UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Desarrollo de un procesador RISC-V optimizado para aplicaciones de control para ser utilizado en el sistema de levitación de Hyperloop

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Ramón Alamán, David

Tutor/a: Olguín Pinatti, Cristian Ariel

Cotutor/a: Monzó Ferrer, José María

CURSO ACADÉMICO: 2022/2023

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

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

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

Abstract . . . II
Contents . . . V
List of Figures . . . X
List of Tables . . . XIII
List of Code snippets . . . XV
Acronyms & Initials . . . XVIII

I Memory report **1**
1 Scope . . . 2
1.1 RISC-V . . . 2
1.2 Programmable Logic Devices . . . 3
1.3 SystemVerilog . . . 4
1.4 Hyperloop concept & Hyperloop UPV . . . 4
2 Needs study . . . 8
3 Alternative solutions . . . 9
4 Development . . . 12
4.1 RISC-V . . . 12
4.1.1 Instruction Set Architecture . . . 13
4.1.1.1 Registers . . . 13
4.1.1.2 Instructions . . . 15
4.2 Single-cycle processor . . . 17
4.2.1 Datapath . . . 18
4.2.1.1 Register file . . . 18
4.2.1.2 Immediate generator . . . 19
4.2.1.3 Arithmetic Logic Unit . . . 21
4.2.1.4 Data memory . . . 24
4.2.1.5 Instruction memory . . . 25
4.2.2 Control logic . . . 26
4.2.2.1 Control unit . . . 26
4.2.2.2 ALU control unit . . . 28

4.2.2.3 Branch logic 29
4.2.3 Analysis & Synthesis 31
4.3 Multicycle processor 32
4.3.1 Datapath 35
4.3.1.1 Instruction and Data memory 35
4.3.2 Control logic 36
4.3.2.1 Control unit 36
4.3.2.2 Branch logic 40
4.3.3 Memory bus & Peripherals 42
4.3.3.1 Memory Controller 45
4.3.3.2 PID-Timer Link 47
4.3.3.3 Seven-segments decoder 48
4.3.3.4 Analog to Digital Converter 49
4.3.3.5 Timer 51
4.3.3.6 PID controller 54
4.3.4 Analysis & Synthesis 58
4.4 Current controller 59
4.4.1 Controller design 60
4.4.2 Software development 63
4.4.2.1 PID configuration 64
4.4.2.2 Timer configuration 64
5 Verification 66
5.1 Single-cycle 66
5.1.1 Modules verification 67
5.1.2 Processor verification 70
5.1.2.1 Instruction set verification 71
5.1.2.2 Fibonacci sequence 76
5.1.2.3 Bubble sort 76
5.2 Multicycle 77
5.2.1 Processor 77
5.2.2 Memory bus 78
5.2.3 Peripherals 80
5.2.3.1 PID-Timer link 80
5.2.3.2 Seven-Segments decoder 81
5.2.3.3 Timer 82
5.2.3.4 PID controller 84
5.2.3.5 Analog to Digital Converter 85
6 Conclusions 86

6.1 Future lines . . . 86
Annexes . . . 87
A Detailed peripheral memory map . . . 87
B LPM_MULT configuration . . . 89
C Pinout . . . 94
D Sustainable Development Goals . . . 96
Bibliography . . . 98

II Plans & drawings **99**
1 Scope . . . 100

III Written specifications **101**
1 Scope . . . 102
2 Materials conditions . . . 102
2.1 Electronic devices . . . 102
2.2 Software . . . 102
3 Test prior to commissioning . . . 103

IV Project schedule **104**
1 Gantt diagram . . . 105

V Budget **106**
1 Materials . . . 107
2 Labour . . . 108
3 Proyect costs . . . 108

VI Development files **109**
1 Development . . . 110
1.1 Common modules . . . 110
1.2 Single-cycle . . . 121
1.2.1 Datapath . . . 121
1.2.1.1 Register file . . . 121
1.2.1.2 Immediate generator . . . 122
1.2.1.3 Arithmetic Logic Unit . . . 123
1.2.1.4 Data memory . . . 125
1.2.1.5 Instruction memory . . . 126

1.2.2 Control logic . . . 127
1.2.2.1 Control unit . . . 127
1.2.2.2 Branch logic . . . 130
1.2.2.3 ALU Control . . . 131
1.2.3 Processor implementation . . . 132
1.3 Multicycle . . . 136
1.3.1 Datapath . . . 136
1.3.1.1 Memory . . . 136
1.3.2 Control logic . . . 137
1.3.2.1 Branch logic . . . 147
1.3.3 Multicycle processor . . . 148
1.3.4 Memory bus . . . 152
1.3.4.1 Memory controller . . . 152
1.3.4.2 PID-Timer link . . . 155
1.3.4.3 Seven-segment decoder . . . 157
1.3.4.4 Analog to Digital Converter . . . 160
1.3.4.5 Timer . . . 161
1.3.4.6 PID Controller . . . 165
1.3.5 Multicycle microcontroller . . . 170
2 Verification . . . 176
2.1 Single-cycle . . . 176
2.1.1 Datapath . . . 176
2.1.1.1 Register file . . . 176
2.1.1.2 Immediate generator . . . 179
2.1.1.3 Arithmetic Logic Unit . . . 181
2.1.1.4 Data memory . . . 183
2.1.1.5 Instruction memory . . . 185
2.1.2 Control logic . . . 188
2.1.2.1 Control unit . . . 188
2.1.2.2 Branch logic . . . 191
2.1.2.3 ALU Control . . . 194
2.1.3 Processor . . . 199
2.2 Multicycle . . . 206
2.2.1 Multicycle microcontroller . . . 206
2.2.2 Memory bus . . . 207
2.2.3 Peripherals . . . 208
2.2.3.1 PID-Timer link . . . 208
2.2.3.2 Seven-segment decoder . . . 210

2.2.3.3 Timer 212

2.2.3.4 PID controller 215

List of Figures

1.1.1 RISC-V Logotype, (RISC-V Foundation, 2018) . . . 3
1.1.2 SystemVerilog Logotype, (Antmicro, 2019). . . . 4
1.1.3 Hyperloop concept by Hyperloop UPV, (Hyperloop UPV) . . . 5
1.1.4 Auran and infrastructure, (Hyperloop UPV) . . . 6
1.1.5 Levitation and guiding system electromagnets, (Hyperloop UPV) . . . 7
1.3.1 Terasic DE10-Lite, (Terasic, nd) . . . 10
1.4.1 Single-cycle processor . . . 18
1.4.2 Register file interface . . . 19
1.4.3 Immediate generator interface . . . 20
1.4.4 Arithmetic Logic Unit interface . . . 21
1.4.5 Arithmetic Logic Unit internal logic . . . 23
1.4.6 Data memory interface . . . 24
1.4.7 Instruction memory interface . . . 25
1.4.8 Single-cycle datapath . . . 26
1.4.9 Control unit interface . . . 27
1.4.10 ALU control inteface . . . 28
1.4.11 Control unit interface . . . 30
1.4.12 Analysis & Synthesis . . . 32
1.4.13 Single-cycle and multicycle time diagram . . . 33
1.4.14 Multicycle processor . . . 34
1.4.15 Memory interface . . . 35
1.4.16 Memory interface . . . 37
1.4.17 Control state machine diagram . . . 38
1.4.18 Removed components from single-cycle implementation . . . 41
1.4.19 Branch logic interface . . . 42
1.4.20 Memory bus . . . 44
1.4.21 Memory map . . . 45
1.4.22 Memory controller interface . . . 46
1.4.23 PID-Timer Link interface . . . 47
1.4.24 PID-Timer Link interface . . . 47
1.4.25 Memory controller interface . . . 48

1.4.26 ADC - IP Parameter Editor 49
1.4.27 Memory controller interface 50
1.4.28 Timer interface. 51
1.4.29 Timer interface. 52
1.4.30 PID circuit diagram. 56
1.4.31 PID controller interface 58
1.4.32 Pin Planner. 58
1.4.33 Analysis & Synthesis 59
1.4.34 Simulink diagrams 60
1.4.34 Continuous PID simulation. 62
1.4.35 PID circuit Simulink block. 63
1.4.36 Discrete PID simulation 63
1.5.1 Verification error log. 67
1.5.2 Register file verification results. 67
1.5.3 Immediate generator verification results. 68
1.5.4 ALU verification results 68
1.5.5 Data memory verification results 69
1.5.6 Program memory verification results 69
1.5.7 Control unit verification results 69
1.5.8 ALU control verification results 70
1.5.9 Branch logic verification results 70
1.5.10 Types I, S verification waveforms. 72
1.5.11 Type I, S register file waveforms. 72
1.5.12 Types R, U, J verification waveforms 73
1.5.13 Types R, U, J register file waveforms 74
1.5.14 Type B register file waveforms 75
1.5.15 Fibonacci sequence 76
1.5.16 Bubble sort 77
1.5.17 Fibonacci sequence 78
1.5.18 Bubble sort 78
1.5.19 PID 1 registers verification. 79
1.5.20 PID 2 registers verification. 79
1.5.21 ADC registers verification 79
1.5.22 Timer registers verification 80
1.5.23 7-segment and PID-Timer link registers verification. 80
1.5.24 PID-Timer link verification 80
1.5.25 Seven-segment decoder verification 81
1.5.26 Seven-segment decoder verification FPGA. 81

1.5.27 Timer counter limit verification 82
1.5.28 Timer prescaler verification 82
1.5.29 Timer outputs verification 83
1.5.30 Timer bypass and dead-time verification 83
1.5.31 Golden model simulation 84
1.5.32 Control action comparison 84
1.5.33 PID saturation verification. 85
1.5.34 ADC verification 85

1.B.1 LPM_MULT configuration screen 1 89
1.B.2 LPM_MULT configuration screen 2 90
1.B.3 LPM_MULT configuration screen 3 91
1.B.4 LPM_MULT configuration screen 4 92
1.B.5 LPM_MULT configuration screen 5 93

4.1.1 Gantt diagram 105

List of Tables

1.2.1 Design requirements . . . 8
1.4.1 Standard ISA extensions suffixes . . . 13
1.4.2 RISC-V register convention, (Waterman and Asanović, 2019) . . . 14
1.4.3 RISC-V register convention, (Harris, 2022) . . . 15
1.4.4 RV32I Instructions, (Harris, 2022) . . . 15
1.4.5 Immediate generator encoding . . . 20
1.4.6 Instruction type OpCodes . . . 20
1.4.7 Operations performed in each instruction . . . 22
1.4.8 Arithmetic Logic Unit control encoding . . . 24
1.4.9 Instructions OpCodes . . . 27
1.4.10 Control unit outputs . . . 28
1.4.11 ALU Control logic . . . 29
1.4.12 Branch logic . . . 31
1.4.13 Memory module configuration . . . 36
1.4.14 Control unit module outputs . . . 39
1.4.15 Branch logic . . . 42
1.4.16 Memory bus interfaces . . . 43
1.4.17 Base addresses . . . 46
1.4.18 PID-Timer link register description . . . 48
1.4.19 Seven-segment decoder register description . . . 49
1.4.20 ADC register description . . . 50
1.4.21 Dead-time calculation . . . 53
1.4.22 Timer registers description . . . 53
1.4.23 PID controller registers description . . . 57
1.5.1 Bubble sort execution . . . 77

1.A.1 PID controller registers description . . . 87

1.C.1 Pinout . . . 94

1.D.1 Degree to which the project relates to the Sustainable Development Goals . . . 96

5.1.1 Unitary cost calculation . . . 107

5.1.2 Materials costs 107
5.2.1 Labour unitary prices 108
5.2.2 Labour costs 108
5.3.1 Project costs 108

List of Code snippets

1.4.1 Load compiled code . . . 25
1.4.2 State machine implementation . . . 40
1.5.3 Types I, S verification . . . 71
1.5.4 Types R, U, J verification . . . 72
1.5.5 Type B verification . . . 74
1.5.6 Memory bus verification . . . 79
6.1.1 Adder . . . 110
6.1.2 Register . . . 111
6.1.3 2-input register . . . 112
6.1.4 Full adder . . . 113
6.1.5 2:1 multiplexer . . . 113
6.1.6 3:1 multiplexer . . . 114
6.1.7 4:1 multiplexer . . . 115
6.1.8 6:1 multiplexer . . . 116
6.1.9 7:1 multiplexer . . . 117
6.1.10 10:1 multiplexer . . . 118
6.1.11 Prescaler . . . 119
6.1.12 Zero extender . . . 120
6.1.13 Register file . . . 121
6.1.14 Immediate generator . . . 122
6.1.15 Arithmetic Logic Unit . . . 123
6.1.16 Data memory . . . 125
6.1.17 Instruction memory . . . 126
6.1.18 Control unit . . . 127
6.1.19 Branch logic . . . 130
6.1.20 ALU Control . . . 131
6.1.21 RV32I - Single cycle . . . 132
6.1.22 Memory . . . 136
6.1.23 Control unit . . . 137

6.1.24 Branch logic 147
6.1.25 RV32I - Multicycle processor 148
6.1.26 Memory controller 152
6.1.27 PID-Timer link 155
6.1.28 Seven-segment decoder 157
6.1.29 Analog to Digital Converter 160
6.1.30 PWM Generator 161
6.1.31 Timer 162
6.1.32 PID Controller 165
6.1.33 PID peripheral 167
6.1.34 RV32I - Mulcicycle microcontroller 170
6.1.35 Current controller assembly 175
6.2.36 Register file testbench 176
6.2.37 Register file verification vector 1 178
6.2.38 Register file verification vector 2 178
6.2.39 Register file verification vector 3 178
6.2.40 Immediate generator testbench 179
6.2.41 Immediate generator verification vector 180
6.2.42 ALU testbench 181
6.2.43 ALU verification vector 182
6.2.44 Data memory testbench 183
6.2.45 Data memory verification vector 184
6.2.46 Instruction memory testbench 185
6.2.47 Intruction memory verification instructions file 187
6.2.48 Intruction memory verification vector 187
6.2.49 Control unit testbench 188
6.2.50 Control unit verification vector 190
6.2.51 Branch logic testbench 191
6.2.52 Branch logic verification vector 193
6.2.53 ALU control testbench 194
6.2.54 ALU control verification vector 196
6.2.55 RV32I single-cycle processor testbench 199
6.2.56 Types I, S verification 201
6.2.57 Types R, U, J verification 202
6.2.58 Type B verification 203
6.2.59 Fibonacci sequence 204
6.2.60 Bubble sort 205
6.2.61 RV32I Multicycle microcontroller testbench 206

6.2.62 Memory bus verification assembly 207
6.2.63 PID-Timer link testbench 208
6.2.64 Seven-segment decoder testbench 210
6.2.65 Seven-segment display 211
6.2.66 Timer testbench 212
6.2.67 PID controller testbench 215
6.2.68 PID controller MATLAB simulation script 220
6.2.69 PID Peripheral testbench 222

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

Part I

Memory report

1 Scope 2
2 Needs study 8
3 Alternative solutions 9
4 Development 12
5 Verification 66
6 Conclusions 86
Annexes 87
A Detailed peripheral memory map 87
B LPM_MULT configuration 89
C Pinout 94
D Sustainable Development Goals 96
Bibliography 98

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 1.1.1: 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 1.1.2: SystemVerilog Logotype, (Antmicro, 2019)

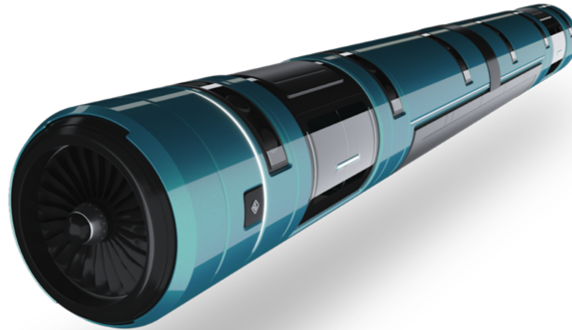
1.4 Hyperloop concept & Hyperloop UPV

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 1.1.3: Hyperloop concept by Hyperloop UPV, (Hyperloop UPV)



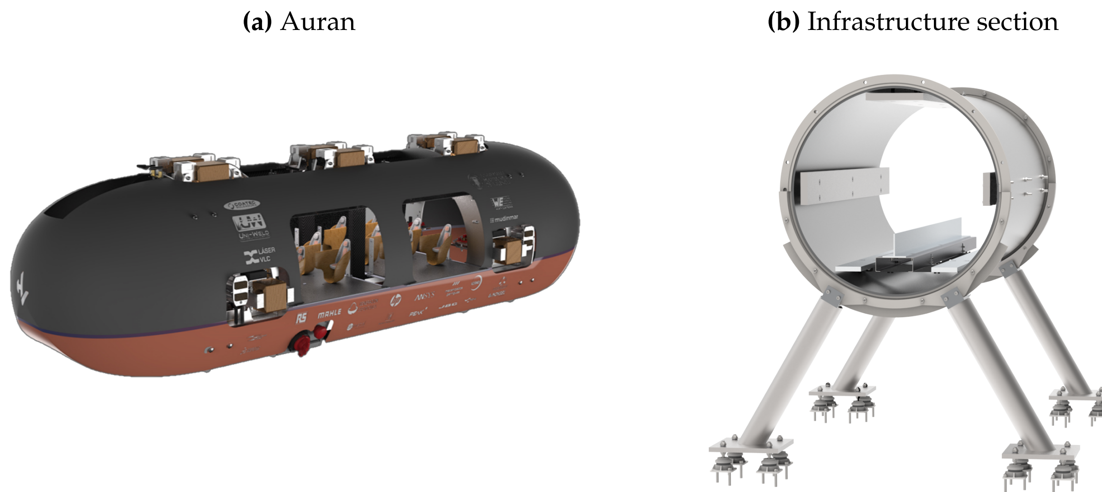
In 2015 Spacex began organising the Hyperloop Pod Competition. That led to the creation of a team at the Polytechnic University of Valencia, Hyperloop UPV (in which I have been involved since 2021 as a firmware and hardware engineer), to participate in the competition. Since then, the team has participated 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 **1.1.4a**). 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 **1.1.4b**) without any contact with the infraestructure.

Figure 1.1.4: 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.

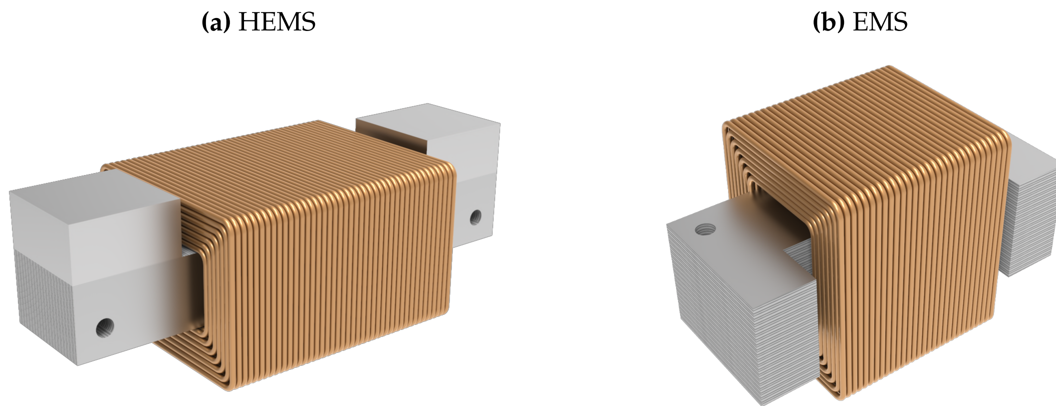
Due to the aforementioned situation, a control system must be implemented where the actuators are the electromagnet, which modifies the magnetic field to return to the equilibrium distance. Thus, a permanent magnetic field generates the force to suspend the vehicle, and the energy is employed only to 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 **1.1.5a**. 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 **1.1.5b**.

Figure 1.1.5: 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 **1.2.1**

Table 1.2.1: 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.

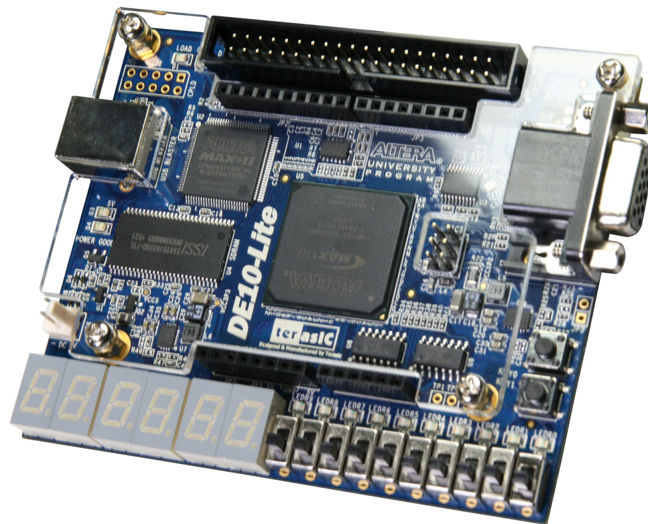
Regarding the technical criteria, both the microcontroller-based and the programmable logic-based solutions have their benefits and drawbacks. On the one hand, MCUs consume less power than PLDs. This fact makes them more suitable for battery-powered applications, such as hyperloop.

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 1.3.1: Terasic DE10-Lite, (Terasic, nd)



The device selected for this project is the Intel MAX10 10M50DAF484C7G FPGA (Figure **1.3.1** 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 **VI**. 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 **VI**.

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 **1.4.1**.

Table 1.4.1: 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 **1.4.2**).

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 1.4.2: 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 **1.4.3** depicts the format of the instructions.

Table 1.4.3: 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 **1.4.4** displays the instruction along with its format, and the operation it performs.

Table 1.4.4: 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 **1.4.4** 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

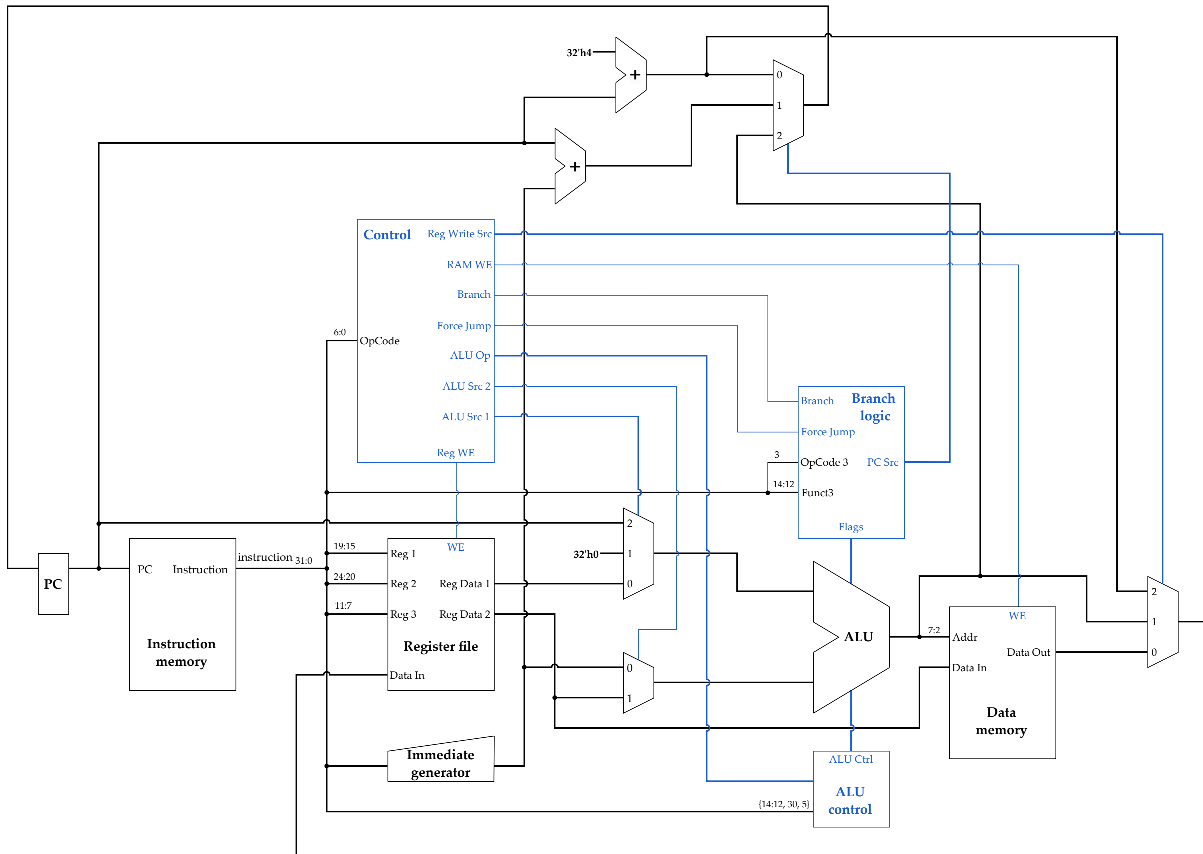
The main characteristic of a single-cycle processor is that it executes a complete instruction each cycle. Consequently, the processor mainly implements combinational logic, which is responsible for updating the processor state (register file, program counter and data memory) before the next active clock 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 **1.4.1**, 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 1.4.1: Single-cycle processor



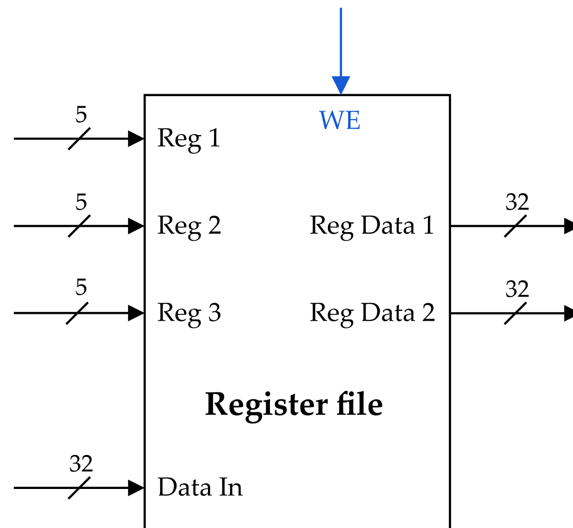
4.2.1 Datapath

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 **1.4.2**. 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 1.4.2: 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 **1.4.3**). The immediate generator behaviour is shown in Table **1.4.5**.

Figure 1.4.3: Immediate generator interface

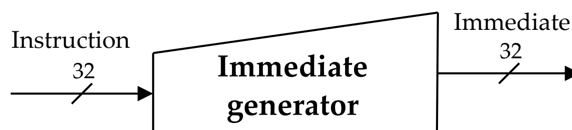


Table 1.4.5: Immediate generator encoding

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 **1.4.5** 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 **1.4.6**, each *OpCode* is shown with its corresponding instruction type.

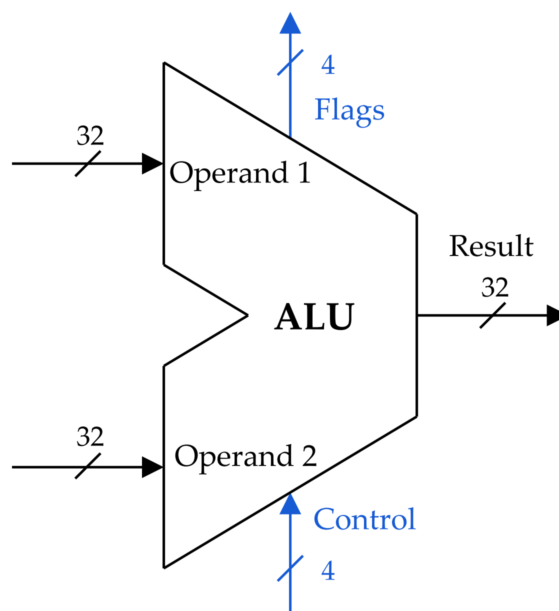
Table 1.4.6: Instruction type OpCodes

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 **1.4.4**.

Figure 1.4.4: 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 **1.4.7**, it is specified which arithmetic or logical operation demands each instruction.

Table 1.4.7: Operations performed in each instruction

Instruction	Operation	Instruction	Operation	Instruction	Operation
lw	$+$	addi	$+$	slli	$<<$
slti	$A < B$	sltiu	$A < B$	xori	$\wedge$
srli	$>>$	srai	$>>>$	ori	$\vert$
andi	$\&$	auipc	$+$	sw	$+$
add	$+$	sub	$-$	sll	$<<$
slt	$A < B$	sltu	$A < B$	xor	$\wedge$
srl	$>>$	sra	$>>>$	or	$\vert$
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 **1.4.5**.

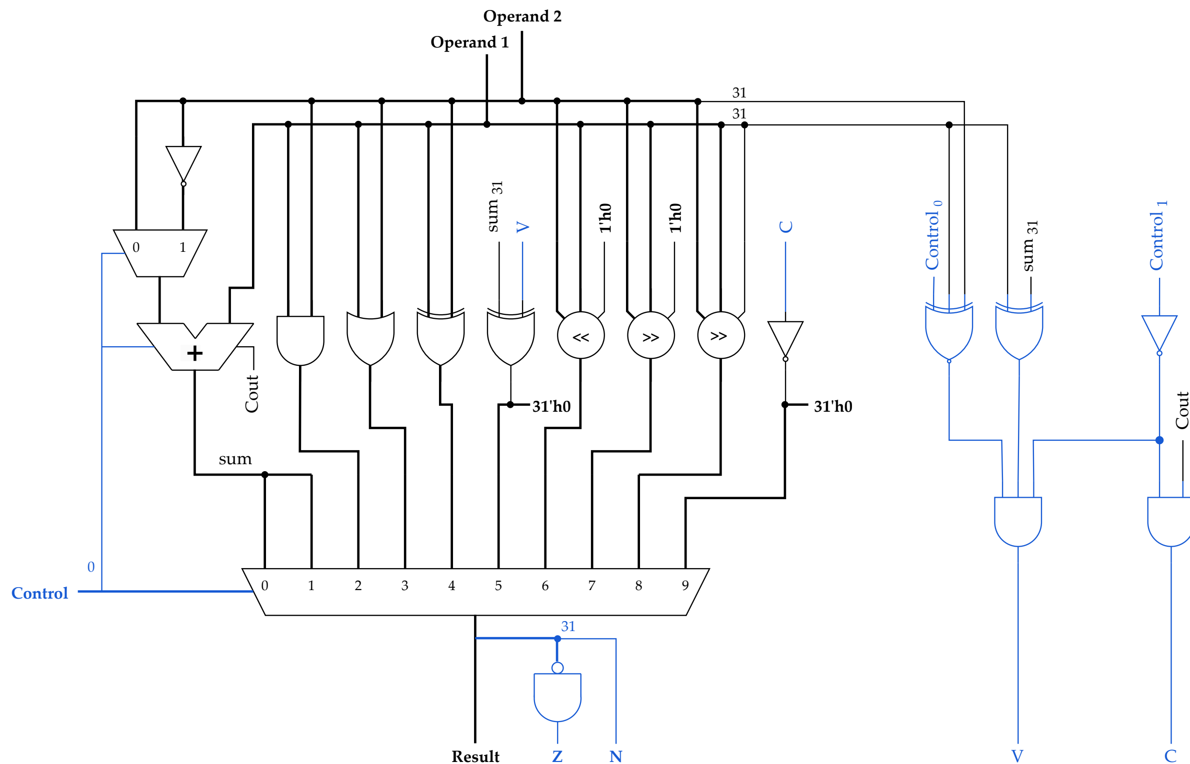
It is important to remark that to reduce the hardware necessary to implement the ALU, the addition and subtraction operations have been implemented using a single full adder circuit. Thanks to the 2's complement that specifies that $\overline{B} + 1 = -B$, it is possible to invert the second operand 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 1.4.5: 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∧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 **1.4.8** shows the requirements for each operation and the final encoding selected.

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

Table 1.4.8: 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

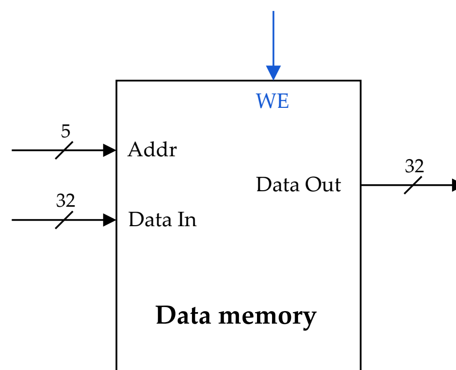
4.2.1.4 Data memory

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 **1.4.6** shows data memory interface.

Figure 1.4.6: 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 **1.4.1**). As previously mentioned, the memory must feature asynchronous reading.

