer boards, for example in the *ADP-XC7K160/410* [22].

3.1.2 SiFive

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

In 2017 SiFive launched *U54* [17] witch was the first RISC-V CPU capable of running a full fledge Operating System like Linux. With it they launched the *U54-MC* [18] 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 witch RISC-V systems must be compliant if they proclaim to use either one. One year after, in 2018, they launched *HiFive Unleashed* [15] witch was the first board that implemented the *U54* CPU and

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

SiFive has since then extended their *U Cores* product lineup. All *U* cores are 64-bit application processors capable of running Linux. The highest performance core is the *U74* [19]. The core architecture is RV64GBC witch means it supports the RISC-V I, M, A, F, D, B and C ISA extensions (explained in ***ref section 2.3.x***). This CPU has already been applied to multiple boards, for example, the *BeagleV* has a SoC with dual core SiFive *U74* CPU. SiFive as also launched its own development board, *HiFive Unmatched* [16], with four *U74* cores on the *U74-MC* [20] SoC. Furthermore, in 2021, Canonical the developers behind Ubuntu have announced the OS support for both the HiFive Unmatched and HiFive Unleashed.

3.2 Open-Source Solutions

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

RISC-V CPUs are most popular in embedded systems and IoT devices. Consequently there exists a widely variety of open-source CPUs witch are implemented on multiple embedded microcontrollers. A few examples of those CPUs would be the *PicoRV32* [26], *NEORV32* [21], *DarkRISCV* [7] and *Ibex* [10] from lowRISC. But those will not be discussed in detail on this paper since they do not meet the requirements to run the Linux Kernel. This CPUs either only support Machine (M) level privilege mode or support Machine (M)+Supervisor (S) mode. Moreover, none of the given examples support the Atomic RISC-V ISA extension. This extension is essential to run Linux. Since the kernel explicitly executes instructions from the Atomic extension.

To run a Linux based Operating System an application processor is needed. A CPU is considered an application processor if it has the hardware required to run a full-feature Operating System (OS) and user applications. This means that the processor should have the required Control and status register (CSR), support M+S+U privilege modes and support atomic instructions. An open-source solution would be either the *CVA6* [27] (previously known as Ariane), *BOOM* [28] or *VexRiscv* [13].

3.2.1 *CVA6*

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

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

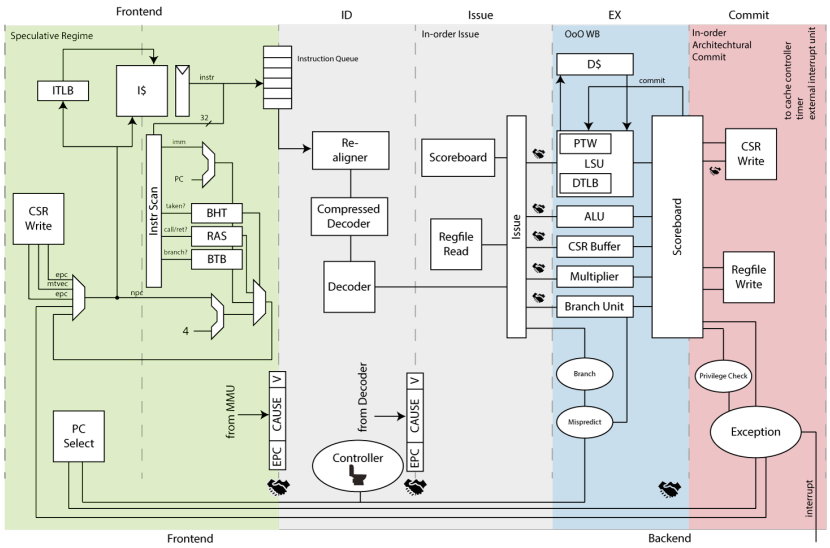

Figure 3.1: CVA6 core design architecture.

