UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEVELOPMENT OF A RISC-V PROCESSOR OPTIMISED FOR CONTROL APPLICATIONS TO BE USED IN THE LEVITATION SYSTEM OF HYPERLOOP

BACHELOR'S DEGREE FINAL PROJECT

Bachelor's Degree in Industrial Electronics and Automation Engineering

Author: RAMÓN ALAMÁN, David

Tutor: OLGUÍN PINATTI, Cristian Ariel

Co-tutor: MONZÓ FERRER, José María

Acknowledgements

I would like to thank my tutors Cristian Ariel Olguín and José María Monzó for the consistent support, keen interest, and invaluable assistance provided throughout the development of this project. And also for introducing me to the field of programmable logic devices in the subject Digital electronics.

I would also like to thank my gratitude to my parents for their unwavering support, I am quite sure they understand microcontrollers better now. I would like to extend to my siblings and friends.

Abstract

The present work develops, in SystemVerilog, a RISC-V IP core, both single-cycle and multicycle, employing the RV32I (32-bit integer handling RISC-V architecture) ISA (Instruction Set Architecture). A PID controller core IP has been developed to work alongside the processor. The main objective is to create a microprocessor optimised for control applications to be used in a hyperloop prototype. Hyperloop is a means of transportation concept that consists of a capsule that levitates in a vacuum tube to achieve high speeds with the lowest possible energy usage. Levitation control is a resource-intensive task in a microcontroller. A dedicated control peripheral reduces the use of the CPU, allowing it to perform other tasks. The use of FPGAs increases hardware flexibility to be modified, improving its performance without altering the PCBs, allowing process parallelisation and reducing power consumption. This project is based on the control system of the hyperloop prototype, Auran, developed by the Hyperloop UPV team for the 2022 European Hyperloop Week.

Resumen

El presente trabajo desarrolla, en SystemVerilog, un core IP de procesador RISC-V tanto single-cycle como multicycle implementando el ISA (*Instruction Set Architecture*) RV32I (RISC-V con arquitectura de 32 bits y manejo de números enteros), junto con un core IP de controlador PID que será conectado a él. El objetivo principal es desarrollar un microprocesador optimizado para aplicaciones de control para poder ser usado en un prototipo hyperloop. Hyperloop es un concepto de medio de transporte que consiste en una cápsula que circula levitando en un tubo de vacío para lograr altas velocidades con el menor uso de energía posible. El control de la levitación consume una gran cantidad de recursos en un microcontrolador, por lo que un periférico específico para realizar el control reduce la carga sobre la CPU, permitiendo ejecutar otras tareas. El uso de FPGAs mejora la flexibilidad del hardware para ser modificado incrementando las prestaciones, sin cambio en las PCBs, posibilita la paralelización de tareas y reduce el consumo. Este proyecto tiene como base el sistema de control del prototipo hyperloop, Auran, desarrollado por el equipo Hyperloop UPV para la European Hyperloop Week de 2022.

Resum

El present treball desenvolupa, en SystemVerilog, un core IP de processador RISC-V tant single-cycle com multicycle implementant l'ISA (*Instruction Set Architecture*) RV32I (RISC-V amb arquitectura de 32 bits i maneig de nombres enters), juntament amb un core IP de controlador PID que serà connectat a ell. L'objectiu principal és desenvolupar un microprocessador optimitzat per a aplicacions de control per a poder ser usat en un prototip hyperloop. Hyperloop és un concepte de mitjà de transport que consisteix en una càpsula que circula levitant en un tub de buit per a aconseguir altes velocitats amb el menor ús d'energia possible. El control de la levitació consumeix una gran quantitat de recursos en un microcontrolador, per la qual cosa un perifèric específic per a realitzar el control redueix la càrrega sobre la CPU, permetent executar altres tasques. L'ús de FPGAs millora la flexibilitat del maquinari per a ser modificat incrementant les prestacions, sense canvi en les PCBs, possibilita la paral·lelització de tasques i redueix el consum. Aquest projecte té com a base el sistema de control del prototip hyperloop, Auran, desenvolupat per l'equip Hyperloop UPV per a l'European Hyperloop Week de 2022.

Contents

List of Figures

List of Tables

List of Code snippets

Acronyms & Initials

ADC	Analogue to Digital Converter
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DLIM	Dual Linear Induction Motor
DSP	Digital Signal Processor
DUT	Device Under Test
EHW	European Hyperloop Week
EMS	Electromagnetic Suspension
EX	Execution – Instruction stage
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
HDL	Hardware Description Language
HDVL	Hardware Description Verification Language
HEMS	Hybrid Electromagnetic System
IC	Integrated Circuit
ID	Instruction Decode – Instruction stage

IF	Instruction Fetch – Instruction stage
IP	Intellectual Property
ISA	Instruction Set Architecture
LPU	Levitation Power Unit
LSB	Least Significant Bit
LVCMOS33	3.3 V Low Voltage Complementary Metal-Oxide Semiconductor
LVTTL	Low Voltage Transistor-Transistor Logic
MCU	Microcontroller unit
MEM	Memory access – Instruction stage
MSB	Most Significant Bit
PC	Program Counter
PCB	Printed Circuit Board
PID	Proportional Integral Derivative
PLA	Programmable Logic Arrays
PLD	Programmable Logic Device
PLL	Phase-locked Loop
PWM	Pulse Width Modulation
PWMN	Complementary Pulse Width Modulation
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SoC	System-on-Chip
TTL	Transistor-Transistor Logic
WB	Write Back – Instruction stage

1 Scope

This project aims to develop a RISC-V core IP (Intellectual Property) focused on being employed in control applications. The MCU (microcontroller unit) has been developed employing SystemVerilog, a hardware description language. This microcontroller has been designed employing the RISC-V architecture, and partially supports the Instruction Set Architecture (ISA) RV32I[1].

In the following sections, two different implementations of the MCU core IP have been described. The first version, which is the implementation of a single-cycle microcontroller, has been developed for simulation purposes only. Moreover, a multicycle version has also been developed, which has not only been simulated but also synthesized and loaded into a FPGA based PLD (Programmable Logic Device).

Regarding the control application of the MCU, a series of peripherals have also been developed to allow the parallel computation of up to two PID controllers and the MCU core itself. This fact is possible thanks to the implementation of a PID controller peripheral that, along with a timer and ADC (Analogue to Digital Converter) peripherals, can run a control loop autonomously.

The capabilities of this control-oriented microcontroller perfectly fit the requirements of the levitation system of a hyperloop prototype. This application has been tested by substituting the dedicated commercial microcontroller employed to control the electromagnets from the levitation and guiding system of Auran by the developed MCU. Auran is the fifth hyperloop prototype developed by the university team Hyperloop UPV.

1.1 RISC-V

RISC-V is an open-source ISA based on the RISC concept, which stands for Reduced Instruction Set Computer. The basis of RISC is to have a reduced number of instructions that can be executed faster. This approach contrasts with the CISC (Complex Instruction Set Computer) architectures, which implement a large number of instructions, including complex operations, therefore sacrificing hardware simplicity and speed but simplifying programming.

The RISC-V architecture was born at the University of California, Berkeley, in 2010 and "was originally designed to support computer architecture research and education" (Waterman and Asanović, 2019). Nevertheless, this architecture is gaining popularity at all levels (educational, industrial, for hobbyists) and becoming a genuine alternative

[1] RV32I is the RISC-V ISA base that supports 32-bit integer architecture.

in the industry because of its open-source character that allows anyone to create their own implementation without paying royalties.

Figure 1.1: RISC-V Logotype, (RISC-V Foundation, 2018)

The RISC-V community is continuously growing. RISC-V Foundation currently has over 3,100 members from 70 countries (RISC-V Foundation, nd) and expects that by 2025 the market will consume 62.4 billion RISC-V cores (Osier-Mixon, 2019).

1.2 Programmable Logic Devices

Programmable Logic Devices are electronic components that implement reconfigurable digital circuits. Therefore, its function is not determined by the manufacturer but by the user. PLDs are a fundamental part of contemporary digital electronics. They originated in the 1970s when Programmable Logic Arrays (PLA) were first developed. Since then, PLDs have evolved, and different types have arisen, such as CPLDs (Complex Programmable Logic Devices) and FPGAs (Field Programmable Gate Array).

Modern FPGAs do not only incorporate programmable logic. They also include ADC circuitry, memory block and arithmetic circuits, among others. Furthermore, they can have outstanding performance and characteristics. For example, the Xilinx Virtex UltraScale+ VU19P FPGA has 9 million logic cells, over 2,000 GPIOs (General Purpose Input Output) and supports up to eight DDR4 at 1.5 Tb/s bandwidth. (Xilinx®, nd)

Due to their characteristics, FPGAs can be employed in a wide range of applications. However, its use is prominent in parallel computing, hardware accelerators and digital integrated circuit prototyping.

> While currently only a tiny share of FPGAs are used for data processing in data centers, this is likely going to change in the near future as FPGAs not only provide very high performance, but they are also extremely energy efficient computing devices. (Koch et al., 2016)

1.3 SystemVerilog

SystemVerilog is a hardware description language (HDL) presented in 2002 as an extension of Verilog. Since then, SystemVerilog has become one of the mainly used HDLs. A hardware description language is a computer language used to define the structure and behaviour of electronic circuits. They also allow to simulate or synthesise the circuits to generate hardware implementations that can be implemented in an FPGA or an ASIC. Other HDLs are VHDL and Verilog.

They differ from programming languages (such as C/C++, Python or Java) mainly in their purpose. While HDLs are used to describe and design digital electronics circuits, programming languages generate a set of instructions (software) executed by a CPU.

The most remarkable characteristic of SystemVerilog is its support for verification techniques. These technics include testbenches, functional coverage, constraint random testing and specifying assertions; thus making SystemVerilog an HDVL – Hardware Description Verification Language – (Cerny et al., 2010).

Figure 1.2: SystemVerilog Logotype, (Antmicro, 2019)

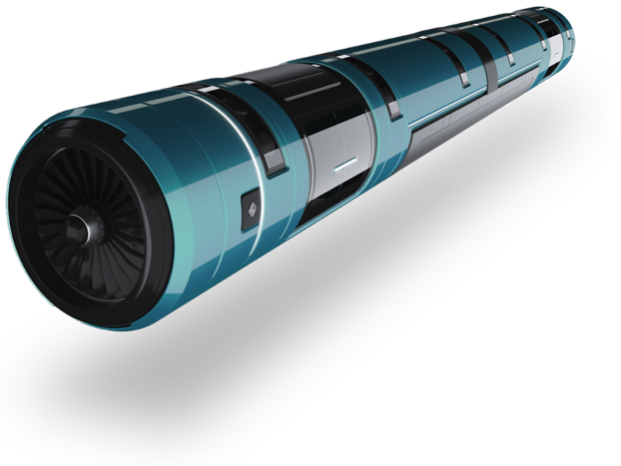
1.4 Hyperloop concept & Hyperloop UPV

Hyperloop a is means of transportation concept. It was first devised in the 19th century. However, it was not until 2013 that its popularity began to increase its popularity when the companies SpaceX and Tesla retrieved the idea and published the Hyperloop Alpha concept.

The main idea behind the hyperloop concept is to achieve a high-speed means of transportation, up to 1,000 km/h, with minimal energy consumption. Hyperloop consists of a capsule, or pod, that circulates levitating electromagnetically inside a vacuum tube propelled by a linear electric motor. Both the electromagnetic levitation and the linear electric motor eliminate the friction with the infrastructure. Moreover, the vacuum tube reduces the air resistance effect, which is proportional to the square of

the speed.

Figure 1.3: Hyperloop concept by Hyperloop UPV, (Hyperloop UPV)



In 2015 Spacex began organising the Hyperloop Pod Competition. That led to the creation of a team at the Polytechnic University of Valencia, Hyperloop UPV (in which I have been involved since 2021 as a firmware and hardware engineer), to participate in the competition. Since then, the team has participated in all editions until 2019, but due to COVID-19, the event was discontinued.

Later, in 2021, four European teams (Hyperloop UPV, Delft Hyperloop, Swissloop and HYPED) founded the European Hyperloop Week (EHW), a worldwide competition that would maintain the university hyperloop ecosystem.

For the second edition of the EHW, Hyperloop UPV developed the prototype Auran (Figure **1.4a**). Auran was an approach to a full-scale hyperloop vehicle. A functional electromagnetic levitation and guiding system was successfully implemented. And along with the dual linear induction motor (DLIM) allowed the prototype to travel through a 20-meter-long tubular infrastructure (Figure **1.4b**) without any contact with the infraestructure.

Figure 1.4: Auran and infrastructure, (Hyperloop UPV)

(a) Auran

(b) Infrastructure section

Electromagnetic levitation cannot only rely on electromagnets to suspend the hyperloop vehicle as this would be a power-intensive application. A combination of permanent magnets and electromagnets must be employed to approach the objective of zero power consumption. Each type of magnet accomplishes a specific function. Permanent magnets are employed to suspend the whole mass of the vehicle as they generate a magnetic field without applying energy.

The magnetic force generated by the permanent magnets decreases with the distance to the infrastructure. Therefore, there is a distance where the magnetic force and the gravitational force cancel each other. This equilibrium point depends on the mass of the vehicle and the electromagnetic unit. However, the equilibrium is unstable, meaning that a small perturbance or position error results in the vehicle either falling or adhering to the track.

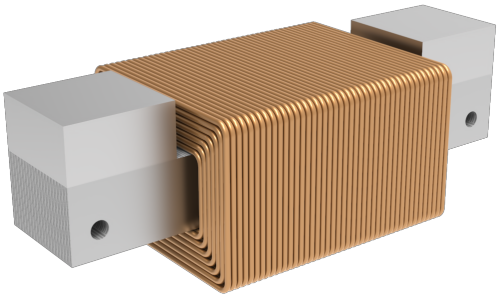
Due to the aforementioned situation, a control system must be implemented where the actuators are the electromagnet, which modifies the magnetic field to return to the equilibrium distance. Thus, a permanent magnetic field generates the force to suspend the vehicle, and the energy is employed only to correct deviation from the equilibrium distance. In this project, the controlled variable is the current flowing through the coil (electromagnet).

The levitation and guiding system is mainly composed of two types of electromagnets. The levitation-oriented electromagnets are a Hybrid Electromagnetic Suspension (HEMS) system. They consist of two permanent magnets and a coil around a laminated steel yoke, as shown in Figure **1.5a**. The guiding-oriented electromagnets are an Elec-

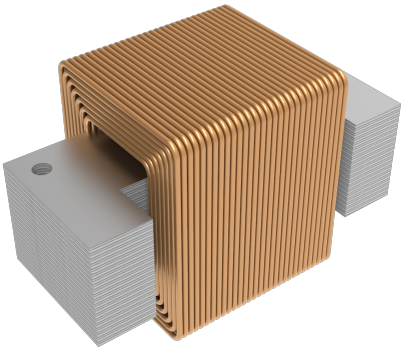
tromagnetic Suspension (EMS) system. They consist only of a coil around a laminated steel yoke, as shown in Figure **1.5b**.

Figure 1.5: Levitation and guiding system electromagnets, (Hyperloop UPV)

(a) HEMS



(b) EMS



The prototype has individual PCBs (Printed Circuit Board) known as LPUs (Levitation Power Units) whose function is to drive the coils of the levitation and guiding systems. These PCBs also include a microcontroller from the STM32F3 family that executes a PI controller to generate the control signals for the power stage of the driver, given a current reference.

The objective of the project is to substitute the aforementioned general-purpose microcontroller with a microcontroller implemented into FPGA capable of performing the PI controller freeing the CPU to perform other operations and demonstrate the applicability of programmable logic devices in hyperloop applications.

2 Needs study

As per the nature of the project, there are no significant limiting conditions. The microcontroller is intended to be employed in different control-oriented applications. However, a fundamental reason for the development of this project is the application of the system in the levitation system of a hyperloop prototype. Specifically, to control the current through electromagnets of the Hyperloop UPV prototype, Auran. Therefore, the new system must be able to fit with the hardware of the prototype. It must also be able to handle the current control loop. Given the aforementioned conditions, the requirements are defined in Table **2.1**

Table 2.1: Design requirements

ID	Requirement	Peripheral
RQMT 1	The MCU must be able to perform at least one PID loop at 1 kHz.	PID
RQMT 2	Once configured, the PID controller must be able to run autonomously; without using CPU cycles.	PID / ADC
RQMT 3	The MCU must have at least one ADC channel capable of reading voltages up to 3.3 V	Timer
RQMT 4	The MCU must be able to generate two PWM (Pulse Width Modulation) signals and their complementary.	Timer
RQMT 5	The PWM signals frequency must be 10 kHz.	Timer
RQMT 6	The PWM signals must have a configurable dead time.	Timer
RQMT 7	PWM signal output must be 3.3 V or 5 V TTL[2]/CMOS[3] logic. (TTL, LVTTL, CMOS, LVCMOS33).	Timer

[2]Transistor-Transistor Logic
[3]Complementary Metal-oxide Semiconductor

3 Alternative solutions

Embedded control systems can be developed employing a variety of hardware solutions. For instance, some alternatives are microcontrollers, programmable logic devices, digital signal processors (DSP) or System-on-Chip (SoC). Selecting one alternative or another may be based on technical, economic or know-how criteria. In this document, the focus will be placed on the MCU and PLD solutions.

The microcontroller-based solution stands out in the economic and know-how criteria. Inside the Hyperloop UPV team, every embedded system has been developed employing MCUs. Therefore, there is a knowledge base from which new integrants can learn and develop from already-developed code libraries. Since a new prototype is designed every year, rapid development is highly valued. Also, microcontrollers are a cheaper alternative when compared to FPGAs. Thus for the economic criterion, this solution also highlights.

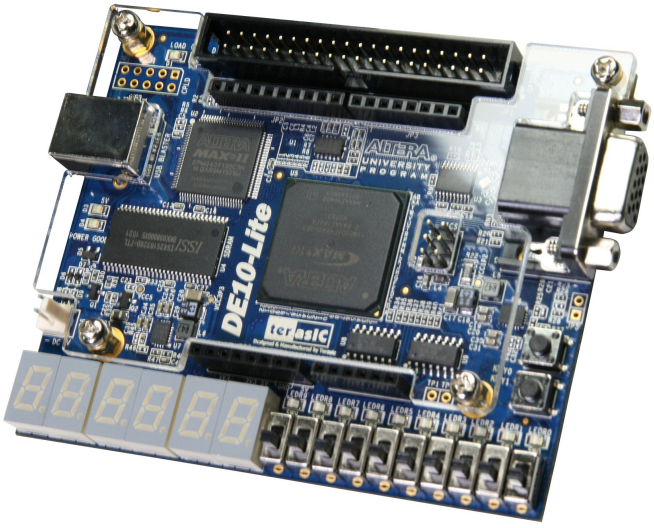
Regarding the technical criteria, both the microcontroller-based and the programmable logic-based solutions have their benefits and drawbacks. On the one hand, MCUs consume less power than PLDs. This fact makes them more suitable for battery-powered applications, such as hyperloop.

However, on the other hand, PLDs have the main benefit of being reconfigurable. This characteristic allows the integrated circuit (IC) to generate the logic needed for a specific application. That means that overheads are generated on a microcontroller due to the ability to run a single instruction per clock cycle. They are also more suitable for high-speed processing and data throughput is demanded. Additionally, they can adapt to changing needs at lower costs (Zhang, 2010), which is crucial in fast-developing environments such as Hyperloop UPV.

Furthermore, FPGAs allow the configuration of a soft IP of a CPU; that is that the hardware of a processing unit can be programmed into the FPGA. It is even possible to generate a multicore processor inside an FPGA and provide it with custom peripherals.

Regarding the requirements in Section **2**, the programmed-logic solution allows configuring the PLD to fulfil all the conditions. So this will be the solution employed. Moreover, one of the objectives of this work is to show the potential of programmed logic devices in control applications.

Figure 3.1: Terasic DE10-Lite, (Terasic, nd)



The device selected for this project is the Intel MAX10 10M50DAF484C7G FPGA (Figure **3.1** The reasons behind this decision are that it is a low-price device and that Terasic has developed an evaluation board for this device. Some of its characteristics are:

- 50,000 logic elements
- 484 pins
- 2 ADCs (17 channels, 12-bit resolution)
- 144 18-bit multipliers
- 4 PLLs[4]
- Internal flash memory

The software employed for the development of the project is the following:

- **Visual Studio Code** (Version: 1.78.2) with the following extensions:
  - **SystemVerilog - Language Support** (Version: 0.13.3): Used for developing the modules and testbenches.
  - **RISC-V Venus Simulator** (Version: 1.7.0): Development of the RISC-V assembly files.
- **Venus**[5]: Online RISC-V assembler and simulator

[4]Phase-locked Loop

[5]Venus assembler can be accessed through: `https://venus.kvakil.me`

- **Quartus Prime Lite** (Version 18.1.0): Used for synthesis and timing analysis.
- **ModelSim - Intel FPGA Starter Edition** (Version: 10.5b): Verification of the SystemVerilog modules.
- **MATLAB** (Version: R2022a) with add-ons **Simulink** (Version: 10.5) and **Control System Toolbox** (Version 10.11.1): Development and simulation of the controller.

4 Development

Throughout Sections **4** and **5**, the project development and verification process will be described. Section **4** details the design of the microcontroller. Firstly, the RISC-V architecture and instruction set architecture will be further explained. Then a single-cycle version, Section **4.2**, will be introduced to evaluate the RISC-V architecture, followed by the multicycle implementation, Section **4.3** along with the memory bus and the peripherals implemented. The multicycle implementation will not remain as a simulation model, as it will be synthesised in the FPGA. The SystemVerilog that describes every module developed can be found in Part **??**. Section **5** describes the verification process of the different modules developed and reports the results obtained. The different testbenches, assembly code, and other testing files can also be found in Part **??**.

4.1 RISC-V

RISC-V architecture is based on a modular basis. The concept of modularity allows the architecture to support extensive customisation and specialisation. "The main advantage of explicitly separating base ISAs is that each base ISA can be optimized for its needs without requiring to support all the operations needed for other base ISAs" (Waterman and Asanović, 2019). This modularity is obtained by designing a base ISA with minimal support and a batch of instruction-set extensions that allow increasing functionality of the base ISA. RISC-V base ISA has been developed to stand integer numbers. Then, depending on the application of the designed processor, one or multiple of the previously mentioned instruction-set extensions are added to create an application-designed ISA.

There are three types of instruction-set extensions: *standard*, *reserved* and *custom*. *Standard* instruction sets are defined by the RISC-V Foundation. *Reserved* instruction sets are encodings that have not been defined yet but that the RISC-V Foundation has earmarked for future ISA extensions or updates. Finally, *custom* instruction sets are encodings left for designers to create new instructions outside standard instruction sets. If a custom extension employs a reserved encoding or an encoding already defined inside the RISV-C standard sets, this extension is defined as *non-conforming*.

The RISC-V base integer ISA is named "I", preceded by the word size of the architecture (RV32 for 32-bit word size, RV64 for 64-bit word size and RV128 for 128-bit word size). When an instruction-set extension is added to the base ISA, a suffix is added to the ISA name. Suffixes are shown in Table **4.1**.

Table 4.1: Standard ISA extensions suffixes

Suffix	Extension
M	Integer multiplication and division
A	Atomic instructions
F	Single-precision floating-point
D	Double-precision floating-point
C	Compressed instructions[6]

For example, an ISA including a 32-bit integer base, integer multiplication and division, atomic instructions, and single-precision floating-point extension will be named RV32IMAF.

4.1.1 Instruction Set Architecture

4.1.1.1 Registers

The selected instruction set architecture for the project is the RV32I. This ISA determines that there are 32, 32-bit wide, registers (`x0-x31`). From which, register `x0` is hardwired to 0, and the registers `x1-x31` are defined as general-purpose registers. Even though the ISA does not specify the function of the general-purpose registers, there is a usage convention (shown in Table **4.2**).

This table also indicates which function (caller or callee) must save the registers in the stack when calling another function.

Finally, a register called PC (program counter) is also defined in the RV32I. The program counter stores the memory address of the instruction to be executed in the following cycle.

[6]Defines a 16-bit form for common instructions.

Table 4.2: RISC-V register convention, (Waterman and Asanović, 2019)

Register	ABI Name	Description	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 register	Callee
x28–31	t3–t6	Temporaries	Caller

4.1.1.2 Instructions

Regarding the instructions defined in the ISA, there are four types of instructions (R, I, S and U) and two variations; B, derived from S-type; and J, derived from U-type. Table **4.3** depicts the format of the instructions.

Table 4.3: RISC-V register convention, (Harris, 2022)

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**

The definition RV32I includes 37 instructions divided into three main groups, integer computational instructions, control transfer instructions and load and store instructions. It also incorporates two system instructions (*ebreak* and *ecall*). Table **4.4** displays the instruction along with its format, and the operation it performs.

