



QUENTIN MADARIAGA

NANOTECH

2022

OMEGA MICROELECTRONICS - RTE DE SACLAY, 91120 PALAISEAU

Master Thesis Report

DEVELOPMENT OF A RISC-V MICROPROCESSOR FOR ASIC INTEGRATION

UNDER THE SUPERVISION OF :

FREDERIC DULUCQ , OMEGA MICROELECTRONICS - fdulucq@in2p3.fr

KATELL MORIN-ALLORY, GRENOBLE-INP katell.morin-allory@grenoble-inp.fr

---

Ecole nationale
supérieure de physique,
électronique, matériaux

Phelma
Bât. Grenoble INP - Minatec
3 Parvis Louis Néel - CS 50257
F-38016 Grenoble Cedex 01

Tél +33 (0)4 56 52 91 00
Fax +33 (0)4 56 52 91 03

http://phelma.grenoble-inp.fr

GLOSSARY

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit. As the name suggests,ASICs are destined to execute the task they have been designed for, ranging from simple logic function to very complex processors. Large ASICs are often called SoC.

CISC Complex Instruction Set Computer.

Corner A Corner is defined as a set of libraries characterized for process, voltage, and variations. Corners are not dependent on functionnal settings; they are meant to capture variations in manufacturing process, along with expected variations in the voltage and temperature of environment in which the chip will operate.

CPI Cycles Per Instruction.

CPU Central Processing Unit.

FPGA Field Programmable Gate Array. Opposite concept of an ASIC. FPGAs are fully programmable, which means anything can be emulated by the FPGA, even processors, within the limits of the available hardware one the FPGA chip..

GAFA Google Facebook Apple Facebook(Meta) and Amazon.

GPI General Purpose Input. Term used in this paper to differentiate those one way input from GPIOs.

GPIO General Purpose Input/Output. A GPIO is an uncommitted digital signal pin on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by software..

GPO General Purpose Output. Same as GPIs.

IRQ Interruption ReQuest. A hardware interrupt on a PC: used to signal the CPU that a peripheral event has started or terminated..

ISA Instruction Set Architecture. In computer science, an instruction set architecture, also called computer architecture, is an abstract model of a computer. In general, an ISA defines the supported instructions, data types, registers of such implementation.

LSB Least Significant Bit.

MMMC Multi Mode Multi Corner. See Corner et Mode definitions.

Mode A mode is defined by a set of clocks, suppply voltages, timing constraints, and libraries. It can also have annotation data, such as SDF or parasitics files. Many chip have multiple modes such as functional modes, test mode, sleep mode, jtag mode etc..

MSB Most Significant Bit.

PnR Place and Route.

RISC Reduced Instruction Set Computer.

RTL Register Transfer Level : it is an abstraction for defining the digital portions of a design. When referring to a code (RTL code), it refers to the behavioral code, often written in VHDL, Verilog or SystemVerilog.

SEE Single Effect Event.

SoC System on Chip.

Verilog An Hardware description language, such as VHDL or SystemVerilog.

VHDL VHSIC Hardware Description Language.

Contents

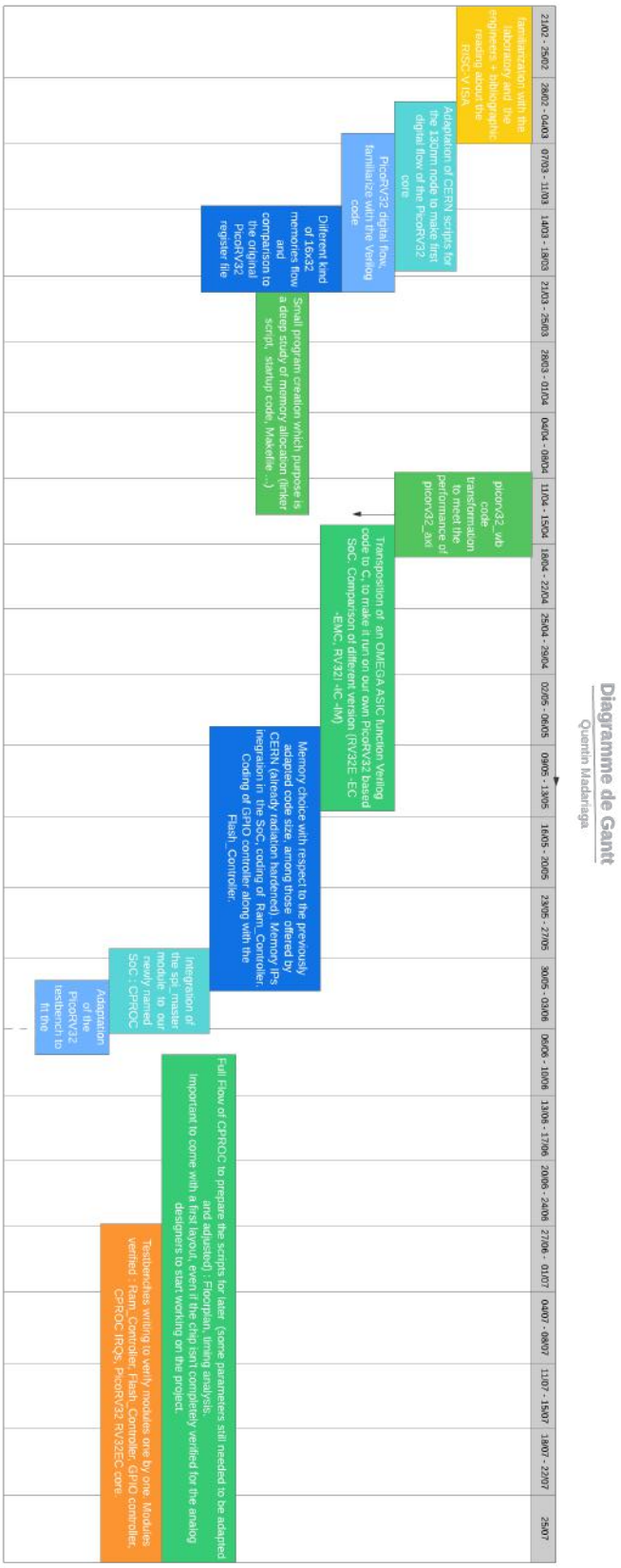

FIGURE 1
Internship Gantt Diagram

CHAPTER 1

INTRODUCTION

1.1 OMEGA MICROELECTRONICS

OMEGA Microelectronics (Organisation Microélectronique Générale Avancée) is a CNRS/IN2P3-Ecole Polytechnique microelectronics design center located in Ecole Polytechnique Campus in Palaiseau, in the southwest suburb of Paris. It was founded in 2013 and is currently composed of a team of around fifteen microelectronics engineers, mostly analog designers. They design sophisticated ASICs for nuclear physics, particle physics and astrophysics detectors, contributing in large scale scientific experimentations like CMS HGCAL at the CERN. These state-of-the-art detectors and signal processing chips require cutting edge technologies. The chips working environment and the particles to be detected being always more evanescent require the chips to be radiation hardened and have high integration degree and performance levels, some chips being driven by clocks with a few GHz magnitude.

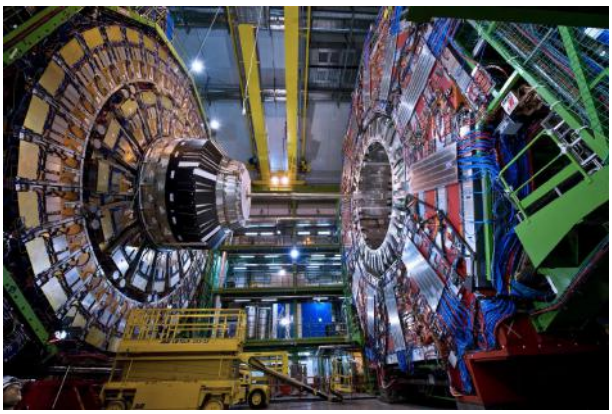

FIGURE 1.1
CMS HGCAL experiment at CERN

The number of chips involved in an experiment can be tremendous (about 10^5 in CMS HGCAL), so one of the main focus is for the chips to be as low power as possible. They range between 1 mW and 15 mW per channels depending on their functionalities. OMEGA currently works on designing chips for three major experiments:

- CMS HGCAL, a high granularity calorimeter at CERN. The chip HGCROC3 designed by

OMEGA provides charge and time measurements for millions of small sensors (channels) with integrated processing at 1,28 Gbps. Figure [3] present the actual calorimeter, 21 meters long, 15 meters wide and high, which will become HGCAL in five years from now.

- Hyper-Kamiokande a neutrino observatory. The chip HKROC is involved in the charge measurement. 3k ASICs would be integrated in this experiment [4].
- ATLAS HGTD a high granularity timing detector, involved the LHC high luminosity upgrade phase. The chip ALTIROC2 is a 225 channels ASIC designed for LGAD (Low Gain Avalanche Diodes) readout, capable of detecting a 2 fC charge with a 95% accuracy [5].

FIGURE 1.2
ATLAS experiment at CERN

1.2 THE RISC-V SPECIFICATION

1.2.1 WHAT IS RISC-V

RISC-V (pronounced "risk-five") is an open standard implementation of a RISC instruction set, but is also the name of the foundation that oversees all the activity around it. RISC-V was developed at UC Berkeley, the same place David Patterson and John Hennessey developped the original RISC chips back in the 1980s. It was the result of an academic exercise, when Krste Asanović, a Berkeley professor was looking for a chip that could be used to help teach his students. All the available chips with their attached ISA had drawbacks for teaching, an ARM or x86 architecture being way too complex. More importantly, these chips design are all proprietary and cannot be used without paying important fees and royalties to their owner. Krste Asanović and his students then started to design their own super basic, efficient and extensible instruction set architecture, resulting in 2010 in the most basic form

FIGURE 1.3
RISC-V Foundation logo

of the RISC-V ISA we know today.

Over the next years, the industrial interest on RISC-V has constantly grown, some companies wishing to exploit this new standard. However commercial users require an ISA to be stable before they can use it in a product that may last many years. To address this issue, the RISC-V Foundation was formed in 2015 to own, maintain, and publish intellectual property related to RISC-V's definition. The Foundation is based in Switzerland since 2019. There are 29 founding members among which are Google, IBM or Qualcomm [11].

1.2.2 WHY USING RISC-V

One thing to know is that there is no major breakthrough with the RISC-V architecture: the RISC-V itself does not represent any new technology. The RISC-V ISA is based on computer architecture ideas that date back at least forty years. What makes it unique is that it's an open standard: both the ISA and some basic chip design are available online. In the era of the IOT (Internet of tings) processors are everywhere, managing power delivery or battery life or processing signals from sensors, and it is not uncommon to have dozen of chips within one component running proprietary instruction set from different companies. These tiny SoC and controllers come from companies like Texas Instrument, Synopsys or Renesas, offering massive catalog of chips. However if one need something they don't offer, unless being a GAFA company and asking for a custom design, it is necessary to find the closest chip that does at least the wanted functionality. The problem with this is that the chip considered will have excess and unwanted hardware, with which comes a bigger footprint and power consumption, problematic when targeting low power and embedded purposes.
It doesn't get easier when trying to design a chip since most companies do not share their ISA. Some companies like ARM do license theirs, but it comes in exchange of hundred thousands of dollars to use them, in addition with royalties and costly updates 1.4.

Tiers	DesignStart	Entry	Standard
Access fees	$0	$75k per annum $0 for startups*	$200k per annum
License fees (due on project manufacture)	Calculated per design based on IP used. **		
Royalty	Calculated per project and paid per unit shipped **		

FIGURE 1.4
ARM ISA licence costs [7]

Supposing one can afford such price, it would take month for the legal end of the licence deal, while lawyers hammers out every details of it, an eternity in the microelectronics field.

The benefit of RISC-V comes right there, any engineer can start designing any RISC-V chip at this very moment and this without any legal problems. Some companies like Western Digital already transitioned numerous controllers in the benefits of RISC-V chips. In this very case, Western digital needed a dual threaded storage controller, which wasn't available on the market, resulting in having two controllers, adding fabrication cost and power consumption to the chip.

1.3 AIM OF THE INTERNSHIP

All the chips currently designed and produced by OMEGA are ASICs that serve a specific purpose and nothing else, their functionality is engraved in silicon. And this has benefits. Since it is designed and optimized to perform one specific task, it is generally very efficient, fast, reliable, and "simple" to use: once it is plugged (meaning correctly biased, bound and no active reset signal), it does what it is designed to do. As stated before, chips designed at OMEGA are intended for particle physics application, more specifically to be integrated in a more complex particle physics experiment. Their functionality, which is a small part of the whole process, is based on calculus and results obtained by physicist that sets the specifications to meet. However some physicists raised a concern: since ASICs take years to design and test, the physics results they're based on could shift in the meantime, making the chip data processing less efficient. Extra calculus units could be added outside the detector, making up for that loss, but adding inevitable hardware and power consumption.
This is why OMEGA's interest on RISC-V processors is growing. The processors have the advantage of being capable of running code that can be replaced at will. They can execute any task within the limits of the available hardware. The objective in the long term is not to replace every chip with microprocessors, since those would be way slower and more power consuming, but rather to replace some part of an ASIC that are based of physic result subject to change.
The short term goal, this internship goal, is to design a microprocessor, based on an existing open source core, able to perform low speed data processing, or to be used as a microcontroller in the control path of a bigger system. The integration environment being limited, the main focus will be to have the smallest footprint as possible, and to be as low power as possible. The number of metal layer available is also limited. Performance is not targeted here, as well as radiation hardening. The reasons of such choices will be addressed later in this paper.

CHAPTER 2

THEORICAL BACKGROUND

2.1 DEFINITIONS

2.1.1 THE CPU

A CPU, also called processor, is the electronic circuitry that executes instructions comprising a computer program. Regardless of the physical implementation of it, its fundamental operation is to execute a sequence of memory stored instructions. It follows this pattern, called the instruction cycle:
Fetch an instruction from the memory,
Decode this instruction,
Execute this instruction,
Store the result in memory [14].
It performs basic arithmetic, logic, controlling, and input/output operations specified by the instructions fetched. After each execution of an instruction, a register called the program counter see its value incremented by a fixed number, and the entire process repeats, the next instruction cycle normally fetching the next-in-sequence instruction defined by the new value of the program counter. Some instructions directly change the value of the program counter without producing any results.