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

The CVA6 can be found implemented on *OpenPiton* [4]. *OpenPiton* is an open-source project developed by the Princeton Parallel Group. With it one can easily create a 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* [28]) is a superscalar Out-of-Order (OoO) processor executing the RV64GC variant of the RISC-V ISA. BOOM was created at

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

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

This CPU design is written in Chisel [3] Hardware Description Language (HDL). The Constructing Hardware in a Scala Embedded Language (Chisel) allows to produce synthesizable Verilog designs while using a high level language to describe the hardware. Chisel is an adaptation of Scala [11] programing language, adding hardware construction primitives.

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

3.2.3 *VexRiscv*

The *VexRiscv* [13] CPU is a 32-bit Linux Capable RISC-V CPU written in the *SpinalHDL* [12]. The hardware description of this CPU is accomplished by utilizing a software-oriented approach. Similarly to Chisel, *SpinalHDL* is based on the Scala programing language.

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

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

There exists an open-source project that runs Linux with *VexRiscv*, *linux-on-litex-vexriscv* [9]. *LiteX* is used to create a System on a chip (SoC) around the *VexRiscv* core. *LiteX* SoC design and peripherals are written in *Migen* [6] another hight level HDL. *Migen* unlike *SpinalHDL* and Chisel is based on Python 3.5. On account of the language describing its hardware and the way the *linux-on-litex-vexriscv* project is structured it is very hard to understand how the system works, where the generated RTL is and how to add custom hardware. Furthermore, *linux-on-litex-vexriscv* uses FPGA specific hardware, making it impossible to port the system to ASIC.

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

3.3 Overall CPU comparison

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

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

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

Table 3.1: CPU comparison table: Y means the CPU supports the feature; X means the CPU does not supports the feature; N/A means the feature is no applicable to the respective CPU.

4 Hardware Developed

During the development of this thesis there was both hardware and software developed. In this chapter we are going to go through the hardware developed in order 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 to develop this project SoC. Firstly, it is open-source hardware. Witch means there are no royalties and the source code is publicly available. Secondly, adding new peripherals is very easy and intuitive, as it was previously seen in section 2.1. Finally, it already has some features that make it ideal to use. *IOb-SoC* already implements the interface with an internal (SRAM) and an external (DRAM) memory, contains iob-cache system and boot hardware unit. The boot hardware unit controls the first boot stage (also known as stage zero) that is 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*) is not capable of running a full-feature Operating System. The UART had to be swapped since there were no compatible Linux drivers that worked with *iob-UART*. Besides swapping a few components from the chip new hardware had to be added. The additional hardware is the CLINT and the PLIC both compatible with RISC-V specifications. The CLINT was added to implement timer and software interrupts on the SoC. The PLIC was added to manage interrupts generated by other peripherals, for example from the UART. A sketch of the SoC developed can be seen in figure 4.1.

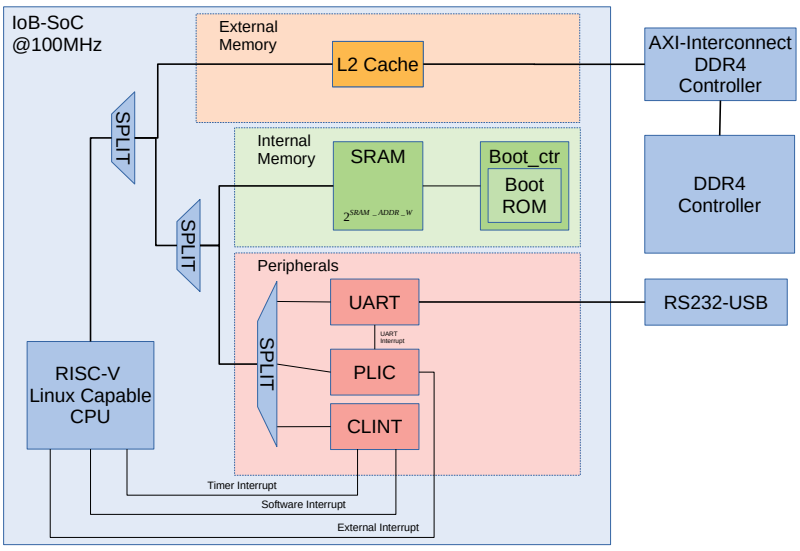

Figure 4.1: Developed SoC sketch.