Code snippet 1.4.1: Load compiled code

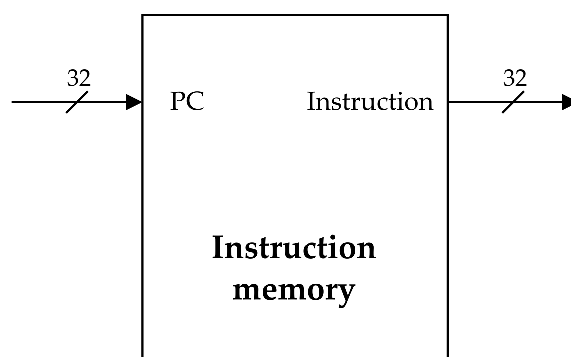
```
logic[31:0] memory[DEPTH-1:0];

initial begin
    $readmemh("compiledcode.hex", memory);
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 **1.4.7** shows instruction memory interface.

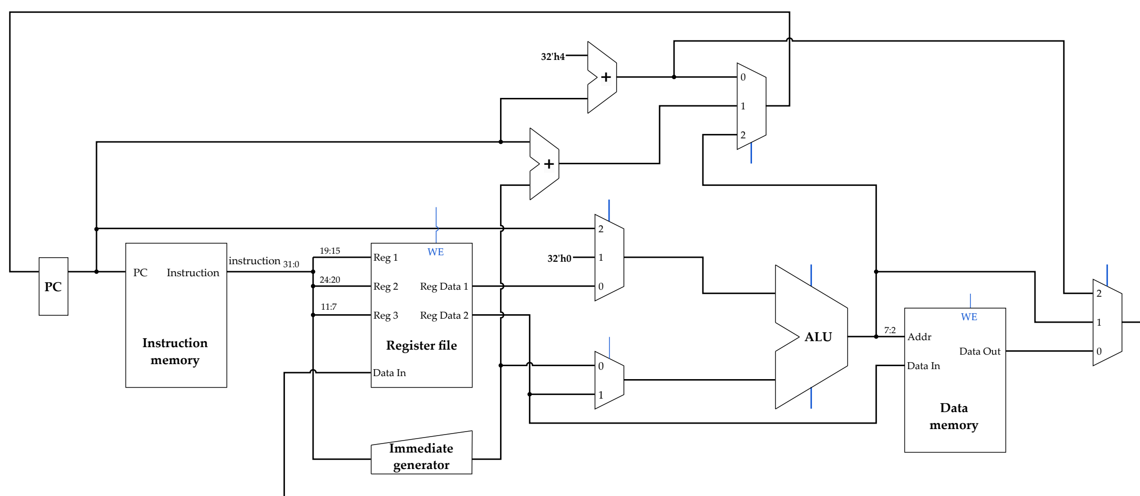
Figure 1.4.7: Instruction memory interface



4.2.2 Control logic

The main building blocks of the datapath have been explained in the previous sections. However, as shown in Figure **1.4.8**, 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 1.4.8: Single-cycle datapath

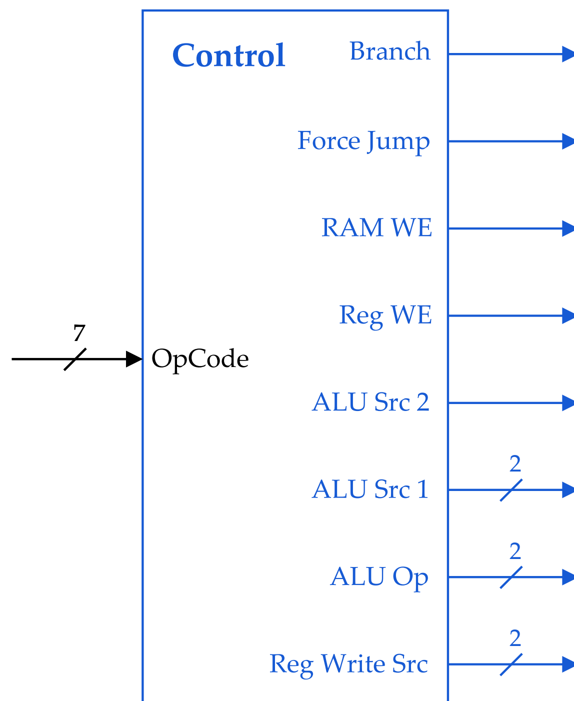


4.2.2.1 Control unit

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 **1.4.9** depicts the module interface.

Figure 1.4.9: Control unit interface



The "OpCode" determines the type of the instruction being executed. Table **1.4.9** shows the corresponding group of instructions per each "OpCode".

Table 1.4.9: 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 **1.4.10** shows the values of each output concerning the "OpCode".

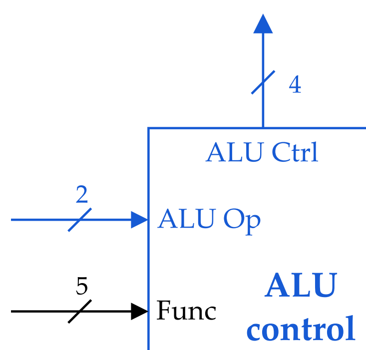
Table 1.4.10: Control unit outputs

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 **1.4.10**).

Figure 1.4.10: ALU control inteface



With the defined inputs, the ALU control unit determines the "ALU op" according to Table **1.4.11**.

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 1.4.11: 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	$\wedge$
	$1010X_2$	0111_2	$>>$
	$1011X_2$	1000_2	$>>>$
	$110XX_2$	0011_2	$\vert$
	$111XX_2$	0010_2	&
01_2	N/A	0001_2	−
10_2	N/A	0000_2	+

4.2.2.3 Branch logic

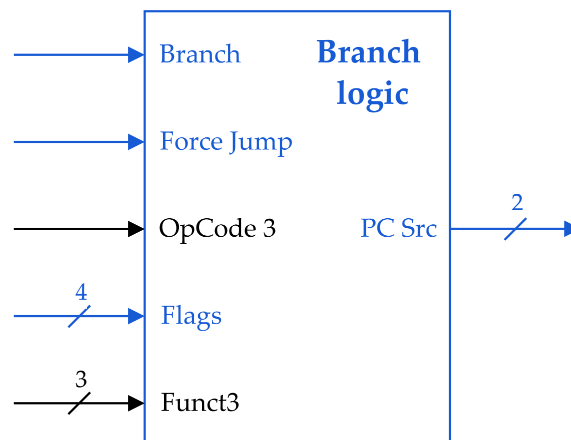
The program counter stores the value of the address where the current instruction is placed in the program memory. This section is aimed to define how the value of the PC changes, that is, how jumps between instructions are performed.

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 1.4.11: 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 **1.4.11**). The logic underlying this module is depicted in Table **1.4.12**.

Table 1.4.12: 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	$\{0, N \land V\}_2$	blt
Branch	101_2	$\{0, \overline{N \land V}\}_2$	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 **1.4.12**, 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 1.4.12: 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 %)

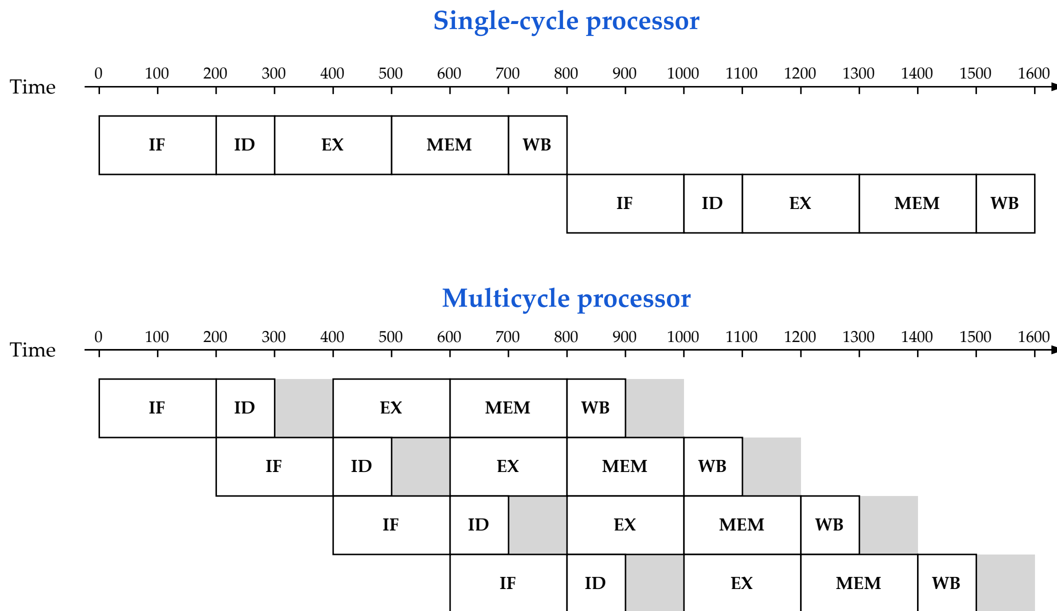
4.3 Multicycle processor

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 **1.4.13** 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 **1.4.13** 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 1.4.13: 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.

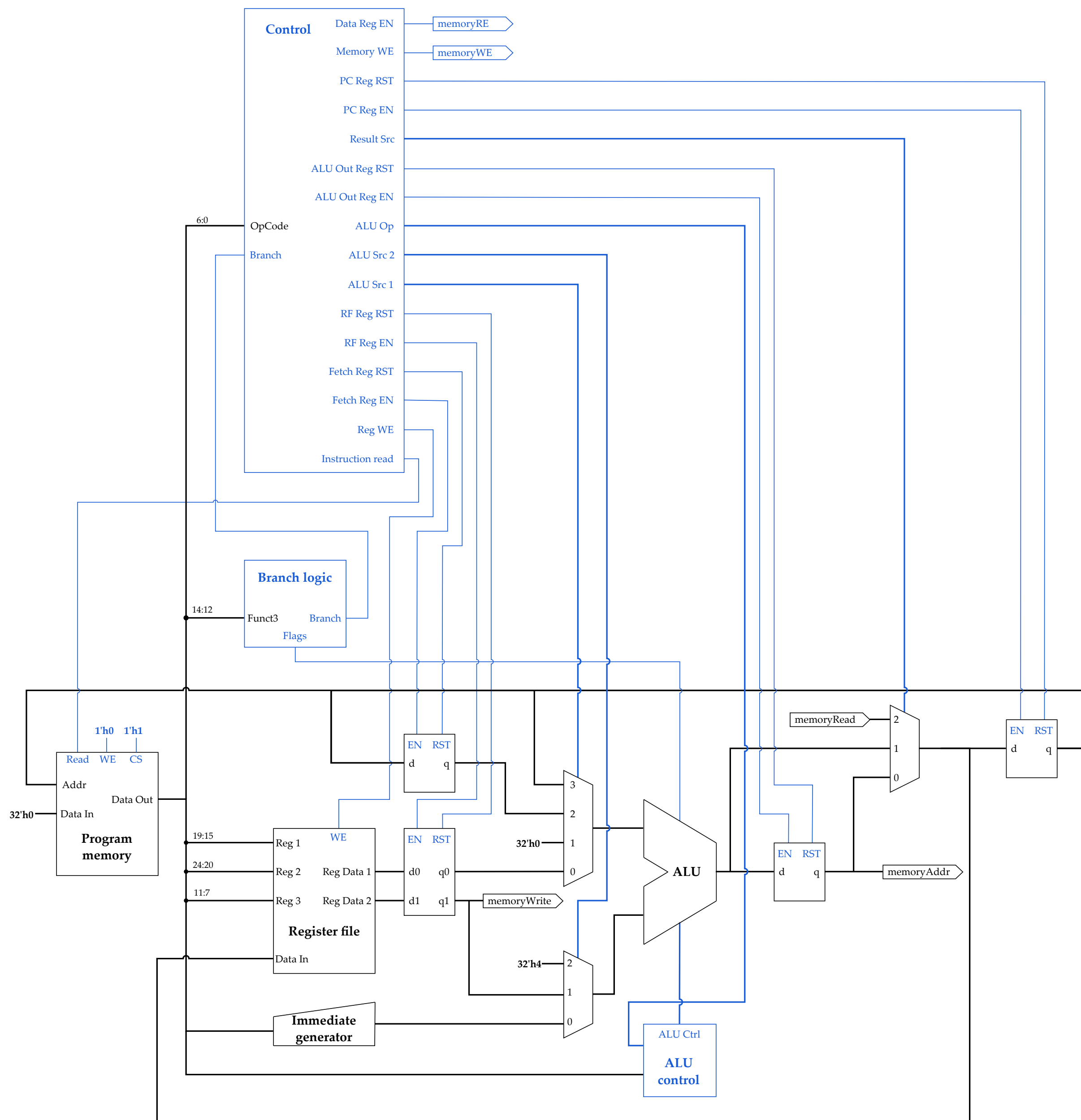
These hazards can be controlled either by software or hardware. Concerning software solutions, it is left to the programmer to manage the hazards by modifying the order of operations or adding `nop`[9] instruction whenever a conflict is detected. On the other hand, hardware solutions involve some techniques:

- 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 **1.4.14** shows the scheme of the processor, the datapath is drawn in black and the control in blue.

Figure 1.4.14: Multicycle processor



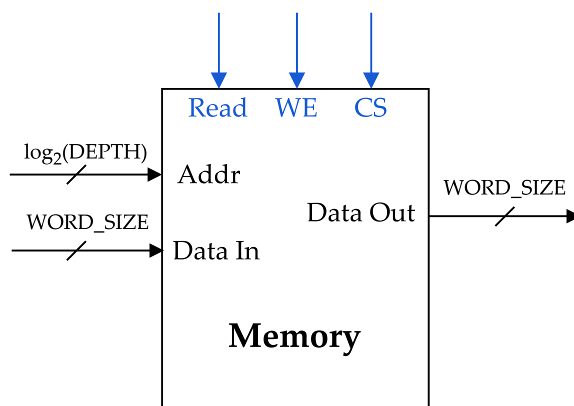
4.3.1 Datapath

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 **1.4.15**). 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 1.4.15: Memory interface



The RAM or ROM configuration is generated by configuring the module inputs according to the values in Table **1.4.13**.

Table 1.4.13: 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

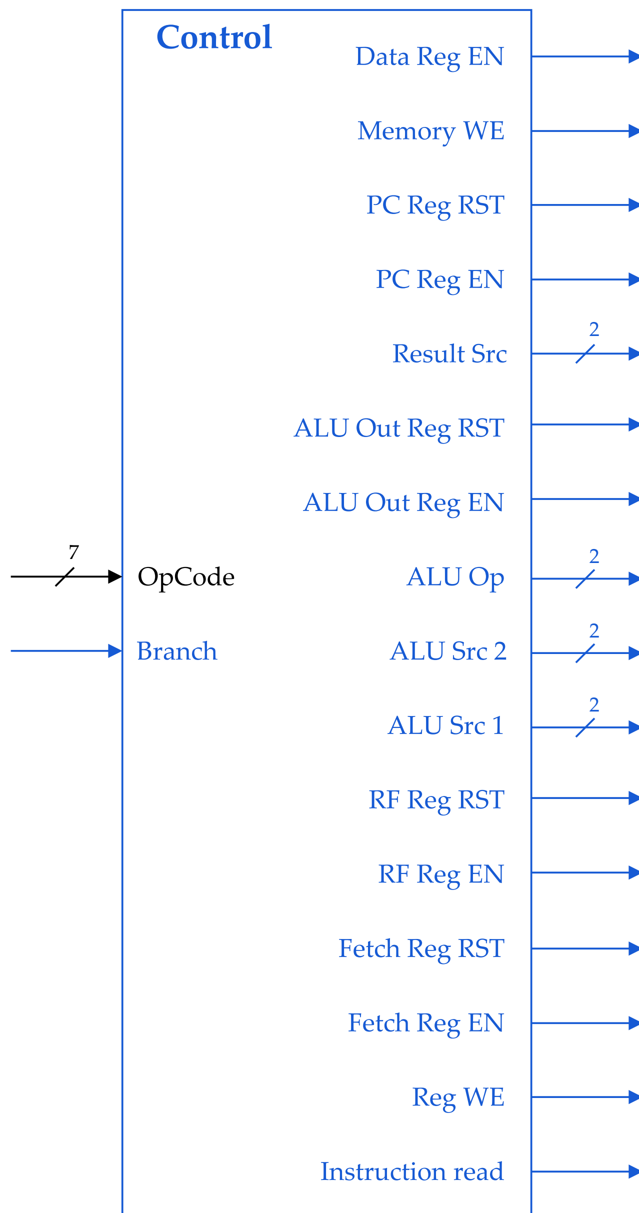
4.3.2 Control logic

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 **1.4.16** shows the control unit module interface).

Figure 1.4.16: 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 **1.4.17**.

Figure 1.4.17: Control state machine diagram

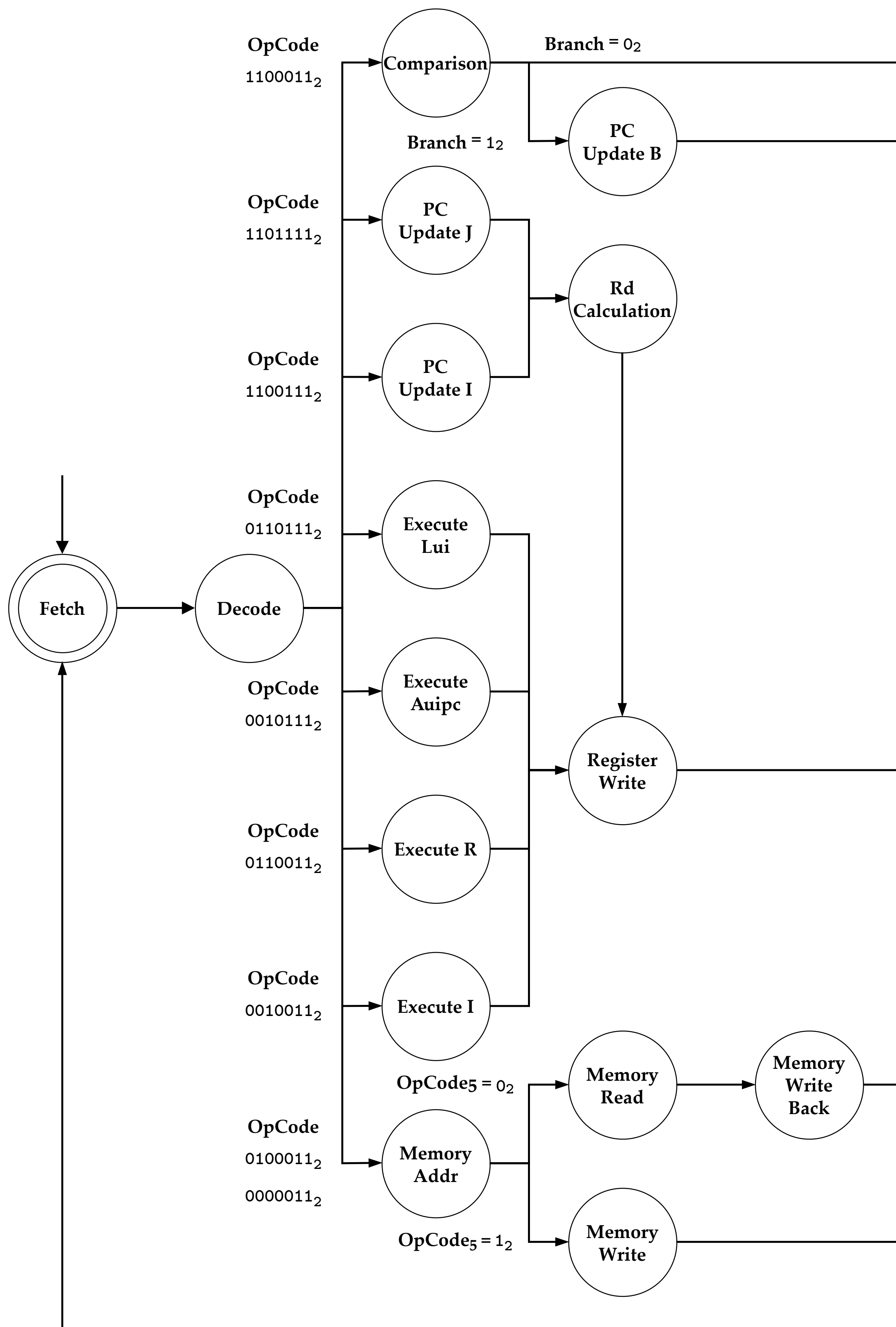


Table 1.4.14: Control unit module outputs

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 **1.4.14** 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 **1.4.2** shows the basic scheme of the SystemVerilog implementation.

Code snippet 1.4.2: 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
```

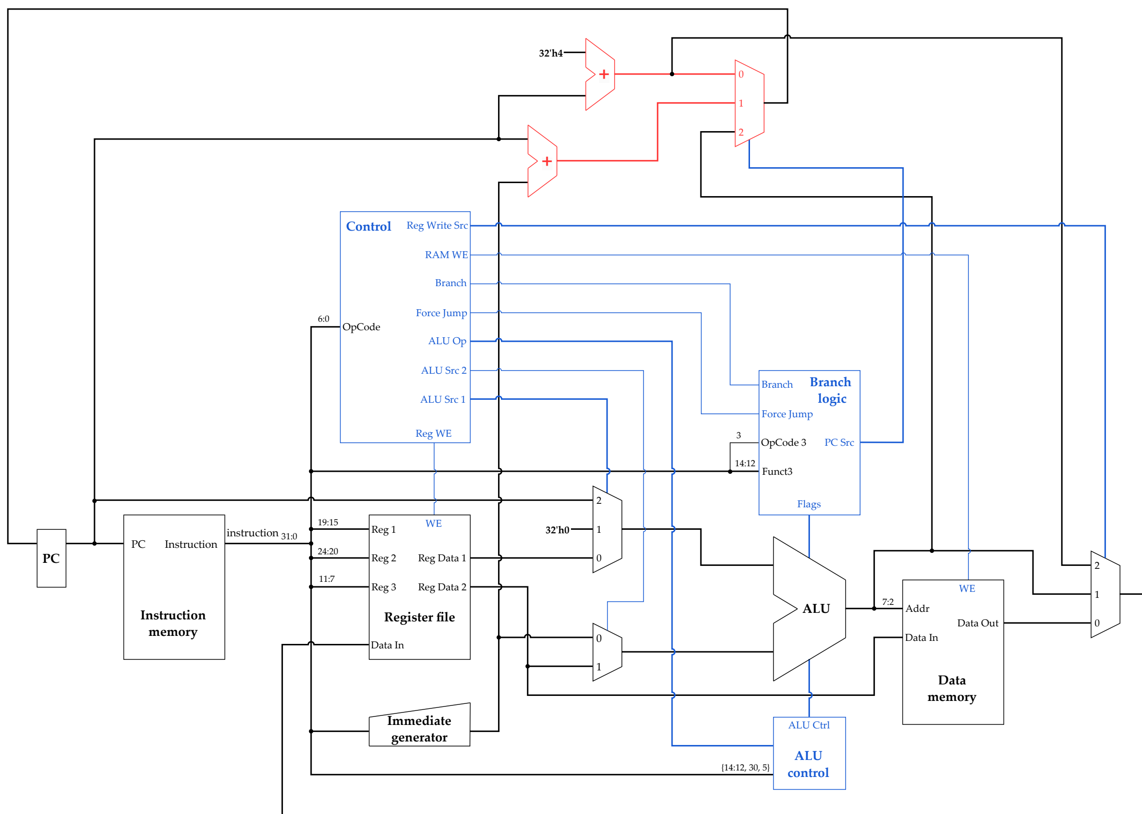
4.3.2.2 Branch logic

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 **1.4.18** shows in red the components that have been removed.

Figure 1.4.18: Removed components from single-cycle implementation



The branch logic module (see Figure **1.4.19**), 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 **1.4.15**.

Figure 1.4.19: Branch logic interface

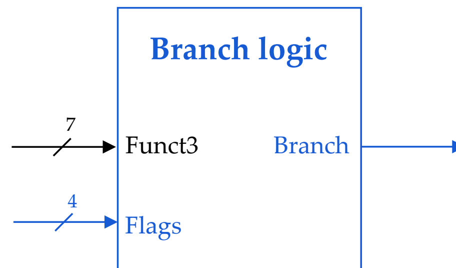


Table 1.4.15: Branch logic

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 **1.4.16**). That means that every peripheral must implement the same interface to be connected to the bus.

Table 1.4.16: 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

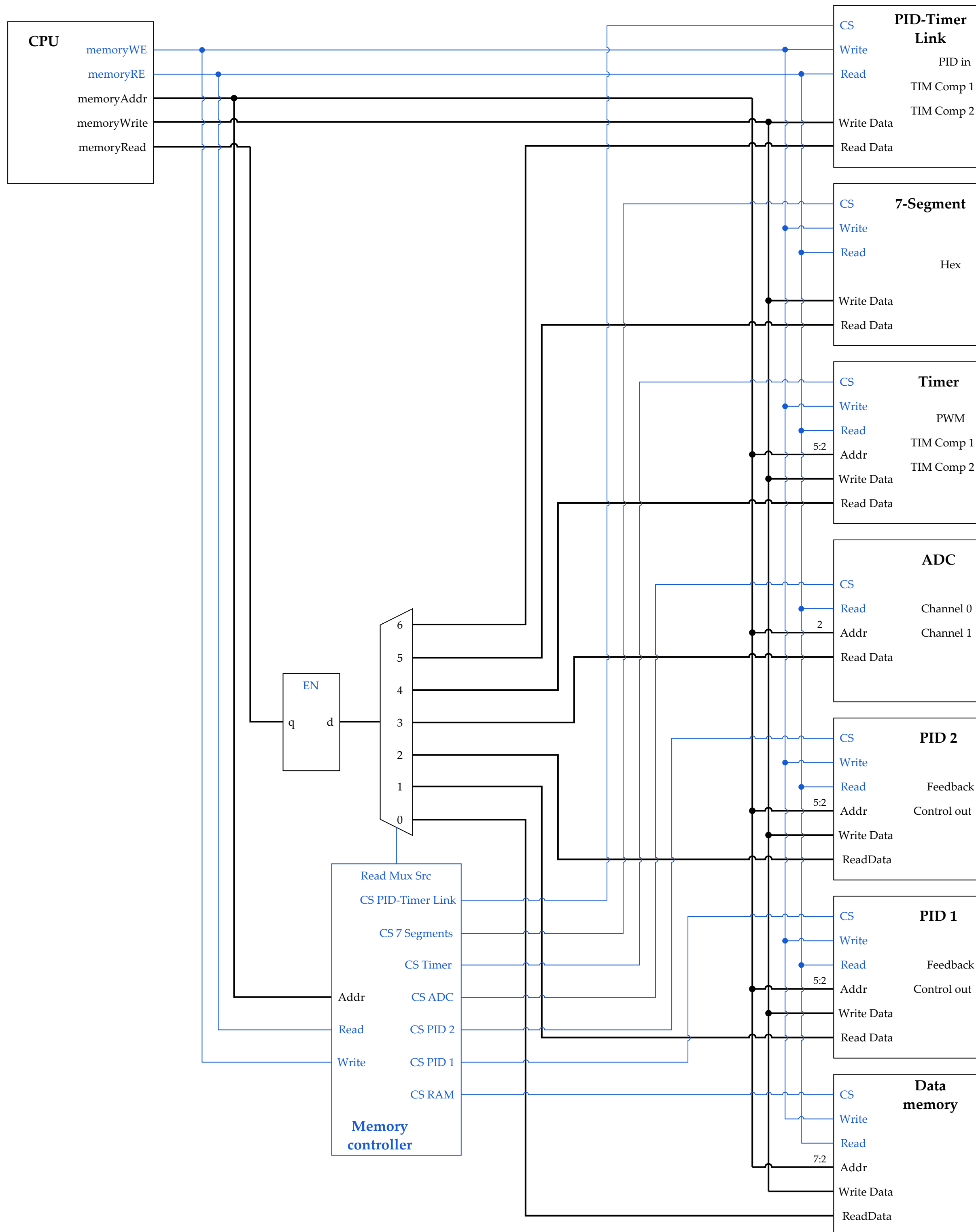
Slave interface

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 **1.4.20**. 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.

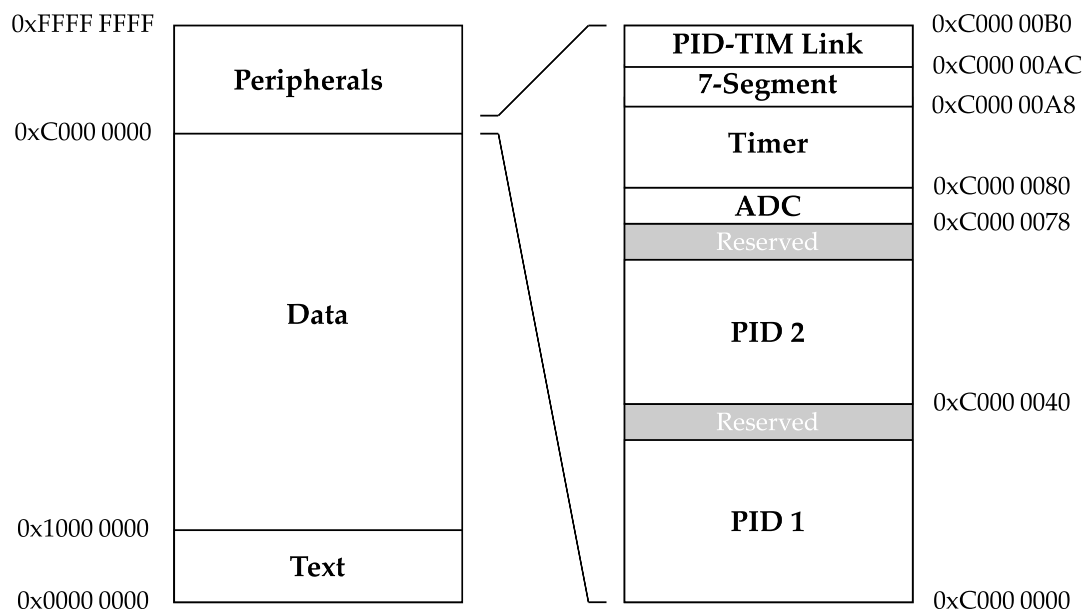
Figure 1.4.20: Memory bus



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 **1.4.21**) 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 1.4.21: Memory map

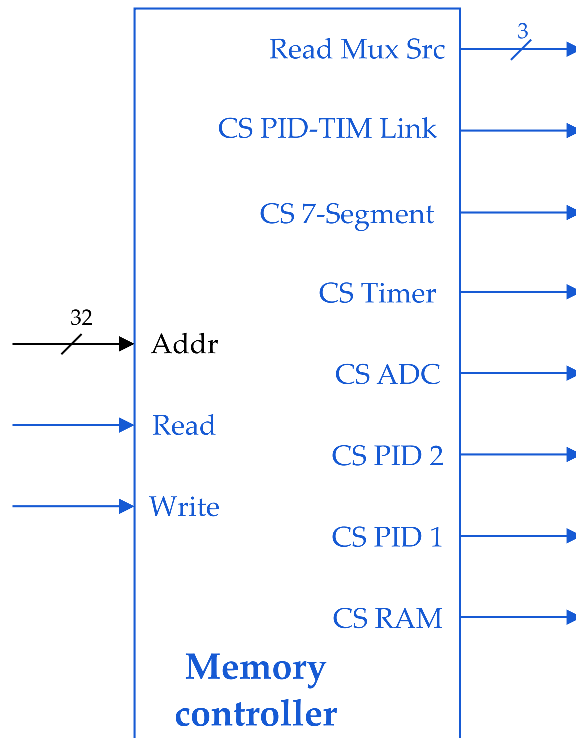


4.3.3.1 Memory Controller

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 **1.4.22**).

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

Figure 1.4.22: 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 **1.4.17** indicates the base addresses of each of the peripherals.

Table 1.4.17: 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 **1.4.23**). It also incorporates a shifter (see Table **1.4.18**), that can be programmed from the CPU, to adequate the control output value to fit the duty cycle range. Figure **1.4.24** shows the interface of the module.

Figure 1.4.23: PID-Timer Link interface

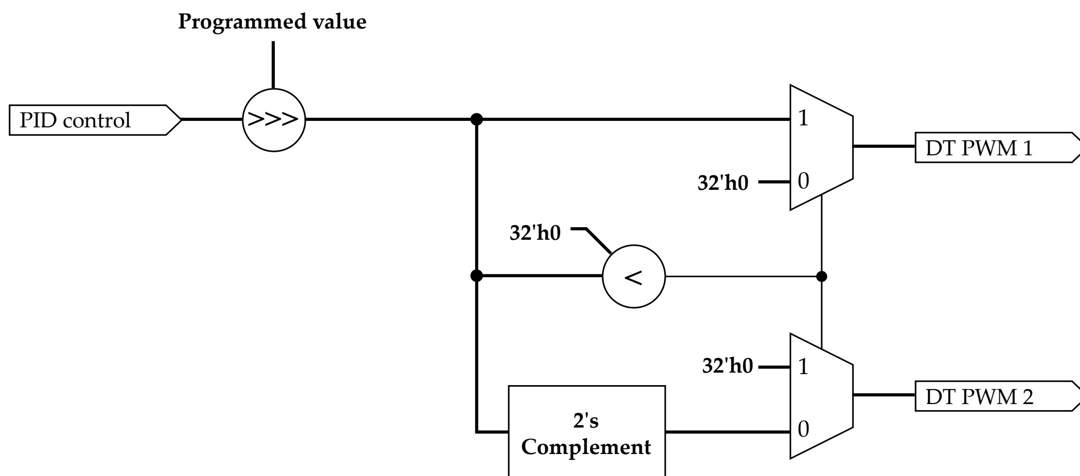


Figure 1.4.24: PID-Timer Link interface

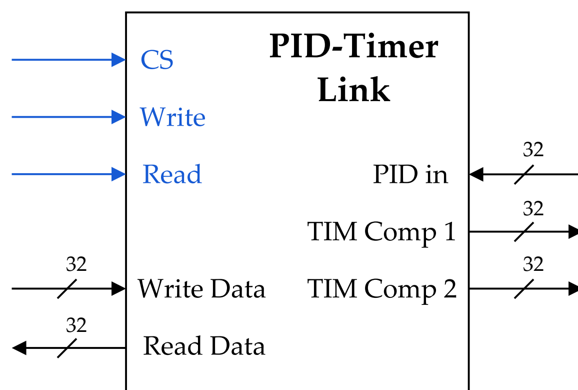


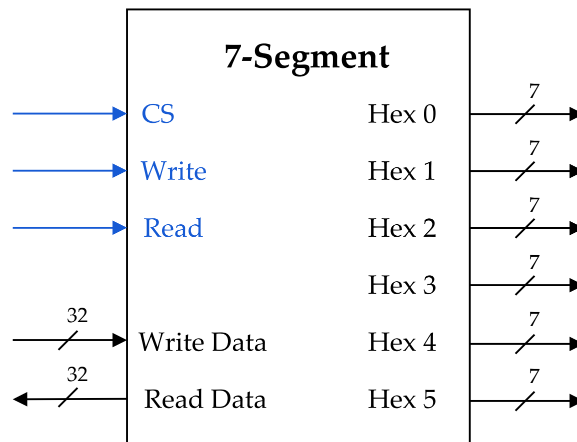
Table 1.4.18: PID-Timer link register description

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 **1.4.25**).

Figure 1.4.25: 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 **1.4.19**).

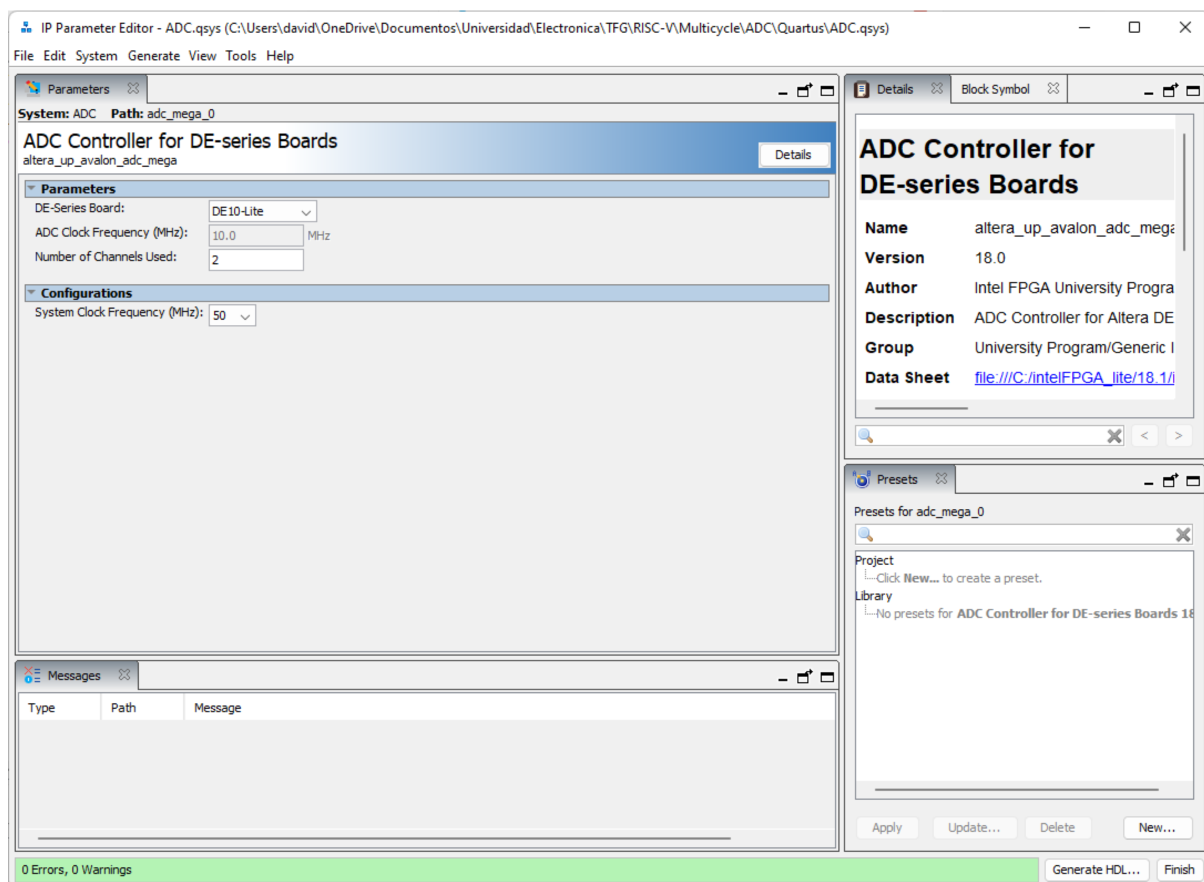
Table 1.4.19: Seven-segment decoder register description

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 1.4.26: 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

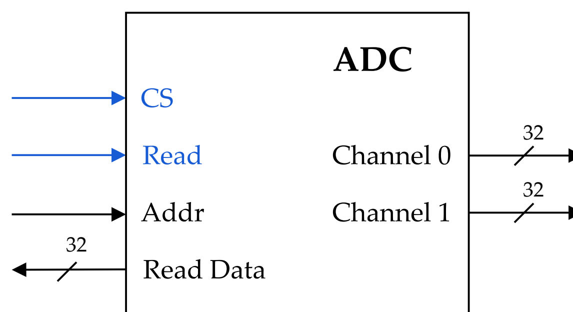
Figure **1.4.26** shows a screenshot from the IP Parameter Editor.

