Development Environment for a RISC-V Processor

António Pedro Charana e Silva
Electrical and Computer Engineering Department
Instituto Superior Técnico
Lisbon, Portugal
antonio.pedro.silva@tecnico.ulisboa.pt

***Abstract*—Central Processing Unit (CPU) based systems are complex systems which took several years to develop and needed extraordinary amounts of capital investment. For a long time, only large companies could afford creating these Systems On Chip (SoCs), where many of the components are licensed from other companies who serve many other customers. Recently, thanks to a large open source community, it is also becoming possible for smaller companies to build their own SoCs using free and high quality hardware and software components. These are typically available in repositories hosted in web-based platforms like GitHub, Gitlab or Bitbucket. The largest initiative so far to develop an open source processor and respective ecosystem is arguably the RISC-V Instruction Set Architecture (ISA), whose ambition is to become the standard ISA for all computing devices, from microcontrollers to supercomputers. This dissertation presents a development environment to create open source SoCs that use the RISC-V processor architecture. A base SoC called IObSoC is created which can be easily edited to create more complex SoCs. The IObSoC hardware is written in Verilog and the software is written in C. It uses a Makefile tree and scripts written in Python, Tcl and Bash to enable simulation, synthesis and place and route with various tools, free and commercial, for both Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) development flows.**

***Index Terms*—RISC-V Instruction Set Architecture, PicoRV32 processor, Open Source, System On Chip, Internet of Things.**

I. Introduction

With the advent of the Internet of Things (IoT) and portable devices, there has been a high demand for low power CPU-based which need to meet stringent area, battery life, power and performance specifications. Having just one or two providers of CPU cores, such as ARM, cannot satisfy the enormous appetite of many smaller businesses and consultancies who may – and effectively can – provide services in this domain. Therefore, open source processor cores are the way to go since the CPU is one of the most complex parts of the system but is also a commodity which, by itself, adds little value to the final product. In this context, after previous attempts such as the OpenRISC project, the RISC-V ISA has finally emerged and promises to become for hardware what the Linux operating system is for software.

Having access to a complete enough set of hardware and software building blocks to build useful systems is either too expensive or time consuming, and constitutes a huge barrier for start-ups and smaller businesses. The IoT market is competitive and forces companies to produce high quality hardware and software systems in a short window of time and with a low budget.

To solve this problem, this work proposes a base SoC equipped with a CPU, memory system and serial communication, which can be easily customized by users to implement more complex and specific SoCs. The IObSoc hardware is written in Verilog and the software is written in C. It uses a Makefile tree and scripts written in Python, Tcl and Bash to enable simulation, synthesis and place and route with various tools, free and commercial, for both FPGA and ASIC development flows.

This project was entirely developed at company IObundle Lda and is the continuation of the work in [1] and [2], which were previous attempts to build a SoC with open source components.

II. The RISC-V ISA

There are two types of ISA: Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). The performance debate between both types of ISA is irrelevant nowadays [3][4], but the only CISC ISA still used today is Intel's proprietary x86, mainly for compatibility with legacy systems. There are, however, several free and proprietary RISC ISAs.

The main free (to use) ISA nowadays is RISC-V, a reduced instruction set architecture that allows standard and custom extensions of its base ISA. Open source hardware designs and software toolchains are now vastly available for download by any interested individual or organization. Large companies such as NVidia and Western Digital Corp. have already decided to use RISC-V in their designs [5].

1) Unprivileged ISA: At the time of writing of this document, the latest version of the RISC-V unprivileged architecture is described in [6] and its status is summarized in page *i* of its preface. It supports several base ISAs and several standard extensions, but it is also possible to build custom extensions to create, for example, Graphics Processing Unit (GPU) based instruction sets.

The base modules of the RISC-V ISA are:

- **RV32I**, **RV64I** and **RV128I**, the base ISAs for 32-bit, 64-bit and 128-bit processor implementations, respectively;
- **RV32E**, the base ISA, useful for embedded systems;
- **RISC-V Weak Memory Ordering (RVWMO)**, the RISC-V default memory consistency model.

The most important RISC-V extensions for this project are:

- **M**, for integer multiplication and division instructions;

- **F**, for single-precision floating-point instructions;
- **C**, for compressed instructions;

The base RISC-V ISA (RV32I) has 32-bit-wide instructions. Because RISC architectures can cause large program sizes, RISC-V provides the C extension, which uses compressed versions of the regular instructions with 16 bits. RISC-V's compressed ISA (or simply RVC) is a quite important feature in IoT because it reduces code size by around 25% [7] while allowing regular and compressed instructions be mixed in a single program, improving CPU energy efficiency.

RVC can be used alongside the RV32E base ISA, which uses only 16 registers (instead of RV32I's 32 registers) and disables floating point instructions (substituted by software routines). This combination is useful in low-power and embedded systems, as it reduces energy consumption and area.

2) Privileged ISA: There is also a privileged level instruction set in the RISC-V ISA [8] for privileged instructions and other functionalities required for Operating System (OS) support and attaching external devices. However, these are out of the scope of this project for now.

III. System on Chip Development

A. Architecture

An SoC is an electronic system completely incorporated in a single Integrated Circuit (IC). SoCs can be implemented as an ASIC or on a programmable platform such as an FPGA. An SoC typically consists of one or more CPUs controlling other hardware peripheral components.

1) Master-slave paradigm: Digital hardware systems usually follow the master/slave paradigm, in which there are components (masters) that control others (slaves). In SoCs, CPUs can be both masters or slaves of other CPUs or devices. Devices like memory units, accelerators or communication interfaces are normally slaves.

2) Valid-ready handshake protocol: A handshake protocol is needed to control data flow between masters and slaves in digital systems. A widely used standard is the **valid-ready handshake protocol**, which uses two 1-bit signals called **valid** and **ready**. `ready` can depend on `valid`, but the opposite must not happen. In any data transaction, the master asserts `valid` when data is valid to send/receive. The slave then asserts `ready` (if it has not yet done so) and the data transaction begins, after which both `valid` and `ready` are de-asserted.

