DESIGN OF AN ENERGY-EFFICIENT RISC-V MICRO-ARCHITECTURE FOR LOW-COST APPLICATIONS

GERSON LEANDRO GUALDRON MANRIQUE

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
BUCARAMANGA
2024

DESIGN OF AN ENERGY-EFFICIENT RISC-V MICRO-ARCHITECTURE FOR LOW-COST APPLICATIONS

GERSON LEANDRO GUALDRON MANRIQUE

Degree work presented as a requirement to qualify for the title of Electronic Engineer

Advisor:
JAVIER FERNEY ARDILA OCHOA
INGENIERO ELECTRÓNICO. PhD.

Co-Advisor:
SERGIO ALBERTO ABREO CARRILLO
INGENIERO ELECTRÓNICO. PhD.

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
BUCARAMANGA
2024

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to several people who have been fundamental in my academic and personal journey. First of all, I thank God for blessing me and being with me at every moment of my life.

I especially thank my parents for their love, effort, and financial support during my academic formation.

I want to express gratitude to my grandparents and relatives who were also helpful and supportive during this journey.

I also want to acknowledge professors Javier and Sergio for their guidance in this project. I also thank Hanssel for his comments and suggestions that contributed to the improvement of the development of this project.

I cannot fail to mention my OnChip colleagues, in particular Nicolás, Daniel, Sergio, Jorge, and others, whose valuable conversations and contributions enriched this project and my personal growth.

I thank Laura, Viviana, Eduard, and Silvia for being with me and making this last part of my career more enjoyable. Their friendship and support were a great help.

I want to express my gratitude to my cousin Brayan, who has been an incredible support and has become more than a cousin, he has become a brother to me.

Finally, I want to thank me, for believing in myself, for doing all this hard work, for never quitting, I want to thank me for always give and keep trying to give more than I receive, and for trying to do more right than wrong, I want to thank myself for just being me at all times.

Dedicated to God and my parents.

TABLE OF CONTENTS

	Page.
INTRODUCTION	**12**
1 PROJECT OVERVIEW	**13**
1.1 OBJECTIVES	14
1.1.1 General Objective	14
1.1.2 Specific Objectives	14
1.2 STATE OF THE ART	14
2 MICRO-ARCHITECTURE DESIGN	**16**
2.1 ISA SELECTION	16
2.2 FETCH STAGE	18
2.2.1 Program Memory	18
2.2.2 Program Counter	18
2.3 DECODE STAGE	19
2.3.1 Register File	19
2.3.2 Sort & Extend Unit	21
2.4 EXECUTE STAGE	22
2.4.1 Arithmetic Logic Unit	22
2.4.2 Multiplexers	23
2.4.3 Branch Unit	24
2.4.4 Multiplication and Division Unit	25
2.5 MEMORY STAGE	25
2.5.1 Data Memory	26
2.5.2 Data Memory Controller	26

2.6 WRITE-BACK STAGE 27
2.7 CONTROL UNIT 27
2.7.1 Main Decoder 28
2.7.2 Sort & Extend Decoder 28
2.7.3 ALU Decoder 29
2.7.4 Branch Decoder 29
2.7.5 Memory Decoder 29
2.7.6 MDU Decoder 29
2.8 TOP 29
2.9 FUNCTIONAL SIMULATION 30

3 MICRO-ARCHITECTURE IMPLEMENTATION 36
3.1 FPGA 36
3.2 PERIPHERAL DESIGN 37
3.2.1 Interrupt Unit 37
3.2.2 Timer 37
3.2.3 LCD and Leds 38
3.3 UART AND BOOTLOADER DESIGN 38
3.4 CLOCK UNIT DESIGN 38
3.5 SYNTHESIS AND IMPLEMENTATION 39

4 RESULTS 42
4.1 CONCLUSIONS 48

BIBLIOGRAPHY 49

APPENDICES 51

LIST OF FIGURES

		Page.
Figure 1	RISC instruction stages data flow.	16
Figure 2	RTL design program counter.	19
Figure 3	Register file RTL.	20
Figure 4	ALU RTL.	23
Figure 5	ALU's multiplexers connection.	24
Figure 6	Branch unit RTL.	24
Figure 7	Multiplication & division unit RTL.	25
Figure 8	Data memory and its controller.	26
Figure 9	Write-back multiplexer connection.	27
Figure 10	RTL micro-architecture design.	30
Figure 11	Simulation U type instructions.	31
Figure 12	Simulation I type instructions.	31
Figure 13	Simulation R type instructions.	32
Figure 14	Simulation J and B type instructions.	33
Figure 15	Simulation S type instructions.	33
Figure 16	Simulation I type memory instructions.	34
Figure 17	Simulation M extension R type instructions.	35
Figure 18	KC705 Platform.	36
Figure 19	RTL design interrupts unit.	37
Figure 20	RTL design timer.	37
Figure 21	RTL design LCD and leds controllers.	38
Figure 22	Default implementation.	39
Figure 23	Clock region Implementation.	40

Figure 24 Final implementation. 41

Figure 25 RTL micro-architecture implementation. 43

Figure 26 Device implementation. 43

Figure 27 Programming Flow. 45

Figure 28 FPGA demo setup. 46

Figure 29 Fibonacci Demo. 47

LIST OF TABLES

		Page.
Table 1	Layout of the implemented instructions.	17
Table 2	RV32 registers.	21
Table 3	Sort & Extend Configuration.	22
Table 4	ALU operations.	23
Table 5	Branch unit functions.	24
Table 6	Multiplication & division unit functions.	25
Table 7	Encoding main decoder.	28
Table 8	Encoding Sort & Extend.	28
Table 9	Encoding ALU.	29
Table 10	Performance & utilization results.	44

RESUMEN

TÍTULO: DISEÑO DE UNA MICROARQUITECTURA RISC-V ENERGÉTICAMENTE EFICIENTE PARA APLICACIONES DE BAJO COSTO. *

AUTOR: GERSON LEANDRO GUALDRON MANRIQUE **

PALABRAS CLAVE: RISC-V, MICROARQUITECTURA, MICROCONTROLADOR, SoC, SEMICONDUCTORES, DIGITAL, BAJA POTENCIA, FPGA.