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 **1.4.20**), channel one and channel two, respectively. Additionally, both channels have been bypassed as outputs of the module (as shown in Figure **1.4.27**) to be used as feedback by the PID modules.

Table 1.4.20: 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

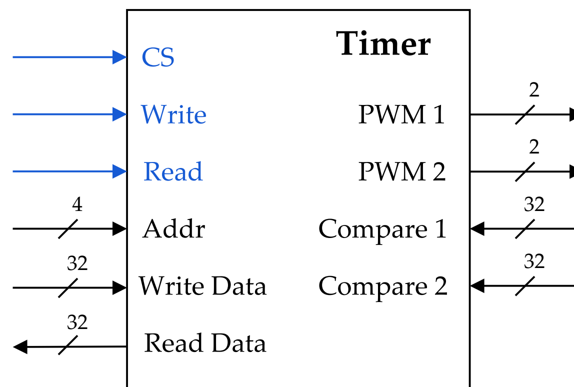
Figure 1.4.27: Memory controller interface



4.3.3.5 Timer

Timer peripherals are circuits intended to perform time-related tasks. Specifically, the timer designed (Figure **1.4.28** 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 1.4.28: Timer interface

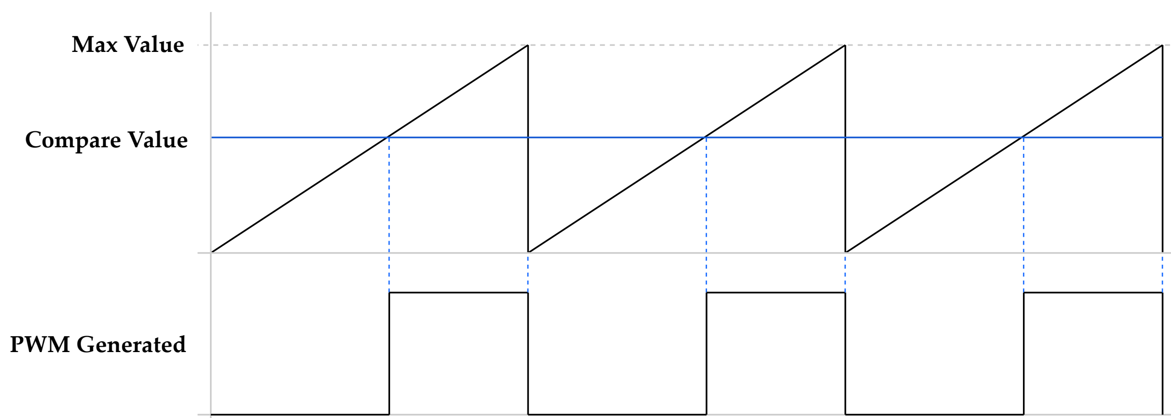


The timer peripheral has been designed by developing smaller modules that perform a single task and are then interconnected to generate the peripheral. The modules are a counter, a prescaler, and two PWM generators. The counter is the primary element of the timer. It is based on a 32-bit register that 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 **1.4.29**). 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 **1.4.21**. 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 1.4.29: 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 **1.4.22** indicates the address assigned to each peripheral and its function.

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

Table 1.4.21: 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 1.4.22: 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.

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

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

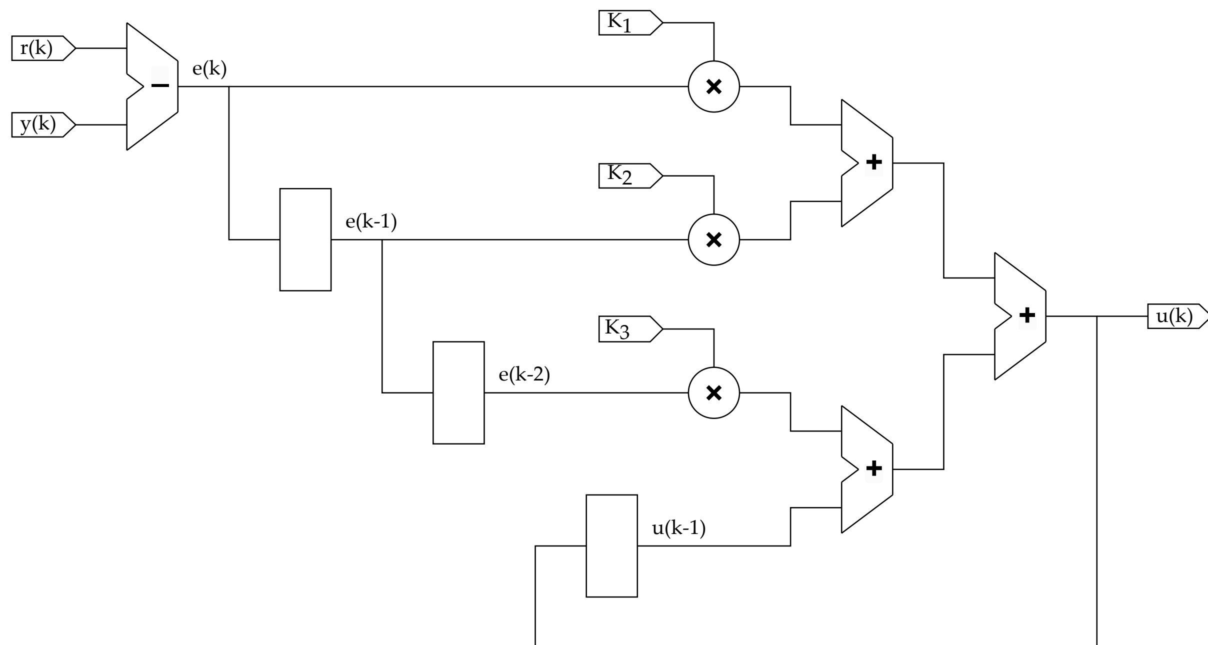
$$u(k) = u(k-1) + e(k) \cdot \left(K_p + \frac{T_d}{T}\right) + e(k-1) \cdot \left(-K_p + \frac{T}{T_i} - 2\frac{T_d}{T}\right) + e(k-2)\frac{T_d}{T} \tag{6}$$

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

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

Once obtained the difference equation has been obtained, the electronic circuit can be implemented. The electronic diagram is shown in Figure **1.4.30**.

Figure 1.4.30: 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 **1.4.23**). 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

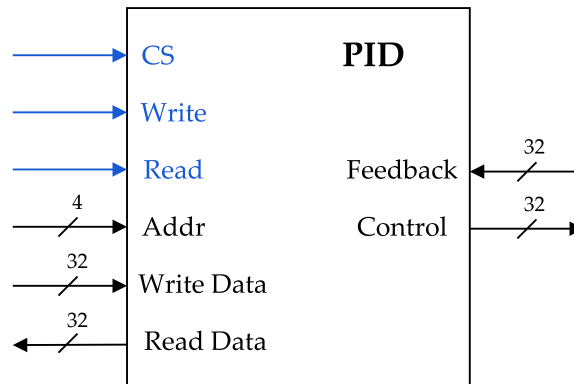
Table 1.4.23: PID controller registers description

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 **1.4.31**) which is directly connected to an ADC.

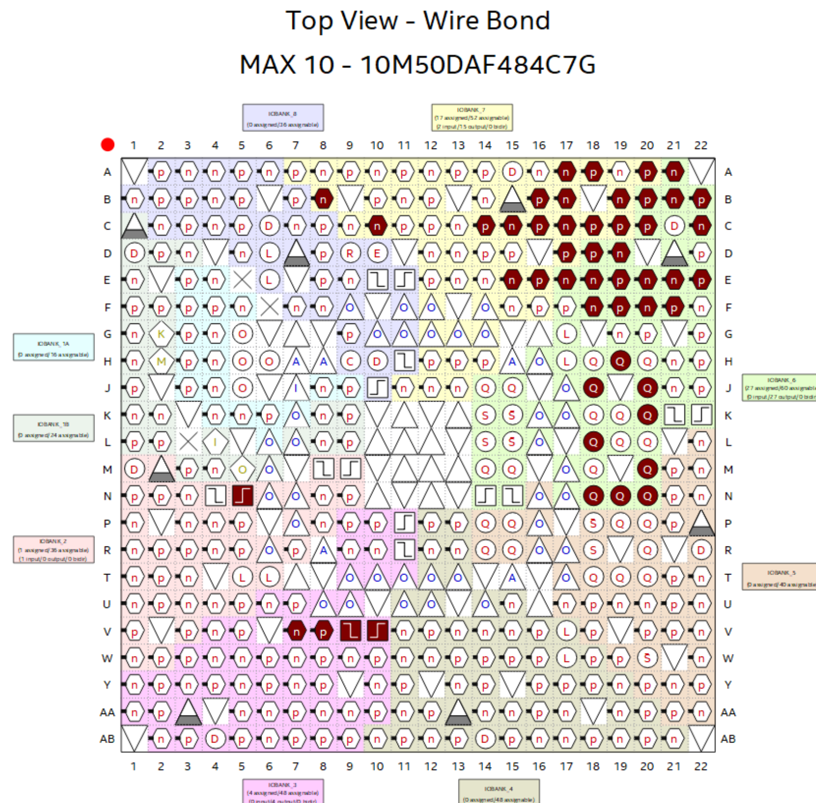
Figure 1.4.31: PID controller interface



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 **1.4.32** shows the pinout of the FPGA. However, the complete list is located in Annex **C**.

Figure 1.4.32: 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 **1.4.33**. 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 1.4.33: 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:

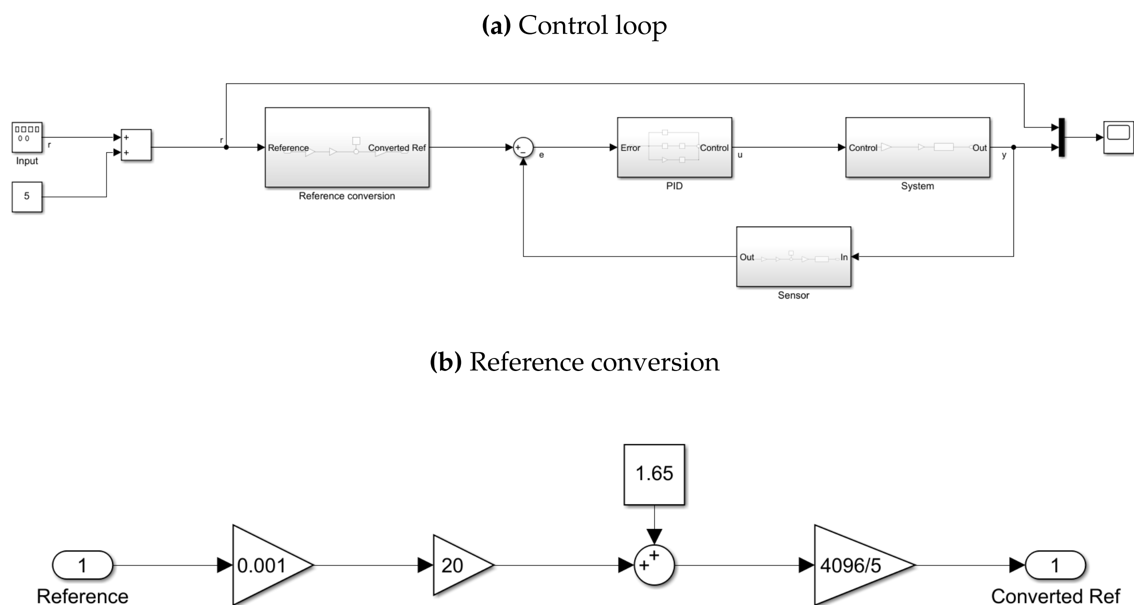
$$G(s) = \frac{48}{4096} \cdot \frac{1}{0.0048 \cdot s + 0.4488} \tag{9}$$

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.

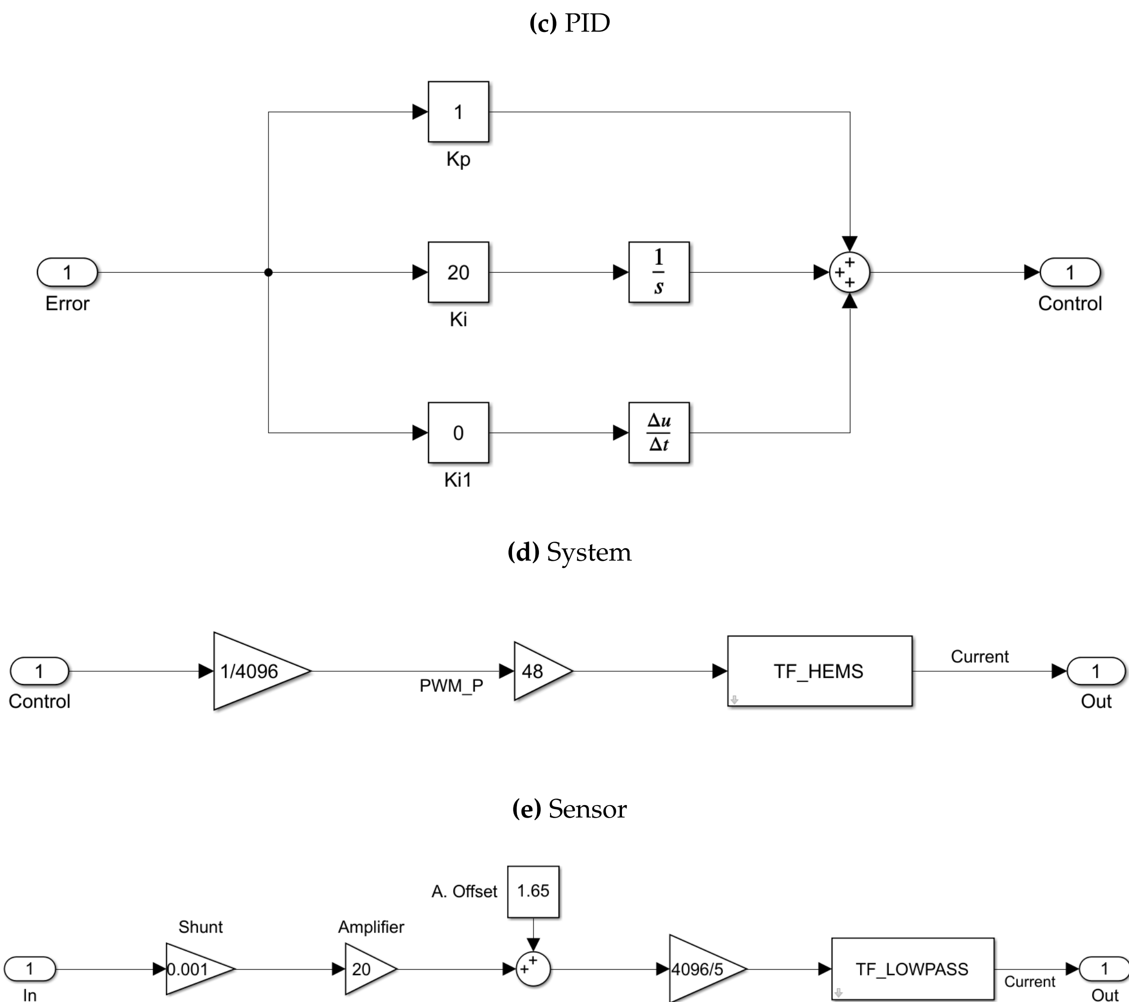
The controller employed is a PI (proportional-integral) controller. This decision has been taken because, due to the control requirements, a PI controller fulfils all the requirements expressed. Adding the derivative term may cause instability due to high-frequency noise, as the variation rate affects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

Figure 1.4.34: Simulink diagrams



(a) Control loop

(b) Reference conversion



(c) PID

(d) System

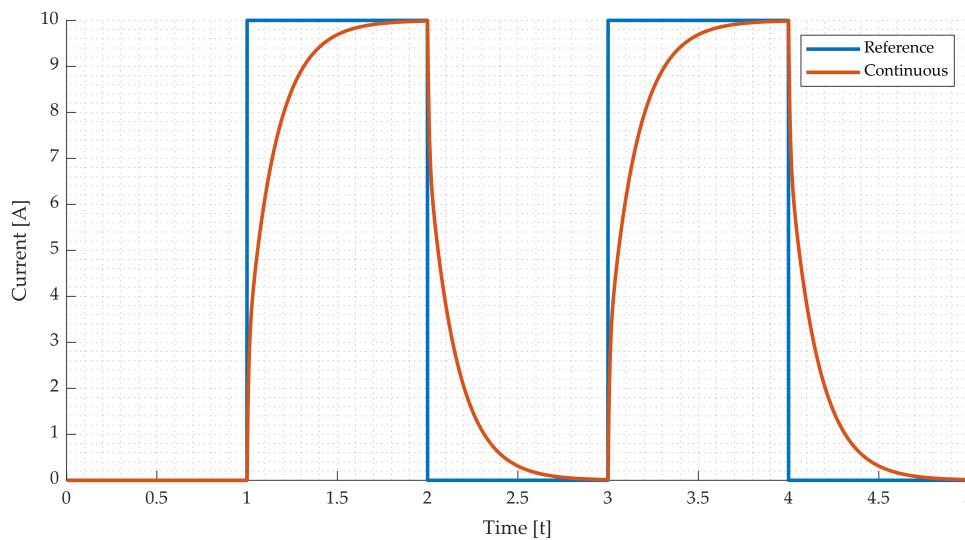
(e) Sensor

The system has been replicated in MATLAB Simulink to tune the PI controller. The control loop consists of four subsystems (shown in Figure **1.4.34**):

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 **1.4.34**.

Figure 1.4.34: 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 **1.4.35**), it can now be simulated. Figure **1.4.36** 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 1.4.35: PID circuit Simulink block

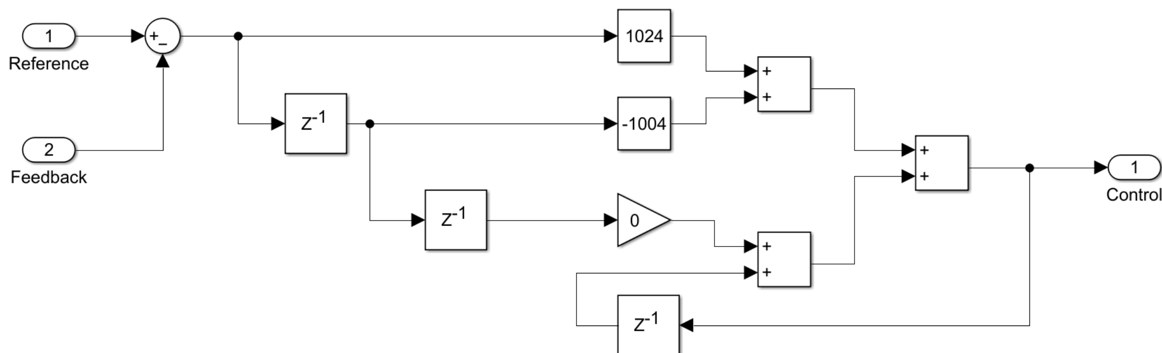
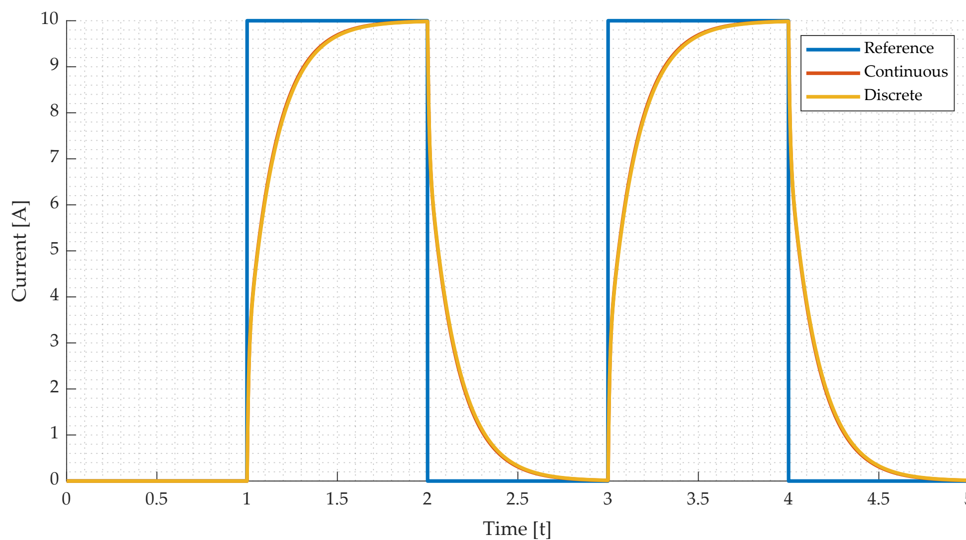


Figure 1.4.36: Discrete PID simulation



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.

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

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 **1.4.21**, 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 **1.3.5** of Part **VI**.

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 **2** of Document **VI**.

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 **1.5.1**). Finally, a message is printed showing the number of tests executed and the errors that have arisen in the verification process (Figure **1.5.1**).

Figure 1.5.1: 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

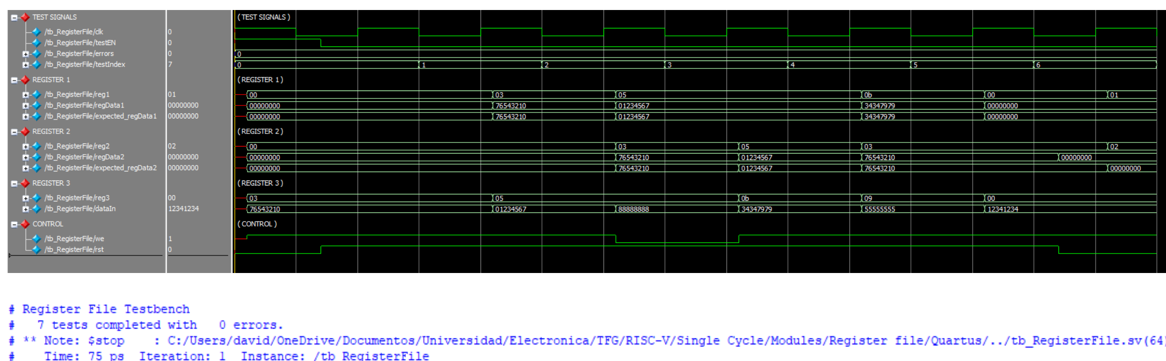
The following figures show the results obtained for the different modules of the single-cycle processor.

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 1.5.2: Register file verification results



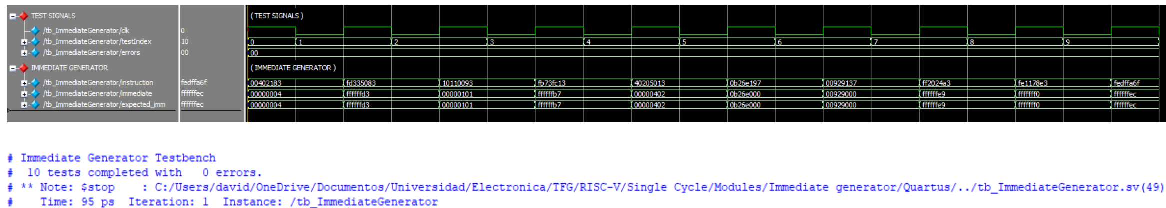
```
# Register File Testbench
#   7 tests completed with   0 errors.
# ** Note: $stop    : C:/Users/david/OneDrive/Documentos/Universidad/Electronica/TFG/RISC-V/Single Cycle/Modules/Register file/Quartus/../tb_RegisterFile.sv(64)
#    Time: 75 ps  Iteration: 1  Instance: /tb_RegisterFile
```

Immediate generator:

Test cases analysed:

- The immediate is generated correctly for every instruction type.

Figure 1.5.3: Immediate generator verification results

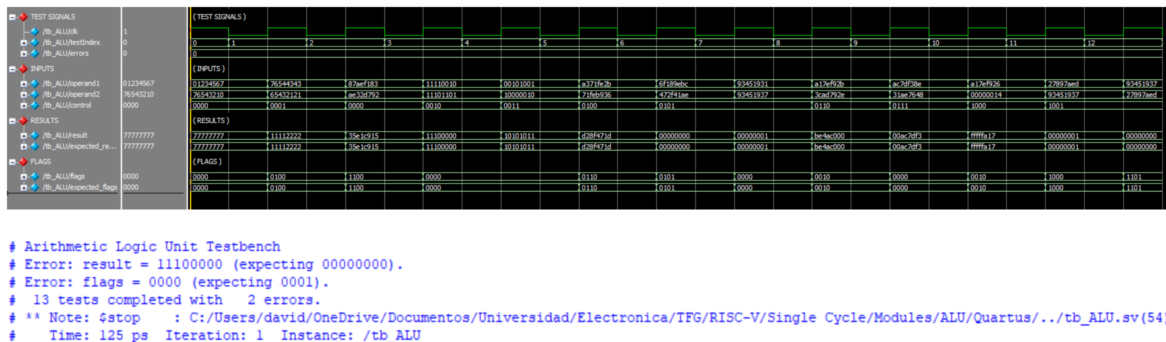


Arithmetic Logic Unit:

Test cases analysed:

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

Figure 1.5.4: ALU verification results

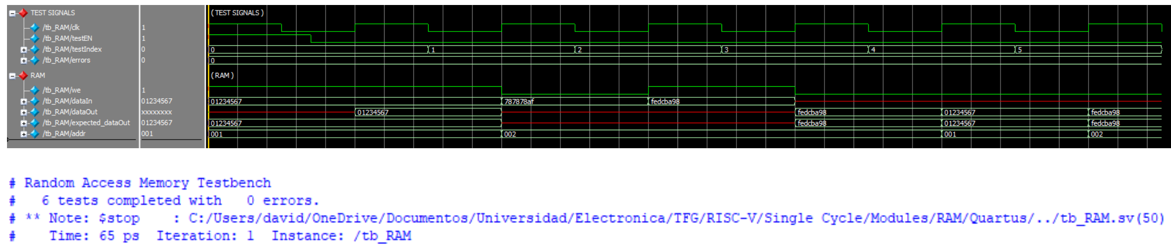


Data memory:

Test cases analysed:

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

Figure 1.5.5: Data memory verification results

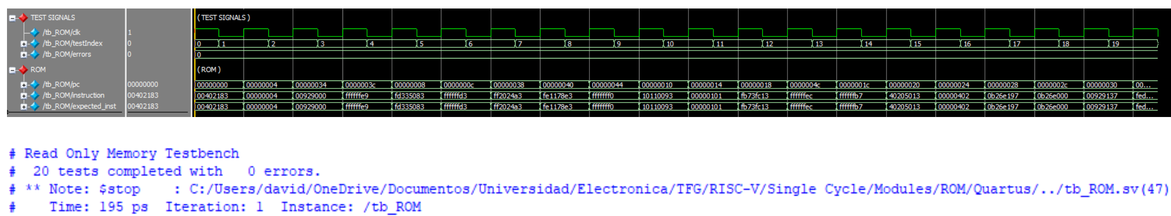


Program memory:

Test cases analysed:

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

Figure 1.5.6: Program memory verification results

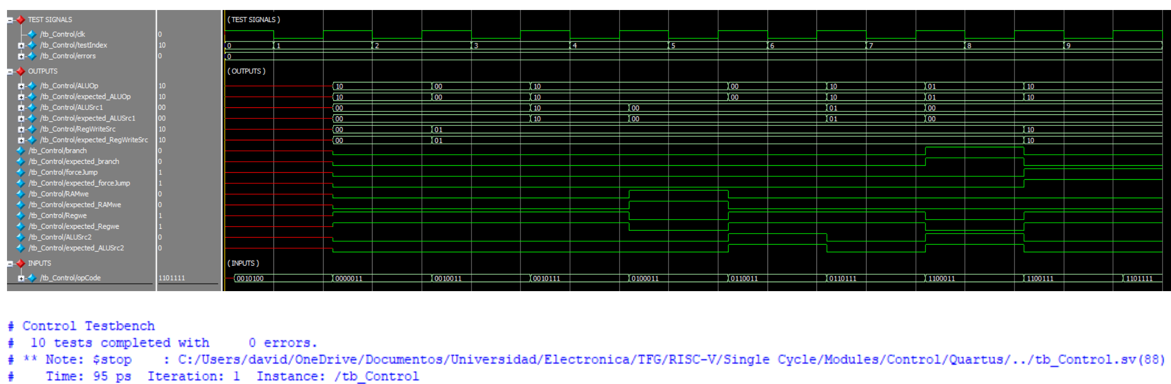


Control unit:

Test cases analysed:

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

Figure 1.5.7: Control unit verification results

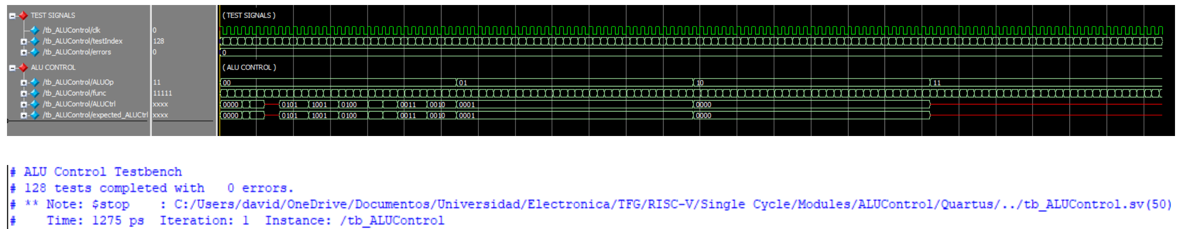


ALU control:

Test cases analysed:

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

Figure 1.5.8: ALU control verification results

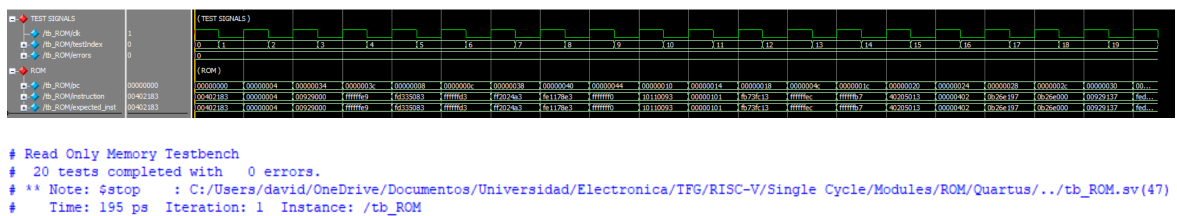


Branch logic:

Test cases analysed:

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

Figure 1.5.9: Branch logic verification results



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 **1.5.10** shows the simulation waveforms. Figure **1.5.11**, on the other side, shows the register file and data memory (only the written address) at the end of the simulation.

Code snippet 1.5.3: Types I, S verification

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

Figure 1.5.10: Types I, S verification waveforms

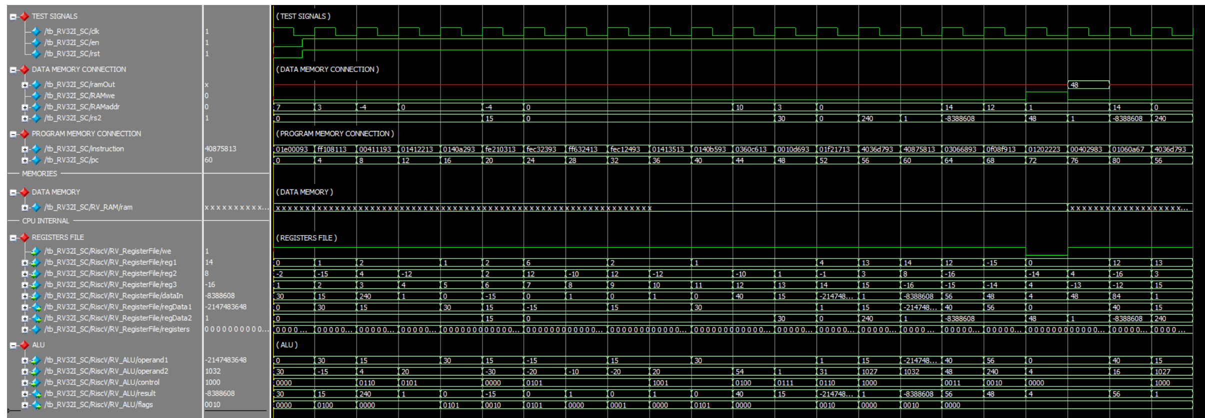
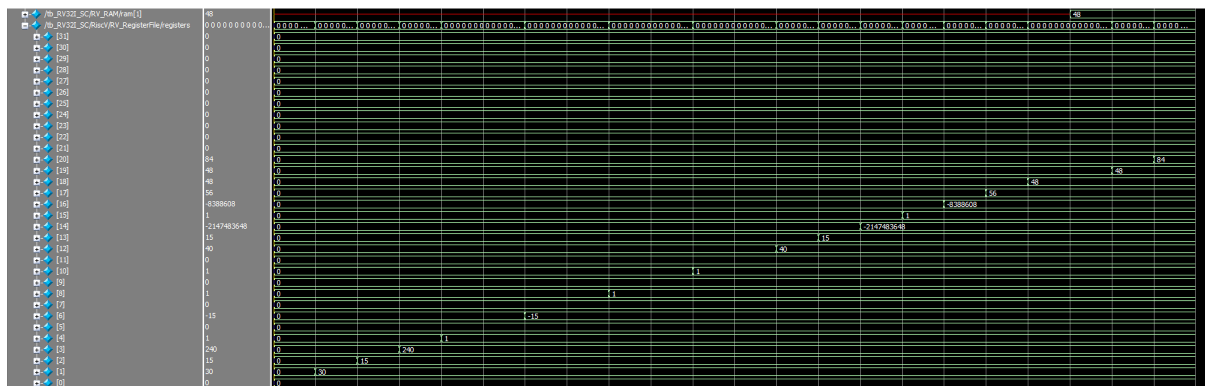


Figure 1.5.11: Type I, S register file waveforms



Proceeding as with types I and S, types R, U and J have been tested altogether. Figure **1.5.12** shows an overview of the complete CPU, while Figure **1.5.12** shows a detailed view of the registers file.