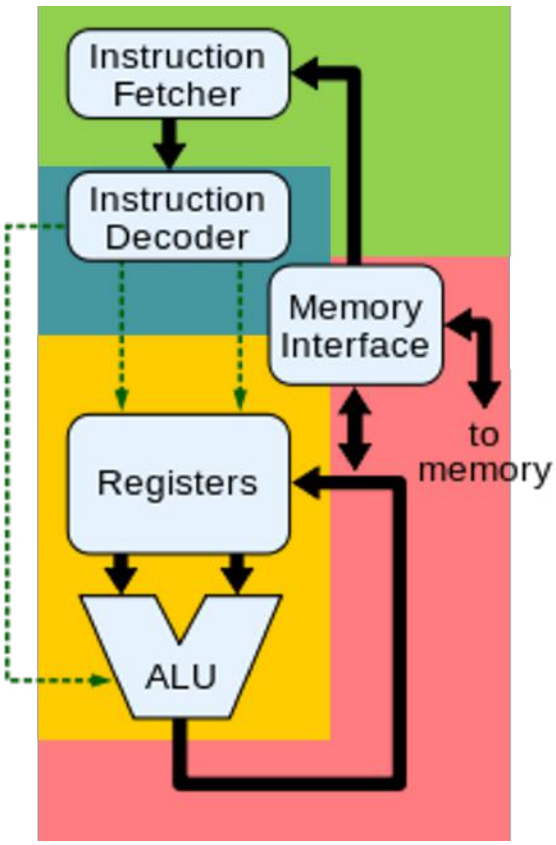

FIGURE 2.1
Simple CPU diagram

There are multiple such instructions, but they are generally referred as "jumps". Any loop, conditional behavior, or instruction to execute a program stored at a certain address, will contain a type of jump instruction.

Fetching is the action to retrieve an instruction at a location (its address) determined by the program counter, which stores a value that identifies the next instruction to be fetched. The fixed number increment after each instruction execution corresponds to the length of instruction just executed, in bytes. Supposing the PC value is currently 100, and the instruction just executed was coded on 4 bytes, the new PC value will be 104.
The **Decode** step is performed by the instruction decoder 2.1. The binary instruction is analyzed by this decoder, and the way in which it is interpreted is defined by the ISA. The instruction often contains significant group of bits such as the opcode that defines the operation to be made, or other information such as the operands or an address.
The **Execute** step is done by the ALU. It is a digital circuit that performs integer arithmetic and bitwise logic operations. Its inputs are the operands, status from the last operation (whether it is finished or not) and the opcode. The initial data and intermediate results are stored in the internal registers 2.1.
Store step is in charge of redirecting the output data to its destination, whether is it to be stored in the memory or displayed at the outputs.

2.1.2 THE ISA

The ISA, also called computer architecture,is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA is the link between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done. It is the only way an external user is able to interact with the hardware, it is the only portion of the machine that is visible to the assembly language programmer, the compiler writer and the application programmer. The ISA defines the supported data types, the internal registers, the memory management that a microprocessor can execute. An ISA is not something immutable, it can always be extended by adding instructions or support for larger addresses and data values, as long as the hardware is also updated.
There are two main types of ISA 2.2: **CISC** and **RISC**. Those two ISAs represent two different approaches. CISC instruction set is mainly exploited by Intel and AMD, with respectively the x86 and x86-64 ISAs. This type of ISA tends to have very specific instructions for a lot of actions, which makes them numerous and long, sometimes coded on several word length. For instance, x86-64 contains 981 different mnemonics and 3684 instruction variants [2]. On the other hand, RISC architectures tends to have way less and simpler instructions, reducing the amount of hardware needed to decode and process the instructions. The CPI is also very low on

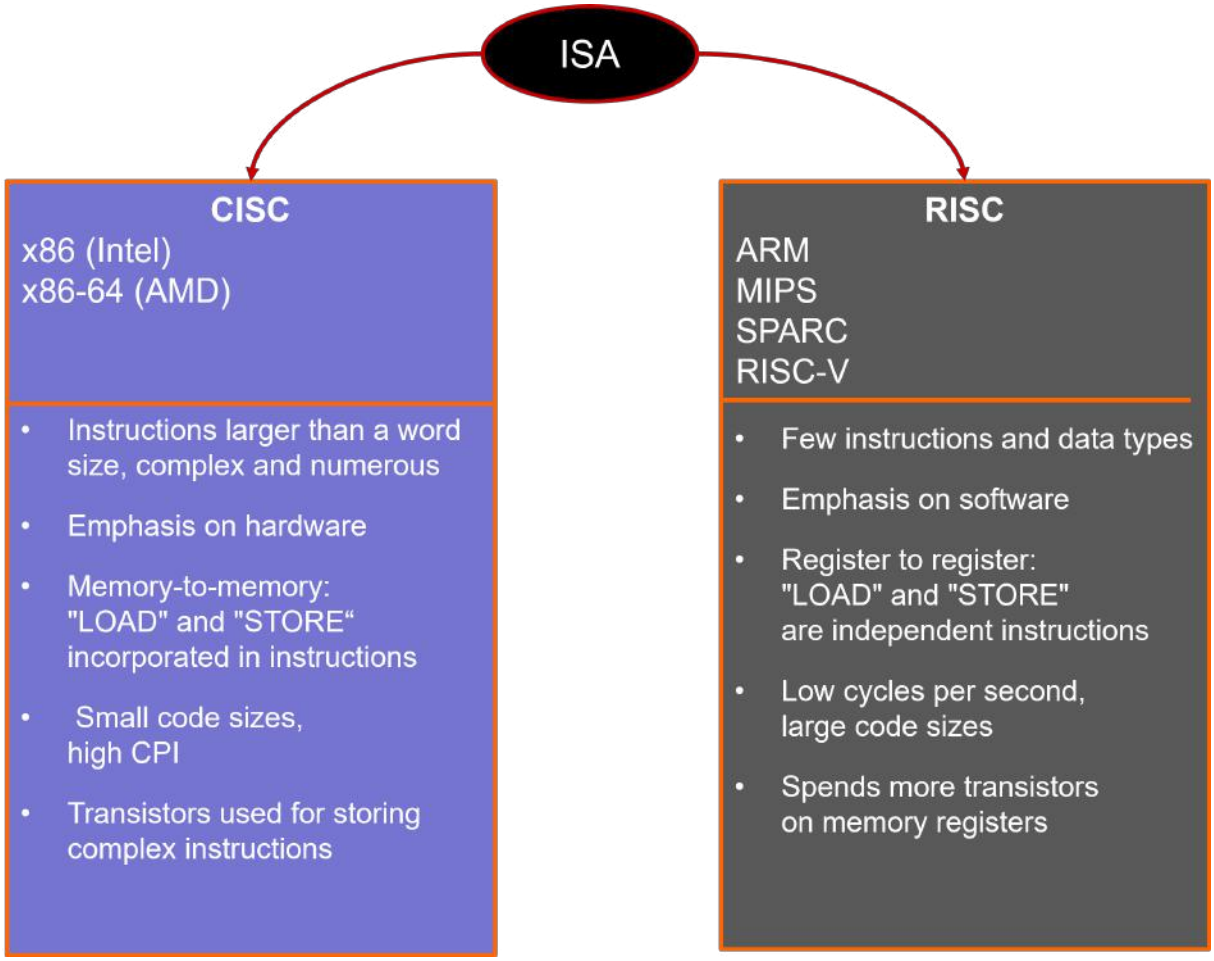

FIGURE 2.2
Two families of ISA

RISC architecture with respect to the CISC one, but for one instruction executed in the CISC case, multiple ones will have to be executed when working with the RISC standard. This means that the software will tends to occupy more space in the memory since there are more instructions for one action, so the hardware economy on decoding and executing is compensated by the hardware needed to have bigger memories. [8]

2.2 RISC-V SPECIFICATION

2.2.1 RISC-V BASE SETS

The RISC-V ISA is fairly recent, which means that the errors from the past from other RISC ISA have been taken into account while designing it. Furthermore, being an open standard, most of the work is done by the community and carried by the foundation, and it is accessible by everyone, implying a fast evolution.
The RISC-V is also a "modular" ISA. When one wish to use this ISA, a choice between four

base set must be first done. Every set is fully documented is the RISC-V instruction set manual [1] Those base sets are:

- RV32I Base Integer Instruction Set
- RV32E Base Integer Instruction Set
- RV64I Base Integer Instruction Set
- RV128I Base Integer Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

FIGURE 2.3
RV32I base instructions

RV32I

This base set features 32 32-bit registers, named x0-x31. x0 is the zero register, always containing zero and not editable. This set has 40 or 47 base instructions depending on the version used. The version we work with in this paper in the one where FENCE and CSSR instructions are not included, consequently having 40 base instructions. In addition to these instructions, there are pseudo-instruction that are not per say considered as instructions, since they are a combination of some. For instance the "nop" pseudo instruction is

addi x0, x0, 0 : adds x0 and 0, stores the value in x0

while the frequently used load immediate "li" pseudo instruction, also called the Myriad sequence and used to load any number in an internal register, has a different combination depending on the number to be loaded. For instance if the value 1024 is to be loaded in the register a0, the li instruction is:

addi a0, 1024 :

but if a value coded on more than 12 bits is to be loaded, such as 2048, the li instruction is:

lui a0, 0x1
addi a0, a0, -2048

The source registers (rs1 and rs2) the destination register (rd) and the opcode are always at the same place in the instructions, but not all the instructions are the same. There are six main types of instruction: R-type, I-type, S-type, B-type, U-type and J-type. The complete listing of these instructions and their format is displayed on figure 2.3.

RV32E

With respect to the base RV32I set, this base set has for only difference the fact that it only uses 16 32-bit registers instead of 32. The instructions are exactly the same, but the available registers to perform the operations now only range from x0 to x15. It is called -E since it is intended for embedded purposes.

RV64I

This base set features 32 64-bit registers. This set shares the same base instructions as the RV32I set, but they deal with 64 bits wide data instead of 32 bits. Furthermore, it contains additional instructions allowing to perform operations on the 32 LSB of a 64 bit register, using the sign extension. It provides a 32 bits result with a sign extension to fill the remaining bit to reach 64 bits.

RV128I

Similarly to the RV64I, the RV128I base set has 32 registers of 128 bits. The instruction are the same than RV64I, but with additional instruction allowing for operations on the 64 LSB, with the sign extension to add up to 128 bits.

2.2.2 THE EXTENSIONS

As stated before, in addition to the base set that must be first chosen, RISC-V allows for adding support to subsidiary extensions to deal with specific operations. There is a total of 17 RISC-V extensions. Only three of them are briefly described in this paper, as they might be considered for the CPU project. Those extensions are:

- "M" Standard Extension for Integer Multiplication and Division
- "C" Standard Extension for Compressed Instructions
- "F" Standard Extension for Single-Precision Floating-Point

It is important to note that using an extension means new instructions with their own format, which also means that the hardware must be capable of decoding and executing them. Every extension used implies an transistor number increase.

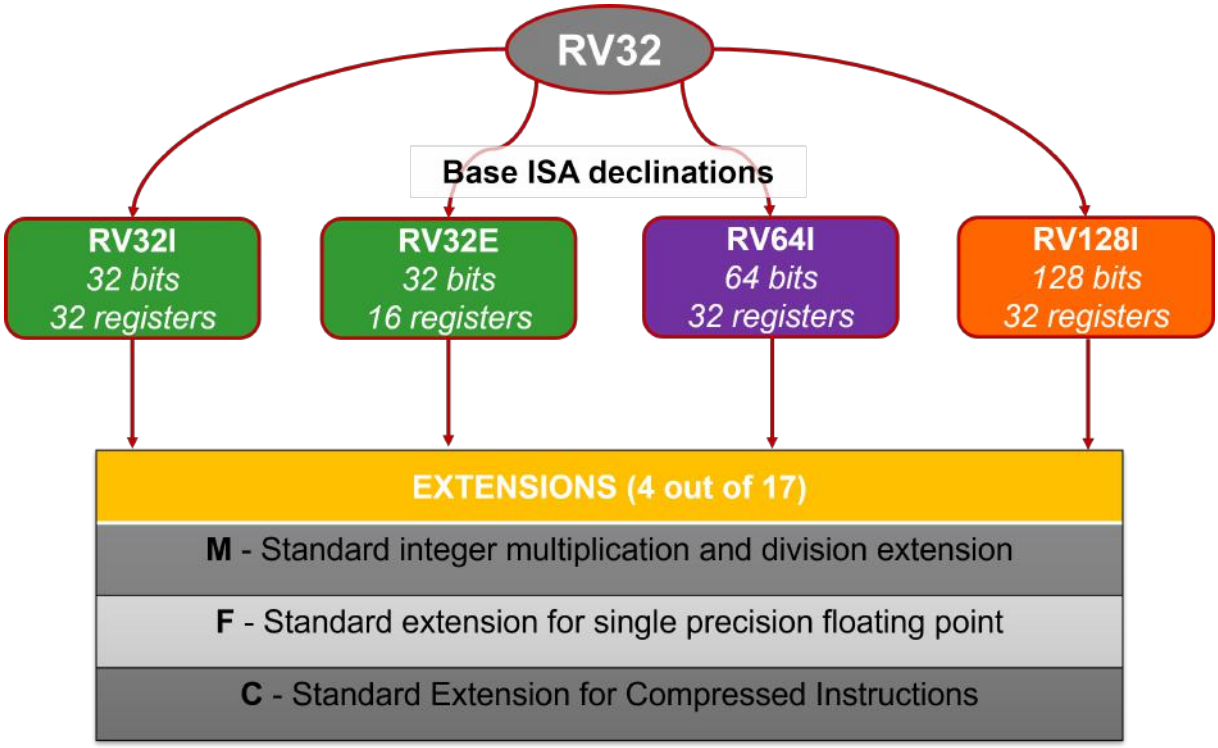

FIGURE 2.4
RISC-V ISA declinations

"M" Standard Extension for Integer Multiplication and Division

This extension contains instructions allowing the multiplication and division of two integers stored in two integers register. This extension is not included in the base sets for simplification purposes, for systems that would not have the use for division or multiplication operation.
These instructions allows operations between two signed integers, two unsigned integers and a signed and an unsigned integer, and are available for any base set listed previously.