Table 4.4: RV32I Instructions, (Harris, 2022)

Instruction	Format	Operation
lb	I	`rd = SignExt([Address]7:0)`
lh	I	`rd = SignExt([Address]15:0)`
lw	I	`rd = [Address]31:0`
lbu	I	`rd = ZeroExt([Address]7:0)`
lhu	I	`rd = ZeroExt([Address]15:0)`
addi	I	`rd = rs1 + SignExt(imm)`
slli	I	`rd = rs1 << uimm`
slti	I	`rd = (rs1 < SignExt(imm))`
sltiu	I	`rd = (rs1 < SignExt(imm))`
xori	I	`rd = rs1 ^ SignExt(imm)`

srli	I	`rd = rs1 >> uimm`
srai	I	`rd = rs1 >>> uimm`
ori	I	`rd = rs1 | SignExt(imm)`
andi	I	`rd = rs1 & SignExt(imm)`
auipc	U	`rd = {upimm, 12'b0} + PC`
sb	S	`[Address]`$_{7:0}$ `= rs2`$_{7:0}$
sh	S	`[Address]`$_{15:0}$ `= rs2`$_{15:0}$
sw	S	`[Address]`$_{31:0}$ `= rs2`
add	R	`rd = rs1 + rs2`
sub	R	`rd = rs1 - rs2`
sll	R	`rd = rs1 <<` `rs2`$_{4:0}$
slt	R	`rd = (rs1 < rs2)`
sltu	R	`rd = (rs1 < rs2)`
xor	R	`rd = rs1 ^ rs2`
srl	R	`rd = rs1 >>` `rs2`$_{4:0}$
sra	R	`rd = rs1 >>>` `rs2`$_{4:0}$
or	R	`rd = (rs1 | rs2)`
and	R	`rd = (rs1 & rs2)`
lui	U	`rd = {upimm, 12'b0}`
beq	B	`if (rs1 == rs2) PC = BTA`
bne	B	`if (rs1 != rs2) PC = BTA`
blt	B	`if (rs1 < rs2) PC = BTA`
bge	B	`if (rs1 >= rs2) PC = BTA`
bltu	B	`if (rs1 < rs2) PC = BTA`
bgeu	B	`if (rs1 >= rs2) PC = BTA`
jalr	I	`PC = rs1 + SignExt(imm), rd = PC + 4`
jal	J	`PC = JTA, rd = PC + 4`

The nomenclature employed in Table **4.4** is the following:

- imm: signed immediate in imm11:0
- uimm: 5-bit unsigned immediate in imm4:0
- upimm: 20 upper bits of a 32-bit immediate, in imm31:12
- Address: memory address: rs1 + SignExt(imm11:0)
- [Address]: data at memory location Address
- BTA: branch target address: PC + SignExt(imm12:1,1'b0)
- JTA: jump target address: PC + SignExt(imm20:1,1'b0)
- label: text indicating instruction address
- SignExt: value sign-extended to 32 bits
- ZeroExt: value zero-extended to 32 bits
- rs1 & rs2: Register outputs of the register file

4.2 Single-cycle processor

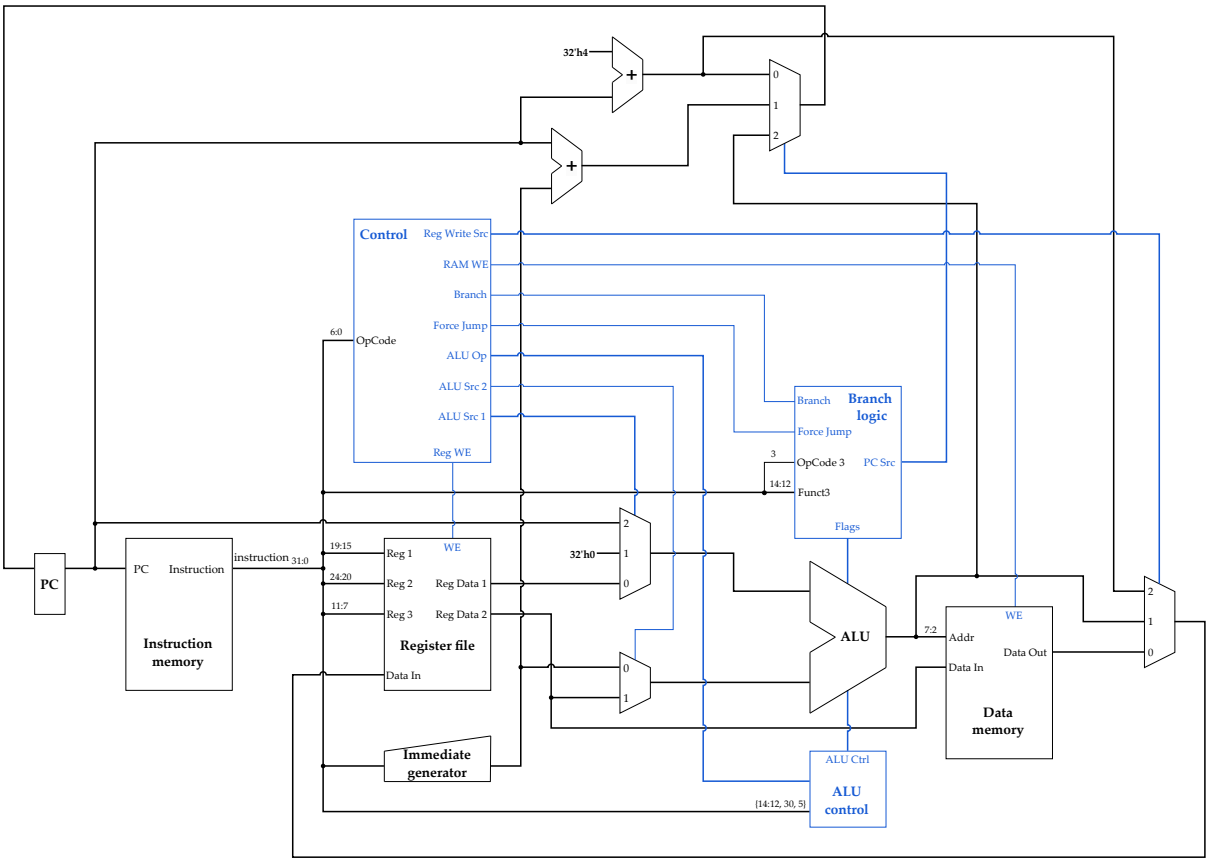
The main characteristic of a single-cycle processor is that it executes a complete instruction each cycle. Consequently, the processor mainly implements combinational logic, which is responsible for updating the processor state (register file, program counter and data memory) before the next active clock edge.

A processor is divided into two blocks:

- Datapath: implements the logic necessary to execute the instructions.
- Control: defines the behaviour of datapath according to the instruction that is being executed.

In Figure **4.1**, the datapath is represented in black and the control in blue. The following sections explain every building block implementation, beginning with the datapath and then moving to the control.

Figure 4.1: Single-cycle processor



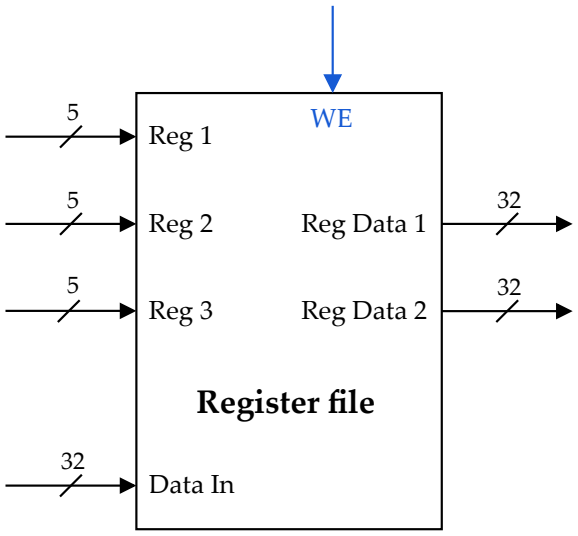
4.2.1 Datapath

4.2.1.1 Register file

Following the requirements given in the ISA, the register file must implement 32 registers (32-bit wide), two reading ports and one writing port, and the register `x0` must be hardwired to zero. Furthermore, as required in a single-cycle processor, the reading port must be asynchronous, and the writing port must be synchronous.

Therefore, the final implementation will result, as shown in Figure **4.2**. The register file has four input ports corresponding to the addresses of the ports and the writing port itself. The address inputs are 5-bit, needed to map all 32 registers ($\log_2 32 = 5$). There are two output ports, which are the reading ports. Additionally, a write enable input has been added in order not to modify the contents of the registers if it is not specified in the current instruction.

Figure 4.2: Register file interface



Regarding the implementation in SystemVerilog, the module has been designed by implementing the previously mentioned ports. Additionally, a clock input and an asynchronous reset have been added. The former allows synchronous reading, and the latter to reset the module from an external source.

The asynchronous reading employs an `assign` statement and the conditional operator. The conditional block asks whether the address is not equal to zero, and if true, it assigns the value of the desired register or zero (hardwired `x0` register) otherwise.

Finally, the synchronous read has been developed using an `always_ff` block, in the sensitivity list the rising edge of the clock and the falling edge of the reset. The registers are set to zero if the reset is active (low level). Furthermore, if the reset is not active and the write enable signal is, the "dataIn" port value is assigned to the corresponding register.

4.2.1.2 Immediate generator

The purpose of the Immediate generator is to transform the immediate encoded in the instruction to a 32-bit number. To achieve this, the block receives as an input the instruction and according to the type of instruction it generates a 32-bit immediate, which is the output of the block (see Figure **4.3**). The immediate generator behaviour is shown in Table **4.5**.

Figure 4.3: Immediate generator interface

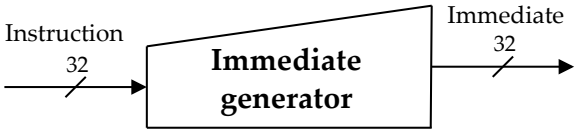


Table 4.5: Immediate generator encoding

Type	Immediate
I	`{{21{inst[31]}}, inst[30:20]}`
S	`{{21{inst[31]}}, inst[30:25], inst[11:7]}`
B	`{{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0}`
U	`{inst[31:12], 12'b0}`
J	`{{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0}`

Note: The encoding employs SystemVerilog notation

The immediate generator is the implementation of Table **4.5** inside a `always_comb` block in order to be able to use the `case` statement. The condition has been defined employing the *OpCode*. In Table **4.6**, each *OpCode* is shown with its corresponding instruction type.

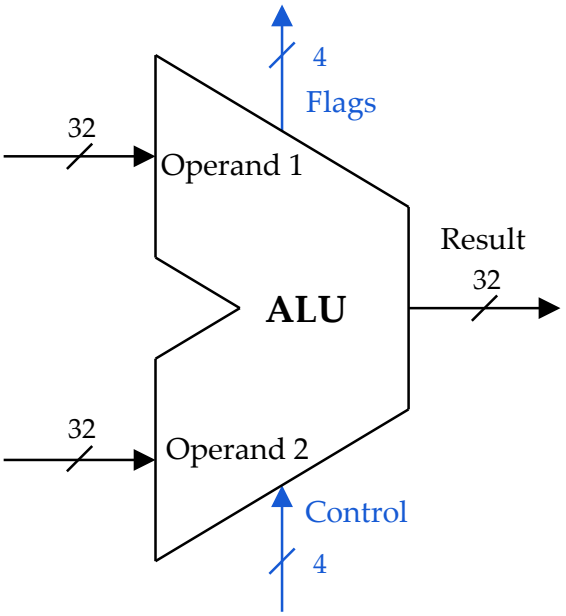
Table 4.6: Instruction type OpCodes

Type	OpCode
I	0000011_2, 0010011_2, 1100111_2
S	0100011_2
B	1100011_2
U	0010111_2, 0110111_2
J	1101111_2

4.2.1.3 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the combinational circuit that performs the mathematical (i.e. addition) and logical operations (i.e. logical AND). The ALU implements an interface with two inputs for the operands and one output for the result. Additionally, it has an input for the control signals and the *flags* output that will be used in the *branch logic* module. The former will be further explained in Section **4.2.2.3**. The interface of this module is shown in Figure **4.4**.

Figure 4.4: Arithmetic Logic Unit interface



The ALU must perform the following operations: addition, subtraction, AND, OR, XOR, logical and arithmetical shift and numerical comparison. Each instruction requires a determined operation. In Table **4.7**, it is specified which arithmetic or logical operation demands each instruction.

Table 4.7: Operations performed in each instruction

Instruction	Operation	Instruction	Operation	Instruction	Operation
lw	+	addi	+	slli	<<
slti	A < B	sltiu	A < B	xori	∧
srli	>>	srai	>>>	ori	|
andi	&	auipc	+	sw	+
add	+	sub	−	sll	<<
slt	A < B	sltu	A < B	xor	∧
srl	>>	sra	>>>	or	|
and	&	lui	+	beq	A < B
bne	A < B	blt	A < B	bge	A < B
bltu	A < B	bgeu	A < B	jalr	+
jal	+				

For each pair of operands, the ALU performs every operation concurrently. Each result is then input to a multiplexer, where the control input signals select the output of the ALU. The internal diagram of the ALU is depicted in Figure **4.5**.

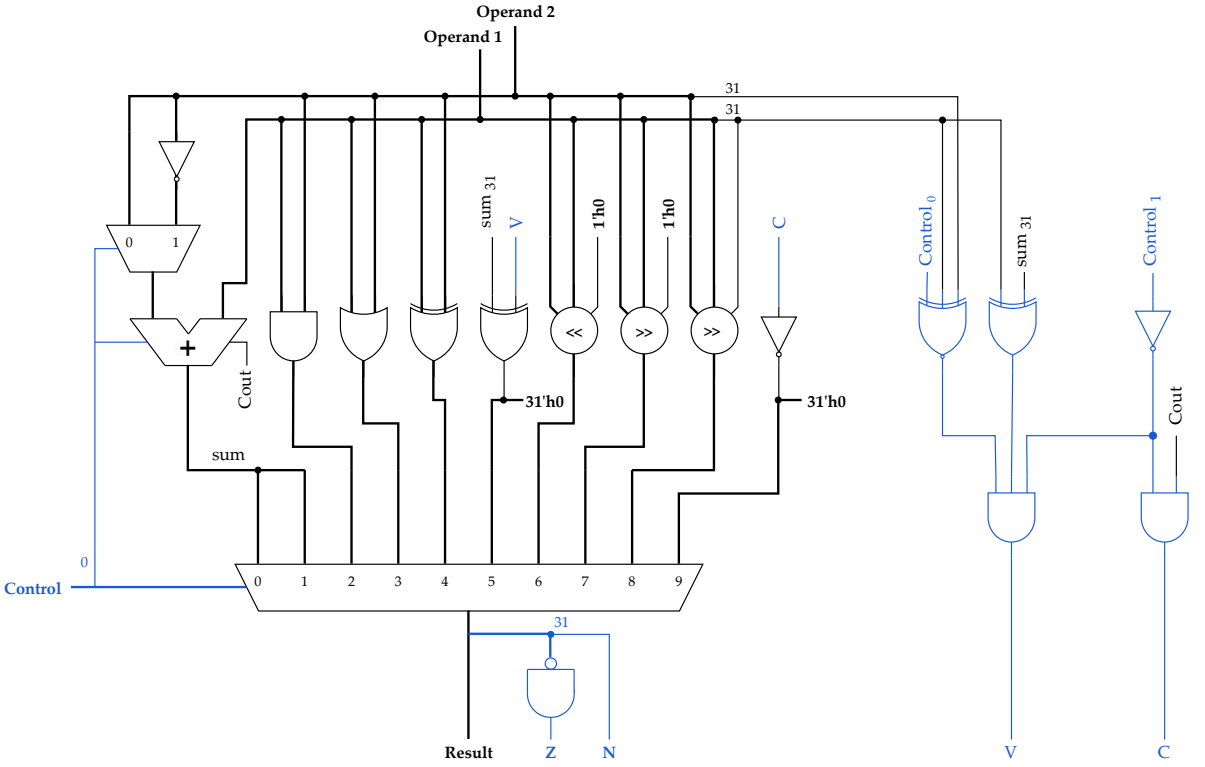
It is important to remark that to reduce the hardware necessary to implement the ALU, the addition and subtraction operations have been implemented using a single full adder circuit. Thanks to the 2's complement that specifies that $\overline{B} + 1 = -B$, it is possible to invert the second operand and, using the carry input of the full adder, perform $A + \overline{B} + 1 = A - B$. The carry input of the full adder is provided by the LSB (least significant bit) of the control signal.

Another relevant circuit of the ALU is the "flags" signals, which have two purposes. The first is to allow the computation of the jump condition in type B instructions. This calculation is performed in the branch logic module. The second purpose is to compute the result of the "slt" and "sltu" instructions.

There are four flags "Zero" (Z), "Negative" (N), "Carry" (C) and "oVerflow"[7] (V). "Zero" is active when all values of the "result" signal are set to zero. The "negative" signal corresponds to the MSB (most significant bit) of the "result" signal. The "carry" signal is set to a high level when the full adder produces a carry, and the operation performed is an addition or a subtraction (control bit 1 equal to zero). Finally, the

[7]Overflow is denoted by letter V to avoid confusions with number zero.

Figure 4.5: Arithmetic Logic Unit internal logic



"overflow" signal is asserted when the addition of two signed numbers produces a result with the opposite sign. Three conditions must occur to assert the "overflow" signal: An addition or subtraction must be performed. The first operand and the MSB of the full adder output must differ. Overflow can occur. If an addition is being calculated, A and B must have the same sign. In case a subtraction is being performed, A and B must have different signs. Hence, to execute "slt" (set less than) and "sltu" (set less than unsigned) instructions, a comparison[8] must be made. A subtraction operation shall be conducted to perform a comparison, and then the flags shall be used to evaluate the conditions. To perform a "less than" operation, N$\wedge$V must be evaluated for signed operands and $\overline{C}$ for unsigned operands.

Lastly, it is worthwhile to indicate the logic behind the control signals encoding. Table **4.8** shows the requirements for each operation and the final encoding selected.

[8]Comparisons are further explained in Section **4.2.2.3**

Table 4.8: Arithmetic Logic Unit control encoding

Operation	Rqmt.	Code	Operation	Rqmt.	Code
addition	XXX0	0000	SLT	XX01	0101
subtraction	XXX1	0001	<<	XXXX	0110
AND	XXXX	0010	>>	XXXX	0111
OR	XXXX	0011	>>>	XXXX	1000
XOR	XXXX	0100	SLTU	XX01	1001

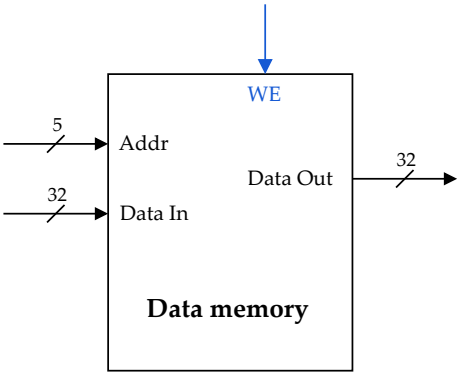
4.2.1.4 Data memory

The data memory is the module where the program variables can be stored. As variables can be either read or written, this module implements a RAM (Random Access Memory). Also, as stated in Section **4.2.1.1**, the memory must be designed for asynchronous reading and synchronous reading.

The data memory module has been designed to be versatile by employing the parameters functionality of SystemVerilog. Both word size and memory depth are defined by the parameters `WORD_SIZE` and `DEPTH`, respectively. For this application, the default values selected for the data memory are `WORD_SIZE` = 32 and `DEPTH` = 1024, resulting in a 4 KiB memory.

The interface defines `WORD_SIZE` wide (32-bit) read and write ports. Moreover, a write enable input has been added in order not to modify stored data in other instructions different from "sw". Finally, the address port depends on the `DEPTH` parameter. Its width is defined as $\log_2$ (`DEPTH`). Figure **4.6** shows data memory interface.

Figure 4.6: Data memory interface



4.2.1.5 Instruction memory

The instruction memory is the module where the instructions are stored. These instructions cannot be written on runtime, and as the processor has not been designed to be programmed, this memory shall only be read. Hence, the module implements a Read Only Memory (ROM). In order to load the compiled program into the memory, an `initial` block is used along with the `$readmemh()` instruction (see Code snippet **4.1**). As previously mentioned, the memory must feature asynchronous reading.

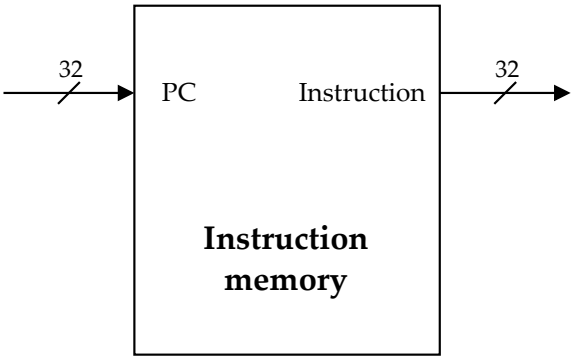
Code snippet 4.1: Load compiled code

```
1    logic[31:0] memory[DEPTH-1:0];
2
3    initial begin
4        $readmemh("compiledcode.hex", memory);
5    end
```

Similarly to the data memory, the instruction memory has been designed to be configurable. Also, it implements the `DEPTH` parameter, which implies the number of instructions that can be stored. The default value selected for the parameter is 1024 (4 KiB memory), though it could be extended to up to 2^{30} (4 GiB memory).

The interface defines a 32-bit wide read port from where the instruction is output and a 32-bit address port to which the program counter is input. Figure **4.7** shows instruction memory interface.

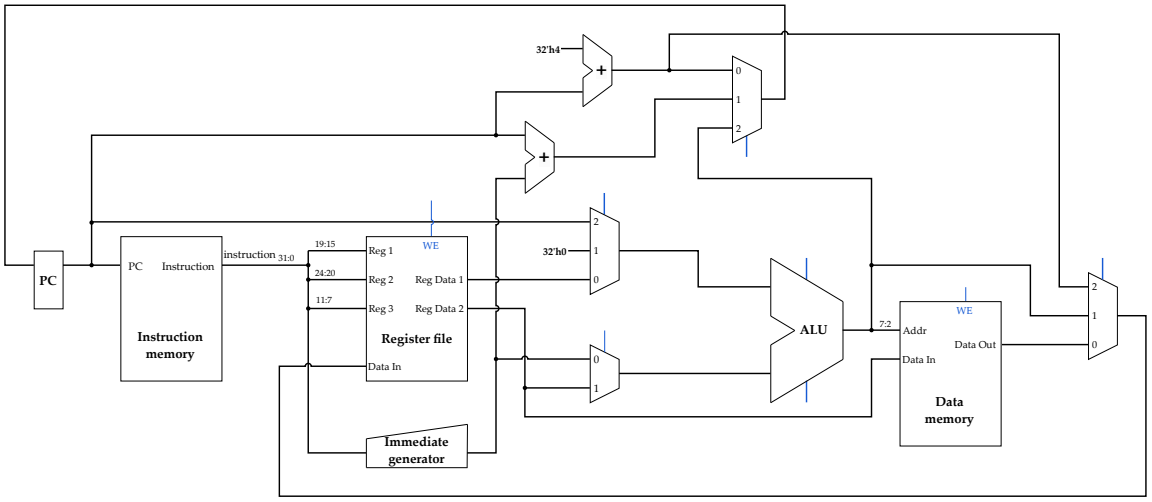
Figure 4.7: Instruction memory interface



4.2.2 Control logic

The main building blocks of the datapath have been explained in the previous sections. However, as shown in Figure **4.8**, the datapath is not only built from these blocks. Several multiplexors modify the datapath depending on the instruction being executed. Additionally, even the main modules have some control inputs that modify their behaviour. Throughout the following sections, the control modules that modify those multiplexors and control signals will be described.

Figure 4.8: Single-cycle datapath

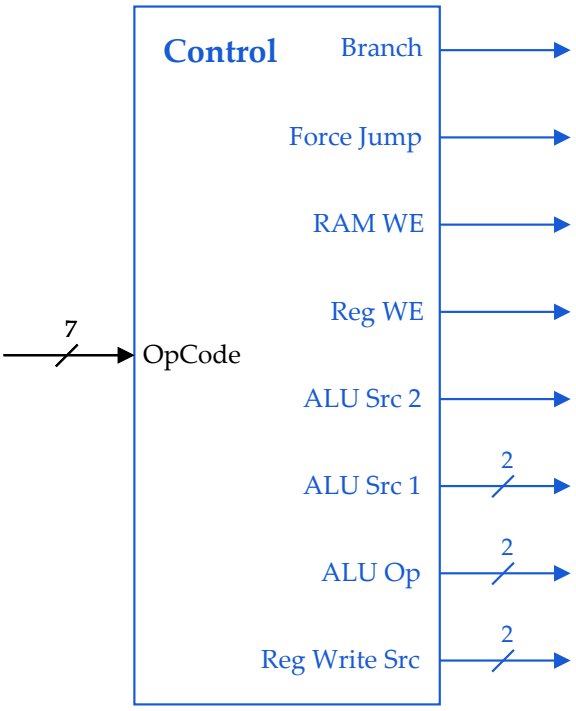


4.2.2.1 Control unit

The Control unit is the core module of the processor. It implements the logic that manages the datapath by controlling the multiplexers. Furthermore, it also governs the other control and datapath modules.

The Control unit module inputs the "OpCode" of the instruction and then outputs the control signals of the ALU source multiplexers, the write enable signals of the register file and the data memory, the ALU operation signals, "branch" and "ForceJump" signals. Figure **4.9** depicts the module interface.

Figure 4.9: Control unit interface



The "OpCode" determines the type of the instruction being executed. Table **4.9** shows the corresponding group of instructions per each "OpCode".

Table 4.9: Instructions OpCodes

OpCode	Instruction
0000011_2	lw
0010011_2	addi, slli, slti, sltiu, xori, srli, srai, ori, andi
0010111_2	auipc
0100011_2	sw
0110011_2	add, sub, sll, slt, sltu, xor, srl, sra, or, and
0110111_2	lui
1100011_2	beq, bne, blt, bge, bltu, bgeu
1100111_2	jalr
1101111_2	jal

Regarding the programming of the control unit, the module implements pure combinational logic. Therefore, the module has been coded employing an `always_comb` block with a `case` statement where the case items are the possible values of the "OpCode" and determines the output values. Table **4.10** shows the values of each output concerning the "OpCode".

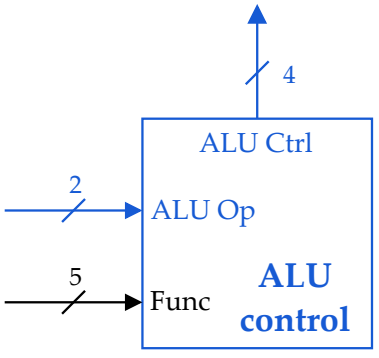
Table 4.10: Control unit outputs

OpCode	Branch	Jump	RAM WE	RF WE	ALU src 2	ALU op	ALU src 1	RF WB Src
0000011_2	0_2	0_2	0_2	1_2	0_2	10_2	00_2	00_2
0010011_2	0_2	0_2	0_2	1_2	0_2	00_2	00_2	01_2
0010111_2	0_2	0_2	0_2	1_2	0_2	10_2	10_2	01_2
0100011_2	0_2	0_2	1_2	0_2	0_2	10_2	00_2	01_2
0110011_2	0_2	0_2	0_2	1_2	1_2	00_2	00_2	01_2
0110111_2	0_2	0_2	0_2	1_2	0_2	10_2	01_2	01_2
1100011_2	1_2	0_2	0_2	0_2	1_2	01_2	00_2	01_2
1100111_2	0_2	1_2	0_2	1_2	0_2	10_2	00_2	10_2
1101111_2	0_2	1_2	0_2	1_2	0_2	10_2	00_2	10_2

4.2.2.2 ALU control unit

The ALU control unit is the module in charge of controlling the result multiplexer of the arithmetic logic unit. To perform this task, the module takes as input the value of the "ALU op" signal output from the control unit, as well as the instruction bits 30, 14-12 ("Funct3") and 5 (see Figure **4.10**).

Figure 4.10: ALU control inteface



With the defined inputs, the ALU control unit determines the "ALU op" according to Table **4.11**.

As it was explained in Section **4.2.2.1**, the "ALU op" is determined by the "OpCode". Therefore, for those groups of instructions that share the same "OpCode" and require different ALU operations, the value 0 is read at the "ALU op" input. For those groups of instructions that demand the ALU to perform a subtraction, the value 1 will be read, and finally, when an addition is compelled, the value read is 2.

Table 4.11: ALU Control logic

1st Condition ALU Op	2nd Condition {Funct3, Funct7, OpCode}	Output	Operation
00_2	$0000X_2$	0000_2	+
	00001_2	0000_2	+
	00011_2	0001_2	−
	$0010X_2$	0110_2	<<
	$010XX_2$	0101_2	SLT
	$011XX_2$	1001_2	SLTU
	$100XX_2$	0100_2	∧
	$1010X_2$	0111_2	>>
	$1011X_2$	1000_2	>>>
	$110XX_2$	0011_2	|
	$111XX_2$	0010_2	&
01_2	N/A	0001_2	−
10_2	N/A	0000_2	+

4.2.2.3 Branch logic

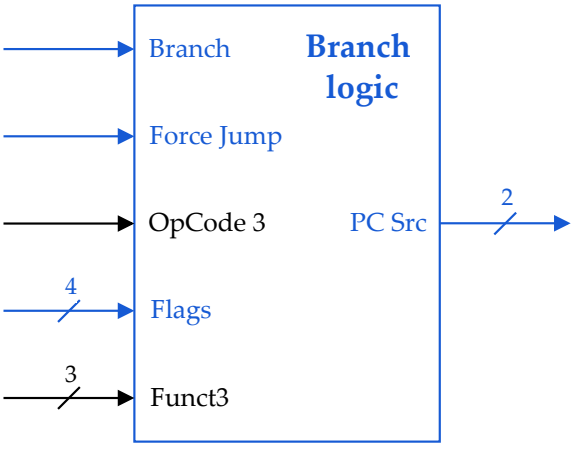
The program counter stores the value of the address where the current instruction is placed in the program memory. This section is aimed to define how the value of the PC changes, that is, how jumps between instructions are performed.

For most of the instructions, the following instruction to execute is the one that is stored next to it in the memory. It means that the address where the following instruction is stored is the current address plus four. Since this operation is so recurrent,

an adder has been explicitly added to perform this function. One of its inputs is the PC, and a constant value of four has been hardwired in the other.

