POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

MC2101: A RISC-V-based Microcontroller for Security Assessment and Training

Author: Luca Dalmasso

Advisor: Paolo Ernesto Prinetto
Co-Advisor: Gianluca Roascio

October, 2022

Summary

Today, as the Internet Of Things (IoT) world keeps growing, billions of microcontrollers are used as edge devices in a very wide range of applications, from industrial automation to health care dimension. Such massive diffusion of embedded devices in safety-critical and business-critical missions brings up important consequences in terms of security due to hardware and software vulnerabilities of such devices.

In literature, many security solutions for microcontrollers have been proposed, relying on both hardware and software techniques. While software-based solutions can be easily evaluated with a proper software toolchain suited for the target ISA, hardware solutions are more critical as they require an open architecture that can be easily customized starting from the core itself. The aforementioned requirements lead to choose an ISA that is well documented and allows to be extended and modified. For all these reasons, the most suitable architecture is the open-source RISC-V platform, widely used by the embedded systems research community. Other benefit of choosing RISC-V is to have a small and simple standard ISA that is adequate not only for testing new security solutions, but also for security training activities for students and professionals, e.g., through artificial insertion of vulnerabilities and exercises on exploits and remediations.

The aim of my thesis project is to design and implement MC2101, a soft microcontroller similar to PULPino, compatible with the RISC-V ISA and synthesizable on FPGA with the purpose of being a starting point to be extended, modified and used to perform security tests in a realistic environment. Starting from a custom 32-bit RISC-V processor named AFTAB, the goal was to design in VHDL a minimal set of peripherals that, together with the core and a proper bus architecture, are able to provide all the necessary I/O functionalities for running test software on a real board. In particular, the peripherals introduced are a 32-bit GPIO module that is used to handle both incoming and outgoing digital signals, and a UART peripheral containing a subset of the UART 16550 specifications selected for our needs. The UART peripheral plays a very important role, allowing a serial communication between MC2101 and a PC. This is a crucial feature to interact with the board through standard input and output interfaces.

To verify the correctness of the architecture, MC2101 has been synthesized and tested with custom Assembly and C applications with the aim of testing the correct behaviour of all interconnected peripherals, as well as the software libraries necessary for driving them. In conclusion, the synthesis results on FPGA have been compared with PULPino microcontroller to see what are the differences in terms of resources utilization.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded systems are the real driving force of today's society and almost every aspect of our life is becoming more and more dependant on them. They appear in every industrial sector as edge devices engineered to operate safety-critical and mission-critical environments with minimal human intervention. They play crucial role in functioning of cars, home appliances, medical devices and many other equipments that are the heart of today's digital, connected and automated world we live in.

The key benefits of such devices is their capability to perform in real time, with no or little delay, even sophisticated tasks and guaranteeing at same time low power consumption and low manufacturing cost. On one hand, having a low power device is crucial in order to keep the cost down, also because power consumption is critical when the device is battery-powered. On the other hand, this implies a couple of consequences in terms of security:

1. Usually, built-in device security is minimal. An IoT system must be lightweight, and it is very difficult to have a system that is cheap but at the same time secure, low-power, and real-time reliable;

2. Limited resources cannot support a fully featured operating system with all common robust security features. For instance, it is not unusual to see IoT devices running software in bare-metal with few KB of on-chip memory.

For these reasons, a technology with these limits is by nature weak, and exposed to possible malicious attacks. Furthermore, because of the surging usage of such devices in mission-critical and safety-critical applications, their security is a growing concern.

In literature, there exist a wide number of solutions aimed to secure embedded systems, many of which rely on software techniques, like binary instrumentation, aimed at protecting the system at application level. Other solutions, on the other hand, are based on direct interventions on the hardware aimed at introducing additional modules capable of protecting the system from attacks that exploit vulnerabilities of other components of the system itself. While software-based techniques can be easily tested outside the operating environment, with a proper toolchain, hardware-based techniques need to be evaluated in a realistic environment, meaning that once designed they must be synthesized and tested, e.g., on a FPGA. For the reasons mentioned above, in order to make research at hardware

level, it is crucial to have the availability of an open-source architecture, modular and customizable without incurring any fees.
Among all the known open-source designs, the most widely used is the RISC-V platform, not only for academic purposes but also in industry. Among the reasons that made RISC-V such a success in the embedded systems research community, the following for sure should be mentioned:

- It is provided under open source licenses that do not require fees to use: thanks to this, derivative designs are allowed to be published, reused and modified;
- Instruction set is now supported by commonly available language compilers, e.g, RISC-V GNU GCC;
- It features a small base instruction set architecture engineered for extensibility: RISC-V ISA can be addressed to many possible uses, from performances to low-power real-world applications;
- Unlike legacy ISAs that are decades old, RISC-V is a modern, clean state-of-the-art architecture designed to handle the latest computer load.

The work behind the present thesis consisted in the design of MC2101, which is a simple, modular, and synthesizable embedded system entirely described in VHDL language, meant to be used as a reliable platform on which is possible to run real applications, as well as integrate and evaluate security solutions for IoT in a realistic environment. Furthermore, the platform can be used for training activities in the cybersecurity domain, e.g., for modeling specific hardware security issues in Capture-the-Flag exercises, where vulnerabilities are intentionally inserted with the aim of being exploited and/or mitigated.

MC2101 microcontroller integrates a RISC-V core with a proper set of peripherals necessary to provide all basic I/O functionalities for running software. In particular, the peripherals selected are a GPIO module for handling input and output digital signals and a UART module used to allow serial communication between a computer and the microcontroller itself. The relative simplicity and modularity of the system makes it suitable, in the future, to be also used as a platform for teaching microcontrollers architecture at master students involved in embedded systems curriculum.

The development of the microcontroller included also a software design part. In particular, all system libraries used for driving and configuring the peripherals were written, and also interrupt service routines have been included in the processor's bootloaders. The pre-existing software toolchain for automatic compilation and RTL simulation, based on CMake and ModelSim commands and derived from the PULPino project [14], has been extended with new test programs aimed at verify MC2101 activity on a board and at RTL level. Synthesis automation features have been included with the purpose of running Quartus Prime commands in a proper shell environment, in order to allow automatic synthesis and memory update for a fast deployment on FPGA.

To conclude the work, MC2101 synthesis results have been compared with the PULPino microcontroller, used as reference in many projects for both RISC-V hardware and software design, to understand what are the differences in terms of complexity and resource usage in a FPGA.

The remainder of the document is organized as follows. Chapter 2 contains a brief description of the RISC-V ISA, with particular attention to the subset of instructions executable by our core. It is underlined also the importance of the RISC-V ISA today. Also, are analyzed some of the most relevant scientific works in SoCs design present in literature, and all the motivations that led us to choose a new design from scratch. Some details on our RISC-V core are also provided, focusing on the most important features implemented. Chapter 3 presents a high-level description of the MC2101 microcontroller. In particular, the architecture features, hardware/software co-design choices and testing framework are presented. In Chapter 4, synthesis results are evaluated and compared with our reference architecture PULPino. Finally, Chapter 5 concludes the thesis, providing ideas for future improvements of the system.

Chapter 2

Background

In this Chapter, all technical elements needed for the understanding of the covered topics are presented. In particular, since MC2101 is a RISC-V-based microcontroller, the RISC-V ISA is briefly described, with focus on the details of the implemented features. It is also underlined the importance of having an open-source ISA, like RISC-V, which allows the embedded system community to grow continuously, and consequently, to bring technological innovation to the IoT world without being limited by closed-source proprietary solutions. Some relevant scientific works are also presented, that are the today's state-of-the-art for system on a chip design, and all the motivations that led us to choose a new design from scratch. Regarding the RISC-V core present in the architecture, AFTAB [5], its design is not described in details, and only some technical characteristics will be discussed, with the purpose of being useful just for the comprehension of some architectural choices made for the microcontroller design.

2.1 The RISC-V Architecture

RISC-V is the fifth generation of the open RISC ISA design from Berkeley UC. It was born originally to support academic research in computer architecture and today it has become the most widely used open-source instruction set in the embedded systems community. Starting from its introduction in 2010, the project has been growing continuously. As a consequence, the number of extensions and features is now quite significant. Since it would be too much verbose explaining all details documented in the official specifications, this Section only presents the ones of interest for our system.

Any further details on RISC-V ISA can be found in the official documentation, which is comprehensively explained into two volumes that can be found in the public Github repository: `https://github.com/riscv/riscv-isa-manual`.

2.1.1 Software Execution Environment

Every RISC-V platform is first described by its Software Execution Environment Interface (EEI). The EEI defines: the initial state of a program running on a RISC-V core, privilege modes supported, memory and I/O accessibility and attributes, the ISA itself, the handling mode for any interrupt and exception including environment calls. An example of EEI implementation can be a RISC-V operating system that manages multiple user-level applications controlling their accesses to physical memory through a virtual memory interface. Regarding our RISC-V machine, since the core is not sufficiently equipped to run an operating system yet, it can be considered a bare-metal platform where all programs have full access to the physical address space.

2.1.2 Instruction Set Architecture

RISC-V has a modular design: the ISA consists of a base integer set of instructions, that must be included in every machine, plus a wide number of optional standard extensions. The base integer ISA implements a minimal set of instructions that are enough for building up a simplified computer with full software support by themselves, including general-purpose compilers, linkers, and assemblers. In particular, there are four groups of base ISA available:

- **RV32I** & **RV64I**: two primary integer variants, with the only difference residing in their data and address parallelism (32-bit and 64-bit, respectively)
- **RV128I**: future variant of integer set supporting 128-bit parallelism;
- **RV32E**: a subset variant of the RV32I, implemented to support small 32-bit microcontrollers with just 16 registers.

Regarding the standard extensions, they are group of instructions developed as a result of a collective effort between industry, research community, and educational institutions to allow general-purpose software development in a wide range of applications. Extensions are a crucial key for the RISC-V flexibility, as they can be included in a specialized design, without conflict, in such a way that only the exact set of ISA features required by the application are implemented, avoiding over-architecting a particular microarchitecture style. The number of standard extensions is quite large, and here are reported just some very commonly used ones:

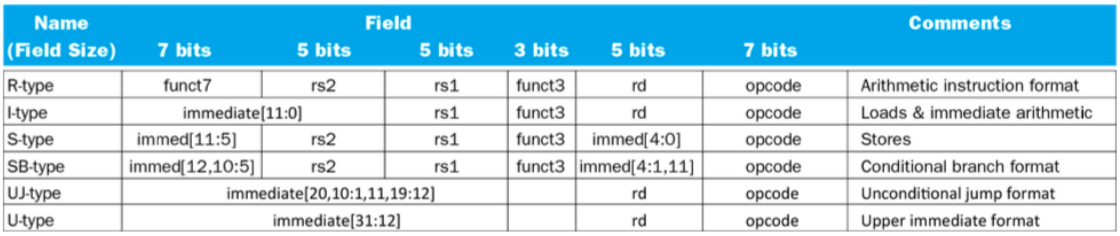
- **"M" extension • Integer Multiplication and Division**
  Adds instructions to multiply and divide values held in integer registers.
- **"A" extension • Atomic Instructions**
  Adds instructions that atomically read, modify and write memory locations for interprocessor synchronization.
- **"F" extension • Single-Precision Floating-Point**
  Adds floating-point registers, single precision computational instructions as well as load and stores instructions.
- **"D" extension • Double-Precision Floating-Point**
  Variant of the "F" extension for double precision arithmetic.
- **"C" extension • Compressed Instructions**
  Provides 16-bit forms of common instructions.

The combination of all the previous standard extensions plus a base integer is commonly known as the "G" extension, and together with the supervisor instructions implemented in the "S" extension defines all instructions needed to fully support a general-purpose operating system.

2.1.3 Implemented Extensions

Here is presented the set of instructions currently implemented in MC2101. The ISA currently includes the RV32I Base Integer Set, the "M" Integer Multiplication and Division extension, and finally the "Zicsr" Control and Status Registers (CSRs) extension.

The base integer ISA **RV32I** is divided into four core instruction *formats* (R, I, S, and U Type), plus two further variants (B, J) for handling immediate values. From Figure 2.1, it is possible to see that the opcode, the source operands (`rs1` and `rs2`) and destination operand (`rd`) are maintained in the same bit positioning in order to reduce the decoding phase complexity. The integer register file, used by all integer instructions, is composed of 32 general-purpose 32-bit registers. Table 2.1 shows the registers names together with the ABI conventions. Notice that the program counter is not part of the register file itself, as the software cannot directly address it.



Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Figure 2.1. RISC-V ISA Encoding Formats [11].