Code snippet 1.5.4: Types R, U, J verification

```
1    addi x1, x0, 30     # x1 = 30
2    addi x2, x0, 15     # x2 = 15
3    add x3, x1, x2      # x3 = 45
4    sub x4, x1, x2      # x4 = 15
5    sub x5, x2, x1      # x5 = -15
6    addi x6, x0, 2      # x6 = 2
7    sll x7, x2, x6      # x7 = 60
8    slt x8, x4, x3      # x8 = 1
9    slt x9, x3, x4      # x9 = 0
10   slt x10, x4, x5     # x10 = 0
```

```
11    slt x11, x5, x4      # x11 = 1
12    sltu x12, x4, x3     # x12 = 1
13    sltu x13, x3, x4     # x13 = 0
14    xor x14, x2, x7      # x14 = 51
15    srl x15, x2, x6      # x15 = 3
16    sra x16, x2, x6      # x16 = 3
17    sra x17, x5, x6      # x17 = -4
18    or x18, x2, x7       # x18 = 63
19    and x19, x2, x7      # x19 = 12
20    lui x20, 2           # x20 = 8192
21    auipc x21, 2         # x21 = 8272
22    jal x22, .link       # x22 = 88
23    nop
24    nop
25    nop
26    .link:
27    addi x23, x0, 1      # x23 = 1
28    nop
```

Figure 1.5.12: Types R, U, J verification waveforms

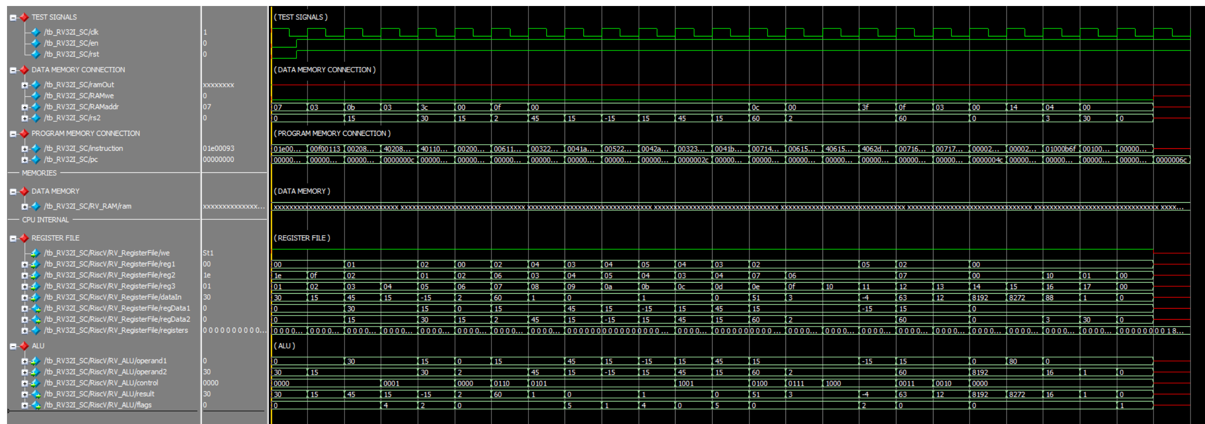
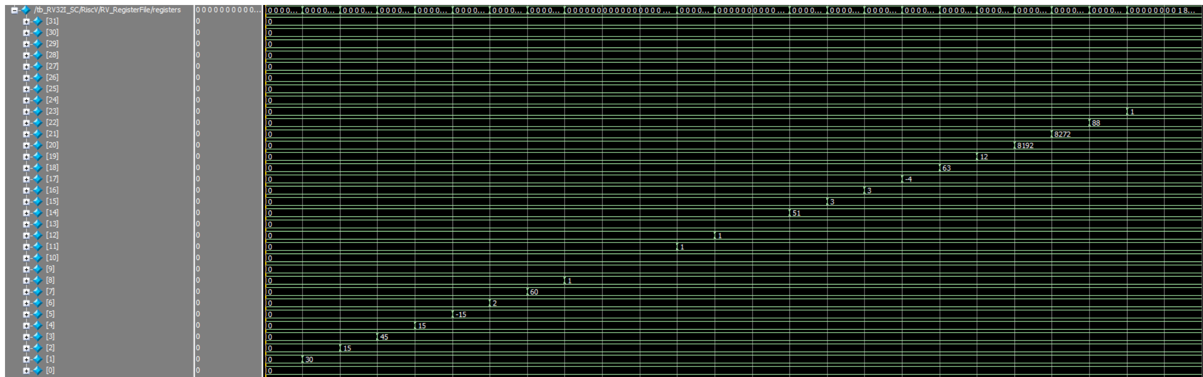


Figure 1.5.13: Types R, U, J register file waveforms



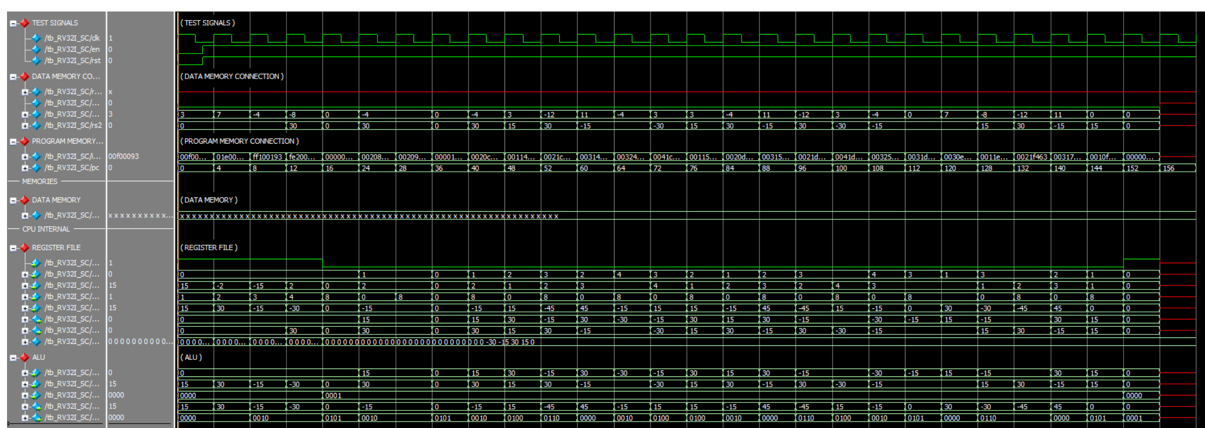
Finally, the same procedure is applied to branch instructions (type B). However, for this case, the main focus is on the PC value. As it can be seen in the Code Snippet **1.5.5**, every instruction is evaluated for a case in which it must perform a jump and a situation for which it is not supposed to perform it. The results can be seen in Figure **1.5.14**.

Code snippet 1.5.5: Type B verification

```
1   addi x1, x0, 15         # x1 = 15
2   addi x2, x0, 30         # x2 = 30
3   addi x3, x0, -15        # x3 = -15
4   addi x4, x0, -30        # x4 = -30
5   beq x0, x0, .jump1      # if x0 == x0 then target
6   nop
7   .jump1:
8   beq x1, x2, .jump1
9   bne x1, x2, .jump2      # if x1 != x2 then target
10  nop
11  .jump2:
12  bne x0, x0, .jump2      # if x0 != x0 then target
13  blt x1, x2, .jump3      # if x1 < x2 then target
14  nop
15  .jump3:
16  blt x2, x1, .jump3      # if x2 < x1 then target
17  blt x3, x2, .jump4      # if x3 < x2 then target
18  nop
19  .jump4:
20  blt x2, x3, .jump4      # if x2 < x3 then target
21  blt x4, x3, .jump5      # if x4 < x3 then target
22  nop
23  .jump5:
24  blt x3, x4, .jump5      # if x3 < x4 then target
25  bge x2, x1, .jump6      # if x2 >= x1 then target
```

```
26      nop
27      .jump6:
28      bge x1, x2, .jump6     # if x1 >= x2 then target
29      bge x2, x3, .jump7     # if x2 >= x3 then target
30      nop
31      .jump7:
32      bge x3, x2, .jump7     # if x3 >= x2 then target
33      bge x3, x4, .jump8     # if x3 >= x4 then target
34      nop
35      .jump8:
36      bge x4, x3, .jump8     # if x4 >= x3 then target
37      .back1:
38      bge x3, x3, .jump9     # if x3 >= x3 then target
39      jal x0, .back1
40      .jump9:
41      bltu x1, x3, .jump10   # if x1 < x3 unsigned then target
42      nop
43      .jump10:
44      bltu x3, x1, .jump10   # if x3 < x1 unsigned then target
45      bgeu x3, x2, .jump11   # if x3 >= x2 unsigned then target
46      nop
47      .jump11:
48      bgeu x2, x3, .jump11   # if x2 >= x3 unsigned then target
49      .back2:
50      bgeu x1, x1, .jump12   # if x1 >= x1 unsigned then target
51      jal x0, .back2
52      .jump12:
53      nop
```

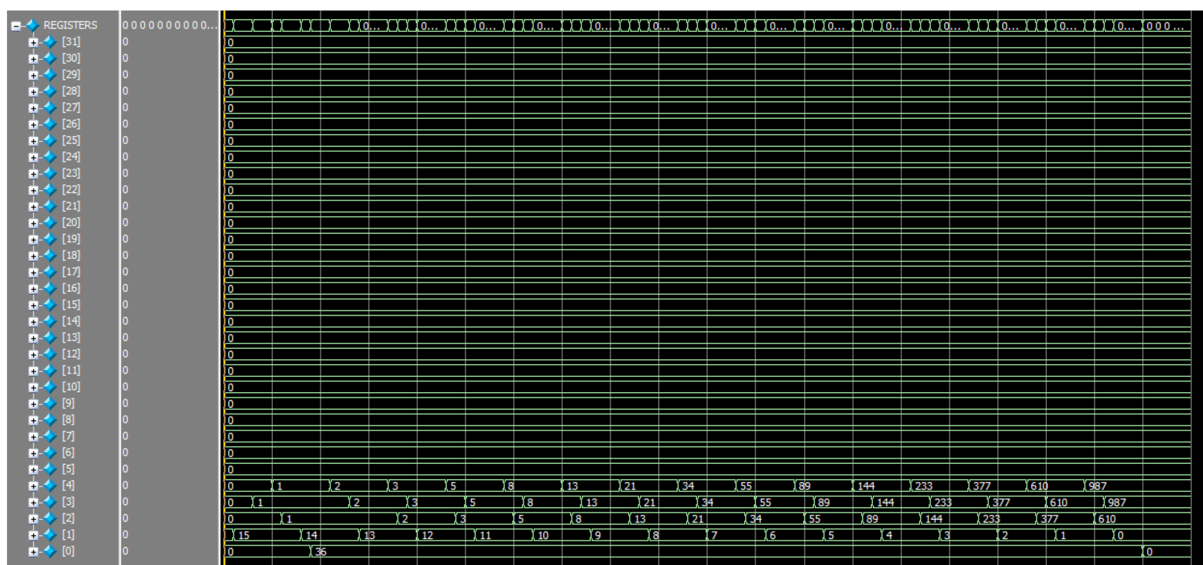
Figure 1.5.14: Type B register file waveforms



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 1.5.15 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 1.5.15: 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 **1.5.1**. The first column holds the unordered array, and the last column shows the result of the bubble sort algorithm.

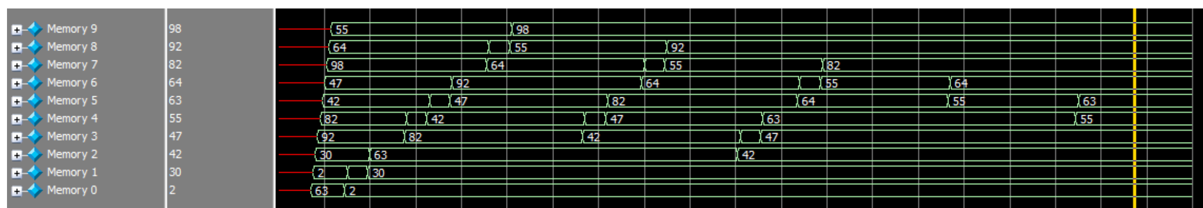
The assembly code generates a 10-number array and applies bubble sort to order the array. The simulation waveforms are shown in Figure **1.5.16**

Table 1.5.1: 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	1st			2nd			3rd			

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 1.5.16: Bubble sort



5.2 Multicycle

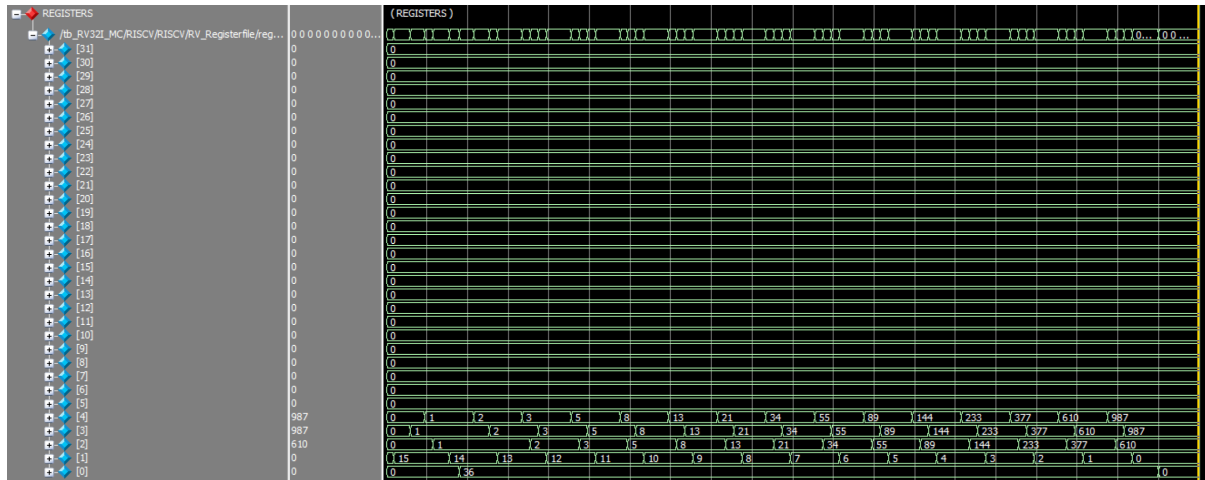
5.2.1 Processor

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 **1.5.17** 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.

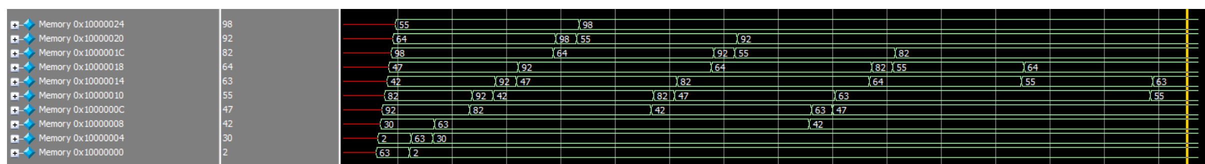
Figure 1.5.17: Fibonacci sequence



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 **1.5.18** 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 1.5.18: 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 **1.5.19** to **1.5.23** show the validation process results.

The test code (Code snippet **1.5.6**) 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 1.5.6: 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 1.5.19: PID 1 registers verification

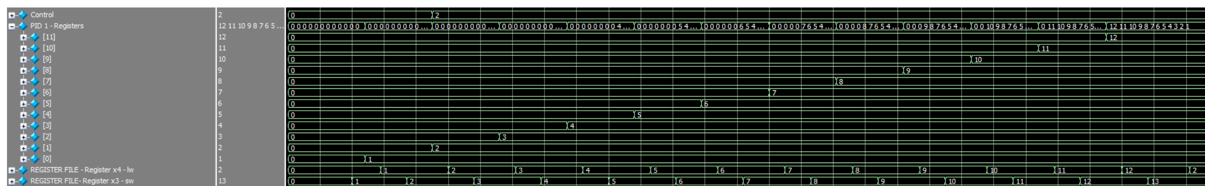


Figure 1.5.20: PID 2 registers verification

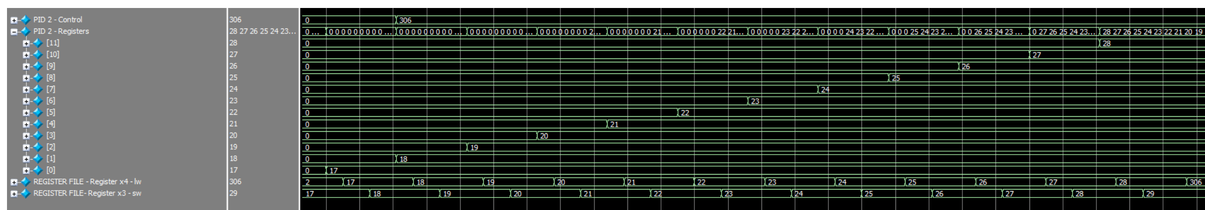
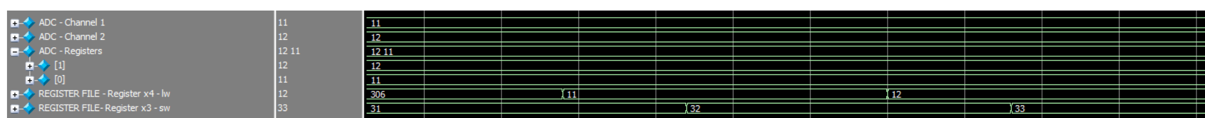


Figure 1.5.21: ADC registers verification



It can be observed in Figure **1.5.22** 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 1.5.22: Timer registers verification

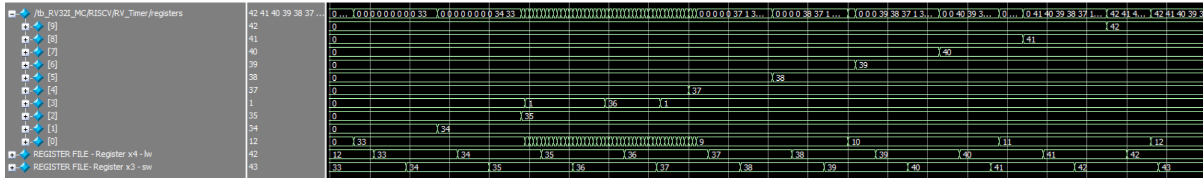
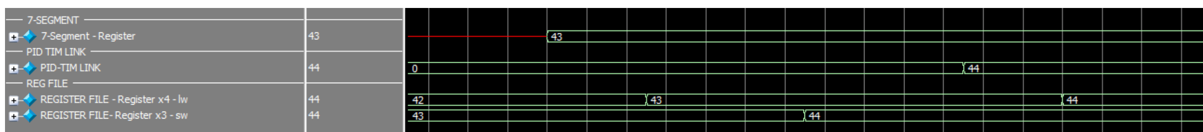


Figure 1.5.23: 7-segment and PID-Timer link registers verification



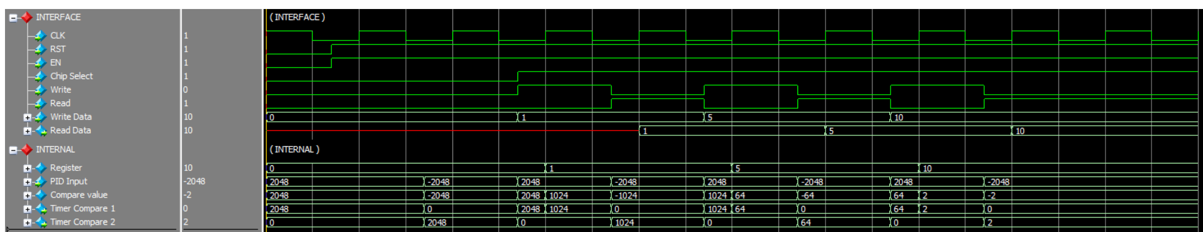
5.2.3 Peripherals

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 **1.5.24**.

Figure 1.5.24: 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 **1.5.25**). The input value was 0x00654321; therefore, the seven-segment shall display "654321", as shown in Figure **1.5.26**.

Figure 1.5.25: Seven-segment decoder verification

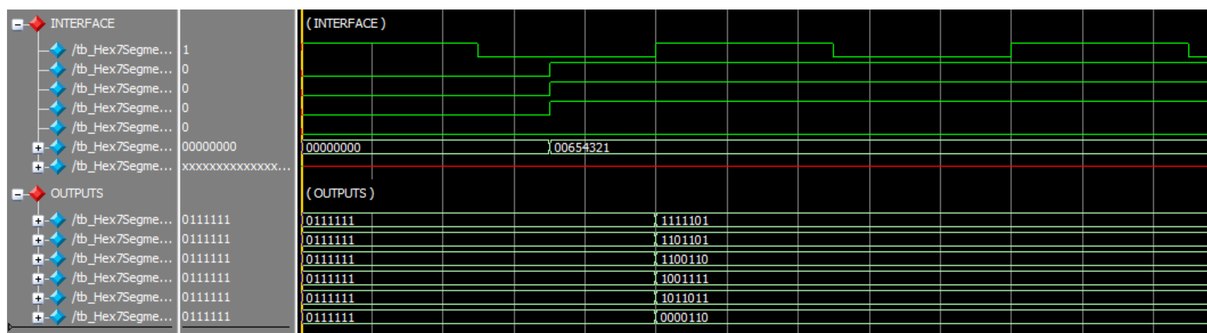
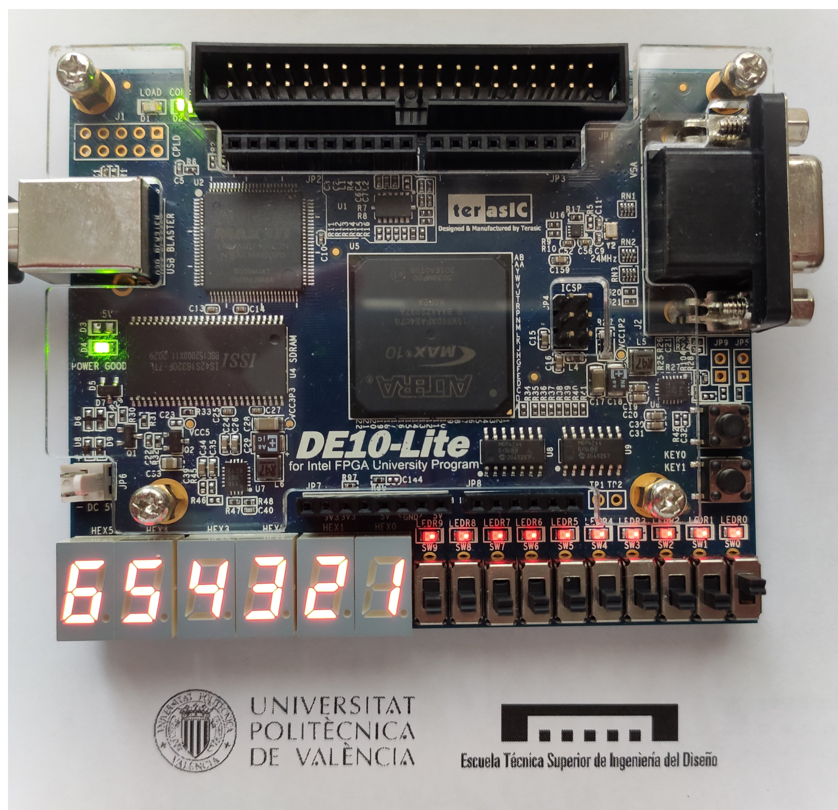


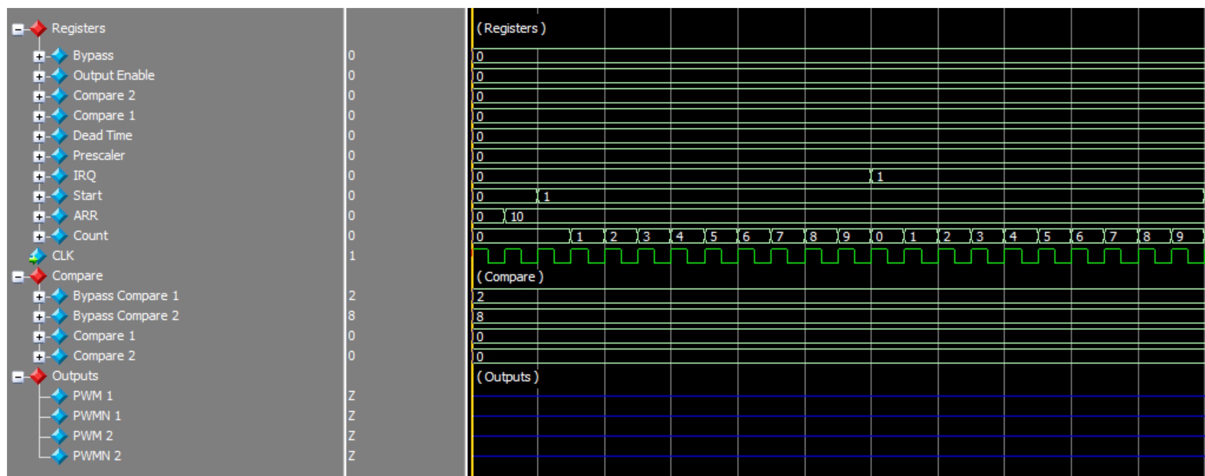
Figure 1.5.26: Seven-segment decoder verification FPGA



5.2.3.3 Timer

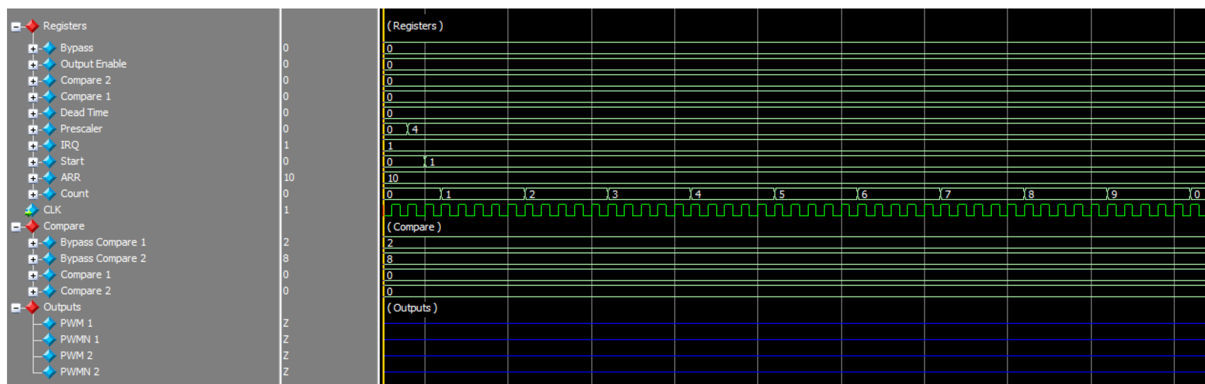
The different features of the timer have been tested individually. Firstly, it has been verified that the maximum count value limits the count value and that the end of the count indicator is asserted. Figure **1.5.27** shows how the output is limited to 10 and that the "IRQ" value is asserted at the end of the count.

Figure 1.5.27: 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 **1.5.28**.

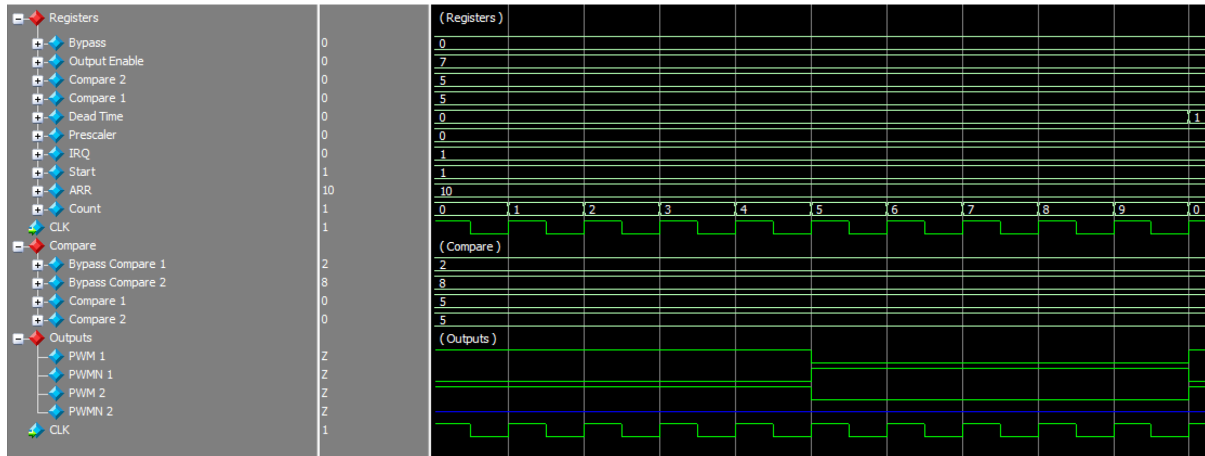
Figure 1.5.28: Timer prescaler verification



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

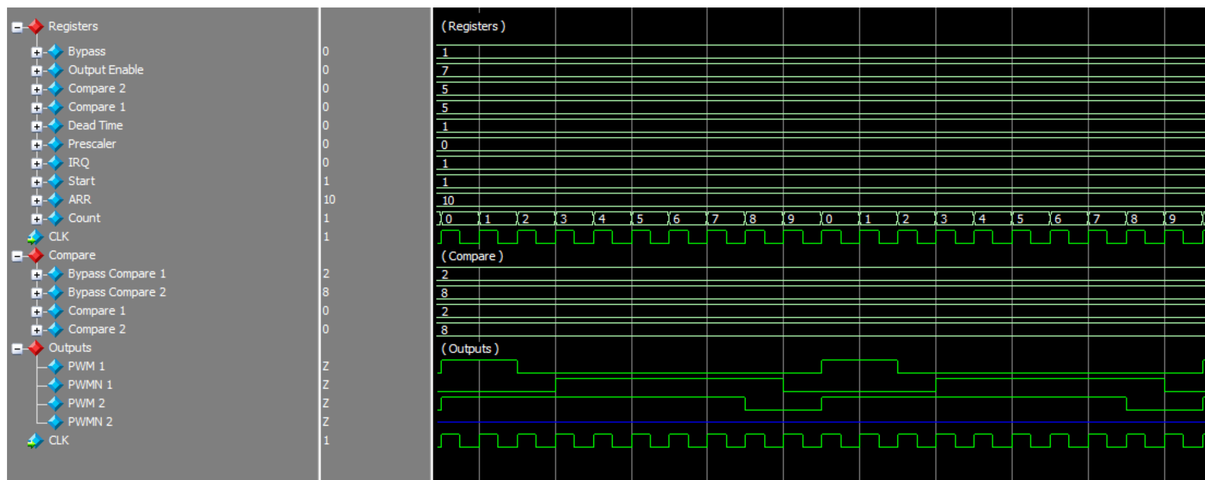
PWM signal, both channels with a duty cycle of 50% (Figure **1.5.29**).

Figure 1.5.29: 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 **1.5.30** shows the result of the verification.

Figure 1.5.30: 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 **1.5.31** shows the golden model simulation. Figure **1.5.32** compares the control action of the golden model and the hardware-implemented controller.

Figure 1.5.31: Golden model simulation

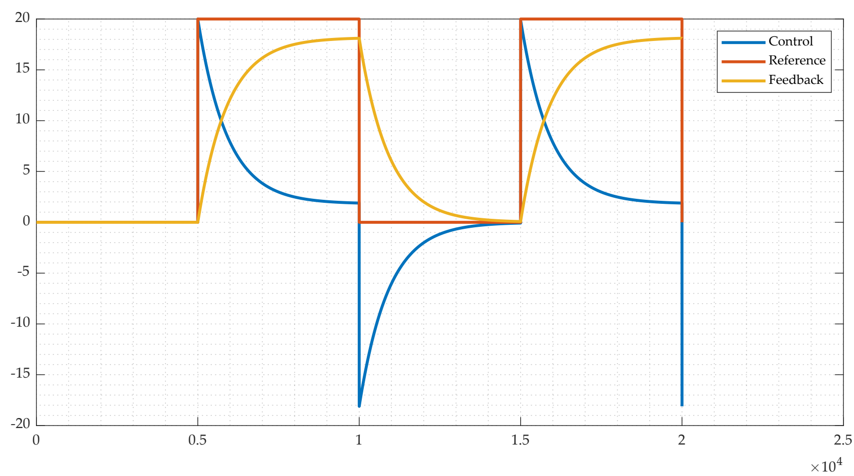
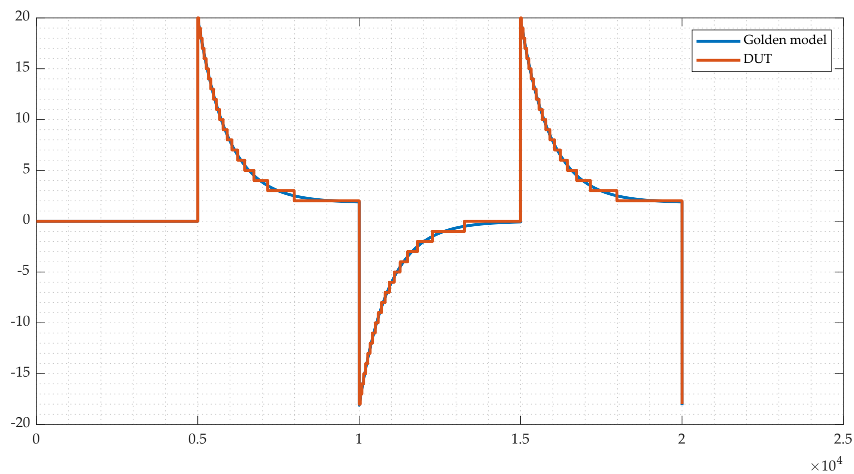
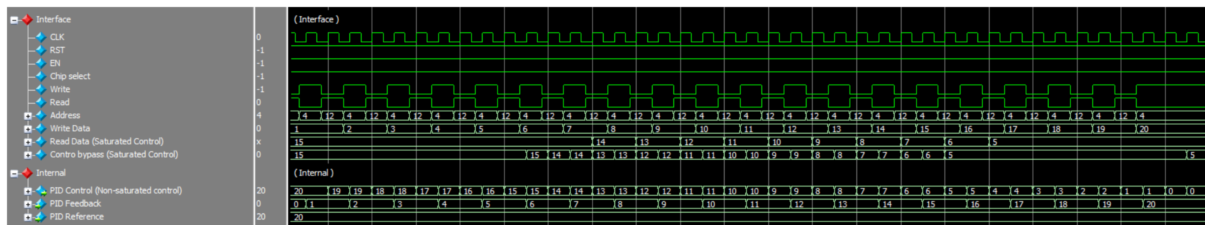


Figure 1.5.32: Control action comparison



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 **1.5.33**.