For the rest of the instructions, which are seven in the RV32I ISA, the next instruction to execute is not necessarily the next one in memory. There are two types of instructions, jump and branch. The former performs a jump without any condition, and the latter only jumps if the condition (determined by the instruction) is met. A dedicated adder that adds the current PC plus the immediate encoded in the instruction has been implemented to calculate the new PC value. The main reason behind this design decision is that the ALU is performing the subtraction for the branch condition. The instruction "jalr" is an exception and employs the ALU to compute the new PC. The ALU is used because no condition has to be evaluated, and a multiplexor (with its corresponding control signal) is not needed to be implemented to allow the input in the adder of the value of "Reg Data 1".

Figure 4.11: Control unit interface



Since the program counter is calculated in different sources, a multiplexor is needed to select the one needed in each instruction. A control module, the Branch logic module, implements the logic to drive the multiplexor. This module receives the "branch" and "force jump" signals from the control module, the "Funct3" and "$OpCode_3$" from the program memory and the ALU flags (see Figure **4.11**). The logic underlying this module is depicted in Table **4.12**.

Table 4.12: Branch logic

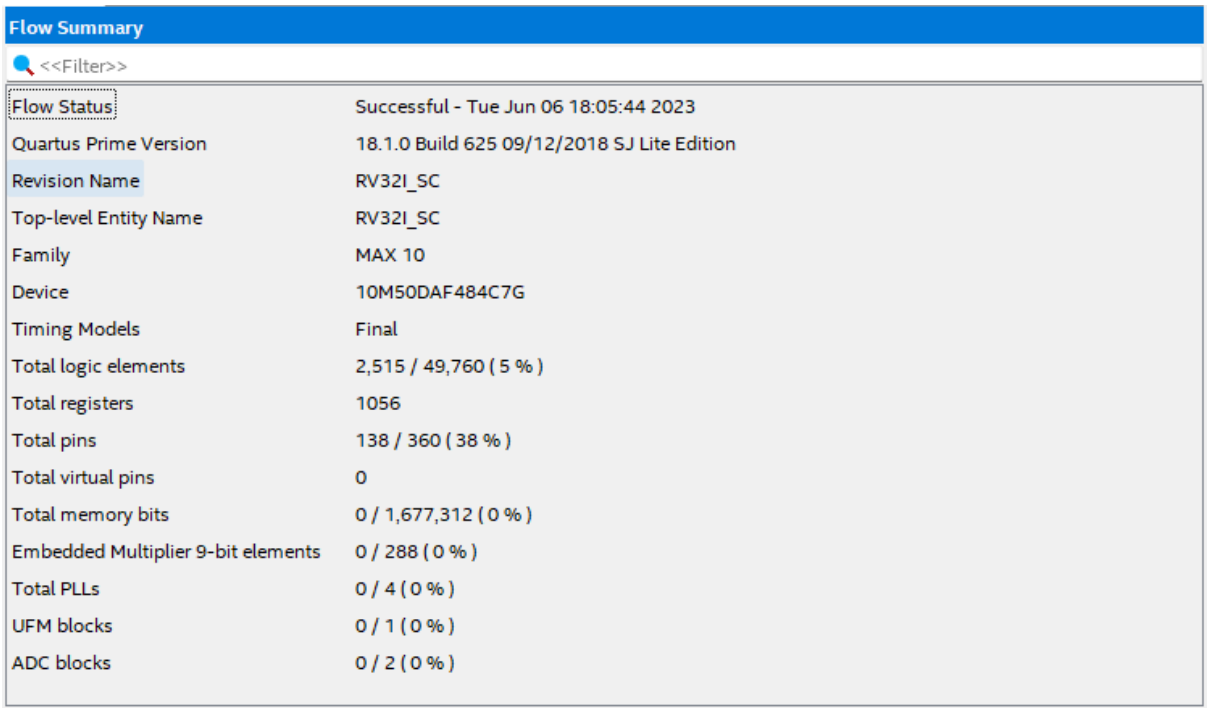
Jump signal	OpCode3	Output	Instruction
Force jump	0_2	10_2	jalr
Force jump	1_2	01_2	jal
Jump signal	**Funct3**	**Output**	**Instruction**
Branch	000_2	$\{0,\ Z\}_2$	beq
Branch	001_2	$\{0,\ \overline{Z}\}_2$	bne
Branch	100_2	$\{0,\ N \wedge V\}_2$	blt
Branch	101_2	$\{0,\ \overline{N \wedge V}\}_2$	bge
Branch	110_2	$\{0,\ \overline{C}\}_2$	bltu
Branch	111_2	$\{0,\ C\}_2$	bgeu
Jump signal	**N/A**	**Output**	**Instruction**
None	—	00_2	Other

Flags abbreviations: Z (Zero), N (Negative), C (Carry), V (Overflow)

4.2.3 Analysis & Synthesis

After synthesis, Quartus software provides a summary of the resources employed by the developed model. As can be seen in Figure **4.12**, the report shows that only 5% of the logic elements of the FPGA, meaning that additional hardware could be included to work parallel to the processor or that more processors could be synthesised in the same FPGA, resulting in a single IC with multiple processors. Quartus also provides a timing analysis, which provides a maximum operating frequency of 64.48 MHz for this model.

Figure 4.12: Analysis & Synthesis



Flow Summary

<<Filter>>

Flow Status	Successful - Tue Jun 06 18:05:44 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	RV32I_SC
Top-level Entity Name	RV32I_SC
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,515 / 49,760 (5 %)
Total registers	1056
Total pins	138 / 360 (38 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

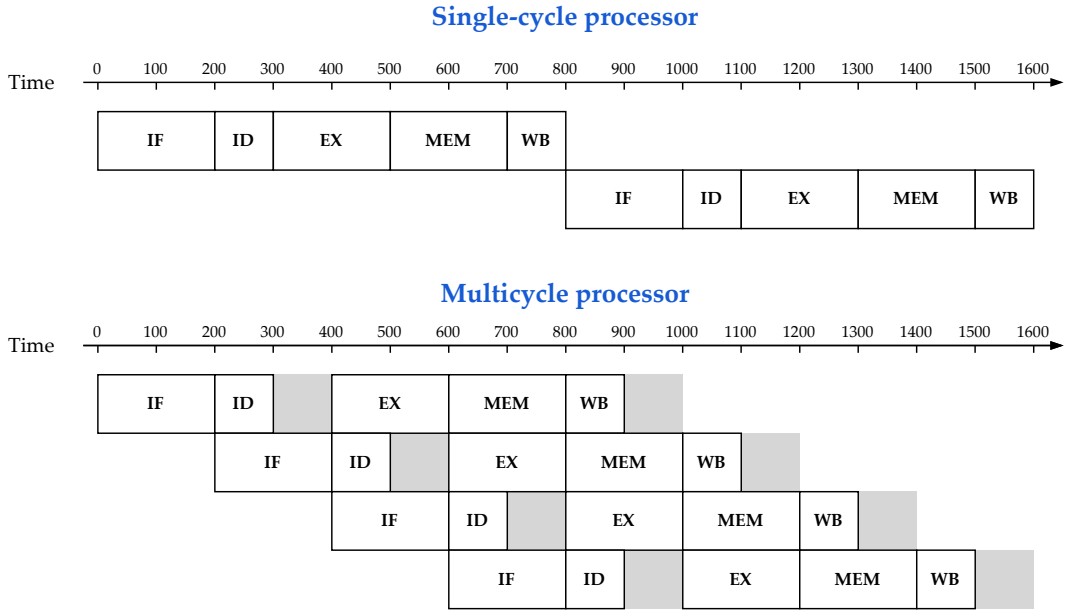
4.3 Multicycle processor

A multicycle processor executes an instruction in more than one cycle. Instructions are divided into five stages: instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB). This division allows the implementation of a pipelined structure in which more than one instruction can be inside the datapath occupying the different stages. Figure **4.13** shows the difference between a single-cycle processor and a multicycle processor.

Regarding timing considerations, the single-cycle processor clock frequency is limited by the time it takes the longest instruction to execute. However, since in the multicycle processor, the instructions have been segmented, the maximum clock frequency is determined by the slowest stage. As expected, the multicycle processor will take longer to process a single instruction. Nonetheless, since the pipeline can be executing multiple instructions (each one at a different stage), the throughput, which is the number of instructions executed per unit of time, increases because the frecuency of operation of the processor increases and in the ideal case a instruction finishes its execution every clock cycle. Figure **4.13** represents this concept; it can be shown that the throughput of the single cycle is one instruction per 800 units of time, while the

multicycle processor has a higher throughput of four instructions per 800 units of time, even though it takes 200 units of time more to execute each instruction.

Figure 4.13: Single-cycle and multicycle time diagram



Nevertheless, it is not all advantages for the multicycle. A pipelined processor entails a series of hazards that must be considered:

- **Structural hazard:** A needed resource for the execution of the instruction is not available.
- **Data hazard:** It is necessary to wait for a previous instruction to perform a read/write operation.
- **Control hazard:** The execution of an instruction depends on a previous jump instruction.

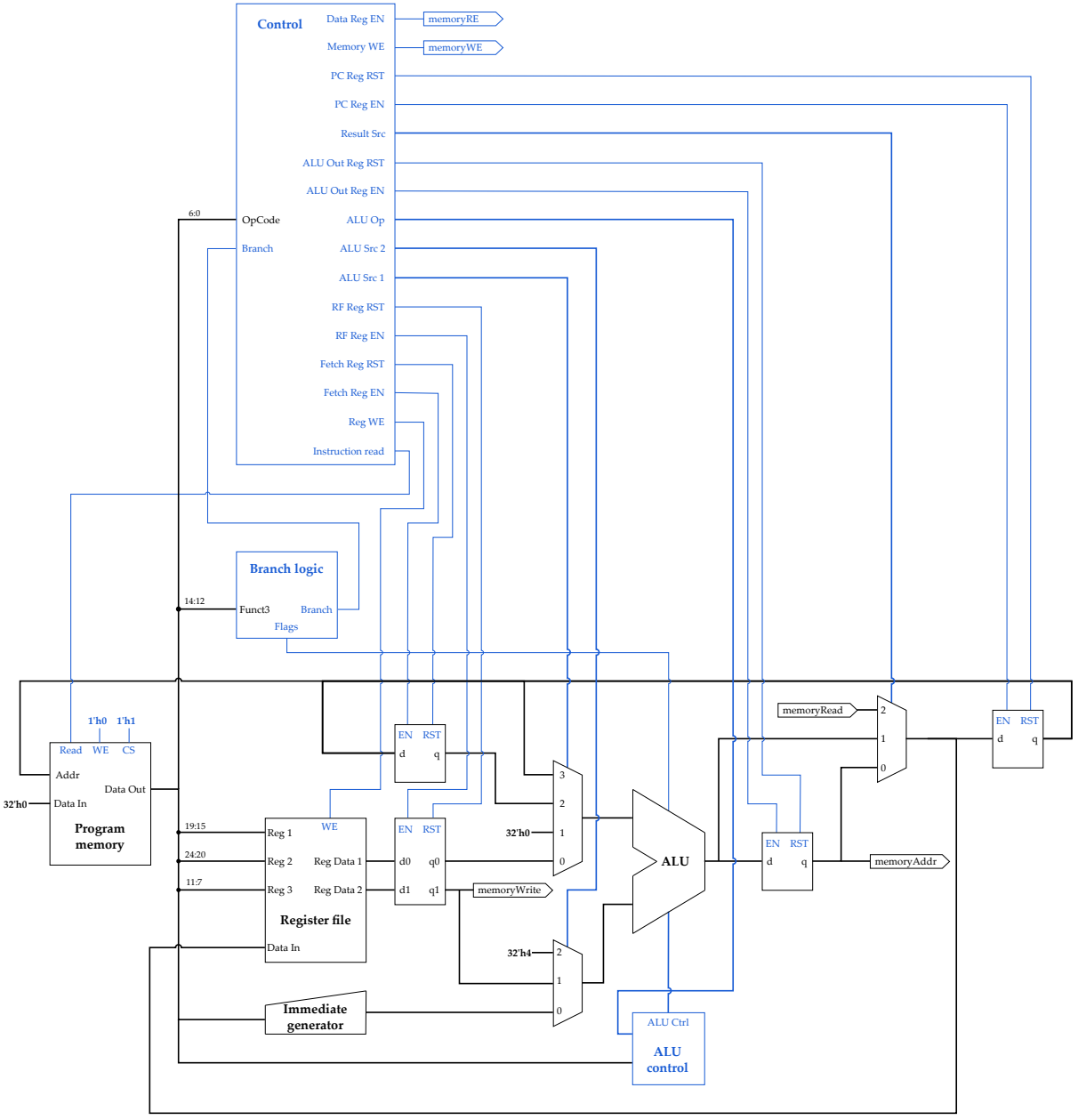
These hazards can be controlled either by software or hardware. Concerning software solutions, it is left to the programmer to manage the hazards by modifying the order of operations or adding `nop`[9] instruction whenever a conflict is detected. On the other hand, hardware solutions involve some techniques:

- Data forwarding allows the utilisation of a calculated result before it has been written in the register files.
- Jump prediction, hardware predicts if a jump will be taken or not. It can be based either on static or dynamic criteria based on the previous results.

[9] `nop` is a pseudoinstruction equivalent to `addi x0, x0, 0`

The solution applied in the multicycle processor is not to pipeline the instructions, therefore avoiding these hazards but maintaining the advantages of a multicycle core, such as using external memories, since synchronous reading is possible in this configuration or using a single memory. Figure **4.14** shows the scheme of the processor, the datapath is drawn in black and the control in blue.

Figure 4.14: Multicycle processor



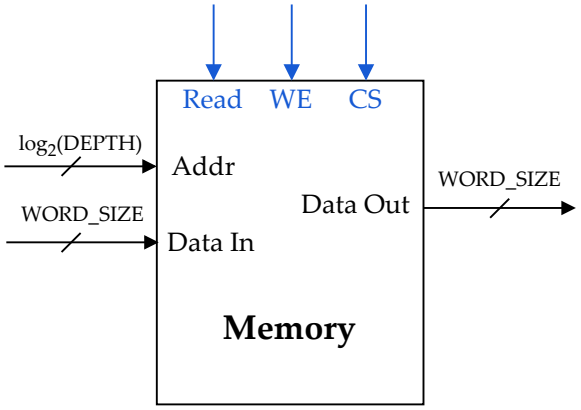
4.3.1 Datapath

The datapath is not significantly modified compared to the single-cycle versions, although it is not free of modifications. These modifications are mainly due to processor segmentation. There are two noteworthy changes. The first of them is the segmentation itself, meaning that registers have been added to make the execution phases independent from each other.

4.3.1.1 Instruction and Data memory

The second modification is the refactoring of the memories to support synchronous reading. This modification allows to use the embedded memories in the FPGA. A new memory has been developed to be highly configurable and support either RAM or ROM memories (its interface is shown in Figure **4.15**). The module has four parameters that allow modifying the memory size (`WORD_SIZE` and `SIZE`) and deciding whether to initialise it with the contents of a file or not (`INITIALIZE` and `FILE`).

Figure 4.15: Memory interface



The RAM or ROM configuration is generated by configuring the module inputs according to the values in Table **4.13**.

Table 4.13: Memory module configuration

Parameters	RAM	ROM
WORD_SIZE	Application dependant	Application dependent
DEPTH	Application dependant	Application dependent
INITIALIZE	0	1
FILE	""	Memory file
Port	**RAM**	**ROM**
Read	Connect to control unit	Connect to control unit
Write enable (WE)	Connect to control unit	0
Chip Select (CS)	Application dependant	Application dependant
Data In	Memory input	X

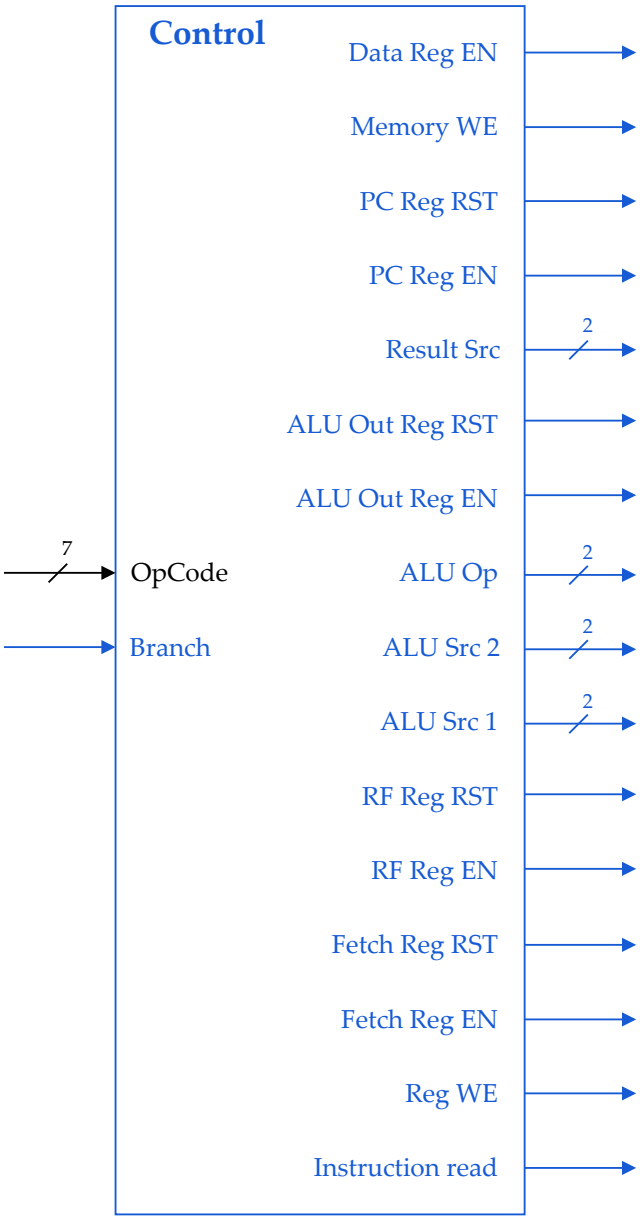
4.3.2 Control logic

In contrast to the datapath, the control logic must be refactored to support the processor segmentation. The appearance of different stages that must be coordinated and the fact that every type of instruction requires a different behaviour of the datapath forces the control logic to transition from basic combinational logic to sequential logic and the implementation of a state machine to govern the processor behaviour. Implementing a state machine permits optimising hardware utilisation and reducing circuitry, further explanation in Section **4.3.2.2**.

4.3.2.1 Control unit

As explained in the previous section, the segmentation of the processor forces the control unit to transition from a decoder to a state machine. The Control unit module uses the "OpCode" of the instruction and the "branch" signal generated by the branch logic module to determine the following state (Figure **4.16** shows the control unit module interface).

Figure 4.16: Memory interface



The datapath is divided into five stages: instruction fetch, instruction decode, execution, memory access, and write back. The first two stages are common for every instruction, meaning that the outputs generated by the control unit module shall be the same. Each type of instruction requires a different set of outputs for the rest of the stages. The state machine diagram is shown in Figure **4.17**.

Figure 4.17: Control state machine diagram

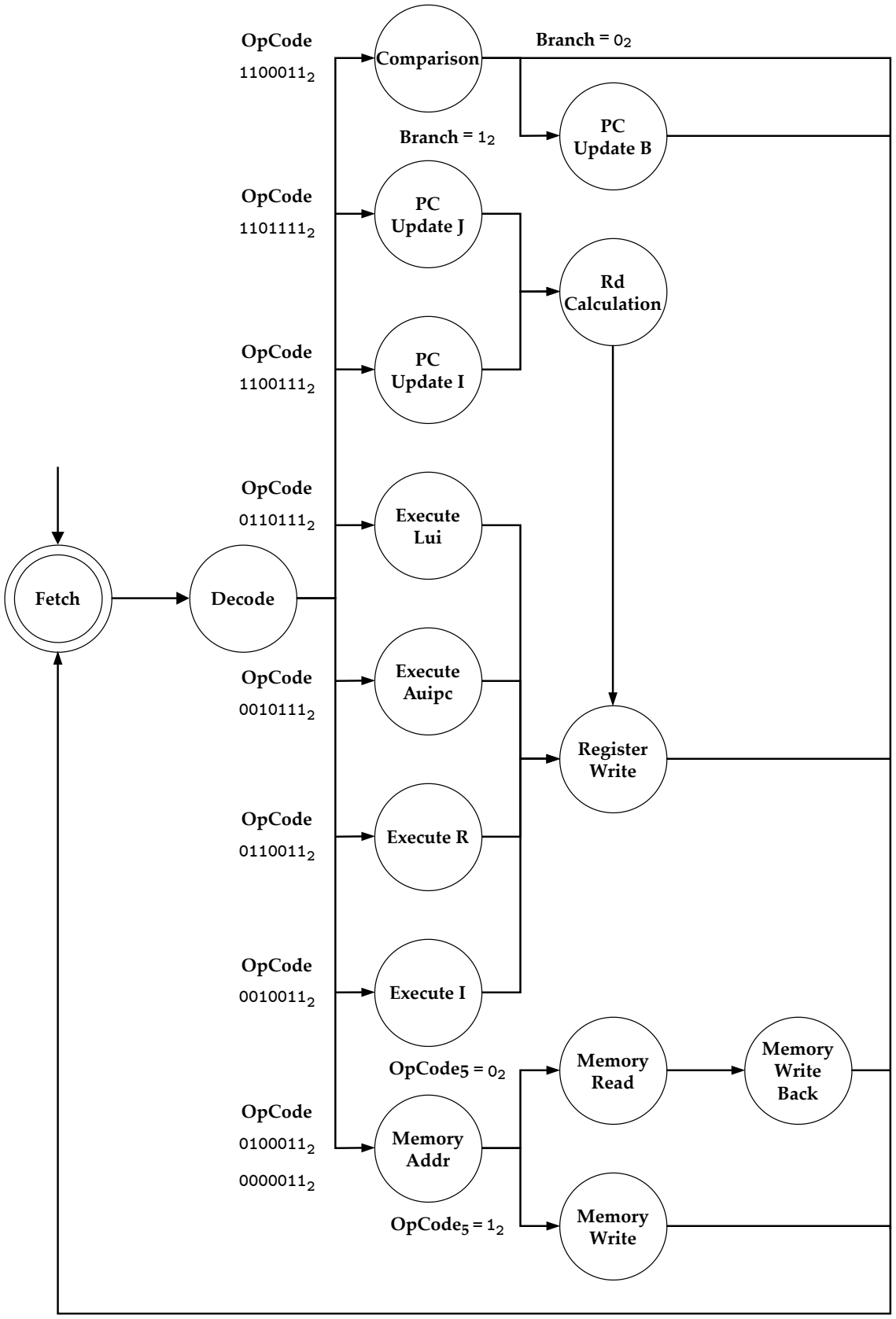


Table 4.14: Control unit module outputs

State	PCRegEN	FetchRegEN	InstructionRead	RFRegEN	ALUOutRegEN	DataRegEN	PCRegRST	FetchRegRST	RFRegRST	ALUOutRegRST	ALUIn1Src	ALUIn2Src	ResultSrc	memWE	RegWE	ALUOp
Fetch	1_2	1_2	1_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	11_2	10_2	01_2	0_2	0_2	10_2
Decode	0_2	0_2	0_2	1_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	0_2	0_2	00_2
Memory Addr	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	0_2	0_2	10_2
Memory Read	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	0_2	0_2	00_2
Memory WB	0_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	10_2	0_2	1_2	00_2
Memory Write	0_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	1_2	0_2	00_2
Execute R	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	00_2	01_2	00_2	0_2	0_2	00_2
Execute I	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	0_2	0_2	00_2
Execute Auipc	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	10_2	00_2	00_2	0_2	0_2	10_2
Execute Lui	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	01_2	00_2	00_2	0_2	0_2	10_2
PC Update I	1_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	01_2	0_2	0_2	10_2
PC Update J	1_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	10_2	00_2	01_2	0_2	0_2	10_2
Rd Calculation	0_2	0_2	0_2	0_2	1_2	0_2	1_2	1_2	1_2	1_2	10_2	10_2	00_2	0_2	0_2	10_2
Comparison	0_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	01_2	00_2	0_2	0_2	01_2
PC Update B	1_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	10_2	00_2	01_2	0_2	0_2	10_2
Register Write	0_2	0_2	0_2	0_2	0_2	0_2	1_2	1_2	1_2	1_2	00_2	00_2	00_2	0_2	1_2	00_2

The implementation of the state machine is a Moore state machine. Thus, the outputs only depend on the current state. Table **4.14** shows the output value for each state.

Regarding the implementation in SystemVerilog, a new `enum` datatype (`statetype`) has been defined, setting as values every state of the state machine. Two variables of this kind have been instantiated, `state` and `nextstate`. A register will store the `state` value, which will be updated with the value of `nextstate` every clock cycle. Moreover, the value of `nextstate` is obtained using combinational logic, which inputs the current state, the "OpCode", and the "Branch" signal. Lastly, the output values are obtained through combinational logic depending on the `state`, as required by a Moore state machine. Code Snippet **4.2** shows the basic scheme of the SystemVerilog implementation.

Code snippet 4.2: State machine implementation

```
// State datatype
typedef enum logic [3:0] {
    <STATES>
} statetype;
statetype state, nextstate;

// State register
always_ff @(posedge clk or negedge arst) begin
    if(~arst) state <= Fetch;
    else if(en) state <= nextstate;
end

// Next state logic
always_comb begin
    case(state)
        <STATE>: nextstate = <NEXTSTATE>;
        default: nextstate = Fetch;
    endcase
end

// Output logic
always_comb begin
    case(state)
        <STATE>: begin
            <OUTPUTS>;
        end
    endcase
end
```

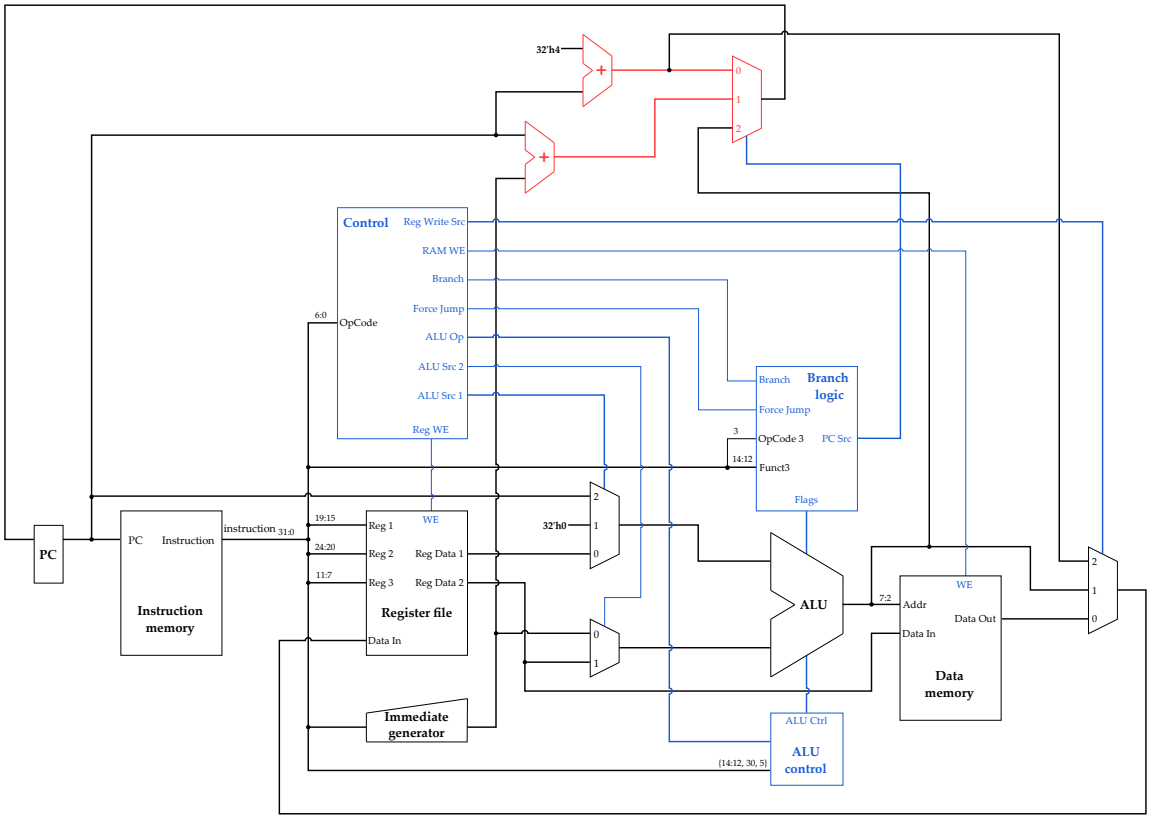
4.3.2.2 Branch logic

The segmentation of the processor and the approach taken of not pipelining instructions result in only one portion of the hardware being utilised in each clock cycle. Despite the fact that it might sound like a waste of resources, it means that it can be employed in other stages of the instruction. This characteristic has been implemented in the processor to reduce the hardware employed in jump and branch instructions.