Part of the implemented ISA is also the Integer Multiplication and Division extension **RV32M**, which contains instructions that multiply or divide values held in two integer registers (see Table 2.1). The reason that made designers to separate multiply and divide out from the base integer is because usually multiplications and divisions operations are either infrequent or better handled by specialized accelerators. Thanks to this extension, the ISA can support multiplications, divisions, and reminder operations (No operation allows computing both Quotient and Reminder of a division).

The last extension implemented is the Control and Status Registers extension **Zicsr**, containing instructions needed to operate on CSRs registers. CSRs are a particular class of registers that can only be addressed by the Zicsr instructions and they are primarily used by privileged architecture to perform very specific tasks, e.g., handling exceptions, interrupts and traps. RISC-V ISA counts a total amount of 4096 CSRs but allow to implement

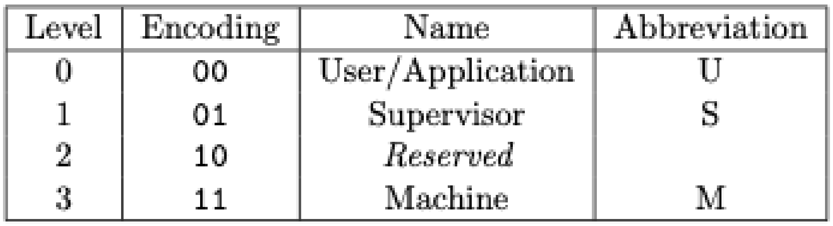
Register	**ABI Name**	**Description**	**ABI Saver**
`x0`	`zero`	Hard-wired zero	/
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	/
`x4`	`tp`	Thread pointer	/
`x5`	`t0`	Temporary/alternate link register	Caller
`x6-7`	`t1-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller

Table 2.1. RISC-V Integer Register File summary.

just a subset of them. Our platform, for example, includes only those required for handling exceptions, interrupts, traps and for supporting User and Machine privilege levels.

2.1.4 Privilege Levels

Privilege levels are used to provide a protection mechanism, embedded in the hardware itself, between the different components of software stack, in such a way that any attempt to perform operations not permitted by the current privilege mode will rise an exception. Figure 2.2 shows all privilege modes supported by a RISC-V machine. These modes are



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

Figure 2.2. RISC-V Supported Privileges [2].

listed in order of increasing privilege, where *Machine* mode is the most privileged and *User* mode being the least. All RISC-V systems must implement the *M* mode, while the other are optional extensions.
Thus, any RISC-V platform must fall in one of the following configurations, depending on the requirements:

- **M**
  Simple embedded system.
- **M, U**
  Secure embedded system.
- **M, S, U**
  Systems running Unix-like operating systems.

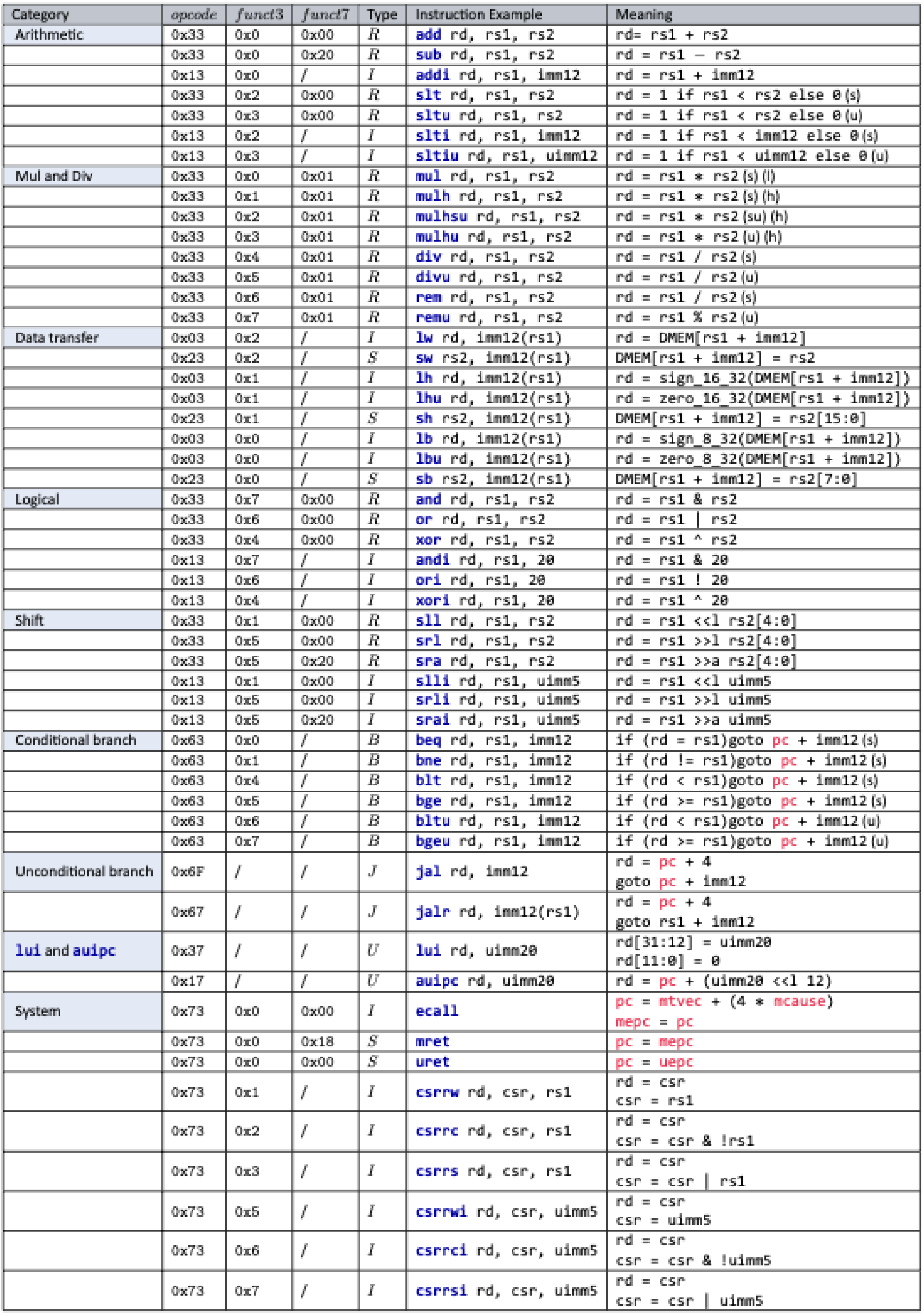
When designing a RISC-V platform, the proper set of privileges must be carefully chosen based on the complexity of the software it will run. For instance, a small device that runs a single, *thrusted* application may choose to only support *Machine* mode: in this case, the application must be secure because it has full low-level access to the hardware. The modality *Machine - User* is suitable for secure embedded systems. It provides a level of isolation between the application and a direct access to the hardware, by using `ecall` instruction to perform machine-level operations, that can fully and directly access the hardware. The full software stack separation is reached when the machine implements the *Machine - Supervisor - User* mode. In this case the ISA is able to support a more robust system where there is an operating system working as intermediate between user and hardware.

Since our microcontroller is intended to be used as a security platform, it implements both *M* and *U* privileges required for a secure embedded system.

2.2 RISC-V Importance in IoT era

Provided the main features of the RISC-V ISA, it should be clear the advantage that a modular architecture of this type can bring in terms of design freedom and use cases. In fact, the various extensions allow to design any kind of chip, from very small 16-bit low-power and cost-constrained microcontrollers, to extremely complex 64-bit full-featured high-performance SoCs. But why is there any need for a new instruction set architecture when there are several popular commercial ISAs available, that could be reused avoiding the significant effort and cost of porting software to a newer one?

The first reason behind the adoption of the RISC-V ISA as the standard architecture in academia and research is that all popular commercial ISAs, like x86 or ARM, are *proprietary.* This means that vendors have a lucrative business in selling implementations, both in form of IP cores and silicon. Because of this, it is economically not sustainable for many companies, universities, or individuals, to develop custom designs. Another reason is that all popular commercial ISAs are massively *complex.* In particular, the increasing complexity of legacy microprocessors has become profound, requiring hundreds of engineers,



Category	*opcode*	*funct3*	*funct7*	Type	Instruction Example	Meaning
Arithmetic	0x33	0x0	0x00	*R*	add rd, rs1, rs2	rd= rs1 + rs2
	0x33	0x0	0x20	*R*	sub rd, rs1, rs2	rd = rs1 – rs2
	0x13	0x0	/	*I*	addi rd, rs1, imm12	rd = rs1 + imm12
	0x33	0x2	0x00	*R*	slt rd, rs1, rs2	rd = 1 if rs1 < rs2 else 0 (s)
	0x33	0x3	0x00	*R*	sltu rd, rs1, rs2	rd = 1 if rs1 < rs2 else 0 (u)
	0x13	0x2	/	*I*	slti rd, rs1, imm12	rd = 1 if rs1 < imm12 else 0 (s)
	0x13	0x3	/	*I*	sltiu rd, rs1, uimm12	rd = 1 if rs1 < uimm12 else 0 (u)
Mul and Div	0x33	0x0	0x01	*R*	mul rd, rs1, rs2	rd = rs1 * rs2 (s) (l)
	0x33	0x1	0x01	*R*	mulh rd, rs1, rs2	rd = rs1 * rs2 (s) (h)
	0x33	0x2	0x01	*R*	mulhsu rd, rs1, rs2	rd = rs1 * rs2 (su) (h)
	0x33	0x3	0x01	*R*	mulhu rd, rs1, rs2	rd = rs1 * rs2 (u) (h)
	0x33	0x4	0x01	*R*	div rd, rs1, rs2	rd = rs1 / rs2 (s)
	0x33	0x5	0x01	*R*	divu rd, rs1, rs2	rd = rs1 / rs2 (u)
	0x33	0x6	0x01	*R*	rem rd, rs1, rs2	rd = rs1 / rs2 (s)
	0x33	0x7	0x01	*R*	remu rd, rs1, rs2	rd = rs1 % rs2 (u)
Data transfer	0x03	0x2	/	*I*	lw rd, imm12(rs1)	rd = DMEM[rs1 + imm12]
	0x23	0x2	/	*S*	sw rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2
	0x03	0x1	/	*I*	lh rd, imm12(rs1)	rd = sign_16_32(DMEM[rs1 + imm12])
	0x03	0x1	/	*I*	lhu rd, imm12(rs1)	rd = zero_16_32(DMEM[rs1 + imm12])
	0x23	0x1	/	*S*	sh rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2[15:0]
	0x03	0x0	/	*I*	lb rd, imm12(rs1)	rd = sign_8_32(DMEM[rs1 + imm12])
	0x03	0x0	/	*I*	lbu rd, imm12(rs1)	rd = zero_8_32(DMEM[rs1 + imm12])
	0x23	0x0	/	*S*	sb rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2[7:0]
Logical	0x33	0x7	0x00	*R*	and rd, rs1, rs2	rd = rs1 & rs2
	0x33	0x6	0x00	*R*	or rd, rs1, rs2	rd = rs1 | rs2
	0x33	0x4	0x00	*R*	xor rd, rs1, rs2	rd = rs1 ^ rs2
	0x13	0x7	/	*I*	andi rd, rs1, 20	rd = rs1 & 20
	0x13	0x6	/	*I*	ori rd, rs1, 20	rd = rs1 ! 20
	0x13	0x4	/	*I*	xori rd, rs1, 20	rd = rs1 ^ 20
Shift	0x33	0x1	0x00	*R*	sll rd, rs1, rs2	rd = rs1 <<l rs2[4:0]
	0x33	0x5	0x00	*R*	srl rd, rs1, rs2	rd = rs1 >>l rs2[4:0]
	0x33	0x5	0x20	*R*	sra rd, rs1, rs2	rd = rs1 >>a rs2[4:0]
	0x13	0x1	0x00	*I*	slli rd, rs1, uimm5	rd = rs1 <<l uimm5
	0x13	0x5	0x00	*I*	srli rd, rs1, uimm5	rd = rs1 >>l uimm5
	0x13	0x5	0x20	*I*	srai rd, rs1, uimm5	rd = rs1 >>a uimm5
Conditional branch	0x63	0x0	/	*B*	beq rd, rs1, imm12	if (rd = rs1)goto pc + imm12 (s)
	0x63	0x1	/	*B*	bne rd, rs1, imm12	if (rd != rs1)goto pc + imm12 (s)
	0x63	0x4	/	*B*	blt rd, rs1, imm12	if (rd < rs1)goto pc + imm12 (s)
	0x63	0x5	/	*B*	bge rd, rs1, imm12	if (rd >= rs1)goto pc + imm12 (s)
	0x63	0x6	/	*B*	bltu rd, rs1, imm12	if (rd < rs1)goto pc + imm12 (u)
	0x63	0x7	/	*B*	bgeu rd, rs1, imm12	if (rd >= rs1)goto pc + imm12 (u)
Unconditional branch	0x6F	/	/	*J*	jal rd, imm12	rd = pc + 4 goto pc + imm12
	0x67	/	/	*J*	jalr rd, imm12(rs1)	rd = pc + 4 goto rs1 + imm12
lui and auipc	0x37	/	/	*U*	lui rd, uimm20	rd[31:12] = uimm20 rd[11:0] = 0
	0x17	/	/	*U*	auipc rd, uimm20	rd = pc + (uimm20 <<l 12)
System	0x73	0x0	0x00	*I*	ecall	pc = mtvec + (4 * mcause) mepc = pc
	0x73	0x0	0x18	*S*	mret	pc = mepc
	0x73	0x0	0x00	*S*	uret	pc = uepc
	0x73	0x1	/	*I*	csrrw rd, csr, rs1	rd = csr csr = rs1
	0x73	0x2	/	*I*	csrrc rd, csr, rs1	rd = csr csr = csr & !rs1
	0x73	0x3	/	*I*	csrrs rd, csr, rs1	rd = csr csr = csr | rs1
	0x73	0x5	/	*I*	csrrwi rd, csr, uimm5	rd = csr csr = uimm5
	0x73	0x6	/	*I*	csrrci rd, csr, uimm5	rd = csr csr = csr & !uimm5
	0x73	0x7	/	*I*	csrrsi rd, csr, uimm5	rd = csr csr = csr | uimm5