3) SoC components: The following are examples of components that make up an SoC:

- **CPUs** and **GPUs**;
- **Memory units**, such as Random-Access Memory (RAM), Read-Only Memory (ROM), Double Data Rate (DDR) and Flash;
- **Memory controllers**, such as Dynamic Random-Access Memory (DRAM) controllers and Direct Memory Access (DMA) modules;
- **On-chip module interconnection and communication interface** such as an Advanced eXtensible Interface (AXI) Interconnect [9] [10] or other custom native interface controllers;
- **Serial protocol interfaces/controllers**, such as Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Ethernet and Universal Serial Bus (USB);
- **Co-processors**, such as floating-point arithmetic modules;
- **Coarse-Grain Reconfigurable Arrays (CGRAs)**, such as IObundle's Versat [11];

CPUs control peripherals by running software programs that write to and read from them. Peripherals are added to each CPU according to each application's specificities. A simple way for a CPU to communicate with other external electronic systems is through serial interfaces because they are easy to implement. For example, a UART can be used to send/receive char bytes to/from a computer and an SPI module can be used to access an SPI flash memory unit, which is the cheapest kind of off-chip non-volatile memory nowadays.

4) SoC intermodule communication: Traditionally, SoC components communicate with each other via electrical bus infrastructures. This kind of intermodule communication is not scalable in IoT because area and power consumption greatly increase with the number of SoC components. A solution to this problem is a Network On Chip (NoC) infrastructure, based on network protocols such as the Transmission Control Protocol (TCP) and the Internet Protocol and routing algorithms such as Dijkstra's, which greatly reduces power consumption and silicon area occupied by wires and also improves communication throughput and latency [12].

B. RISC-V CPU Architectures

Several RISC-V CPUs were considered to build IObSoC:

- **Rocket Chip** [13], a synthesizable SoC Register-Transfer Level (RTL) generator made by the same team that introduced RISC-V. It uses the Rocket or the BOOM 32-bit and 64-bit CPU generators and has yielded functional silicon prototypes that boot the Linux OS;
- **Taiga** [14], a high-performance RISC-V softcore, developed mainly for FPGAs and supporting multicore configurations using an OS.
- **PULPino** [15], a single core microcontroller system that can use two RISC-V CPUs: RI5CY [16], which implements the RV32IFMC ISA and a subset of the privileged-level of RISC-V, or zero-riscy [17] which implements the RV32IMC and the RV32E ISAs.
- **PULP** [18] is an advanced microcontroller architecture which consists of an more complete and complex system than PULPino. It can use a RI5CY or a zero-riscy CPU as main core;
- **PicoRV32** [19], a size-optimized RISC-V processor core that implements the RV32IMC and RV32E ISAs with a high maximum clock frequency;

Rocket Chip has a great amount of features but it has been found too hard to manipulate in the past at IObundle, so it was

put aside. Taiga was also put aside as it is solely optimized for FPGAs, thus not being suited for ASICs. PULPino proved hard to detach from its environment in the past at IObundle and its features go beyond the scope of this project. PULP is not yet implemented on FPGA.

PicoRV32 is easy to deal with and well documented, with several validated examples available, such as Raven [20], an ASIC SoC. PicoRV32 has been chosen for this project given its simplicity and its complete repository, which features an example SoC provided that runs code directly from an SPI flash memory chip and can be used as a simple controller in FPGA or ASIC designs.

C. Tools

In order to successfully design an SoC for an ASIC or FPGA, several software tools must be used to build the software and hardware components. Most of these tools are commercial although effective open source tools are becoming more and more popular.

1) Software toolchain: In order to produce software for a RISC-V CPU, both the GNU [21] and LLVM toolchains [22] have added support for the RISC-V architectures. These toolchains offer compilers, profilers, debuggers, assemblers, linkers, etc.

2) RTL simulators: RTL simulation is an essential step for verifying the correctness of the design. NCSim [23] from Cadence and ModelSim [24] from Mentor are well known commercial RTL simulators which are supported in this work. Free and open-source RTL simulators such as Icarus Verilog [25] and Verilator [26] also exist.

3) FPGA implementation tools: After successfully simulating a system's RTL code, the FPGA emulation phase can begin. The largest FPGA manufacturers are Xilinx and Intel. Both provide proprietary software suites for RTL synthesis and FPGA implementation tools for their own FPGAs: Xilinx's Vivado [27] and Intel's Quartus [28], which require paid licenses that are provided under different commercial options (such as when acquiring their FPGA development boards).

4) Toolchain integration: In order to integrate all the necessary tools to design SoCs in a single environment, a build automation tool such as Make and FuseSoC can be used. Make is a well known program for Unix-based OSs that can call different tools to build an SoC according to user defined rules that set dependencies to avoid rerunning time-consuming programs if they have already been ran.

FuseSoC [29] works as a package manager for reusable hardware blocks and as a toolchain integration mechanism. FuseSoC's operation revolves around core description files, which configure the Hardware Description Language (HDL) sources and the tools necessary to compile, simulate and/or synthesize a core. The PicoRV32 repository supports both Makefile and FuseSoC flows, which are executed in a terminal.

IV. IOBSOC ARCHITECTURE

IObSoC [30] is a new 32-bit SoC using the PicoRV32 [19] RISC-V CPU architecture and it is presented for the first time in this document. At the time of writing, it is implemented on three different FPGA board models – Xilinx Kintex UltraScale KU040 Development Board [31], Spartan-6 FPGA SP605 Evaluation Kit [32] and Cyclone V GT FPGA Development Kit [33] – and is currently undergoing ASIC implementation in the UMC 130 nm process at IObundle.

This document details IObSoC's implementation on a Xilinx XCKU040-1FBVA676 FPGA with -1 Speed Grade, which is hosted by an AES-KU040-DB-G Xilinx Kintex UltraScale Development Board manufactured by Avnet [31], as it is the most complete implementation at the time of writing because, unlike the other two mentioned FPGA boards, it features data access and code execution from an external DDR memory chip. Throughout the rest of this document, the AES-KU040-DB-G board will simply be referred to as **KU040 board** for readability and writing simplicity.

IObSoC has several operating modes, each one having its own set of hardware and software components, which are included or not in the SoC by commenting/uncommenting the `USE_BOOT` and `USE_DDR` Verilog macros, which enable the use of the bootloader and the DDR Memory, respectively. Table I shows how SoC operating modes relate to these macros.

TABLE I
IOBSOC OPERATING MODES.