"C" Standard Extension for Compressed Instructions

This extension set introduce equivalent reduced instructions, some instructions and pseudo instructions being now coded on 16 bits. This has a huge impact on the code size. According to the official documentation [1], 50 to 60 % of a program using RISC-V assembly code can be replaced by equivalent reduced instructions from the C extension, which approximately result in a total program size significant reduction of 25 to 30 %. With this extension, the PC has to be handled differently, some instructions being 2 bytes long.
To illustrate the benefit of using this extension, a really simple RISC-V assembly program is presented in table 2.1. After the compilation, the two right columns present the code in its hexadecimal form. The C extension reduced the code size by 33%.

assembly code	hex code,no C extension	hex code,C extension
lui x7,10	0000a3b7	846363a9
	00038663	13fd0003
loop:	fff38393	0385bfed
beq x7, zero, ext	ff9ff06f	90020001
addi x7,x7,-1	00138393	
j loop	00100073	
ext:		
addi x7,x7, 1		
.balign 4		
ebreak		

TABLE 2.1
C extension impact on the code length

"F" Standard Extension for Single-Precision Floating-Point

The F extension set adds instructions capable of dealing with single precision floating point data. With it comes 32 32-bits floating point registers and two status and control register. The status register enable the definition of the "rounding mode", coded on 3 bits. When executing an

instruction asking for the rounding mode, two possibilities exists. Either the instruction is able to define a intrinsic "rounding mode", or either it uses the one define by the status register. All the rounding modes are defined in the official documentation [1].
Load/store, arithmetic (add/sub, mul/div, sqrt...), conversion (between floating point and (un)signed integers), comparison (equal, less than, less or equal than) et classification of floating point number (negative, positive, 0, NaN...) instructions are added to handle this new type of data.

CHAPTER 3

CPROC

This section is dedicated to the actual first version of the processor, named CPROC developed during the internship. As a reminder, CPROC is intended to do digital data post processing and internal monitoring in mixed signal ASICS. It target a small footprint, low power consumption, and a maximum of four metal layers. In this context, four different part will be addressed, namely "CPU core and ISA choice", "Compilation flow", "CPROC" and "digital flow". Both hardware and firmware/software choices are discussed within this section, as it is impossible to not consider and understand the firmware while designing the CPU hardware and vice-versa.

3.1 CPU CORE AND ISA CHOICE

CPROC is not designed from scratch. The advantage of using the RISC-V standard is, as stated before, having access to already existing cores and their documentation. Numerous criteria were to be considered when choosing the base core of the project. The first and main criteria for the considered cores is for them to be licensed under a free license. An other one is to find a core which code is written in a language supported by the design tools (Cadence Genus). The available documentation is also a very important parameter to consider. For the design part to be the more effective possible and for understanding purposes, the core has to be up to date, well documented and active: an active community still working on the project is clearly a plus in this scope.
The extensions available on the core, meaning the code actually supporting the instructions added by these extensions, is also to be considered, but is less troublesome as support for theses extensions can always be added to the core, mimicking the already existing instructions in the code.

3.1.1 PICORV32

PicoRV32 [16] is the core that has been chosen for the base of CPROC. It is under the ISC licence, meaning it is free and open hardware. It is a RV32 configurable core, providing support for the base sets RV32I and RV32E, and both the M and C extension., which are the only extensions that are considered for the CPROC project. All the RTL code is in Verilog, the community is still very active, and the documentation is up to date and very dense. Besides, it has served as a base core for several project, namely the *Raven SoC*, an ASIC by efabless using the 180 nm node [9], or the *Caravel SoC* by Google and Efabless [13].

The PicoRV32 CPU is originally mean to be used as an auxiliary processor in FPGA designs and ASICs. As stated previously, it includes a variety of parameters that allow configuration for the two different base set (ENABLE_REGS_16_31 in the code), inclusion of the "M" and "C" RISC-V extension, and other functionalities such as a custom handling of IRQs incorporating custom instructions for the latter. An other significant feature is the Pico Co-Processor interface (PCPI) that can be used to implement non-branching instructions in an external coprocessor. The code features an implementations of PCPI cores that implement the M Standard Extension instructions MUL[H[SU|U]] and DIV[U]/REM[U] [1].

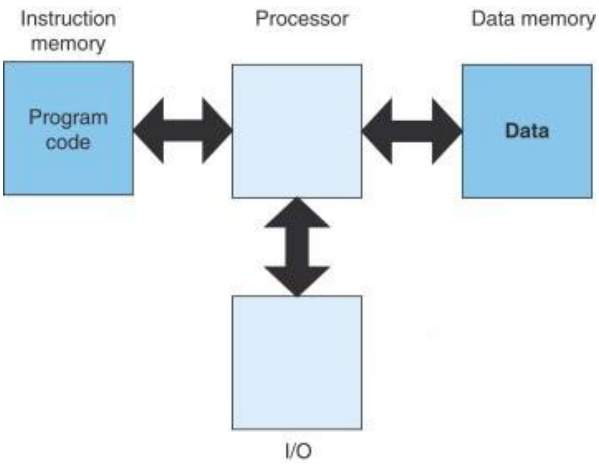

FIGURE 3.1
Simple CPU diagram

A CPU is always associated to memories, that can be internal or external, and I/Os 3.1. To communicate with those elements, it needs communication protocols, which PicoRV32 offers. The core exists in three variations: picorv32, picorv32_axi and picorv32_wb. The first provides a simple native memory interface, that is easy to use in simple environments. picorv32_axi provides an AXI-4 Lite Master interface that can easily be integrated with existing systems that are already using the AXI standard. picorv32_wb provides a Wishbone master interface. At first, CPROC was meant to implement the picorv32_wb interface, which for the same testbench was two times slower than the picorv32_axi interface. Among the first lines of work of this internship was to make the wishbone version as fast as the axi one, which was successful. Nevertheless, for module compatibility issues, CPROC will incorporate the native memory interface, using an external spi master for external integration.

As stated by the author on the picoRV32 git [16], its design targets a small footprint and high maximum frequency. Consequently, it lacks features that would increase its complexity, such as a multi-stage pipeline. However, while not pipelined in the sense that different stages of multiple instructions execute simultaneously, the processor does feature a state machine that divides the

execution of each instruction into stages. In all, there are eight possible states: fetch, ld_rs1, ld_rs2, exec, ldmem, stmem, shift, and trap. By default, the ENABLE_REGS_DUALPORT parameter of the PicoRV32 will be enabled, allowing the processor to read two values from the register file simultaneously and eliminating any need for the ld_rs2 state. The trap state, meanwhile, exists to let the processor handle unrecognized instructions. All six other states may be used during normal operation.

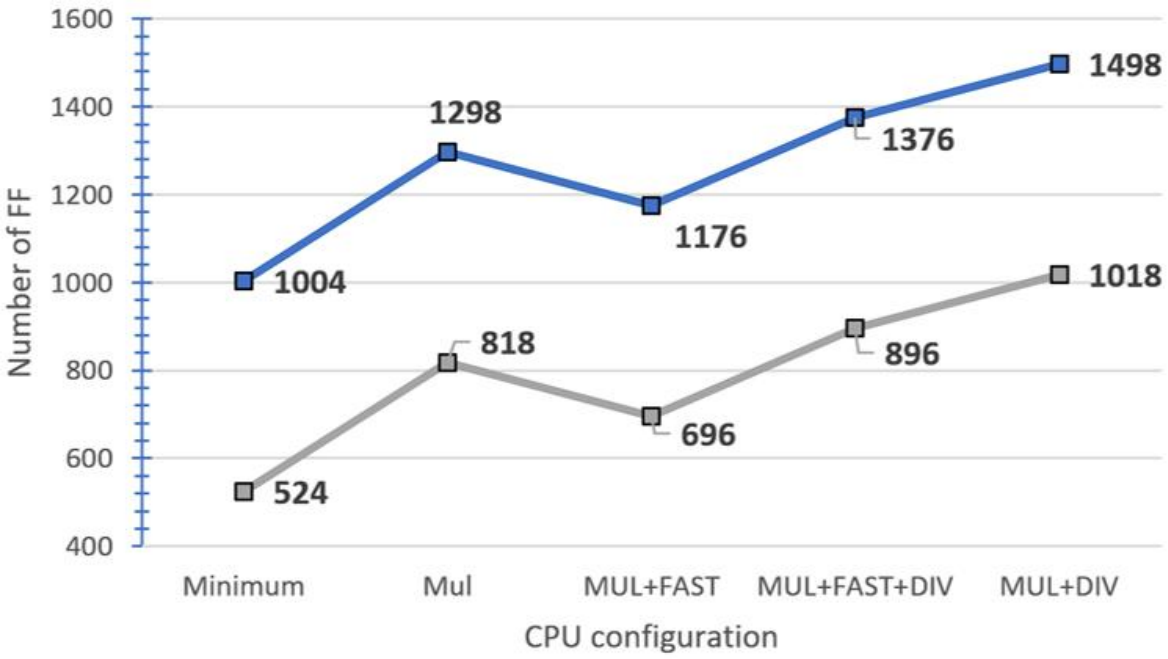

FIGURE 3.2
Total number of Flip Flops with and without the register file FF, RV32E

3.1.2 CONFIGURATION CHOICE

To determine how to configure the core, meaning choosing the value of the 25 parameters in the Verilog code, we first had to get a better grasp of how changing those values impacts the hardware, and especially the core's footprint. The latter also have an impact on the memory mapping, thus the software and compilation too. There are a lot of different parameters to be set [16] but only three are considered for this quick study. Those parameters are:

- ENABLE_MUL that internally enables PCPI and instantiates the picorv32_pcpi_mul core that implements the MUL[H[SU|U]] instructions.
- ENABLE_FAST_MUL that internally enables PCPI and instantiates the picorv32_pcpi_fast_mul core that implements the MUL[H[SU|U]] instructions using a single cycle multiplier.
- ENABLE_DIV that internally enables PCPI and instantiates the picorv32_pcpi_div core that implements the DIV[U]/REM[U] instructions.

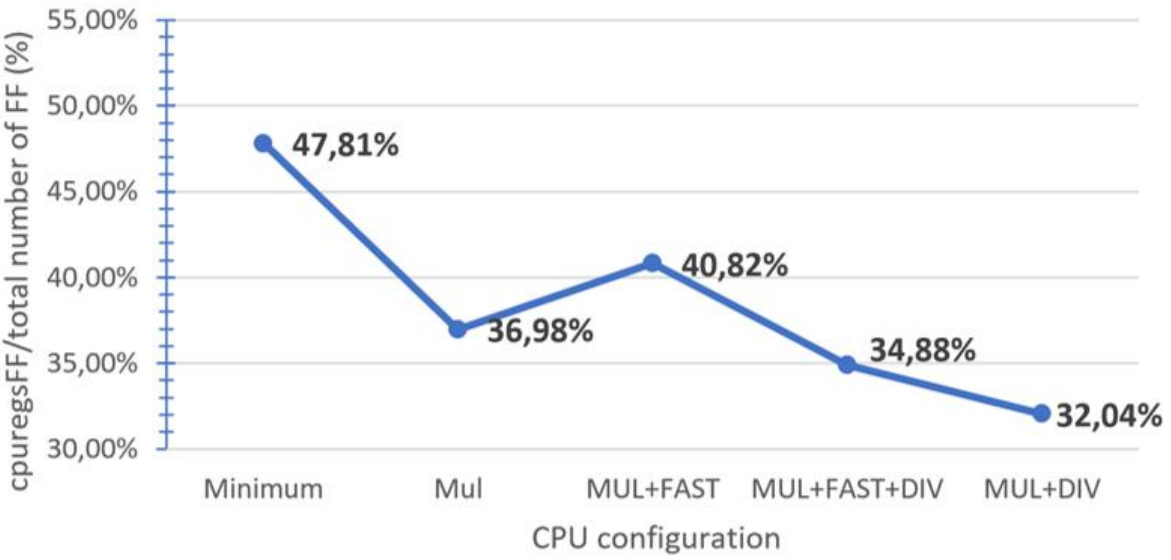

FIGURE 3.3
Register file FF share with respect to the core, RV32E

This lead up to five different configurations: the *Minimum* configuration with all the previously cited parameters set to 0 (meaning not instantiated), the *Mul* configuration with only the multiplier activated, the *MUL+FAST* configuration replacing the normal multiplier with the single cycle multiplier, the *MUL+FAST+DIV* featuring the fast multiplier and the division blocks, and finally the *MUL+DIV* configuration instantiating the normal multiplier and the the division block. The digital flow used to synthesize and place and route the different cores configuration uses a 100MHz clock (10ns period) and is set to allow only for four metal layers while generating the layout as in a mixed-signal ASIC, the superior metal layers are dedicated to power routing to have ultra lo< noise circuits. The slack and footprint values are from Cadence Innovus reports, post PnR and extraction (with the worst case corner).

In the CPROC scope, some parameters are already set with respect to the constraints: we are using the C extension (COMPRESSED_ISA = 1) and E extension (ENABLE_REGS_16_31 = 1) to ensure respectively the minimum code size and register file footprint. Since the register x0, the "zero register", is constant zero, the E extension features 15x32 = 480 registers only for the register file. The IRQs are also enabled along with the custom IRQ handling offered by picoRV32 since we want the final SoC to be able to deal with external data. It is important to note that the custom IRQ handling adds four 32 bits registers to the register file, and it has to be taken into account when using a custom implementation of the register file. In this quick study, this parameter is not enabled, this is why the register file only features 480 registers, and not 608 registers.