Figure 2.3. Implemented Instruction Set [6].

millions in investment, years in development, and all this complexity lead to a very inefficient and power-hungry design, for sure not suitable for IoT applications. It is therefore very difficult to fully implement one of those legacy architectures. Also, there is a little incentive to create subset ISAs, for more specialized design, because the software cannot run without being modified as well. For the aforementioned reasons, having an open source architecture like RISC-V, with tons of materials available for free, as well as a wide variety of projects going on, is a fundamental architecture that can help to overcome the barriers imposed by proprietary solutions. Among all the available open-source RISC-based ISA, RISC-V was designed to face these problems, through its modular architecture that allows to be used in any kind of applications, suitable also for educational purposes thanks to the simple base integer subset.

Barriers	**Legacy ISA**	**RISC-V ISA**
Complexity	1500+ instructions, incremental ISA	47 base instructions, modular ISA
Design freedom	$$$ - Limited	Free - Unlimited
License and Royalty fees	$$$	Open Source
Design ecosystem	Moderate	Growing rapidly. Numerous extensions, numerous open cores.
Software ecosystem	Extensive	Growing rapidly

Table 2.2. Summary of RISC-V advantages with respect to Legacy ISAs [7].

The Table 2.2 is useful to summarize the advantages that the RISC-V ISA can offer with respect to popular commercial ISAs. These are the main reasons that allowed RISC-V to grow and become the today standard ISA used for embedded systems design in academia and research world.

It is important to underline that RISC-V is not only a revolution for hardware design in academia and research: it is also cracking a new commercial market wide open. As the IoT era keeps growing day by day, more and more companies are starting to develop their own chip, and companies that long prized for their proprietary architectures, like ARM, Intel and others, suddenly have to contend and compete with this new way of thinking and building. Therefore, it is essential that universities train engineers capable of designing efficient and reliable microarchitectures and also that the research can continue to produce innovative ideas that allow IoT technology to be widespread.

2.3 State of the Art in Embedded System Design

In this Chapter, some relevant state-of-the-art works are presented, featuring relevant scientific works related to RISC-V. As the purpose of my thesis is the design of a microcontroller, this Chapter discusses some of the most popular open-source RISC-V based SoCs, designed for energy-efficient computing. To stress again on the wide possibility that the RISC-V ISA can offer, three different types of architecture are here presented.

The first one is the BOOM [13] SoC, part of the Berkeley Hardware CPUs, a very complex platform with an infrastructure usable for personal, supercomputer and warehouse-scale computers. The second example is about PicoRV32 [10] by Claire Wolf, a 32-bit microcontroller class RV32IMC implemented in Verilog HDL. The third and last example focuses on PULPino [14] from ETH Zürich and University of Bologna, which is a simple microcontroller that can be extended for being used as custom embedded DSP. The architecture of this last microcontroller has been particularly important for my thesis work, because it has been used as a reference.

2.3.1 Berkeley Out of Order Machine

The BOOM machine is a synthesizable and parameterizable open-source RV64GC RISC-V core, written in Chisel [1]. As its name suggest, BOOM is an out-of-order processor, inspired by the MIPS R10000 [12] and the Alpha 21264 [8] processors . Created at the University of Berkeley (California) in the Berkeley Architecture Research group, its focus is to provide a high performance, synthesizable and parameterizable SoC for architecture research. This is a perfect example to underline how much degree of complexity can be reached by using RISC-V extensions.

From Figure 2.4, it is possible to distinguish the classical pipeline stages of out-of-order architectures. Conceptually, the processor implements 10 stages: `Fetch`, `Decode`, `Register Rename`, `Dispatch`, `Issue`, `Register Read`, `Execute`, `Memory`, `Writeback` and `Commit`. The architecture is then able to execute instructions in out-of-order fashion, pushing even more the throughput and performances of a standard pipeline. The fact that it also implements a 64-bit version of the standard `G` indicates this core is designed exclusively for high performance systems.

Further details can be found in the official Github repository[1], where it is also possible to access at the RTL description.

2.3.2 Picorv32 microcontroller

The PicoRV32 is a microcontroller that implements the RV32IMC instruction set written in Verilog. This microcontroller can be configured in many different ways, starting from the core itself that can be configured to support the following extensions: `RV32I`, `RV32IC`, `RV32IM` and the `RV32IMC`. The architecture of the microcontroller is also available in the following configurations:

- `picorv32`: simplest version that implements a simple native memory interface, usable in simple environments;
- `picorv32_axi`: provides an AXI-4 Lite Master interface that can be easily integrated within other systems based on AXI standard bus infrastructure;
- `picorv32_wb`: provides a Wishbone bus master interface;

[1] https://github.com/riscv-boom/riscv-boom

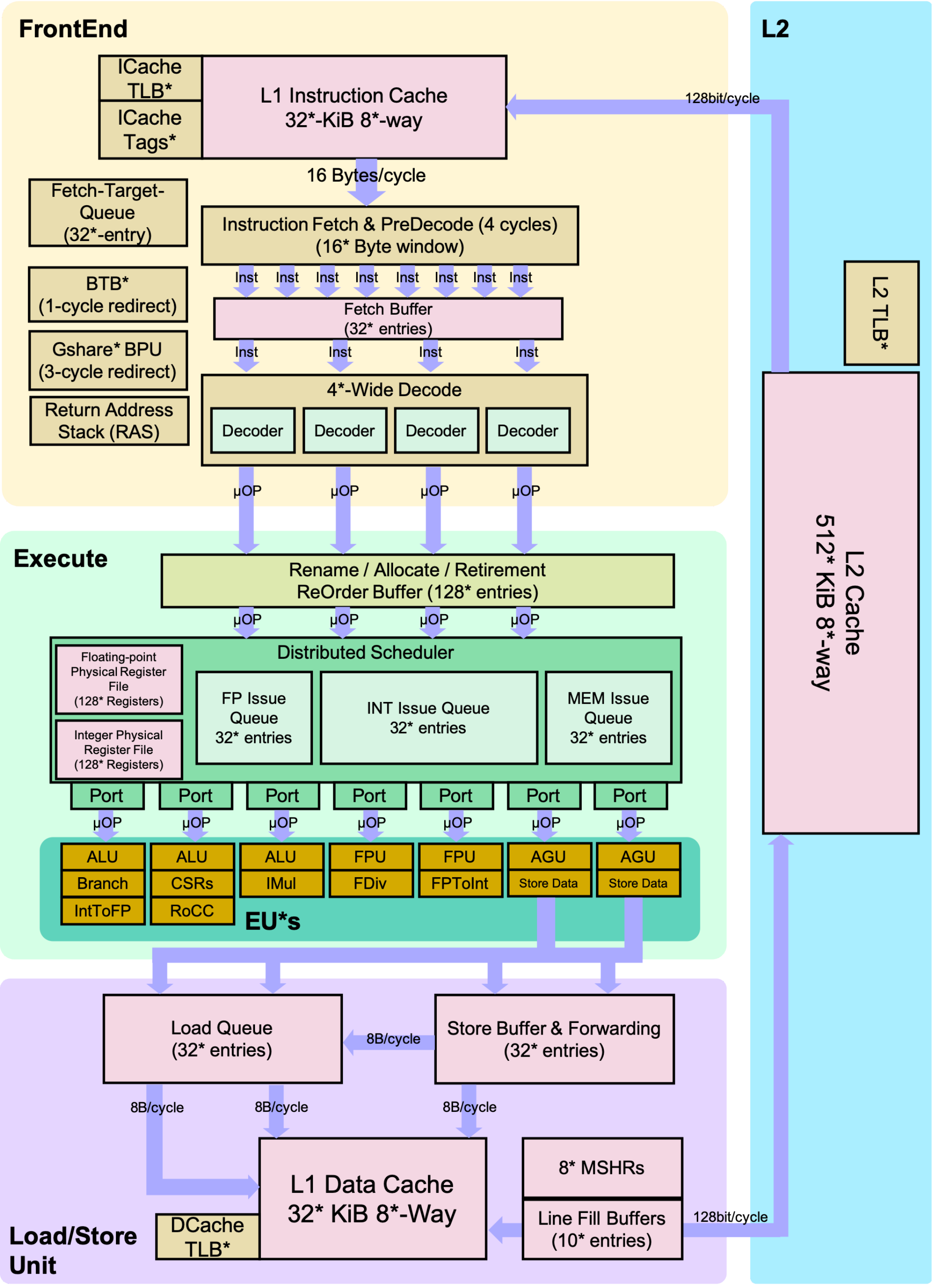


Figure 2.4. BOOM Architecture Overview [13].

- `picorv32_axi_adapter`: This core provides a bridge between the native memory interface and an AXI4 infrastructure. This implementation can be used to create custom cores that include one or more `picorv32` cores together within a compact microcontroller that internally can communicate using a native custom lightweight interface, and externally can be attached through AXI4.

It is possible to understand that this embedded system is meant to be used as auxiliary processing device in FPGA designs and ASICs. The architecture includes also a UART peripheral and a SPI Flash Controller. Any additional information can be found in the official Github repository[2].

2.3.3 PULPino

Developed by ETH Zurich and the University of Bologna, PULPino is a RISC-V-based single-core SoC written in SystemVerilog, which represents a small part (one core) of the Parallel Ultra-Low Power (PULP) platform designed for energy-efficient IoT parallel computing.

In Particular, the PULP SoC is a cluster that embeds a configurable number of RISC-V based cores with a design focused for being extremely low power consuming. Perfectly suited for IoT devices that requires high computational capabilities. PULPino represents a first step towards the release of the full PULP platform. In fact, PULPino inherits from its bigger brother some of the IPs and the core, focusing on ease of use and simplicity.

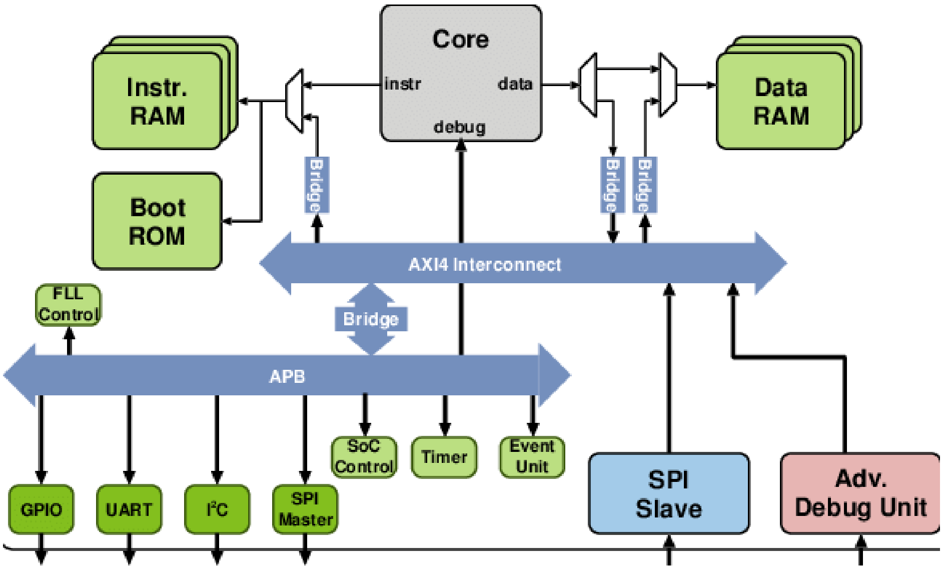


Figure 2.5. PULPino Microcontroller [14].