Jump and branch instructions require the use of two mathematical operations. Branch instructions, in particular, perform a comparison between two registers (a subtraction) and an addition, which can either be PC + 4 or PC + immediate, to calculate the new PC. On the other side, jump instructions must perform two additions, the following value of PC (register + immediate or PC + immediate) and PC + 4, which will be stored in the Register file.

In the single-cycle implementation, it was impossible to use the ALU for two different operations in a single instruction. As a consequence, two 32-bit adders had to be included. In the multicycle implementation, is it possible to use the ALU twice during the execution of an instruction just by designing the control state machine to perform those operations. Therefore, both adders and the multiplexor and its control logic have been removed from the single-cycle implementation. Figure **4.18** shows in red the components that have been removed.

Figure 4.18: Removed components from single-cycle implementation



The branch logic module (see Figure **4.19**), as a consequence, has reduced its functionality. Its only function is to analyse the ALU flags when a comparison is made in a branch instruction and indicate to the state machine whether a jump must be performed. The logic implemented in the Branch logic is depicted in Table **4.15**.

Figure 4.19: Branch logic interface

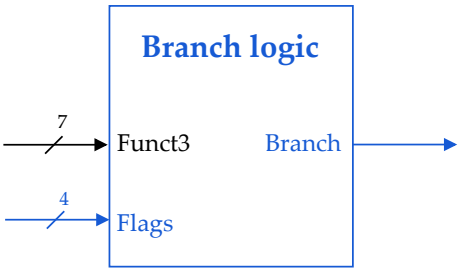


Table 4.15: Branch logic

OpCode3	Output	Instruction
000_2	$\{0,\ Z\}_2$	beq
001_2	$\{0,\ \overline{Z}\}_2$	ben
100_2	$\{0,\ N \wedge V\}_2$	blt
101_2	$\{0,\ \overline{N \wedge V}\}_2$	bge
110_2	$\{0,\ \overline{C}\}_2$	bltu
111_2	$\{0,\ C\}_2$	bgeu

Flags abbreviations: Z (Zero), N (Negative), C (Carry), V (Overflow)

4.3.3 Memory bus & Peripherals

A microprocessor is an integrated circuit incorporating a CPU (a computing element). To operate, a microprocessor requires additional hardware, such as memories and peripherals, to handle input/output endeavours. On the other hand, microcontrollers are also integrated circuits that include on-chip memories and peripherals instead. In previous sections, the development of a processor has been described. In the upcoming sections, the implementation of peripherals and their connection with the CPU will be explained, moving from the implementation of a processor to a microcontroller.

In order to connect the peripherals to the central processing unit, a memory bus has been designed inspired by the Intel Avalon Memory-Mapped Interface (Avalon-MM), which is an address-based read/write interface. The resultant memory bus employs a master/slave architecture where the CPU acquires the role of master and the data memory and peripherals the role of slave. An interface has been designed for each role (described in Table **4.16**). That means that every peripheral must implement the same interface to be connected to the bus.

Table 4.16: Memory bus interfaces

Master interface

Signal	Width	In/Out	Description
Read	32	In	Master data input
Write	32	Out	Master data output
Addr	32	Out	Read/Write address
WE	1	Out	Slave write enable
RE	1	Out	Slave read request

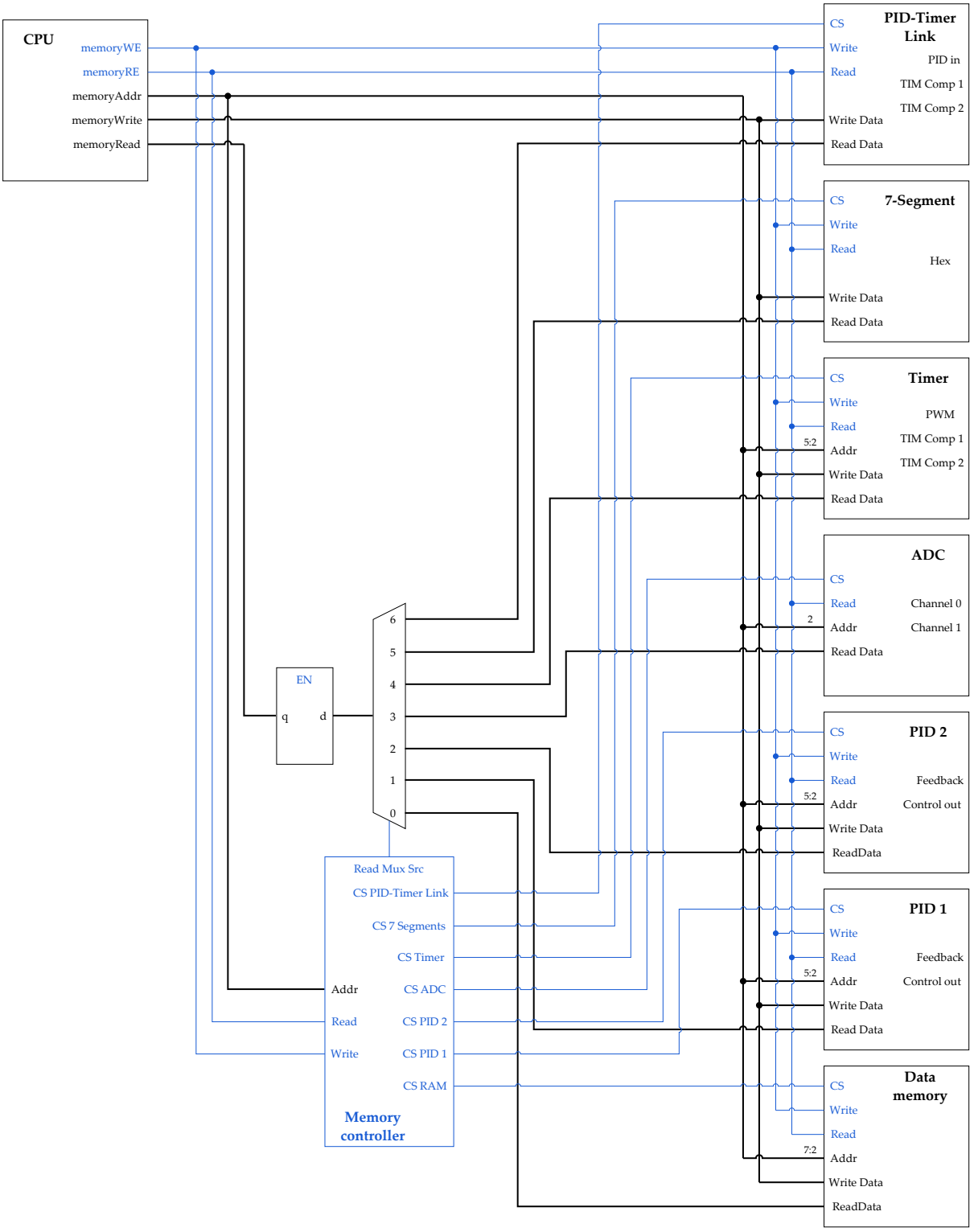
Slave interface

Signal	Width	In/Out	Description
DataRead	32	Out	Slave data output
DataWrite	32	In	Slave data input
Addr	0-32	In	Offset in the slave memory space
CS	1	In	Chip Select - if not active the slave ignores all signals
Read	1	In	Read request
Write	1	In	Write enable

The memory bus consists of "Address" and "Write" busses, individual "Read" lines and enable signals. A complete scheme of the memory bus can be observed in Figure **4.20**. The "Address" bus is connected to every component connected to the bus. The peripherals connected to the memory bus are assigned a range of addresses to identify their internal peripherals (memory mapping). Even though each peripheral is assigned a specific range of addresses, their address port is dimensioned to the minimum size to map their internal peripherals; this port is used as an offset to their base address[10]. Since the peripherals cannot identify whether the address on the bus belongs to their range, a "Chip select" signal is assigned to each peripheral. It will be active whenever the address on the bus belongs to the particular peripheral. "Chip select" signals are managed by the memory controller (further explained in Section **4.3.3.1**).

[10]The base address is the lowest address assigned to a specific peripheral.

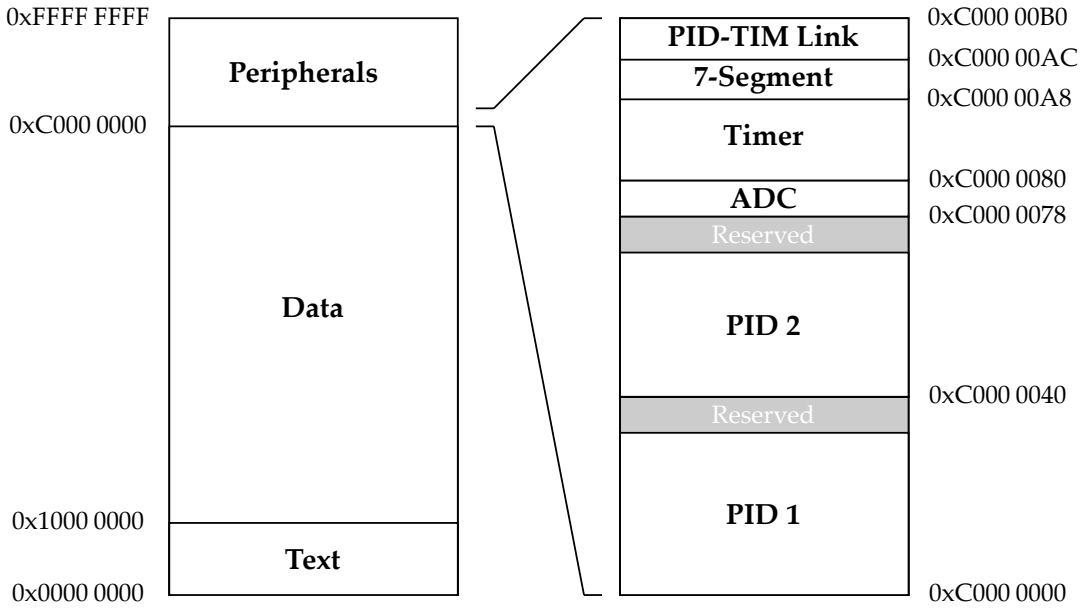
Figure 4.20: Memory bus



The “Write” bus is connected to the CPU and those peripherals that allow writing operations. For a write operation to occur, the CPU must send the data through the “Write” bus and activate the “Write enable” signal. The “Chip select” signal must be active for the aimed peripheral. Reading topology differs slightly from writing. Since there is no “Read” bus, the individual signals are multiplexed. Read operation requires the “Read” signal and the “Chip select” signal to be active.

The memory map diagram (Figure **4.21**) comprises the address ranges assigned to each peripheral[11] and memory. For this project, it has been implemented two PID controllers, a timer, an ADC, a six-digit seven-segments decoder and PID-Timer Link peripheral. These modules will be further explained in the upcoming sections.

Figure 4.21: Memory map

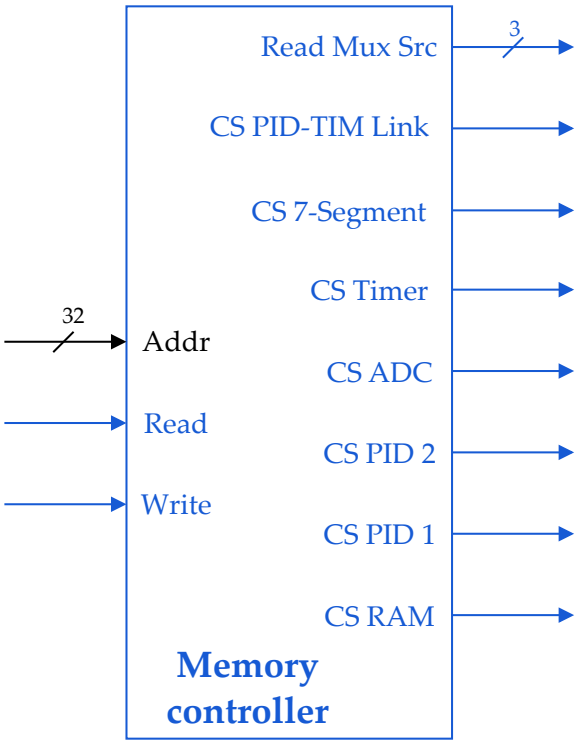


4.3.3.1 Memory Controller

The Memory controller is the module that decodes the addresses to activate the correct “Chip select”. It also uses the addresses to control the multiplexer of the “Read” signals. The module takes as input the address and the “memoryWE” and “memoryRE” signals and generates the previously mentioned signal (see Figure **4.22**).

[11] A detailed memory map and peripheral registers list can be found at Appendix **A**

Figure 4.22: Memory controller interface



The memory controller accepts as parameters the base address of each peripheral and the data memory, allowing a rapid modification of the reserved memory for each peripheral if necessary. The inner logic compares the address received with the value of the parameters to generate the outputs. To activate a ”Chip select”, either the ”memoryWE” or the ”memoryRE” signals must be active. Table **4.17** indicates the base addresses of each of the peripherals.

Table 4.17: Base addresses

Parameter	Base address	Peripheral
RAM_ADDR	0x10000000	Data memory
PID1_ADDR	0xC0000000	PID Controller 1
PID2_ADDR	0xC0000040	PID Controller 2
ADC_ADDR	0xC0000078	ADC
TIM_ADDR	0xC0000080	Timer
H7S_ADDR	0xC00000A8	7-Segment decoder
PTL_ADDR	0xC00000AC	PID-Timer Link

4.3.3.2 PID-Timer Link

The PID-Timer Link peripheral bypasses the output of the PID peripheral to the timer peripheral. As it will be explained in Section **4.4**, the electromagnets are driven by an H-bridge controlled by two PWM signals and their complementaries. These electromagnets require both positive and negative currents flowing through them. The way to achieve the change in the sign of the current is by changing the branch of the H-bridge that is commuting. However, the PID control outputs positive and negative values but cannot generate the duty cycle value for each branch. The PID-Timer Link interprets the output sign and generates the duty cycle values for each branch (a functional diagram is shown in Figure **4.23**). It also incorporates a shifter (see Table **4.18**), that can be programmed from the CPU, to adequate the control output value to fit the duty cycle range. Figure **4.24** shows the interface of the module.

Figure 4.23: PID-Timer Link interface

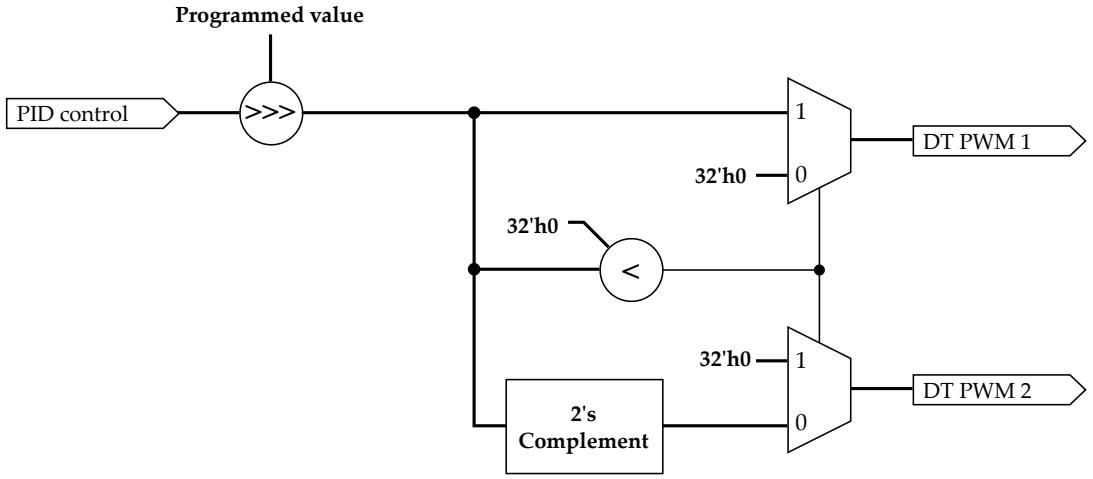


Figure 4.24: PID-Timer Link interface

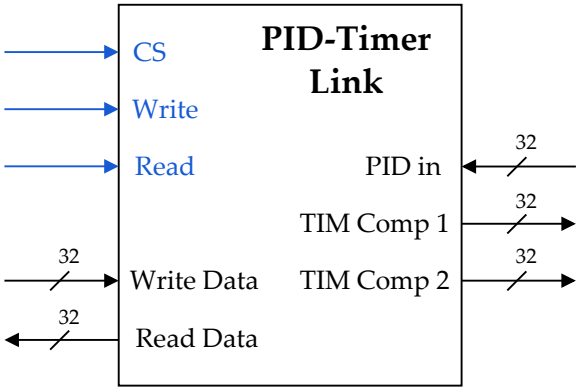


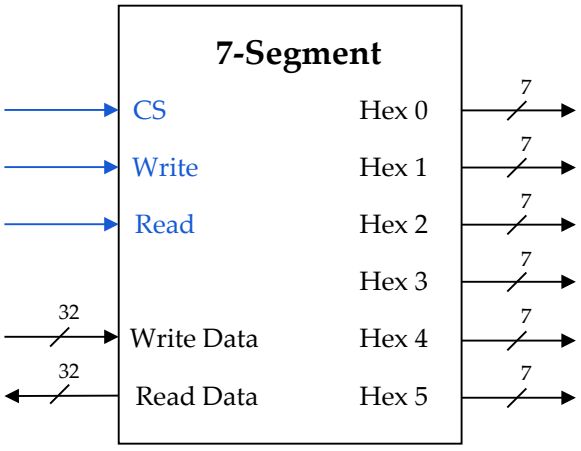
Table 4.18: PID-Timer link register description

Memory offset	Read/ Write	Name	Function
00_{16}	R/W	Shift	Input shift value

4.3.3.3 Seven-segments decoder

The DE10-Lite evaluation board incorporates a six-digit seven-segment display. The display can be used to debug or as an output of the microcontroller. The aim of this peripheral is to act as a link between the processor and the display. The interface of the module consists of the slave interface of the memory bus and six seven-bit wide ports that are connected to the display (see Figure **4.25**).

Figure 4.25: Memory controller interface



For this application, a single-digit decoder has been designed and then instantiated six times inside the peripheral. The single digit receives a hexadecimal digit (4 bits) and generates the decoded 7-bit wide output. It has also been developed to adapt to different displays (common cathode or common anode) by means of a parameter that depends on its value inverts or not the output signal.

The seven-segment decoder implements a single read/write register to store the value to be displayed. The register has been mapped to the memory direction `0xC00000A4` (Table **4.19**).

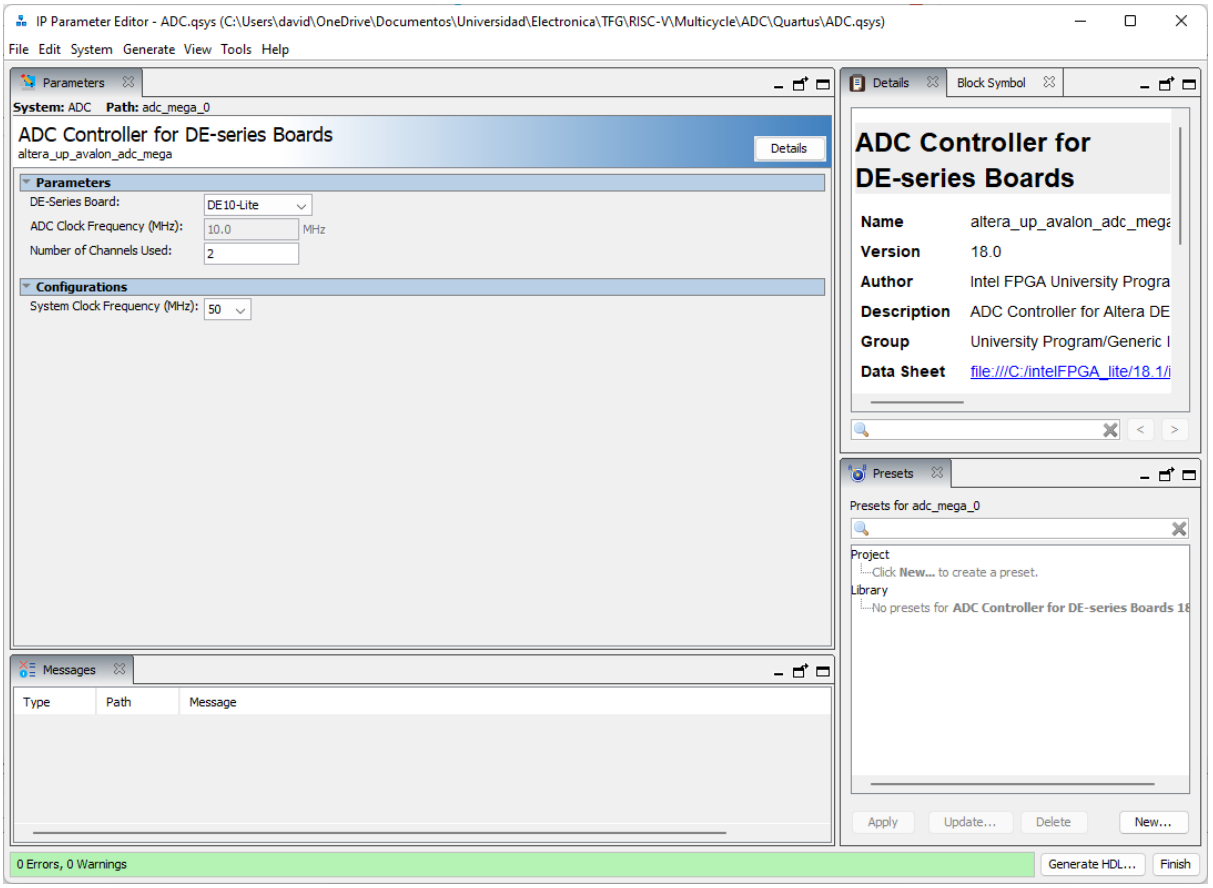
Table 4.19: Seven-segment decoder register description

Memory offset	Read/ Write	Name	Function
00_{16}	R/W	Value	Number to be displayed

4.3.3.4 Analog to Digital Converter

The MAX10M50DAF484C7G FPGA features a dual ADC supporting 18 channels (one dedicated pin and eight dual-function pins per ADC). The University Program ADC Controller for DE-series Boards IP block from Intel FPGA has been employed to implement the ADC peripheral. The IP block has been configured using the IP Parameter Editor tool (it can be accessed through Quartus software).

Figure 4.26: ADC - IP Parameter Editor



The parameters introduced are the following:

- DE-Series Board: DE10-Lite
- Number of Channels Used: 2
- System Clock Frequency (MHz): 50

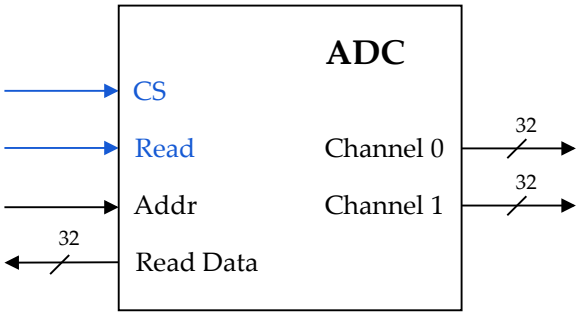
Figure **4.26** shows a screenshot from the IP Parameter Editor.

A wrapper has been designed to fit the IP block generated to the microcontroller. The wrapper is developed around the ADC block. It implements two 32-bit registers that are updated with the values of the ADC channels. Then, the wrapper implements a read-only slave interface to connect the peripheral to the memory bus. The ADC values are available to the processor at the addresses `0xC0000078` and `0xC000007C` (Table **4.20**), channel one and channel two, respectively. Additionally, both channels have been bypassed as outputs of the module (as shown in Figure **4.27**) to be used as feedback by the PID modules.

Table 4.20: ADC register description

Memory offset	Read/ Write	Name	Function
00_{16}	R	Channel 1	ADC Channel 1 reading
04_{16}	R	Channel 2	ADC Channel 2 reading

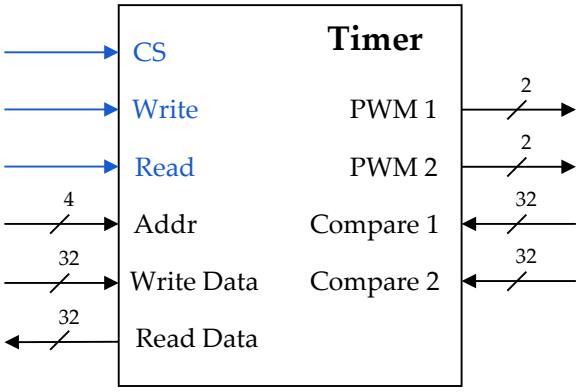
Figure 4.27: Memory controller interface



4.3.3.5 Timer

Timer peripherals are circuits intended to perform time-related tasks. Specifically, the timer designed (Figure **4.28** shows the timer interface) implements two functions, time tracking, the peripheral activates a flag signal when the programmed time has elapsed, and PWM generation.

Figure 4.28: Timer interface

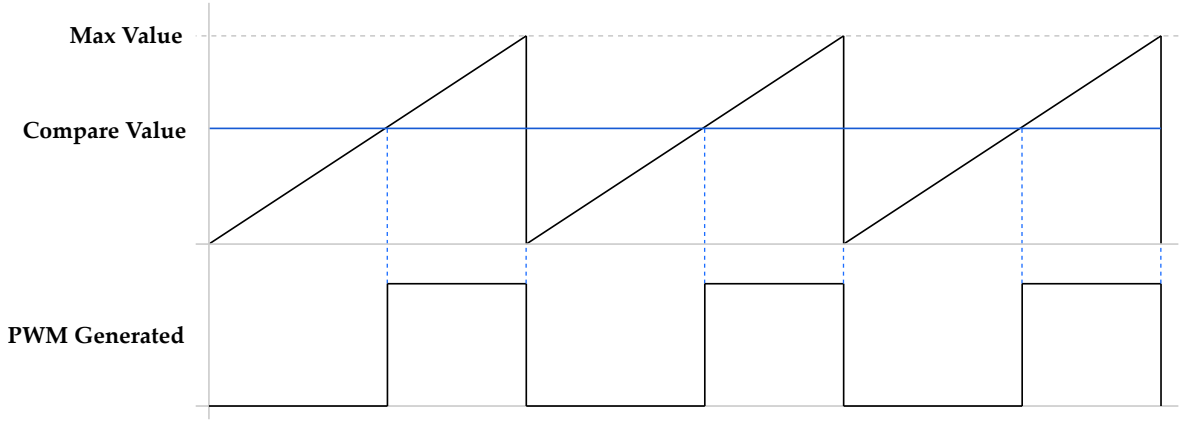


The timer peripheral has been designed by developing smaller modules that perform a single task and are then interconnected to generate the peripheral. The modules are a counter, a prescaler, and two PWM generators. The counter is the primary element of the timer. It is based on a 32-bit register that increments its value by one each clock cycle if the enable signal is active. This register must be accessible from the CPU, allowing reading and writing operations; therefore, it is mapped in memory, as it will be explained further in this section. Writing on the counter register allows resetting, modifying or setting an initial value to the timer. On the other side, reading the counter allows knowing the current value of the count. An enable signal is also assigned to a register to be accessible from the microcontroller. When the counter value arrives at its maximum value, a flag sets. The maximum value can be settable from the CPU.