Comparing figure 4.1 with the original design of *IOb-SoC* (figure 2.1) we can see that there were a few additional alterations. In the first place, it can be seen that the L1 Cache was removed. Since every application processor studied had a 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 slaves (the internal memory, the external memory and the peripheral bus). This meant that there were 2 selection bits, when '00' then the internal memory bus was active, when '01' it was the peripheral bus and when '10' it was the external memory. This caused a problem because when addressing the external memory if its size is bigger than 1GB the selection bits would be '11'. The desmultiplexer (DEMUX) output selected by '11' is not connected to anywhere, so this caused an internal hardware error. The solution was to include two *iob-split* modules each one with two slaves. The first would chose between the external memory and either the internal memory or peripheral bus. The second would chose between the internal memory and the peripheral bus. Another advantage of using this method is that now the selection bits position does not vary depending on if we are using the DDR or not. This makes it easier to use external software that does not make use of the *iob-soc* Makefiles. Before the peripheral addressing on external software had to be changed every time the developer wanted to test with or without the external memory.

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

4.1 Central Processing Unit

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

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

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

Listing 4.1: Generate *verilog* from *SpinalHDL*

The *VexRiscv* can be configured by adding and removing plugins. Plugins are hardware components described in *SpinalHDL* that can be reused in different designs by simply adding "new Plugin_Name(...)," to the plugins list in the top CPU description file. The existing plugins are described in the *VexRiscv* repository on the "src/main/scala/vexriscv/plugin" directory. Looking at the available plugins it can be seen that there are two different plugins for the instruction bus and data bus each. They are "IBusSimplePlugin", "IBusCachedPlugin", "DBusSimplePlugin" and "DBusCachedPlugin". The difference is that the cached plugins have the L1 Cache integrated, while on the simple plugins does not. An additional different between the data cached and simple plugin it that, although the "DBusCachedPlugin" fully supports the RISC-V atomic extension, the "DBusSimplePlugin" supports 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, but since I do not understand how to code in *SpinalHDL* it would be very time consuming.

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

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

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

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

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

4.1.1 VexRiscv Wrapper

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

The input signals of *iob_vexriscv* are: the clock signal which is the system clock derivative from the development board where the SoC is implemented; the reset signal which is set to high (’1’) when the system reboots and when the stage 0 bootloader finishes; the boot signal that has the value ’1’ while the stage 0 bootloader is executing, after it finishes the boot signal

value drops to '0' at the same time the reset signal is set to high; the instruction bus response signal that is connected to "cpu_i_resp"; the data bus response signal that is connected to "cpu_d_resp"; the timer interrupt and software interrupt signals which are set to '1' or '0' by the CLINT unit; the external interrupt signal which is controlled by the PLIC unit. The output signals are the instruction bus request signal and the data bus request signal, which connect to the "cpu_i_req" and "cpu_d_req" respectably. The "cpu_i_resp", "cpu_d_resp", "cpu_i_req" and "cpu_d_req" signals were reviewed in section 2.1.

The input signals of the *VexRiscv* core are: .... The output signals of the *VexRiscv* core are: ....