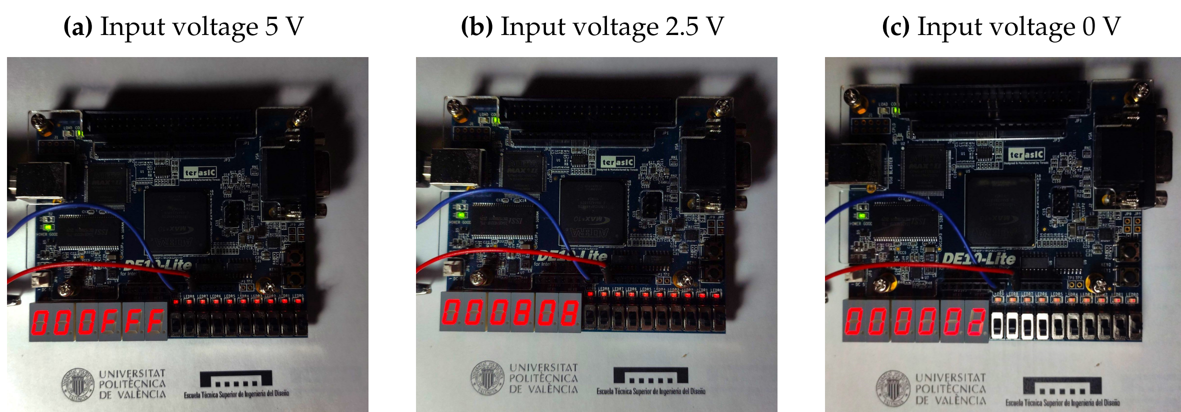
Figure 1.5.33: PID saturation verification



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 **1.5.34** shows the results for different input voltages.

Figure 1.5.34: 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 1.A.1: PID controller registers description

Memory addres	Read/Write	Name	Peripheral
`0xC0000000`	R/W	Reference	PID 1
`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	
`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`	Reserved		
`0xC0000038`			
`0xC000003C`			

0xC0000040	R/W	Reference	PID 2
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	
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	Timer
0xC0000084	R/W	ARR	
0xC0000088	R/W	Start	
0xC000008C	R/W	IRQ	
0xC0000090	R/W	Prescaler	
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 1.B.1: LPM_MULT configuration screen 1

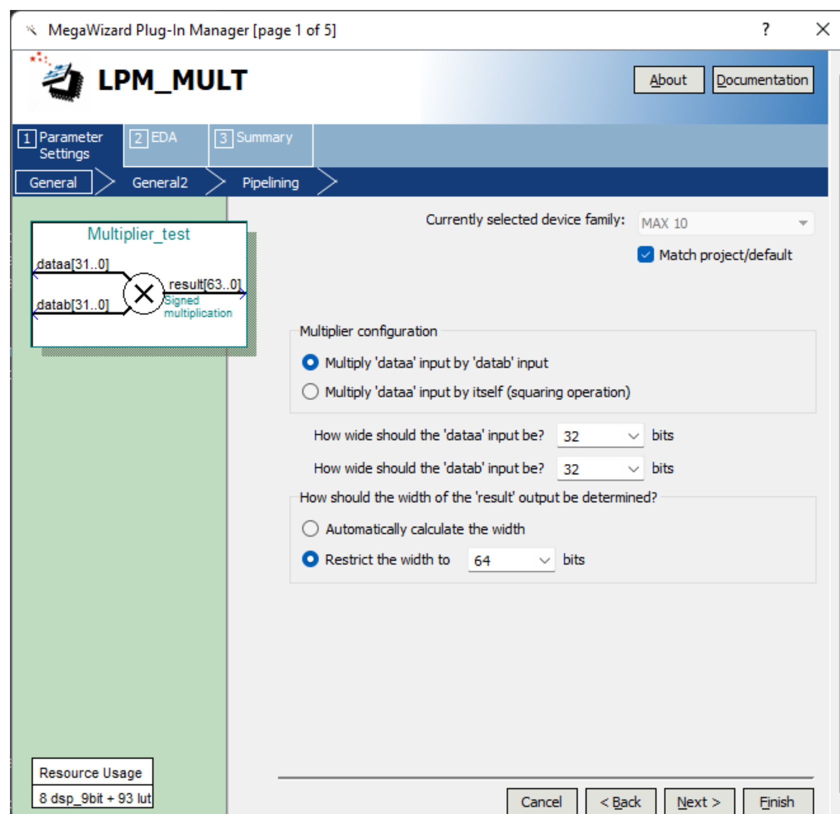


Figure 1.B.2: LPM_MULT configuration screen 2

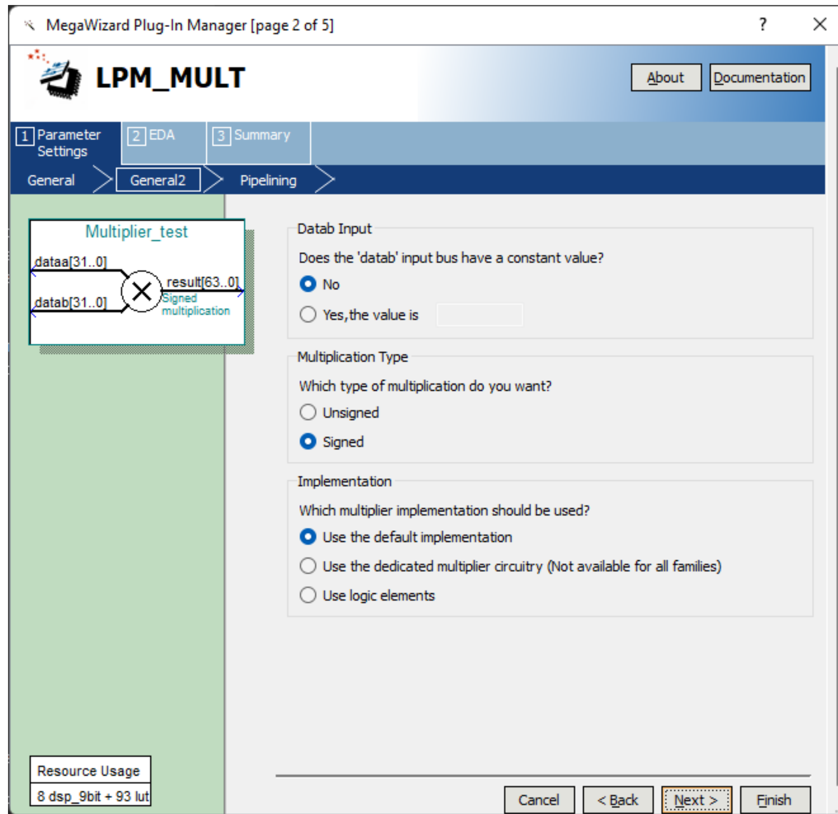


Figure 1.B.3: LPM_MULT configuration screen 3

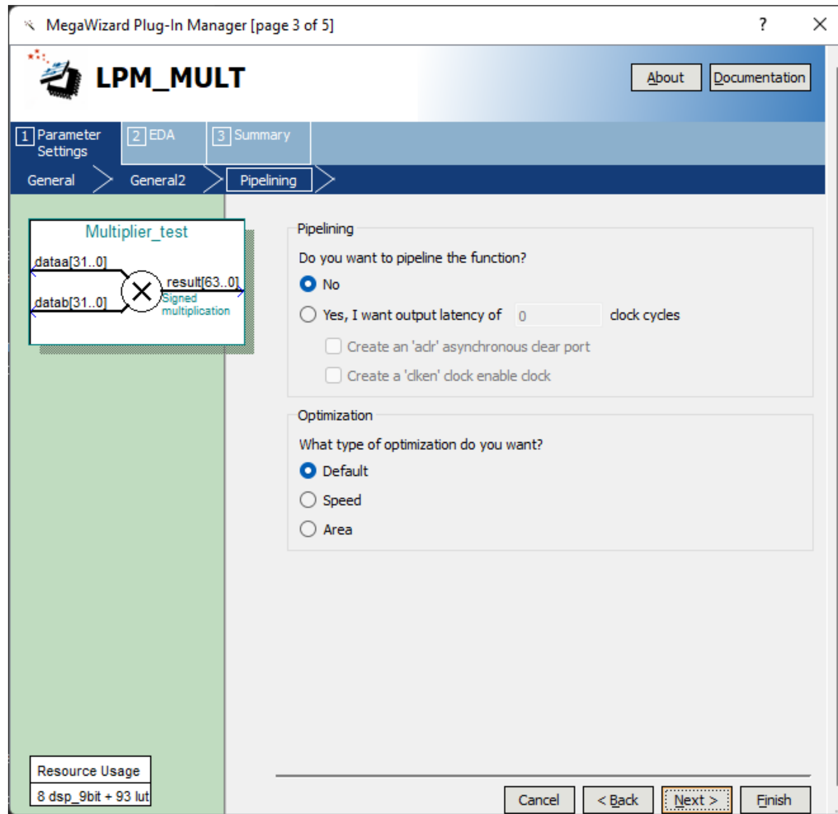


Figure 1.B.4: LPM_MULT configuration screen 4

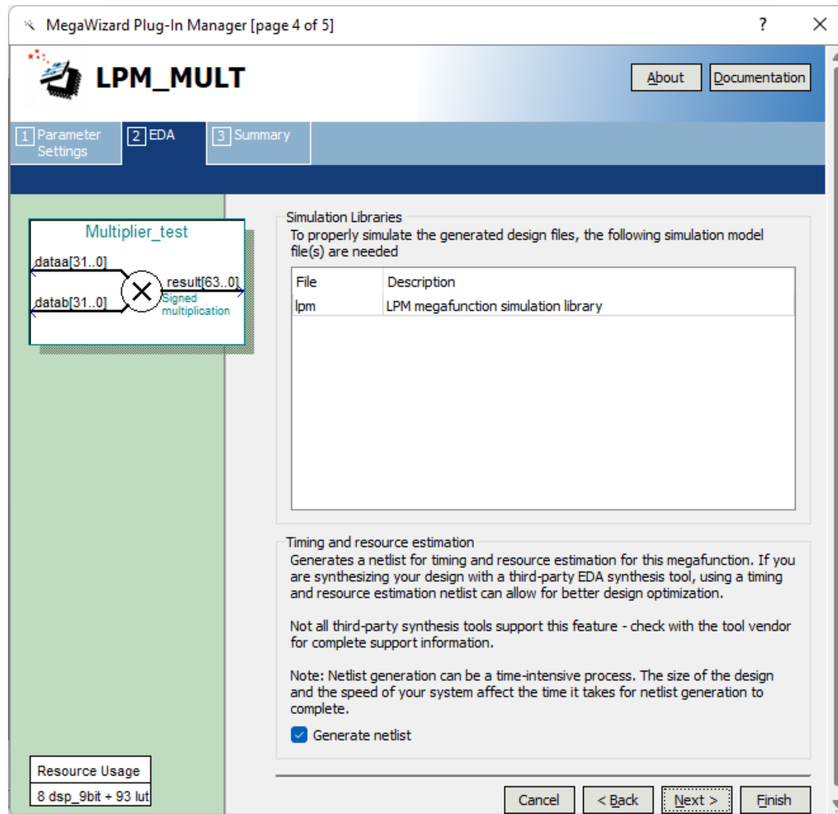
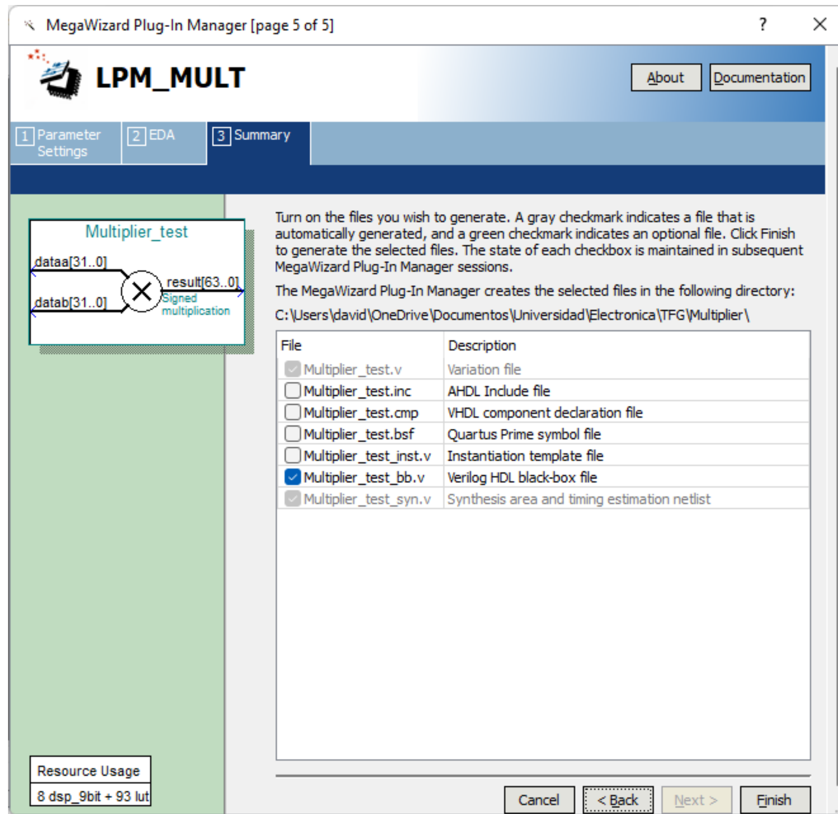


Figure 1.B.5: LPM_MULT configuration screen 5



Appendix C

Pinout

Table 1.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			

Appendix D

Sustainable Development Goals

Table 1.D.1: Degree to which the project relates to the Sustainable Development Goals

Sustainable Development Goals		High	Some	Low	N/A
SDG 1.	**No poverty**				X
SDG 2.	**Zero hunger**				X
SDG 3.	**Good health and well-being**				X
SDG 4.	**Quality education**				X
SDG 5.	**Gender equality**				X
SDG 6.	**Clean water and sanitation**				X
SDG 7.	**Affordable and clean energy**		X		
SDG 8.	**Decent work and economic growth**				X
SDG 9.	**Industry, innovation and infrastructure**	X			
SDG 10.	**Reduced inequalities**				X
SDG 11.	**Sustainable cities and comunities**			X	
SDG 12.	**Responsible consumption and production**				X
SDG 13.	**Climate action**				X
SDG 14.	**Life below water**				X
SDG 15.	**Life on land**				X
SDG 16.	**Peace, justice and strong institutions**				X
SDG 17.	**Partnerships for the goals**				X

The Sustainable Development Goals can be developed from two points of view, the first would be the development of a microcontroller, and the second would be the development related to hyperloop technology.

Regarding the microcontroller, the project is firmly related to goal nine, ”Industry, Innovation and Infrastructure”, concretely to target 9.B. This target aims to support domestic technology development. This project is based on an open-source architecture (RISC-V). It demonstrates the capabilities this architecture offers as an open-source standard to develop new technology without spending resources on proprietary architectures.

On the other side, hyperloop technology is related to goal 7 (”affordable and clean energy”) by focusing on creating a means of transportation with the lowest energy consumption possible. It is also associated with target 9.1, as the concept for the hyperloop infrastructure is to build an auto-sufficient infrastructure that would be powered by renewable energies.

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.

Part II

Plans & drawings

1 Scope 100

1 Scope

Due to the nature of the project, development of programmable logic, there are no plans to be added to this document.

Part III

Written specifications

1 Scope 102
2 Materials conditions 102
3 Test prior to commissioning 103

1 Scope

The written specifications document refers to the project "Development of a RISC-V processor optimised for control applications to be used in the levitation system of hyperloop". And it contains the work developed on the development and verification processes of the RISC-V processor. However, the work related to neither the electronics nor the guiding and levitation units of the hyperloop prototype Auran is not included in this document.

The selected programmable logic device chosen for this application is the FPGA 10M50DAF484C7G from the family Intel FPGA MAX10, implemented in the evaluation board DE10-Lite from Terasic.

In the development of the project, it has only been considered the particularities of the selected device. However, due to the availability of several alternatives for this device, the functioning of the work developed can only be assured in the aforementioned device.

The objective of this document is to define the criteria and requirements for the implementation of the project.

2 Materials conditions

In advance of the beginning of the project, it must be assured that the materials needed for the correct development of the project are available and that the software is properly installed and ready for operation.

2.1 Electronic devices

As mentioned above, the project is intended to be implemented to be used is the Terasic DE10-Lite evaluation board. However, it is possible to use other devices that must follow the following conditions. It must have at least 5,000 logic elements, 3,000 memory bits, 36 9-bit multipliers, two 12-bit ADC channels and one PLL. Also, it shall implement 3.3 V general-purpose input/output pins for the PWM outputs. Moreover, ADC channels shall be referenced at 5V.

2.2 Software

The software employed for the synthesis and programming of the FPGA is Quartus Prime Lite 18.1. It can only be assured that the project will function with the indicated

version; however, it is possible to employ other versions of Quartus (including other licensing types) as far as it supports the 10M50DAF484C7G FPGA.

For the development of the SystemVerilog code, any text editor able to work with plain text or SystemVerilog files could be used, even though it is recommended to use a software with SystemVerilog language support.

Finally, for the RISC-V assembly files development, any software able to assemble the RISC-V assembly files supporting the RV32I ISA can be used. If the assembly code is wanted to be modified, it is left to the user to ensure the correctness of the new code.

3 Test prior to commissioning

The project modules have been tested individually and as a whole, according to Section **5** in Part**I**, ensuring the correct operation of the RISC-V microcontroller. In case any modification is made to the original microcontroller, it must be verified before being introduced in the microcontroller.

Part IV

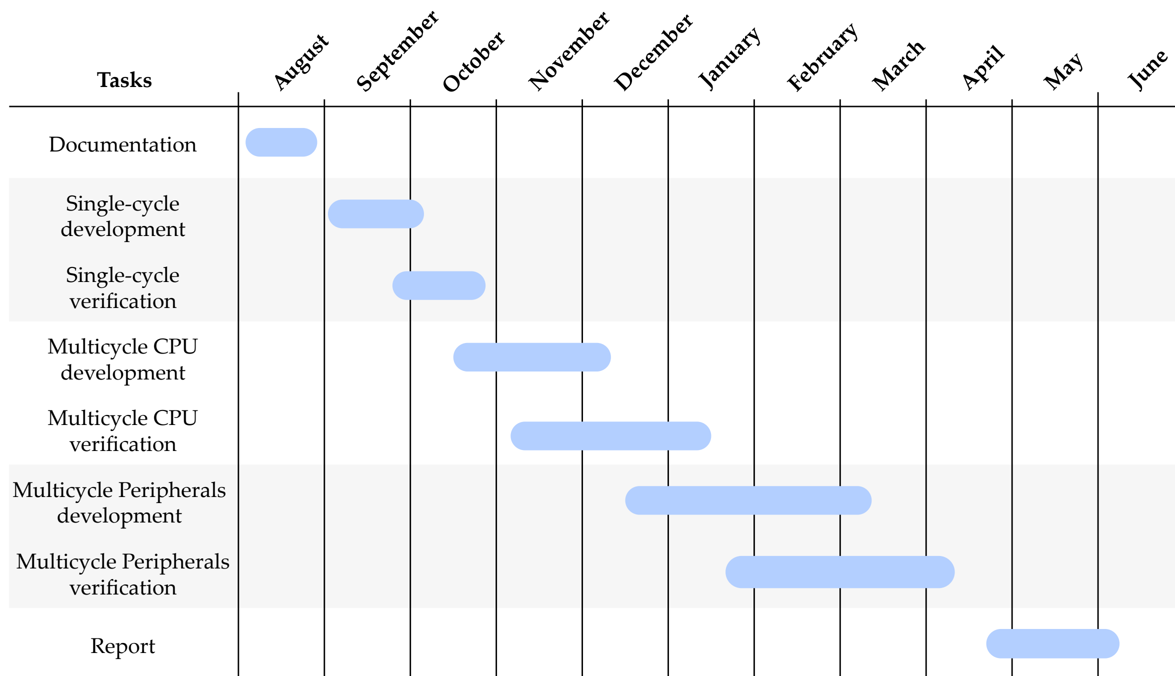
Project schedule

1 Gantt diagram 105

1 Gantt diagram

Figure **4.1.1** represents the scheduling of the project.

Figure 4.1.1: Gantt diagram



Part V

Budget

1 Materials 107
2 Labour 108
3 Proyect costs 108

The budget document encompasses the expenses associated with the development of the microcontroller for control applications. The budget is divided into materials and labour costs.

1 Materials

Firstly, the unitary prices for the equipment not used exclusively for the project must be calculated.

Table 5.1.1: Unitary cost calculation

Ref.	Unit	Description	Price	Life time	Cost/hour
E1	h	HP Omen 16-b1022ns	1700 €	12000 h	0.15 €/h

The materials chart represents the material costs at acquisition cost.

Table 5.1.2: Materials costs

Ref.	Description	Quantity	Unitary C.	Total
E1	HP Omen 16-b1022ns	500	0.15 €	75.00 €
E2	Terasic DE10-Lite Board	1	146.64 €	146.64 €
SW1	Visual Studio Code License	1	0.00 €	0.00 €
SW1.1	SystemVerilog - Language Support	1	0.00 €	0.00 €
SW1.2	RISC-V Venus Simulator	1	0.00 €	0.00 €
SW2	Venus	1	0.00 €	0.00 €
SW3	Quartus Prime Lite	1	0.00 €	0.00 €
SW4	ModelSim - Intel FPGA Starter Edition	1	0.00 €	0.00 €
SW5	MATLAB - Student Licence	1	69.00 €	69.00 €
SW5.1	Simulink	1	0.00 €	0.00 €
SW5.2	Control System Toolbox	1	0.00 €	0.00 €
			Subtotal:	**290.64 €**

2 Labour

Table 5.2.1: Labour unitary prices

Ref.	Unit	Description	Cost/hour
MP1	h	Development	15.00 €/h[1]
MP2	h	Verification	15.00 €/h

Table 5.2.2: Labour costs

Ref.	Description	Quantity	Unitary C.	Total
MP1	Development	200	15.00 €	3,000.00 €
MP2	Verification	300	15.00 €	4,500.00 €
			Subtotal:	**7,500.00 €**

3 Proyect costs

Table 5.3.1: Project costs

Description		Cost
Material costs		290.64 €
Labour costs		7,500.00 €
	Subtotal:	7,790.64 €
Industrial benefits (7%)		545.35 €
Additional expenses (10%)		779.07 €
	Before-tax:	9,115.06 €
VAT (21%)		1,914.16 €
	Total costs:	11,029.22 €

The total cost of the project (including VAT) is: **ELEVEN THOUSAND TWENTY-NINE EUROS AND TWENTY-TWO CENTS (11,029.22 €)**

[1]Unitary labour cost considered as the mean salary of graduated engineer

Part VI

Development files

1 Development 110
2 Verification 176

This document contains the development and verification files of the different modules explained throughout Part **I**. They are structured in the same way as in the Memory part. Additionally, a common modules section has been added for those shared between other modules.

1 Development

1.1 Common modules

Code snippet 6.1.1: Adder

```
//--------------------------------------------------------------
// Title          : Adder
//--------------------------------------------------------------
// File           : Adder.sv
// Author         : David Ramon Alaman
// Created        : 03.10.2022
//--------------------------------------------------------------

module Adder #(
    parameter WIDTH = 32
) (
    input logic[WIDTH - 1:0] in1, in2,
    output logic[WIDTH - 1:0] s
);

    assign s = in1 + in2;

endmodule
```

Code snippet 6.1.2: Register

```
//-----------------------------------------------------------------
// Title          : Resetable Flip flop with enable
//-----------------------------------------------------------------
// File           : FlipFlop.sv
// Author         : David Ramon Alaman
// Created        : 03.10.2022
//-----------------------------------------------------------------

module FlipFlop #(
    parameter WIDTH = 32
) (
    input logic clk, en, rst,
    input logic[WIDTH - 1:0] d,
    output logic[WIDTH - 1:0] q
);

    always_ff @(posedge clk) begin
        if(~rst) q <= 0;
        else if(en) q <= d;
    end

endmodule
```

Code snippet 6.1.3: 2-input register

```
//------------------------------------------------------------
// Title          : Resetable 2-input Flip flop with enable
//------------------------------------------------------------
// File           : FlipFlop2.sv
// Author         : David Ramon Alaman
// Created        : 16.10.2022
//------------------------------------------------------------

module FlipFlop2 #(
    parameter WIDTH = 32
) (
    input logic clk, en, srst, arst,
    input logic[WIDTH - 1:0] d0, d1,
    output logic[WIDTH - 1:0] q0, q1
);

    always_ff @(posedge clk or negedge arst) begin
        if(~arst) begin // Reset
            q0 <= 0;
            q1 <= 0;
        end
      else if(~srst) begin // Reset
        q0 <= 0;
        q1 <= 0;
      end
        else if(en) begin // If enable
            q0 <= d0;
            q1 <= d1;
        end
    end
endmodule
```

Code snippet 6.1.4: Full adder

```
//--------------------------------------------------------------
// Title          : Parameterised full adder
//--------------------------------------------------------------
// File           : fullAdder.sv
// Author         : David Ramon Alaman
// Created        : 18.09.2022
//--------------------------------------------------------------

module fullAdder #(
    parameter WIDTH = 32
) (
    input logic[WIDTH-1:0] operand1, operand2,
    input logic cin,
    output logic[WIDTH-1:0] result,
    output logic cout
);

    assign {cout, result} = operand1 + operand2 + cin;

endmodule
```

Code snippet 6.1.5: 2:1 multiplexer

```
//--------------------------------------------------------------
// Title          : 2 Input multiplexer
//--------------------------------------------------------------
// File           : mux2.sv
// Author         : David Ramon Alaman
// Created        : 18.09.2022
//--------------------------------------------------------------

module mux2 #(
    parameter WIDTH = 32
) (
    input logic[WIDTH-1:0] data1, data2,
    input logic s,
    output logic[WIDTH-1:0] dout
);

    assign dout = s ? data2 : data1;

endmodule
```

Code snippet 6.1.6: 3:1 multiplexer

```
//----------------------------------------------------------------
// Title          : 3 Input multiplexer
//----------------------------------------------------------------
// File           : mux3.sv
// Author         : David Ramon Alaman
// Created        : 03.10.2022
//----------------------------------------------------------------

module mux3 #(
    parameter WIDTH = 32
)  (
    input logic[WIDTH - 1:0] data0, data1, data2,
    input logic[1:0] s,
    output logic[WIDTH - 1:0] dout
);

    always_comb begin
        case (s)
            2'b00: dout = data0;
            2'b01: dout = data1;
            2'b10: dout = data2;
            default: dout = 'bx;
        endcase
    end

endmodule
```

Code snippet 6.1.7: 4:1 multiplexer

```
//-------------------------------------------------------------
// Title          : 4 Input multiplexer
//-------------------------------------------------------------
// File           : mux4.sv
// Author         : David Ramon Alaman
// Created        : 16.10.2022
//-------------------------------------------------------------

module mux4 #(
    parameter WIDTH = 32
) (
    input logic[WIDTH - 1:0] data0, data1, data2, data3,
    input logic[1:0] s,
    output logic[WIDTH - 1:0] dout
);

    always_comb begin
        case (s)
            2'b00: dout = data0;
            2'b01: dout = data1;
            2'b10: dout = data2;
            2'b11: dout = data3;
            default: dout = 'bx;
        endcase
    end

endmodule
```

Code snippet 6.1.8: 6:1 multiplexer

```
//-------------------------------------------------------------
// Title          : 6 Input multiplexer
//-------------------------------------------------------------
// File           : mux6.sv
// Author         : David Ramon Alaman
// Created        : 16.10.2022
//-------------------------------------------------------------

module mux6 #(
    parameter WIDTH = 32
) (
    input logic[WIDTH - 1:0] data0, data1, data2, data3, data4, data5,
    input logic[2:0] s,
    output logic[WIDTH - 1:0] dout
);

    always_comb begin
        case (s)
            3'b000: dout = data0;
            3'b001: dout = data1;
            3'b010: dout = data2;
            3'b011: dout = data3;
            3'b100: dout = data4;
            3'b101: dout = data5;
            default: dout = 'bx;
        endcase
    end

endmodule
```

Code snippet 6.1.9: 7:1 multiplexer

```
//---------------------------------------------------------------
// Title          : 7 Input multiplexer
//---------------------------------------------------------------
// File           : mux7.sv
// Author         : David Ramon Alaman
// Created        : 25.05.2023
//---------------------------------------------------------------

module mux7 #(
    parameter WIDTH = 32
) (
    input logic[WIDTH - 1:0] data0, data1, data2, data3, data4, data5,
    ↪ data6,
    input logic[2:0] s,
    output logic[WIDTH - 1:0] dout
);

    always_comb begin
        case (s)
            3'b000: dout = data0;
            3'b001: dout = data1;
            3'b010: dout = data2;
            3'b011: dout = data3;
            3'b100: dout = data4;
            3'b101: dout = data5;
            3'b110: dout = data6;
            default: dout = 'bx;
        endcase
    end

endmodule
```

Code snippet 6.1.10: 10:1 multiplexer

```
//-------------------------------------------------------------
// Title          : 10 Input multiplexer
//-------------------------------------------------------------
// File           : mux10.sv
// Author         : David Ramon Alaman
// Created        : 18.09.2022
//-------------------------------------------------------------

module mux10 (
    input logic[31:0] data[9:0],
    input logic[3:0] s,
    output logic[31:0] dout
);

    always_comb begin
        if(s < 10) begin
            dout = data[s];
        end
        else begin
            dout = 32'bx;
        end
    end
endmodule
```

Code snippet 6.1.11: Prescaler

```
//-----------------------------------------------------------------
// Title          : Prescaler
//-----------------------------------------------------------------
// File           : Prescaler.sv
// Author         : David Ramon Alaman
// Created        : 24.12.2022
//-----------------------------------------------------------------

module Prescaler #(
    parameter INITIAL_VALUE = 1
)
(
    input logic clk, arst, en,
    input logic[31:0] new_value,
    output logic clk_en
);

logic[31:0] count, value;

always @(posedge clk or negedge arst) begin
    if(~arst) begin
        count <= 0;
        value <= INITIAL_VALUE;
    end

    else if (en) begin
        count <= count + 1;

        if(count === value) begin
            count <= 0;
            clk_en <= 1;
            value <= new_value;
        end
        else begin
            clk_en <= 0;
        end
    end
end

endmodule
```

Code snippet 6.1.12: Zero extender

```
//-------------------------------------------------------------
// Title          : Zero extender
//-------------------------------------------------------------
// File           : zeroExtender.sv
// Author         : David Ramon Alaman
// Created        : 19.09.2022
//-------------------------------------------------------------

module zeroExtender (
    input logic inbit,
    output logic[31:0] extended
);

    assign extended = {31'b0, inbit};

endmodule
```

1.2 Single-cycle

1.2.1 Datapath

1.2.1.1 Register file

Code snippet 6.1.13: Register file

```
//---------------------------------------------------------------
// Title          : Register File
//---------------------------------------------------------------
// File           : RegisterFile.sv
// Author         : David Ramon Alaman
// Created        : 17.09.2022
//---------------------------------------------------------------

module RegisterFile(
    input logic clk, we, rst,
    input logic[4:0] reg1, reg2, reg3,
    input logic[31:0] dataIn,
    output logic[31:0] regData1, regData2
);

    logic[31:0] registers[31:0];

    // Asynchronous read
    assign regData1 = (reg1 != 0) ? registers[reg1] : 32'b0;
    assign regData2 = (reg2 != 0) ? registers[reg2] : 32'b0;

    // Synchronous write
    always_ff @(posedge clk or negedge rst) begin
        if (~rst) begin
            registers <= '{default: '0};
        end
        else if(we) begin // Write enable
            registers[reg3] <= dataIn;
        end
    end

endmodule
```

1.2.1.2 Immediate generator

Code snippet 6.1.14: Immediate generator

```
//-------------------------------------------------------------
// Title          : Immediate Generator for RISC-V
//-------------------------------------------------------------
// File           : ImmediateGenerator.sv
// Author         : David Ramon Alaman
// Created        : 17.09.2022
//-------------------------------------------------------------

module ImmediateGenerator(
    input logic[31:0] instruction,
    output logic[31:0] immediate
);

    always_comb begin
        case (instruction[6:0]) // Opcode
            7'b0000011, 7'b0010011, 7'b1100111: immediate = {{21{
    ↪ instruction[31]}}, instruction[30:20]}; // I type
            7'b0100011: immediate = {{21{instruction[31]}}, instruction
    ↪ [30:25], instruction[11:7]}; // S type
            7'b1100011: immediate = {{20{instruction[31]}}, instruction[7],
    ↪  instruction[30:25], instruction[11:8], 1'b0}; // B type
            7'b0010111, 7'b0110111: immediate = {instruction[31:12], 12'b0
    ↪ }; // U type
            7'b1101111: immediate = {{12{instruction[31]}}, instruction
    ↪ [19:12], instruction[20], instruction[30:21], 1'b0}; // J type
            default: immediate = 32'dx;
        endcase
    end
endmodule
```

1.2.1.3 Arithmetic Logic Unit

Code snippet 6.1.15: Arithmetic Logic Unit

```
//--------------------------------------------------------------
// Title          : Arithmetic Logic Unit
//--------------------------------------------------------------
// File           : ALU.sv
// Author         : David Ramon Alaman
// Created        : 18.09.2022
//--------------------------------------------------------------

module ALU(
    input logic[31:0] operand1, operand2,
    input logic[3:0] control,
    output logic[31:0] result,
    output logic[3:0] flags
);

    logic[31:0] muxedOperand2, sum, ored, anded, xored, slt, sll, srl, sra,
    ↪  sltu;
    logic cout, sltnonext, sltunonext;

    // Allows to use only one full adder
    mux2 substractionmux(operand2, ~operand2, control[0], muxedOperand2);

    // Addition and substraction adder
    fullAdder adder(operand1, muxedOperand2, control[0], sum, cout);

    // OR, AND and XOR operations
    assign ored = operand1 | operand2;
    assign anded = operand1 & operand2;
    assign xored = operand1 ^ operand2;

    //SLL, SRL and SRA operations;
    assign sll = operand1 << operand2[4:0];
    assign srl = operand1 >> operand2[4:0];
    assign sra = $signed(operand1) >>> operand2[4:0];

    // SLT implementation
    assign sltnonext = sum[31] ^ flags[3];
    zeroExtender zero(sltnonext, slt);

    // SLTU implementation
    assign sltunonext = ~flags[2];
    zeroExtender zero_u(sltunonext, sltu);