The prescaler is a module that permits reducing the counting frequency by an integer division. It divides the clock signal by a configurable value, the division factor. The module is based on a counter that activates an output when it arrives at the programmed value. The prescaler is set to actuate on the enable signal of the counter. Its output is anded with the enable counter so that the timer enable signal is only activated at the end of each prescaler count.

Lastly, the PWM generator is a module able to generate a PWM and its complementary modifying their duty cycle using the counter value. The module inputs the count value, the maximum value and a programmable value or the bypassed PID output, which will actuate as the duty cycle. While the value of the counter remains below the comparison value, the output will remain in LOW state. However, if the counter exceeds the comparison value, the output changes to HIGH state (see Figure **4.29**). The PWM can be configured to output the complemented signal of the PWM with a programmable dead-time. A 2-bit number must be input to the PWM generator where the least significant bit enables the PWM signal, and the most significant bit enables the PWMN (complementary PWM). To enable PWMN, the PWM signal must be enabled. The PWM generator takes an 8-bit integer to configure the dead-time value. A progressive formula, designed by ST Microelectronics[12], has been applied to calculate the dead-time values to obtain a larger range without losing precision for short dead-times. The formula is shown in Table **4.21**. The values obtained in the formula refer to the number of clock cycles while the dead-time is active. To obtain the dead-time they shall be multiplied by the clock period.

Figure 4.29: Timer interface



As mentioned, the peripheral contains several parameters that shall be written or read by the CPU to configure the peripheral or to input information from it. Each parameter has been assigned a register that is mapped in the memory. Table **4.22** indicates the address assigned to each peripheral and its function.

[12]The formula is defined in the application note AN4043 - ST Microcontroelectronics (2016)

Table 4.21: Dead-time calculation

Configuration value (DTV)	Formula	Clock cycles range
0 - 127	DTV	0 - 127
128 - 191	$(64 + DTV_{5\text{-}0}) \cdot 2$	128 -254
192 - 223	$(32 + DTV_{4\text{-}0}) \cdot 8$	256 - 504
224 - 255	$(32 + DTV_{4\text{-}0}) \cdot 16$	512 - 1008

Table 4.22: Timer registers description

Memory offset	Read/ Write	Name	Function
00_{16}	R/W	Count	Stores counter value
04_{16}	R/W	ARR	Count maximum value
08_{16}	R/W	Start	Counter enable — '1' Enable, '0' Disable
$0C_{16}$	R/W	IRQ	Sets when maximum value is reached
10_{16}	R/W	Prescaler	Prescaler value
14_{16}	R/W	Dead-time	Dead-time configuration value
18_{16}	R/W	Compare 1	PWM generator 1 compare value
$1C_{16}$	R/W	Compare 2	PWM generator 2 compare value
20_{16}	R/W	Output enable	Output enable — '1' Enable, '0' Disable Bit [0]: PWM 1 Bit [1]: PWMN 1 Bit [2]: PWM 2 Bit [3]: PWMN 2
24_{16}	R/W	Bypass	Activates duty cycle bypass

4.3.3.6 PID controller

As the main objective, and requirement of the project, the development of a control-oriented microcontroller demands the implementation of a PID controller peripheral. The aim of this peripheral is to relieve the computational cost of performing this operation by software. The hardware implementation of the controller allows reducing the computational cost to just the parameter setting operations.

A proportional-integral-derivative controller is a closed-loop control algorithm whose aim is to reduce the error (the difference between the setpoint and the actual state of the controlled system). It employs three terms:

1. The proportional term (K_p) generates a control action proportional to the current error of the system.
2. The integral term ($1/T_i$) considers the past of the signal integrating the accumulated error, reducing the steady-state error.
3. The derivative term (T_d) is intended to anticipate future trends by actuating over the variation of error.

By adding the three terms, it is obtained the control action. The equation of the PID controller results as follows:

$$u(t) = K_p \cdot e(t) + \frac{1}{T_i} \int_0^t e(\tau)\, d\tau + T_d \frac{de(t)}{dt} \tag{1}$$

Where:

$u(t)$ is the control action

$e(t)$ is the error

K_p is the proportional gain

T_i is the integral time

T_d is the derivative time

t is the time

τ is the variable of integration

The previous equation represents the PID controller equation for continuous time. However, digital systems operate under discrete time conditions, meaning that the presented equation must be adapted to discrete time. For this application, the integral

has been approximated by employing the rectangular rule, while the differentiator employed is the first-difference differentiator. The obtained difference equation is the following:

$$u(k) = K_p \cdot e(k) + \frac{T}{T_i}\sum_{i=0}^{k-1} e(i) + \frac{T_d}{T}(e(k) - e(k-1)) \tag{2}$$

A drawback to directly implementing this difference equation is that in the summation, every error value is added from $e(0)$ to $e(k)$, meaning that they must be stored. The issue can be solved by implementing the recursive PID algorithm, obtained by performing $u(k) - u(k-1)$. Therefore, the equation for $u(k)$ is also needed and is as follows.

$$u(k-1) = K_p \cdot e(k-1) + \frac{T}{T_i}\sum_{i=0}^{k-2} e(i) + \frac{T_d}{T}(e(k-1) - e(k-2)) \tag{3}$$

Subtracting Equations **2** & **3** and grouping terms, the recursive PID algorithm is obtained.

$$\begin{aligned} u(k) - u(k-1) = K_p \cdot e(k) - K_p \cdot e(k-1) + \frac{T}{T_i}\sum_{i=0}^{k-1} e(i) - \frac{T}{T_i}\sum_{i=0}^{k-2} e(i) + \\ + \frac{T_d}{T}(e(k) - e(k-1)) - \frac{T_d}{T}(e(k-1) - e(k-2)) \end{aligned} \tag{4}$$

$$\begin{aligned} u(k) - u(k-1) = K_p \cdot (e(k) - e(k-1)) + \frac{T}{T_i}e(k-1) + \\ \frac{T_d}{T}(e(k) - 2e(k-1) + e(k-2)) \end{aligned} \tag{5}$$

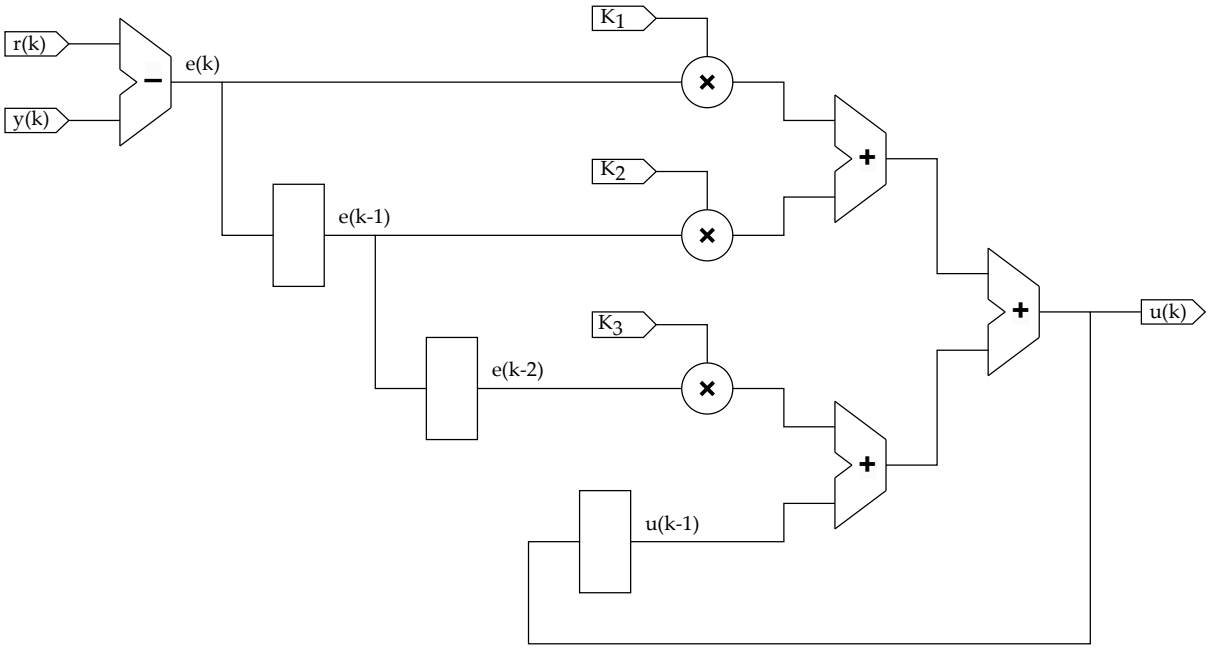
$$u(k) = u(k-1) + e(k) \cdot \left(K_p + \frac{T_d}{T}\right) + e(k-1) \cdot \left(-K_p + \frac{T}{T_i} - 2\frac{T_d}{T}\right) + e(k-2)\frac{T_d}{T} \tag{6}$$

$$u(k) = u(k-1) + e(k) \cdot K_1 + e(k-1) \cdot K_2 + e(k-2) \cdot K_3 \tag{7}$$

$$\text{Where:} \qquad K_1 = K_p + \frac{T_d}{T} \qquad K_2 = -K_p + \frac{T}{T_i} - 2\frac{T_d}{T} \qquad K_3 = \frac{T_d}{T} \tag{8}$$

Once obtained the difference equation has been obtained, the electronic circuit can be implemented. The electronic diagram is shown in Figure **4.30**.

Figure 4.30: PID circuit diagram



A multiplier module has to be developed to perform the PID controller implementation. As in the case of the ADC peripheral, the LPM_MULT Intel FPGA IP block has been employed. It has been configured to multiply (with sign) 'dataa' and 'datab', which are 32-bit inputs and generates a 64-bit result. Configuration can be shown in Appendix **B**.

The designed controller cannot operate as a peripheral by itself; a wrapper containing the memory bus interface and configuration registers must be added (registers descriptions are shown in Table **4.23**). Additionally, clock prescaling, saturation, and feedback bypass features have been added to work alongside the controller and their configuration registers are also contained in the wrapper logic.

The prescaler module of the PID controller peripheral is an instance of the prescaler designed for the timer peripheral. This module, as in the timer, actuates over the enable signal of the controller. The aim of this module is to configure the step time of the PID controller.

The saturation module has been designed in order to protect the system and the electronics from output values that fall outside the maximum or minimum rating of

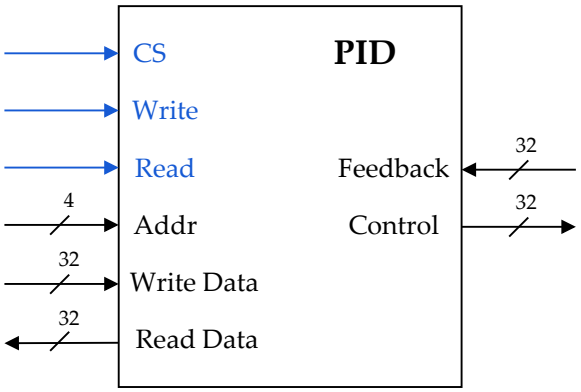
Table 4.23: PID controller registers description

Memory offset	Read/ Write	Name	Function
00_{16}	R/W	Reference	Reference value — R(k)
04_{16}	R/W	K_1	PID constant K_1
08_{16}	R/W	K_2	PID constant K_2
$0C_{16}$	R/W	K_3	PID constant K_3
10_{16}	R/W	Feedback	Feedback value — F(k)
14_{16}	R/W	Clk prescaler	Time step configuration
18_{16}	R/W	Status	PID status — Active (1), Stop (0))
$1C_{16}$	R/W	Clear	Flush PID controller registers
20_{16}	R/W	Bypass	FB Selection — No bypass (0), Bypass (0)
24_{16}	R/W	Saturation	Sat. enable — Enabled (0), Disabled (1)
28_{16}	R/W	Upper saturation	Upper saturation value
$2C_{16}$	R/W	Lower saturation	Lower saturation value
30_{16}	R	Control	PID controller output — U(k)

the system or values that can harm either the system operated or the operator. The saturation module operates over the output value of the controller. Regarding the configuration of the module, it depends on three values. (1) The enable/disable register, whose default value is zero, meaning that the saturation module is enabled by default. (2) The upper saturation value, and (3) the lower saturation value; both registers default value is zero. This configuration forces the PID peripheral to have a zero value in its default state that has to be manually disabled or modified.

Finally, the feedback bypass is a multiplexor that selects whether the feedback value of the controller is taken from the feedback registers (thus must be written by the CPU) or by the bypass port of the peripheral (see peripheral interface diagram in Figure **4.31**) which is directly connected to an ADC.

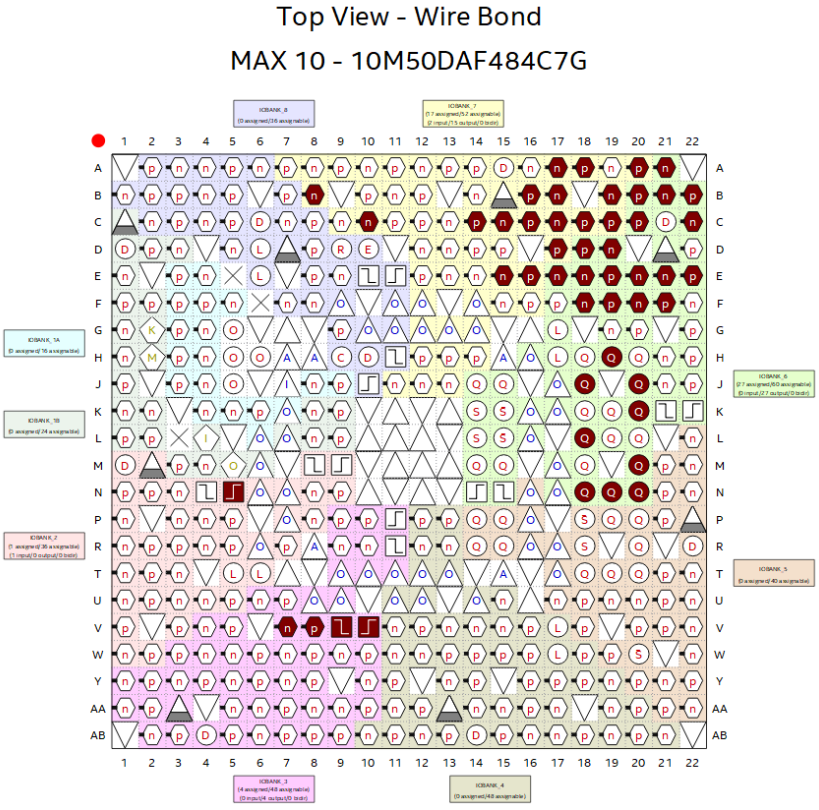
Figure 4.31: PID controller interface



4.3.4 Analysis & Synthesis

To bring the design from the simulation to the physical world, it is necessary to locate the input and output pins in the FPGA. Quartus incorporates the "Pin Planner" tool to locate the inputs and outputs. Figure **4.32** shows the pinout of the FPGA. However, the complete list is located in Annex **C**.

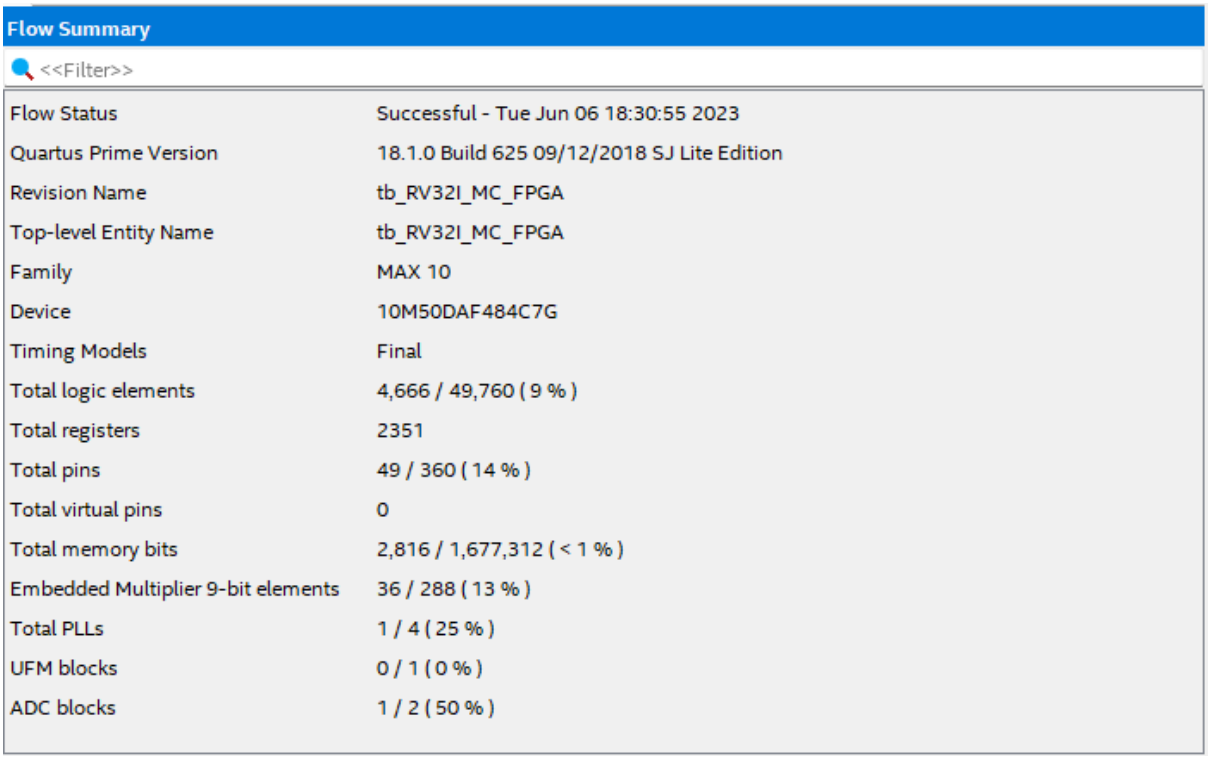
Figure 4.32: Pin Planner[13]



[12]Used pins are marked in red.

As a difference with the single-cycle, and as mentioned in Section **4.3.1.1**, the new memory module allows the use of the embedded memories. The usage of the memories can be observed in the "Total memory bits" field of Figure **4.33**. Furthermore, it can be seen that the logic elements employed are only 9% of the total, meaning that other logic could be running in parallel to the design. Even the same design could be duplicated in the same FPGA, resulting in an IC with two independent MCUs. Lastly, the timing analysis indicates that the maximum clock frequency for the MCU is 52.33 MHz.

Figure 4.33: Analysis & Synthesis



Flow Summary

<<Filter>>

Flow Status	Successful - Tue Jun 06 18:30:55 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	tb_RV32I_MC_FPGA
Top-level Entity Name	tb_RV32I_MC_FPGA
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	4,666 / 49,760 (9 %)
Total registers	2351
Total pins	49 / 360 (14 %)
Total virtual pins	0
Total memory bits	2,816 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	36 / 288 (13 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	1 / 2 (50 %)

4.4 Current controller

The levitation system of Auran utilizes an H-bridge topology (powered at 48V) to drive the coils. This configuration employs four signals, corresponding to two PWM and their complementary. Each pair of signals drive an individual half-bridge. In the feedback part, the applied current is measured with a shunt resistor. The current generates a voltage difference in the shunt resistor, which is then amplified and fed back to the controller. The current sensor, comprising the shunt and the amplifier, has a gain of 0.02 V/A.

Regarding the coil, it has been defined as a first-order system with the following transfer function:

$$G(s) = \frac{48}{4096} \cdot \frac{1}{0.0048 \cdot s + 0.4488} \tag{9}$$

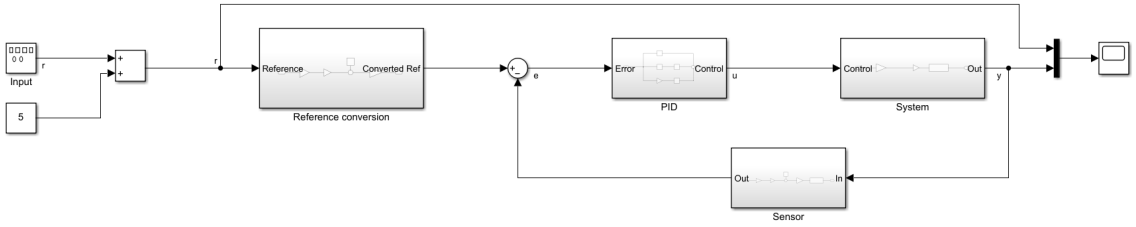
4.4.1 Controller design

Now that the system has been described, it is necessary to consider the control restrictions. Two restrictions must be followed. Firstly, the system cannot be underdamped, which means that no overshoot is accepted. The other parameter affected is the settling time, which must be within 0.6 to 0.8 seconds.

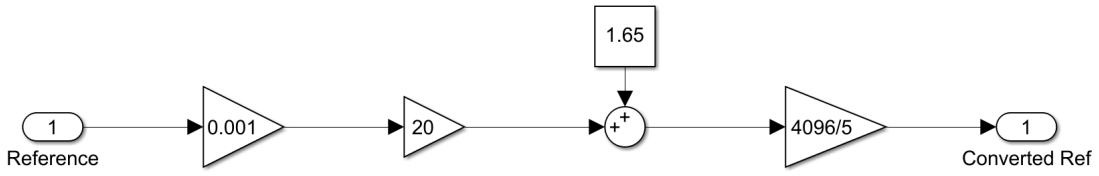
The controller employed is a PI (proportional-integral) controller. This decision has been taken because, due to the control requirements, a PI controller fulfils all the requirements expressed. Adding the derivative term may cause instability due to high-frequency noise, as the variation rate affects the derivative term response.

Figure 4.34: Simulink diagrams

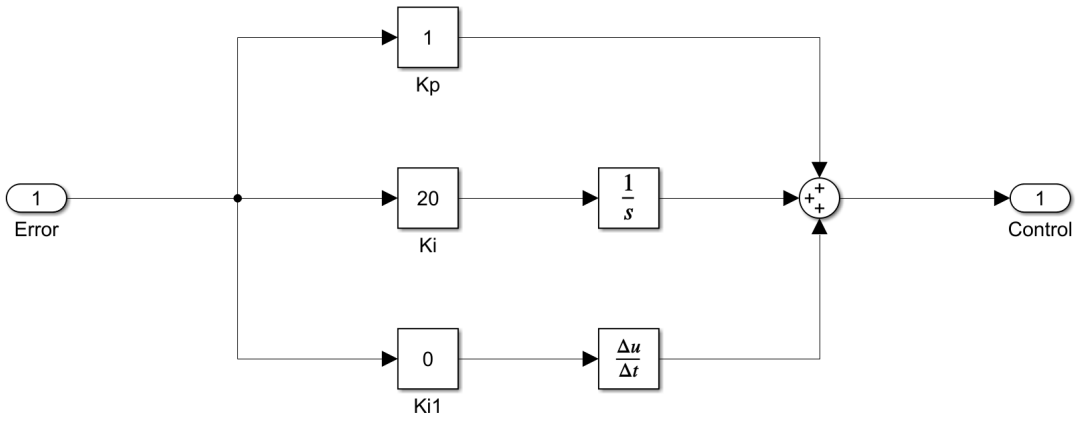
(a) Control loop



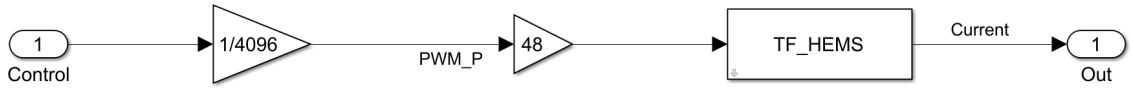
(b) Reference conversion



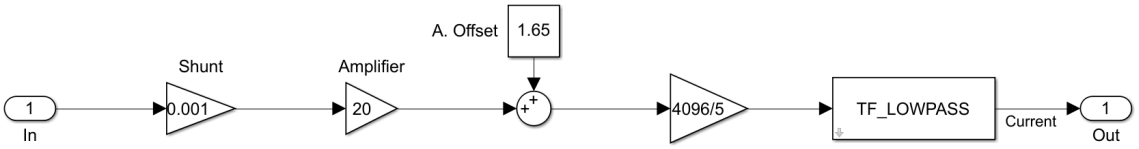
(c) PID



(d) System



(e) Sensor

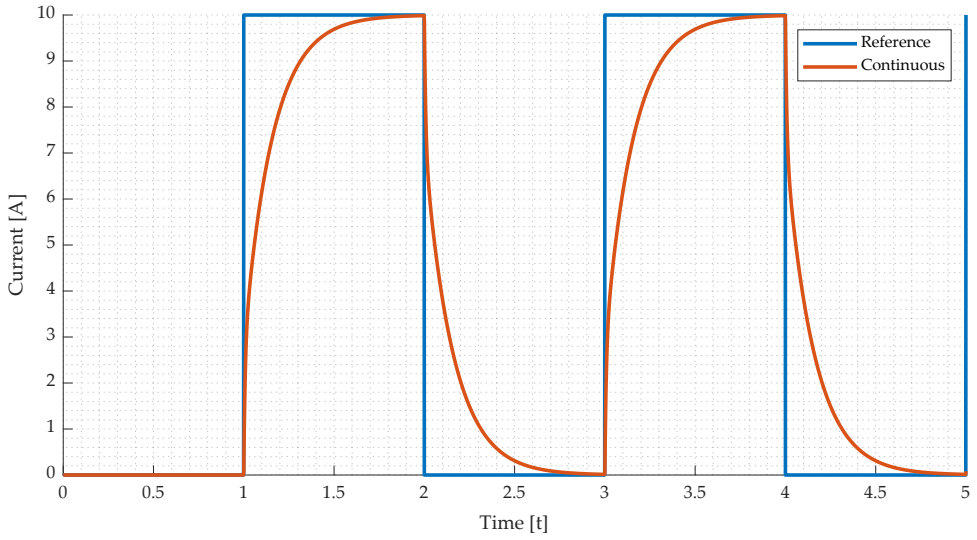


The system has been replicated in MATLAB Simulink to tune the PI controller. The control loop consists of four subsystems (shown in Figure **4.34**):

1. **Reference conversion:** The objective of this block is to take a reference in amperes to get the ADC would read.
2. **PID:** Contains the blocks corresponding to the parallel PID controller. The "slider gain" block has been chosen to implement the control gains because their value can be modified in runtime, so that the control can be tuned dynamically. Since the controller to be implemented is a PI, the gain for the K_d has been fixed to zero.
3. **System:** Represents the electromagnet. It also includes the transformation between the value output from the control to the equivalent applied voltage.
4. **Sensor:** This block includes the transformation from the current measurement to the ADC read value.

To tune the control dynamically, the simulation time has been set as infinite and a square waveform as the reference. The two gain sliders will be carefully modified to achieve a settling time of around 0.6-0.8 seconds and 0% of overshoot. The result for the gains $K_p = 1$ and $1/T_i = K_i = 20$ can be observed in Figure **4.34**.

Figure 4.34: Continuous PID simulation



Having checked that the control simulation operates correctly, the parameters must be transformed to adequate to the PID peripheral implementation employing the expressions defined in Equation **8**, considering $T = 0.001$, since the PI will be executed at 1 kHz. The final values of the control are the following:

$$K_1 = K_p + \frac{T_d}{T} = K_p = 1 \qquad K_2 = -K_p + \frac{T}{T_i} - 2\frac{T_d}{T} = -1 + \frac{0.001}{20^{-1}} = -0.98$$

$$K_3 = \frac{T_d}{T} = 0$$

Since K_2 is a decimal number and by rounding to units, the integral term would disappear, the solution is to shift the gains to the left by 10 (in their binary expression) or multiply by 2^{10} resulting in the following values:

$$K_1 = 1 << 10 = 1024 \qquad K_2 = -0.98 << 10 = -1004 \qquad K_3 = 0$$

With the values calculated and substituting the continuous PID module from the Simulink diagram with a model of the implemented PID (see Figure **4.35**), it can now be simulated. Figure **4.36** shows the time response of the control design. It can be observed that this control fulfils the requirements set by the Hyperloop UPV team.