After understanding the inputs and outputs of each module it is easy to see which wire should be connected to each other. But after connecting all the wires there were two problems. The fist being that the *IOb-SoC* internal bus did not contain all the signals that were needed by the *VexRiscv* core. In order to successfully make the handshake with the *VexRiscv* core an instruction and data request ready signal had to be generated. .... **How was it generated?**

valid_reg	valid	resp_ready	req_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 4.1: First try at identifying the rules the req_ready should follow.

$$(valid\_reg \cdot resp\_ready) + (valid \cdot \overline{valid\_reg} \cdot \overline{resp\_ready}) \tag{4.1}$$

valid_reg	resp_ready	req_ready
0	0	1
0	1	N/A
1	0	0
1	1	1

Table 4.2: Simplified truth table.

$$(valid_reg \cdot resp_ready) + (\overline{valid_reg} \cdot \overline{resp_ready}) = valid_reg \odot resp_ready \tag{4.2}$$

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

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

4.2 *UART 16550*

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

In the RISC-V Platform Specification [14] it is defined that every embedded Operating System (OS) is required to have a UART port implementation that is register-compatible with the industry standard *UART 16550*, which was studied in chapter **??**. The *UART 16550* already exists for a long time, it was released by *National Semiconductor* in 1987, and is present and supported by a large number of software and hardware. The *UART 16550* is often used connected to an RS-232 interface and in this project the development boards used are connected through RS-232 to the computer.

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*[8] that has been made available by *freecores* on *github*. This UART was written in verilog, although it was a older version of verilog it is still synthesizable by modern tools and easy to understand. The *UART16550* core used implements a wishbone interface to interact with the System on a chip (SoC). Similarly to what was done with the CPU I had to create a wrapper to adapt the core to the *IOb-SoC*.

4.2.1 *UART 16550* Wrapper

The whishbones

4.3 CLINT Unit

CLINT

4.4 PLIC Unit Wrapper

PLIC

4.5 UUT Top Hardware

This top module creates a *verilog* wrapper of the Unit Under Test (UUT) that allows it to interact with the different hardware logic simulators. The top module file is an adaptation of the previous *verilog* file used on icarus simulation.

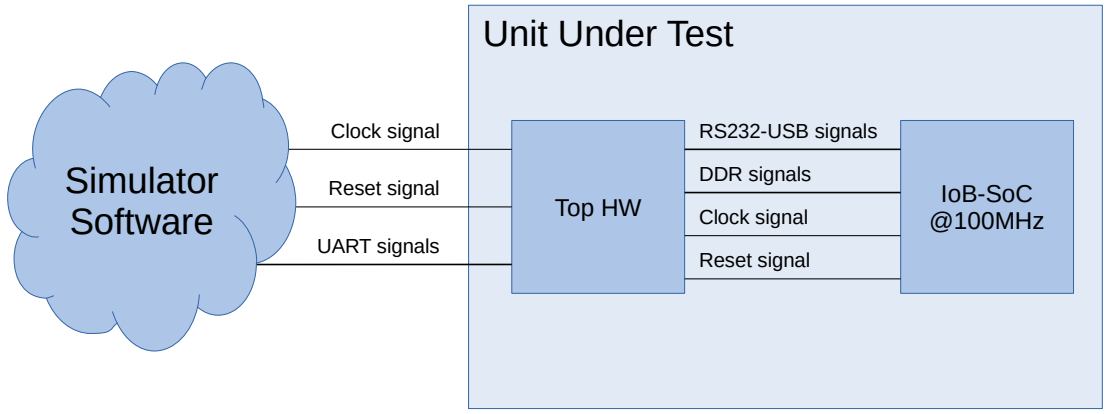

Figure 4.2: Simulated hardware interfaces.

5 Software Developed

5.1 Python Console

5.2 Verilator Testbench

5.3 Barebones Interrupt Routine

5.4 IOb-SoC Linux Stage 0 Bootloader

5.5 IOb-SoC on OpenSBI

5.6 IOb-SoC Device Tree

5.7 IOb-SoC Linux *'rootfs'*

5.8 Makefiles

6 Products of the expedition, AKA Project Results

6.1 FPGA Resources Consumption

6.2 Run/Boot Linux Performance

7 Contributed Repositories

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

8 Conclusions

8.1 Achievements

8.2 Future Work

A Annex

A.1 Annex 1

Annex 1

A.2 Annex 2

Annex 2

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] 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/>.

[25] 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/>.

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

[27] 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.

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