```

```
43     // Flags
44     // Flags = {Overflow, Carry, Negative, Zero}
45 
46     assign flags[0] = &(~result);
47     assign flags[1] = result[31];
48     assign flags[2] = cout & ~control[1];
49     assign flags[3] = ~(control[0] ^ operand1[31] ^ operand2[31]) &
50         (operand1[31] ^ sum[31]) & ~control[1];
51 
52     // Result
53   mux10 resultmux('{sltu, sra, srl, sll, slt, xored, ored, anded, sum, sum
     ↪ }, control, result);
54 
55 endmodule
```

1.2.1.4 Data memory

Code snippet 6.1.16: Data memory

```
//--------------------------------------------------------------
// Title          : Data Memory
//--------------------------------------------------------------
// File           : RAM.sv
// Author         : David Ramon Alaman
// Created        : 22.09.2022
//--------------------------------------------------------------

module RAM #(
    parameter DEPTH = 1024,
    parameter WORD_SIZE = 32
)(
    input logic clk, we,
    input logic[$clog2(DEPTH)-1:0] addr,
    input logic[WORD_SIZE-1:0] dataIn,
    output logic[WORD_SIZE-1:0] dataOut
);
    logic[WORD_SIZE-1:0] ram[DEPTH-1:0];

    assign dataOut = ram[addr];

    always @(posedge clk) begin
        if(we) begin
            ram[addr] <= dataIn;
        end
    end

endmodule
```

1.2.1.5 Instruction memory

Code snippet 6.1.17: Instruction memory

```
//------------------------------------------------------------------
// Title          : Instruction Memory
//------------------------------------------------------------------
// File           : ROM.sv
// Author         : David Ramon Alaman
// Created        : 17.09.2022
//------------------------------------------------------------------

module ROM #(
    parameter DEPTH = 1024
)
(
    input logic[31:0] pc,
    output logic[31:0] instruction
);

    logic[31:0] memory[DEPTH-1:0];

    initial begin
        $readmemh("test_instructions.txt", memory);
    end

    assign instruction = memory[pc[$clog2(DEPTH)+1:2]];

endmodule
```

1.2.2 Control logic

1.2.2.1 Control unit

Code snippet 6.1.18: Control unit

```
//-----------------------------------------------------------------
// Title          : Control module
//-----------------------------------------------------------------
// File           : Control.sv
// Author         : David Ramon Alaman
// Created        : 30.09.2022
//-----------------------------------------------------------------

module Control(
    input logic[6:0] opCode,
    output logic branch, forceJump, RAMwe, Regwe, ALUSrc2,
    output logic[1:0] ALUOp, ALUSrc1, RegWriteSrc
);

    always_comb begin
        case (opCode)
            7'd3: begin
                branch =      1'b0;
                forceJump =   1'b0;
                RAMwe =       1'b0;
                Regwe =       1'b1;
                ALUSrc2 =     1'b0;
                ALUOp =       2'b10;
                ALUSrc1 =     2'b00;
                RegWriteSrc = 2'b00;
            end
            7'd19: begin
                branch =      1'b0;
                forceJump =   1'b0;
                RAMwe =       1'b0;
                Regwe =       1'b1;
                ALUSrc2 =     1'b0;
                ALUOp =       2'b00;
                ALUSrc1 =     2'b00;
                RegWriteSrc = 2'b01;
            end
            7'd23: begin
                branch =      1'b0;
                forceJump =   1'b0;
                RAMwe =       1'b0;
                Regwe =       1'b1;
```

```
42                  ALUSrc2 =      1'b0;
43                  ALUOp =        2'b10;
44                  ALUSrc1 =      2'b10;
45                  RegWriteSrc = 2'b01;
46              end
47              7'd35: begin
48                  branch =       1'b0;
49                  forceJump =    1'b0;
50                  RAMwe =        1'b1;
51                  Regwe =        1'b0;
52                  ALUSrc2 =      1'b0;
53                  ALUOp =        2'b10;
54                  ALUSrc1 =      2'b00;
55                  RegWriteSrc = 2'b01;
56              end
57              7'd51: begin
58                  branch =       1'b0;
59                  forceJump =    1'b0;
60                  RAMwe =        1'b0;
61                  Regwe =        1'b1;
62                  ALUSrc2 =      1'b1;
63                  ALUOp =        2'b00;
64                  ALUSrc1 =      2'b00;
65                  RegWriteSrc = 2'b01;
66              end
67              7'd55: begin
68                  branch =       1'b0;
69                  forceJump =    1'b0;
70                  RAMwe =        1'b0;
71                  Regwe =        1'b1;
72                  ALUSrc2 =      1'b0;
73                  ALUOp =        2'b10;
74                  ALUSrc1 =      2'b01;
75                  RegWriteSrc = 2'b01;
76              end
77              7'd99: begin
78                  branch =       1'b1;
79                  forceJump =    1'b0;
80                  RAMwe =        1'b0;
81                  Regwe =        1'b0;
82                  ALUSrc2 =      1'b1;
83                  ALUOp =        2'b01;
84                  ALUSrc1 =      2'b00;
85                  RegWriteSrc = 2'b01;
86              end
87              7'd103: begin
```

```
88                      branch =       1'b0;
89                      forceJump =    1'b1;
90                      RAMwe =        1'b0;
91                      Regwe =        1'b1;
92                      ALUSrc2 =      1'b0;
93                      ALUOp =        2'b10;
94                      ALUSrc1 =      2'b00;
95                      RegWriteSrc = 2'b10;
96                  end
97                  7'd111: begin
98                      branch =       1'b0;
99                      forceJump =    1'b1;
100                     RAMwe =        1'b0;
101                     Regwe =        1'b1;
102                     ALUSrc2 =      1'b0;
103                     ALUOp =        2'b10;
104                     ALUSrc1 =      2'b00;
105                     RegWriteSrc = 2'b10;
106                 end
107                 default: begin
108                     branch =       1'bx;
109                     forceJump =    1'bx;
110                     RAMwe =        1'bx;
111                     Regwe =        1'bx;
112                     ALUSrc2 =      1'bx;
113                     ALUOp =        2'bx;
114                     ALUSrc1 =      2'bx;
115                     RegWriteSrc = 2'bx;
116                 end
117             endcase
118         end
119 endmodule
```

1.2.2.2 Branch logic

Code snippet 6.1.19: Branch logic

```
//---------------------------------------------------------------
// Title         : Branch Logic
//---------------------------------------------------------------
// File          : BranchLogic.sv
// Author        : David Ramon Alaman
// Created       : 27.09.2022
//---------------------------------------------------------------

module BranchLogic(
    input logic branch, forceJump, opCode_3,
    input logic[2:0] funct3,
    input logic[3:0] flags,
    output logic[1:0] PCSrc
);

    always_comb begin
        if(forceJump) begin
            if(opCode_3) PCSrc = 2'b01;
            else PCSrc = 2'b10;
        end

        else if(branch) begin
            PCSrc[1] = 0;
            case (funct3)
                3'b000: PCSrc[0] = flags[0]; // beq
                3'b001: PCSrc[0] = ~flags[0]; // bne
                3'b100: PCSrc[0] = flags[1] ^ flags[3]; // blt
                3'b101: PCSrc[0] = ~(flags[1] ^ flags[3]); // bge
                3'b110: PCSrc[0] = ~flags[2]; // bltu
                3'b111: PCSrc[0] = flags[2]; // bgeu
                default: PCSrc[0] = 0;
            endcase
        end

        else PCSrc = 2'b0;
    end
endmodule
```

1.2.2.3 ALU Control

Code snippet 6.1.20: ALU Control

```
//------------------------------------------------------------------
// Title          : Arithmetic Logic Unit Controller
//------------------------------------------------------------------
// File           : ALUControl.sv
// Author         : David Ramon Alaman
// Created        : 27.09.2022
//------------------------------------------------------------------

module ALUControl(
    input logic[1:0] ALUOp,
    input logic[4:0] func, // {Funct3, Funct7_5, OP_5}
    output logic[3:0] ALUCtrl
);

    always_comb begin
        case (ALUOp)
            2'b00: begin
                casez (func[4:1])
                    4'b000?: begin
                        if(func[0] === 1) begin
                            if(func[1] === 0) ALUCtrl = 4'd0; // add
                            else ALUCtrl = 4'd1; // sub
                        end
                        else ALUCtrl = 4'd0; // addi
                    end
                    4'b0010: ALUCtrl = 4'd6; // sll, slli
                    4'b010?: ALUCtrl = 4'd5; // slt, slti
                    4'b011?: ALUCtrl = 4'd9; // sltu, sltiu
                    4'b100?: ALUCtrl = 4'd4; // xor, xori
                    4'b1010: ALUCtrl = 4'd7; // srl, srli
                    4'b1011: ALUCtrl = 4'd8; // sra, srai
                    4'b110?: ALUCtrl = 4'd3; // or, ori
                    4'b111?: ALUCtrl = 4'd2; // and, andi
                    default: ALUCtrl = 4'bx;
                endcase
                end
            2'b01: ALUCtrl = 4'd1;
            2'b10: ALUCtrl = 4'd0;
            default: ALUCtrl = 4'bx;
        endcase
    end
endmodule
```

1.2.3 Processor implementation

Code snippet 6.1.21: RV32I - Single cycle

```
//------------------------------------------------------------------
// Title          : Risc-V 32bit Integer Single Cycle
//------------------------------------------------------------------
// File           : RV32I_SC.sv
// Author         : David Ramon Alaman
// Created        : 29.09.2022
//------------------------------------------------------------------

module RV32I_SC #(
    parameter ROM_SIZE = 64,
    parameter RAM_DEPTH = 64,
    parameter WORD_SIZE = 32
)
(
    input logic clk, en, rst,
    input logic[WORD_SIZE - 1:0] ramOut,
    input logic[31:0] instruction,
    output logic RAMwe,
    output logic[$clog2(RAM_DEPTH)-1:0] RAMaddr,
    output logic[WORD_SIZE - 1:0] rs2,
    output logic[31:0] pc
);

    // Signal definition
    logic Regwe, branch, forceJump, ALUSrc2;
    logic[1:0] ALUOp, ALUSrc1, RegWriteSrc, PCSrc;
    logic[3:0] ALUCtrl, flags;
    logic[31:0] rd, rs1, ALUIn1, ALUIn2, ALUOut, immediate, pcAdded,
        pcImmediate, nextPC;

    assign RAMaddr = ALUOut[$clog2(RAM_DEPTH)+1 : $clog2(WORD_SIZE/8)];

    // Datapath modules instantiation
    RegisterFile RV_RegisterFile(
        .clk(clk),
        .we(Regwe),
        .rst(rst),
        .reg1(instruction[19:15]),
        .reg2(instruction[24:20]),
        .reg3(instruction[11:7]),
        .dataIn(rd),
        .regData1(rs1),
        .regData2(rs2)
```

```
44     );
45 
46     ALU RV_ALU (
47         .operand1(ALUIn1),
48         .operand2(ALUIn2),
49         .control(ALUCtrl),
50         .result(ALUOut),
51         .flags(flags)
52     );
53 
54     ImmediateGenerator RV_ImmGenerator(
55         .instruction(instruction),
56         .immediate(immediate)
57     );
58 
59     // Control modules instantiation
60     Control RV_Control(
61         .opCode(instruction[6:0]),
62         .branch(branch),
63         .forceJump(forceJump),
64         .RAMwe(RAMwe),
65         .Regwe(Regwe),
66         .ALUSrc2(ALUSrc2),
67         .ALUOp(ALUOp),
68         .ALUSrc1(ALUSrc1),
69         .RegWriteSrc(RegWriteSrc)
70     );
71 
72     BranchLogic RV_BranchLogic(
73         .branch(branch),
74         .forceJump(forceJump),
75         .opCode_3(instruction[3]),
76         .funct3(instruction[14:12]),
77         .flags(flags),
78         .PCSrc(PCSrc)
79     );
80 
81     ALUControl RV_ALUControl(
82         .ALUOp(ALUOp),
83         .func({instruction[14:12], instruction[30], instruction[5]}),
84         .ALUCtrl(ALUCtrl)
85     );
86 
87     // Adders and multiplexers
88     Adder RV_PCAdder(
89         .in1('d4),
```

```
90          .in2(pc),
91          .s(pcAdded)
92      );
93
94      Adder RV_PCImmediateAdder(
95          .in1(pc),
96          .in2(immediate),
97          .s(pcImmediate)
98      );
99
100     mux3 RV_ALUSrcMux1(
101         .data0(rs1),
102         .data1('d0),
103         .data2(pc),
104         .s(ALUSrc1),
105         .dout(ALUIn1)
106     );
107
108     mux3 RV_RdSrcMux(
109         .data0(ramOut),
110         .data1(ALUOut),
111         .data2(pcAdded),
112         .s(RegWriteSrc),
113         .dout(rd)
114     );
115
116     mux3 RV_PCSrcMux(
117         .data0(pcAdded),
118         .data1(pcImmediate),
119         .data2(ALUOut),
120         .s(PCSrc),
121         .dout(nextPC)
122     );
123
124     mux2 RV_ALUSrcMux2(
125         .data1(immediate),
126         .data2(rs2),
127         .s(ALUSrc2),
128         .dout(ALUIn2)
129     );
130
131     FlipFlop RV_PCFlipFlop(
132         .clk(clk),
133         .en(en),
134         .rst(rst),
135         .d(nextPC),
```

```
136         .q(pc)
137     );
138 
139 endmodule
```

1.3 Multicycle

1.3.1 Datapath

1.3.1.1 Memory

Code snippet 6.1.22: Memory

```
//-----------------------------------------------------------------
// Title          : Memory
//-----------------------------------------------------------------
// File           : Memory.sv
// Author         : David Ramon Alaman
// Created        : 22.09.2022
//-----------------------------------------------------------------

module Memory #(
    parameter DEPTH = 10,
    parameter WORD_SIZE = 32,
    parameter INITIALIZE = 0,
    parameter FILE = "Memory.txt"
)(
    input logic clk, we, read, cs,
    input logic[$clog2(DEPTH)-1:0] addr,
    input logic[WORD_SIZE-1:0] dataIn,
    output logic[WORD_SIZE-1:0] dataOut
);
    logic[WORD_SIZE-1:0] memory[DEPTH-1:0];

    initial begin
        if(INITIALIZE == 1'b1) begin
            $readmemh(FILE, memory);
        end
    end

    always @(posedge clk) begin
        if(cs) begin
            if(we) begin
                memory[addr] <= dataIn;
            end
            else if(read) begin
                dataOut <= memory[addr];
            end
        end
    end
endmodule
```

1.3.2 Control logic

Code snippet 6.1.23: Control unit

```
//--------------------------------------------------------------
// Title         : Control module
//--------------------------------------------------------------
// File          : Control.sv
// Author        : David Ramon Alaman
// Created       : 16.10.2022
//--------------------------------------------------------------

module Control(
    input logic clk, arst, en, branch,
    input logic[6:0] opCode,
    output logic PCRegEN, FetchRegEN, RFRegEN, ALUOutRegEN, DataRegEN,
        PCRegRST, FetchRegRST, RFRegRST,
        ALUOutRegRST, memWE, Regwe, InstructionRead,
    output logic[1:0] ALUIn1Src, ALUIn2Src, ResultSrc, ALUOp
);

    typedef enum logic [3:0] {Fetch, Decode, ExecuteR, ExecuteI,
        ExecuteLui, ExecuteAuipc, PCUpdateI, PCUpdateJ,
        RdCalculation, Comparison, RegisterWrite, PCUpdateB,
        memoryAddr, memoryRead, memoryWrite, memoryWriteBack
        } 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)
            Fetch: nextstate = Decode;
            Decode:
                case(opCode)
                    7'b0000011, 7'b0100011: nextstate = memoryAddr;
                    7'b0010011: nextstate = ExecuteI;
                    7'b0110011: nextstate = ExecuteR;
                    7'b0010111: nextstate = ExecuteAuipc;
                    7'b0110111: nextstate = ExecuteLui;
                    7'b1100111: nextstate = PCUpdateI;
                    7'b1101111: nextstate = PCUpdateJ;
```

```
44                     7'b1100011: nextstate = Comparison;
45                     default: nextstate = Fetch;
46                 endcase
47             memoryAddr:
48                 if(opCode[5]) nextstate = memoryWrite;
49                 else nextstate = memoryRead;
50             memoryWrite: nextstate = Fetch;
51             memoryRead: nextstate = memoryWriteBack;
52             memoryWriteBack: nextstate = Fetch;
53             ExecuteR: nextstate = RegisterWrite;
54             ExecuteI: nextstate = RegisterWrite;
55             ExecuteAuipc: nextstate = RegisterWrite;
56             ExecuteLui: nextstate = RegisterWrite;
57             PCUpdateI: nextstate = RdCalculation;
58             PCUpdateJ: nextstate = RdCalculation;
59             RdCalculation: nextstate = RegisterWrite;
60             RegisterWrite: nextstate = Fetch;
61             Comparison:
62                 if(branch) nextstate = PCUpdateB;
63                 else nextstate = Fetch;
64             PCUpdateB: nextstate = Fetch;
65             default: nextstate = Fetch;
66         endcase
67     end
68 
69     // Output logic
70     always_comb begin
71         case(state)
72             Fetch: begin
73                 PCRegEN = 1'b1;
74                 FetchRegEN = 1'b1;
75                 InstructionRead = 1'b1;
76                 RFRegEN = 1'b0;
77                 ALUOutRegEN = 1'b0;
78                 DataRegEN = 1'b0;
79 
80                 PCRegRST = 1'b1;
81                 FetchRegRST = 1'b1;
82                 RFRegRST = 1'b1;
83                 ALUOutRegRST = 1'b1;
84 
85                 ALUIn1Src = 2'b11;
86                 ALUIn2Src = 2'b10;
87                 ResultSrc = 2'b01;
88 
89                 memWE = 1'b0;
```

```
90                  Regwe = 1'b0;
91
92                  ALUOp = 2'b10;
93              end
94
95              Decode: begin
96                  PCRegEN = 1'b0;
97                  FetchRegEN = 1'b0;
98                  InstructionRead = 1'b0;
99                  RFRegEN = 1'b1;
100                 ALUOutRegEN = 1'b0;
101                 DataRegEN = 1'b0;
102
103                 PCRegRST = 1'b1;
104                 FetchRegRST = 1'b1;
105                 RFRegRST = 1'b1;
106                 ALUOutRegRST = 1'b1;
107
108                 ALUIn1Src = 2'b00;
109                 ALUIn2Src = 2'b00;
110                 ResultSrc = 2'b00;
111
112                 memWE = 1'b0;
113                 Regwe = 1'b0;
114
115                 ALUOp = 2'b00;
116             end
117
118             memoryAddr: begin
119                 PCRegEN = 1'b0;
120                 FetchRegEN = 1'b0;
121                 InstructionRead = 1'b0;
122                 RFRegEN = 1'b0;
123                 ALUOutRegEN = 1'b1;
124                 DataRegEN = 1'b0;
125
126                 PCRegRST = 1'b1;
127                 FetchRegRST = 1'b1;
128                 RFRegRST = 1'b1;
129                 ALUOutRegRST = 1'b1;
130
131                 ALUIn1Src = 2'b00;
132                 ALUIn2Src = 2'b00;
133                 ResultSrc = 2'b00;
134
135                 memWE = 1'b0;
```

```
136                 Regwe = 1'b0;
137
138                 ALUOp = 2'b10;
139             end
140
141             memoryRead: begin
142                 PCRegEN = 1'b0;
143                 FetchRegEN = 1'b0;
144                 InstructionRead = 1'b0;
145                 RFRegEN = 1'b0;
146                 ALUOutRegEN = 1'b0;
147                 DataRegEN = 1'b1;
148
149                 PCRegRST = 1'b1;
150                 FetchRegRST = 1'b1;
151                 RFRegRST = 1'b1;
152                 ALUOutRegRST = 1'b1;
153
154                 ALUIn1Src = 2'b00;
155                 ALUIn2Src = 2'b00;
156                 ResultSrc = 2'b00;
157
158                 memWE = 1'b0;
159                 Regwe = 1'b0;
160
161                 ALUOp = 2'b00;
162             end
163
164             memoryWriteBack: begin
165                 PCRegEN = 1'b0;
166                 FetchRegEN = 1'b0;
167                 InstructionRead = 1'b0;
168                 RFRegEN = 1'b0;
169                 ALUOutRegEN = 1'b0;
170                 DataRegEN = 1'b0;
171
172                 PCRegRST = 1'b1;
173                 FetchRegRST = 1'b1;
174                 RFRegRST = 1'b1;
175                 ALUOutRegRST = 1'b1;
176
177                 ALUIn1Src = 2'b00;
178                 ALUIn2Src = 2'b00;
179                 ResultSrc = 2'b10;
180
181                 memWE = 1'b0;
```

```
182                    Regwe = 1'b1;
183
184                    ALUOp = 2'b00;
185                end
186
187                memoryWrite: begin
188                    PCRegEN = 1'b0;
189                    FetchRegEN = 1'b0;
190                    InstructionRead = 1'b0;
191                    RFRegEN = 1'b0;
192                    ALUOutRegEN = 1'b0;
193                    DataRegEN = 1'b0;
194
195                    PCRegRST = 1'b1;
196                    FetchRegRST = 1'b1;
197                    RFRegRST = 1'b1;
198                    ALUOutRegRST = 1'b1;
199
200                    ALUIn1Src = 2'b00;
201                    ALUIn2Src = 2'b00;
202                    ResultSrc = 2'b00;
203
204                    memWE = 1'b1;
205                    Regwe = 1'b0;
206
207                    ALUOp = 2'b00;
208                end
209
210                ExecuteR: begin
211                    PCRegEN = 1'b0;
212                    FetchRegEN = 1'b0;
213                    InstructionRead = 1'b0;
214                    RFRegEN = 1'b0;
215                    ALUOutRegEN = 1'b1;
216                    DataRegEN = 1'b0;
217
218                    PCRegRST = 1'b1;
219                    FetchRegRST = 1'b1;
220                    RFRegRST = 1'b1;
221                    ALUOutRegRST = 1'b1;
222
223                    ALUIn1Src = 2'b00;
224                    ALUIn2Src = 2'b01;
225                    ResultSrc = 2'b00;
226
227                    memWE = 1'b0;
```

```
228                 Regwe = 1'b0;
229 
230                 ALUOp = 2'b00;
231             end
232 
233             ExecuteI: begin
234                 PCRegEN = 1'b0;
235                 FetchRegEN = 1'b0;
236                 InstructionRead = 1'b0;
237                 RFRegEN = 1'b0;
238                 ALUOutRegEN = 1'b1;
239                 DataRegEN = 1'b0;
240 
241                 PCRegRST = 1'b1;
242                 FetchRegRST = 1'b1;
243                 RFRegRST = 1'b1;
244                 ALUOutRegRST = 1'b1;
245 
246                 ALUIn1Src = 2'b00;
247                 ALUIn2Src = 2'b00;
248                 ResultSrc = 2'b00;
249 
250                 memWE = 1'b0;
251                 Regwe = 1'b0;
252 
253                 ALUOp = 2'b00;
254             end
255 
256             ExecuteAuipc: begin
257                 PCRegEN = 1'b0;
258                 FetchRegEN = 1'b0;
259                 InstructionRead = 1'b0;
260                 RFRegEN = 1'b0;
261                 ALUOutRegEN = 1'b1;
262                 DataRegEN = 1'b0;
263 
264                 PCRegRST = 1'b1;
265                 FetchRegRST = 1'b1;
266                 RFRegRST = 1'b1;
267                 ALUOutRegRST = 1'b1;
268 
269                 ALUIn1Src = 2'b10;
270                 ALUIn2Src = 2'b00;
271                 ResultSrc = 2'b00;
272 
273                 memWE = 1'b0;
```

```
274                 Regwe = 1'b0;
275
276                 ALUOp = 2'b10;
277             end
278
279             ExecuteLui: begin
280                 PCRegEN = 1'b0;
281                 FetchRegEN = 1'b0;
282                 InstructionRead = 1'b0;
283                 RFRegEN = 1'b0;
284                 ALUOutRegEN = 1'b1;
285                 DataRegEN = 1'b0;
286
287                 PCRegRST = 1'b1;
288                 FetchRegRST = 1'b1;
289                 RFRegRST = 1'b1;
290                 ALUOutRegRST = 1'b1;
291
292                 ALUIn1Src = 2'b01;
293                 ALUIn2Src = 2'b00;
294                 ResultSrc = 2'b00;
295
296                 memWE = 1'b0;
297                 Regwe = 1'b0;
298
299                 ALUOp = 2'b10;
300             end
301
302             PCUpdateI: begin
303                 PCRegEN = 1'b1;
304                 FetchRegEN = 1'b0;
305                 InstructionRead = 1'b0;
306                 RFRegEN = 1'b0;
307                 ALUOutRegEN = 1'b0;
308                 DataRegEN = 1'b0;
309
310                 PCRegRST = 1'b1;
311                 FetchRegRST = 1'b1;
312                 RFRegRST = 1'b1;
313                 ALUOutRegRST = 1'b1;
314
315                 ALUIn1Src = 2'b00;
316                 ALUIn2Src = 2'b00;
317                 ResultSrc = 2'b01;
318
319                 memWE = 1'b0;
```

```
320                 Regwe = 1'b0;
321 
322                 ALUOp = 2'b10;
323             end
324 
325             PCUpdateJ: begin
326                 PCRegEN = 1'b1;
327                 FetchRegEN = 1'b0;
328                 InstructionRead = 1'b0;
329                 RFRegEN = 1'b0;
330                 ALUOutRegEN = 1'b0;
331                 DataRegEN = 1'b0;
332 
333                 PCRegRST = 1'b1;
334                 FetchRegRST = 1'b1;
335                 RFRegRST = 1'b1;
336                 ALUOutRegRST = 1'b1;
337 
338                 ALUIn1Src = 2'b10;
339                 ALUIn2Src = 2'b00;
340                 ResultSrc = 2'b01;
341 
342                 memWE = 1'b0;
343                 Regwe = 1'b0;
344 
345                 ALUOp = 2'b10;
346             end
347 
348             RdCalculation: begin
349                 PCRegEN = 1'b0;
350                 FetchRegEN = 1'b0;
351                 InstructionRead = 1'b0;
352                 RFRegEN = 1'b0;
353                 ALUOutRegEN = 1'b1;
354                 DataRegEN = 1'b0;
355 
356                 PCRegRST = 1'b1;
357                 FetchRegRST = 1'b1;
358                 RFRegRST = 1'b1;
359                 ALUOutRegRST = 1'b1;
360 
361                 ALUIn1Src = 2'b10;
362                 ALUIn2Src = 2'b10;
363                 ResultSrc = 2'b00;
364 
365                 memWE = 1'b0;
```

```
366                     Regwe = 1'b0;
367
368                     ALUOp = 2'b10;
369                 end
370
371                 Comparison: begin
372                     PCRegEN = 1'b0;
373                     FetchRegEN = 1'b0;
374                     InstructionRead = 1'b0;
375                     RFRegEN = 1'b0;
376                     ALUOutRegEN = 1'b0;
377                     DataRegEN = 1'b0;
378
379                     PCRegRST = 1'b1;
380                     FetchRegRST = 1'b1;
381                     RFRegRST = 1'b1;
382                     ALUOutRegRST = 1'b1;
383
384                     ALUIn1Src = 2'b00;
385                     ALUIn2Src = 2'b01;
386                     ResultSrc = 2'b00;
387
388                     memWE = 1'b0;
389                     Regwe = 1'b0;
390
391                     ALUOp = 2'b01;
392                 end
393
394                 PCUpdateB: begin
395                     PCRegEN = 1'b1;
396                     FetchRegEN = 1'b0;
397                     InstructionRead = 1'b0;
398                     RFRegEN = 1'b0;
399                     ALUOutRegEN = 1'b0;
400                     DataRegEN = 1'b0;
401
402                     PCRegRST = 1'b1;
403                     FetchRegRST = 1'b1;
404                     RFRegRST = 1'b1;
405                     ALUOutRegRST = 1'b1;
406
407                     ALUIn1Src = 2'b10;
408                     ALUIn2Src = 2'b00;
409                     ResultSrc = 2'b01;
410
411                     memWE = 1'b0;
```

```
412                 Regwe = 1'b0;
413 
414                 ALUOp = 2'b10;
415             end
416 
417             RegisterWrite: begin
418                 PCRegEN = 1'b0;
419                 FetchRegEN = 1'b0;
420                 InstructionRead = 1'b0;
421                 RFRegEN = 1'b0;
422                 ALUOutRegEN = 1'b0;
423                 DataRegEN = 1'b0;
424 
425                 PCRegRST = 1'b1;
426                 FetchRegRST = 1'b1;
427                 RFRegRST = 1'b1;
428                 ALUOutRegRST = 1'b1;
429 
430                 ALUIn1Src = 2'b00;
431                 ALUIn2Src = 2'b00;
432                 ResultSrc = 2'b00;
433 
434                 memWE = 1'b0;
435                 Regwe = 1'b1;
436 
437                 ALUOp = 2'b00;
438             end
439 
440         endcase
441     end
442 
443 endmodule
```

1.3.2.1 Branch logic

Code snippet 6.1.24: Branch logic

```
//--------------------------------------------------------------
// Title          : Branch Logic
//--------------------------------------------------------------
// File           : BranchLogic.sv
// Author         : David Ramon Alaman
// Created        : 27.09.2022
//--------------------------------------------------------------

module BranchLogic(
    input logic[2:0] funct3,
    input logic[3:0] flags,
    output logic branch
);

    always_comb begin
        case (funct3)
            3'b000: branch = flags[0]; // beq
            3'b001: branch = ~flags[0]; // bne
            3'b100: branch = flags[1] ^ flags[3]; // blt
            3'b101: branch = ~(flags[1] ^ flags[3]); // bge
            3'b110: branch = ~flags[2]; // bltu
            3'b111: branch = flags[2]; // bgeu
            default: branch = 0;
        endcase
    end
endmodule
```

1.3.3 Multicycle processor

Code snippet 6.1.25: RV32I - Multicycle processor

```
//---------------------------------------------------------------
// Title          : RV32I - Mulcicycle implementation - MPU
//---------------------------------------------------------------
// File           : RV32I_Multicycle.sv
// Author         : David Ramon Alaman
// Created        : 21.10.2022
//---------------------------------------------------------------

module RV32I_Multicycle(
    input logic clk, arst, en,
    input logic[31:0] data, instruction,
    output memWE, InstructionRead, DataRegEN,
    output[31:0] memAddr, memIn, pcOut
);
    // Signal definition
    logic RegWE, PCRegEN, FetchRegEN, RFRegEN, ALUOutRegEN, PCRegRST,
        FetchRegRST, RFRegRST, ALUOutRegRST, branch;
    logic[1:0] ALUIn1Src, ALUIn2Src, ResultSrc, ALUOp;
    logic[3:0] ALUCtrl, flags;
    logic[31:0] rs1, rs2, ALUIn1, ALUIn2, ALUOut,
        pc, pcOld, rs1Reg, rs2Reg, ALUOutReg, result, immediate;

    assign memIn = rs2Reg;
    assign pcOut = pc;
    //assign memAddr = result;
    assign memAddr = ALUOutReg;

    // Datapath modules instantiation
    RegisterFile RV_Registerfile(
        .clk(clk),
        .we(RegWE),
        .rst(arst),
        .reg1(instruction[19:15]),
        .reg2(instruction[24:20]),
        .reg3(instruction[11:7]),
        .dataIn(result),
        .regData1(rs1),
        .regData2(rs2)
    );

    ALU RV_ALU(
        .operand1(ALUIn1),
        .operand2(ALUIn2),
```

```
44         .control(ALUCtrl),
45         .result(ALUOut),
46         .flags(flags)
47     );
48 
49     ImmediateGenerator RV_ImmediateGenerator(
50         .instruction(instruction),
51         .immediate(immediate)
52     );
53 
54     // Control modules instantiation
55     Control RV_Control(
56         .clk(clk),
57         .arst(arst),
58         .en(en),
59         .branch(branch),
60         .opCode(instruction[6:0]),
61         .PCRegEN(PCRegEN),
62         .FetchRegEN(FetchRegEN),
63         .RFRegEN(RFRegEN),
64         .ALUOutRegEN(ALUOutRegEN),
65         .DataRegEN(DataRegEN),
66         .PCRegRST(PCRegRST),
67         .FetchRegRST(FetchRegRST),
68         .RFRegRST(RFRegRST),
69         .ALUOutRegRST(ALUOutRegRST),
70         .InstructionRead(InstructionRead),
71         .ALUIn1Src(ALUIn1Src),
72         .ALUIn2Src(ALUIn2Src),
73         .ResultSrc(ResultSrc),
74         .memWE(memWE),
75         .Regwe(RegWE),
76         .ALUOp(ALUOp)
77     );
78 
79     ALUControl RV_ALUControl(
80         .ALUOp(ALUOp),
81         .func({instruction[14:12], instruction[30], instruction[5]}),
82         .ALUCtrl(ALUCtrl)
83     );
84 
85     BranchLogic RV_BranchLogic(
86         .funct3(instruction[14:12]),
87         .flags(flags),
88         .branch(branch)
89     );
```

```
90
91      // Multiplexers
92
93      mux4 ALUIn1Mux(
94          .data0(rs1Reg),
95          .data1(32'b0),
96          .data2(pcOld),
97          .data3(pc),
98          .s(ALUIn1Src),
99          .dout(ALUIn1)
100     );
101
102     mux3 ALUIn2Mux(
103         .data0(immediate),
104         .data1(rs2Reg),
105         .data2(32'd4),
106         .s(ALUIn2Src),
107         .dout(ALUIn2)
108     );
109
110     mux3 ResultMux(
111         .data0(ALUOutReg),
112         .data1(ALUOut),
113         .data2(data),
114         .s(ResultSrc),
115         .dout(result)
116     );
117
118     // Registers
119
120     FlipFlop PCReg(
121         .clk(clk),
122         .en(PCRegEN),
123         .srst(PCRegRST),
124         .arst(arst),
125         .d(result),
126         .q(pc)
127     );
128
129     FlipFlop FetchReg(
130         .clk(clk),
131         .en(FetchRegEN),
132         .srst(FetchRegRST),
133         .arst(arst),
134         .d(pc),
135         .q(pcOld)
```

```
136     );
137
138     FlipFlop2 RFReg(
139         .clk(clk),
140         .en(RFRegEN),
141         .srst(RFRegRST),
142         .arst(arst),
143         .d0(rs1),
144         .d1(rs2),
145         .q0(rs1Reg),
146         .q1(rs2Reg)
147     );
148
149     FlipFlop ALUReg(
150         .clk(clk),
151         .en(ALUOutRegEN),
152         .srst(ALUOutRegRST),
153         .arst(arst),
154         .d(ALUOut),
155         .q(ALUOutReg)
156     );
157
158 endmodule
```