[2] https://github.com/YosysHQ/picorv32

Figure 2.5 shows the design overview of the SoC. From an architectural point of view, it is a simple single-core AMBA-based embedded system. It offers a modular design that does not include complex features like caching mechanisms, memory hierarchy or DMA. The core is directly connected to the IRAM and DRAM in a single access, no waiting manner. There is a central AXI interconnection, that interconnects the processor with the memories, allowing pipelined high-bandwidth operations. Regarding the core, the microcontroller is based on a 32-bit RISC-V architectures and can be configured to use either the `RISCY` or the `zero-riscy` core. Both developed at ETH Zürich.

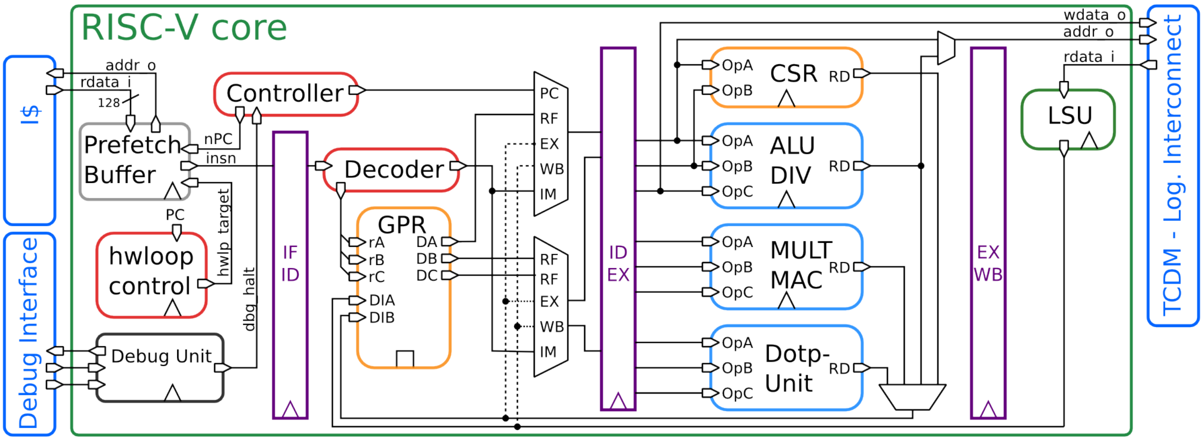


Figure 2.6. RISCY core.

The `RISCY` core, shown in Figure 2.6, is an in-order, single-issue core with 4 pipeline stages able to guarantee an IPC close to 1. The ISA supports the base integer instruction set (RV32I), compressed instructions (RV32C), multiplication and division instruction extension (RV32M). In can also be configured to implement the RV32F extension for single-precision floating point operations. It also implements several custom ISA extensions such as hardware loops, post-incrementing load and store instructions, bit-manipulation instructions, MAC operations, fixed-point operations, packed-SIMD instructions and the dot product. All these additional features have been implemented for allowing ultra-low-power signal processing applications. Also, a subset of the 1.9 privileged instructions are supported.

The `Zero-riscy` core, depicted in Figure 2.7, is instead a much more simpler processor. Suitable for ultra-low-power ultra-low area constrained embedded systems. It is an in-order, single issue core with 2 pipeline stages supporting the RV32I, RV32C and a subset of 1.9 privileged ISA. Can also be configured to implement the RV32M and the reduced number of register extension RV32E. The architecture offers a secondary bus level that interconnects a broad set of peripherals for the communication with the outside world. Part of the implementations are also the SPI Slave and the Debug Unit, which are used to pre-load the RAMs with executable code from SPI flash and to allow access to the whole memory map via JTAG for debugging purposes.

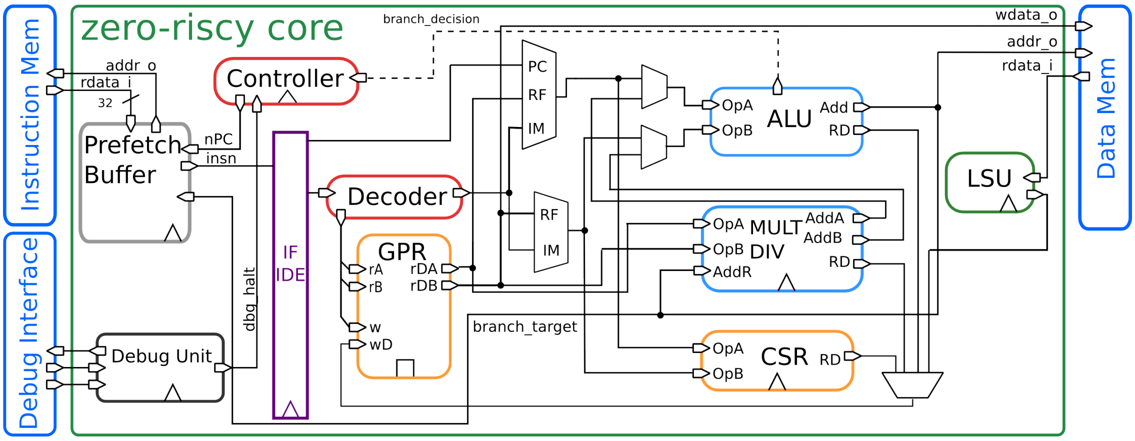


Figure 2.7. Zero-riscy core.

Among all the available microarchitectures, PULPino is the one chosen to be our reference architecture. This is because it implements in a very simple and compact design all the features necessary to have a complete usable microcontroller-style platform, closer to our requirements. Also, what makes PULPino perfect for being used as a reference soft microcontroller is for sure its modular and clean design, the extensible software toolchain built around it for cross-compilation, RTL simulation and synthesis, together with a great documentation to support usage and learning.

The RTL description, as well as the entire toolchain for simulations, compilation and FPGA synthesis can be found in the official Github repository[3].

2.4 Our contribution to the RISC-V community

Whenever an team of engineers decide to start designing a new RISC-V platform, the first possible way to act is to start from an already implemented open-source platform, like one of those seen before, where it is possible to include custom integrations and modify the existing design to fulfil the desired behaviour. The second choice would be to start a new design from scratch, by using some well known architecture as a reference. There are two factors on which the choice of the path to take is based: the first one regards the specifications and use cases of the embedded device. For instance, if the target is to design a low-power IoT device, if one wants to start from something already developed, he or she must have a soft lightweight microcontroller to customise, otherwise there is the need to start a new design from scratch. The second factor is related to the technical background of the team on HDL languages, which require a certain degree on know-how in order to develop clean and synthesizable designs. Embedded Systems engineers in Politecnico di Torino, for example, have strong expertise in VHDL, but not in SystemVerilog or Chisel

[3] https://github.com/pulp-platform/pulpino

HDLs, and this reflects to the students.

Regarding our RISC-V based platform MC2101, the research team that hosted me needed to have a RISC-V based architecture entirely customizable and synthesizable. The particular aim of our platform is to offer the possibility to integrate hardware security solutions and also to assess and evaluate them through software testing. Those kind of solutions first require a simple architecture that can be easily customised starting from the core itself, and then a proper toolchain for automating all the processes of RTL simulation, synthesis and compilation.

Among all the most popular solutions available in literature, the PULPino architecture mentioned before is for sure the one that comes closest to our needs. Referring again to the Figure 2.5, it is a modular design, easy to use, that includes all the set of peripherals necessary to interact with the microcontroller once synthesized on FPGA, as well as the necessary interfaces used to debug and flash the firmware on it. All these features are what is needed to test the architecture and any custom integration. Of great importance is also the *software toolchain*, created with the aim of automating all the processes of synthesis, RTL simulations, and software compilation. The big problems with PULPino, common to many other RISC-V most popular open-hardwares, is that it is fully described using SystemVerilog hardware description language, which is not part of our background as computer engineers.

It was therefore mandatory to opt for our own design, developed from scratch, fully implemented in VHDL. The AFTAB processor, briefly described in the next Section, was the first step of the development.

2.5 The AFTAB Processor

AFTAB, acronym for "A Fine Turin/Tehran Architectural Being", is an in-order fully sequential (not pipelined) 32-bit RISC-V core, developed by the CINI Cybersecurity National Laboratory in cooperation with University of Tehran. The processor works according to the Von Neumann architecture, so there is only a single RAM containing both data and instructions.

As for the implementation details, its enough to give a high-level description of the processor, starting from its interface and going directly to the programmer's model.

2.5.1 AFTAB Interfacing Ports

The core can be described as a blackbox component (Figure 2.8), whose pins can be grouped into the Memory Interface set and the Interrupt Interface set. The Memory Interface includes these signals:

- `memRead` & `memWrite`: read and write signals for the memory. Forwarded in the MC2101 bus in order to start a transaction that can target the main memory or any peripheral
- `memReady`: fundamental signal used to stall the processor in the fetch stage. In this way, by controlling this signal, the bus infrastructure can take the necessary time to complete read and write operations;

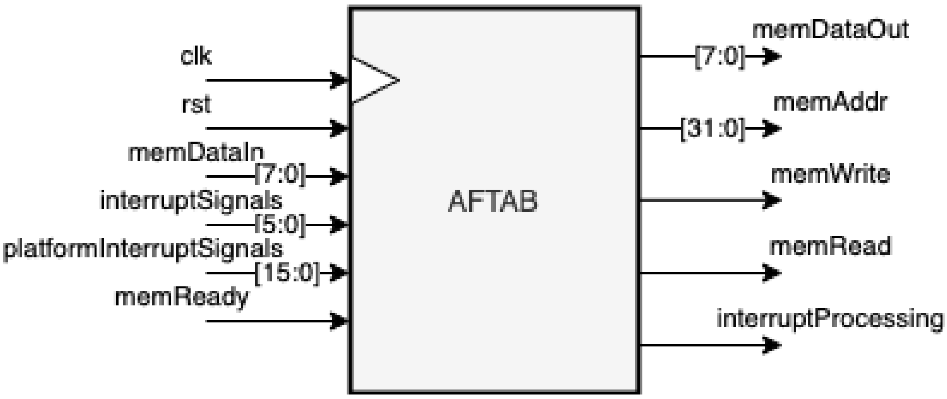


Figure 2.8. AFTAB microprocessor interface.

- `memAddr`: AFTAB address bus is 32-bit wide;
- `memDataIn` & `memDataOut`: two 8-bit ports that together build the data bus of the microcontroller. During load and store operations, they carry variable number of bytes per transaction (1, 2, or 4) in a sequential way, so that only one byte per clock cycle can be read or written.

As for the Interrupt Interface set, there are a total of 22 independent interrupt lines, where 16 of them are for platform use, plus an additional signal (`interruptProcessing`) indicating when the processor is in the interrupt processing states. The core is also able to handle the following hardware-level exceptions:

- **Illegal Instruction**: the decode phase does not recognise the opcode as a valid one;
- **Illegal CSR instruction**: raised when there is an attempt to read or write a CSR register with an inappropriate privilege level, or when trying to access a non-existing CSR register;
- **Instruction address misaligned**: raised after an attempt to access the instruction memory without the proper 4-bytes alignment.

For the programmer's point of view, as previously anticipated, AFTAB implements the RV32I, RV32M, and Zicsr extensions. Therefore, the instruction set supported is the one depicted in Figure 2.3, and the registers available are the integer ones, listed in Figure 2.1.

The RTL description, as well as the entire toolchain for simulations, compilation and the manuals can be found in the official Github repository[4].

[4] `https://github.com/RHESGroup/aftab`

Chapter 3

Design and Development of MC2101

The present Chapter is entirely dedicated to MC2101 microcontroller. The first Section explains the hardware-level architecture of MC2101, from the bus infrastructure to the various peripherals, and all the related design and development choices. The software part is described in the second Section, dedicated precisely to describe all the implemented system's libraries provided to support the programmer job with a proper set of low-level C functions, useful to program all peripherals without the need of accessing their registers "by hand". A third Section is dedicated to the testing part, in particular to the testing methodology followed to assess the functional correctness of the various designed hardware components, and their interconnection. The same Section is also dedicated to describe how does the serial communication between FPGA and a PC have been implemented, with the purpose of providing a way to allow the interaction between a personal computer and the synthesized MC2101, through a command-line terminal.