Figure 4.35: PID circuit Simulink block

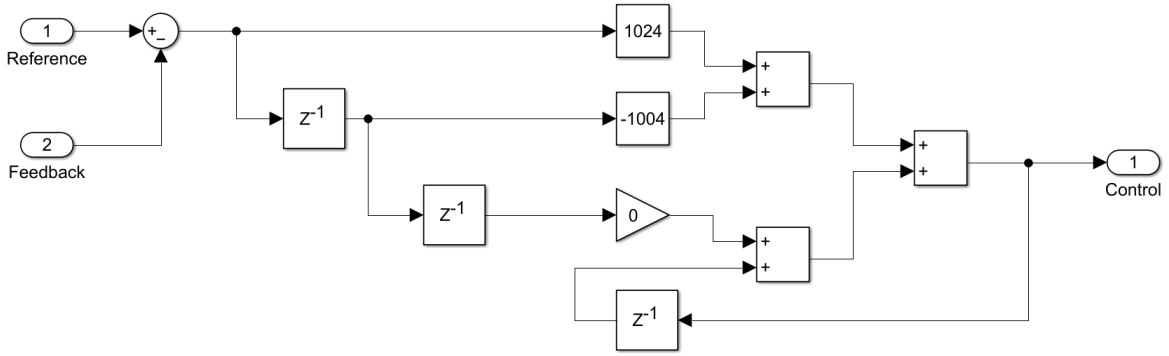
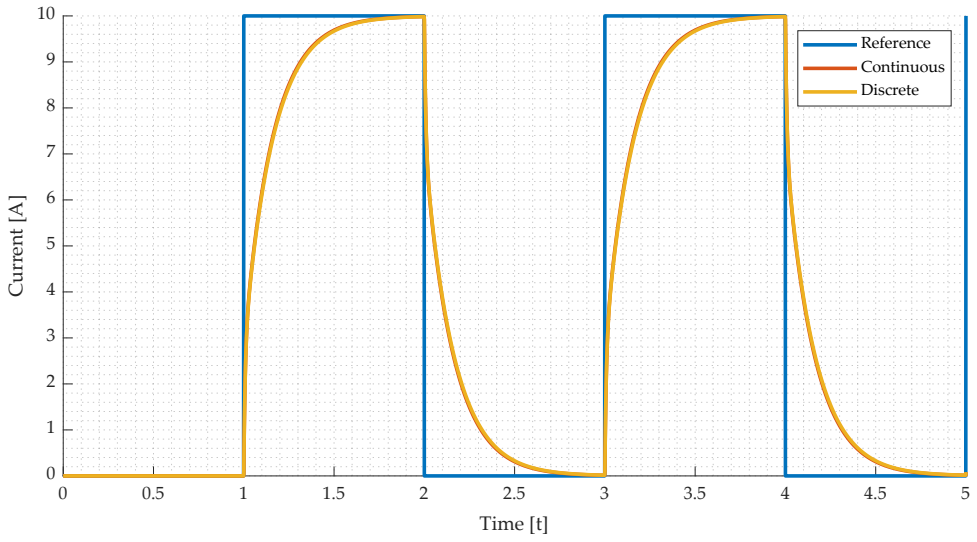


Figure 4.36: Discrete PID simulation



4.4.2 Software development

Once the hardware and the controller have been designed and in order to get the microcontroller to operate, a software must be developed. This software aims to set the peripherals to the desired conditions for the PI controller to operate without using the CPU. The peripherals involved in the operation of the controller are the ADC, the PID 1, the PID-Timer Link and the timer. Their configuration is following:

- **ADC**: The ADC does not need any configuration.
- **PID 1**: This peripheral must be configured to operate with a clock frequency of 1 kHz, which is the control frequency. The saturation must be activated and set

to not allow output currents over 30 amperes. Also, the controller gains must be loaded to the peripheral.

- **PID-Timer Link**: The shift value of the peripheral must be set to 10, to undo the shift in the PID constants.
- **Timer**: The timer must be set to output a PWM and its complementary in both channels, with a dead time of 200 ns. The frequency of the PWM signals must be set to 10 kHz. Finally, the duty cycle bypass must be activated.

To ease the configuration of the peripherals, the base addresses of the peripherals are stored in registers `x1` to `x3`.

4.4.2.1 PID configuration

To achieve an operation value of 1 kHz, the prescaler value must be modified. The value has been calculated by dividing the frequency of the clock signal by the desired operating frequency.

$$\text{Prescaler value} = \frac{f_{clk}}{f_{PID}} = \frac{50\ MHz}{1\ kHz} = 50,000 \tag{10}$$

Once configured the prescaler, gains calculated in Section **4.4.1** must be introduced. Finally, the saturation values must be configured. Since the maximum current allowed in the system is 30 amperes, according to Equation **11**, the saturation values have been set to 1,179,648 and -1,179,648 for the high and low saturation values, respectively.

$$\begin{aligned} \text{Saturation} &= \pm I_{sat} \cdot \left(\frac{V_{supply}}{R_{coil}}\right)^{-1} \cdot ADC_{max} \cdot Shift \\ \text{Saturation} &= \pm 30 \cdot \left(\frac{48}{0.45}\right)^{-1} \cdot 4096 \cdot 1024 = \pm 1,179,648 \end{aligned} \tag{11}$$

4.4.2.2 Timer configuration

The PWM signals shall be configured to 10 kHz. However, in the autonomous operation of the PID controller, it is not possible precisely adjust this value. Thus, obtaining a frequency close to the 10 kHz target is crucial. When the control output is the same as the ADC maximum count, the duty cycle of the PWM is 100%. Therefore, the frequency of the PWM is calculated as the timer frequency divided by the ADC maximum count.

$$f_{PWM} = \frac{f_{clk}}{ADC_{max}} = \frac{50\ MHz}{4096} = 12.207\ kHz \tag{12}$$

The obtained frequency has an error of 22.07% compared to the target. However, this error cannot be reduced. If the shifting value of the PID-Link is reduced to 9, the obtained frequency would be 6.104 kHz, what will represent an error of 38.96%.

In order to output two PWM and their complementary, the output enable register of the timer must be set to 1111_2.

Finally, the dead time must be configured to 200 ns. It is necessary to calculate the number of clock cycles that add up to 200 ns.

$$\text{DTV} = 50\ MHz \cdot 200\ ns = 10 \tag{13}$$

Applying the formula described in Table **4.21**, the value for the dead time register is 10.

Once the configuration of each peripheral has been explained, the resultant assembly code can be found in Section **??** of Part **??**.

5 Verification

Verification is the process of checking the correct operation and functionality of a hardware. Hardware description languages (HDL) offer commands that cannot be synthesisable (hardware cannot be generated) but support functionality for verification and simulation purposes. The non-synthesisable code can be used to create testbenches. One of the main reasons behind the selection of SystemVerilog is its support for verification tools and commands, for example, assertion-based verification, coverage and testbench constructs and hierarchical design interfaces.

A testbench is a module able to apply inputs to a module, the device under test (DUT), and analyse the correctness of the outputs generated by the DUT. Testbenchs have been employed to verify every module designed for this project. The testbenches have been structured in the following way:

- Signal declaration
- Device under test instatiation
- Clock generation
- Initialisation
- Input signal generation
- Output verification

Complete testbenches, input vectors, expected outputs and assembly files can be found in Section **??** of Document **??**.

5.1 Single-cycle

As mentioned previously, the modules have been verified, employing testbenches. For the single-cycle processor modules, the testbenches have been implemented using the specified structure.

The testbenches load a file containing all the test cases defined for each module. Every test case comprehends a set of input signals and their expected output values. When a test case is applied to the DUT, the outputs are compared to the expected output values. If the output value coincides with the expected value, the test case is considered as passed. However, an error message is printed in the simulation console if a discrepancy is encountered between the output and the expected value. Each testbench incorporates specific error messages to identify better the issues that

generated that error (see Figure **5.1**). Finally, a message is printed showing the number of tests executed and the errors that have arisen in the verification process (Figure **5.1**).

Figure 5.1: Verification error log

```
# Arithmetic Logic Unit Testbench
# Error: result = 11100000 (expecting 00000000).
# Error: flags = 0000 (expecting 0001).
#  13 tests completed with   2 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/ALU/Quartus/../tb_ALU.sv(54)
#    Time: 125 ps  Iteration: 1  Instance: /tb_ALU
```

5.1.1 Modules verification

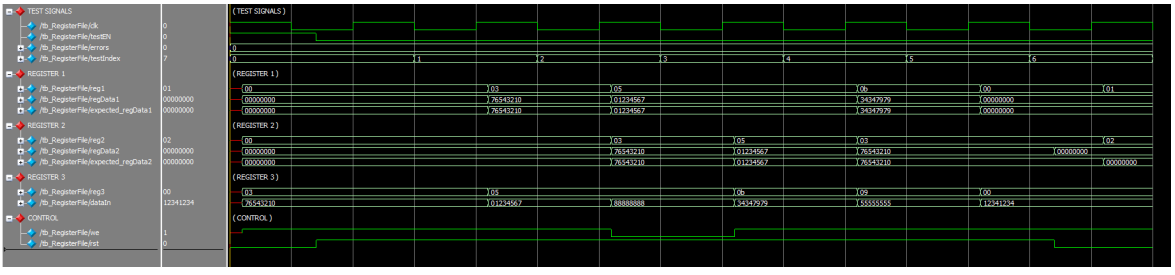
The following figures show the results obtained for the different modules of the single-cycle processor.

Register file

Test cases analysed:

- Correctness in the data input and output by assuring the data outputted coincides with the input data and that the register is the same as the one in which it was written.
- Check reset and write enable signals.
- Register `x0` is always zero

Figure 5.2: Register file verification results



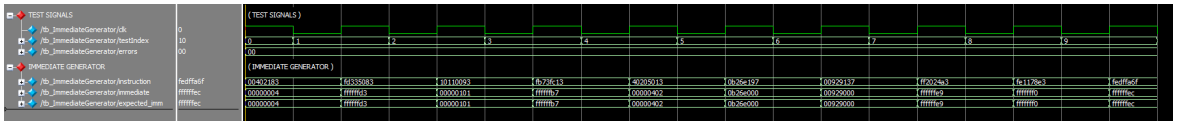

```
# Register File Testbench
#   7 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/Register file/Quartus/../tb_RegisterFile.sv(64)
#    Time: 75 ps  Iteration: 1  Instance: /tb_RegisterFile
```

Immediate generator:

Test cases analysed:

- The immediate is generated correctly for every instruction type.

Figure 5.3: Immediate generator verification results



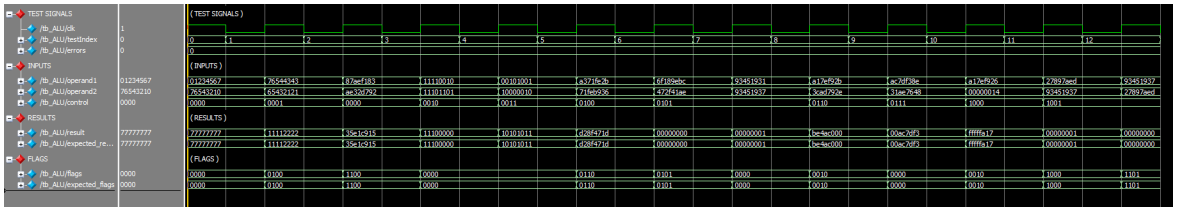
```
# Immediate Generator Testbench
#  10 tests completed with  0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/Immediate generator/Quartus/../tb_ImmediateGenerator.sv(49)
#    Time: 95 ps  Iteration: 1  Instance: /tb_ImmediateGenerator
```

Arithmetic Logic Unit:

Test cases analysed:

- Correctness in the results of the operations considering overflow situations and signed numbers.
- Check flags generation

Figure 5.4: ALU verification results



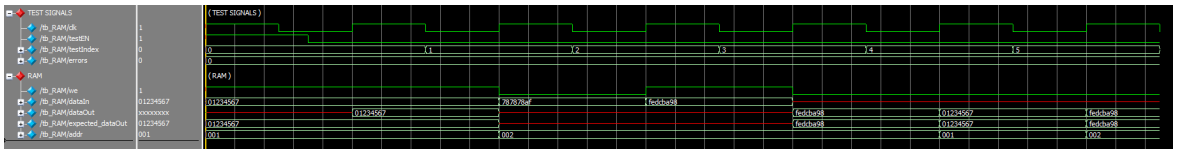
```
# Arithmetic Logic Unit Testbench
# Error: result = 11100000 (expecting 00000000).
# Error: flags = 0000 (expecting 0001).
#  13 tests completed with  2 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/ALU/Quartus/../tb_ALU.sv(54)
#    Time: 125 ps  Iteration: 1  Instance: /tb_ALU
```

Data memory:

Test cases analysed:

- Assure that data writes to the correct address and can be read.
- Check write enable functionality.

Figure 5.5: Data memory verification results



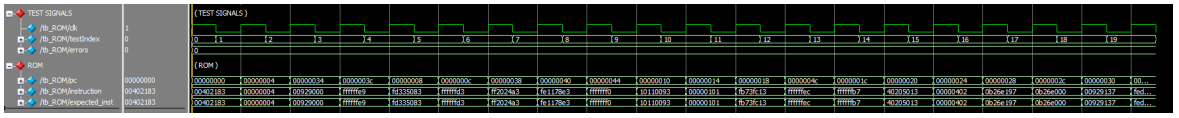
```
# Random Access Memory Testbench
#   6 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/RAM/Quartus/../tb_RAM.sv(50)
#    Time: 65 ps  Iteration: 1  Instance: /tb_RAM
```

Program memory:

Test cases analysed:

- Assure that data written by the initialisation file is correctly applied.
- Instructions are outputted correctly.

Figure 5.6: Program memory verification results



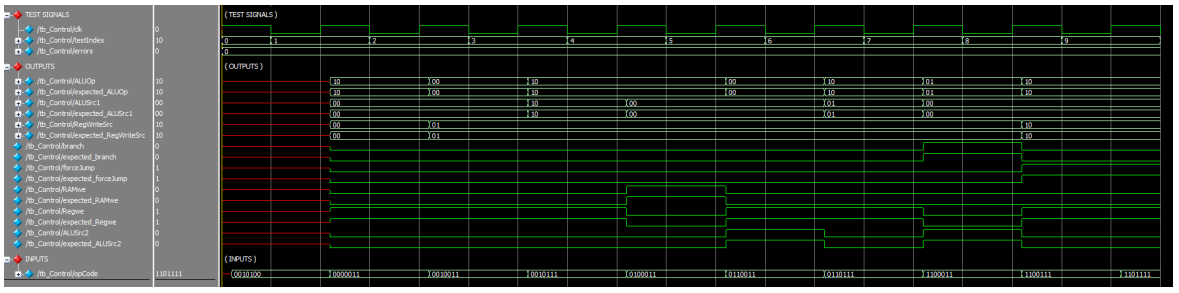
```
# Read Only Memory Testbench
#  20 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/ROM/Quartus/../tb_ROM.sv(47)
#    Time: 195 ps  Iteration: 1  Instance: /tb_ROM
```

Control unit:

Test cases analysed:

- Control signals are generated accordingly to the instruction being executed.

Figure 5.7: Control unit verification results



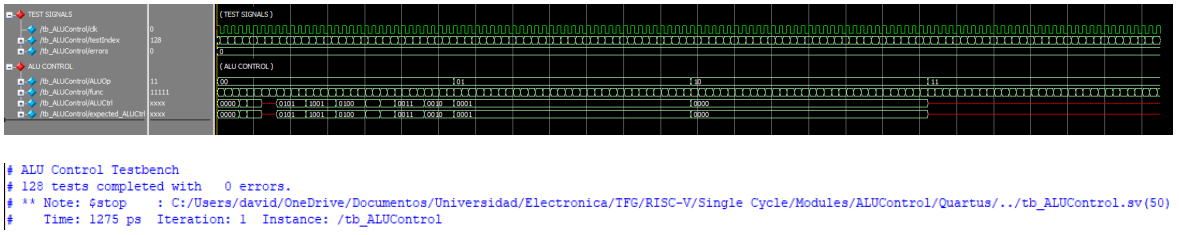
```
# Control Testbench
#  10 tests completed with    0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/Control/Quartus/../tb_Control.sv(88)
#    Time: 95 ps  Iteration: 1  Instance: /tb_Control
```

ALU control:

Test cases analysed:

- ALU control signals are generated according to the instruction being executed.

Figure 5.8: ALU control verification results



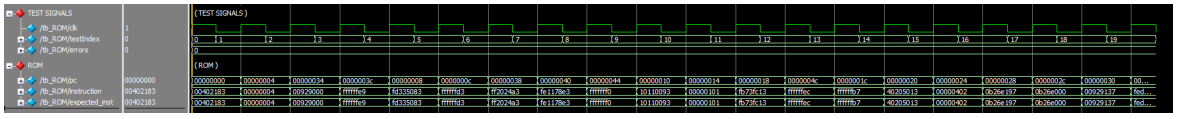
```
# ALU Control Testbench
# 128 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/ALUControl/Quartus/../tb_ALUControl.sv(50)
#    Time: 1275 ps  Iteration: 1  Instance: /tb_ALUControl
```

Branch logic:

Test cases analysed:

- "PC Src control" signals are generated according to the instruction being executed and the flags output from the ALU.

Figure 5.9: Branch logic verification results



```
# Read Only Memory Testbench
#  20 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/ROM/Quartus/../tb_ROM.sv(47)
#    Time: 195 ps  Iteration: 1  Instance: /tb_ROM
```

5.1.2 Processor verification

To verify the correct implementation of the modules, a testbench has been developed to simulate the complete implementation of the processor. The input of the testbench is the assembled files for different software scripts that allow the implementation of the microcontroller.

The verification has been developed in two different stages. The first one is the input of sequences with all the different instructions supported by the CPU. The second is the execution of two programs for actual application conditions. The programs developed are the bubble sort algorithm and the Fibonacci series, which were both implemented in RISC-V assembly and then assembled and input as machine code.

5.1.2.1 Instruction set verification

In the first place, type I and type S instructions have been tested. The script developed executes each instruction (at least once), and the results of each operation are stored in a different register so the proper functioning of the CPU can be verified at the end of the simulation. Figure **5.10** shows the simulation waveforms. Figure **5.11**, on the other side, shows the register file and data memory (only the written address) at the end of the simulation.

Code snippet 5.3: Types I, S verification

```
addi x1, x0, 30        # x1 = 30
addi x2, x1, -15       # x2 = 15
slli x3, x2, 4         # x3 = 240
slti x4, x2, 20        # x4 = 1
slti x5, x1, 20        # x5 = 0
addi x6, x2, -30       # x6 = -15
slti x7, x6, -20       # x7 = 0
slti x8, x6, -10       # x8 = 1
slti x9, x2, -20       # x9 = 0
sltiu x10, x2, 20      # x10 = 1
sltiu x11, x1, 20      # x11 = 0
xori x12, x1, 54       # x12 = 40
srli x13, x1, 1        # x13 = 15
slli x14, x4, 31       # x14 = -2.147.483.648
srai x15, x13, 3       # x15 = 1
srai x16, x14, 8       # x16 = -8.388.608
ori x17, x12, 48       # x17 = 56
andi x18, x17, 240     # x18 = 48
sw x18, 64(x0)         # RAM 0x00000004 = 48
lw x19, 64(x0)         # x19 = 48
jalr x20, x12, 16      # x20 = 88 --> PC to srai x15, x13, 3
nop
```

Figure 5.10: Types I, S verification waveforms

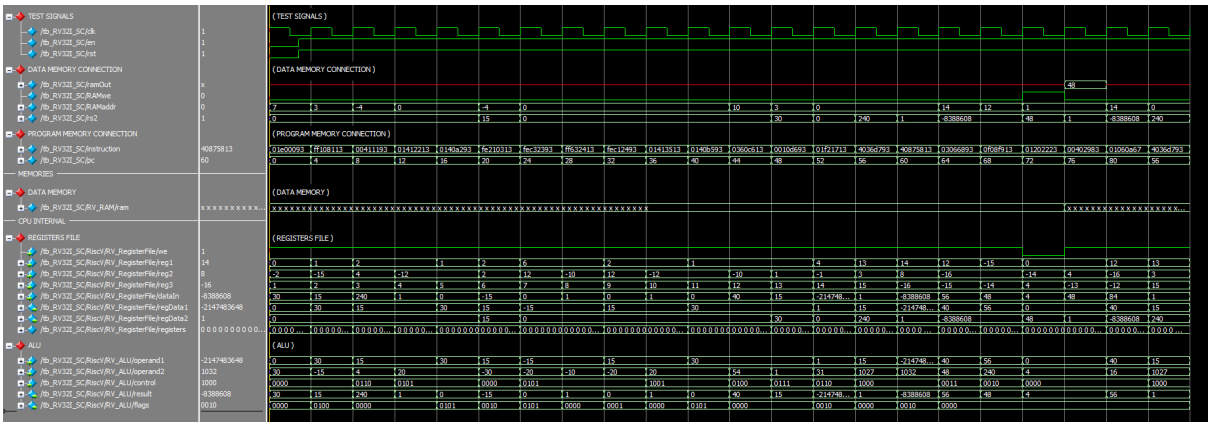
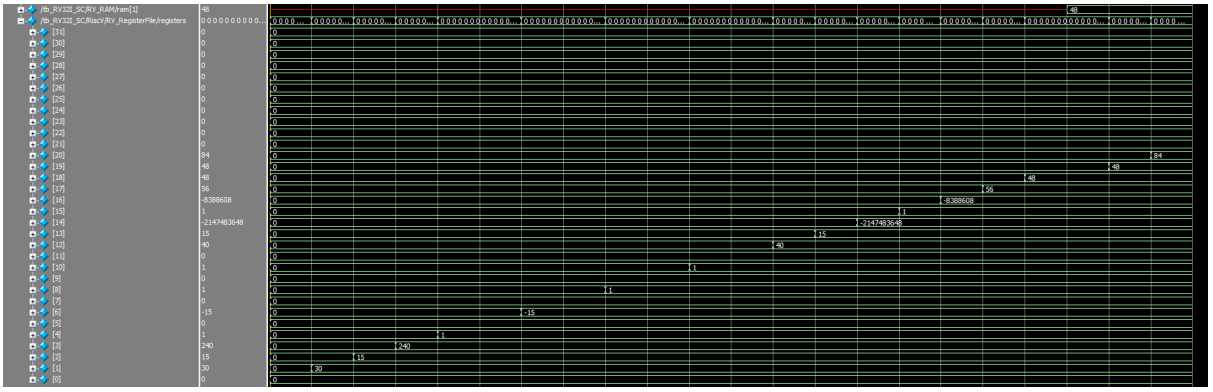


Figure 5.11: Type I, S register file waveforms



Proceeding as with types I and S, types R, U and J have been tested altogether. Figure **5.12** shows an overview of the complete CPU, while Figure **5.12** shows a detailed view of the registers file.

Code snippet 5.4: Types R, U, J verification

```
    addi x1, x0, 30     # x1 = 30
    addi x2, x0, 15     # x2 = 15
    add x3, x1, x2      # x3 = 45
    sub x4, x1, x2      # x4 = 15
    sub x5, x2, x1      # x5 = -15
    addi x6, x0, 2      # x6 = 2
    sll x7, x2, x6      # x7 = 60
    slt x8, x4, x3      # x8 = 1
    slt x9, x3, x4      # x9 = 0
    slt x10, x4, x5     # x10 = 0
```

```
    slt x11, x5, x4     # x11 = 1
    sltu x12, x4, x3    # x12 = 1
    sltu x13, x3, x4    # x13 = 0
    xor x14, x2, x7     # x14 = 51
    srl x15, x2, x6     # x15 = 3
    sra x16, x2, x6     # x16 = 3
    sra x17, x5, x6     # x17 = -4
    or x18, x2, x7      # x18 = 63
    and x19, x2, x7     # x19 = 12
    lui x20, 2          # x20 = 8192
    auipc x21, 2        # x21 = 8272
    jal x22, .link      # x22 = 88
    nop
    nop
    nop
    .link:
    addi x23, x0, 1     # x23 = 1
    nop
```

Figure 5.12: Types R, U, J verification waveforms

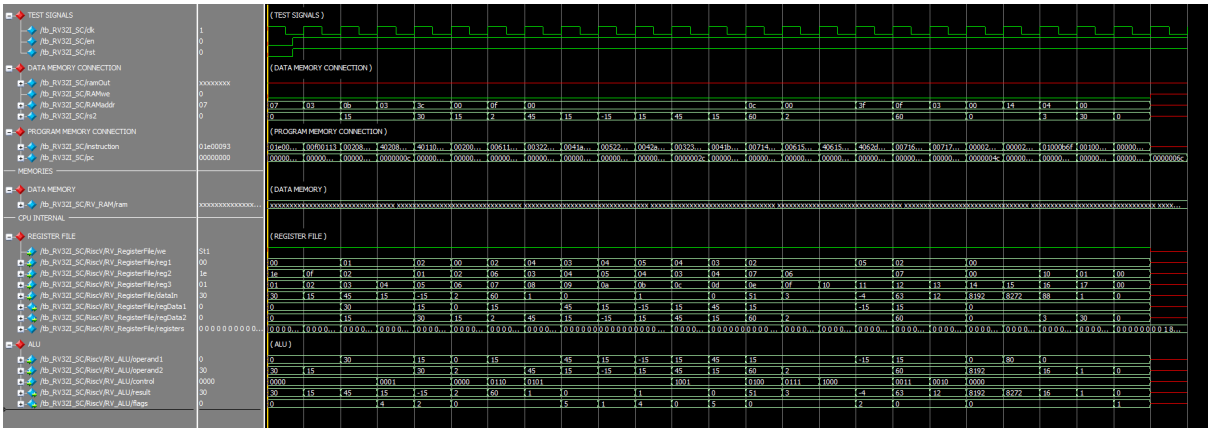
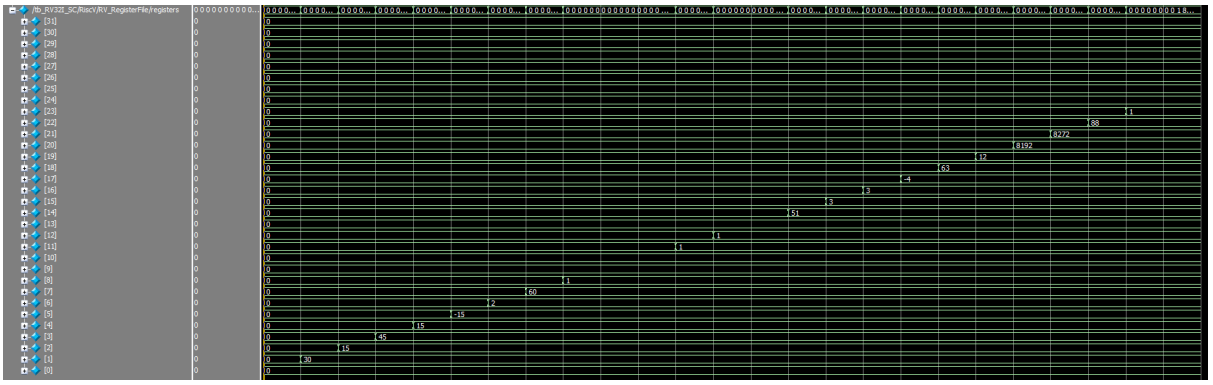


Figure 5.13: Types R, U, J register file waveforms



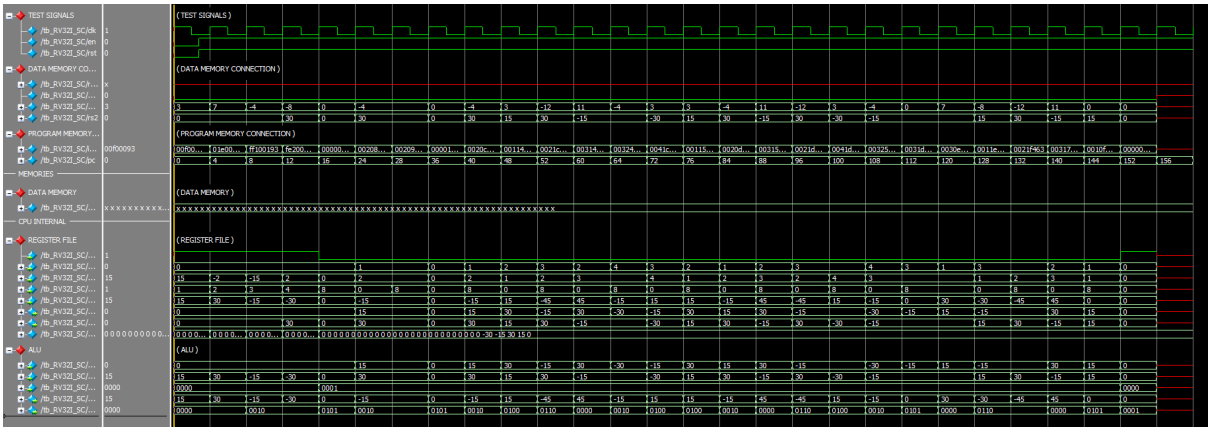
Finally, the same procedure is applied to branch instructions (type B). However, for this case, the main focus is on the PC value. As it can be seen in the Code Snippet **5.5**, every instruction is evaluated for a case in which it must perform a jump and a situation for which it is not supposed to perform it. The results can be seen in Figure **5.14**.