DESCRIPCIÓN:

La creciente demanda de dispositivos portátiles enfocados a la salud, el bienestar físico y el entretenimiento, combinada con el lento avance tecnológico de las baterías respecto a estos dispositivos, ha generado la necesidad de diseñar dispositivos con bajo consumo energético manteniendo un rendimiento óptimo. Este trabajo tiene como objetivo diseñar e implementar una microarquitectura RISC-V de bajo consumo de potencia, buscando la ejecución de cada instrucción en un único ciclo de reloj. Además, se propone un proceso de diseño muy detallado que puede replicarse e implementarse fácilmente. Aunque existen varias implementaciones de procesadores RISC-V de código abierto, ninguna está lo suficientemente documentada o detallada como para permitir que un principiante pueda replicarlas. Cada componente está diseñado en el lenguaje de descripción de hardware (HDL) de Verilog. La implementación del diseño se realiza en la plataforma Xilinx KC705, que cuenta con una FPGA en tecnología de proceso CMOS de 28 nm.

---

* Trabajo de Grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones.
Director: JAVIER FERNEY ARDILA OCHOA. PhD.

ABSTRACT

TITLE: DESIGN OF AN ENERGY-EFFICIENT RISC-V MICRO-ARCHITECTURE FOR LOW-COST APPLICATIONS *

AUTHOR: GERSON LEANDRO GUALDRON MANRIQUE **

KEYWORDS: RISC-V, MICRO-ARCHITECTURE, MICROCONTROLLER, SoC, SEMICONDUCTORS, DIGITAL, LOW-POWER, FPGA.

DESCRIPTION:
The rising demand for wearable devices focused on health, fitness, and entertainment, combined with the slow technological advance in batteries compared to these devices, has generated the need to design devices with low power consumption while maintaining optimal performance. This work aims to design and implement a low-power RISC-V micro-architecture, looking for the execution of each instruction in a single clock cycle. In addition, a highly detailed design process is proposed that can be easily replicated and implemented. Although several open-source RISC-V processor implementations exist, none are sufficiently documented or detailed to allow a beginner to replicate them. Each component is designed in the Verilog hardware description language (HDL). The design implementation is on the Xilinx KC705 platform, which features an FPGA in 28 nm CMOS process technology.

---

* BSc Thesis

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones.
Director: JAVIER FERNEY ARDILA OCHOA. PhD.

INTRODUCTION

In today's technology landscape, where mobility and connectivity are crucial, the demand for low-power electronic devices, such as wearables and IoT devices, has grown exponentially. The wearable technology market was valued at over USD 55 billion in 2022 and is expected to exceed USD 125 billion by 2028, with a compound annual growth rate (CAGR) of over 13% [1] [2]. These devices, which range from smartwatches to environmental sensors and health monitoring systems, have become ubiquitous daily. However, despite technological progress, we face a persistent and pressing challenge: energy storage remains a weak link in the technological ecosystem. Batteries [3], which power much of our modern technology, have experienced marginal advances compared to other fields, limiting the autonomy and mobility of our devices. This contrast between rapid technological advancement and relatively slow evolution in energy storage capabilities poses a critical dilemma as we seek to further drive innovation and sustainability in electronics and mobility. In this context, designing and implementing low-power micro-architectures are essential to achieve devices with longer battery life without compromising performance and user experience.

---

[1] IMARCGROUP. *Wearable Technology Market: Global Industry Trends, Share, Size, Growth, Opportunity and Forecast 2023-2028*. Tech. rep.

[2] GRANDVIEWRESEARCH. *Wearable Technology Market Size, Share & Trends Analysis Report By Product (Head & Eyewear, Wristwear), By Application (Consumer Electronics, Healthcare), By Region (Asia Pacific, Europe), And Segment Forecasts, 2023 - 2030*. Tech. rep.

[3] Lie WANG, Ye ZHANG, and Peter G BRUCE. "Batteries for wearables". In: *National Science Review* 10.1 (Mar. 2022), nwac062. DOI: `10.1093/nsr/nwac062`. eprint: `https://academic.oup.com/nsr/article-pdf/10/1/nwac062/48727960/nwac062.pdf`

1. PROJECT OVERVIEW

OnChip research group has worked over the past six years on designing and implementing a family of microcontrollers based on RISC-V, being the first in the world to develop one on silicon, achieving the manufacture of 3 generations in 180 nm [4] and 130 nm [5] CMOS process. The justification for this project lies in its role as an opportunity to establish a minimal functional foundation for exploring the possibilities presented by this architecture in 28 nm technology.

In the research process, existing RISC-V cores lack detailed documentation for their digital design flow. This gap provides an opportunity to develop comprehensive guides for those entering the field of RISC-V-based computer architecture. This project aims to fill this documentation gap by providing a clear and detailed reference to simplify the understanding and application of the fundamental principles and processes in the digital design of RISC-V cores.

In summary, the main objective of this project is to create a base micro-architecture for the core of the next OnChip 28 nm microcontroller. In addition, it seeks to generate solid documentation that can be useful in academia. Developing this project provides the opportunity to gain experience in computer architecture for professional growth.

---

[4] Ckristian DURAN et al. "An Energy-Efficient RISC-V RV32IMAC Microcontroller for Periodical-Driven Sensing Applications". In: (2020), pp. 1–4. DOI: 10.1109/CICC48029.2020.9075877

[5] Ckristian DURAN et al. "A 32-bit RISC-V AXI4-lite bus-based microcontroller with 10-bit SAR ADC". in: (2016), pp. 315–318. DOI: 10.1109/LASCAS.2016.7451073

1.1. OBJECTIVES

1.1.1. General Objective To design a 32-bit RISC-V-based micro-architecture for a microcontroller focused on low-power consumption and implemented in a 28 nm technology FPGA.

1.1.2. Specific Objectives

- To design the micro-architecture in compliance with the ISA RISC-V RV32IM specification.
- To implement the micro-architecture on a 28 nm FPGA and assess its power consumption and performance characteristics.
- To document the design and implementation process.

1.2. STATE OF THE ART

RISC-V has emerged as an open and adaptable instruction set architecture with great potential in diverse applications. In recent years, it has experienced exponential growth, establishing itself as an attractive option in the technology industry.
Leading companies such as Google, Qualcomm, Samsung, and Nvidia have adopted RISC-V, driving its development and use in several fields. A dynamic ecosystem sup-

ports the growth, involving software tools [6], compilers[7], frameworks [8] [9] [10], and developer communities that actively foster innovation around this architecture.

RISC-V has expanded its applications from IoT devices and wearables to high-performance servers and data centers. Several RISC-V cores available on the market respond to specific approaches.