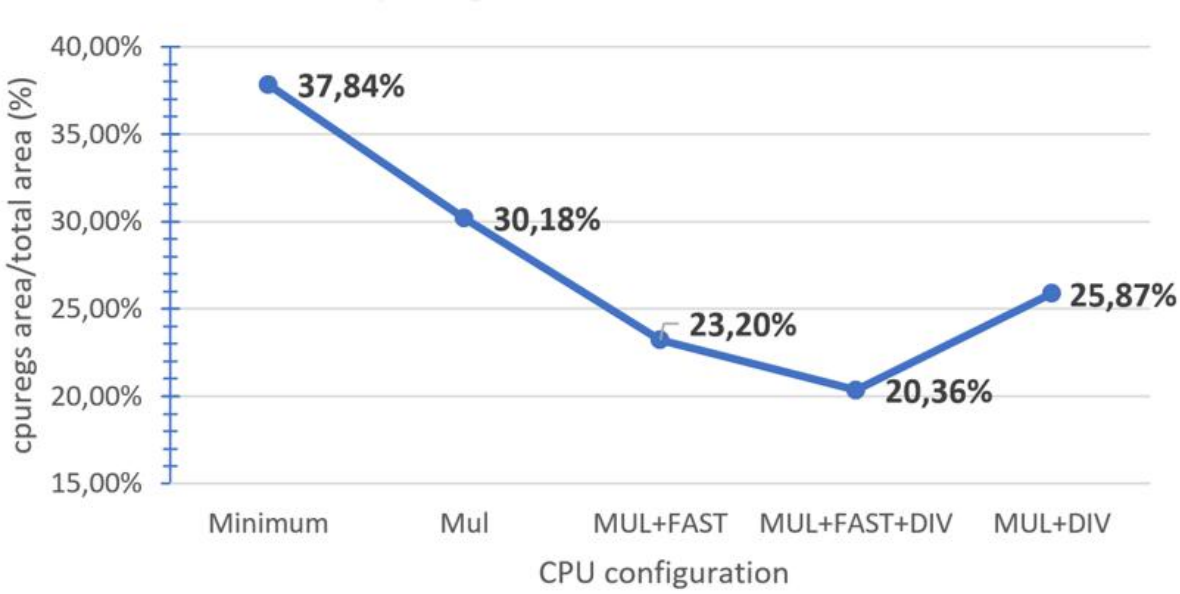

FIGURE 3.4
Register file area share with respect to the core, RV32E

When looking at the results on figure 3.2, 3.3 and 3.4, we can clearly see that the register file register takes up to half of the total number of registers of the core in the minimum configuration and 40% of the total area. The area here designs the output of the *report_area* command in Cadence Genus, not to be confused with the footprint which is the actual space, usually a square, in which the core can physically fit in, depending on the number of metal layer used and so on. A first conclusion that can be made right away, is that making the core resistant to radiations is way more complex than expected: the usual way to make radiation hardened chips in the laboratory is to triplicate every register within the *control path*. Every now triplicated registers are spaced in the layout, so a SEE is less likely to affect all of these registers. A voter placed a the output of every triplicated register make sure that the majority value, most surely the right value, is propagated through the system, and auto-corrects the register being faulty 3.5.

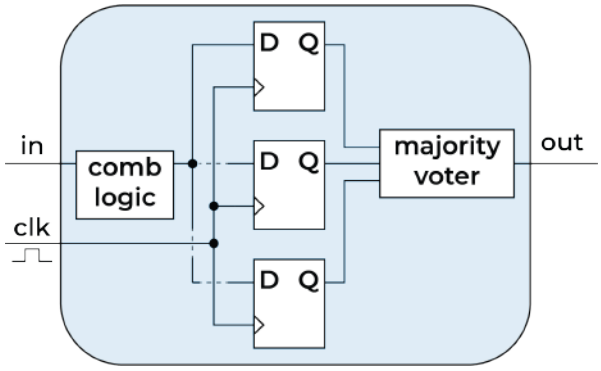

FIGURE 3.5
Triplication example

In our case, if the register file is not triplicated, there are still 524 registers 3.2 to potentially triplicate in the minimum configuration. Furthermore, triplicating means deeply understanding the way the code is written, which isn't something we could afford during a five and half months internship.

Concerning the configurations themselves, the minimum configuration is non surprisingly the most compact in terms of footprint 3.6, the most efficient in terms of slack 3.7 and counts the lowest number of registers. It is by far the best option for a very compact use and fairly simple calculus not involving loads of multiplication nor divisions.
The *Mul* configuration adds nearly three hundreds registers, however the Slack is comparable to the minimum configuration and the footprint is the same. Regarding that last point, the minimum configuration could have fitted in a smaller square, but the Mul one is nevertheless still compact. Every configuration instantiating the Fast Multiplication presents a high footprint and a problematic slack value. Usually we consider that an acceptable slack value is above 10% of the clock period, which in our case would be more than 1000 ps. This is caused by the constraint of having single cycle operation for the multiplication which inevitably adds to the propagation delay.
The last configuration is the *MUL+DIV* configuration: the slack is acceptable because slightly over the limit, the footprint is manageable, however the number of total registers is by far the highest, 1498 registers, making the future radiation hardening much harder.

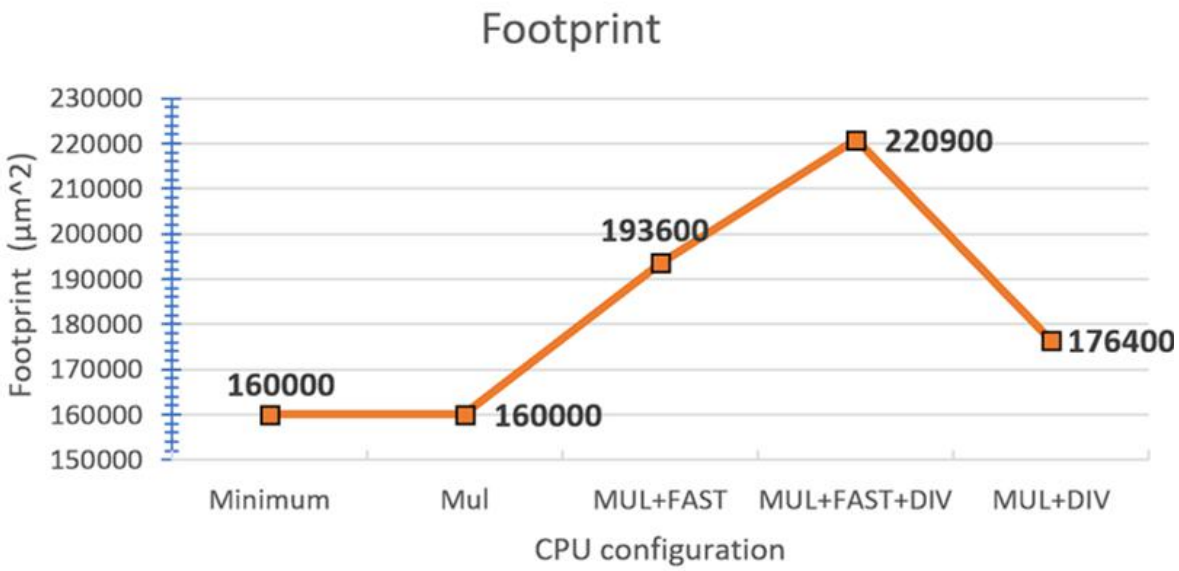

FIGURE 3.6
Footprint in μm, RV32E

3.1.3 REGISTER FILE USING LATCHES

As stated before, the register file represent a non negligible part of the total area share. The RV32E configuration already cuts half of the registers, leaving 480 of them. However, there's no way to further reduce the total number of register without undermining the processor functionalities. An other solution could be to find something that takes less space than a register, but still provides the same functionality: a latch, which is half a register. Cadence Genus offers the possibility to create memories using its own IPs, and it is possible to instantiate the equivalent of the register

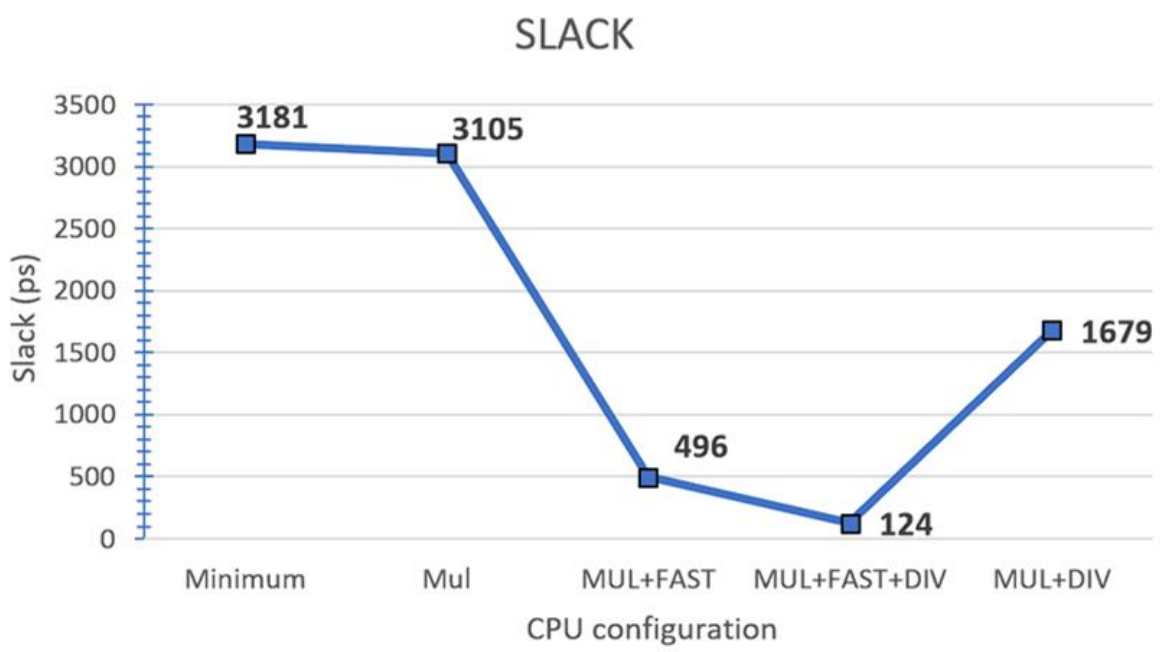

FIGURE 3.7
Slack in ps, RV32E

file, but using latches.
The way this quick study has been done is the following one: only the register file using registers and latches have been synthesized using Cadence Genus, not the entire core. The registers files have both a depth of 8 and a length of 16, meaning eight sixteen bits registers. The IPs used are all Low Voltage Threshold (LVT) ones.
The results are presented in figure 3.8. Even though a small number of instances are present in each memory, there is a 27% gain in area, but two times less slack, which can be a problem when having stricter constraints. having two times less slack using latches is normal, since they update on levels and not on edges like registers. We could have expected half the area, however only the latches takes 2 times less space, and not all the logic involved in the row and column decoding. Greatly increasing the size of the memory would lead to a better gain in area.

Even though there might be a gain using latches in the place of registers, the register file in our SoC will use registers, since there is less risk that everything breaks down when running a program, especially for such an vital component. This study has been done at the very beginning of the internship to explore different possibilities and mainly for the sake of it.

In conclusion, the final configuration chosen for our project is the *minimum* configuration, corresponding to the RV32EC ISA, with 19x32 registers in the register file, including registers for the custom IRQ handling. Adding the multiplier is still on the table, since the trade-off between the area and the code size could nevertheless be interesting , nonetheless we are not

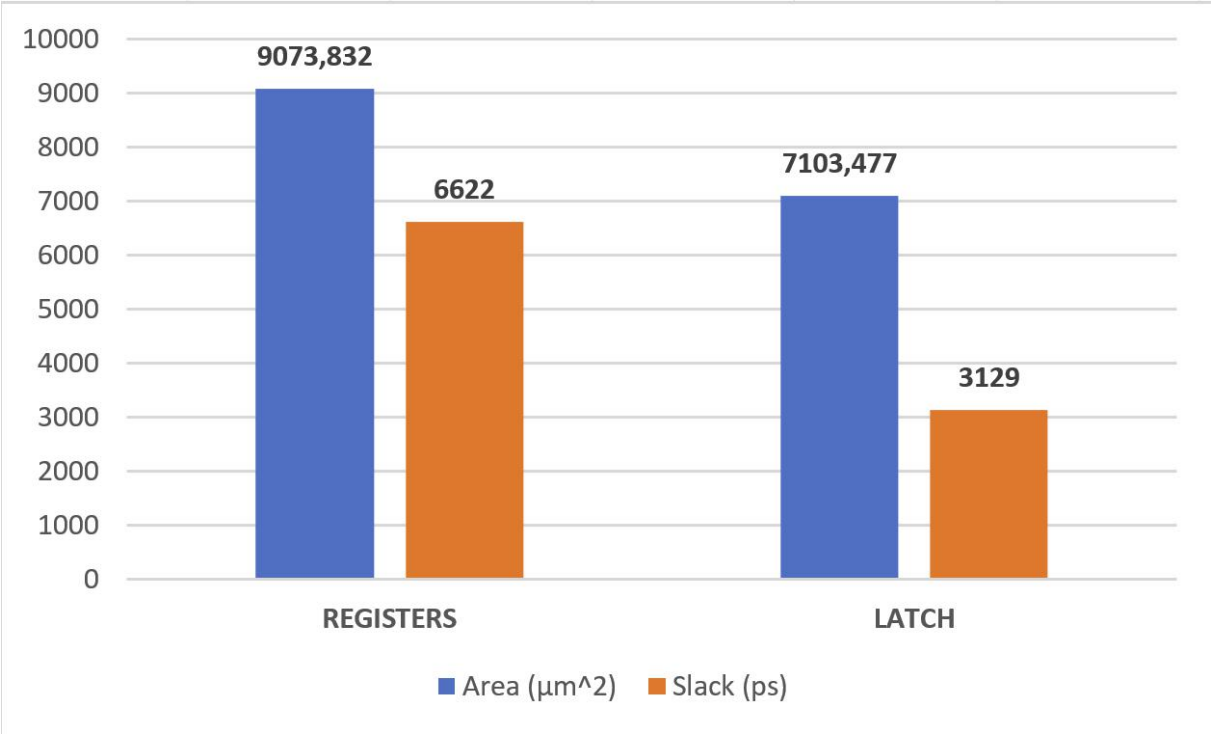

FIGURE 3.8
8x16 register file using registers and latches

using it in this study.