Boot	DDR	Operating mode
No	No	App. pre-loaded on SRAM
No	Yes	App. pre-loaded on DDR (simulation only)
Yes	No	App. loaded to SRAM from UART
Yes	Yes	App. loaded to DDR from UART

IObSoC's schematic is presented in Figure 1. Its board-dependent wrapper for the KU040 board is shown in Figure 2, where blue, red and yellow blocks/lines represent components/connections that are used on all SoC operating modes, when the DDR Memory is used and when it is not, respectively. The SoC's several operating modes widens the range of contexts and applications supported by it, which facilitates meeting various customers' needs.

A. SoC Input/Output Ports

IObSoC's has two input ports for the clock and reset signals and an output port for the CPU's trap signal. It also has four additional ports which are the two serial lines of its UART and two flow control signals Request-To-Send (RTS) and Clear-To-Send (CTS). The SoC may also have ports for an AXI4 Interface [9] [10] which is used to communicate with the DDR when it is used.

B. Clock Scheme

The Differential Clock Oscillator feeds either the DDR Controller (when the DDR is used) or the Clock Controller (otherwise) with two differential 250 MHz clock signals. The Clock Controller then produces IObSoC's clock signal.

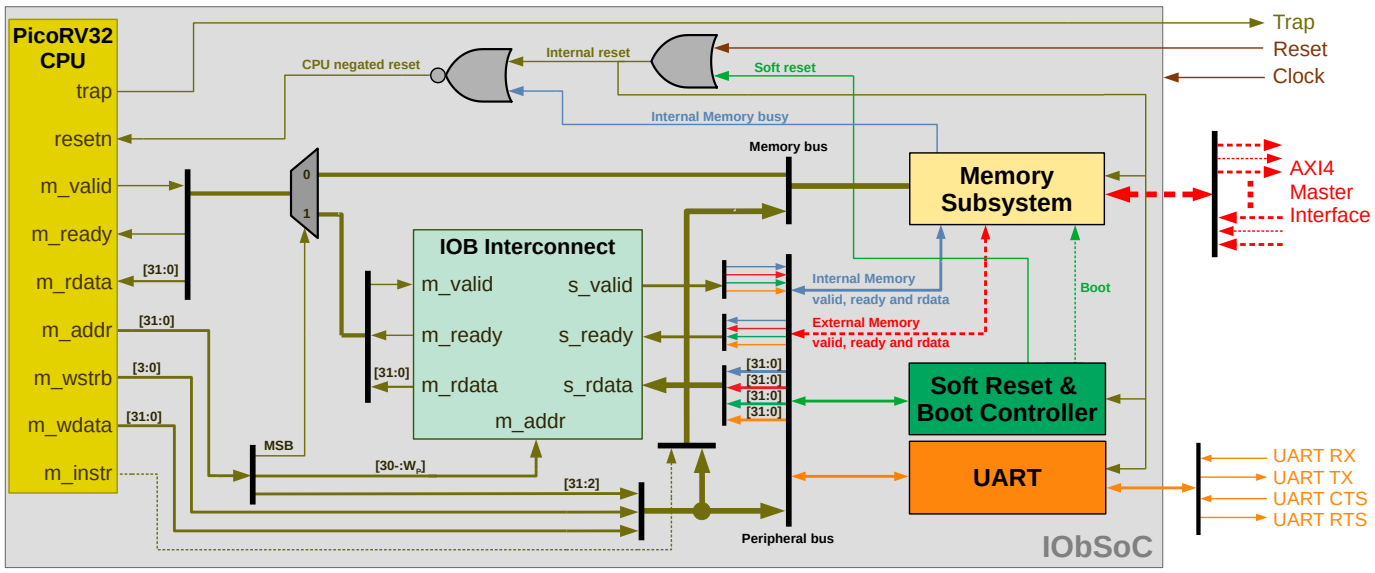

Fig. 1. IObSoC's top level schematic.

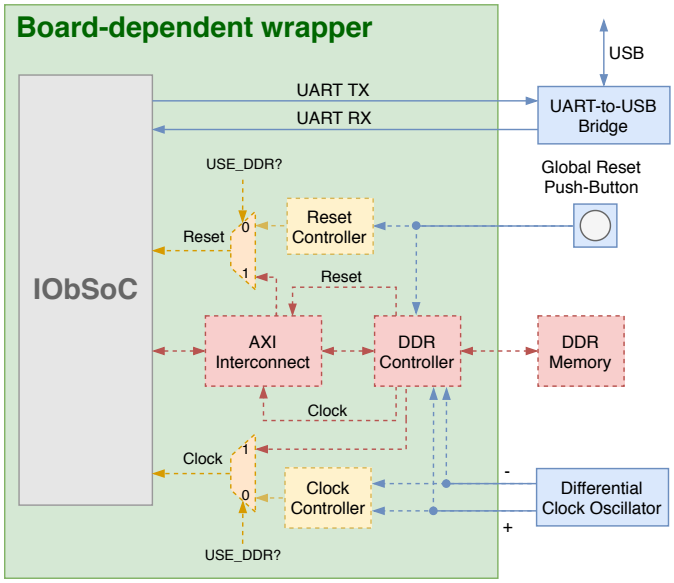

Fig. 2. IObSoC's board-dependent wrapper and external components.

The DDR Controller generates the AXI Interconnect's and the DDR's internal clocks and the AXI Interconnect's master side and slave side (IObSoC) reference clocks.

C. Reset Scheme

1) External reset: IObSoC's reset signal is either provided by the DDR Controller (when the DDR is used) or the Reset Controller (otherwise) and it is triggered on SoC start-up (after loading the bitstream to the FPGA) or by pressing the Global Reset Push-Button (hard reset for emergency situations).

When using the DDR, the DDR Controller first calibrates the DDR and then resets the AXI Interconnect, which then generates a negated reset pulse that is passed through an inverter and fed into IObSoC's reset port, resetting it. This makes IObSoC restart the bootloader and resets the board-dependent wrapper components (equivalent to a SoC start-up).

2) Soft reset: IObSoC can also be internally soft reset via a software write to an address of the Soft Reset & Boot Controller. The soft reset's is used to reset the SoC after the bootloader finishes loading the application to the Main Memory.

3) IObSoC's internal reset: Inside IObSoC, the external reset input and the internal soft reset signals are fed into an OR gate to produce the internal reset signal, used to reset the CPU and its peripherals. The CPU is also reset while the Boot ROM is copying its contents to the Static Random-Access Memory (SRAM).