1.3.4 Memory bus

1.3.4.1 Memory controller

Code snippet 6.1.26: Memory controller

```
//-----------------------------------------------------------------
// Title          : Memory Controller
//-----------------------------------------------------------------
// File           : MemoryController.sv
// Author         : David Ramon Alaman
// Created        : 31.10.2022
//-----------------------------------------------------------------

module MemoryController #(
    parameter RAM_ADDR = 32'h1000_0000,
    parameter PID1_ADDR = 32'hC000_0000,
    parameter PID2_ADDR = 32'hC000_0040,
    parameter ADC_ADDR = 32'hC000_0078,
    parameter TIM_ADDR = 32'hC000_0080,
    parameter H7S_ADDR = 32'hC000_00A8,
    parameter PTL_ADDR = 32'hC000_00AC
)(
    input logic read, write,
    input logic[31:0] addr,
    output logic cs_RAM, cs_PID1, cs_PID2, cs_ADC, cs_TIM, cs_H7S, cs_PTL,
    output logic[2:0] read_mux
);

    always_comb begin
        if((addr >= RAM_ADDR) && (addr < PID1_ADDR) && (read || write))
        ↪ begin
            cs_RAM = 1'b1;
            cs_PID1 = 1'b0;
            cs_PID2 = 1'b0;
            cs_ADC = 1'b0;
            cs_TIM = 1'b0;
            cs_H7S = 1'b0;
            cs_PTL = 1'b0;
        end
        else if ((addr >= PID1_ADDR) && (addr < PID2_ADDR) && (read ||
        ↪ write)) begin
            cs_RAM = 1'b0;
            cs_PID1 = 1'b1;
            cs_PID2 = 1'b0;
            cs_ADC = 1'b0;
            cs_TIM = 1'b0;
```

```
40              cs_H7S = 1'b0;
41              cs_PTL = 1'b0;
42          end
43          else if ((addr >= PID2_ADDR) && (addr < ADC_ADDR) && (read || write
    ↪ )) begin
44              cs_RAM = 1'b0;
45              cs_PID1 = 1'b0;
46              cs_PID2 = 1'b1;
47              cs_ADC = 1'b0;
48              cs_TIM = 1'b0;
49              cs_H7S = 1'b0;
50              cs_PTL = 1'b0;
51          end
52          else if ((addr >= ADC_ADDR) && (addr < TIM_ADDR) && (read || write)
    ↪ ) begin
53              cs_RAM = 1'b0;
54              cs_PID1 = 1'b0;
55              cs_PID2 = 1'b0;
56              cs_ADC = 1'b1;
57              cs_TIM = 1'b0;
58              cs_H7S = 1'b0;
59              cs_PTL = 1'b0;
60          end
61          else if ((addr >= TIM_ADDR) && (addr < H7S_ADDR) && (read || write)
    ↪ ) begin
62              cs_RAM = 1'b0;
63              cs_PID1 = 1'b0;
64              cs_PID2 = 1'b0;
65              cs_ADC = 1'b0;
66              cs_TIM = 1'b1;
67              cs_H7S = 1'b0;
68              cs_PTL = 1'b0;
69          end
70          else if (addr == H7S_ADDR && (read || write)) begin
71              cs_RAM = 1'b0;
72              cs_PID1 = 1'b0;
73              cs_PID2 = 1'b0;
74              cs_ADC = 1'b0;
75              cs_TIM = 1'b0;
76              cs_H7S = 1'b1;
77              cs_PTL = 1'b0;
78          end
79          else if (addr == PTL_ADDR && (read || write)) begin
80              cs_RAM = 1'b0;
81              cs_PID1 = 1'b0;
82              cs_PID2 = 1'b0;
```

```
83              cs_ADC = 1'b0;
84              cs_TIM = 1'b0;
85              cs_H7S = 1'b0;
86              cs_PTL = 1'b1;
87          end
88          else begin
89              cs_RAM = 1'b0;
90              cs_PID1 = 1'b0;
91              cs_PID2 = 1'b0;
92              cs_ADC = 1'b0;
93              cs_TIM = 1'b0;
94              cs_H7S = 1'b0;
95              cs_PTL = 1'b0;
96          end
97      end
98
99      always_comb begin
100         if((addr >= RAM_ADDR) && (addr < PID1_ADDR)) begin
101             read_mux = 3'b000;
102         end
103         else if ((addr >= PID1_ADDR) && (addr < PID2_ADDR)) begin
104             read_mux = 3'b001;
105         end
106         else if ((addr >= PID2_ADDR) && (addr < ADC_ADDR)) begin
107             read_mux = 3'b010;
108         end
109         else if ((addr >= ADC_ADDR) && (addr < TIM_ADDR)) begin
110             read_mux = 3'b011;
111         end
112         else if ((addr >= TIM_ADDR) && (addr < H7S_ADDR)) begin
113             read_mux = 3'b100;
114         end
115         else if (addr == H7S_ADDR) begin
116             read_mux = 3'b101;
117         end
118         else if (addr == PTL_ADDR) begin
119             read_mux = 3'b110;
120         end
121         else begin
122             read_mux = 3'b000;
123         end
124     end
125 endmodule
```

1.3.4.2 PID-Timer link

Code snippet 6.1.27: PID-Timer link

```
//-----------------------------------------------------------------
// Title          : PID to Timer Link
//-----------------------------------------------------------------
// File           : PID_TIM_Link.sv
// Author         : David Ramon Alaman
// Created        : 25.05.2023
//-----------------------------------------------------------------

module PID_TIM_Link (
    input clk, chipSelect, write, read, rst, en,
    input logic[31:0] pid_in,
    input logic[31:0] writeData,
    output logic[31:0] timOut1, timOut2,
    output logic[31:0] readData
);

logic[31:0] register, compare;
assign compare = $signed(pid_in) >>> $signed(register);

always_ff @(posedge clk or negedge rst) begin
    if(!rst) begin
        register = 32'd0;
    end
    else begin
        if(chipSelect) begin
            if(write) begin
                register <= writeData;
            end
            else begin
                readData <= register;
            end
        end
    end
end

always_comb begin
    if($signed(pid_in) >= $signed(32'd0)) begin
        timOut1 = compare;
        timOut2 = 32'd0;
    end
    else begin
        timOut1 = 32'd0;
        timOut2 = -compare;
```

```
    end
end

endmodule
```

1.3.4.3 Seven-segment decoder

Code snippet 6.1.28: Seven-segment decoder

```
//---------------------------------------------------------------
// Title          : 7-segment 6-digit display decoder
//---------------------------------------------------------------
// File           : Hex7Segments.sv
// Author         : David Ramon Alaman
// Created        : 10.11.2022
//---------------------------------------------------------------

module Hex7Segments (
    input logic clk, chipSelect, write, read, rst,
    input logic[31:0] writeData,
    output logic[6:0] hex5, hex4, hex3, hex2, hex1, hex0,
    output logic[31:0] readData
);
    logic[31:0] register;

    always_ff @(posedge clk or negedge rst) begin
        if(!rst) begin
            register = 32'd0;
        end
        else begin
            if(chipSelect) begin
                if(write) begin
                    register <= writeData;
                end
                else if(read) begin
                    readData <= register;
                end
            end
        end
    end

    hex7seg #(.INVERT(1'b0)) disp0 (
        .inhex(register[3:0]),
        .out7seg(hex0)
    );

    hex7seg #(.INVERT(1'b0)) disp1 (
        .inhex(register[7:4]),
        .out7seg(hex1)
    );

    hex7seg #(.INVERT(1'b0)) disp2 (
```

```
44         .inhex(register[11:8]),
45         .out7seg(hex2)
46     );
47
48     hex7seg #(.INVERT(1'b0)) disp3 (
49         .inhex(register[15:12]),
50         .out7seg(hex3)
51     );
52
53     hex7seg #(.INVERT(1'b0)) disp4 (
54         .inhex(register[19:16]),
55         .out7seg(hex4)
56     );
57
58     hex7seg #(.INVERT(1'b0)) disp5 (
59         .inhex(register[23:20]),
60         .out7seg(hex5)
61     );
62
63 endmodule
64
65 module hex7seg #(
66     parameter INVERT = 0
67 )(
68     input [3:0] inhex,
69     output [6:0] out7seg
70 );
71
72     logic [6:0] out7seg_aux;
73
74     always_comb begin
75         case (inhex)
76             4'h0: out7seg_aux = 7'h3F;
77             4'h1: out7seg_aux = 7'h06;
78             4'h2: out7seg_aux = 7'h5B;
79             4'h3: out7seg_aux = 7'h4F;
80             4'h4: out7seg_aux = 7'h66;
81             4'h5: out7seg_aux = 7'h6D;
82             4'h6: out7seg_aux = 7'h7D;
83             4'h7: out7seg_aux = 7'h07;
84             4'h8: out7seg_aux = 7'h7F;
85             4'h9: out7seg_aux = 7'h67;
86             4'hA: out7seg_aux = 7'h77;
87             4'hB: out7seg_aux = 7'h7C;
88             4'hC: out7seg_aux = 7'h39;
89             4'hD: out7seg_aux = 7'h5E;
```

```
90             4'hE: out7seg_aux = 7'h79;
91             4'hF: out7seg_aux = 7'h71;
92             default: out7seg_aux = 7'h7F;
93         endcase
94     end
95 
96     assign out7seg = INVERT ? ~(out7seg_aux) : out7seg_aux;
97 
98 endmodule
```

1.3.4.4 Analog to Digital Converter

Code snippet 6.1.29: Analog to Digital Converter

```
//-----------------------------------------------------------------
// Title          : ADC Peripheral
//-----------------------------------------------------------------
// File           : ADC_reg.sv
// Author         : David Ramon Alaman
// Created        : 01.02.2023
//-----------------------------------------------------------------

module ADC_reg (
    input logic clk, chipSelect, read, rst, en,
    input logic addr,
    output logic[31:0] channel0, channel1, readData
);

    logic[11:0] ch0, ch1;
    logic[31:0] registers[1:0];

    assign channel0 = {20'b0, ch0};
    assign channel1 = {20'b0, ch1};

    always_ff @(posedge clk or negedge rst) begin
        if (!rst) begin
            registers <= '{default: '0};
        end
        else if (en) begin
            if(chipSelect) begin
                if(read) begin
                    readData <= registers[addr];
                end
            end

            registers[0] = {20'b0, ch0};
            registers[1] = {20'b0, ch1};
        end
    end

    ADC_block IP_block(
        .rst(rst),
        .clk(clk),
        .CH0(ch0),
        .CH1(ch1)
    );
endmodule
```

1.3.4.5 Timer

Code snippet 6.1.30: PWM Generator

```
//---------------------------------------------------------------
// Title          : PWM generator with dead time
//---------------------------------------------------------------
// File           : PWMGenerator.sv
// Author         : David Ramon Alaman
// Created        : 25.01.2023
//---------------------------------------------------------------

module PWMGenerator(
    input logic[7:0] value,
    input logic[31:0] compare, maxval, count,
    output logic pwm, pwmn
);
    logic[9:0] deadtime;

    always_comb begin
        if (value <= 8'd127) begin
            deadtime = value;
        end
        else if (value <= 8'd191) begin
            deadtime = (64 + value[5:0]) * 2;
        end
        else if (value <= 8'd223) begin
            deadtime = (32 + value[4:0]) * 8;
        end
        else begin
            deadtime = (32 + value[4:0]) * 16;
        end
        if(count < compare) begin
            pwm = 1'b1;
        end
        else begin
            pwm = 1'b0;
        end
        if(count >= (compare + deadtime) && count < (maxval - deadtime))
    ↪ begin
            pwmn = 1'b1;
        end
        else begin
            pwmn = 1'b0;
        end
    end
endmodule
```

Code snippet 6.1.31: Timer

```
//--------------------------------------------------------------
// Title          : Timer Peripheral
//--------------------------------------------------------------
// File           : Timer.sv
// Author         : David Ramon Alaman
// Created        : 24.01.2023
//--------------------------------------------------------------

module Timer #(
    parameter PRESCALER_DEFAULT_VALUE = 1
)  (
    input logic clk, chipSelect, write, read, rst, en,
    input logic[4:0] addr,
    input logic[31:0] writeData, bypass1, bypass2,
    output logic[1:0] pwm1out, pwm2out,
    output logic[31:0] readData
);

    logic tim_en, clk_en, pwm1, pwmn1, pwm2, pwmn2;
    logic[31:0] compare1, compare2;
    logic[31:0] registers[9:0];
    //registers[0] --> Count
    //registers[1] --> ARR
    //registers[2] --> Start
    //registers[3] --> IRQ
    //registers[4] --> Prescaler
    //registers[5] --> Dead time
    //registers[6] --> Compare 1
    //registers[7] --> Compare 2
    //registers[8] --> Output enable -> 0(disabled) 1(enable)
        // registers[8][0] -> PWM 1 enable
        // registers[8][1] -> PWMN 1 enable (PWM 1 must be enabled)
        // registers[8][2] -> PWM 2 enable
        // registers[8][3] -> PWMN 2 enable (PWM 2 must be enabled)
    //registers[9] --> Bypass

    assign tim_en = registers[2][0] & en & clk_en;
    assign pwm1out[0] = registers[8][0] ? pwm1 : 1'bz;
    assign pwm1out[1] = (registers[8][1] & registers[8][0]) ? pwmn1 : 1'bz;
    assign pwm2out[0] = registers[8][2] ? pwm2 : 1'bz;
    assign pwm2out[1] = (registers[8][3] & registers[8][2]) ? pwmn2 : 1'bz;
    assign compare1 = registers[9] ? bypass1 : registers[6];
    assign compare2 = registers[9] ? bypass2 : registers[7];

    always_ff @(posedge clk or negedge rst) begin
```

```
46          if(!rst) begin
47              registers <= '{default: '0};
48          end
49          else if (en) begin
50
51              if(chipSelect) begin
52                  if (~(write && (addr == 4'b0))) begin
53                      registers[0] <= tim_en ? (registers[0] + 1) : registers
    ↪ [0];
54                  end
55
56                  if (write) begin
57                      registers[addr] <= writeData;
58                  end
59                  else if (read) begin
60                      readData <= registers[addr];
61                  end
62              end
63              else begin
64                  registers[0] <= tim_en ? (registers[0] + 1) : registers[0];
65              end
66              if(registers[0] >= (registers[1] - 1) && tim_en == 1) begin
67                  registers[0] <= '0;
68                  registers[3] <= 32'h0000_0001;
69              end
70          end
71      end
72
73      Prescaler #(
74          .INITIAL_VALUE(PRESCALER_DEFAULT_VALUE)
75      ) prescaler (
76          .clk(clk),
77          .arst(rst),
78          .en(en),
79          .new_value(registers[4]),
80          .clk_en(clk_en)
81      );
82
83      PWMGenerator PWMGen1 (
84          .value(registers[5][7:0]),
85          .count(registers[0]),
86          .compare(compare1),
87          .maxval(registers[1]),
88          .pwm(pwm1),
89          .pwmn(pwmn1)
90      );
```

```
91
92     PWMGenerator PWMGen2 (
93         .value(registers[5][7:0]),
94         .count(registers[0]),
95         .compare(compare2),
96         .maxval(registers[1]),
97         .pwm(pwm2),
98         .pwmn(pwmn2)
99     );
100
101 endmodule
```

1.3.4.6 PID Controller

Code snippet 6.1.32: PID Controller

```
//-------------------------------------------------------------
// Title          : PID Controller
//-------------------------------------------------------------
// File           : PID.sv
// Author         : David Ramon Alaman
// Created        : 06.12.2022
//-------------------------------------------------------------

module PID (
    input clk, en, arst, srst,
    input logic[31:0] reference, feedback, k1, k2, k3,
    output logic[31:0] control
);

    logic[31:0] error, error1, error2, control1, feedback_d;
    logic[63:0] multiplied1, multiplied2, multiplied3;

    assign error = reference - feedback_d;
    assign control = multiplied1[31:0] + multiplied2[31:0] + multiplied3
    ↪ [31:0] + control1;

    FlipFlop delay1(
        .clk(clk),
        .en(en),
        .srst(srst),
        .arst(arst),
        .d(error),
        .q(error1)
    );

    FlipFlop delay2(
        .clk(clk),
        .en(en),
        .srst(srst),
        .arst(arst),
        .d(error1),
        .q(error2)
    );

    FlipFlop delayCtrl(
        .clk(clk),
        .en(en),
        .srst(srst),
```

```
43          .arst(arst),
44          .d(control),
45          .q(control1)
46      );
47
48      FlipFlop feedback_ff(
49          .clk(clk),
50          .en(en),
51          .srst(srst),
52          .arst(arst),
53          .d(feedback),
54          .q(feedback_d)
55      );
56
57      Multiplier_test mult_1(
58          .dataa(k1),
59          .datab(error),
60          .result(multiplied1)
61      );
62
63      Multiplier_test mult_2(
64          .dataa(k2),
65          .datab(error1),
66          .result(multiplied2)
67      );
68
69      Multiplier_test mult_3(
70          .dataa(k3),
71          .datab(error2),
72          .result(multiplied3)
73      );
74
75  endmodule
```

Code snippet 6.1.33: PID peripheral

```
//-----------------------------------------------------------------
// Title          : PID Peripheral
//-----------------------------------------------------------------
// File           : PID_Reg.sv
// Author         : David Ramon Alaman
// Created        : 24.12.2022
//-----------------------------------------------------------------

module PID_Reg #(
    parameter PRESCALER_DEFAULT_VALUE = 1
) (
    input logic clk, chipSelect, write, read, rst, en,
    input logic [3:0] addr,
    input logic [31:0] writeData, feedback_bypass,
    output logic [31:0] readData, control_bypass
);

    logic pid_en, clk_en;
    logic[31:0] feedback;
    logic[31:0] registers[11:0];
    logic[31:0] control_sat;
  logic[31:0] control_reg;
    //registers[0]  --> Reference        (R(k))
    //registers[1]  --> PID ctt. 1       (k1)
    //registers[2]  --> PID ctt. 2       (k2)
    //registers[3]  --> PID ctt. 3       (k3)
    //registers[4]  --> Feedback         (F(k))          (Bypass)
    //registers[5]  --> Clk prescaler
    //registers[6]  --> Status           (Active/Stop)
    //registers[7]  --> Clear PID
    //registers[8]  --> Bypass feedback
    //registers[9]  --> Saturation       (Enabled/Disabled)
    //registers[10] --> Upper saturation
    //registers[11] --> Lower saturation

    //address 12    --> Control action   (U(k))

    assign pid_en = en & registers [6][0] & clk_en;
    assign feedback = registers[8] ? feedback_bypass : registers[4];
    assign control_bypass = registers[9] ? control_reg : control_sat;

    always_comb begin
        if(control_reg > registers[10]) begin
            control_sat = registers[10];
        end
```

```
46          else if (control_reg < registers[11]) begin
47              control_sat = registers [11];
48          end
49          else begin
50              control_sat = control_reg;
51          end
52      end
53
54      always_ff @(posedge clk or negedge rst) begin
55          if(!rst) begin
56              registers <= '{default: '0};
57          end
58          else begin
59              if(chipSelect) begin
60                  if (write) begin
61                      if(addr < 12) begin
62                          registers [addr] <= writeData;
63                      end
64                  end
65                  else if(read) begin
66                      if(addr < 12) begin
67                          readData <= registers[addr];
68                      end
69                      else if(addr === 12) begin
70                          if(registers[9]) begin
71                              readData <= control_reg;
72                          end
73                          else begin
74                              readData <= control_sat;
75                          end
76                      end
77                  end
78              end
79          end
80      end
81
82      PID pid(
83          .clk(clk),
84          .en(pid_en),
85          .arst(rst),
86          .srst(~registers[7][0]),
87          .reference(registers[0]),
88          .feedback(feedback),
89          .k1(registers[1]),
90          .k2(registers[2]),
91          .k3(registers[3]),
```

```
92          .control(control_reg)
93      );
94
95      Prescaler #(
96          .INITIAL_VALUE(PRESCALER_DEFAULT_VALUE)
97      ) prescaler (
98          .clk(clk),
99          .arst(rst),
100         .en(en),
101         .new_value(registers[5]),
102         .clk_en(clk_en)
103     );
104
105 endmodule
```

1.3.5 Multicycle microcontroller

Code snippet 6.1.34: RV32I - Mulcicycle microcontroller

```
//---------------------------------------------------------------
// Title          : RV32I - Mulcicycle implementation - MCU
//---------------------------------------------------------------
// File           : RV32I_MC.sv
// Author         : David Ramon Alaman
// Created        : 25.12.2022
//---------------------------------------------------------------

module RV32I_MC #(
  parameter DEPTH = 64,
    parameter PROGRAM_FILE = "memotest.hex"
) (
    input logic clk, rst, en,
    output logic[1:0] pwm1out, pwm2out,
    output logic[6:0] hex0, hex1, hex2, hex3, hex4, hex5
);

    // Signals definition
    logic memWE, InstructionRead, DataRegEN,
        cs_RAM, cs_PID1, cs_PID2, cs_ADC, cs_TIM, cs_H7S, cs_PTL;
    logic[2:0] read_mux_src;
    logic[31:0] memOut, memAddr, memIn, instruction, pcOut,
        RAMOut, PID1Out, PID2Out, ADCOut, TIMOut, H7SOut, PTLOut,
   ↪ ADCChannel0,
        ADCChannel1, pid_in, timOut1, timOut2, nc;

    // Device under test
    RV32I_Multicycle RISCV(
        .clk(clk),
        .arst(rst),
        .en(en),
        .data(memOut),
        .instruction(instruction),
        .memWE(memWE),
        .InstructionRead(InstructionRead),
        .DataRegEN(DataRegEN),
        .memAddr(memAddr),
        .memIn(memIn),
        .pcOut(pcOut)
    );

    MemoryController RV_MemoryController (
        .read(DataRegEN),
```

```
43          .write(memWE),
44          .addr(memAddr),
45          .cs_RAM(cs_RAM),
46          .cs_PID1(cs_PID1),
47          .cs_PID2(cs_PID2),
48          .cs_ADC(cs_ADC),
49          .cs_TIM(cs_TIM),
50          .cs_H7S(cs_H7S),
51          .cs_PTL(cs_PTL),
52          .read_mux(read_mux_src)
53      );
54
55      mux7 RV_ReadMux (
56          .data0(RAMOut),
57          .data1(PID1Out),
58          .data2(PID2Out),
59          .data3(ADCOut),
60          .data4(TIMOut),
61          .data5(H7SOut),
62          .data6(PTLOut),
63          .s(read_mux_src),
64          .dout(memOut)
65      );
66
67      Memory #(
68          .DEPTH(DEPTH),
69          .WORD_SIZE(32),
70          .INITIALIZE(1),
71          .FILE(PROGRAM_FILE)
72      ) RV_ROM (
73          .clk(clk),
74          .we(1'b0),
75          .read(InstructionRead),
76          .cs(1'b1),
77          .addr(pcOut[$clog2(DEPTH)+1:2]),
78          .dataIn(0),
79          .dataOut(instruction)
80      );
81
82      Memory #(
83          .DEPTH(DEPTH),
84          .WORD_SIZE(32),
85          .INITIALIZE(0),
86          .FILE("")
87      ) RV_RAM (
88          .clk(clk),
```

```
89          .we(memWE),
90          .read(DataRegEN),
91          .cs(cs_RAM),
92          .addr(memAddr[$clog2(DEPTH)+1:2]),
93          .dataIn(memIn),
94          .dataOut(RAMOut)
95      );
96
97      PID_Reg #(
98          .PRESCALER_DEFAULT_VALUE(50000)
99      ) RV_pid1 (
100         .clk(clk),
101         .chipSelect(cs_PID1),
102         .write(memWE),
103         .read(DataRegEN),
104         .rst(rst),
105         .en(en),
106         .addr(memAddr[5:2]),
107         .writeData(memIn),
108         .feedback_bypass(ADCChannel0),
109         .readData(PID1Out),
110         .control_bypass(pid_in)
111     );
112
113     PID_Reg #(
114         .PRESCALER_DEFAULT_VALUE(50000)
115     ) RV_pid2 (
116         .clk(clk),
117         .chipSelect(cs_PID2),
118         .write(memWE),
119         .read(DataRegEN),
120         .rst(rst),
121         .en(en),
122         .addr(memAddr[5:2]),
123         .writeData(memIn),
124         .feedback_bypass(ADCChannel1),
125         .readData(PID2Out),
126         .control_bypass(nc)
127     );
128
129     ADC_reg RV_ADC(
130         .clk(clk),
131         .chipSelect(cs_ADC),
132         .read(DataRegEN),
133         .rst(rst),
134         .en(en),
```

```
135         .addr(memAddr[2]),
136         .channel0(ADCChannel0),
137         .channel1(ADCChannel1),
138         .readData(ADCOut)
139     );
140 
141     Timer #(
142         .PRESCALER_DEFAULT_VALUE(1)
143     ) RV_Timer (
144         .clk(clk),
145         .chipSelect(cs_TIM),
146         .write(memWE),
147         .read(DataRegEN),
148         .rst(rst),
149         .en(en),
150         .addr(memAddr[5:2]),
151         .writeData(memIn),
152         .bypass1(timOut1),
153         .bypass2(timOut2),
154         .pwm1out(pwm1out),
155         .pwm2out(pwm2out),
156         .readData(TIMOut)
157     );
158 
159     Hex7Segments RV_Hex7Segments(
160         .clk(clk),
161         .chipSelect(cs_H7S),
162         .write(memWE),
163         .read(DataRegEN),
164         .writeData(memIn),
165         .hex5(hex5),
166         .hex4(hex4),
167         .hex3(hex3),
168         .hex2(hex2),
169         .hex1(hex1),
170         .hex0(hex0),
171         .readData(H7SOut)
172     );
173 
174     PID_TIM_Link RV_PIDTIMLink(
175         .clk(clk),
176         .chipSelect(cs_PTL),
177         .write(memWE),
178         .read(DataRegEN),
179         .rst(rst),
180         .en(en),
```

```
181         .pid_in(pid_in),
182         .writeData(memIn),
183         .timOut1(timOut1),
184         .timOut2(timOut2),
185         .readData(PTLOut)
186     );
187 
188 endmodule
```

Code snippet 6.1.35: Current controller assembly

```
li x1, 0xC0000000   # PID 1 base address
li x2, 0xC0000080   # Timer base address
li x3, 0xC00000AC # PID-Timer link base address
li x7, 0xC00000A8   # 7-Segment base address
li x5, 1

# PID 1 Config
li x4, 1433
sw x4, 0x00(x1)    # Set reference to 5 A
li x4, 1024
sw x4, 0x04(x1)    # Set K1 to 1024
li x4, -1004
sw x4, 0x08(x1)    # Set K2 to -1004
li x4, 50001
sw x4, 0x14(x1)    # Set PID clk to 1kHz
sw x5, 0x20(x1)      # Activate bypass
li x4, 1179648
sw x4, 0x28(x1)    # Upper saturation
li x4, -1179648
sw x4, 0x2C(x1)      # Lower saturation

# Timer Config
li x4, 4096
sw x4, 0x04(x2)    # Set ARR
li x4, 10
sw x4, 0x14(x2)    # Set Dead time
li x4, 0xF
sw x4, 0x20(x2)    # Output enable
sw x5, 0x24(x2)    # Activate bypass

# PID-Timer link Config
li x4, 10
sw x4, 0x00(x3)    # Set link value

# Start peripherals
sw x5, 0x18(x1)    # Start PID 1
sw x5, 0x08(x2)    # Start Timer

.loop:
lw x6, 0x78(x1)
sw x6, 0x00(x7)    # Display ADC value
j .loop
```

2 Verification

2.1 Single-cycle

2.1.1 Datapath

2.1.1.1 Register file

Code snippet 6.2.36: Register file testbench

```
//-------------------------------------------------------------
// Title         : Testbench for the Register File
//-------------------------------------------------------------
// File          : tb_RegisterFile.sv
// Author        : David Ramon Alaman
// Created       : 17.09.2022
//-------------------------------------------------------------

module tb_RegisterFile();
    logic clk, we, rst, testEN;
    logic[4:0] reg1, reg2, reg3;
    logic[31:0] dataIn, regData1, regData2;
    logic[31:0] expected_regData1, expected_regData2;
    logic[7:0] testIndex, errors;
    logic[36:0] testvector1[25:0];
    logic[36:0] testvector2[25:0];
    logic[37:0] testvector3[25:0];

    // Device under test
    RegisterFile dut(clk, we, rst, reg1, reg2, reg3, dataIn, regData1,
    ↪ regData2);

    // Generate Clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Register File Testbench");
        $readmemh("vector_RF1.txt", testvector1);
        $readmemh("vector_RF2.txt", testvector2);
        $readmemh("vector_RF3.txt", testvector3);
        testIndex = 0; errors = 0;
        testEN = 1; rst = 0; #7; testEN = 0; rst = 1;
```

```
36          #60; rst = 0;
37      end
38
39      // Applay test vectors;
40      always @(posedge clk) begin
41          #1;
42          {reg1, expected_regData1} = testvector1[testIndex];
43          {reg2, expected_regData2} = testvector2[testIndex];
44          {we, reg3, dataIn} = testvector3[testIndex];
45      end
46
47      // Check results
48      always @(negedge clk) begin
49          if(~testEN) begin
50              if(regData1 !== expected_regData1) begin
51                  $display("Error in Register 1 (expecting %h).",
    ↪ expected_regData1);
52                  errors = errors + 1;
53              end
54
55              if(regData2 !== expected_regData2) begin
56                  $display("Error in Register 2 (expecting %h).",
    ↪ expected_regData2);
57                  errors = errors + 1;
58              end
59
60              testIndex = testIndex + 1;
61
62              if(testvector1[testIndex] === 37'bx) begin
63                  $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
64                  $stop;
65              end
66          end
67      end
68  endmodule
```

Code snippet 6.2.37: Register file verification vector 1

```
00_0000_0000
03_7654_3210
05_0123_4567
05_0123_4567
0b_3434_7979
00_0000_0000
01_0000_0000
```

Code snippet 6.2.38: Register file verification vector 2

```
00_0000_0000
00_0000_0000
03_7654_3210
05_0123_4567
03_7654_3210
03_7654_3210
02_0000_0000
```

Code snippet 6.2.39: Register file verification vector 3

```
23_7654_3210
25_0123_4567
05_8888_8888
2b_3434_7979
29_5555_5555
20_1234_1234
20_1234_1234
```

2.1.1.2 Immediate generator

Code snippet 6.2.40: Immediate generator testbench

```
//--------------------------------------------------------------
// Title          : Testbench for the Immediate Generator
//--------------------------------------------------------------
// File           : tb_ImmediateGenerator.sv
// Author         : David Ramon Alaman
// Created        : 17.09.2022
//--------------------------------------------------------------

module tb_ImmediateGenerator();

    logic clk;
    logic[31:0] instruction, immediate, expected_imm;
    logic[7:0] testIndex, errors;
    logic[63:0] testvector[25:0];

    // Device under test
    ImmediateGenerator dut(instruction, immediate);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Immediate Generator Testbench");
        $readmemh("vector_ImmediateGenerator.txt", testvector); // Load
    ↪ test vector
        testIndex = 0; errors = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        {instruction, expected_imm} = testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
        if(immediate !== expected_imm) begin
            $display("Error: instruction = %h (Op = %d)", instruction,
    ↪ instruction[6:0]);
```

```
41              $display(" inmmediate obtained = %h (expecting %h)", immediate,
    ↪   expected_imm);
42              errors = errors + 1;
43          end
44
45          testIndex = testIndex + 1;
46
47          if(testvector[testIndex] === 64'bx) begin
48              $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
49              $stop;
50          end
51      end
52
53 endmodule
```

Code snippet 6.2.41: Immediate generator verification vector

```
1  00402183_00000004
2  fd335083_ffffffd3
3  10110093_00000101
4  fb73fc13_ffffffb7
5  40205013_00000402
6  0b26e197_0b26e000
7  00929137_00929000
8  ff2024a3_ffffffe9
9  fe1178e3_fffffff0
10 fedffa6f_ffffffec
```

2.1.1.3 Arithmetic Logic Unit

Code snippet 6.2.42: ALU testbench

```
//--------------------------------------------------------------
// Title          : Testbench for the Arithmetic Logic Unit
//--------------------------------------------------------------
// File           : tb_ALU.sv
// Author         : David Ramon Alaman
// Created        : 18.09.2022
//--------------------------------------------------------------

module tb_ALU();
    logic[31:0] operand1, operand2, result, expected_result;
    logic[3:0] control;
    logic[3:0] flags, expected_flags;
    logic clk;
    logic[7:0] testIndex, errors;
    logic[103:0] testvector[25:0];

    // Device under test
    ALU dut(operand1, operand2, control, result, flags);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Arithmetic Logic Unit Testbench");
        $readmemh("vector_ALU.txt", testvector);
        testIndex = 0; errors = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        {control, expected_flags, expected_result, operand2, operand1} =
    ↪ testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
        if(expected_result !== result) begin
            $display("Error: result = %h (expecting %h).", result,
    ↪ expected_result);
```

```
42              errors = errors + 1;
43          end
44
45          if(expected_flags !== flags) begin
46              $display("Error: flags = %b (expecting %b).", flags,
    ↪ expected_flags);
47              errors = errors + 1;
48          end
49
50          testIndex = testIndex + 1;
51
52          if(testvector[testIndex] === 104'bx) begin
53              $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
54              $stop;
55          end
56      end
57  endmodule
```

Code snippet 6.2.43: ALU verification vector

```
1  0_0_77777777_76543210_01234567
2  1_4_11112222_65432121_76544343
3  0_c_35e1c915_ae32d792_87aef183
4  2_0_11100000_11101101_11110010
5  3_0_10101011_10000010_00101001
6  4_6_d28f471d_71feb936_a371fe2b
7  5_5_00000000_472f41ae_6f189ebc
8  5_0_00000001_93451937_93451931
9  6_2_be4ac000_3cad792e_a17ef92b
10 7_0_00ac7df3_31ae7648_ac7df38e
11 8_2_fffffa17_00000014_a17ef926
12 9_8_00000001_93451937_27897aed
13 9_d_00000000_27897aed_93451937
```

2.1.1.4 Data memory

Code snippet 6.2.44: Data memory testbench

```
//----------------------------------------------------------------
// Title          : Testbench for the Data Memory
//----------------------------------------------------------------
// File           : tb_RAM.sv
// Author         : David Ramon Alaman
// Created        : 22.09.2022
//----------------------------------------------------------------

module tb_RAM();
    logic clk, we, testEN;
    logic [31:0] dataIn, dataOut, expected_dataOut;
    logic [$clog2(1024)-1:0] addr;
    logic[7:0] testIndex, errors;
    logic[96:0] testvector[25:0];

    // Device under test
    RAM dut(clk, we, addr, dataIn, dataOut);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Random Access Memory Testbench");
        $readmemh("vector_RAM.txt", testvector);
        testIndex = 0; errors = 0;
        testEN = 1; #7; testEN = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        {we, addr, dataIn, expected_dataOut} = testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
        if(~testEN) begin
            if(dataOut !== expected_dataOut) begin
                $display("Error: Data = %h (expecting %h).", dataOut,
   ↪ expected_dataOut);
```

```
                errors = errors + 1;
            end

            testIndex = testIndex + 1;

            if(testvector[testIndex] === 97'bx) begin
                $display("%d tests completed with %d errors.", testIndex,
   ↪ errors);
                $stop;
            end
        end
    end
endmodule
```