Code snippet 5.5: Type B verification

```
    addi x1, x0, 15        # x1 = 15
    addi x2, x0, 30        # x2 = 30
    addi x3, x0, -15       # x3 = -15
    addi x4, x0, -30       # x4 = -30
    beq x0, x0, .jump1     # if x0 == x0 then target
    nop
    .jump1:
    beq x1, x2, .jump1
    bne x1, x2, .jump2     # if x1 != x2 then target
    nop
    .jump2:
    bne x0, x0, .jump2     # if x0 != x0 then target
    blt x1, x2, .jump3     # if x1 < x2 then target
    nop
    .jump3:
    blt x2, x1, .jump3     # if x2 < x1 then target
    blt x3, x2, .jump4     # if x3 < x2 then target
    nop
    .jump4:
    blt x2, x3, .jump4     # if x2 < x3 then target
    blt x4, x3, .jump5     # if x4 < x3 then target
    nop
    .jump5:
    blt x3, x4, .jump5     # if x3 < x4 then target
    bge x2, x1, .jump6     # if x2 >= x1 then target
```

```
    nop
    .jump6:
    bge x1, x2, .jump6     # if x1 >= x2 then target
    bge x2, x3, .jump7     # if x2 >= x3 then target
    nop
    .jump7:
    bge x3, x2, .jump7     # if x3 >= x2 then target
    bge x3, x4, .jump8     # if x3 >= x4 then target
    nop
    .jump8:
    bge x4, x3, .jump8     # if x4 >= x3 then target
    .back1:
    bge x3, x3, .jump9     # if x3 >= x3 then target
    jal x0, .back1
    .jump9:
    bltu x1, x3, .jump10   # if x1 < x3 unsigned then target
    nop
    .jump10:
    bltu x3, x1, .jump10   # if x3 < x1 unsigned then target
    bgeu x3, x2, .jump11   # if x3 >= x2 unsigned then target
    nop
    .jump11:
    bgeu x2, x3, .jump11   # if x2 >= x3 unsigned then target
    .back2:
    bgeu x1, x1, .jump12   # if x1 >= x1 unsigned then target
    jal x0, .back2
    .jump12:
    nop
```

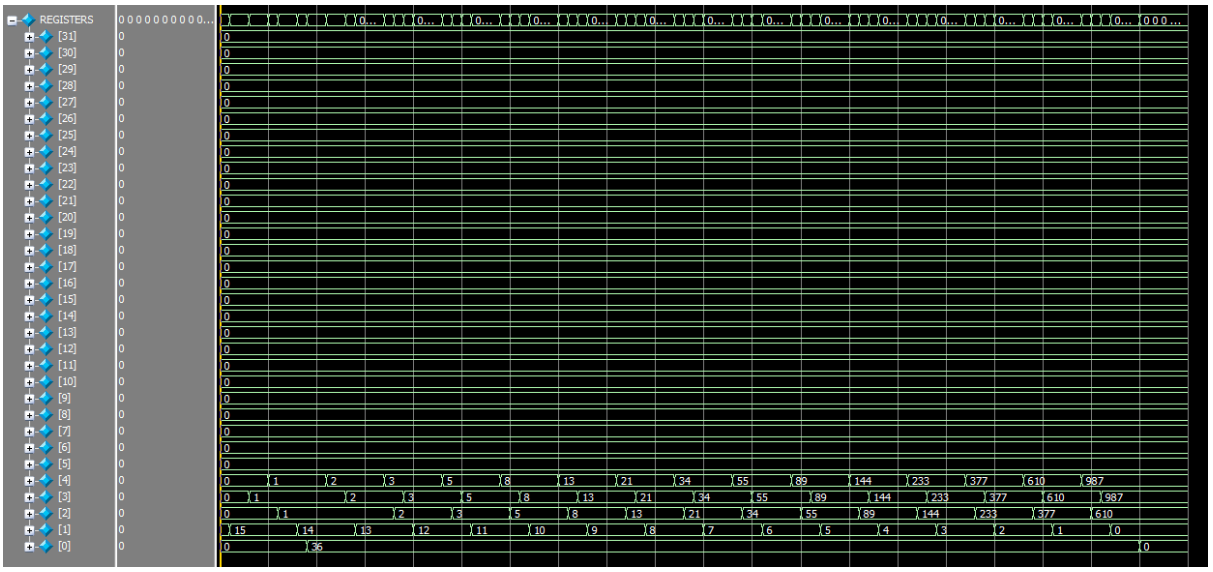
Figure 5.14: Type B register file waveforms



5.1.2.2 Fibonacci sequence

Once all the instructions have been individually verified, two test programs have been developed. In the first place the Fibonacci sequence. The program starts with the first two values and calculates the 15 first numbers (it could be modified by the initialisation value of register `x1`). Figure 5.15 shows the value of the registers modified during the execution. Register `x3` shall have the 16 first values of the series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987.

Figure 5.15: Fibonacci sequence



5.1.2.3 Bubble sort

Bubble sort is a simple sorting algorithm that steps through an array of elements swapping the element *n* with the element *n+1* if they are in the wrong order. The algorithm goes through the array until no swaps have been performed.

An example can be found in Table **5.1**. The first column holds the unordered array, and the last column shows the result of the bubble sort algorithm.

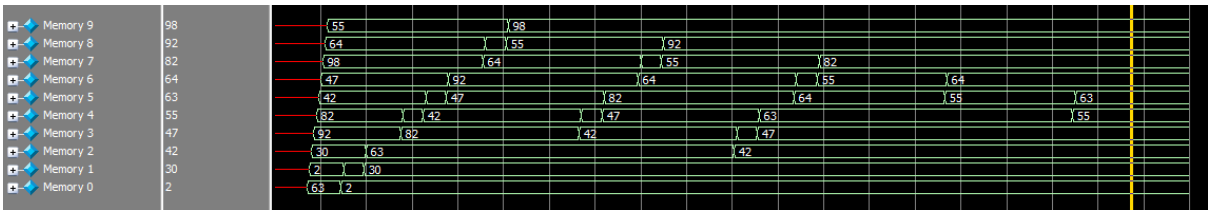
The assembly code generates a 10-number array and applies bubble sort to order the array. The simulation waveforms are shown in Figure **5.16**

Table 5.1: Bubble sort execution

Initial array	Execution									Result
3	3	3	3	4	4	4	4	4	4	**4**
1	1	1	4	3	3	3	3	3	3	**3**
2	2	4	1	1	2	2	2	2	2	**2**
4	4	2	2	2	1	1	1	1	1	**1**
Iterations	1^{st}			2^{nd}			3^{rd}			

Note: Red cells stand for a comparison that results in a swap action. Green cells indicate a comparison that do not perform a swap

Figure 5.16: Bubble sort



5.2 Multicycle

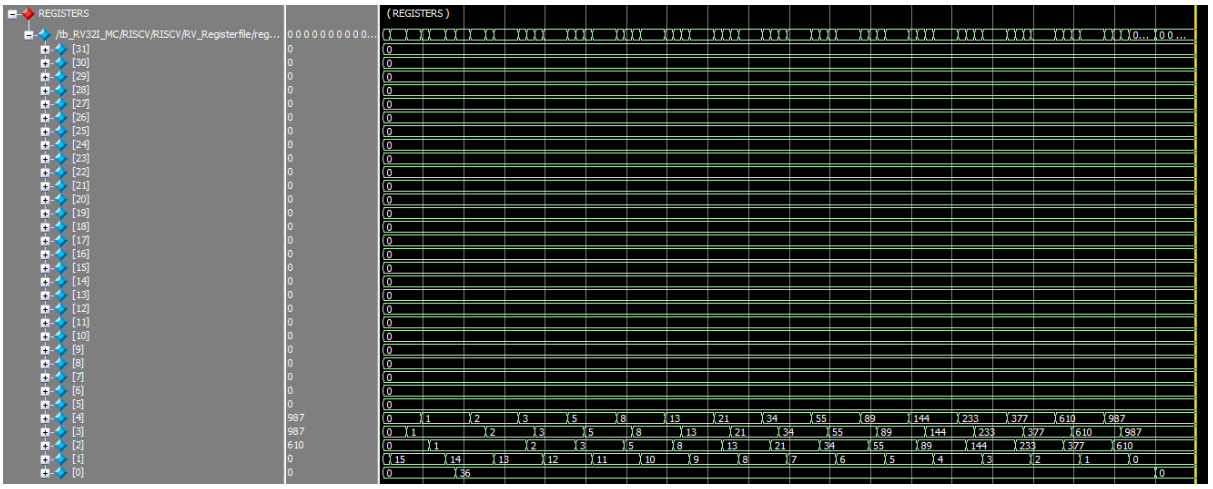
5.2.1 Processor

The most significant module modification of the migration from the single-cycle processor to the multicycle processor is the control module which changes from an instruction decoder (combinational logic) to a state machine (sequential logic). Even though this module could be independently verified, the module has been tested integrated into the CPU, so not only has the control module been designed according to the specification, but the specifications have been correctly designed.

Therefore the test conducted to test the CPU[14] has been the Fibonacci sequence. It can be seen in Figure **5.17** that the Fibonacci sequence is generated in the register file correctly.

[14]Even though the test was aimed to test the CPU, the complete MCU was simulated. Including memories and peripherals.

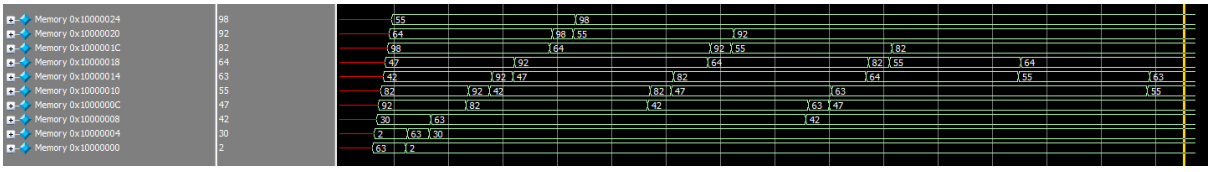
Figure 5.17: Fibonacci sequence



5.2.2 Memory bus

Stepping to the test of the memory bus, the bubble sort algorithm allows to test that the logic for interacting with the memory bus works adequately and that data can be written into the bus and read from it. Figure **5.18** shows the result of the bubble sort algorithm simulation. It is noteworthy to indicate that the bubble sort assembly code has been modified to fit with the memory map.

Figure 5.18: Bubble sort



Finally, it has been tested that peripheral registers are accessible from the CPU, therefore thoroughly validating the memory bus. However, the peripherals themselves have not been tested as they have been individually validated, as shown in Section **5.2.3**. Figures **5.19** to **5.23** show the validation process results.

The test code (Code snippet **5.6**) writes into the registers and then reads the same register to check. Then if the written value is both in the peripheral register and the register file `x4` register, both write and read operations were successful. For read-only addresses, the read value is not determined by the write operation. In the PID case, it can be observed that the output is generated by the control signal generated by the PID

controller. Also, in the ADC peripheral, it can be seen that the outputs are the values of channels 1 and 2.

Code snippet 5.6: Memory bus verification

```
li x1, 0xC0000000     # Start address
li x2, 0xC00000B0     # End address
li x3, 1              # Start value

.bucle:
sw x3, 0(x1)          # Write register
lw x4, 0(x1)          # Read Register
addi x1, x1, 0x4      # Next address
addi x3, x3, 1        # Increase value
bne x1, x2, .bucle    # while address != End address
nop
```

Figure 5.19: PID 1 registers verification

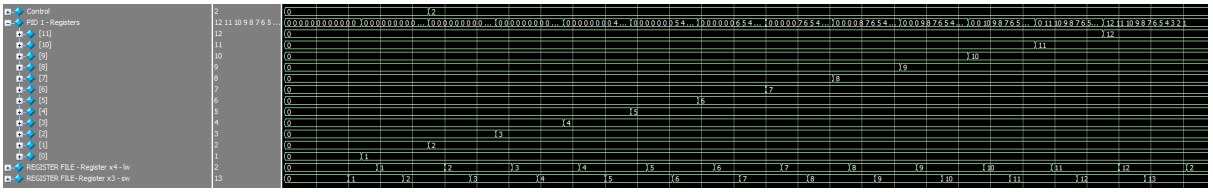


Figure 5.20: PID 2 registers verification

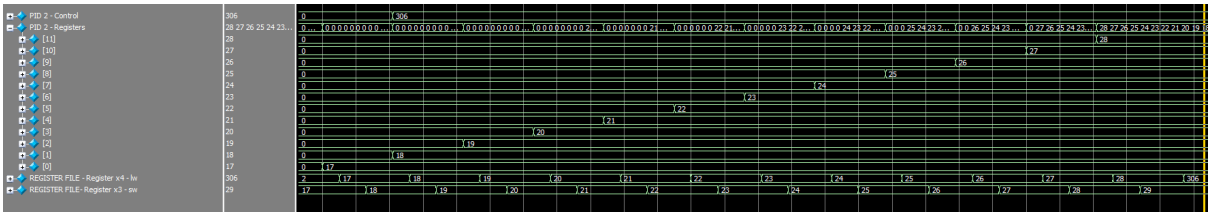
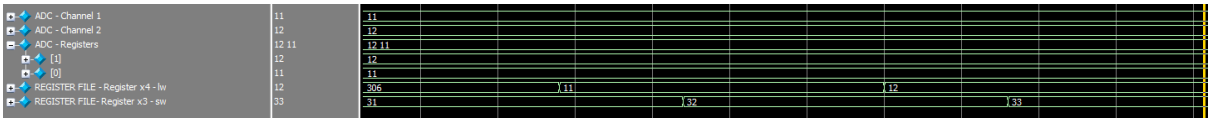


Figure 5.21: ADC registers verification



It can be observed in Figure **5.22** how modifying the registers affect the timer peripheral, as it starts counting once the ”Start” (timer register 2) register has been written. Also, the effect of the prescaler on the counter period can be seen.

Figure 5.22: Timer registers verification

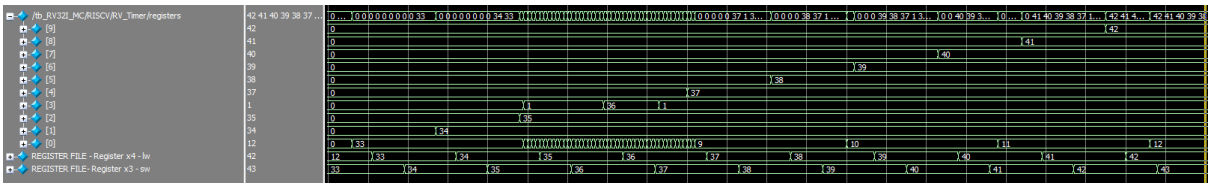
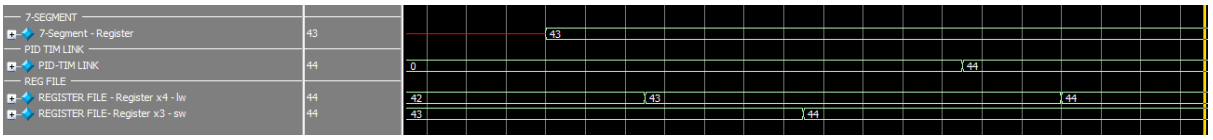


Figure 5.23: 7-segment and PID-Timer link registers verification



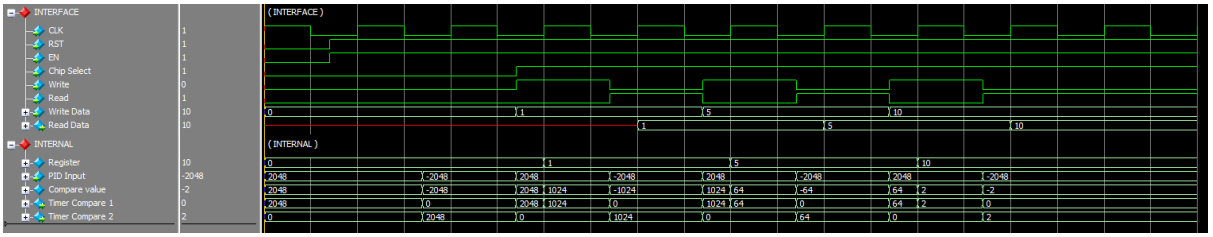
5.2.3 Peripherals

5.2.3.1 PID-Timer link

The procedure followed to test the PID-Timer Link peripheral has been to input a value (alternating positive and negative) through the PID input port and modify the ”Shift” register utilising the bus memory interface.

The expected results were that the timer compare outputs acquire the value of the PID input shifted by the “Shift” register or zero depending on the input value sign. The obtained results are shown in Figure **5.24**.

Figure 5.24: PID-Timer link verification



5.2.3.2 Seven-Segments decoder

Verifying the seven-segment decoder consists of inputting a value and observing that the outputs have been generated correctly (Figure **5.25**). The input value was 0x00654321; therefore, the seven-segment shall display "654321", as shown in Figure **5.26**.

Figure 5.25: Seven-segment decoder verification

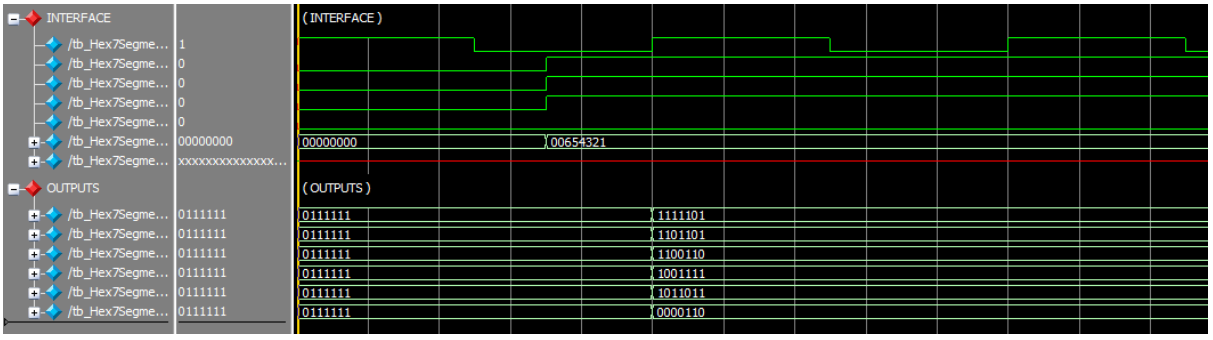
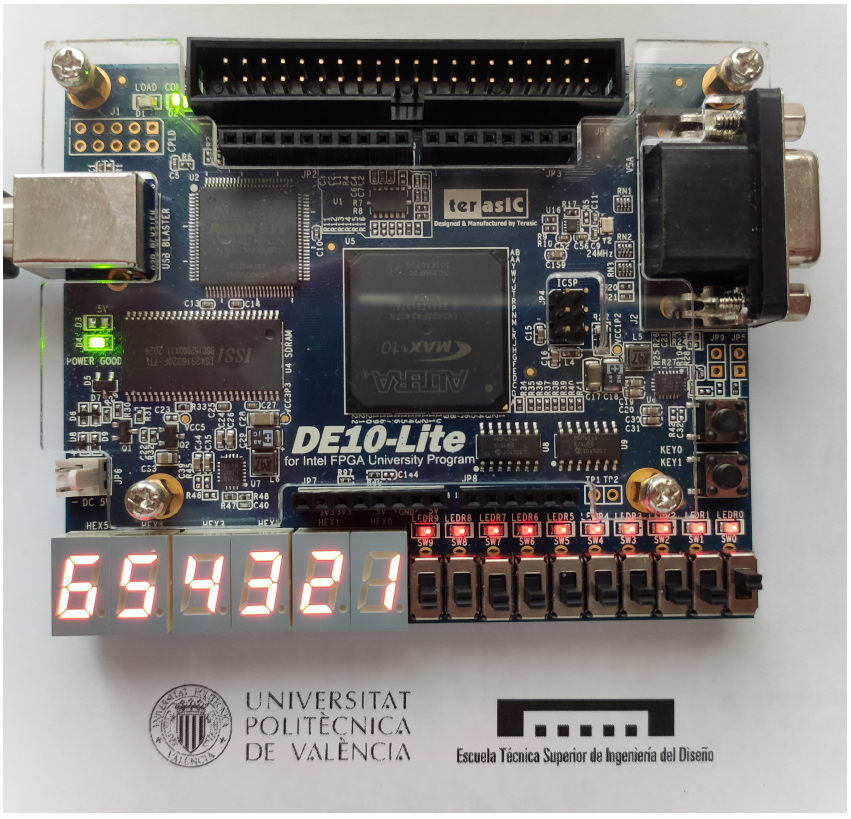


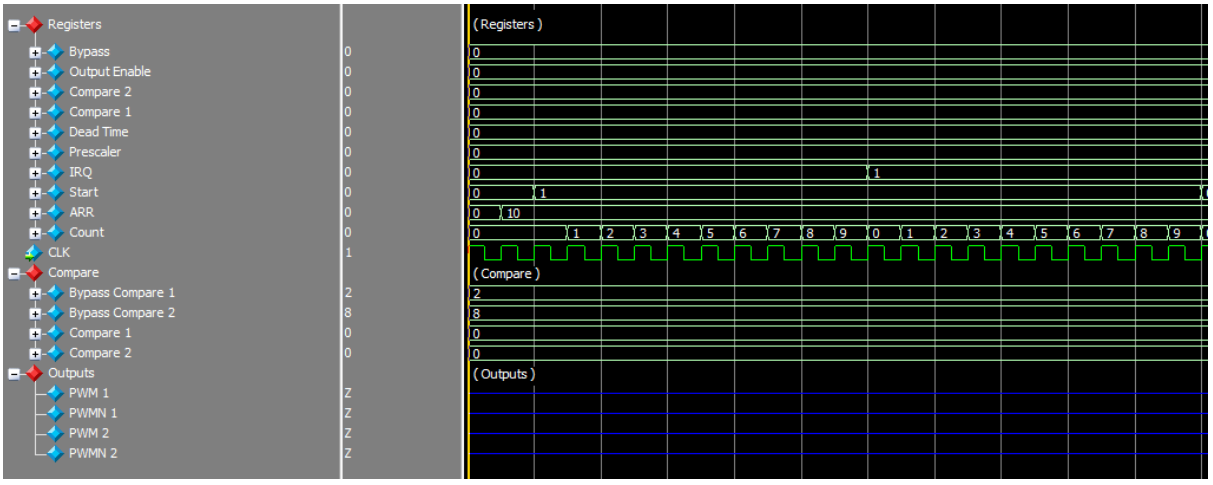
Figure 5.26: Seven-segment decoder verification FPGA



5.2.3.3 Timer

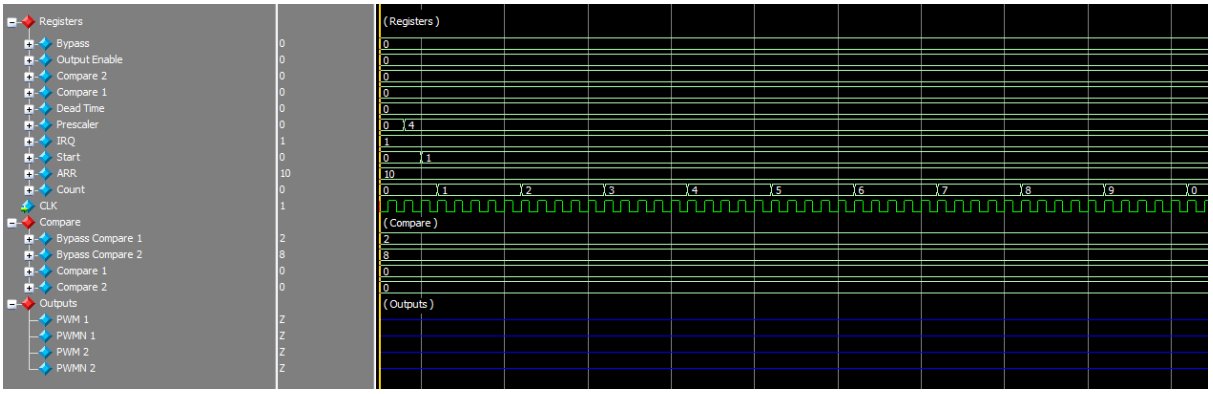
The different features of the timer have been tested individually. Firstly, it has been verified that the maximum count value limits the count value and that the end of the count indicator is asserted. Figure **5.27** shows how the output is limited to 10 and that the "IRQ" value is asserted at the end of the count.

Figure 5.27: Timer counter limit verification



The following feature verified is the prescaler module. For the verification process, it was set to reduce the frequency by five. Therefore counter period was increased to five clock cycles, as shown in Figure **5.28**.