Choosing a RISC-V core involves considering the trade-off between performance and power usage. High-performance cores aim for maximum speed with higher clock frequencies, more pipelines, and execution units, but they come with higher power consumption. On the other hand, low-power cores prioritize energy efficiency, featuring lower clock frequencies and reduced power consumption, making them suitable for devices that need to conserve power.

The conclusion highlights the rapid growth of RISC-V and its adaptability to various applications, confirming that it can compete effectively with existing architectures across different fields, from high performance to low power consumption.

---

[6] Gianfranco MARIOTTI and Roberto GIORGI. “WebRISC-V: A 32/64-bit RISC-V pipeline simulation tool”. In: *SoftwareX* 18 (2022), p. 101105. DOI: https://doi.org/10.1016/j.softx.2022.101105

[7] Mehrdad POORHOSSEINI, Wolfgang NEBEL, and Kim GRÜTTNER. “A Compiler Comparison in the RISC-V Ecosystem”. In: (2020), pp. 1–6. DOI: 10.1109/COINS49042.2020.9191411

[8] Andrea COLUCCIO et al. “RISC-Vlim, a RISC-V Framework for Logic-in-Memory Architectures”. In: *Electronics* 11.19 (2022). DOI: 10.3390/electronics11192990

[9] Eric MATTHEWS and Lesley SHANNON. “TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features”. In: (2017), pp. 1–4. DOI: 10.23919/FPL.2017.8056766

[10] Chen BAI et al. “BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework”. In: (2021), pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643455

2. MICRO-ARCHITECTURE DESIGN

The design process begins by defining the Instruction Set Architecture (ISA). Afterward, the hardware needed for each stage is designed, considering the typical data stages utilized in RISC architectures, as illustrated in Figure 1. You can locate the Verilog files for these components in the appendices.

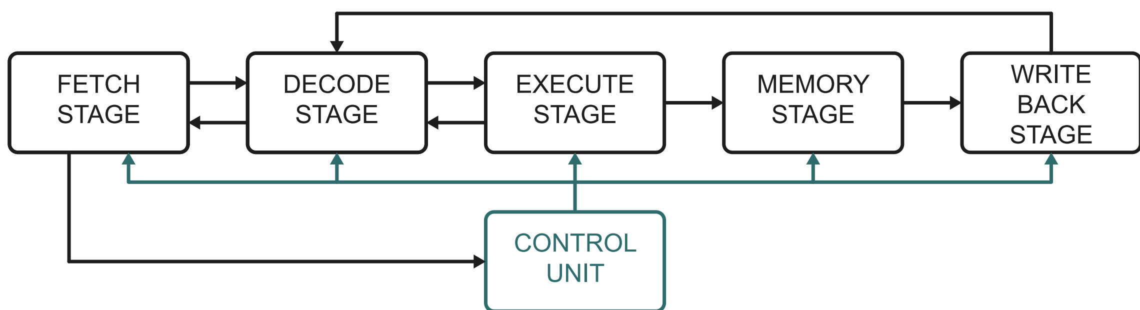

Figure 1. RISC instruction stages data flow.

2.1. ISA SELECTION

The proposed micro-architecture incorporates 37 instructions from the complete RV32I base instruction set, intentionally excluding instructions tailored for multi-core configurations and operating systems, such as FENCE, SYSTEM, and ENVIRONMENT. Alongside RV32I, this micro-architecture includes the RV32M extension, introducing multiplication and division operations to enhance its functionality and versatility. Table 1 details the implemented RV32IM instructions.

Table 1. Layout of the implemented instructions with opcodes, format type, and names. Taken from [11].

<table>
<tr><th>31 25</th><th>24 20</th><th>19 15</th><th>14 12</th><th>11 7</th><th>6 0</th><th>Bit</th></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd</td><td>0110111</td><td>U lui</td></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd</td><td>0010111</td><td>U auipc</td></tr>
<tr><td colspan="4">imm[20|10:1|11|19:12]</td><td>rd</td><td>1101111</td><td>J jal</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>1100111</td><td>I jalr</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>000</td><td>imm[4:1|11]</td><td>1100011</td><td>B beq</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>001</td><td>imm[4:1|11]</td><td>1100011</td><td>B bne</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>100</td><td>imm[4:1|11]</td><td>1100011</td><td>B blt</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>101</td><td>imm[4:1|11]</td><td>1100011</td><td>B bge</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>110</td><td>imm[4:1|11]</td><td>1100011</td><td>B bltu</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>111</td><td>imm[4:1|11]</td><td>1100011</td><td>B bgeu</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>0000011</td><td>I lb</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>001</td><td>rd</td><td>0000011</td><td>I lh</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>0000011</td><td>I lw</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>100</td><td>rd</td><td>0000011</td><td>I lbu</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>101</td><td>rd</td><td>0000011</td><td>I lhu</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>000</td><td>imm[4:0]</td><td>0100011</td><td>S sb</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>001</td><td>imm[4:0]</td><td>0100011</td><td>S sh</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>010</td><td>imm[4:0]</td><td>0100011</td><td>S sw</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>0010011</td><td>I addi</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>0010011</td><td>I slti</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>011</td><td>rd</td><td>0010011</td><td>I sltiu</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>100</td><td>rd</td><td>0010011</td><td>I xori</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>110</td><td>rd</td><td>0010011</td><td>I ori</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>111</td><td>rd</td><td>0010011</td><td>I andi</td></tr>
<tr><td>0000000</td><td>shamt</td><td>rs1</td><td>001</td><td>rd</td><td>0010011</td><td>I slli</td></tr>
<tr><td>0000000</td><td>shamt</td><td>rs1</td><td>101</td><td>rd</td><td>0010011</td><td>I srli</td></tr>
<tr><td>0100000</td><td>shamt</td><td>rs1</td><td>101</td><td>rd</td><td>0010011</td><td>I srai</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>000</td><td>rd</td><td>0110011</td><td>R add</td></tr>
<tr><td>0100000</td><td>rs2</td><td>rs1</td><td>000</td><td>rd</td><td>0110011</td><td>R sub</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>001</td><td>rd</td><td>0110011</td><td>R sll</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>010</td><td>rd</td><td>0110011</td><td>R slt</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>011</td><td>rd</td><td>0110011</td><td>R sltu</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>100</td><td>rd</td><td>0110011</td><td>R xor</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>101</td><td>rd</td><td>0110011</td><td>R srl</td></tr>
<tr><td>0100000</td><td>rs2</td><td>rs1</td><td>101</td><td>rd</td><td>0110011</td><td>R sra</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>110</td><td>rd</td><td>0110011</td><td>R or</td></tr>
<tr><td>0000000</td><td>rs2</td><td>rs1</td><td>111</td><td>rd</td><td>0110011</td><td>R and</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>000</td><td>rd</td><td>0110011</td><td>R mul</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>001</td><td>rd</td><td>0110011</td><td>R mulh</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>010</td><td>rd</td><td>0110011</td><td>R mulhsu</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>011</td><td>rd</td><td>0110011</td><td>R mulhu</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>100</td><td>rd</td><td>0110011</td><td>R div</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>101</td><td>rd</td><td>0110011</td><td>R divu</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>110</td><td>rd</td><td>0110011</td><td>R rem</td></tr>
<tr><td>0000001</td><td>rs2</td><td>rs1</td><td>111</td><td>rd</td><td>0110011</td><td>R remu</td></tr>
</table>

With the list of instructions now defined, the subsequent step involves designing the requisite hardware for each stage. This design is crucial for achieving efficient execution and ensuring a smooth data flow for each instruction.