3.2 COMPILATION FLOW

In this part we are going to focus on the compilation flow, a key part of processor designing. What the processor is designed to be able to do relies nearly entirely on what firmware/software it is going to run. Having a clear idea of what we want to do with the processor is essential. For instance, knowing approximately the size of the code to be loaded into the processor memory will dictate what internal memory size should be used.

To compile any piece of code into a language that the processor can understand, we need the GNU GCC cross-compiler for RISC-V C and C++. This compiler, based on gcc can be downloaded on the RISC-V GitHub page [12] can translates any coded in C/C++ in RISC-V assembly code, and further into machine code. It works the same way as the standalone gcc software, with the same command line options and the same functionalities. The version of the RISC-V GNU toolchain is also an important parameter: PicoRV32 is intended to be build with along with the RV32I, RV32IC, RV32IM or RV32IMC ISA, even though the compatibility with the RV32E ISA is offered in the code. When cloning the git on a local machine, the RV32E RISC-V toolchain is therefore not installed, and it needs to be so manually, or by modifying the Makefile provided on the git to do it automatically. Trying to compile a code intended for the RV32E ISA with the

```
4
5  firwmare.elf : sections.lds main.o start.o
6          /opt/riscv32e/bin/riscv32-unknown-elf-gcc -01, -mabi=ilp32e -march=rv32e [...] -o firmware.elf main.o start.o -lgcc
7
8
```

FIGURE 3.9
A Makefile compilation command line

RV32I toolchain won't work, and vice-versa. Figure 3.9 shows a compilation line command were the compiler version (RV32E) and the exclusive RISC-V command line options -march and -mabi correspond. The -march=ISA selects the architecture to target. This controls which instructions and registers are available for the compiler to use. The -mabi=ABI selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and the layout of data in memory.

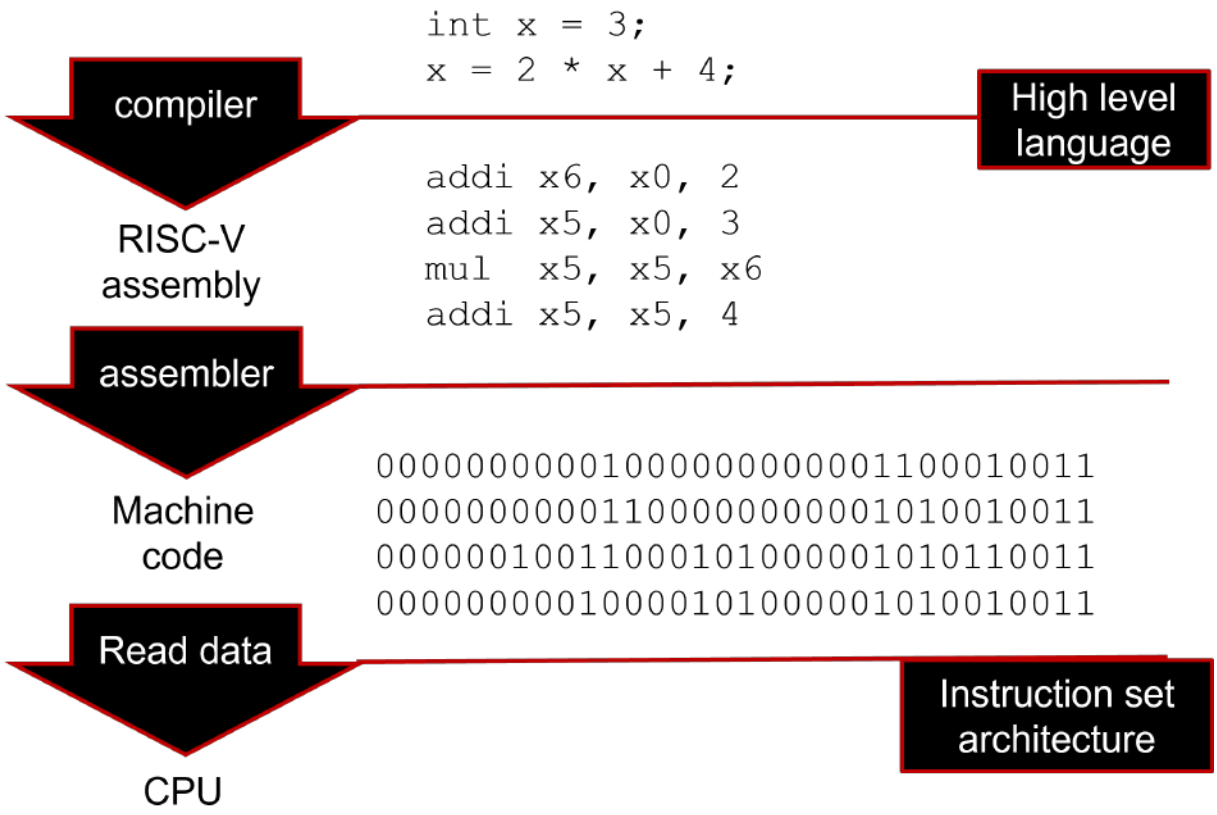

FIGURE 3.10
Compilation flow of a small piece of code

A short example of of the RISC-V compilation flow is depicted in figure 3.10: the small piece of code is first ran through the compiler which translates it in RISC-V assembly code, or instructions. The compiler used here is the RV32IM version, as it can be observed by the presence of a "mul" instruction. This intermediary result needs to be further processed, this time through the assembler that will transform these instructions into machine code, 0s and 1s, in a

file that will be loaded into the SoC memory by a loader.

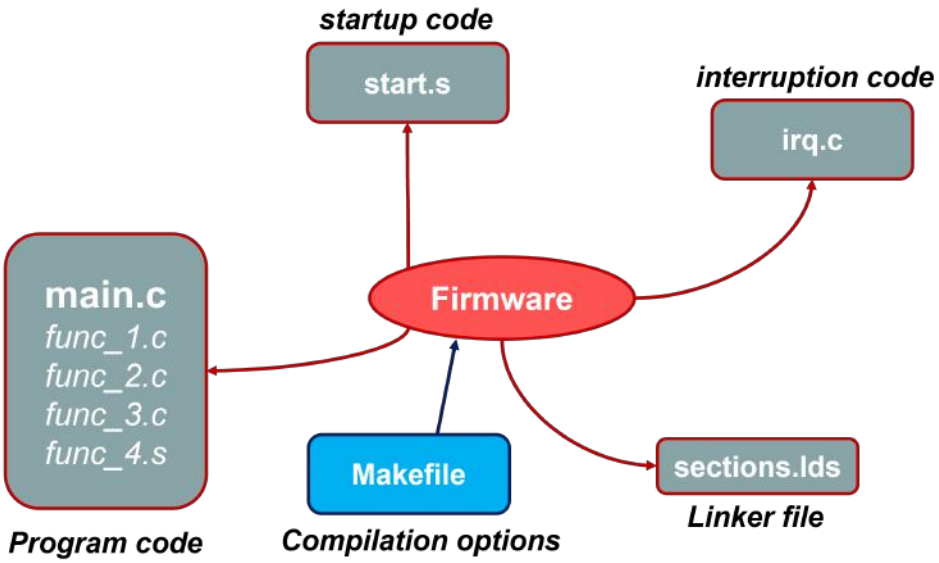

FIGURE 3.11
Diagram of what is included in the SoC firmware

However, one can guess that an SoC firmware is not that simple and involves multiple files to create the rather complex firmware environment our SoC is going to execute. The example of such environment is depicted in figure 3.11.
The first group of file includes the files containing the "program code", meaning the main function that has to be ran on the processor. There is nothing very special about this group of file, as long as no unrecognized operation are written in C/C++ such as log or exponential functions, which are not supported by the RISC-V compiler, there are no special requirements here. Our SoC supports interruption, which can be external interruptions toggled by the assessment of one of the four external IRQ pins, or internal interruptions. These interruptions, when assessed, make the current processing stops and jumps to the address of the piece of code describing what needs to be done with respect to this interruption. This piece of code is the irq.c file, and it usually do some very simple computing, or calls function of the main program. Its size may be fairly consequent.
The next very important file is the startup code, start.s in figure 3.11. This file contains the first lines of code executed by the processor upon booting. The .s extension means it is written in assembly code, so only very simple code is comprise within this file. When booting the processor, it will typically initialize all the register file register to zero, except the stack pointer (x2) register which value is initialized by the hardware. Then it can run some basic code such as copying data from an external memory to the internal SoC memory, and finally jumps on the main program. This file also contains what is called the IRQ handler: when an IRQ is assessed, the PC jumps to the address of the handler, the latter saves the current value of all register in the register file in the memory, and then jumps to the IRQ program. When the action induced by this IRQ is finished,