3.1 MC2101 architecture

MC2101 microcontroller has been designed with the purpose of being used as a synthesizable and extensible platform for integration and assessment of security solution for IoT implemented inside the AFTAB processor. At the current state, the microcontroller is composed of a minimal set of peripherals properly selected for providing all necessary input/output functionalities useful for interacting with the external world, once synthesized on a FPGA.

The Figure 3.1 shows the design overview of the microcontroller. The architecture includes a single bus, which provides the system with the proper hardware infrastructure necessary to interconnect the AFTAB processor with all peripherals and the main memory. The processor plays the role of the master: it is the only component able to initiate transactions on the bus, which can be composed of single or multi-cycles read/write operations. Peripherals and main memory are accessed by AFTAB using the memory-mapped mode: this means that the entire address space includes both peripherals and main memory. Therefore, the processor uses the same load/store instruction for accessing all components

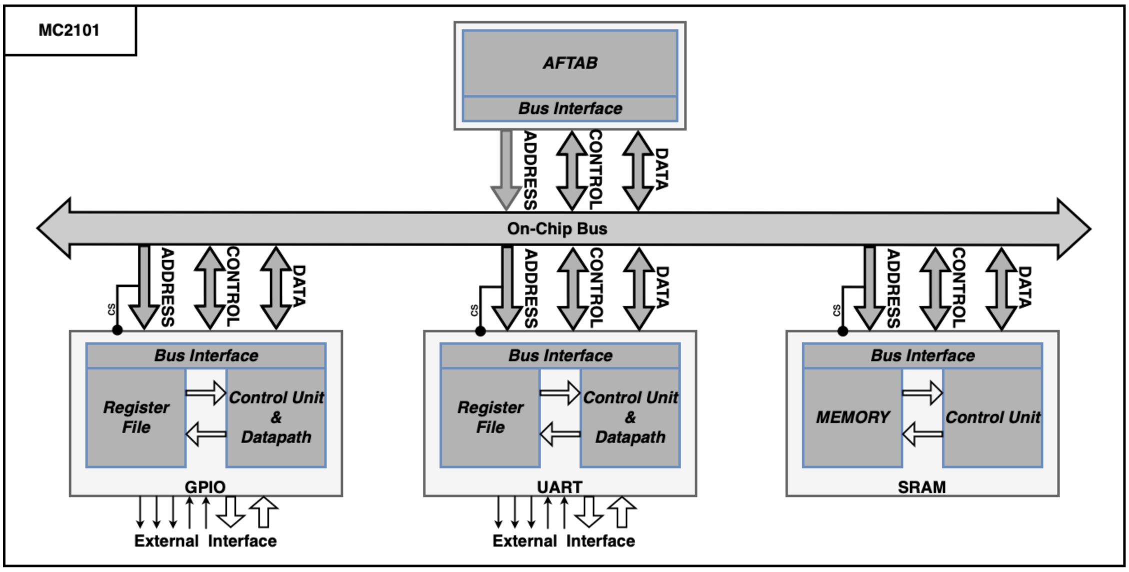


Figure 3.1. MC2101 microcontroller.

attached to the bus.

The communication between the processor and any peripheral happen only through the peripheral's register file, which can contain three different type of registers, with different read/write policy:

- **Control Registers**: used to configure the peripheral functionalities. They are written by the processor and read by the peripheral;
- **Status Registers**: used to report the current state of the peripheral. Written by the peripheral and read by the processor;
- **Data Registers**: used to exchange data. Both processor and peripheral can read and write on them.

The size and the number of registers depends on the functionalities implemented by the peripheral, which can be for instance a 32-bit peripheral, with 32-bit register file or a 8-bit peripheral with 8-bit registers.

The bus implements three different types of interconnections:

- **Data Lines**: set of independent read and write data signals used to read and write data on peripherals registers and on the main memory;
- **Control Lines**: used to provide signals for controlling read or write operations and for allowing peripherals and main memory to feedback the processor about the current state of the transaction;

- **Address Lines & Chip Select**: the most significant bits of the address lines pass through a decoder that drives all chip select signals: in this way, the peripheral is activated only when necessary.

Each component attached to the bus must include the Bus Interface, which is a sort of wrapper to be placed around each component in such a way to bridge signals coming from the bus to the internal hardware (FSM + Datapath) of the peripheral and vice versa. Peripherals can implement an additional External Interface, which is used to handle the communication to the external world. In particular, the conversion of incoming external asynchronous digital signals into the synchronous domain of the microcontroller.

In the following Subsection, some more details of the microcontroller design are reported, in order to present what are the functionalities supported by the system and the relative design choices made.

3.1.1 Bus Infrastructure

In the microcontroller development, the bus infrastructure was the first item to be addressed because of its vital role in interconnecting the whole system. In literature, there exist a lot of bus architectures which, over the years, have evolved to become an open-standard in SoCs design. In particular, the the most used open-source architecture is the ARM AMBA bus [3] [4], that, thanks to its modular design engineered to support high-performance and low power on-chip communications, is today a standard *de facto*, used in most of the SoCs present on the market. The problem with such solutions is that they implements sophisticated features, designed to be used in full-featured high-performance microcontrollers much more complex than our embedded system. In particular, the absence of a pipeline in the our processor prevents the usage of AMBA solutions, that are in fact designed to pipeline all the communications in order to increase the throughput.

The idea was to take inspiration from a standard architecture (AMBA and Avalon, for instance) to build a simpler infrastructure. In particular, the designed architecture preserves a minimal subset of AMBA specifications to build a simpler infrastructure, suitable for our needs, that remains modular and upgradable at the same time.

Figure 3.2 shows the block diagram of the bus system. It is an infrastructure that supports a single master with multiple slaves, thus no arbitration mechanism has been implemented. The bus interconnection logic consists of a single centralised address decoder and a slave-to-master multiplexor. The decoder monitors the address lines driven by the master so that the appropriate slave is selected during each transaction. It also provides control to the multiplexor, which is in charge of routing the corresponding slave output back to the master.

The bus master interface, on the left of Figure 3.3, provides address and control information to initiate read and write operations. The slave, whose interface is shown on the right of Figure 3.3, responds to transfers initiated by the master, by using a `hsel` (chip select) signal from the decoder to control when to respond a request. When selected, the slave will start monitoring the bus lines in order to respond to commands from the master. Therefore, all the slaves will stay into an idle state until they are individually waken up by a signal.

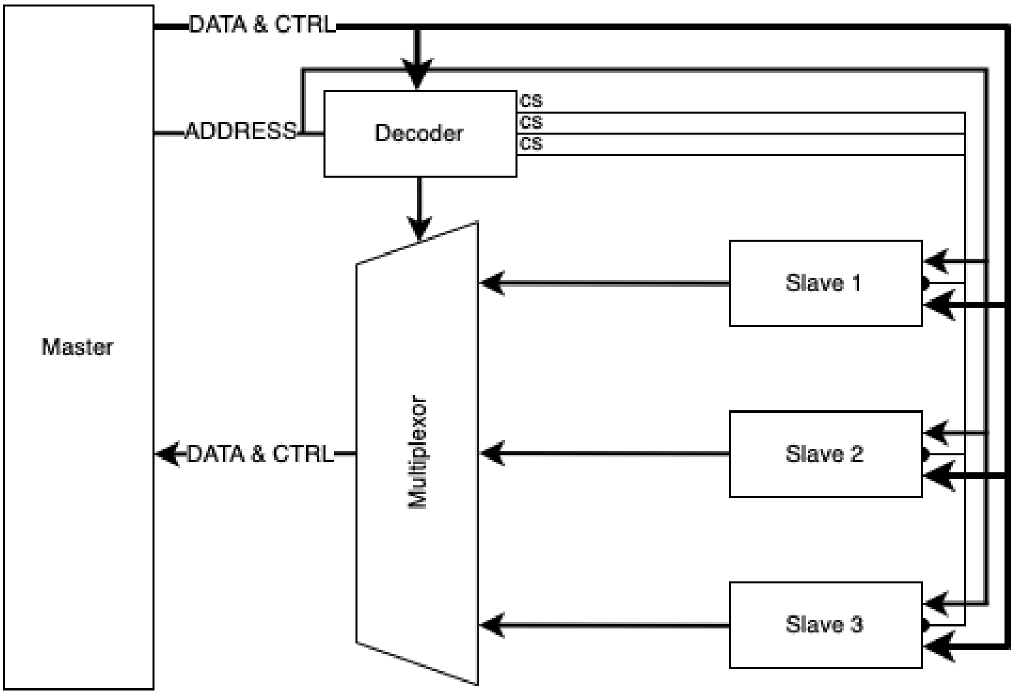


Figure 3.2. MC2101 Bus Block Diagram.

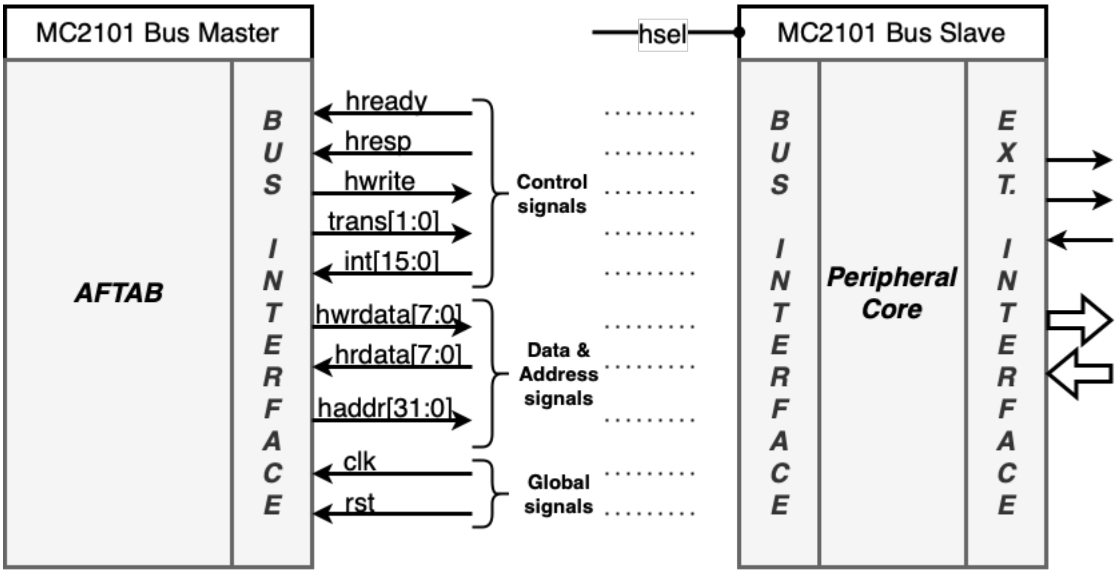


Figure 3.3. Master & Slave Interfaces.

Each slave is also able to feedback the master about:

- the success
- the failure
- or the waiting of the data transfer

by using the `hresp` and `hready` dedicated lines. In this way, a slave is able to extend the data phase when extra time is needed, but also to inform the rest of the system that some bad operations are happening in the bus. The full list of signals is described in Table 3.1.

Name	Type	Description
`hready`	Control	When driven LOW, the transfer is extended.
`hresp`	Control	When HIGH, indicates that the transfer status is on error.
`hwrite`	Control	Indicates the transfer direction. When HIGH the signal implies a write transfer, on the contrary when LOW a read transfer.
`htrans[1:0]`	Control	Shows the current state of the bus.
`int[15:0]`	Control	Independent interrupt lines, can be driven by peripherals to issue a IRQs.
`hwrdata[7:0]`	Data	8-bit data lines from the master to the slaves.
`hrdata[7:0]`	Data	8-bit data lines from the slave to the master.
`haddr[31:0]`	Address	Address space is on 32-bit, thus also the address line.
`clk`	Global	global clock signal
`rst`	Global	global reset signal

Table 3.1. Bus Signals.

In the following Subsections, the main features implemented in the GPIO and UART peripherals are presented. Together, they provide a set of minimal functionalities for interacting with the microcontroller from the external world.

3.1.2 GPIO Peripheral

The GPIO (General Purpose Input Output) is a peripheral module present in all embedded processors. It is used to manage sets of SoC's incoming and outgoing digital signals, by driving and checking the logic state of physical pins. GPIOs can be used in a diverse variety of applications, limited only by the electrical and timing specifications of the peripheral's interface, and the ability of software to interact with it in a sufficiently timely manner. In most of the cases, GPIOs are used to switch LEDs, interface the microcontroller to buttons, user-selectable switches or electronic switches (relays). In other cases, there is also the possibility to use GPIO as a bit banging communication interface, where software is used as substitute for dedicated hardware in order to implement a specific communication protocol, e.g., a software-based SPI bus with 4 GPIO pins.