2.2. FETCH STAGE

During the fetch stage, the instruction pointer (PC) accesses the address of the instruction in program memory. It retrieves the instruction from memory and prepares it for subsequent execution.

2.2.1. Program Memory For program memory, we opt for a 32-bit wide RAM with 256 addresses, equivalent to 1 kilobyte. This configuration is well-suited for storing small programs and test banks used to evaluate performance. The final decision on memory is determined during the generation of the complete processor and the SoC, considering their applications and specific requirements. In this case, we disregard the first two address bits to convert 4-address jumps into one-address jumps, ensuring compatibility with our memory configuration.

2.2.2. Program Counter The program counter utilizes a 32-bit register to store the memory address. It calculates the address of the next instruction by summing the outputs of two multiplexers, which the control unit and branch unit signals control. By default, the program counter increments the current address by sets of 4 unless a specific instruction interrupts.

---

[11] David PATTERSON and Andrew WATERMAN. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017

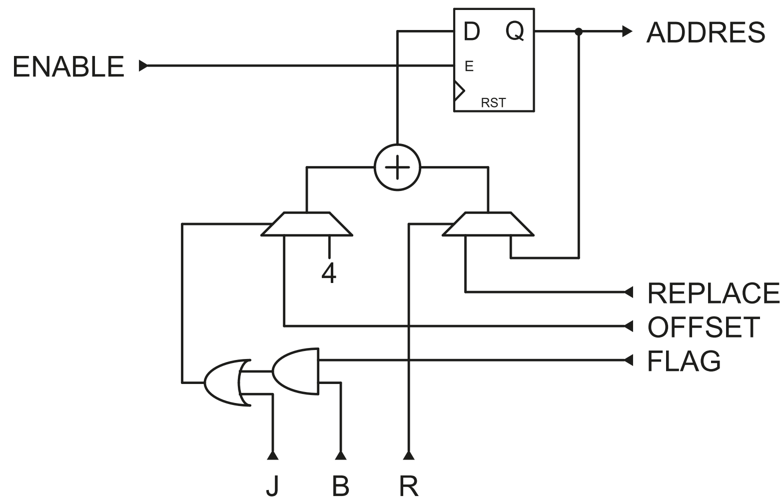

Figure 2. RTL design program counter.

Figure 2 illustrates the RTL design of the program counter implementation. It shows the control signals and inputs of the program counter. Additionally, it presents the output of the decoded instruction's address in the next stage.

2.3. DECODE STAGE

In the decode stage, the micro-architecture dissects the instruction into its components. This process includes identifying the type of instruction, the operands, and the destination of the result. The instruction determines the required operations. The operands consist of the addresses of the source registers or the immediate value. The destination of the result can be either the destination register address or a memory address.

2.3.1. Register File Following the instructions from Chapter 2 of the instruction manual [12], the RV32 micro-architecture incorporates 32 general-purpose registers denoted as x0-x31, along with the PC register. Input and output ports, controlled by 5-bit

---

[12] Andrew WATERMAN and Krste ASANOVIĆ, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation. May 2017

addresses, are connected to three multiplexers (MUX). The two MUX of the read ports have their inputs linked to the outputs of each register. The MUX of the write port establishes a 32-bit bus connected to the ENABLEs of the registers. This bus carries the WRITE ENABLE signal solely to the desired register, setting others to 0. Each register connects to the input data bus for input, except for x0, which actively has a hardwired value of zero. Figure 3 illustrates the RTL design of the register file, and Table 2 provides details on the RV32 registers and their names as determined by the RISC-V ABI (Application Binary Interface).

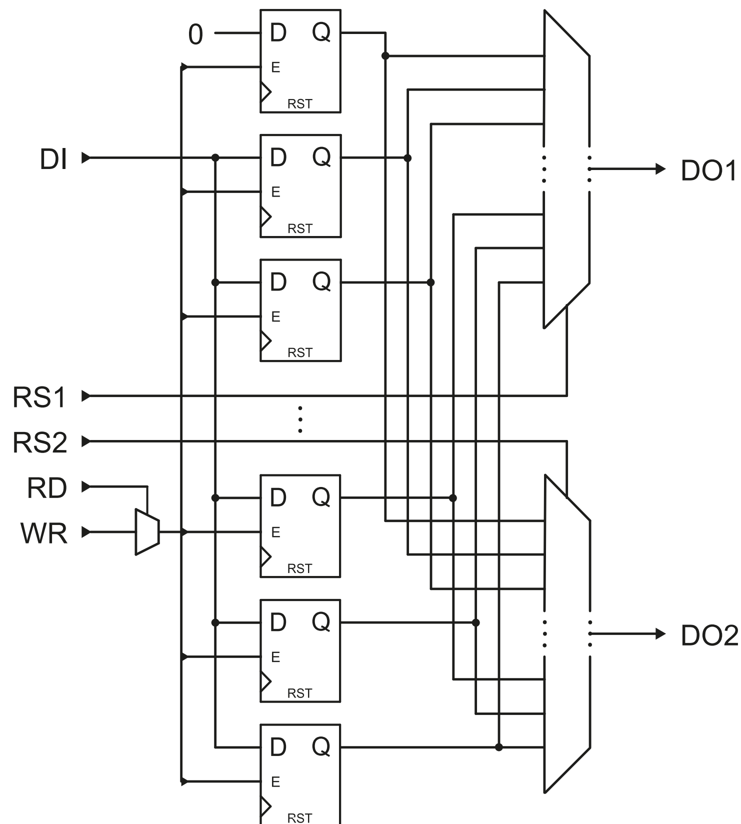

Figure 3. Register file RTL.