the handler loads the previously memory-saved register file values and carry-on its process.
The last, and maybe the most important file, is the linked script file or lds file. Linking is the last stage in compiling a program. It takes a number of compiled object files and merges them into a single program, filling in addresses so that everything is in the right place. The linker takes all of the previously compiled object files and merges them together along with external dependencies like the C Standard Library into an output file, an executable (.bin .elf etc). To figure out which bits go where, the linker relies on a linker script - a blueprint for your program, it controls the memory layout of the output file. Besides, when necessary the linker script can also direct the linker to perform many other operations, using the linker commands. The linker place every element of the input object files into sections. Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. For instance, there might be a section containing all the program code from different object files, a section for initial data, and so on . A section may be marked as loadable , meaning that the contents should be loaded into memory when the output file is run. A section with no contents may be allocatable , which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section, which is neither loadable nor allocatable, typically contains some sort of debugging information [6].
Many linker scripts are fairly simple. The simplest possible linker script has just one command: *SECTIONS*. *SECTIONS* command is used to describe the memory layout of the output file. Let's assume our program consists only of code, initialized data, and uninitialized data. These will be in the *.text*, *.data*, and *.bss* sections, respectively. Let's assume further that these are the only sections, which appear in the input files. In this example, the code is loaded at address 0x1000, and the data starts at address 0x40000. The following linker script will do this function 3.12:

```
SECTIONS
{
. = 0x1000;
.text : { *(.text) }
. = 0x40000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

FIGURE 3.12
Short linker script example

The linker script also declares the the name of the different memory accessible, the size, by setting an origin and a length value, and whether they should only be read from (xr command in the linker script) or also written in (xrw command).

Figure 3.13 sums up all the compilation flow. The input files, containing RISC-V assembly code and C/C++ code are passed through the compiler and assembler to end up translated in

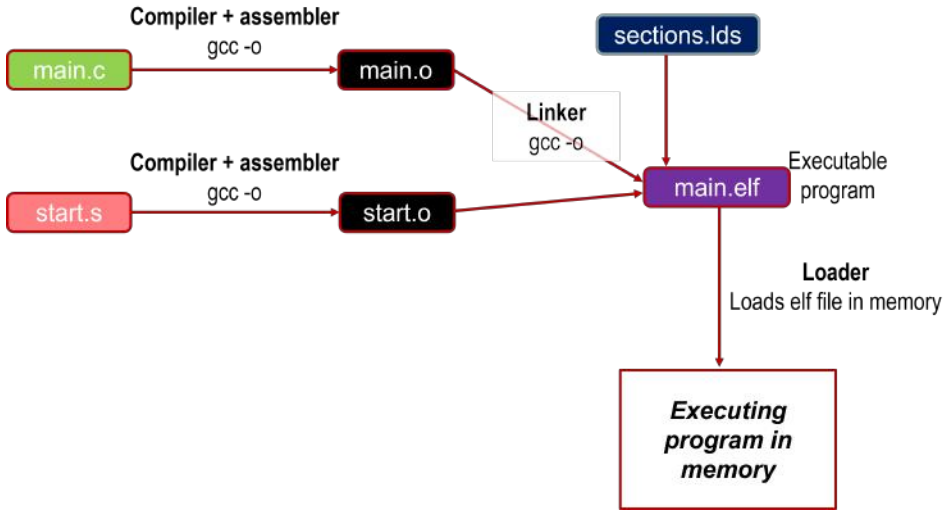

FIGURE 3.13
Full compilation flow, from multiple input file to one executable output file

machine code (command line option gcc -o). The latter, now called object files, are linked using the same command line option (gcc -o) along with the linker script, and the output file is now a executable file, with the extension .elf in our case. The .elf or .o files can be translated back to assembly code with the command line option -objdump in place of -gcc. When used for the executable file, the memory mapping of the .text section can be observed as every instruction is displayed with its address. This is very useful for debugging since it is easier to track what instruction the pc counter is pointing on.
Then the loader loads the data within the executable file into the internal memory. In the case of our SoC, the executable file as first to be translated to hexadecimal (a .hex file), and then fed into the memory.

3.3 CPROC

CPROC is the final name of our SoC, and its digital block diagram is presented in figure 3.14. The analog modules its missing such as a Power On Reset (POR) or pads for the package modules will be integrated later by OMEGA analog designers. It is partially based on the PicoSoC SoC, a simple SoC design using PicoRV32, developed for the iCE40-HX8K FPGA Breakout Board. CPROC features:

- A single core CPU: PicoRV32, RV32EC ISA, 40 MHz working frequency
- An internal 8ko SRAM memory (2048 words)
- An external flash support and a dedicated spi master. The has already been chosen, it is the new W25Q64NE 1.2 V serial NOR flash from Winbond [15].

- An other SPI master to deal with external SPI compatible modules.
- An UART connection
- 24 General purposes inputs and 16 General purpose outputs
- 4 external and programmable IRQ pins
- A two start address choice

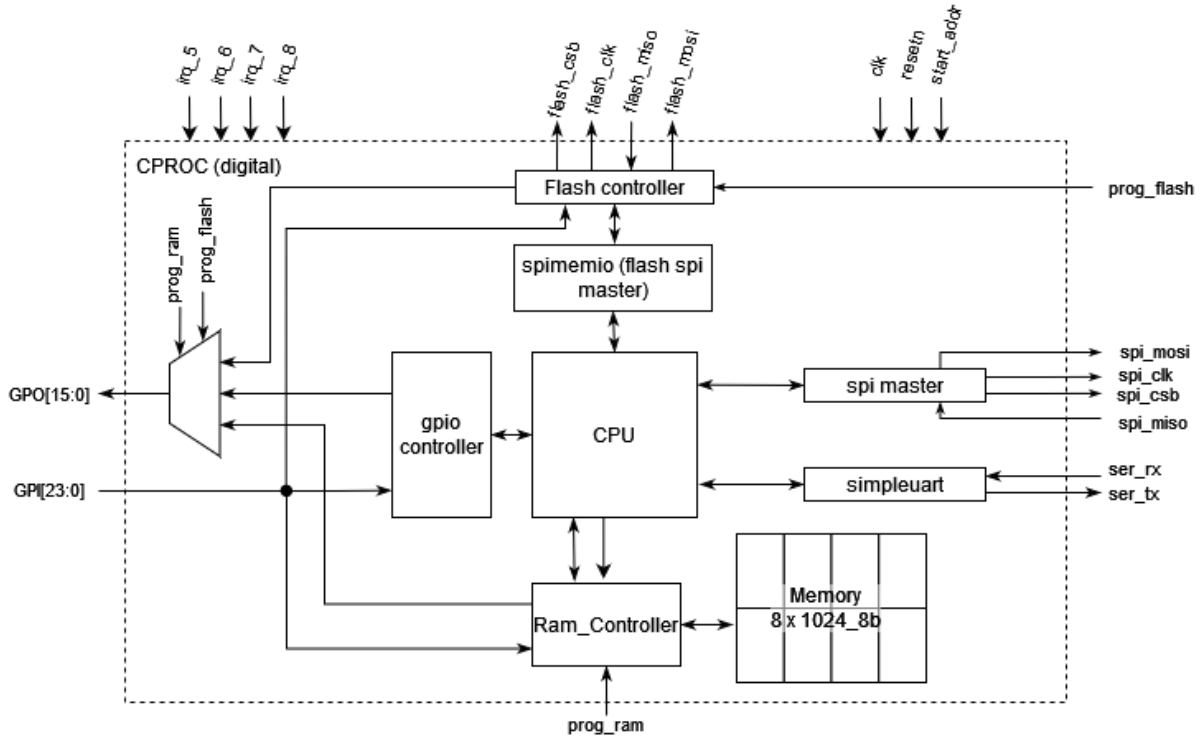

FIGURE 3.14
CPROC digital block diagram

As stated before, CPROC is partially based on PicoSoC as it reuse two of its blocks: a standard UART master protocol (named simpleuart) and an SPI master (named spimemio). The simpleuart module is as its names suggests it, the block managing the UART connection between the exterior and CPROC through the ser_tx and ser_rx pins. Except the readjustment of the baud rate to fit CPROC working frequency (100 MHz in PicoSoC, 40 MHz in CPROC), no modifications were made to this block. The spimemio module, the dedicated flash SPI master, is a rather complex module created by Claire Wolf, the original author of PicoRV32 [16] and PicoSoC [17]. No modifications were made to this block, except permanently deactivating the Quad and Dual SPI functionalities by setting two status register to a permanent zero value. The philosophy behind the first version of CPROC is to make it as simple as possible, using as much as possible what is already in house, i.e. mainly analog blocks and pads for package bonding. As Dual and Quad SPI would mean designing analog pads able to manage dual way I/Os (used

as input and output) at a 40 MHz speed, which hasn't been done in the past at OMEGA, we restricted every pins of the chip to be only input or output since pads for the latter are available for a 40 MHz frequency. Single SPI protocol has only ne way I/Os, 3 outputs and 1 input for the SPI master (the inverse for the SPI slave).

The choice of memory has been a center discussion around this SoC. In the PicoSoC project, the program code i.e. the .text sections, is stored in the external flash, and only some initial data is copied in the internal SRAM. Even with the Quad SPI enabled, it only allows simultaneous 4 bit reading, being at least eight times slower than the internal SRAM, the latter connection with the CPU allows for entire word reading, 32 bits simultaneously. In our case only the Single SPI is available, meaning any program execution from the flash would be at least 32 times slower than when read from the SRAM. This means that the SRAM has to be big enough to fit medium sized program codes. The memory was chosen among the radiation hardened memories offered by CERN. The store byte and store half instructions relies on a 4 bits internal write enable signal, that selects which byte of the 32 bits word has to be written on. This specificity require the use of four 8 bits wide memory instead of a single 32 bits wide one. Among the memory offered in the CERN list, only the 1k words 8 bits wide qdp130_1024w8b_1mrs memory fitted our needs. Eight of them were used in order to make a 2ko internal SRAM. Furthermore, the memory adds a constraint as the chip is forced to run at the same frequency as the memory IP, which is 40 MHz.
In addition, a new feature has been added to PicoRV32, and on a wider scale CPROC: it is the start address choice. In PicoRV32, the start address is set in silicon, it is an hardware parameter, and when chosen during the design phase, it doesn't change. What was wanted with CPROC, was the choice to either start from the SRAM address which is 0x0, or from the external flash address which is 0x100000. It is controlled by an external pin called *start_addr*: when set to logic 0 (before the reset signal going high) the CPU will start in the SRAM, when set to logic 1 the CPU starts in the external flash. This offer the possibility, unlike PicoRV32, to use the SoC without the need of an external flash to hold the program data and initial data. But for this to happen, the internal CPROC SRAM has first to be programmed, before the CPU is booted. For this to happen, a module referred as Ram_Controller has been designed to offer a connection between the internal SRAM and the exterior through the GPO and GPI pins, completely bypassing the rest of the chip. This module has two configuration: programming mode (*prog_ram* = 1, *resetn* = 0) 3.15 which functionality has been described, and normal mode (*prog_ram* = 0, *resetn* = 1), simply linking the CPU to the memory. The internal SRAM is programmed byte per byte, every data, address, and control signals being passed through the GPI 3.1. The GPO pins can be utilized to monitor the process.

A similar module called Flash_Controller allows for the chip to serve as a loader for the

GPI pins	Mapping
GPI [10: 0]	Pin : *prog_RA* & *prog_WA* - The address for the Write and Read Address (WA & RA) pins of the memory, coded on 11 bits
GPI [18:11]	Pin : *prog_WD* - The data intended to be loaded into the SRAM. The data is loaded in the SRAM one byte after the other, by bringing low only one bit of the enable at a time
GPI [22:19]	Pin : *prog_Wb_Rb* - The enable signal for both reading and writing in the SRAM. Whether we are writing or reading is defined by the *prog_Rb_W* pin
GPI [23]	Pin : *prog_Rb_W* - Toggle whether we are writing or reading. 1 means writing, 0 means reading

TABLE 3.1
GPI pins mapping when using the Ram_Controller programming mode

external Flash. Using the flash programming mode (*prog_flash* = 1, *resetn* = 0), the flash is programmed through the GPI and GPO, using the classic SPI protocol 3.16.
The gpio controller module is a simple block managing the connection between the GPIs and GPOs pins and the CPU. Finally, the spi master module is as its name suggests a second SPI master intended for general external SPI connection. This block is borrowed from the caravel [13] project.

The memory mapping of every module is as it follows:

Address range	Description
0x0000_0000 - 0x000F_FFFF	Internal SRAM. The irq handler is located at address 0x0000_0010, in the SRAM.
0x0010_0000 - 0x01FF_FFFF	External Serial Flash
0x0200_0000 - 0x0200_0003	SPI Flash Controller Config Register
0x0200_0004 - 0x0200_0007	UART Clock Divider Register
0x0200_0008 - 0x0200_000B	UART Send/Recv Data Register
0x0200_0014 - 0x0200_0017	SPI Master Controller Config Register
0x0200_0018 - 0x0200_001B	SPI Master Controller (Data)
0x0300_0000 - 0x0300_0003	GPO
0x0300_0004 - 0x0300_0007	GPI

TABLE 3.2
Module address mapping

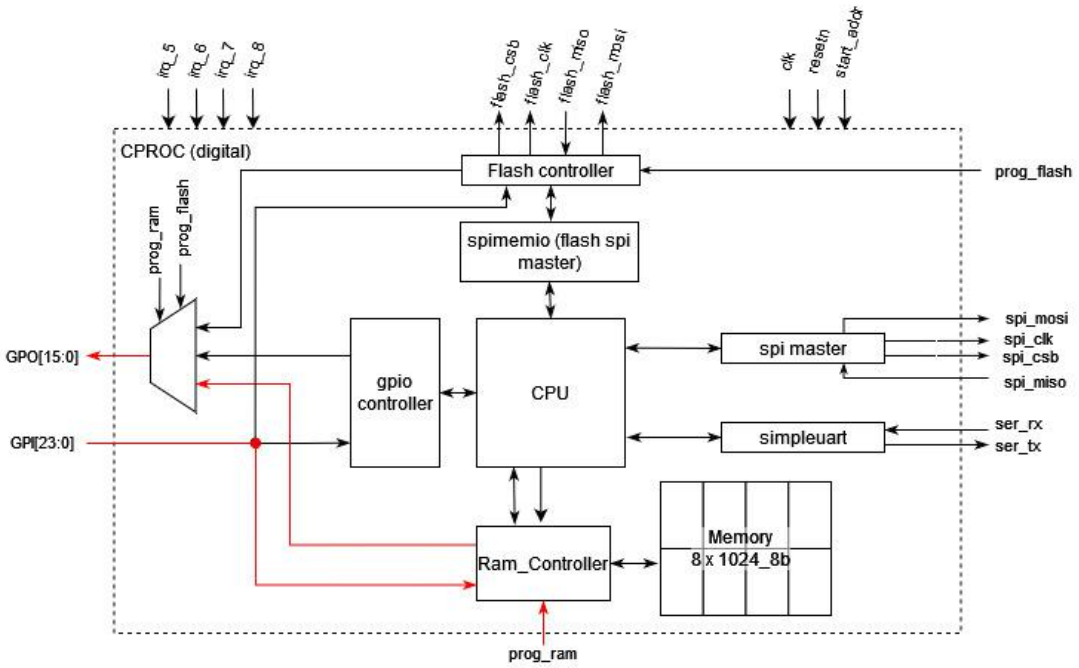

FIGURE 3.15
Ram_Controller programming mode

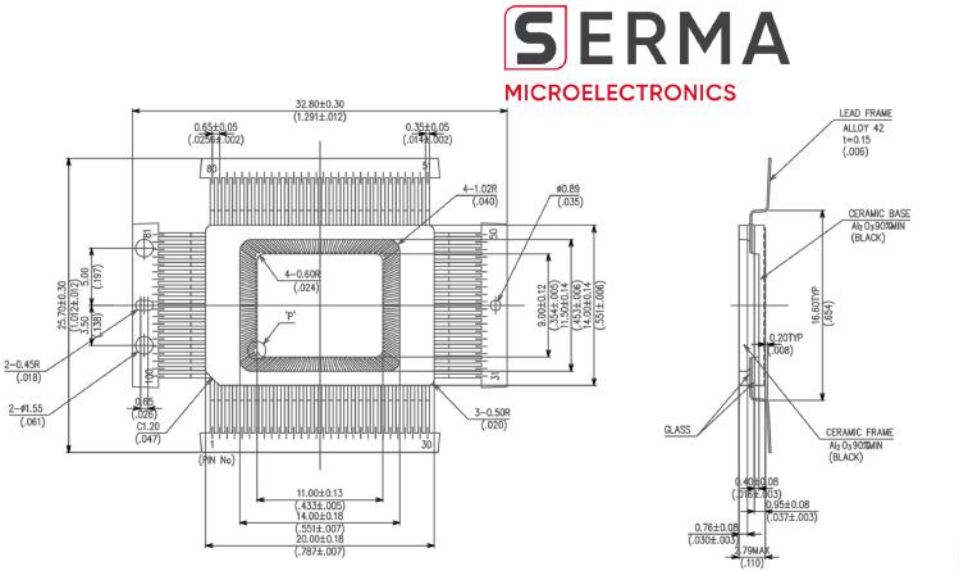

FIGURE 3.17
CPROC package

The last thing to choose is the package. For the moment, the digital part of CPROC has 60 pins, and analog blocks that will be added later will inevitably need some more. A 80 pin package would be too small, as the empiric rule for power biasing is one VSS pin (usually grounded) and one VDD pin every ten pins, so two power pins every ten pins. On a 80 pins package, this represent 16 power pins, which only allocate 4 pins for analog blocks in our case. For CPROC,

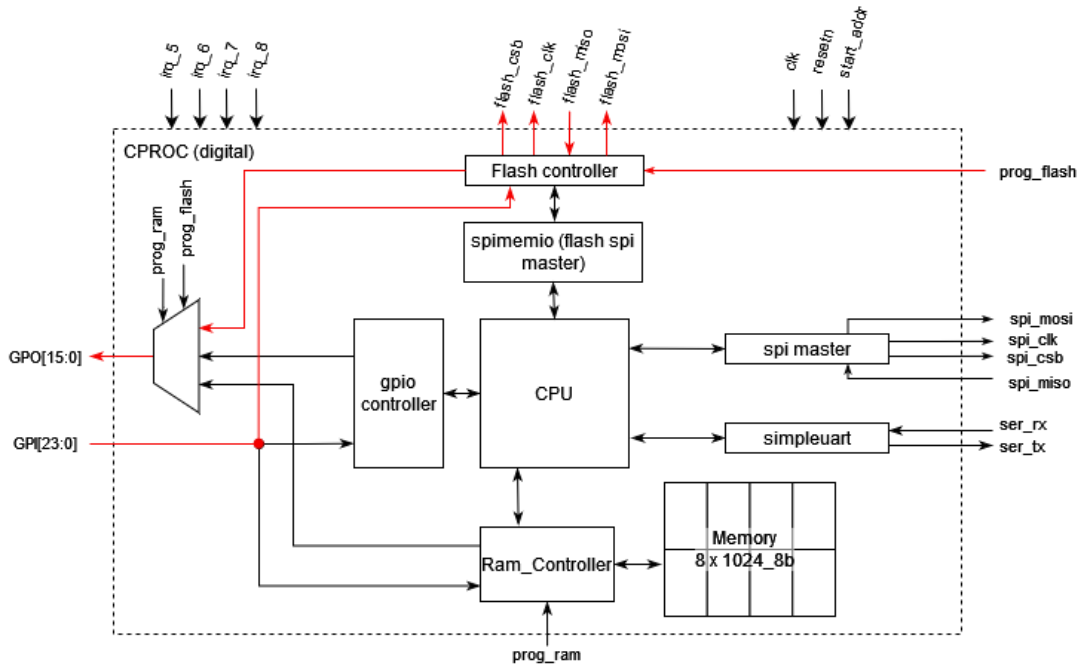

FIGURE 3.16
Flash_Controller programming mode

we opted for a hundred pin package the QFP10014X20 from SERMA Microelectronics 3.17. It has already been used at OMEGA, so every analog bonding pad has already been developed. It's internal dimensions are 11 x 9 mm.

3.4 DIGITAL FLOW

This sections is about the chip testing, synthesis and PnR. The testing phase is not finished, but Cadence Genus (synthesis) and Innovus (PnR) runs were already made to have a better grasp on the chip dimensions, and this to see if it would fit in any candidate package. This also allowed to prepare the scripts those tools rely on for when the chip will be fully tested and debugged.

3.4.1 TESTING CPROC

Functionally speaking, most of the chip has already been tested. First of all, the core itself has been tested according to the officials battery of tests offered by the RISC-V Foundation [10]. Unfortunately, nothing is offered to test RV32E based core, but adapting the one for RV32I based core is fairly simple, and this is what was done. The PicoRV32 project already used the official RV32I official battery of tests, for its core testbench, plus some other functionality testing as the IRQ testing. From there, the last thing to adapt this testbench for the CPROC RV32E core was to adapt the firmware, to suppress any reference to x16 up to x31 registers, as in the register file initialization to zero in the start.s file for instance.

The core being fully tested, all the surrounding blocks have then to be tested. The testbench for the Ram_Controller module tests the functionality of the latter by first testing the programming mode (prog_ram = 1, resetn = 0). The internal SRAM is first filled with zeros, then a very simple program is then written into it (see firmware/firmware.s), and the SoC is set out of programming mode (prog_ram = 0, resetn = 1). The SoC starts in the SRAM (address = 0x00000000) and executes the simple program 3.18 before trapping.
The testbench for the Flash_Controller block is quite similar to the RAM_Controller one. The testbench is adapted from the winbond testbench provided with the verilog behavioral model of the flash. It first tests the flash programming mode by doing simple operations (Read Manufacturer ID, WRITE, READ) while being in programming mode (prog_flash = 1, resetn = 0). The inputs GPI and outputs GPO are used to bypass the spimemio module, through the flash_controller 3.16. A very simple program is then written in the flash 3.18, and the SoC is set out of programation mode (prog_flash = 0, resetn = 1). The SoC then starts in the flash (address = 0x00100000) and executes the simple program before trapping.