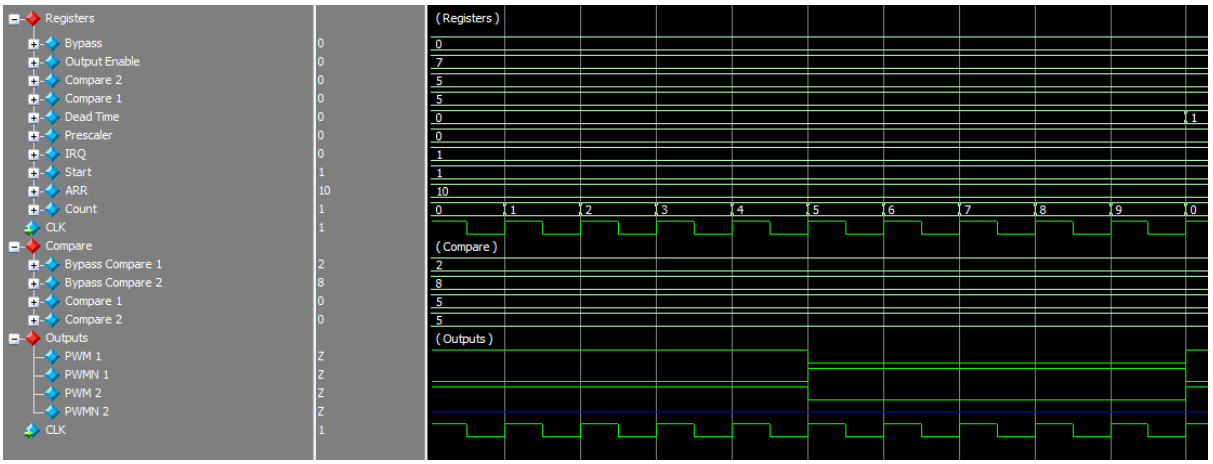
Figure 5.28: Timer prescaler verification



To check the proper functioning of the PWM outputs, they have been configured to output in channel one a PWM and its complementary and in channel two just the

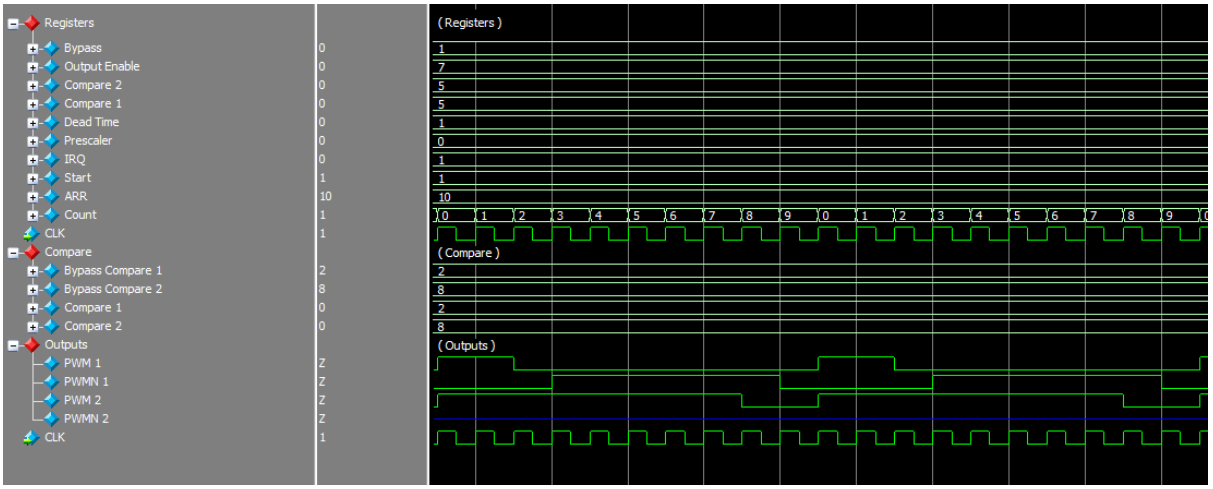
PWM signal, both channels with a duty cycle of 50% (Figure **5.29**).

Figure 5.29: Timer outputs verification



Finally, the compare bypass and the dead-time have been verified. The compare bypassed values have been set to two and eight (channel one and channel two, respectively) and the dead-time to one clock cycle. Figure **5.30** shows the result of the verification.

Figure 5.30: Timer bypass and dead-time verification



5.2.3.4 PID controller

The PID controller has been verified by comparing it with a golden model. The golden model has been generated as a continuous PID controller in MATLAB Simulink. With this model, the reference and the feedback are stored in a file and then are input to the hardware simulation. Moreover, every controller signal is stored in files for further analysis if necessary. Figure **5.31** shows the golden model simulation. Figure **5.32** compares the control action of the golden model and the hardware-implemented controller.

Figure 5.31: Golden model simulation

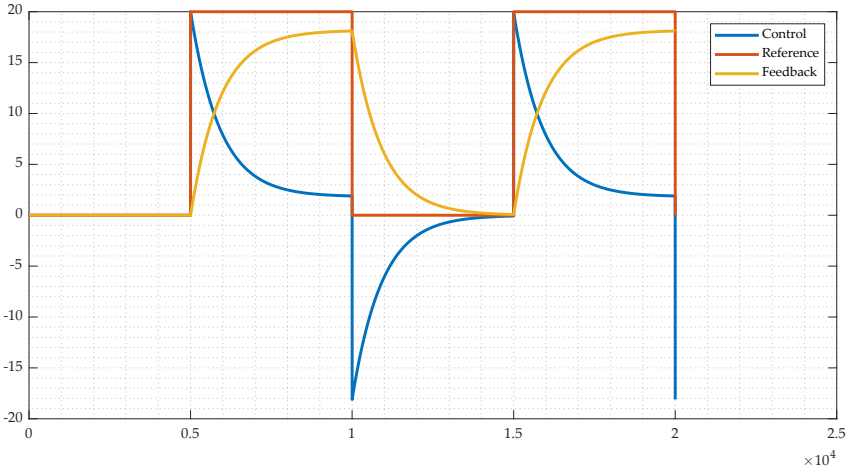
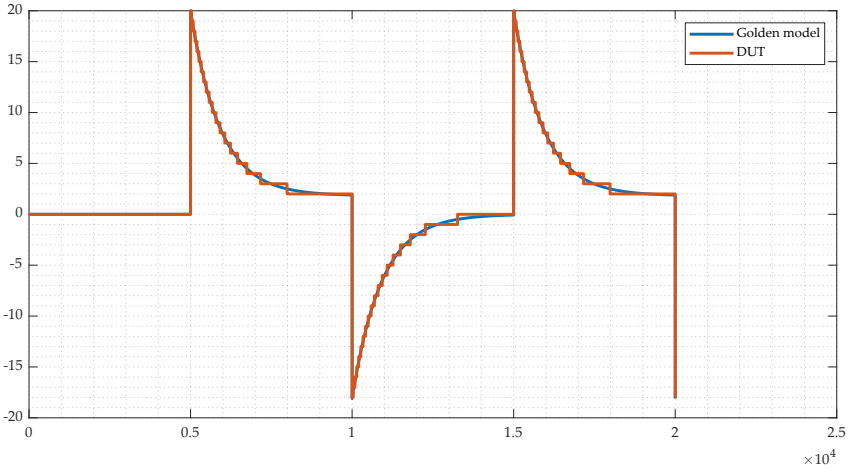
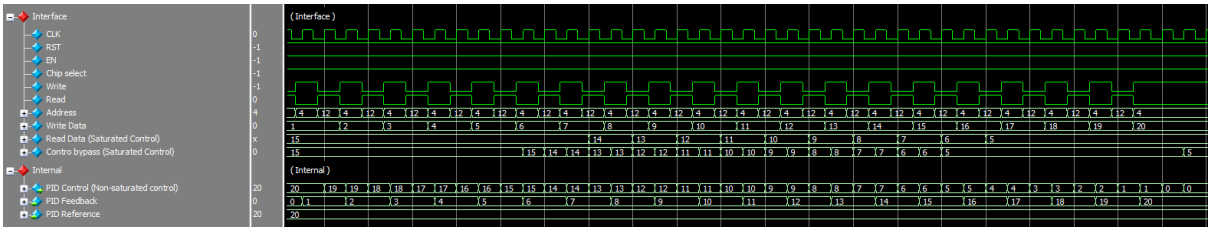


Figure 5.32: Control action comparison



Finally, it is necessary to verify that the control action is saturated correctly. The testbench has been programmed to generate a control signal ranging from 20 to 0. Then the saturation limits were set to 15 and 5. Thus, the output of the peripheral should range within the saturation limits. The results are displayed in Figure **5.33**.

Figure 5.33: PID saturation verification

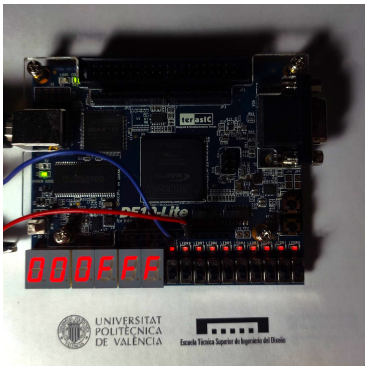


5.2.3.5 Analog to Digital Converter

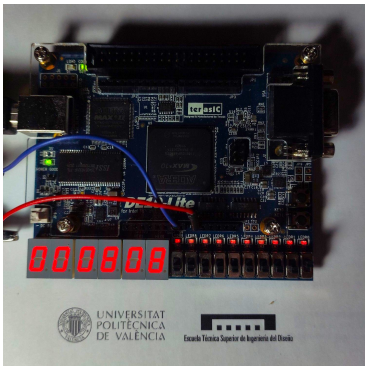
To verify the ADC, it has been used the final implementation assembly code since after the configuration of the peripherals, it performs a loop that updates the seven-segment display with the value measured in the channel one of the ADC. It has been tested with varying the voltage at the pin with a potenciometer and compared with a voltmeter. Figure **5.34** shows the results for different input voltages.

Figure 5.34: ADC verification

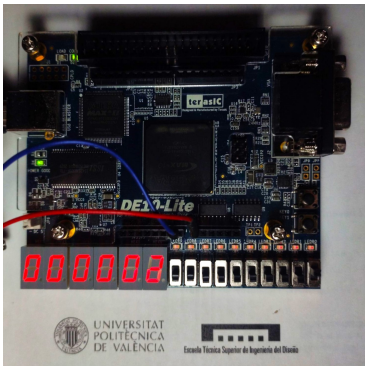
(a) Input voltage 5 V



(b) Input voltage 2.5 V



(c) Input voltage 0 V



6 Conclusions

To conclude, it can be said that the project developed has achieved its objectives. A RISC-V microcontroller has been successfully developed for the application in a hyperloop prototype. However, due to the characteristics of the microcontroller, it could be used for general control applications.

The project also accomplishes the requirements defined in Section **2** by being able to perform the control without the necessity of employing the CPU for the operation. It has also shown how programmable logic devices can replace general-purpose microcontrollers by demonstrating their versatility and adaptability capacities. Also, the system can be escalated by replicating the hardware designed to achieve more control outputs or generate more controllers connected to the same CPU.

6.1 Future lines

The project only glimpses the possibilities of using programmable logic devices in control and hyperloop applications.

Regarding the processor, there are multiple future lines to follow. Among them, increase the instruction set architecture by adding commonly used extensions such as the multiplication and division extension or float support. A significant improvement to be developed is to implement the pipeline structure with hardware hazard handling.

On the control side, future lines could be developing a PID controller able to execute multiple control loops by modifying the current registers for FIFO (first in, first out) buffers. This would be possible because the clock frequency is greater than the operating frequency of the control loops. For a clock frequency of 2 kHz, two control loops could be executed at 1 kHz by updating the values of the first control loop at one clock cycle, and then the following clock cycle would update the second control loop.

Lastly, about the hyperloop application, the current control is just a stage of the whole levitation control. It would be attractive to develop a hardware-based control for the complete levitation system.

Appendix A

Detailed peripheral memory map

Table 0.A.1: PID controller registers description

Memory addres	Read/Write	Name	Peripheral
0xC0000000	R/W	Reference	
0xC0000004	R/W	K_1	
0xC0000008	R/W	K_2	
0xC000000C	R/W	K_3	
0xC0000010	R/W	Feedback	
0xC0000014	R/W	Clk prescaler	
0xC0000018	R/W	Status	PID 1
0xC000001C	R/W	Clear	
0xC0000020	R/W	Bypass	
0xC0000024	R/W	Saturation	
0xC0000028	R/W	Upper saturation	
0xC000002C	R/W	Lower saturation	
0xC0000030	R	Control	
0xC0000034			
0xC0000038		Reserved	
0xC000003C			

0xC0000040	R/W	Reference	
0xC0000044	R/W	K_1	
0xC0000048	R/W	K_2	
0xC000004C	R/W	K_3	
0xC0000050	R/W	Feedback	
0xC0000054	R/W	Clk prescaler	
0xC0000058	R/W	Status	PID 2
0xC000005C	R/W	Clear	
0xC0000060	R/W	Bypass	
0xC0000064	R/W	Saturation	
0xC0000068	R/W	Upper saturation	
0xC000006C	R/W	Lower saturation	
0xC0000070	R	Control	
0xC0000074		Reserved	
0xC0000078	R	Channel 1	ADC
0xC000007C	R	Channel 2	
0xC0000080	R/W	Count	
0xC0000084	R/W	ARR	
0xC0000088	R/W	Start	
0xC000008C	R/W	IRQ	
0xC0000090	R/W	Prescaler	Timer
0xC0000094	R/W	Dead-time	
0xC0000098	R/W	Compare 1	
0xC000009C	R/W	Compare 2	
0xC00000A0	R/W	Output Enable	
0xC00000A4	R/W	Bypass	
0xC00000A8	R/W	Value	7-Segment
0xC00000AC	R/W	Shift	PID-Timer Link

Appendix B

LPM_MULT configuration

The appendix shows the steps to configure the multiplier module employed in the PID peripheral. The LPM_MULT IP can be generated through the IP Catalog from Quartus.

Tools > IP Catalog > Library > Basic Functions > Arithmetic > LPM_MULT

The following configuration has been employed:

Figure 0.B.1: LPM_MULT configuration screen 1

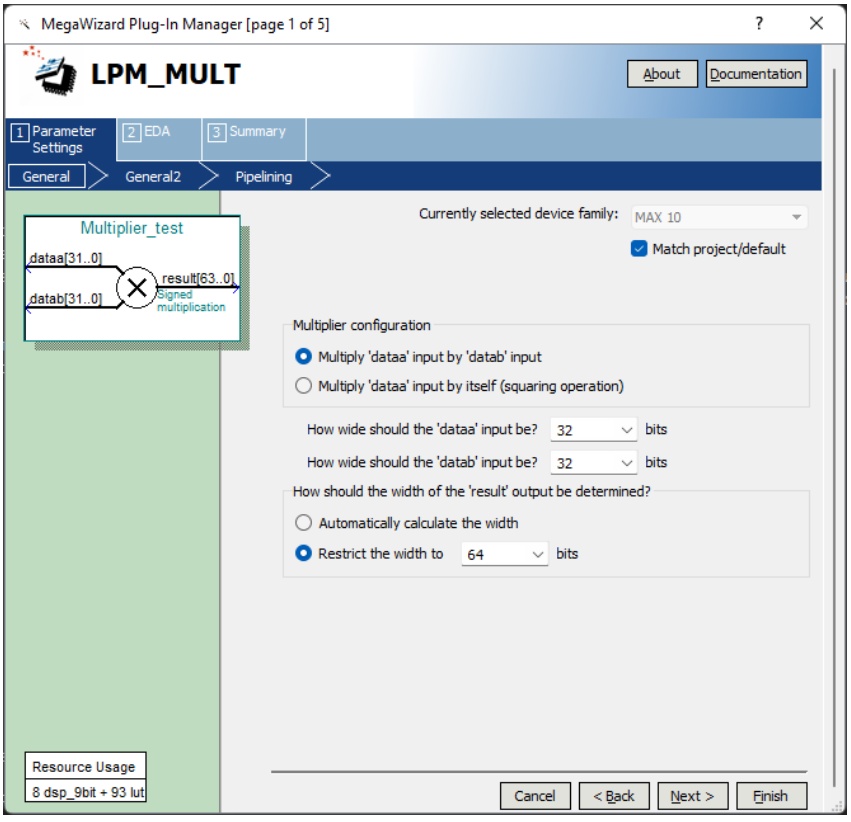


Figure 0.B.2: LPM_MULT configuration screen 2

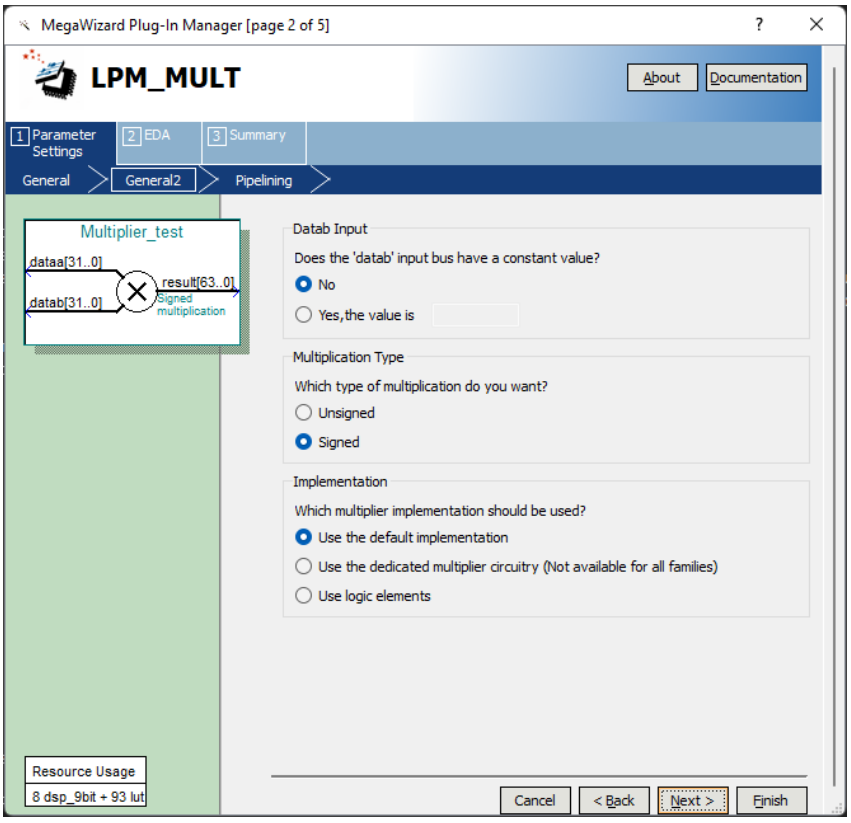


Figure 0.B.3: LPM_MULT configuration screen 3

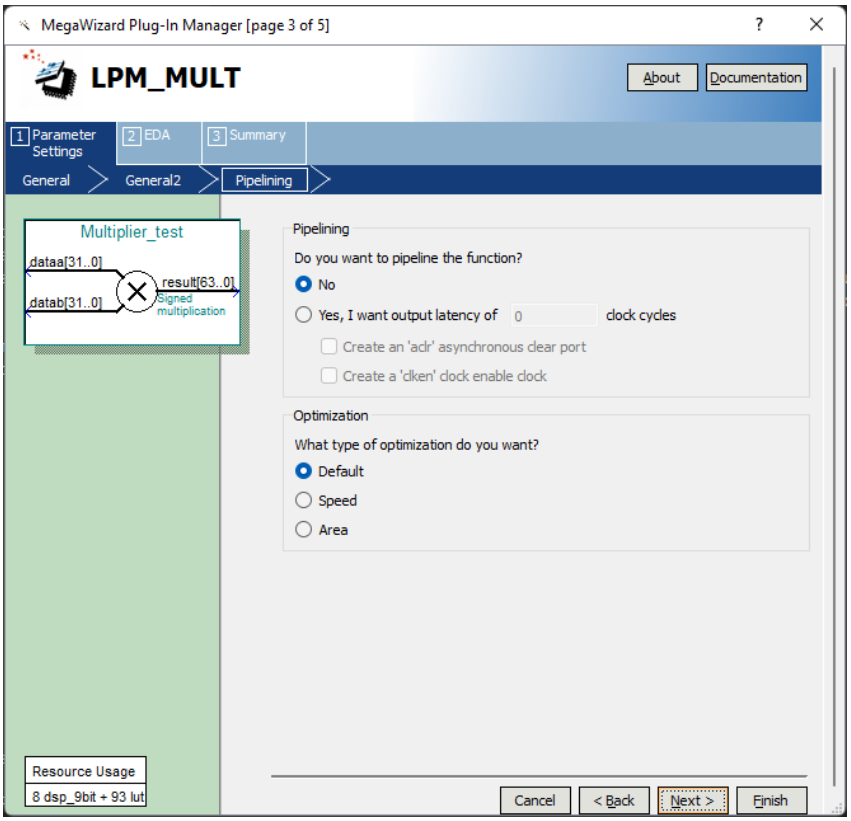


Figure 0.B.4: LPM_MULT configuration screen 4

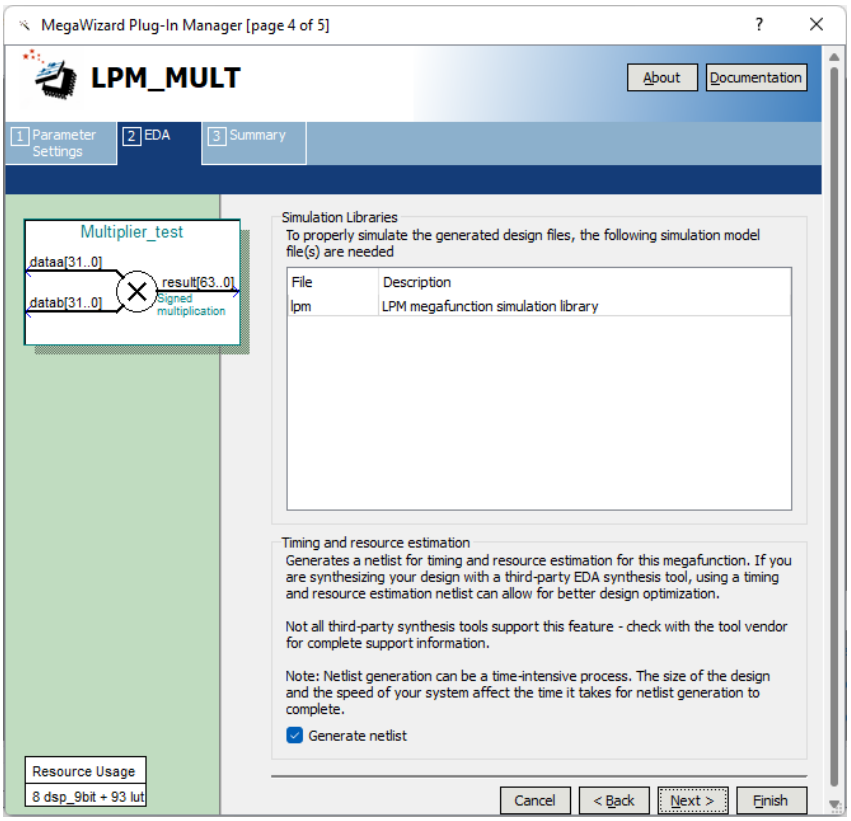
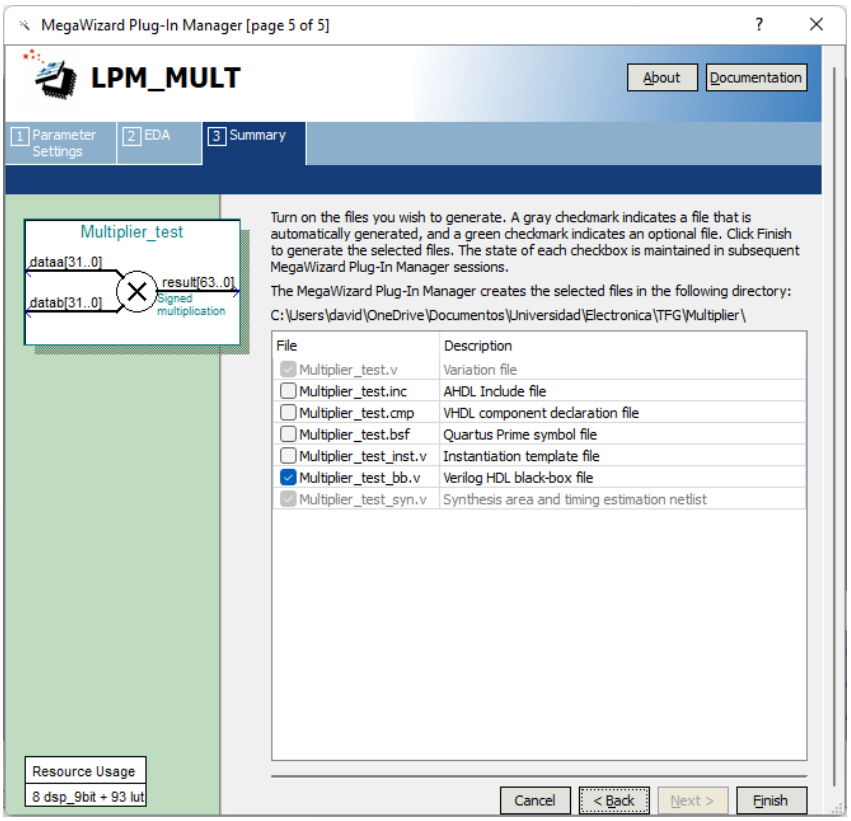


Figure 0.B.5: LPM_MULT configuration screen 5



Appendix C

Pinout

Table 0.C.1: Pinout

Node name	Direction	Location	Node name	Direction	Location
clk	Input	PIN_N5	hex3[4]	Output	PIN_C20
en	Input	PIN_B8	hex3[3]	Output	PIN_C19
hex0[6]	Output	PIN_C17	hex3[2]	Output	PIN_E21
hex0[5]	Output	PIN_D17	hex3[1]	Output	PIN_E22
hex0[4]	Output	PIN_E16	hex3[0]	Output	PIN_F21
hex0[3]	Output	PIN_C16	hex4[6]	Output	PIN_F20
hex0[2]	Output	PIN_C15	hex4[5]	Output	PIN_F19
hex0[1]	Output	PIN_E15	hex4[4]	Output	PIN_H19
hex0[0]	Output	PIN_C14	hex4[3]	Output	PIN_J18
hex1[6]	Output	PIN_B17	hex4[2]	Output	PIN_E19
hex1[5]	Output	PIN_A18	hex4[1]	Output	PIN_E20
hex1[4]	Output	PIN_A17	hex4[0]	Output	PIN_F18
hex1[3]	Output	PIN_B16	hex5[6]	Output	PIN_N20
hex1[2]	Output	PIN_E18	hex5[5]	Output	PIN_N19
hex1[1]	Output	PIN_D18	hex5[4]	Output	PIN_M20
hex1[0]	Output	PIN_C18	hex5[3]	Output	PIN_N18
hex2[6]	Output	PIN_B22	hex5[2]	Output	PIN_L18

hex2[5]	Output	PIN_C22	hex5[1]	Output	PIN_K20
hex2[4]	Output	PIN_B21	hex5[0]	Output	PIN_J20
hex2[3]	Output	PIN_A21	pwm1out[1]	Output	PIN_V10
hex2[2]	Output	PIN_B19	pwm1out[0]	Output	PIN_V9
hex2[1]	Output	PIN_A20	pwm2out[1]	Output	PIN_V8
hex2[0]	Output	PIN_B20	pwm2out[0]	Output	PIN_V7
hex3[6]	Output	PIN_E17	rst	Input	PIN_C10
hex3[5]	Output	PIN_D19			

Bibliography

Antmicro (2019). SystemVerilog Logotype.

Cerny, E., Dudani, S., Havlicek, J., and Korchemny, D. (2010). *Introduction*, pages 3–28. Springer US, Boston, MA.

Harris, S. L. (2022). *Digital design and computer architecture : RISC-V edition*. Morgan Kaufmann, Cambridge.

Koch, D., Ziener, D., and Hannig, F. (2016). *FPGA Versus Software Programming: Why, When, and How?*, pages 1–21. Springer International Publishing, Cham.

Osier-Mixon, J. (2019). Semico Forecasts Strong Growth for RISC-V. *RISC-V Foundation*.

RISC-V Foundation (2018). RISC-V Logotype.

RISC-V Foundation (n.d.). About RISC-V. Accessed on April 1, 2023.

ST Microcontroelectronics (2016). STM32 cross-series timer overview. Technical report, ST Microcontrollers. 6th release.

Terasic (n.d.). Terasic DE10-Lite.

Waterman, A. and Asanović, K. (2019). *The RISC-V Instruction Set Manual*, volume Volume I: Unprivileged ISA. RISC-V Foundation, 20191213 edition.

Xilinx® (n.d.). Xilinx Virtex UltraScale+ VU19P FPGA product brief. Accessed on April 2, 2023.

Zhang, P. (2010). Chapter 1 - industrial control systems. In *Advanced Industrial Control Technology*, pages 3–40. William Andrew Publishing, Oxford.

Intentionally blank page