Table 2. RV32 registers. Taken from [13].

31 … 0 (32 bits)	Bit
x0 / zero	Hard-wired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary / alternate link register
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register / frame pointer
x9 / s1	Saved register
x10 / a0	Function arguments / return values
x11 / a1	Function arguments / return values
x12 / a2	Function arguments
x13 / a3	Function arguments
x14 / a4	Function arguments
x15 / a5	Function arguments
x16 / a6	Function arguments
x17 / a7	Function arguments
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

31 … 0 (32 bits)	
pc	Program counter

2.3.2. Sort & Extend Unit The Sort & Extend Unit (SortExt) generates immediate value for use in the next stage by extracting bits from the instruction, organizing them,

[13] PATTERSON and WATERMAN, *The RISC-V Reader: an open architecture Atlas*

and extending them to 32 bits to ensure compatibility with that stage. Through its signal, the control unit determines the immediate value generated. Table 3 illustrates the generation of the 32-bit immediate value based on the type of instruction.

Table 3. Sort & Extend Configuration.

Immediate extended	Type
{ Instr[31:12] , 12'd0 }	U Signed
{ 20{Instr[31]} , Instr[31:20] }	I Signed
{ 20{Instr[31]} , Instr[7] , Instr[30:25] , Instr[11:8] , 1'b0 }	B Signed
{ 20{Instr[31]} , Instr[31:25] , Instr[11:7] }	S Signed
{ 12{Instr[31]} , Instr[19:12] , Instr[20] , Instr[30:21] , 1'b0 }	J Signed
{ 20'd0 , Instr[31:20] }	I Unsigned
{ 20'd0 , Instr[7] , Instr[30:25] , Instr[11:8] , 1'b0 }	B Unsigned

The outcome of this stage comprises the values for the units of the execution stage, the memory address to be accessed, or the offset for jump instructions.

2.4. EXECUTE STAGE

During the execute stage, the system performs arithmetic or logical operations, or it controls the flow of the program. The execution of arithmetic and logical operations involves the utilization of the Arithmetic and Logic Unit (ALU) or the Multiplication and Division Unit (MDU). On the other hand, the Branch Unit governs the flow of the program, directing the sequence of instructions.

2.4.1. Arithmetic Logic Unit The ALU executes a diverse range of operations, encompassing arithmetic operations (addition and subtraction), logical operations (AND, OR, XOR), and comparison operations (set less than, both signed and unsigned). Additionally, it performs right-shift (both arithmetic and logical) and left-shift (logical) operations, as designated in the base instruction set RV32I. Figure 4 depicts the RTL layout of the arithmetic logic unit, and Table 4 details how the operations are assigned based on the control signal from the control unit.

Table 4. ALU operations.

Operation	Control signal
addition (add)	0000
subtraction (sub)	1000
and	0111
or	0110
xor	0100
set less than (slt)	0010
set less than unsigned (sltu)	0011
shift left logical (sll)	0001
shift right logical (srl)	0101
shift right arithmetic (sra)	1101

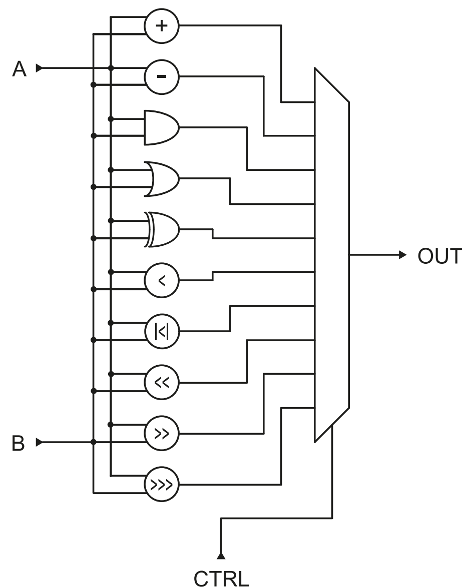

Figure 4. ALU RTL.

2.4.2. Multiplexers Two 4-to-1 multiplexers manage the ALU operands. The inputs to the multiplexers include the outputs of the register file, the SortExt unit, the program counter, and a fixed signal set at 4. The interconnections of these multiplexers are illustrated in Figure 5, with provisions made for potential custom instruction implementations.

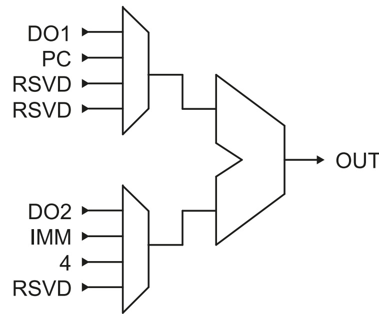

Figure 5. ALU's multiplexers connection.

2.4.3. Branch Unit The branch unit manages program flow by performing comparisons of the register file outputs. If the comparison is satisfied, the output is set to a logic value "one," which affects the program counter and triggers the jump to the indicated address. Table 5 details the comparison operations based on the instruction's decoded "funct3" field. The RTL design can be seen in Figure 6.

Table 5. Branch unit functions.

Comparison function	funct3
branch if equal (beq)	000
branch if not equal (bne)	001
branch if less than (blt)	100
branch if greater than or equal (bge)	101
branch if less than unsigned (bltu)	110
branch if greater than or equal unsigned (bgeu)	111

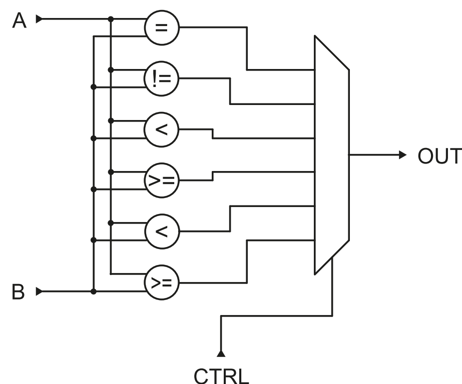

Figure 6. Branch unit RTL.

2.4.4. Multiplication and Division Unit To implement the RV32M extension, a dedicated unit is incorporated to perform the multiplication and division operations included in this extension. This unit comprises seven sub-units, each assigned to execute a specific operation and a multiplexer that controls the output depending on the function designated in the ”funct3” field, as detailed in Table 6. The RTL design of this unit is depicted in Figure 7.

Table 6. Multiplication & division unit functions.