```
lui x7, 5

loop:
beq x7,zero, ext
addi x7,x7, -1
j loop

ext:
addi x7, x7, 1
.balign 4
ebreak
```

This simple program loads $5 * 2^{12}$ in the x7 register, then subtract one to it and loops until it is equal to zero. When so, it exits the loop, adds one to the same register, then breaks.

FIGURE 3.18
Simple program

The IRQ used in CPROC are not the same that are tested in the core testbench: PicoRV32 offers the possibility to program 32 different IRQ through 32 pins, but only few of them are used in both case, but some are not in common. An CPROC IRQ testbench was made to tests the functionality of CPROC, when using the RV32E ISA. Each IRQ pin is asserted several times during the test, and the number of time they are asserted is displayed at the end of the simulation. In this test only pins 0, 1, 2, 5, 6, 7 and 8 are tested, the external IRQ pins being pins 5, 6, 7 and 8 When an IRQ is brought to logic 1, the CPU jumps ton the IRQ handler address (PROGADDR_IRQ = 0x0000_0010) which saves the internal registers state. This part of the program is coded in startup code start.s. It then jumps onto the handler C function which codes what the processors need to do. This part of the program is coded in irq.c .
The simpleuart block is exactly the same as in the picoSoC project, and it has already been tested in the project testbench, and thus, at 40 MHz, with the same baud rate as in CPROC.

Nevertheless, a simple testbench has to be written for this module in the CPROC project, just to make sure.
The GPIO testbench is a very simple testbench: integers are sent to CPROC through the GPI pins, an IRQ pin is brought to logic one to make the CPU process stops and acknowledged the integer value. The value 1 in added to the integer, and the final value is displayed on the GPOs pins.

What is left to be tested is the spi master module, and the chip in general by combining all the testbenchs into one generic CPROC testbench. Layers of UVM verification will then be done to complete the chip testing.

3.4.2 SYNTHESIS, PNR

Doing a full digital flow (synthesis, PnR) on a not yet fully tested chip isn't something that is generally done. However in our case and as stated previously, we are are working with in house options for a lot of components, packaging included. The chosen package, the QFP10014X20 from SERMA Microelectronics 3.17, has a finite space for the full chip to fill in, digital and analog parts included. Furthermore, the metal layer constraints are not helping with making the more compact chip possible. If after this full digital flow, the chip footprint would be too big, some elements would have to be rethought and resized or removed.
The constraints on the layout are having a chip footprint smaller than 11 x 9 mm, and possibly way smaller to fit the additional analog blocks. The maximum number of metal layer available for the digital part is layer 4, as analog designers working to add the analog blocks on CPROC use wider metal tracks, thus using layer 5 and above. We are working with Low Threshold Voltage (LVT) cells, filler and decaps cells included. CPROC will be made using the TSMC 130 node, technology that has recently been approved by CERN for radiation hardened chips. The constraints on the clocks are for most contained in a file called the constraint file or sdc file because of its extension name (i.e. .sdc). We are working with a 40 MHz clock (25 ns clock period), with a 100 ps uncertainty and a transition time, the time it takes to the clock signal to go from logic 0 to logic 1 and vice-versa, of 200 ps. Moreover, for software licence concerns, the total number of cells has to be lower than 50000.

For the synthesis, scripts that were already used on other OMEGA projects were used, they just needed to be adapted to the CPROC project, by adjusting some parameters or including the memory cells library for instance. This step is fairly quick from the script modification to its execution by Cadence Genus 20.1. This step is mainly to verify if everything works as intended, to see if design mistakes led to the presence of latches instead of registers for example. The intermediary result such as the slack, the power consumption or the net length are only predictive and not relevant enough. As the goal of this full digital flow is just to check what is the size of

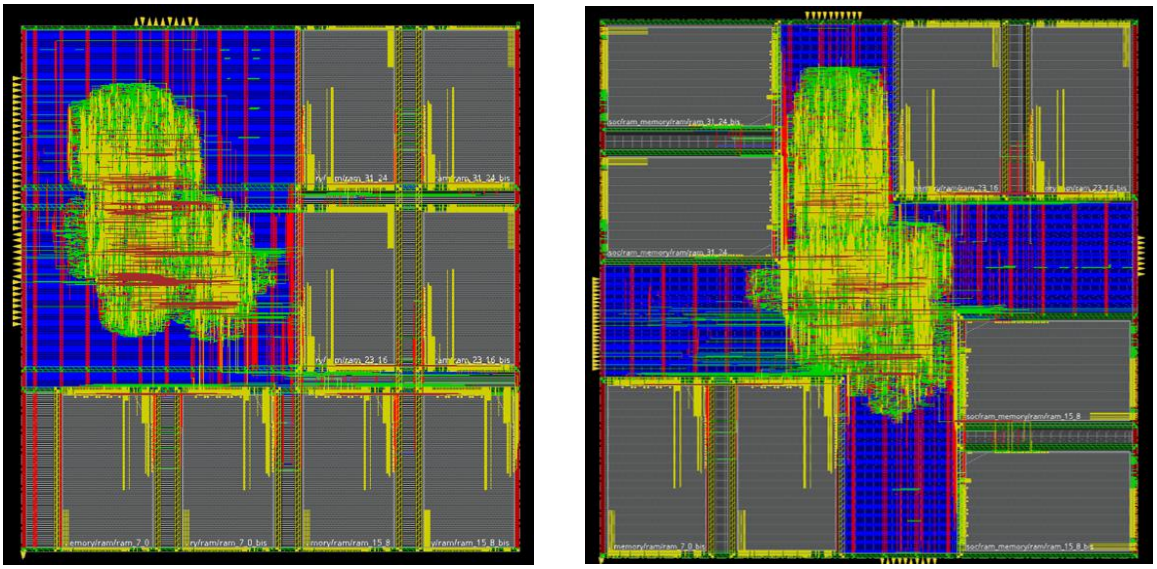

FIGURE 3.19
First floorplan on the left, second one on the right

the layout and prepare the scripts in prevision, no post synthesis simulation were made.
The next step is to proceed to the PnR to obtain a first layout. As we are using 8 memory IPs, the first step is to place them on the layout: it is called the floorplan step. This step also includes making the ring, the stripes, the power routing, placing the pins, and routing the eventual IPs. Every odd metal layer are horizontal, every even one are vertical. The outer ring is Metal 3 (M3) and Metal 2 (M2), the vertical stripes are in M2 and the power routing horizontal stripes are in M1. The

first floorplan design on the left in figure 3.19 was quite compact (1850 μm x 1850μm) but pins weren't disposed evenly around the layout, it used M5 (in dark red) and could be more optimized. The second attempt on the right of figure 3.19 was intended to make it more compact, which it did (1700 μm x 1700 μm), the pins were disposed more evenly than on the first floorplan, however M5 was used, and there was a major problem with the IPs placement. The IPs placed horizontally have their odd metal layers vertically and their even one vertically. This can cause problems when trying to route over them, since it could be the origin of short issues for instance. The third and last in date floorplan presented in figure 3.20 is the more compact yet (1600 μm x 1550μm). It uses only layer up to M4, has no IPs placement issues, has on even number of pins on all four sides, and zero DRC nor LVS violations at the end of the process. This is the layout given to the analog designers to work with, as they only need the dimension and the pin mapping. When it comes to the first technical results, post extraction we have:

- A total number of 44133 cells, filler and decap cells included.
- A slack of 5482 ps, which is twice as much as than the minimum needed (10% of 25 ns = 2500 ps).

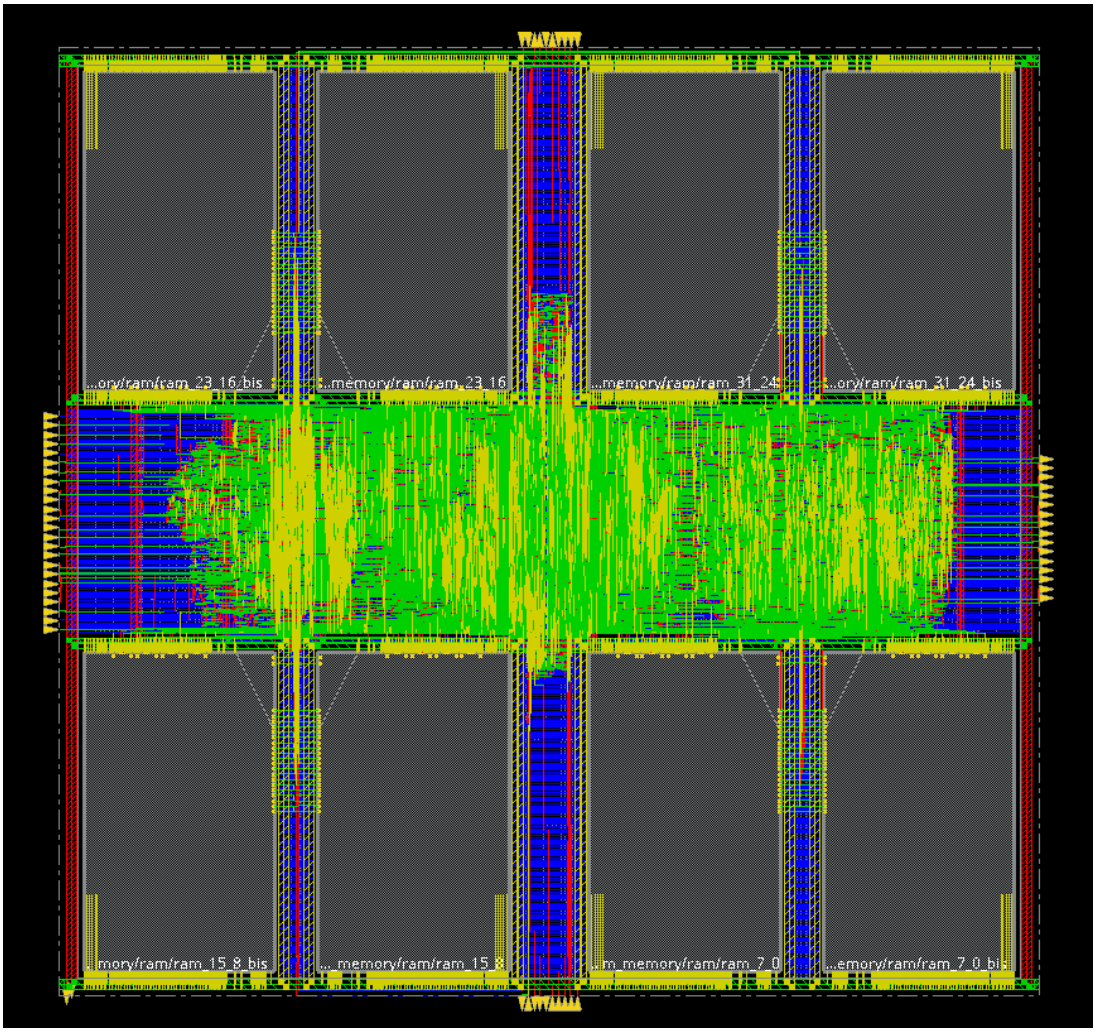

FIGURE 3.20
CPROC final layout

- A worst negative slack (WNS) for the hold mode of 19 ps.
- An announced power consumption of 12,5 mW. This result is actually most certainly not valid, since in the memory IPs datasheet provided by CERN, the worst consumption case for an IP is announced to be around 4mW. In the Innovus report, the total IPs consumption is about 5,77 mW. Cadence Voltus will have to be used to get a better result.

Moreover, it is important to consider results referring to the clock tree 3.21. In our case, the clock tree is distributed on three level, with 34 inverters used to send the same clock signal as much as possible to all sequential elements: 41,6 ps separate the time when the first sequential elements receive the clock (red squares on figure 3.21 and when the last ones receive it (deep blue squares on the left). The clock skew is about 51 ps, and the clock tree in itself has a power consumption of 0.9 mW, which is non negligible.

Four different corners (.mmmc files) are used by Innovus to creates those results: the typical

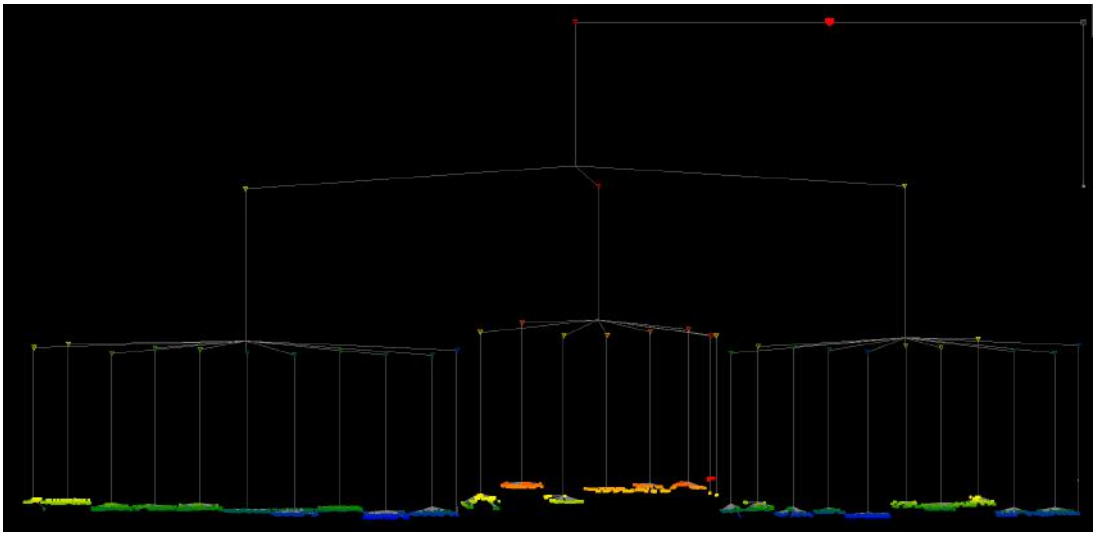

FIGURE 3.21
CPROC clock tree

case, the worst case, the best case, and the low temperature one. For the slack calculus (setup mode) the worst case corner is used, whereas for the WNS hold mode, the best case corner is used.
A lot of time was spent on the floorplan scripts, essentially because I still needed to correctly used the tools and the commands to be used. Some problems were encountered, but eventually overcame. One of them was an issue with the placement of decap filler cells, which are essential for decoupling purposes. Their placement created numbers of short issues, mainly because they were placed under vertical stripes (in M2), creating biasing problems since no metal layer was able to reach those cells pins. The solution was a simple command to prevent the placement of such cells under vertical stripes, and like that about 40K DRC violations disappeared.