D. Address Scheme

IObSoC is a memory-mapped system, i.e., each register and memory word in the SoC has an unique address, which the CPU uses to write or read. IObSoC's address scheme is illustrated in Figure 3.

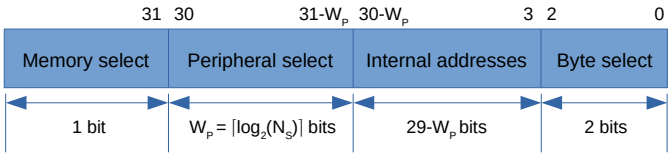

Fig. 3. IObSoC address scheme.

Each address is 32-bit wide and corresponds to a memory byte, which accounts for a total of 4 GB of addressable memory space. If N_S is the number of slaves/peripherals in the SoC, we define W_P as

$$W_P = \lceil \log_2(N_S) \rceil \quad (1)$$

The most significant bit of the address indicates whether the CPU is selecting the Main Memory (value 0) or a peripheral (value 1). When this bit is asserted, the next W_P most significant bits of the address bus indicate which peripheral is selected by the CPU, as each peripheral corresponds to a single combination of these bits – called **prefix**. The remaining bits of the address bus (except the two less significant bits) are the internal addresses of the memories/peripherals. The two less significant bits of the address bus specify a byte from a 32-bit word, but instead the SoC uses the CPU's write strobe for 1-byte or 2-byte reads/writes.

Table II contains IObSoC's peripheral prefixes, while Table III reveals its memory map. The SRAM and the Cache can be both accessed as Main Memory or peripherals (necessary for the bootloader).

TABLE II
IOBSOC'S PERIPHERALS' PREFIXES ASSUMING $W_P = 2$.

Peripheral's name	Prefix
UART	00
Soft Reset & Boot Controller	01
Cache	10
SRAM	11

TABLE III
IOBSOC'S MEMORY MAP ASSUMING $W_P = 2$ AND THE PERIPHERALS' PREFIXES IN TABLE II.

Peripheral or memory	Address space
Main Memory (memory bus)	0x00000000 – 0x7FFFFFFF
UART	0x80000000 – 0x9FFFFFFF
Soft Reset & Boot Controller	0xA0000000 – 0xBFFFFFFF
Cache (peripheral bus)	0xC0000000 – 0xDFFFFFFF
SRAM (peripheral bus)	0xE0000000 – 0xFFFFFFFF

E. Native Interface

The set of signals used to connect the CPU to the memory subsystem and peripherals are what is called the SoC's **native interface**. These signals are the PicoRV32 CPU's input/output ports:

- **wdata** and **rdata**: the CPU's 32-bit write and read data buses;
- **instr**: which indicates if the CPU's read data bus holds an instruction or program data;
- **addr**: the CPU's 32-bit address bus;
- **wstrb**: the CPU's 4-bit write strobe, which is used to control data size on write operations (1, 2 or 4 bytes);
- **valid** and **ready**: which implement a valid-ready handshake protocol between the CPU and its slaves.

The CPU connects to the memory subsystem via the **memory bus** and to the peripherals via the **peripheral bus**. `wdata` and `wstrb` are CPU outputs broadcast to both memory and peripheral buses, while `valid`, `ready` and `rdata` are selected from one of the buses with the most significant `address` bit. `instr` is only used in the memory bus (necessary for the cache in the memory subsystem).

F. IObSoC Components

IObSoC has mandatory and optional components. The first ones are used in all SoC operating modes, while the second ones are not. The mandatory components are the CPU, the Boot ROM, the SRAM, the UART, the Soft Reset & Boot Controller and the IOB Interconnect. The optional components are the Cache [34] and the Native-to-AXI adapters (which are not depicted in Figure 1).

1) CPU: The (single) master in the SoC that controls the peripherals and the memory subsystem (slaves) via software programs stored in the Main Memory. It is a PicoRV32 [19] core, a size-optimized open source processor with high maximum frequency that implements the RV32IMC and RV32E ISAs. IObSoC's CPU code is stored in IObundle's IOb-RV32 repository [35], which is a fork of PicoRV32's repository.

2) Memory Subsystem: It contains a Boot ROM that stores the bootloader, an internal SRAM to run programs from and an IOb-Cache [34] module to access an external DDR. The bootloader is copied to the final part of the SRAM on SoC start-up. Depending on the `USE_BOOT` configuration macro, the SRAM then executes either the pre-loaded application in it or the bootloader, loading the application to the Main Memory via UART, soft resetting the SoC and then running the loaded application from the Main Memory.

The SRAM and the Cache can be accessed from both memory and peripheral buses (necessary for the bootloader). When loading the application to the SRAM, the bootloader copy on its final part becomes overwritten by the heap and stack sections of the application. The Cache connects to the CPU via native interface and to the AXI Interconnect via an AXI4 interface, which then further connects to the DDR.

3) UART: A peripheral that implements the RS-232 serial communication protocol whose purpose is to serve as an interface between IObSoC and the outside world. It is an IObUART [36] module and is mainly used to receive a software application from a computer during the bootloader sequence and to transmit print messages from IObSoC to a computer terminal. It features RTS/CTS data flow control and dedicated C drivers were developed to operate it.

4) Soft Reset & Boot Controller: The Soft Reset & Boot Controller is a peripheral used to soft reset the SoC components and swap the source of the CPU's program instructions after booting is complete. When the CPU writes to this peripheral, it edits the **boot register** inside of it and uses a counter to generate a reset pulse that resets the CPU and its peripherals. After a soft reset, the CPU executes a program from the memory indicated by the boot register.

5) IOB Interconnect: A parametrized bus switch that manages the valid-ready handshake protocol between the peripherals and the CPU. It multiplexes the N_S peripherals' `rdata` and `ready` signals and forwards only the selected one's

to the CPU. Likewise, it demultiplexes the CPU's `valid` signal and forwards it only to the selected peripheral, while de-asserting the others.

6) Native-to-AXI adapters: An optional interface module that can be used to connect peripherals with AXI4-Lite ports [9] [10] in IObSoC. It has a side with native interface ports – connecting to the SoC's peripheral bus – and a side with AXI4-Lite ports – interfacing with the AXI peripheral –, thus converting both sets of signals into the other.