Function	funct3
multiply (mul)	000
multiply high (mulh)	001
multiply high signed-unsigned (mulhsu)	010
multiply high unsigned (mulhu)	011
divide (div)	100
divide unsigned (divu)	101
remainder (rem)	110
remainder unsigned (remu)	111

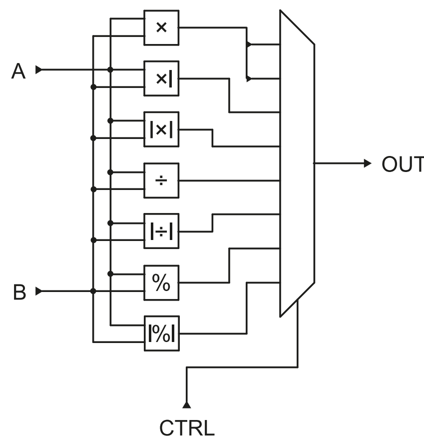

Figure 7. Multiplication & division unit RTL.

2.5. MEMORY STAGE

During this stage, memory access occurs if the instruction requires it, using the address calculated in the execution stage. In the case of write operations, the memory stores

the data from port 2 of the register file. In the case of read operations, the memory sends the read data to the write-back stage.

2.5.1. Data Memory For the implementation of the data memory, the initial consideration was to use 32-bit memory to facilitate operations within one clock cycle. However, a challenge emerged when attempting to implement the ”save byte” (sb) and ”save half” (sh) functions, as writing only half a byte or one byte proved to be complex. Consequently, the decision is to implement an 8-bit wide RAM with 512 addresses, equivalent to 4 kB.

2.5.2. Data Memory Controller To correctly perform read and write operations, it is necessary to implement a memory controller. When the operation is a write operation, the controller takes the 32-bit data, divides it into 4 bytes, and sends it to the memory with its respective address, ordering it in little-endian. When the operation is a read, the controller reads 4 bytes from the base memory address and packages them into 32-bit data for the next stage.

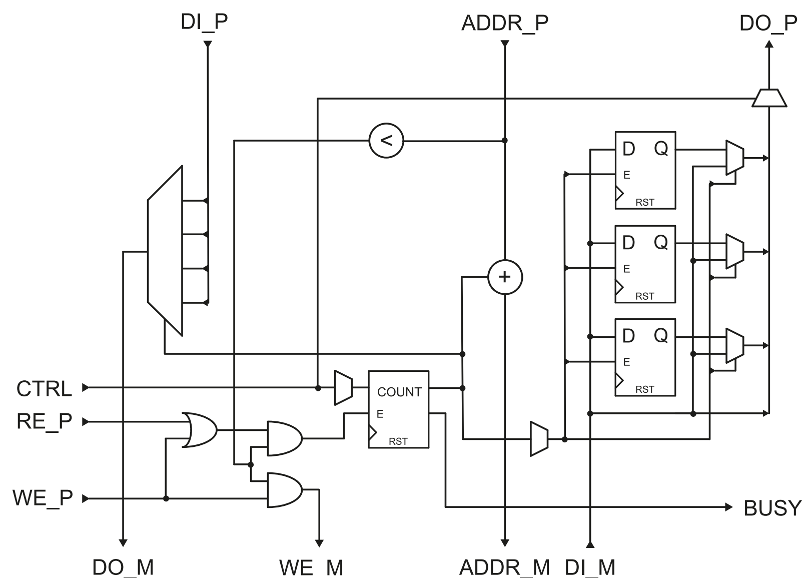

Figure 8. Data memory and its controller.

Figure 8 illustrates the principal components of the controller, including a counter, registers, various multiplexers, and logic gates. The counter governs the enable mux of the registers and the output byte mux. The sum of the base address and the counter generates the addresses of the bytes in memory. Registers and an output mux collaborate to store and assemble the 32-bit output word. The controller creates a busy signal to pause program flow while it operates. Memory operations require more than one clock cycle to execute correctly.

2.6. WRITE-BACK STAGE

In this final phase, a 4-to-1 multiplexer stores the result from the preceding stages in the destination register. The multiplexer, as illustrated in Figure 9, decides whether the information comes from the ALU, the MDU, the immediate extension unit, or the data memory.

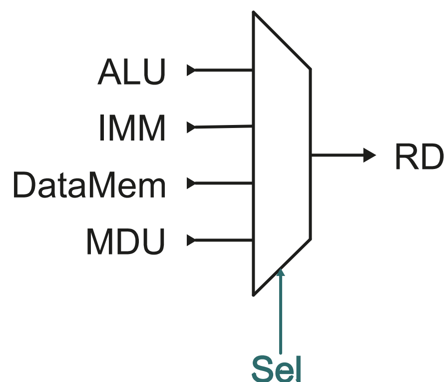

Figure 9. Write-back multiplexer connection.

2.7. CONTROL UNIT

A Control Unit generates the necessary control signals to ensure the accurate execution of each instruction in the mentioned units and stages. This unit comprises six decoders that analyze the information within each instruction and emit the appropriate signals for each respective unit.

2.7.1. Main Decoder The main decoder takes the opcode and funct7 fields to generate signals that direct jump and branch operations in the program counter, signals permitting write operations to the registers file and memory, and signals governing the multiplexers for ALU operands and the selection of the result to be stored. Table 7 provides a comprehensive breakdown of the signals generated in correlation to the opcode and funct7.

Table 7. Encoding main decoder.

Instruction	jump	replace	branch	memory read	memory write	register write	muxA	muxB	muxSave
R-type	0	0	0	0	0	1	00	00	00
I-type	0	0	0	0	0	1	00	01	00
B-type	0	0	1	0	0	0	00	00	00
I-type (load)	0	0	0	1	0	1	00	01	10
S-type	0	0	0	0	1	0	00	01	00
lui	0	0	0	0	0	1	00	00	01
auipc	0	0	0	0	0	1	01	01	00
jal	1	0	0	0	0	1	01	10	00
jalr	1	1	0	0	0	1	01	10	00
R-type (M)	0	0	0	0	0	1	00	00	11

2.7.2. Sort & Extend Decoder A dedicated decoder is integrated for the SortExt unit, examining the funct3 and opcode fields of each instruction. It produces a 3-bit control signal transmitted to the unit outlined in section 2.3.2. The specifications of this signal are detailed in Table 8.

Table 8. Encoding Sort & Extend.

Type	Signal
U Signed	000
I Signed	001
B Signed	010
S Signed	011
J Signed	100
I Unsigned	101
B Unsigned	110

2.7.3. ALU Decoder

The ALU's decoder concatenates the funct3 data, incorporating an additional bit, either "1" or "0," based on the funct3, funct7, and opcode fields of the instruction, as shown in Table 9.