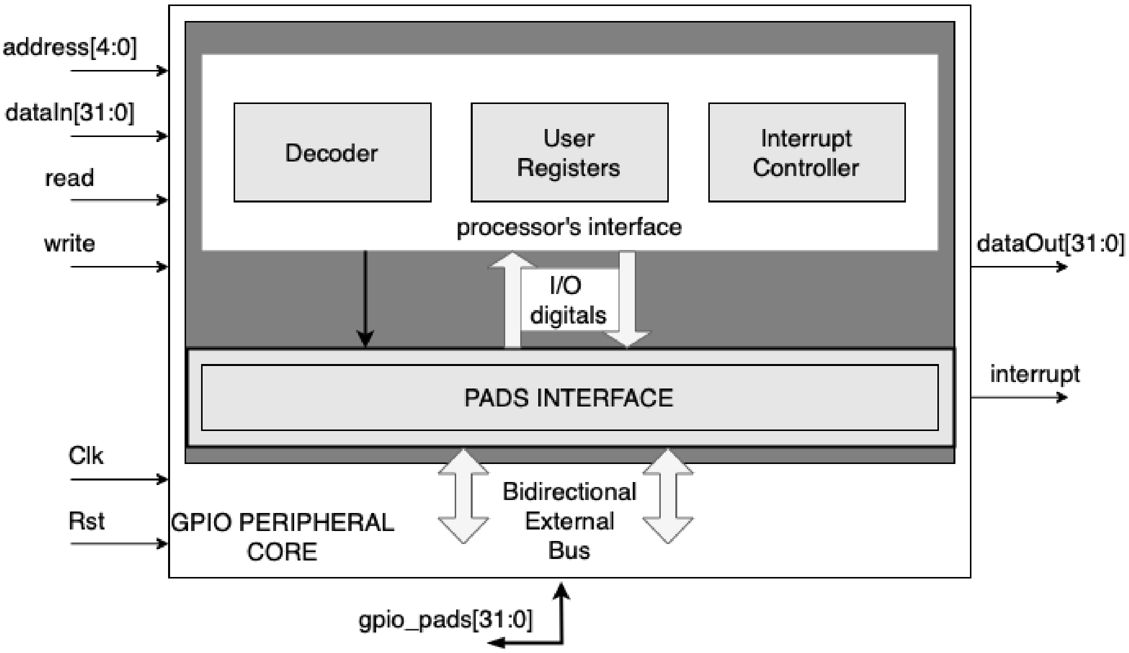


Figure 3.4. MC2101 GPIO Peripheral Core.

In Figure 3.4, the core of the peripheral is showed, together with its relative set of signals coming from the Bus Interface and from the External Interface. The GPIO is designed as a 32-bit peripheral, and so it provides a 32-bit processor's interface with 32-bit wide user registers and data bus (`dataIn` and `dataOut`). The conflict between MC2101's data bus, that is on 8-bit (see `hwrdata` and `hrdata` in Table 3.1), and the peripheral's data bus are solved by the Bus Interface, which provides a bridge for the communications between the two domains. The External Interface (in Figure 3.4 as *Pads Interface*) works as an intermediate between the bidirectional bus connected to physical pins and the peripheral's core. In particular, through the Pads Interface, the asynchronous bidirectional external lines (`gpio_pads`) are separated into independent input and output lines, synchronised with the global clock.

The 5-bit `address` signal selects which user register is being access by the by the processor. As explained before, read and write operations on peripherals are enabled by a proper chip select signal. When the chip select condition is met, the Bus Interface rises one of the two strobe signals, `read` and `write`, for accessing the user registers. The GPIO provides also an `interrupt` line to the processor. This line will rise as soon as an interrupt condition occurs on any of the 32 pad lines. When this happen, the processor will have to execute appropriate read or write operations to deassert the interrupt. The peripheral, programmed through the user registers, is able to support the following functionalities:

- Control the input/output direction of each GPIO pads;
- Enable interrupts for each input bits and configure the triggering behaviour on logic levels or rising/falling edges;
- Drive and control external pins.

Name	Address	Access	Description
`PADDIR`	`0x1A100000`	R/W	Control the direction of each of the GPIO pads. A value of 1 means the pin is configured as output.
`PADIN`	`0x1A100004`	R	Saves the input values coming from input pins.
`PADOUT`	`0x1A100008`	R/W	Drives the output lines with its content.
`PADINTEN`	`0x1A10000C`	R/W	Interrupt enable bits for input lines.
`INTTYPE`	`0x1A100010`	R/W	Two registers: `INTTYPE0`, `INTTYPE1` that are used to control the interrupt triggering behavior of each interrupt-enabled pin.
`INTSTATUS`	`0x1A100018`	R	Contains interrupt status for each GPIO line. The interrupt line is high when a bit is set in this register and will be de-asserted when this register is read.

Table 3.2. GPIO User Registers.

From the programmer's point of view, the GPIO peripheral can be programmed through the set of registers in Table 3.2. More details on the software are provided in the next Section, dedicated to the software part.

3.1.3 UART Peripheral

Communication protocols play very important role in organising communication between devices, which is a fundamental feature to be implemented, as it allows a way for interaction between different platforms. In fact, every embedded system includes at least one hardware peripheral dedicated for this role. Microcontrollers and computers mostly use UART as a form of device-to-device communication protocol, which only requires two wires to implement transmission and reception of data.

The UART module designed for MC2101, whose block diagram is shown in Figure 3.5, provides a transmitter-receiver pair, configurable for different speeds, data widths, parity codifications and information status for several error conditions. The implementation provides a subset of the standard UART 16550 specifications [9], without including some more advanced functionalities for supporting DMA and MODEM communications. The module is designed as an 8-bit peripheral and so it provides an 8-bit data interface to the processor. This characteristic makes the Bus Interface more lighter than the GPIO's

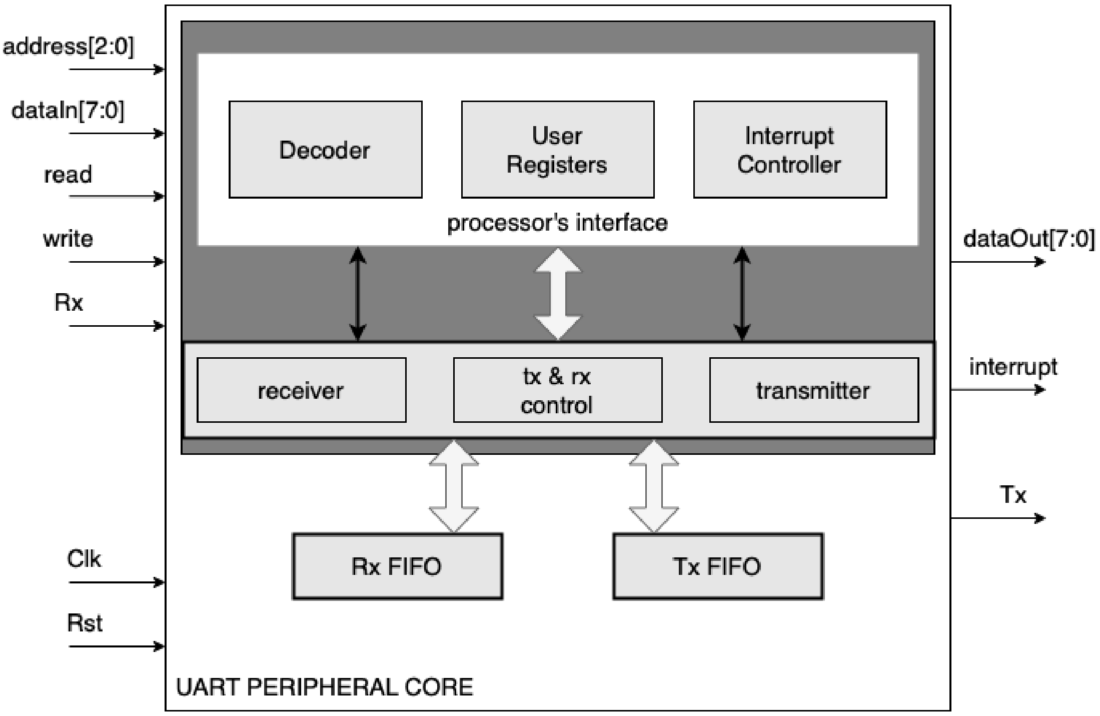


Figure 3.5. MC2101 UART Peripheral Core.

one, because in this case UART's internal data lines are already compatible with the bus infrastructure of the microcontroller. Also the External Interface is lighter because only two external lines, `rx` and `x`, are controlled. Similarly to the GPIO, the signals in Figure 3.5 are generated by the Bus Interface, except for the `rx` and `tx` lines which came from the External Interface, with the difference that, in this case, the user register file is more compact and requires only 3-bit for the addressing. Both receiver and transmitter use a dedicated queue, implemented in hardware as a FIFO memory, used to hold data either received from the `rx` serial port or to be written to the `tx` serial port. This buffering feature is particularly useful when the interrupt mechanism is enabled at the receiver side, which can raise interrupts when its queue surpasses a certain fill level, instead of triggering every time a new character is received. The processor can also benefit from this buffering feature by filling the transmitter's FIFO when multiple character must be transmitted, without the need of wasting polling cycles in waiting for the completed transmission of each character sent. With both the FIFO empty, is possible to have 17 characters simultaneously: in the transmitter, 1 being sent and 16 buffered, while in the receiver, 16 ready to be read and 1 being assembled.

As anticipated, parity conditions and error controls are part of the implemented functionalities. The following error conditions can be detected by the receiver:

- **Break Interrupt**: Error flag asserted when the `rx` line remained stuck at 0 for the entire character time. This error is usually generated from an incorrect wiring, e.g., the `rx` or `tx` line are mistakenly wired to a ground pin.

- **Framing Error**: Asserted when the stop bit was not detected. Usually this type of error is generated when a device is sending data at a different speed with respect to the one used by the receiving device for sampling.
- **Parity Error**: Asserted when the parity of the received character is wrong according to the current one configured. This provides a very simple and useful error detection feature.
- **Overrun Error**: Produced when a character is assembled but there is no more space inside the receiver's FIFO. The UART peripheral must be able to inform the processor that it will lose data if the FIFO is not read.

Regarding the interrupts, the UART module can be configured to assert an interrupt when different conditions are detected, each one with an associated priority.

Name	**Priority**	**Description**
Receiver Line Status	Level 1 (max.)	There is an overrun error, parity error, framing error or break interrupt indication in the received data on the top of receiver's FIFO.
Received Data Ready	Level 2	The number of characters in the reception FIFO is equal or grater than the programmed trigger level.
Reception Timeout	Level 2	There is at least one character in the receiver's FIFO and during a time corresponding to four characters at the selected baud rate no new character has been received and no reading has been executed on the receiver's FIFO.
Transmitter Empty	Level 3	The transmitter's FIFO is empty.

Table 3.3. UART Interrupt Sources.

The Table 3.3 summarises the different conditions that can be a source of interrupt and their relative priorities. From Table 3.4 is possible to see that all registers are on 8-bit and in fact are all byte aligned, with respect to GPIO register file which is word aligned, because all registers are on 32-bit. Another detail is in the RHR and the THR registers, it is possible to see that they are on the same address. This is not a typo. Since RHR is accessed for reading and THR is accessed for writing, it is possible to use the same address for accessing both in an exclusive way. In particular, during a read operation RHR is accessed, instead, during a write operation the THR is addressed. This trick allows to include all 9 registers in a space that theoretically can only be allocated for 8, saving a bit in the address line.

Name	Address	Access	Description
IER	0x1A100000	R/W	Used to individually enable each of the possible interrupt sources.
ISR	0x1A100001	R	Used to identify the interrupt with the highest priority that is currently pending.
FCR	0x1A100002	W	Used to reset the FIFOs and program the receiver trigger level for the Received Data Ready interrupt.
LCR	0x1A100003	R/W	This register controls the way in which transmitted characters are serialized and received characters are assembled and checked.
LSR	0x1A100004	R	Used to inform the user about the status of the transmitter and the receiver.
DLL & DLM	0x1A100005	R/W	Two different registers that together form the 16-bit Divisor Latch, which contains the divisor value used to program the baudrate of the communications.
RHR	0x1A100007	R	Contains the most recent received character.
THR	0x1A100007	W	Contains the character to be transmitted.

Table 3.4. UART User Registers.