G. Board-Dependent Wrapper Components

The **board-dependant wrapper** is the system emulated inside the FPGA. It contains IObSoC and other components (that depend on the SoC's operating mode) to feed it its clock and reset signals and connect it to the DDR Memory.

1) Clock Controller: A component used to generate IObSoC's clock signal when the DDR is not used. It uses the Differential Clock Oscillator's 250 MHz differential clocks to generate IObSoC's 100 MHz clock with 50% duty-cycle via an internal Phase-Locked Loop (PLL) [37] while also providing input and output buffering for a cleaner clock signal. It was generated with Vivado's Clocking Wizard.

2) Reset Controller: An hardware counter with additional logic that generates IObSoC's reset signal when the DDR is not used.

3) AXI Interconnect: A Xilinx component used to connect IObSoC to the DDR Controller via an AXI4 interface when the DDR is used. It also provides IObSoC's reset signal, although it originates from the DDR Controller.

4) DDR Controller: A Xilinx Memory Interface Generator (MIG) core that generates IObSoC's (and others components') clock and reset signals and allows it to access the DDR. It connects to the AXI Interconnect via an AXI4 interface and to the DDR4 memory chip via a DDR4 Synchronous Dynamic Random-Access Memory (SDRAM) interface [31]. It calibrates the DDR on SoC start-up.

H. External Components

Physical components on the KU040 board are necessary for operating the SoC. All of the following are mandatory except the DDR.

1) DDR Memory: A DDR SDRAM chip controlled by the DDR Controller. The DDR4 interface is implemented with two 512 MB Micron EDY4016AABG-DR-F devices [31] [38]. It can be used by IObSoC as Main Memory.

2) UART-to-USB Bridge: A chip that converts RS-232 to USB and vice-versa, thus allowing IObSoC's UART and a computer to communicate in both directions.

3) Differential Clock Oscillator: A Silicon Labs Si510/Si511 or compatible device [31] that generates the board's differential clocks and feeds them to the board-dependent wrapper via the FPGA's pins.

4) Global Reset Push-Button: A physical push-button connected to the board-dependent wrapper via an FPGA pin used to manually hard reset the SoC in emergency situations.

I. SoC Operating Modes

1) Application pre-loaded on SRAM: This operating mode corresponds to the first line of Table I and it uses the blue and yellow components in Figure 2.

A 100 MHz clock signal is fed to IObSoC by the Clock Controller, while the reset signal is provided by the Reset Controller. IObSoC's AXI4 interface and Cache are not used. The application is pre-loaded on the SRAM (Main Memory) and is incorporated in the bitstream.

After loading the bitstream onto the FPGA, IObSoC copies the bootloader to the SRAM (but does not execute it, as the boot register is initialized with 0). It then executes the application stored in the SRAM.

2) Application pre-loaded on DDR (simulation only): This operating mode corresponds to the second line of Table I. It does not use the board-dependent wrapper, but instead uses a Verilog testbench for RTL simulation. The external DDR is substituted by an AXI SRAM [39] on the testbench, directly connected to the AXI4 master interface of IObSoC – the Unit Under Test (UUT).

IObSoC's clock and reset inputs are fed by the testbench and its Cache and AXI4 interface are used to access the AXI SRAM (the Main Memory), which is pre-loaded with the application. The boot register is initialized with 0, so the bootloader is not used. When the simulation starts, IObSoC is reset, the bootloader is copied to the SRAM and then the application stored in the AXI memory is executed.

3) Application loaded to SRAM from UART: This operating mode corresponds the third line of Table I and it uses the blue and yellow components in Figure 2.

A 100 MHz clock signal is fed to IObSoC by the Clock Controller, while the reset signal is provided by the Reset Controller. IObSoC's AXI4 interface and Cache are not used.

After loading the bitstream onto the FPGA, the bootloader is copied to the SRAM and executed, as the boot register is initialized with 1. It loads the application to the initial part of the SRAM (Main Memory) via UART and soft resets the SoC, while updating the boot register to 0. After the soft reset, IObSoC executes the application from the SRAM.

4) Application loaded to DDR from UART: This operating mode corresponds the fourth line of Table I and it uses the blue and red components in Figure 2.

A 100 MHz clock signal is fed to IObSoC by the DDR Controller, while the reset signal is provided by the AXI Interconnect. IObSoC's AXI4 interface and Cache are used.

After loading the bitstream to the FPGA, the DDR Controller calibrates the DDR and resets the AXI Interconnect, which then starts-up the SoC. The bootloader is copied to the SRAM and is executed, as the boot register is initialized with 1. It loads the application to the DDR via UART and soft resets the SoC, while updating the boot register to 0. After the soft reset, IObSoC executes the application from the DDR.

V. IObSoC Development Environment

IObSoC is a base SoC which can be further developed by adding new peripherals which the application firmware then

operates via software drivers to accomplish some goal.

A. Adding Peripherals

1) Creating the peripheral: The peripheral must be designed with the Verilog HDL and its ports must be compatible with IObSoC's native interface or with the AXI4-Lite interface [9];

2) Editing IObSoC's configuration file: Next, IObSoC's configuration file, located in `rtl/include/system.vh`, must be edited according to the following steps:

- Comment/uncomment the `USE_BOOT` and `USE_DDR` macros, depending on the desired SoC operating mode;
- Configure `MAINRAM_ADDR_W` – the SRAM's address width when used as Main Memory –, depending on the size of the application (not needed for DDR applications);
- Increment `N_SLAVES` (i.e., the number of slaves/peripherals N_S);
- Increment `N_SLAVES_W` (i.e., W_P given by Equation 1);
- Add a base address for the new peripheral;
- Uncomment/comment `USE_LA_IF` to use or not PicoRV32's Look-Ahead (LA) memory interface, respectively (simulation only).