Table 9. Encoding ALU.

funct3	funct7	opcode	Signal
xxx	0000000	R-type	{0,funct3}
xxx	0100000	R-type	{1,funct3}
xxx	xxxxxxx	I-type	{0,funct3}
101	0100000	I-type	{1,funct3}

2.7.4. Branch Decoder

The execution of branch-type instructions involves activating the decoder with the corresponding opcode. The decoder then sends the function extracted from the funct3 field to the unit specified in section 2.4.3.

2.7.5. Memory Decoder

When identifying a memory instruction, a decoder becomes activated, distinguishing between write and read operations. Subsequently, it sends the information extracted from the funct3 field to the memory controller, facilitating the execution of the corresponding operation.

2.7.6. MDU Decoder

This decoder identifies the opcode of type R operations and the funct7 field of the M extension, then sends the function contained in the funct3 field to the MDU for executing the corresponding operation from table 6.

2.8. TOP

Figure 10 depicts the comprehensive micro-architecture design, encompassing all previously described units. It shows both the control signals and the various execution stages.

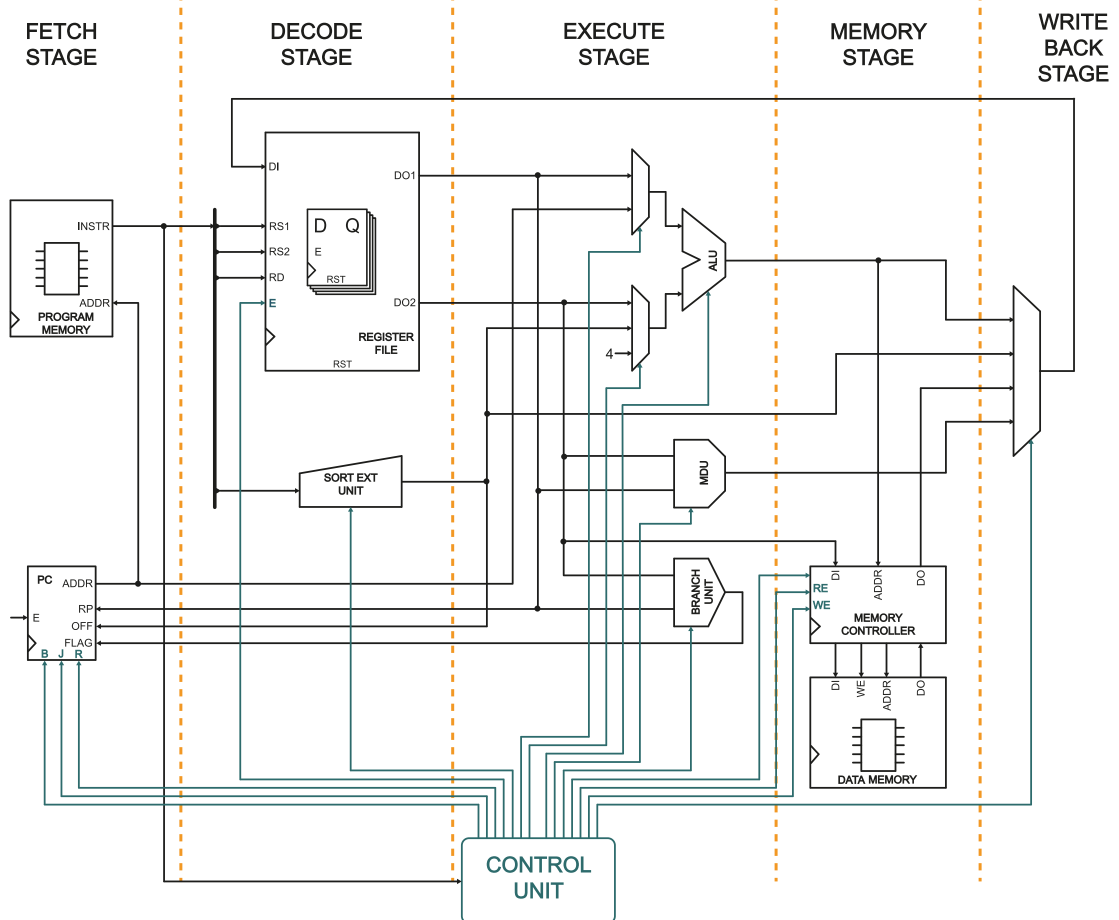

Figure 10. RTL micro-architecture design.

2.9. FUNCTIONAL SIMULATION

A simulation is performed in Vivado by loading a program (see Appendix 1) into memory to confirm the correct execution of all the instructions detailed in Table 1. The following figures show the result of this simulation.

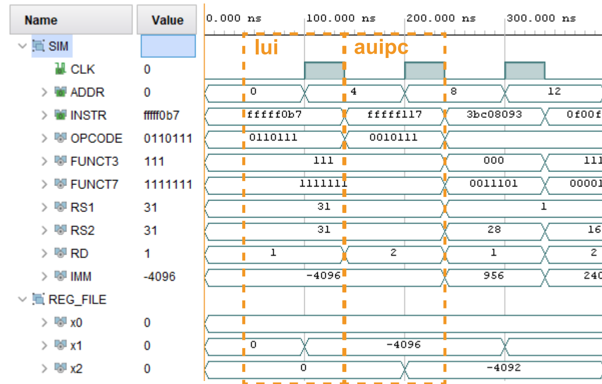

Figure 11. Simulation U type instructions.

Figure 11 shows the load of the immediate value (-4096) from the instruction into register x1 (**lui**). It also illustrates the addition of this immediate value with the program counter value (4) and the subsequent loading of the result into register x2 (**auipc**). The above shows the proper functioning of the U-type instructions.