The UART peripheral can be programmed through the set of registers in Table 3.4. Also in this case, more details about the software are given in the next Section, dedicated to the software part.

3.2 Software libraries

This Section is dedicated to describe the functionalities of the microcontroller from the software point of view. A proper set of libraries is included in the design with the purpose to facilitate the programming of the microcontroller, but also to integrate the possibility to use all the classical string manipulation functions as well as the `printf` and `scanf` that are useful also for testing activities.

Before entering into details, the first thing that usually is presented when talking about the programmer's point of view is the memory map of the microcontroller. MC2101 architecture supports a 32-bit address space, which virtually corresponds to a 4 GB memory.

Figure 3.6 shows the default memory map of MC2101. By carefully observing the size of the memory sections it is possible to note that the physical portion of memory currently mapped is in the order of KB, there is a huge free space that can be used in future expansions. In particular the main memory has a lot of available space for being extended to support more advanced software implementations, e.g., hosting an operating system.

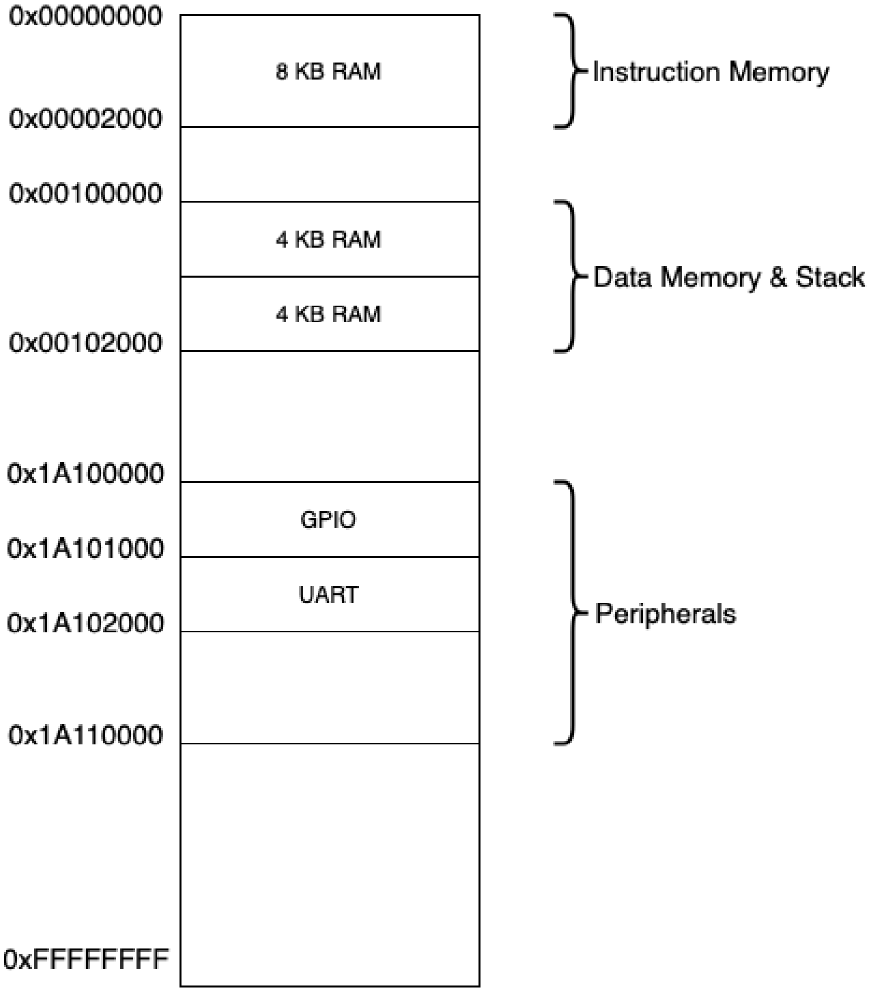


Figure 3.6. MC2101 Memory Map.

The following Subsections are presenting all the possible high-level functions that a programmer can use to write programs for MC2101, maintaining a certain degree of abstraction from the hardware. For this purpose, the GPIO and UART peripherals libraries include a wide amount of functions that, together with macro definitions, allow to program all the functionalities without the need of manually access the user registers manually.

3.2.1 GPIO Library

The Table 3.5 shows a simplified prototype and explanation of the GPIO functions currently implemented in the library. The library implements all the necessary functions to be used for programming the peripheral without the need to directly access its registers, thus providing an adequate level of abstraction. The library offers the possibility to configure

Function	**Description**
`set_pin_direction()`	Used to set a pin direction to input or output.
`get_pin_direction()`	Returns the direction of a given pin.
`set_pin_value()`	Used to set a pin voltage level to low/high.
`get_pin_value()`	Returns a given pin's voltage level.
`set_pin_irq_enable()`	Enable or disable interrupt on a pin.
`get_pin_irq_enable()`	Get the programmed pin's interrupt enable flag.
`set_pin_irq_type()`	Used to configure the interrupt triggering behavior for a given pin. {Logic Levels or Edges}. Pin must have its interrupt flag enabled.
`get_pin_irq_type()`	Returns the programmed pin's interrupt triggering behavior.
`get_gpio_irq_status()`	Returns GPIO's current interrupt status register (`INTSTATUS`) value. Responsible also to deassert the GPIO pending interrupt.
`ISR_GPIO()`	GPIO interrupt handler. When the interrupt is raised, the bootloader will jump to this function.

Table 3.5. GPIO Library functions.

the direction of each of the 32 pins, read and write values on them. But also, provide a set of functions to be used to enable interrupts on any pin and write a custom interrupt service routine.

3.2.2 UART Library

The UART library offers a bigger set of functions with respect to the GPIO library, functions that are also more complex (in terms of execution time) because also the UART features are more complicated. The library provides the possibility to configure the speed of the communications, the size of the frames transmitted/received as well as all the error detection mechanisms. Provides to the programmer functions able to send single characters or strings, and to customise the interrupt behavior.
More details about the library are reported in Table 3.6.

3.2.3 String Library

More advanced I/O functionalities have been implemented in the *string* library. This library is particularly important, because it provides the principal string manipulations functions that together with the UART Library are able to support the `printf` and `scanf` functions. The library is the same implemented in PULPino, with the addition of the

Function	**Description**
uart_set_cfg()	Used to program the LCR register for configuring the character width, number of stop bits, parity type and enable, even/odd parity and the baudrate.
uart_get_cfg()	Return the current LCR register value.
uart_set_int_en()	Configure the IER register to enable the different type of interrupt sources.
uart_get_int_en()	Used to get the enabled interrupt sources by reading the IER content.
uart_rx_rst()	Clear the content of the receiver's FIFO.
uart_tx_rst()	Clear the content of the transmitter's FIFO.
uart_set_trigger_lv()	Set the receiver's FIFO trigger level.
uart_get_lsr()	Used to read the Line Status Register (LSR).
uart_get_isr()	Used to read the Interrupt Status Register (ISR).
uart_sendchar()	Send a character on the transmitter line.
uart_getchar()	Get the character received.
uart_send()	Used to send a string on the transmitter line.
ISR_UART()	UART interrupt handler. When the interrupt is raised, the bootloader will jump to this function.

Table 3.6. UART Library functions.

`scanf` function and some other utilities. The following functions are part of the system's library: `strlen`, `strcpy`, `strcmp`, `puts`, `putchar`, `memset`, `printf` and finally the `scanf`.

3.2.4 Board Library

This is the "top-level" library that should be included in every application developed for MC2101. It contains the `board_setup` function, which was prepared right after the pin planning phase. It is used to initialize the microcontroller hardware once synthesized on FPGA and connected with the various user buttons, switches and LEDs present in the board.

Once called, the microcontroller is configured in this way:

- UART peripheral is programmed with 115200 standard baudrate, no parity, 1 stop bit, 8-bit character width. This is a standard configuration for the `printf` and `scanf` functions.
- GPIO lines interconnected to LEDs are set as output.

- GPIO lines interconnected to User buttons are set as input.
- GPIO lines interconnected to Switches are set as input.
- Al interrupts are disabled.

The `board_setup` function should be the first called in every application, because brings the microcontroller in the correct configuration, according to the pin assignment used for the synthesis, in order to properly interface the external hardware.

3.3 Testing

The microcontroller has been tested with the combination of RTL testbenches and C programs, aimed together to verify the correctness of the architecture. The original AFTAB simulation environment, similar to PULPino, has been adopted and extended for the test of MC2101.

As anticipated, the software environment integrates the RISC-V toolchain with ModelSim commands using CMake build automation tool. In this way, it is possible to compile custom C applications, HDL design files, and also run RTL simulations on ModelSim.

The test of the architecture has been separated into two distinct phases:

1. Testing while-developing.
2. Testing post-synthesis on FPGA.

3.3.1 Testing while-developing

Test activities and design activities happened in parallel during the RTL development of the microcontroller. All the components have been implemented using a structural approach in such a way to have a modular design, easily testable with a bottom-up methodology.

Figure 3.7 shows what is the typical structure used to design all the peripherals. There is always a top-level entity, that works as a wrapper for the bus controller and the peripheral core, which is itself composed of discrete interconnected HDL entities. Each of the entities is at first tested in isolation, then in integration with the other components until the design is functionally tested from the bottom to the top-level entity. The peripheral can be then attached to the bus, through the top-level wrapper, and be finally tested with a proper C/Assembly program to verify the correctness of the interconnection with the processor and the memory. With this approach, the microcontroller can be testes bottom-up, and debugged with custom applications. After these steps, the final test would be to proceed with the synthesis of the design and perform a last RTL test with a gate-level simulation, to properly check that timing constraints are really respected.

3.3.2 FPGA Tests

The test procedure explained before is not actually able to fully test all the functionalities because of the limitations of RTL simulations. Interrupts for instance, are ver difficult to test, because they should be generated asynchronously by the external environment, e.g., a

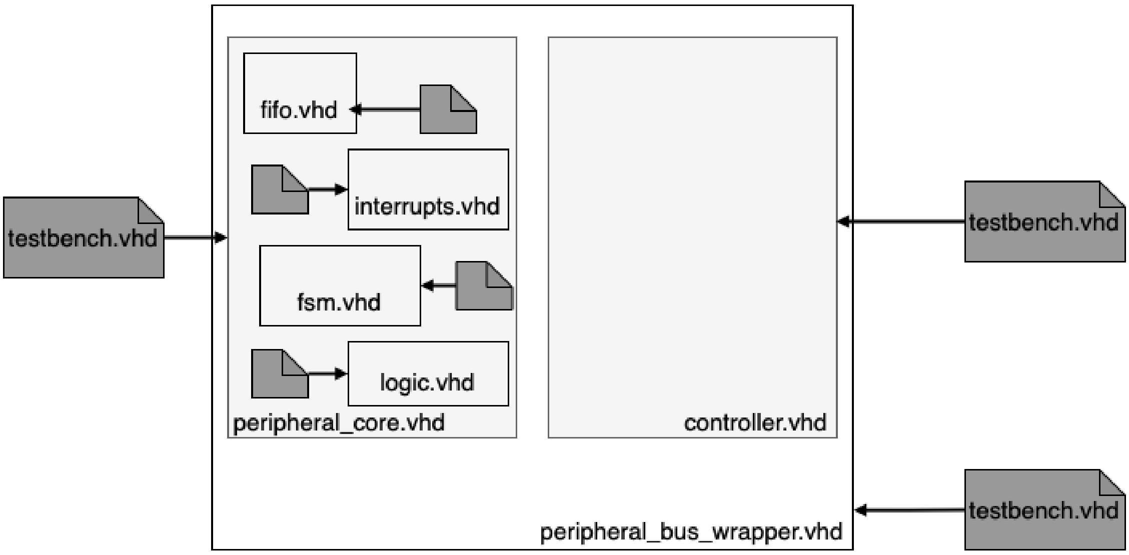


Figure 3.7. Modular structure of a peripheral.

button press. Other behaviors like the `printf` or `scanf` functions would require very long simulation times (in the order of ms) that can really overload ModelSim, with the risk of crashing the workstation.

In general, the functional properties of the microcontroller can be fully tested only when synthesized on FPGA, that in our case is the Cyclone-V FPGA embedded into the DE1-SoC development board. With the purpose of having a proper test infrastructure, a set of make targets, integrating Quartus commands with tcl and bash scripts have been prepared in order to provide to the user a simple, fast and automatic toolchain for the synthesis and deployment on FPGA. The most useful feature of the aforementioned toolchain, is certainly the possibility of updating the memory content of the sythesized microcontroller with new programs, without having to repeat the entire synthesis process from scratch, which can be quite time consuming.