Code snippet 6.2.45: Data memory verification vector

```
4_01_01234567_01234567
0_02_787878af_xxxxxxxx
4_02_fedcba98_xxxxxxxx
0_02_xxxxxxxx_fedcba98
0_01_xxxxxxxx_01234567
0_02_xxxxxxxx_fedcba98
```

2.1.1.5 Instruction memory

Code snippet 6.2.46: Instruction memory testbench

```
//-----------------------------------------------------------------
// Title          : Testbench for the Instruction Memory
//-----------------------------------------------------------------
// File           : tb_ROM.sv
// Author         : David Ramon Alaman
// Created        : 17.09.2022
//-----------------------------------------------------------------

module tb_ROM();
    logic clk;
    logic [31:0] pc, instruction, expected_inst;
    logic[7:0] testIndex, errors;
    logic[63:0] testvector[25:0];

    // Device under test
    ROM dut(pc, instruction);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Read Only Memory Testbench");
        $readmemh("vector_ROM.txt", testvector);
        testIndex = 0; errors = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        {pc, expected_inst} = testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
        if(instruction !== expected_inst) begin
            $display("Error: instruction = %h (expecting %h).", instruction
    ↪ , expected_inst);
            errors = errors + 1;
        end

```

```
43          testIndex = testIndex + 1;
44
45          if(testvector[testIndex] === 64'bx) begin
46              $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
47              $stop;
48          end
49      end
50  endmodule
```

Code snippet 6.2.47: Intruction memory verification instructions file

```
00402183
00000004
fd335083
ffffffd3
10110093
00000101
fb73fc13
ffffffb7
40205013
00000402
0b26e197
0b26e000
00929137
00929000
ff2024a3
ffffffe9
fe1178e3
fffffff0
fedffa6f
ffffffec
```

Code snippet 6.2.48: Intruction memory verification vector

```
00000000_00402183
00000004_00000004
00000034_00929000
0000003c_ffffffe9
00000008_fd335083
0000000c_ffffffd3
00000038_ff2024a3
00000040_fe1178e3
00000044_fffffff0
00000010_10110093
00000014_00000101
00000018_fb73fc13
0000004c_ffffffec
0000001c_ffffffb7
00000020_40205013
00000024_00000402
00000028_0b26e197
0000002c_0b26e000
00000030_00929137
00000048_fedffa6f
```

2.1.2 Control logic

2.1.2.1 Control unit

Code snippet 6.2.49: Control unit testbench

```
//----------------------------------------------------------------
// Title          : Testbench for the Control module
//----------------------------------------------------------------
// File           : tb_Control.sv
// Author         : David Ramon Alaman
// Created        : 30.09.2022
//----------------------------------------------------------------

module tb_Control();
    logic clk;
    logic[6:0] opCode;
    logic branch, forceJump, RAMwe, Regwe, ALUSrc2;
    logic expected_branch, expected_forceJump, expected_RAMwe,
    ↪ expected_Regwe, expected_ALUSrc2;
    logic[1:0] ALUOp, ALUSrc1, RegWriteSrc;
    logic[1:0] expected_ALUOp, expected_ALUSrc1, expected_RegWriteSrc;
    logic[7:0] testIndex;
    logic[15:0] errors;
    logic[17:0] testvector[20:0];

    // Device under test
    Control dut(opCode, branch, forceJump, RAMwe, Regwe, ALUSrc2, ALUOp,
    ↪ ALUSrc1, RegWriteSrc);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Control Testbench");
        $readmemb("vector_Control.txt", testvector); // Load test vector
        testIndex = 0; errors = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        #1;
```

```
39          {opCode, expected_branch, expected_forceJump, expected_RAMwe,
    ↪ expected_Regwe, expected_ALUSrc2, expected_ALUOp, expected_ALUSrc1,
    ↪ expected_RegWriteSrc} = testvector[testIndex];
40      end
41  
42      // Check results
43      always @(negedge clk) begin
44          if(expected_branch !== branch) begin
45              $display("Error: branch = %b (Expecting: %b)", branch,
    ↪ expected_branch);
46              errors = errors + 1;
47          end
48  
49          if(expected_forceJump !== forceJump) begin
50              $display("Error: forceJump = %b (Expecting: %b)", forceJump,
    ↪ expected_forceJump);
51              errors = errors + 1;
52          end
53  
54          if(expected_RAMwe !== RAMwe) begin
55              $display("Error: RAMwe = %b (Expecting: %b)", RAMwe,
    ↪ expected_RAMwe);
56              errors = errors + 1;
57          end
58  
59          if(expected_Regwe !== Regwe) begin
60              $display("Error: Regwe = %b (Expecting: %b)", Regwe,
    ↪ expected_Regwe);
61              errors = errors + 1;
62          end
63  
64          if(expected_ALUSrc2 !== ALUSrc2) begin
65              $display("Error: ALUSrc2 = %b (Expecting: %b)", ALUSrc2,
    ↪ expected_ALUSrc2);
66              errors = errors + 1;
67          end
68  
69          if(expected_ALUSrc1 !== ALUSrc1) begin
70              $display("Error: ALUSrc1 = %b (Expecting: %b)", ALUSrc1,
    ↪ expected_ALUSrc1);
71              errors = errors + 1;
72          end
73  
74          if(expected_ALUOp !== ALUOp) begin
75              $display("Error: ALUOp = %b (Expecting: %b)", ALUOp,
    ↪ expected_ALUOp);
```

```
76              errors = errors + 1;
77          end
78
79          if(expected_RegWriteSrc !== RegWriteSrc) begin
80              $display("Error: RegWriteSrc = %b (Expecting: %b)", RegWriteSrc
    ↪ , expected_RegWriteSrc);
81              errors = errors + 1;
82          end
83
84          testIndex = testIndex + 1;
85
86          if(testvector[testIndex] === 18'bx) begin
87              $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
88              $stop;
89          end
90      end
91  endmodule
```

Code snippet 6.2.50: Control unit verification vector

```
1  0010100_x_x_x_x_x_xx_xx_xx
2  0000011_0_0_0_1_0_10_00_00 // lw
3  0010011_0_0_0_1_0_00_00_01 // addi
4  0010111_0_0_0_1_0_10_10_01 // auipc
5  0100011_0_0_1_0_0_10_00_01 // sw
6  0110011_0_0_0_1_1_00_00_01 // add
7  0110111_0_0_0_1_0_10_01_01 // lui
8  1100011_1_0_0_0_1_01_00_01 // beq
9  1100111_0_1_0_1_0_10_00_10 // jalr
10 1101111_0_1_0_1_0_10_00_10 // jal
```

2.1.2.2 Branch logic

Code snippet 6.2.51: Branch logic testbench

```
//------------------------------------------------------------------
// Title          : Testbench for the Branch Logic
//------------------------------------------------------------------
// File           : tb_BranchLogic.sv
// Author         : David Ramon Alaman
// Created        : 27.09.2022
//------------------------------------------------------------------

module tb_BranchLogic();

    logic clk, branch, forceJump, opCode_3;
    logic[1:0] PCSrc, expected_PCSrc;
    logic[2:0] funct3;
    logic[3:0] flags;
    logic[7:0] testIndex, errors;
    logic[11:0] testvector[33:0];

    // Device under test
    BranchLogic dut(branch, forceJump, opCode_3, funct3, flags, PCSrc);

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("Branch Logic Testbench");
        $readmemb("vector_BranchLogic.txt", testvector); // Load test
    ↪ vector
        testIndex = 0; errors = 0;
    end

    // Apply test vector
    always @(posedge clk) begin
        #1;
        {forceJump, branch, funct3, flags, opCode_3, expected_PCSrc} =
    ↪ testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
```

```
42        if(PCSrc !== expected_PCSrc) begin
43            $display("Error: PCSrc = %b (Expected = %b)", PCSrc,
   ↪ expected_PCSrc);
44            errors = errors + 1;
45        end
46
47        testIndex = testIndex + 1;
48
49        if(testvector[testIndex] ===12'bx) begin
50            $display("%d tests completed with %d errors.", testIndex,
   ↪ errors);
51            $stop;
52        end
53    end
54
55 endmodule
```

Code snippet 6.2.52: Branch logic verification vector

```
1_0_101_1101_0_10
1_0_111_0101_0_10
1_0_001_0110_0_10
1_0_010_1000_0_10
1_0_000_0001_0_10
1_1_101_1001_0_10
1_1_111_0101_0_10
1_1_001_1100_0_10
1_1_010_0000_0_10
1_1_000_1111_0_10
1_0_101_1101_1_01
1_0_111_0101_1_01
1_0_001_0110_1_01
1_0_010_1000_1_01
1_0_000_0001_1_01
1_1_101_1001_1_01
1_1_111_0101_1_01
1_1_001_1100_1_01
1_1_010_0000_1_01
1_1_000_1111_1_01
0_1_000_0001_0_01
0_0_000_0001_0_00
0_1_000_0000_0_00
0_1_001_0000_0_01
0_1_001_0001_0_00
0_1_100_1000_0_01
0_1_100_1010_0_00
0_1_101_1000_0_00
0_1_101_1010_0_01
0_1_110_0100_0_00
0_1_110_1011_0_01
0_1_111_0100_0_01
0_1_111_0000_0_00
0_1_010_1010_0_00
```

2.1.2.3 ALU Control

Code snippet 6.2.53: ALU control testbench

```
//-----------------------------------------------------------------
// Title          : Testbench for the ALU Controller
//-----------------------------------------------------------------
// File           : tb_ALUControl.sv
// Author         : David Ramon Alaman
// Created        : 27.09.2022
//-----------------------------------------------------------------

module tb_ALUControl();
    logic clk;
    logic[1:0] ALUOp;
    logic[4:0] func;
    logic[3:0] ALUCtrl, expected_ALUCtrl;
    logic[7:0] testIndex, errors;
    logic[10:0] testvector[200:0];

    // Device under test
    ALUControl dut(ALUOp, func, ALUCtrl);

    // Generate Clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        $display("ALU Control Testbench");
        $readmemb("vector_ALUControl.txt", testvector);
        testIndex = 0; errors = 0;
    end

    // Applay test vectors;
    always @(posedge clk) begin
        #1;
        {ALUOp, func, expected_ALUCtrl} = testvector[testIndex];
    end

    // Check results
    always @(negedge clk) begin
        if(ALUCtrl !== expected_ALUCtrl) begin
            $display("Error: ALUCtrl = %h (expecting %h).", ALUCtrl,
   ↪ expected_ALUCtrl);
```

```
43              errors = errors + 1;
44          end
45
46          testIndex = testIndex + 1;
47
48          if(testvector[testIndex] === 11'bx) begin
49              $display("%d tests completed with %d errors.", testIndex,
    ↪ errors);
50              $stop;
51          end
52      end
53  endmodule
```

Code snippet 6.2.54: ALU control verification vector

```
00_00000_0000
00_00001_0000
00_00010_0000
00_00011_0001
00_00100_0110
00_00101_0110
00_00110_xxxx
00_00111_xxxx
00_01000_0101
00_01001_0101
00_01010_0101
00_01011_0101
00_01100_1001
00_01101_1001
00_01110_1001
00_01111_1001
00_10000_0100
00_10001_0100
00_10010_0100
00_10011_0100
00_10100_0111
00_10101_0111
00_10110_1000
00_10111_1000
00_11000_0011
00_11001_0011
00_11010_0011
00_11011_0011
00_11100_0010
00_11101_0010
00_11110_0010
00_11111_0010
01_00000_0001
01_00001_0001
01_00010_0001
01_00011_0001
01_00100_0001
01_00101_0001
01_00110_0001
01_00111_0001
01_01000_0001
01_01001_0001
01_01010_0001
01_01011_0001
01_01100_0001
```

```
46 01_01101_0001
47 01_01110_0001
48 01_01111_0001
49 01_10000_0001
50 01_10001_0001
51 01_10010_0001
52 01_10011_0001
53 01_10100_0001
54 01_10101_0001
55 01_10110_0001
56 01_10111_0001
57 01_11000_0001
58 01_11001_0001
59 01_11010_0001
60 01_11011_0001
61 01_11100_0001
62 01_11101_0001
63 01_11110_0001
64 01_11111_0001
65 10_00000_0000
66 10_00001_0000
67 10_00010_0000
68 10_00011_0000
69 10_00100_0000
70 10_00101_0000
71 10_00110_0000
72 10_00111_0000
73 10_01000_0000
74 10_01001_0000
75 10_01010_0000
76 10_01011_0000
77 10_01100_0000
78 10_01101_0000
79 10_01110_0000
80 10_01111_0000
81 10_10000_0000
82 10_10001_0000
83 10_10010_0000
84 10_10011_0000
85 10_10100_0000
86 10_10101_0000
87 10_10110_0000
88 10_10111_0000
89 10_11000_0000
90 10_11001_0000
91 10_11010_0000
```

```
10_11011_0000
10_11100_0000
10_11101_0000
10_11110_0000
10_11111_0000
11_00000_xxxx
11_00001_xxxx
11_00010_xxxx
11_00011_xxxx
11_00100_xxxx
11_00101_xxxx
11_00110_xxxx
11_00111_xxxx
11_01000_xxxx
11_01001_xxxx
11_01010_xxxx
11_01011_xxxx
11_01100_xxxx
11_01101_xxxx
11_01110_xxxx
11_01111_xxxx
11_10000_xxxx
11_10001_xxxx
11_10010_xxxx
11_10011_xxxx
11_10100_xxxx
11_10101_xxxx
11_10110_xxxx
11_10111_xxxx
11_11000_xxxx
11_11001_xxxx
11_11010_xxxx
11_11011_xxxx
11_11100_xxxx
11_11101_xxxx
11_11110_xxxx
11_11111_xxxx
```

2.1.3 Processor

Code snippet 6.2.55: RV32I single-cycle processor testbench

```
//-------------------------------------------------------------
// Title          : RISCV-32I testbench
//-------------------------------------------------------------
// File           : tb_RV32I_SC.sv
// Author         : David Ramon Alaman
// Created        : 04.10.2022
//-------------------------------------------------------------

module tb_RV32I_SC #(
    parameter ROM_SIZE = 64,
    parameter RAM_DEPTH = 64,
    parameter WORD_SIZE = 32
)();

    logic clk, en, rst;
    logic[WORD_SIZE - 1:0] ramOut;
    logic[31:0] instruction;
    logic RAMwe;
    logic[$clog2(RAM_DEPTH)-1:0] RAMaddr;
    logic[WORD_SIZE - 1:0] rs2;
    logic[31:0] pc;

    // Device under test
    RV32I_SC #(ROM_SIZE, RAM_DEPTH, WORD_SIZE) RiscV(
        .clk(clk),
        .en(en),
        .rst(rst),
        .ramOut(ramOut),
        .instruction(instruction),
        .RAMwe(RAMwe),
        .RAMaddr(RAMaddr),
        .rs2(rs2),
        .pc(pc)
    );

    RAM #(RAM_DEPTH, WORD_SIZE) RV_RAM(
        .clk(clk),
        .we(RAMwe),
        .addr(RAMaddr),
        .dataIn(rs2),
        .dataOut(ramOut)
    );

```

```
44     ROM #(ROM_SIZE) RV_ROM(
45         .pc(pc),
46         .instruction(instruction)
47     );
48 
49     // Generate clock
50     always begin
51         clk = 1; #5;
52         clk = 0; #5;
53     end
54 
55     // Initialization
56     initial begin
57         //$readmemh("vector_ALU.txt", testvector);
58         //testIndex = 0; errors = 0;
59         en = 0;
60         rst = 0; #7 rst = 1;
61         en = 1;
62     end
63 
64 endmodule
65 
66 module ROM #(
67     parameter DEPTH = 1024
68 )
69 (
70     input logic[31:0] pc,
71     output logic[31:0] instruction
72 );
73 
74     logic[31:0] memory[DEPTH-1:0];
75 
76     initial begin
77         $readmemh("bubblesort.hex", memory);
78     end
79 
80     assign instruction = memory[pc[$clog2(DEPTH)+1:2]];
81 
82 endmodule
83 
84 module RAM #(
85     parameter DEPTH = 1024,
86     parameter WORD_SIZE = 32
87 )(
88     input logic clk, we,
89     input logic[$clog2(DEPTH)-1:0] addr,
```

```
90     input logic[WORD_SIZE-1:0] dataIn,
91     output logic[WORD_SIZE-1:0] dataOut
92 );
93     logic[WORD_SIZE-1:0] ram[DEPTH-1:0];
94 
95     assign dataOut = ram[addr];
96 
97     always @(posedge clk) begin
98         if(we) begin
99             ram[addr] <= dataIn;
100        end
101    end
102
103 endmodule
```

Code snippet 6.2.56: Types I, S verification

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

Code snippet 6.2.57: 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
```

Code snippet 6.2.58: 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
```

```
46 nop
47 .jump11:
48 bgeu x2, x3, .jump11  # if x2 >= x3 unsigned then target
49 .back2:
50 bgeu x1, x1, .jump12  # if x1 >= x1 unsigned then target
51 jal x0, .back2
52 .jump12:
53 nop
```

Code snippet 6.2.59: Fibonacci sequence

```
1  addi x1, x0, 15 # i (number of cycles)
2  addi x2, x0, 0 # a variable
3  addi x3, x0, 1 # b variable
4  .loop:
5  bgeu x0, x1, .end # while i > 0
6  add x4, x2, x3 # temporary (a+b)
7  add x2, x3, x0 # a = b
8  add x3, x4, x0 # b = c
9  addi x1, x1, -1 # i - 1
10 jal x0, .loop
11 .end:
12 nop
```

Code snippet 6.2.60: Bubble sort

```
li x1, 256      # RAM addr to store words
li x2, 0        # Swapp indicator
li x3, 63       # First element
li x4, 2        # Second element
li x5, 30       # Third element
li x6, 92       # Forth element
li x7, 82       # Fifth element
li x8, 42       # Sixth element
li x9, 47       # Seventh element
li x10, 98      # Eighth element
li x11, 64      # Nineth element
li x12, 55      # Tenth element
li x14, 9       # Number of elements -1
sw x3, 0(x1)    # Store elements
sw x4, 4(x1)
sw x5, 8(x1)
sw x6, 12(x1)
sw x7, 16(x1)
sw x8, 20(x1)
sw x9, 24(x1)
sw x10, 28(x1)
sw x11, 32(x1)
sw x12, 36(x1)
.bucleconfig:
li x13, 0        # Index
add x2, x0, x0   # Reset swap indicator
.bucle:
lw x30, 0(x1)    # Load element n to RF
lw x31, 4(x1)    # Load n+1
blt x31, x30, .swapp  # Swap if n+1 < n
.swappreturn:
addi x1, x1, 4   # Increase mem addr
addi x13, x13, 1 # Index += 1
blt x13, x14, .bucle # Repeat if index < number of elements
li x1, 256       # Reset mem addr
bne x2, x0, .bucleconfig # If swap indicator -> repeat
jal, x0, .end    # end
.swapp:
sw x31, 0(x1)    # Swap n+1
sw x30, 4(x1)    # Swap n
addi x2, x0, 1   # Swap indicator = 1
jal x0, .swappreturn # return from swap
.end:
nop
```

2.2 Multicycle

2.2.1 Multicycle microcontroller

Code snippet 6.2.61: RV32I Multicycle microcontroller testbench

```
//-----------------------------------------------------------------
// Title          : RV32I - Mulcicycle MCU - Testbench
//-----------------------------------------------------------------
// File           : tb_RV32I_MC.sv
// Author         : David Ramon Alaman
// Created        : 27.12.2022
//-----------------------------------------------------------------

`timescale 1 ns / 1 ps
module tb_RV32I_MC();
    logic clk, rst, en;
    logic[1:0] pwm1out, pwm2out;
    logic[6:0] hex0, hex1, hex2, hex3, hex4, hex5;

    RV32I_MC RISCV(
        .clk(clk),
        .rst(rst),
        .en(en),
        .pwm1out(pwm1out),
        .pwm2out(pwm2out),
        .hex0(hex0),
        .hex1(hex1),
        .hex2(hex2),
        .hex3(hex3),
        .hex4(hex4),
        .hex5(hex5)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        en = 0;
        rst = 0; #7 rst = 1;
        en = 1;
    end
endmodule
```

2.2.2 Memory bus

Code snippet 6.2.62: Memory bus verification assembly

```
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
```

2.2.3 Peripherals

2.2.3.1 PID-Timer link

Code snippet 6.2.63: PID-Timer link testbench

```
//---------------------------------------------------------------
// Title          : PID to Timer Link
//---------------------------------------------------------------
// File           : PID_TIM_Link.sv
// Author         : David Ramon Alaman
// Created        : 25.05.2023
//---------------------------------------------------------------

module tb_TIM_PID_Link();

    // Signals definition
    logic clk, rst, en, cs, write, read;
    logic [31:0] writeData, readData, pid_in, compare1, compare2;

    // DUT
    PID_TIM_Link dut(
        .clk(clk),
        .chipSelect(cs),
        .write(write),
        .read(read),
        .rst(rst),
        .en(en),
        .pid_in(pid_in),
        .writeData(writeData),
        .timOut1(compare1),
        .timOut2(compare2),
        .readData(readData)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        en = 0;
        rst = 0;
        cs = 0;
        write = 0;
```

```
42            read = 0;
43            pid_in = 2048;
44            writeData = 0;
45            #7;
46            rst = 1;
47            en = 1;
48            #10;
49            pid_in = -2048;
50            #10;
51            pid_in = 2048;
52            write = 1;
53            cs = 1;
54            writeData = 1;
55            #10;
56            write = 0;
57            read = 1;
58            pid_in = -2048;
59            #10;
60            write = 1;
61            read = 0;
62            writeData = 5;
63            pid_in = 2048;
64            #10;
65            write = 0;
66            read = 1;
67            pid_in = -2048;
68            #10;
69            write = 1;
70            read = 0;
71            writeData = 10;
72            pid_in = 2048;
73            #10;
74            write = 0;
75            read = 1;
76            pid_in = -2048;
77        end
78
79    endmodule
```

2.2.3.2 Seven-segment decoder

Code snippet 6.2.64: Seven-segment decoder testbench

```
//------------------------------------------------------------
// Title          : Testbench Timer with registers
//------------------------------------------------------------
// File           : tb_Hex7Segments.sv
// Author         : David Ramon Alaman
// Created        : 12.11.2022
//------------------------------------------------------------

module tb_Hex7Segments ();

    // Signals definition
    logic clk, rst, cs, write, read;
    logic[6:0] hex5, hex4, hex3, hex2, hex1, hex0;
    logic[31:0] writeData, readData;

    Hex7Segments dut(
        .clk(clk),
        .chipSelect(cs),
        .write(write),
        .read(read),
        .rst(rst),
        .writeData(writeData),
        .hex5(hex5),
        .hex4(hex4),
        .hex3(hex3),
        .hex2(hex2),
        .hex1(hex1),
        .hex0(hex0),
        .readData(readData)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        rst = 0;
        cs = 0;
        write = 0;
        read = 0;
```

```
44          writeData = 0;
45          #7;
46          rst = 1;
47          cs = 1;
48          write = 1;
49          writeData = 32'h00654321;
50      end
51  endmodule
```

Code snippet 6.2.65: Seven-segment display

```
1 li x1, 0xC00000A8
2 li x2, 0x00654321
3 
4 sw x2, 0(x1)
5 .loop:
6 j .loop
```

2.2.3.3 Timer

Code snippet 6.2.66: Timer testbench

```
//----------------------------------------------------------------
// Title          : Testbench Timer with registers
//----------------------------------------------------------------
/// File          : tb_Timer.sv
// Author         : David Ramon Alaman
// Created        : 24.12.2022
//----------------------------------------------------------------

`timescale 1 ns / 1 ps

module tb_Timer ();

    // Signals definition
    logic clk, rst, en, cs, write, read;
    logic[4:0] addr;
    logic[1:0] pwm1out, pwm2out;
    logic[31:0] writeData, readData, compare1, compare2;

    Timer /*#(
        .PRESCALER_DEFAULT_VALUE(1)
    )*/ dut (
        .clk(clk),
        .chipSelect(cs),
        .write(write),
        .read(read),
        .rst(rst),
        .en(en),
        .addr(addr),
        .writeData(writeData),
        .bypass1(compare1),
        .bypass2(compare2),
        .pwm1out(pwm1out),
        .pwm2out(pwm2out),
        .readData(readData)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
```

```
44      initial begin
45          en = 0;
46          rst = 0;
47          cs = 0;
48          write = 0;
49          read = 0;
50          addr = 0;
51          compare1 = 2;
52          compare2 = 8;
53          writeData = 0;
54          #7;
55          en = 1;
56          rst = 1;
57          cs = 1;
58          write = 1;
59          writeData = 10;
60          addr = 1;
61          #10;
62          addr = 2;
63          writeData = 1;
64          #200;
65          writeData = 0;
66          #10;
67          writeData = 0;
68          addr = 0;
69          #10;
70          addr = 4;
71          writeData = 4;
72          #10;
73          addr = 2;
74          writeData = 1;
75          #520;
76          addr = 2;
77          writeData = 0;
78          #10;
79          addr = 4;
80          writeData = 0;
81          #10;
82          addr = 0;
83          writeData = 0;
84          #10;
85          addr = 8;
86          writeData = 32'b0111;
87          #10;
88          addr = 6;
89          writeData = 5;
```

```
90          #10;
91          addr = 7;
92          #10;
93          addr = 2;
94          writeData = 1;
95          #100;
96          writeData = 1;
97          addr = 5;
98          #10;
99          addr = 9;
100         writeData = 1;
101     end
102 endmodule
```

2.2.3.4 PID controller

Code snippet 6.2.67: PID controller testbench

```
//---------------------------------------------------------------
// Title          : Testbench PID Controller
//---------------------------------------------------------------
// File           : tb_PID.sv
// Author         : David Ramon Alaman
// Created        : 06.12.2022
//---------------------------------------------------------------

`timescale 1 ns / 1 ps

module tb_PID ();

    // Signals definition
    logic clk, rst, en;
    logic[31:0] reference, feedback, k1, k2, k3;
    logic[31:0] control;
    int reffile, fbfile, cfile, efile, e1file, e2file, m1file, m2file,
        m3file, ufile, kfile;

    PID dut(
        .clk(clk),
        .en(en),
        .arst(rst),
        .srst(1'b1),
        .reference(reference),
        .feedback(feedback),
        .k1(k1),
        .k2(k2),
        .k3(k3),
        .control(control)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        $fdisplay(cfile, "%0d", $signed(control));
        $fdisplay(efile, "%0d", $signed(dut.error));
        $fdisplay(e1file, "%0d", $signed(dut.error1));
        $fdisplay(e2file, "%0d", $signed(dut.error2));
        $fdisplay(m1file, "%0d", $signed(dut.multiplied1));
        $fdisplay(m2file, "%0d", $signed(dut.multiplied2));
        $fdisplay(m3file, "%0d", $signed(dut.multiplied3));
        $fdisplay(ufile, "%0d", $signed(dut.control1));
```

```
44          $fscanf(reffile, "%d", reference);
45          $fscanf(fbfile, "%d", feedback);
46          if($feof(reffile)) begin
47              $fclose(reffile);
48              $fclose(fbfile);
49              $fclose(cfile);
50              $fclose(efile);
51              $fclose(e1file);
52              $fclose(e2file);
53              $fclose(m1file);
54              $fclose(m2file);
55              $fclose(m3file);
56              $fclose(ufile);
57              $fclose(kfile);
58              $stop;
59          end
60          clk = 0; #5;
61      end
62  
63      // Initialization
64      initial begin
65          en = 0;
66          rst = 0;
67  
68          reffile = $fopen("./Matlab/reffile.txt", "r");
69          if(reffile) $display("reffile opened succesfully");
70          else begin
71              $display("reffile failed to open");
72              $stop;
73          end
74  
75          fbfile = $fopen("./Matlab/fbfile.txt", "r");
76          if(fbfile) $display("fbfile opened succesfully");
77          else begin
78              $display("fbfile failed to open");
79              $fclose(reffile);
80              $stop;
81          end
82  
83          cfile = $fopen("./Matlab/cfile.txt", "w");
84          if(cfile) $display("cfile opened succesfully");
85          else begin
86              $display("cfile failed to open");
87              $fclose(reffile);
88              $fclose(fbfile);
89              $stop;
```

```
90          end
91
92          efile = $fopen("./Matlab/efile.txt", "w");
93          if(efile) $display("efile opened succesfully");
94          else begin
95              $display("efile failed to open");
96              $fclose(reffile);
97              $fclose(fbfile);
98              $fclose(cfile);
99              $stop;
100         end
101
102         e1file = $fopen("./Matlab/e1file.txt", "w");
103         if(e1file) $display("e1file opened succesfully");
104         else begin
105             $display("e1file failed to open");
106             $fclose(reffile);
107             $fclose(fbfile);
108             $fclose(cfile);
109             $fclose(efile);
110             $stop;
111         end
112
113         e2file = $fopen("./Matlab/e2file.txt", "w");
114         if(e2file) $display("e2file opened succesfully");
115         else begin
116             $display("e2file failed to open");
117             $fclose(reffile);
118             $fclose(fbfile);
119             $fclose(cfile);
120             $fclose(efile);
121             $fclose(e1file);
122             $stop;
123         end
124
125         m1file = $fopen("./Matlab/m1file.txt", "w");
126         if(m1file) $display("m1file opened succesfully");
127         else begin
128             $display("m1file failed to open");
129             $fclose(reffile);
130             $fclose(fbfile);
131             $fclose(cfile);
132             $fclose(efile);
133             $fclose(e1file);
134             $fclose(e2file);
135             $stop;
```

```
136         end
137
138         m2file = $fopen("./Matlab/m2file.txt", "w");
139         if(m2file) $display("m2file opened succesfully");
140         else begin
141             $display("m2file failed to open");
142             $fclose(reffile);
143             $fclose(fbfile);
144             $fclose(cfile);
145             $fclose(efile);
146             $fclose(e1file);
147             $fclose(e2file);
148             $fclose(m1file);
149             $stop;
150         end
151
152         m3file = $fopen("./Matlab/m3file.txt", "w");
153         if(m3file) $display("m3file opened succesfully");
154         else begin
155             $display("m3file failed to open");
156             $fclose(reffile);
157             $fclose(fbfile);
158             $fclose(cfile);
159             $fclose(efile);
160             $fclose(e1file);
161             $fclose(e2file);
162             $fclose(m1file);
163             $fclose(m2file);
164             $stop;
165         end
166
167         ufile = $fopen("./Matlab/ufile.txt", "w");
168         if(ufile) $display("ufile opened succesfully");
169         else begin
170             $display("ufile failed to open");
171             $fclose(reffile);
172             $fclose(fbfile);
173             $fclose(cfile);
174             $fclose(efile);
175             $fclose(e1file);
176             $fclose(e2file);
177             $fclose(m1file);
178             $fclose(m2file);
179             $fclose(m3file);
180             $stop;
181         end
```

```
182
183         kfile = $fopen("./Matlab/kfile.txt", "r");
184         if(kfile) $display("kfile opened succesfully");
185         else begin
186             $display("kfile failed to open");
187             $fclose(reffile);
188             $fclose(fbfile);
189             $fclose(cfile);
190             $fclose(efile);
191             $fclose(e1file);
192             $fclose(e2file);
193             $fclose(m1file);
194             $fclose(m2file);
195             $fclose(m3file);
196             $fclose(ufile);
197             $stop;
198         end
199
200         $fscanf(kfile, "%d", k1);
201         $fscanf(kfile, "%d", k2);
202         $fscanf(kfile, "%d", k3);
203
204         #7 rst = 1;
205         en = 1;
206     end
207 endmodule
```

Code snippet 6.2.68: PID controller MATLAB simulation script

```
%% Simulation Continuous PID

kp = 1;
ki = 0;
kd = 0;
t = 0.001;
td = kd;
ti = 1/ki;
k1 = kp+td/t;
k2 = -kp + t/ti - 2*td/t;
k3 = td/t;
k = [k1; k2; k3];

out = sim("Continuous_PID.slx");

dlmwrite('reffile.txt', int32(out.reference), 'precision', 10);
dlmwrite('fbfile.txt', int32(out.feedback), 'precision', 10);
dlmwrite('kfile.txt', int32(round(k)), 'precision', 10)

hold off
figure(1)
plot(out.control,'LineWidth',2)
hold on
plot(out.reference,'LineWidth',2)
plot(out.feedback,'LineWidth',2)
%plot(out.reference - out.feedback,'LineWidth',2)
grid minor
get(gca,'fontname');
set(gca,'fontname','Palatino Linotype')
legend("Control", "Reference", "Feedback")
fig = gca;
exportgraphics(fig, '5_2_ContinuousControl.pdf');

%% Read simulation files

discU = dlmread('ufile.txt', ' ', 1, 0);

hold off
figure(1)
plot(out.control,'LineWidth',2)
hold on
plot(discU,'LineWidth',2)
grid minor
get(gca,'fontname');
set(gca,'fontname','Palatino Linotype')
```

```
46 legend("Golden model", "DUT")
47 fig = gca;
48 exportgraphics(fig, '5_2_ComparisonControl.pdf');
```

Code snippet 6.2.69: PID Peripheral testbench

```
//-------------------------------------------------------------
// Title          : Testbench PID with registers
//-------------------------------------------------------------
// File           : tb_PID_Reg.sv
// Author         : David Ramon Alaman
// Created        : 24.12.2022
//-------------------------------------------------------------

`timescale 1 ns / 1 ps

module tb_PID_Reg ();

    // Signals definition
    logic clk, rst, en, cs, write, read;
    logic[3:0] addr;
    logic[31:0] writeData, readData, control_bypass;
    int response = 0;

    PID_Reg #(
        .PRESCALER_DEFAULT_VALUE(1)
    ) dut (
        .clk(clk),
        .chipSelect(cs),
        .write(write),
        .read(read),
        .rst(rst),
        .en(en),
        .addr(addr),
        .writeData(writeData),
        .feedback_bypass(32'd19),
        .readData(readData),
        .control_bypass(control_bypass)
    );

    // Generate clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
    end

    // Initialization
    initial begin
        en = 0;
        rst = 0;
        cs = 0;
```

```
46          write = 0;
47          read = 0;
48          addr = 0;
49          writeData = 0;
50          #7;
51          rst = 1;
52          en = 1;
53          write = 1;
54          cs = 1;
55          addr = 0;
56          writeData = 20;
57          #10;
58          addr = 1;
59          writeData = 1;
60          #10;
61          addr = 2;
62          writeData = -1;
63          #10;
64          addr = 4;
65          writeData = response;
66          #10;
67          addr = 10;
68          writeData = 15;
69          #10;
70          addr = 11;
71          writeData = 5;
72          #10;
73          addr = 6;
74          writeData = 1;
75          while(response < 20) begin
76              #10;
77              read = 1;
78              write = 0;
79              addr = 12;
80              response = response  + 1;
81              #10;
82              read = 0;
83              write = 1;
84              addr = 4;
85              writeData = response;
86          end
87      end
88  endmodule
```

Intentionally blank page