The following code exemplifies how to configure IObundle's timer Intellectual Property (IP) – IObTimer [40] – in IObSoC's configuration file:

```
//Comment/uncomment to choose IObSoC's operating mode
//`define USE_BOOT
//`define USE_DDR

//main memory address space (log2 of byte size)
`define MAINRAM_ADDR_W 15

//SLAVES
`define N_SLAVES 5 // EDITED: was 4 before

//bits reserved to identify slave
`define N_SLAVES_W 3 // ceil(log2(N_SLAVES)) // EDITED: was
      2 before

//peripheral address prefixes
`define UART_BASE 0
`define SOFT_RESET_BASE 1
`define DDR_BASE 2
`define SRAM_BASE 3
`define TIMER_BASE 4 // EDITED: added this line

//use CPU lookahead interface
//`define USE_LA_IF

...
```

3) Instantiating the peripheral in IObSoC: Next, IObSoC's top source file, located in `rtl/src/system.v`, must be edited. The following code exemplifies how to instantiate IObTimer in IObSoC's top source file:

```
time_counter #(.COUNTER WIDTH(32))
timer (
        .rst     (reset_int),
        .clk     (clk),
        .addr    (m addr[2]),
        .data_in (m_wdata),
        .data_out(s_rdata[`TIMER_BASE]),
        .valid   (s_valid[`TIMER_BASE]),
        .ready   (s_ready[`TIMER_BASE])
);
```

If the peripheral has AXI4-Lite ports, a native-to-AXI adapter must be instantiated between the peripheral and the SoC's peripheral bus.

4) Firmware: Next, the peripheral's software drivers and the SoC's firmware must be written:

- Write one or more C source and header files with the software drivers to operate the peripheral;
- Include the header file(s) in the firmware source file, located in `software/firmware/firmware.c` and develop the software application;
- If not using the DDR, remember to edit the `MAINRAM_ADDR_W` macro in IObSoC's configuration file so that the SRAM is large enough to store the firmware;

5) Editing the Makefiles and the FPGA tools' scripts: Next, the Makefiles and FPGA tools' scripts in IObSoC's repository need to be updated with the new C and Verilog source and header files (or respective include directories). The Makefiles and FPGA tools' scripts to edit are:

- `software/firmware/Makefile`;
- `$SIM_DIR/Makefile`;
- `$FPGA_DIR/Makefile`;
- `$FPGA_DIR/<script>`;

where `$SIM_DIR` and `$FPGA_DIR` must be defined in the top Makefile located in the root directory of IObSoC's repository, depending on the desired tools.

B. Software

The software executed in IObSoC is written with the C programming language and compiled with the GNU RISC-V GCC cross-compiler [21]. The software used inside the SoC is the bootloader (in `software/bootloader`) and the firmware (in `software/firmware`). The firmware can easily be tested while running on an FPGA because it is possible to load a new firmware binary file to the SoC via UART without needing to recompile the bitstream.

The console program used by the computer to interact with IObSoC inside an FPGA is also written in C, but compiled with the regular GNU Compiler Collection (GCC) compiler on a Linux computer. It is stored in `software/ld-sw`.

C. RTL Simulation

At the time of writing, the available RTL simulators for IObSoC are Icarus Verilog [25], NCSim [23] and ModelSim [24]. To run an RTL simulation, first edit `SIM_DIR` in the top Makefile (for example, to use Icarus, define `SIM_DIR = simulation/icarus`). Then `cd` to the repository's root and run `make sim`. This also compiles the bootloader and the firmware before the simulation starts.

D. FPGA Emulation

The available RTL synthesis and FPGA implementation tools for IObSoC are Intel's Quartus [28] and Xilinx's Vivado [27] and ISE [41]. To generate a bitstream, first edit `FPGA_DIR` in the top Makefile (for

example, to use the KU040 board with Vivado, define `FPGA_DIR = fpga/xilinx/AES-KU040-DB-G`). Then `cd` to the repository's root, `source` the FPGA tools' settings (if necessary) and run `make fpga`. If the FPGA board is hosted by a remote machine, the `fpga` target also sends the bitstream and the firmware binary file to it via Secure Copy Protocol (SCP).

Then, on the FPGA board's host machine – which can be accessed via Secure Shell (SSH) –, `cd` to the root directory of IObSoC's repository (which must also be cloned in it as well) and run `make ld-sw` to setup the console to interact with IObSoC. In a new terminal, source the FPGA tools' settings if needed and run `make ld-hw` to load the bitstream onto the FPGA.

VI. Results

A. Dhrystone Benchmark Results For RTL Simulation

To validate IObSoC and measure its performance, several RTL simulations were performed using the Icarus Verilog simulator for different configurations of IObSoC running the Dhrystone benchmark [42] [43]. The several RTL simulations performed correspond to different combinations of the following parameters:

- **IObSoC's operating mode**;
- The usage of PicoRV32's **LA memory interface**;
- **IObSoC's cache configuration**, which changed its size;
- Dhrystone's **number of runs**, i.e., the number of loop iterations of the main code.;
- The usage or not of the optimization `-O3` **GCC flag**.

IObSoC uses a 100 MHz clock signal on all simulations and does not use bootloader (`USE_BOOT` commented). However, all of IObSoC's operating modes (including those with the bootloader feature) were successfully tested and held positive results, validating the SoC.

The obtained results are shown in Tables IV and V. The results of simulations with 500 runs on Dhrystone are not presented in this document because their results' show a deviation from the 100 runs simulations less than % on the relevant performance indicators, which are:

- **Cycles per Instruction (CPI) estimate of the CPU**;
- **Dhrystones per Second (DPS) per MHz**, the number of iterations of the main code loop per second, divided by the CPU's clock frequency in MHz, rounded down;
- **Dhrystone Mega Instructions per Second (DMIPS) per MHz**, which is the DMIPS value (Equation 2) divided by the CPU's clock frequency in MHz.

1757 is the number of Mega Instructions per Second (MIPS) when running Dhrystone on the VAX 11/780, a machine with nominal 1 MIPS [44]. Dividing the results by this value and the CPU's frequency normalizes these parameters across results of several machines.

$$\text{DMIPS} = \frac{\text{DPS}}{1757} \tag{2}$$

TABLE IV
Dhrystone benchmark results on RTL simulation with 100 runs and a clock frequency of 100 MHz.

IObSoC mem. config.	CPI	DPS/MHz	DMIPS/MHz
SRAM	5.496	373	0.212
DDR (8 KB Cache)	5.507	372	0.211
DDR (256 B Cache)	7.956	257	0.146
SRAM + LA	4.063	505	0.287
DDR (8 KB Cache) + LA	4.074	503	0.286
DDR (256 B Cache) + LA	6.587	311	0.177

TABLE V
Dhrystone benchmark results on RTL simulation with 100 runs, a clock frequency of 100 MHz and the -O3 flag on GCC.