Below, are presented some more details about the testing of the UART and GPIO peripherals, thus are also showed the different ways of interacting with the microcontroller.

The GPIO peripheral is interconnected with a proper set of component present on the DE1-SoC. As shown in Figure 3.8, there are 10 LEDs in total, 3 push buttons, and 10 switches that are physically connected with the GPIO's External Interface lines. This set of electronic components are very useful in order to test the behavior of the peripheral, the user can assess if the peripheral works or not by just trying to turn on/off LEDs. Test programs have been implemented with the purpose to check the functional correctness of the the whole peripheral and the interrupt mechanism.

The UART peripheral has been tested together with the `printf` and `scanf` functions. In this case, the test is a little bit more complicated, as it requires the usage of a Terminal Emulator in order to see if the I/O functions are working.

In Figure 3.9, the communication mechanism designed for interconnecting the UART

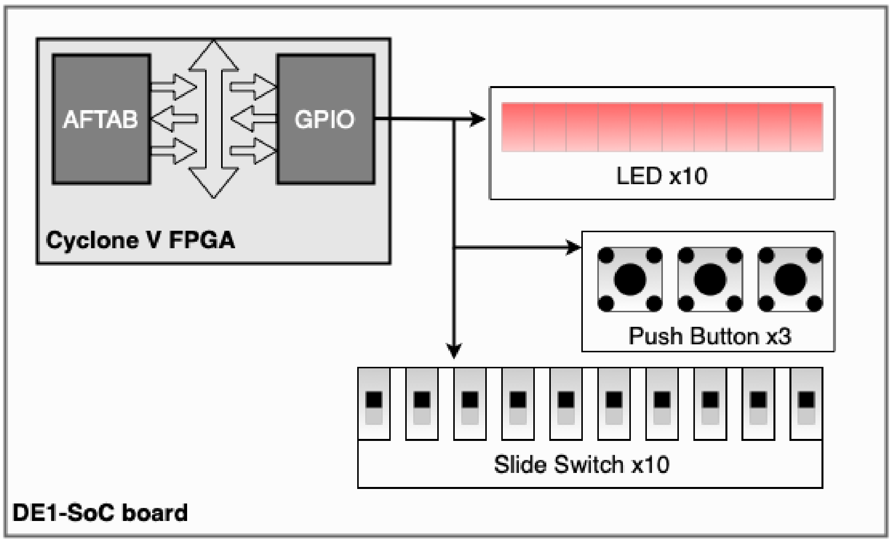


Figure 3.8. GPIO Interconnection on DE1-SoC.

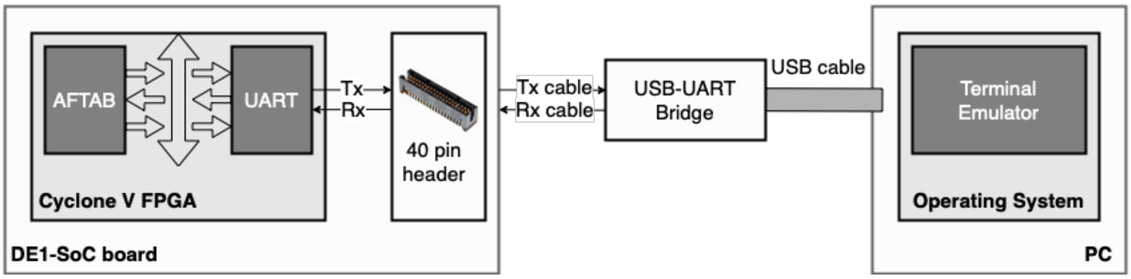


Figure 3.9. UART Interconnections.

peripheral with an external PC is shown. The interconnection is very simple, the `tx` and `rx` lines have been routed with 2 header pins of the DE1-SoC, then two cables are used to connect the `tx` and `rx` lines to a USB-UART bridge that is directly plugged into USB Port of a computer. In this way, every Terminal Emulator is able to print the character transmitted by the microcontroller and also send character to the microcontroller. This communication mechanism, together with the GPIO connections allow a proper test of the entire microcontroller.

Chapter 4

Experimental Results

The aim of this Chapter is to show the results obtained from the design choices implemented. In particular, the specifications of the systems are evaluated as well as the FPGA resource usage, which is very useful to know in order to evaluate which future extensions to implement.

Starting from the specifications, as anticipated, the purpose of this project was to design a simple, modular, and extensible microcontroller able to provide all I/O functionalities to support its usage in a realistic environment. All tests performed have shown that the system fulfils all of the requirements. In fact, MC2101 is able to interface correctly with all of the interconnected hardware components of the DE1-SoC. The infrastructure built for bridging the communications between the UART and a Terminal Emulator allows the user to interact correctly with the system by using any platform that have a USB driver inside. Assessed that also the interrupt mechanism fully responds to external triggers, the system requirements have been fully fulfilled.

Thanks to the fact that the main memory has been designed to be synthesised in the embedded M10K blocks of the Cyclone-V FPGA, it is possible to extend the available RAM without wasting LUTs. Considering that only 3% of the M10K blocks available on the target FPGA are currently used, and that the architecture actually takes advantage of a very small portion of the addressable space there is plenty of room for future developments.

Resource Name	**Used Amount**	**Total Amount**	**Percentage Used**
ALM	2628	32070	8%
FF	3443	64140	5%
PIN	36	457	8%
M10K Bit	131072	4065280	3%
RAM Block	16	397	4%

Table 4.1. MC2101 resource usage on Cyclone-V FPGA.

More details about FPGA resource usage are reported in Table 4.1. It is possible to see that the architecture of MC2101 takes advantage of a very small percentage of the available resources on the Cyclone-V, which allows a great deal of freedom for future extensions.

To conclude the discussion, it is useful also to compare the results of the MC2101 synthesis with those of PULPino, which has been the reference architecture and will still be source of inspiration for future developments. Unfortunately, following several attempts, was not possible to correctly synthesize PULPino on the Cyclone-V using Quartus software.

This is due to the outdated release of the software, that does not support some of the SystemVerilog HDL structures coded inside PULPino. For instance, the memories in PULPino are designed to be synthesized within Vivado HLS for Xilix FPGAs that are not supported in Quartus. Because it was not possible to synthesize PULPino using MC2101 synthesis environment, the two architectures have been synthesized using Vivado HLS in order to have a report generated by the same synthesis software and also in order to have a comparison between the architecture that is useful to understand what results to expect from future developments of our microcontroller.

Resource Name	**PULPino Usage**	**MC2101 Usage**
LUT	15657	3440
FF	9883	3253
PINS	143	36
BRAM	16	4

Table 4.2. PULPino and MC2101 resource usage comparison on Artix-7 FPGA.

Table 4.2 shows the resource usage of both MC2101 and PULPino synthesized on the Artix-7 FPGA using Vivado HLS. As expected, because PULPino implements a typical AMBA-based full-featured design, it is a much more complex embedded system than the current MC2101 and so the resources required for the synthesis are much higher. The memory blocks for instance are 16 instead of 4 because PULPino makes use of 64 KB memory, divided into instruction RAM and data RAM, while MC2101 only uses one 16 KB single port memory. The higher number of peripherals, the pipelined architecture of the processor and the higher complexity of the bus infrastructure are the reasons of which PULPino needs more sequential elements (FF) as well as LUT to implement all the logic required.

Chapter 5

Conclusions and Future Work

The goal of this thesis was to provide my research team a RISC-V microcontroller written in VHDL that can be synthesized and usable on FPGA, where is possible to test, integrate and evaluate hardware and software security solutions for embedded systems. The work carried out includes all the functions necessary to automate the synthesis processes, the RTL simulations, the compilation of custom applications and a soft architecture able to run software on FPGA. The architecture designed integrates all the necessary hardware modules e their software libraries required to support all the basic I/O functionalities that permit its usage on a development board. All tests performed have demonstrated the ability of the system to perform all the functions set up, and therefore to be able to satisfy the initial requirements.

The actual state of the platform offers a good starting point for future improvements. In particular, the following are some of the most relevant features that should be included in future extensions in order to have a more complete microcontroller able to run general purpose software:

- **Pipelining**: Up to now, all the operations are executed and issued in sequence, thus no pipelined behavior is supported yet and so the performances are the main limitation of this device. Introducing the pipeline into core together with a proper upgrade of the bus infrastructure is indeed a precious feature to be included. This upgrade will definitely increase the computing capability of the embedded system and therefore the possibility to run a wider range of applications.

- **ISA Extensions**: In order to support a wider range of applications, the ISA itself must be extended. In particular, for being able to support general purpose computing all the standard extensions *M, A, F, D, C* should be included. Considering that *I* and *M* are already part of the ISA, withe the remaining three standard extensions *F, D, C* the processor will be able to execute floating point single precision (F) and double precision (D) and also the compressed instructions (C).

- **OS Support**: Currently AFTAB is not able to support the *Supervisor* mode required to run an operating system. Thus, in order to be able to support the full software stack separation required to run an operating system, the ISA must include also the

S privileged mode and all the required CSR registers. This upgrade, together with the introduction of all the ISA Standard Extensions, would allow the system to run a wide range of general purpose applications.

- **Timer peripheral**: To extend the platform functionalities, there are a lot of peripherals that could be added. For instance, as in PULPino microcontroller, some of the peripherals missing in MC2101 are the SPI, I2C modules and the Timer. What i would suggest more is for sure the Timer peripheral, which is used to trigger periodic interrupts with a very high timing precision. A periodic interrupt can be used for example by the operating system to control the scheduling of processes with a very high timing precision, mandatory feature when real-time constrains must be met. Another example of period interrupt usage is for example the Watchdog Timer, used to generate a non-maskable interrupt that resets the entire system if not handled. This feature is a must, because allows to recover after some bad internal failure occurs and no user intervention is possible.
- **Debug Interface**: All modern MCUs feature an on-chip debug module that opens up the access to all the addressable space, including CPU registers. This capability allows to debug the code execution, that actually in MC2101 can only be performed by dumping contents with the printf function.

To conclude, I can say I am satisfied about this thesis experience that allowed me to design the architecture of a system where it is possible to execute real software. I had the opportunity to deepen my skills into the low level design of embedded systems and complete my knowledge on microcontrollers architecture, for which i have grown a great deal of interest during my studies and which i have been able to deepen thanks to this experience. In this regard, I would like to underline the importance of the RISC-V ISA in the world of research and academia and in general the importance of having open-source materials on which is possible to learn, get inspired and create original contribute for the community. As the last thing, I would like to express my gratitude to Gianluca, who constantly supported me during my work, and to Professor Prinetto, who gave me this opportunity.

Bibliography

[1] CHIPS Alliance. *Chisel 3: A Modern Hardware Design Language*. https://github.com/chipsalliance/chisel3. 2022.

[2] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, Krste Asanović. *The RISC-V Privileged Architecture Version 1.9.1*. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.pdf. 2016.

[3] ARM. *AMBA AHB Protocol Specification*. https://developer.arm.com/documentation/ihi0033/latest/. 2021.

[4] ARM. *AMBA APB Protocol Specification*. https://developer.arm.com/documentation/ihi0024/latest/. 2021.

[5] CINI Cybersecurity National Laboratory, University of Tehran. *CNL RISC-V AFTAB*. https://github.com/RHESGroup/aftab. 2022.

[6] CINI Cybersecurity National Laboratory, University of Tehran. *CNL RISC-V AFTAB Microprocessor User Manual*. https://github.com/RHESGroup/aftab/blob/master/doc/aftab_user_manual.pdf. 2022.

[7] RISC-V International. *RISC-V Introduction*. https://riscv.org/wp-content/uploads/2021/08/RISC-V-Introduction-_-Aug-2021.pptx. 2021.

[8] R.E. Kessler. «The Alpha 21264 microprocessor». In: *IEEE Micro* 19.2 (1999), pp. 24–36. DOI: 10.1109/40.755465.

[9] SIDSA. *UART 16550 IP Datasheet*. http://caro.su/msx/ocm_de1/16550.pdf. 2001.

[10] Claire Wolf. *PicoRV32 - A Size-Optimized RISC-V CPU*. https://github.com/YosysHQ/picorv32. 2019.

[11] Yonghong Yan. *Lecture 04 RISC-V ISA, CSCE 513 Computer Architecture*. https://passlab.github.io/CSCE513/notes/lecture04_RISCV_ISA.pdf. 2018.

[12] K.C. Yeager. «The Mips R10000 superscalar microprocessor». In: *IEEE Micro* 16.2 (1996), pp. 28–41. DOI: 10.1109/40.491460.

[13] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. «SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine». In: (2020).

[14] ETH Zurich. *PULPino*. https://github.com/pulp-platform/pulpino. 2019.