On figure 3.22, two different Amoeba view of CPROC are displayed. On the left view, each block of the CPROC block diagram 3.14 are displayed in different colors: in red the CPU, in light blue the simpleuart module, in dark blue the gpio controller, in orange the spi master and in green the spimemio module. The two Ram_Controller and Flash_Controller block are two small to be observed here. In the right view, we descended into the hierarchy of the CPU, and we can see that the register file occupies a large part of the CPU footprint.

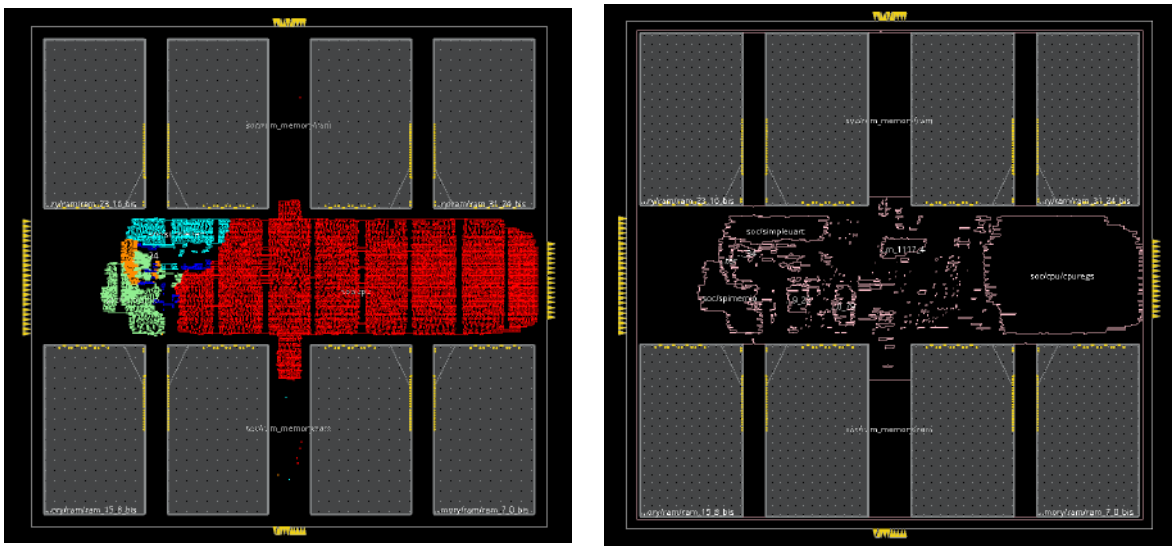

FIGURE 3.22
Amoeba Views of CPROC

CHAPTER 4

CONCLUSION

The work presented in this internship report is only the beginning of the CPROC project, and I will have the pleasure to pursue it and come up with a final first working version of it between October 2022 and February 2023, as an research Engineer at OMEGA Electronics.

To put it in a nutshell, the first chapter introduce the topic and goals of this internship, presenting the laboratory and its projects, but also a brief introduction to what is RISC-V and its history, and finally why using this specification.

The second chapter familiarized us to the basic of CPUs designs by describing its main components and roughly explaining how it works, but also with the notion of ISA that is inherent to CPU design. A broader presentation of the RISC-V base sets and its extensions is also offered in this section.

The third and last chapter presents the CPROC core and ISA choices, but also a different way of designing a register file using latches. The full compilation flow and digital flow are described with the software used, the files and eventual results. Finally, what CPROC is composed of is explained, along with explanations of every block.

Finally, as stated previously, the work in this paper retrace a great part of a CPU design project, but work still need to be done to finish it. The last modules that have not been tested yet have to be, and UVM verification has to be done on the SoC. When the chip will be fully tested, post synthesis testing using the same testbenchs will have to be done to ensure the good working of the chip. Then, using the same scripts already developed for the firsts layouts, the PnR will be done, and better results of power consumption and temperature of the chip while processing will be obtained with Cadence Voltus and Tempus. The Layout might have to be revised with respect to the latter results. Some clock gating might have to be done on the memory IPs to reduce the

power consumption of CPROC. The chip is programmed to be produced by the end of this year, or by the beginning of the next one, it will then have to be physically tested.

CHAPTER 5

ABSTRACT

5.1 FRENCH

À l'ère de l'IOT (Internet of tings), les processeurs sont omniprésents, qu'il s'agisse de gérer l'alimentation électrique, la durée de vie des batteries ou le traitement des signaux provenant de capteurs. La plupart d'entre eux utilisent des jeux d'instructions propriétaires de sociétés privées telles que ARM ou Intel. Si les entreprises qui vendent ces puces ne proposent pas la puce souhaitée, n'importe qui peut concevoir son propre processeur en utilisant l' ISA libre de droits RISC-V. Les processeurs RISC-V n'en sont qu'à leurs débuts, car la norme est relativement récente et, jusqu'à présent, principalement des CPU à un seul cœur ont été fabriquées. Pour l'instant, la majorité de ces processeurs libre de droit sont destinés aux FPGA, et ceux qui sont réélement fondus sous forme d'ASICs sont détenus par des entreprises qui ne partagent généralement que l' IP. Chez OMEGA Microelectronics, la décision de fabriquer un processeur ASIC RISC-V à un seul cœur a été prise : il est destiné à effectuer le post-traitement de données numériques et de la gestion interne dans des ASICS à signaux mixtes. La puce vise un faible encombrement, une faible consommation d'énergie, un maximum de quatre couches métalliques pour la partie numérique afin d'être à faible bruit, et utilisera le nœud 130 nm de TSMC. La première version de la puce apppelée CPROC, comportera deux maîtres SPI (un pour la communication externe, un autre pour communiquer avec une mémoire flash externe), un contrôleur UART, une SRAM interne de 8 ko, 24 entrées et 16 sorties et 4 pin d'IRQ, avec une fréquence de travail de 40 MHz et fonctionnant avec l'ISA RV32EC. Quelques blocs ont été pris dans des projets déjà existants, le reste a été conçu en interne. La puce est encore en cours de développement, elle nécessite des tests supplémentaires, cependant un premier layout est déjà disponible pour permettre aux ingénieurs analogiques de travailler sur les futurs composants analogiques de CPROC.

5.2 ENGLISH

In the era of the IOT (Internet of tings) processors are everywhere, managing power delivery or battery life or processing signals from sensors, and most of them runs proprietary instruction sets from private companies such as ARM or Intel. If companies selling those chips doesn't offer the specifically desired chip, anyone can design it's own CPU using the open-standard ISA RISC-V. RISC-V processors are only at their early stage as the standard is fairly new, and mainly single cor CPUs have been made up to today. As for now, the majority of those open CPUs deign targets FPGAs, and the one actually being ASICs are detained by companies generally sharing only the IP. At OMEGA Microelectronics, the decision to make an ASIC RISC-V single core processor has been made: it is intended to perform digital data post processing and internal monitoring in mixed signal ASICS. The chip targets a small footprint, low power consumption , a maximum of four metal layers for the digital part in order to be low noise, and will use the 130 nm node from TSMC. The first version of the chip, called CPROC, will feature two SPI master (one for external communication, an other to communicate with an external flash), and UART controller, and a 8 ko internal SRAM, 24 inputs and 16 outputs and 4 IRQ pins, with a working frequency of 40 MHz and running the RV32EC ISA. Few blocks were taken from already existing projects, the rest being designed in house. The chip is still under development, it needs further testing, however a first layout is already available to allow analog designers to work on the CPROC future analog components.

BIBLIOGRAPHY

[1] SiFive Inc. Andrew Waterman Krste Asanović. 'The RISC-V Instruction Set Manual'. In: Volume I : User-Level ISA. Document version 2.2 (2017).

[2] J. Todd McDonald William Mahoney. 'Enumerating x86-64 – It's Not as Easy as Counting'. In: (2019).

[3] Frederic Dulucq. *HGCROC3: the front-end readout ASIC for the CMS High Granularity Calorimeter*. 2021. URL: https://indico.cern.ch/event/1019078/contributions/4443949/.

[4] Selma Conforti Di Lorenzo. *HKROC: an integrated front-end ASIC to readout photomultiplier tubes for the Hyper-Kamiokande experiment*. 2022. URL: https://indico.cern.ch/event/1127562/contributions/4904493/.

[5] Maxime Morenas. *Performance of ALTIROC2 readout ASIC with LGADs for ATLAS HGTD picosecond MIP timing detector*. 2022. URL: https://indico.cern.ch/event/1127562/contributions/4904499/.

[6] Fastbit Embedded Brain Academy. *Bare metal embedded lecture-4: Writing linker scripts and section placement*. URL: https://www.youtube.com/watch?v=B7oKdUvRhQQ.

[7] ARM. *ARM flexible access*. URL: https://www.arm.com/products/flexible-access.

[8] Kirk Shimano Crystal Chen Greg Novick. *Risc architecture, CISC vs RISC*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[9] Efabless. *Raven: An ASIC implementation of the PicoSoC PicoRV32*. URL: https://github.com/efabless/raven-picorv32.

[10] RISC-V Foundation. *RISC-V core tests repository*. URL: https://github.com/riscv-software-src/riscv-tests.

[11] RISC-V Foundation. *RISC-V Foundation website*. URL: https://riscv.org/.

[12] RISC-V Foundation. *RISC-V GNU Compiler Toolchain*. URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[13] Efabless - Google. *Caravel SoC, Picorv32 based*. URL: https://github.com/efabless/caravel_pico.

[14] Wikipedia. *Wikipedia, the free encyclopedia*. Wikimedia Foundation. URL: https://www.wikipedia.org/.

[15] WINBOND. *1.2V Serial NOR Flash*. URL: https://www.winbond.com/hq/product/code-storage-flash-memory/1.2v-serial-nor-flash/?__locale=en&selected=64Mb#Density.

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

[17] Claire Wolf. *PicoSoC - A simple example SoC using PicoRV32*. URL: https://github.com/YosysHQ/picorv32/tree/master/picosoc.