IObSoC mem. config.	CPI	DPS/MHz	DMIPS/MHz
SRAM	5.552	442	0.251
DDR (8 KB Cache)	5.564	441	0.250
DDR (256 B Cache)	7.842	313	0.178
SRAM + LA	4.095	599	0.340
DDR (8 KB Cache) + LA	4.108	597	0.339
DDR (256 B Cache) + LA	6.415	382	0.217

1) CPU: The obtained CPI when using the internal SRAM and the CPU's LA memory interface is 4.1, which coincides with PicoRV32's reference CPI. As expected from a RISC CPU, the CPI is greater than 1.

2) Cache: Performance is heavily influenced by the Cache's configuration. Using a large (8 KB) cache, DDR results are very similar to the SRAM ones because the Cache produces few read misses and the external memory is an AXI SRAM. However, using a smaller (256 B) cache increases the CPI by 41%–62% and the DPS/MHz and DMIPS/MHz decrease by 29%–38%

3) -O3 GCC optimization flag: When using this flag, code size and time are optimized and thus the number of cycles and instructions decrease. The CPI varies very slightly than when not using the `-O3` flag, while the DPS/MHz and DMIPS/MHz increase 18.5%–22.8%.

4) LA memory interface: PicoRV32's LA memory interface outputs the `address` and `valid` signals of every memory access one cycle ahead of the regular memory interface, accelerating memory accesses. The LA interface decreases the CPI by 18%–26% and increases the DPS/MHz and DMIPS/MHz by 22%–35%, being more notorious on SRAM and DDR + large (8 KB) cache configurations.

B. FPGA Implementation Results

All three IObSoC operating modes were successfully implemented on FPGA running Dhrystone with 100 runs without the `-O3` flag in GCC, which produced the results in Table VII, thus validating the SoC.

The SoC's clock frequency is 100 MHz and the LA memory interface is disabled. The DDR operating mode uses the same 8 KB cache configuration as the RTL simulation.

FPGA resource utilization results are available in Table VI. As expected, the SoC operating modes that do not use the DDR require a similar quantity of FPGA resources, while the DDR operating mode requires much more resources because it uses additional and larger components such as the Cache, the AXI Interconnect and the MIG core.

TABLE VI
FPGA RESOURCE UTILIZATION FOR EACH IOBSOC OPERATING MODE.

DDR	Boot	LUTs	36 KB BRAMs	DSPs
No	No	1787	8	4
No	Yes	1776	9	4
Yes	Yes	13049	29.5	7

TABLE VII
DHRYSTONE FPGA RESULTS WITH 100 RUNS AND 100 MHZ CLOCK.

DDR	Boot	CPI	DPS/MHz	DMIPS/MHz
No	No	5.496	373	0.212
No	Yes	5.496	373	0.212
Yes	Yes	21.004	97	0.055

SoC operating modes that run code from the SRAM produce the exact same Dhrystone results on FPGA and RTL simulation. The DDR operating mode, however, is less performant on FPGA than in simulation because the external memory in the first case is a DDR, while on the second case is an AXI SRAM, which is faster than DDR. Also, the Cache used on the FPGA for the DDR operating mode is a prior and less performant version of the Cache used in the RTL simulation tests, which does not yet work on FPGA with IObSoC.

VII. CONCLUSIONS

This document introduces IObSoC, a RISC-V SoC developed at company IObundle that uses the PicoRV32 CPU and other open-source components and can be configured to work in various operating modes. IObSoC is presented as a base SoC of a development environment for RISC-V SoCs, which includes addition of new CPU peripherals, software and hardware compilation, RTL simulation and FPGA implementation.

After considering several RISC-V CPU options, the PicoRV32 core was chosen to build IObSoC, given its simplicity and validated examples. The SoC was then built alongside its verification environment, so that it could be tested as its development advanced.

In order to validate IObSoC's operation, the SoC was tested with the Dhrystone benchmark [43], a widely used program for measuring general purpose processor performance. The tests were carried out with RTL simulations of the several SoC's operating modes using the Icarus Verilog simulator. IObSoC was also implemented on FPGA and successfully executed the Dhrystone benchmark on all operating modes.

IObSoC's development continued at IObundle after this dissertation project's conclusion and includes ASIC development flow (still underway), a new CPU-agnostic SoC architecture and possibility of using the DDR as auxiliary data storage.

A. Achievements

The first achievement of this project was a new SoC and development environment using RISC-V, the new standard open and free ISA, which has already been adopted by several large companies to design some of their systems and has great importance in the open-source community.

The second achievement is the highly configurable SoC architecture, which features several operating modes that allow it to be used in different contexts and applications, thus increasing the range of potential clients for companies who adopt it.

The third achievement is the reduction of development time and effort of new SoCs. Further development of IObSoC can be done systematically (as described in Section V) and several verification tools are already configured and ready-to-use via simple `make` commands in a terminal, which accelerates development. This allows small start-up companies to meet narrower deadlines and thus be able to compete in the digital circuit design market.

The fourth and final achievement is the reduction of resources spent on developing SoCs. By using RISC-V and open-source components and tools, acquiring expensive licenses for CPUs IPs and software tools from large companies such as ARM is no longer a necessity. Instead, free open-source CPUs – such as PicoRV32 – can be used, allowing small start-up companies to develop systems with a lower budget and thus increase their competitiveness in the digital circuit design market.

B. Future Work

IObSoC can be further developed in several future work perspectives. The first one is to implement the SoC on an ASIC, which some customers may desire if the volume of SoCs needed is high enough to make the unitary price of each unit lower than that of implementing it on several FPGAs.

The second future work perspective is add support of new CPU architectures for IObSoC. This is an advantage because PicoRV32, although being small, has a relatively higher CPI, thus being inadequate for high CPU performance applications.

The third future work perspective is to create a new SoC operating mode for running code from a flash memory unit, which many customers may be interested in because flash is a cheap, reprogrammable and widely used non-volatile solid-state memory type.

The fourth and final future work perspective is to add OS functionality to IObSoC. This is an important feature because it widens the range of supported applications by IObSoC, such as software programs that require the use of a file system. The RISC-V privileged ISA provides privileged instructions required for OSs.

REFERENCES

[1] José Sousa, Carlos Rodrigues, Nuno Barreiro, and João Fernandes. Building Reconfigurable Systems Using Open Source Components. April 2014.

[2] Luís Fiolhais and José Sousa. Warpbird: an Untethered System on Chip Using RISC-V Cores and the Rocket Chip Infrastructure. January 2018.

[3] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013.

[4] Prof. Pravin R. Lakhe. A Technology In Most Recent Processor Is Complex Reduced Instruction Set Computers (CRISC): A Survey. *International Journal of Innovative Research & Studies*, Volume 2(Issue 6), 2013. ISSN 2319-9725.

[5] The rise of RISC - [Opinion]. *IEEE Spectrum*, 55(8):18–18, Aug 2018.

[6] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019.

[7] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master's thesis, EECS Department, University of California, Berkeley, May 2011.

[8] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, June 2019.

[9] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*, 2011. [Online] Available at: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf. Accessed on March 2020.

[10] Xilinx. *AXI Reference Guide*, March 2011. [Online] Available at: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Accessed on March 2020.

[11] João D. Lopes and José T. de Sousa. Versat, a Minimal Coarse-Grain Reconfigurable Array. In *High Performance Computing for Computational Science - VECPAR 2016*, Lecture Notes in Computer Science, pages 174–187, Porto, Portugal, July 2017. Springer International Publishing.

[12] Rajeev Kamal and Neeraj Yadav. NOC AND BUS ARCHITECTURE: A COMPARISON. *International Journal of Engineering Science and Technology*, 4, April 2012.

[13] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[14] Eric Matthews and Lesley Shannon. Taiga: a Configurable RISC-V Soft-processor Framework for Heterogeneous Computing Systems Research. 2017.

[15] PULP-Platform. PULPino, May 2019. [Online] Available at: https://github.com/pulp-platform/pulpino. Accessed on March 2020. GIT code repository.

[16] PULP-Platform. RI5CY, 2020. [Online] Available at: https://github.com/pulp-platform/riscv. Accessed on March 2020. GIT code repository.

[17] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sep. 2017.

[18] PULP-Platform. PULP, January 2020. [Online] Available at: https://github.com/pulp-platform/pulp. Accessed on March 2020. GIT code repository.

[19] C. Wolf and et. al. PicoRV32 - A Size-Optimized RISC-V CPU, November 2019. [Online] Available at: https://github.com/cliffordwolf/picorv32. Accessed on March 2020. GIT code repository.

[20] Tim Edwards. Raven: An ASIC implementation of the PicoSoC PicoRV32, October 2019. [Online] Available at: https://github.com/efabless/raven-picorv32. Accessed on March 2020. GIT code repository.

[21] RISC-V Foundation. RISC-V GNU Toolchain, 2018. [Online] Available at: https://github.com/riscv/riscv-gnu-toolchain. Accessed on March 2020. GIT code repository.

[22] The LLVM Compiler Infrastructure, 2020. [Online] Available at: https://llvm.org/. Accessed on March 2020.

[23] Incisive Enterprise Simulator, 2020. [Online] Available at: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html. Accessed on March 2020.

[24] Mentor® ModelSim®, 2020. [Online] Available at: https://www.mentor.com/products/fv/modelsim/. Accessed on March 2020.

[25] Icarus Verilog, 2020. [Online] Available at: http://iverilog.icarus.com. Accessed on March 2020.

[26] Introduction to Verilator, 2020. [Online] Available at: https://www.veripool.org/wiki/verilator. Accessed on March 2020.

[27] Xilinx Vivado Design Suite - HLx Editions, 2020. [Online] Available at: https://www.xilinx.com/products/design-tools/vivado.html. Accessed on March 2020.

[28] Intel® Quartus® Prime Software Suite, 2020. [Online] Available at: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. Accessed on March 2020.

[29] Olof Kindgren and et. al. FuseSoC, 2020. [Online] Available at: https://github.com/olofk/fusesoc. Accessed on March 2020. GIT code repository.

[30] José T. de Sousa and et. al. IOb-SoC, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-soc/src/master. Accessed on March 2020. GIT code repository.

[31] Avnet, Inc. *Kintex UltraScale KU040 Development Board*, 2015. [Online] Available at: https://www.avnet.com/opasdata/d120001/medias/docus/13/aes-AES-KU040-DB-G-User-Guide.pdf. Accessed on March 2020.

[32] Spartan-6 FPGA SP605 Evaluation Kit, 2020. [Online] Available at: https://www.xilinx.com/products/boards-and-kits/ek-s6-sp605-g.html. Accessed on March 2020.

[33] Cyclone V GT FPGA Development Kit, 2020. [Online] Available at: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gt.html. Accessed on March 2020.

[34] José T. de Sousa and et. al. IOb-Cache, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-cache/src/master. Accessed on March 2020. GIT code repository.

[35] José T. de Sousa and et. al. IOb-RV32, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-rv32/src/master. Accessed on March 2020. GIT code repository.

[36] José T. de Sousa and et. al. IOb-UART, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-uart/src/master. Accessed on March 2020. GIT code repository.

[37] Xilinx. *UltraScale Architecture Clocking Resources*, October 2019. [Online] Available at: https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf. Accessed on March 2020.

[38] Micron Technology. *DDR4 SDRAM EDY4016A - 256Mb x 16 Datasheet*, July 2017. [Online] Available at: https://www.micron.com/products/dram/ddr4-sdram/part-catalog/edy4016aabg-dr-f. Accessed on March 2020.

[39] José T. de Sousa and et. al. Verilog AXI Components, March 2020. [Online] Available at: https://github.com/jjts/verilog-axi/tree/bce3d5e93398e3ee628f60a755f46fd6d92ad8db. Accessed on March 2020. GIT code repository.

[40] José T. de Sousa and et. al. IOb-Timer, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-timer/src/master. Accessed on March 2020. GIT code repository.

[41] Xilinx ISE Design Suite, 2020. [Online] Available at: https://www.xilinx.com/products/design-tools/ise-design-suite.html. Accessed on March 2020.

[42] José T. de Sousa and et. al. IOb-SoC-Dhrystone, March 2020. [Online] Available at: https://bitbucket.org/jjts/iob-soc-dhrystone/src/master/. Accessed on March 2020. GIT code repository.

[43] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[44] Roy Longbottom. Dhrystone Benchmark Results On PCs and Later Devices Roy Longbottom, 08 2017.