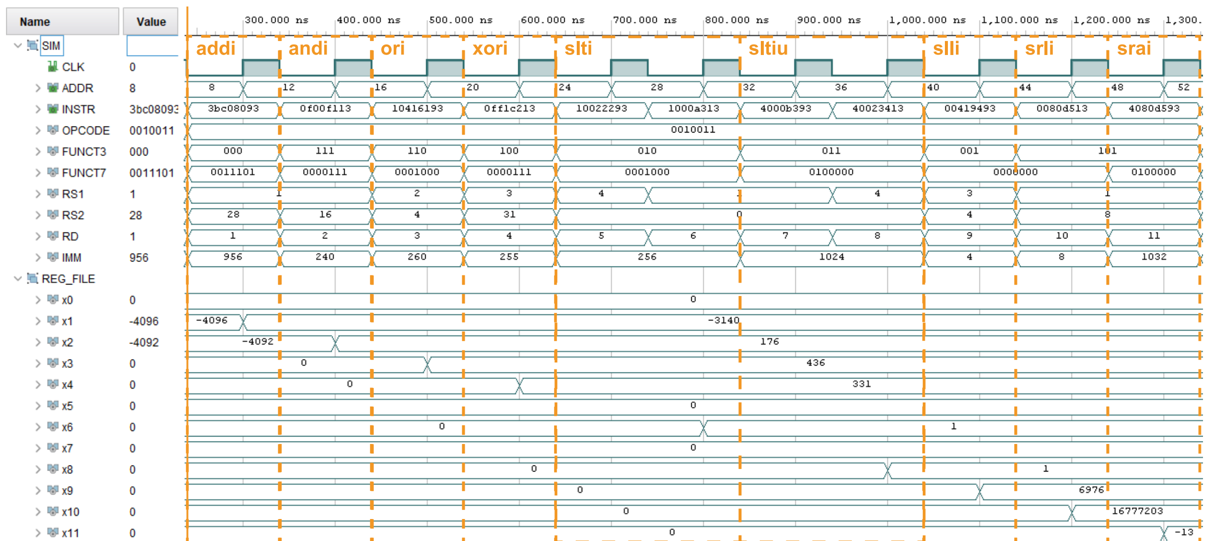

Figure 12. Simulation I type instructions.

In Figure 12, we observe the immediate value (IMM) extracted from the instruction, along with the source register address (RS1) and the funct3 and funct7 fields that specify each operation. For instance, in the **andi** operation, IMM is 240, and RS1 is

1 (-3140), resulting in (176) being loaded into register x2 after the operation. For both **slti** and **sltiu** operations, we consider two scenarios: one resulting in a true outcome and the other not, to verify the correct operation of these instructions. This allows us to confirm the proper functionality of the I-type instructions.

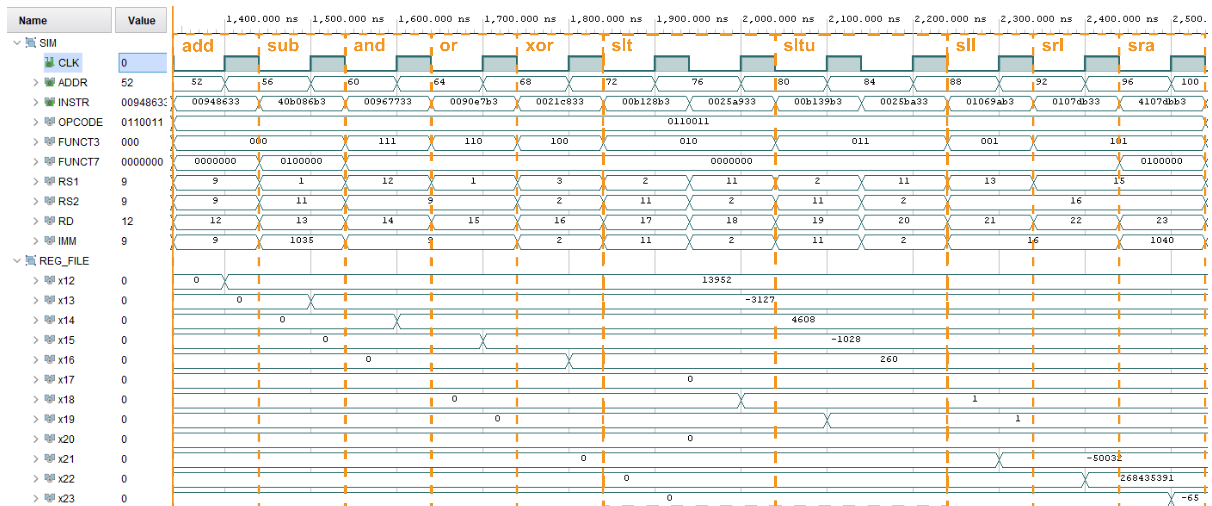

Figure 13. Simulation R type instructions.

Figure 13 shows the result of operating the source registers, whose addresses RS1 and RS2 are extracted from the instruction. Each operation is identified with the fields funct3 and funct7 of the instruction. For example, for the operation **sub** the value of register x11 = -13 is subtracted from the value of register x1 = -3140, resulting in -3127 loaded in register x13. As in the case of the I-Type instructions, the two situations arise for **slt** and **sltu**. Considering the results of the operations we can see the correct operation of the R-type instructions.

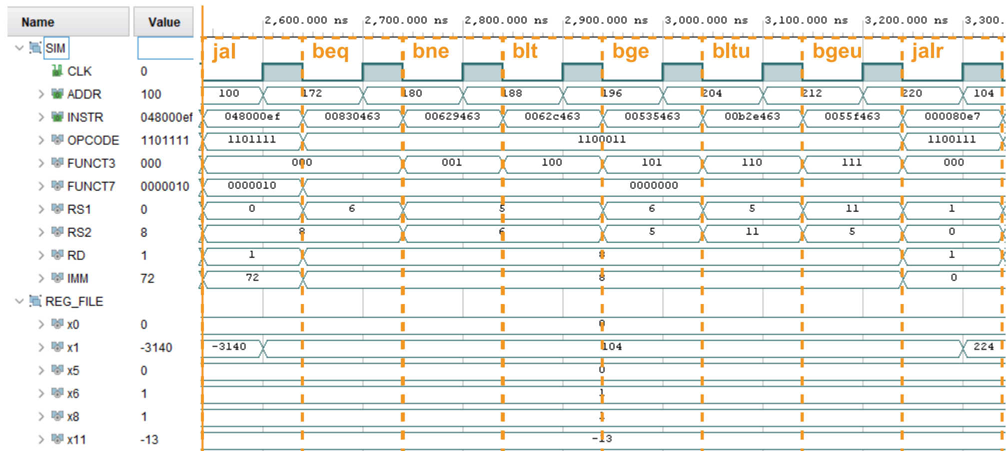

Figure 14. Simulation J and B type instructions.

Figure 14 shows how jump and branch instructions work by adding the offset (IMM) from the instruction to the program counter (ADDR) in the simulation. Additionally, it demonstrates how the **jal** and **jalr** operations use register x2 to store the next address for a later return to the program flow. This helps confirm the correct operation of J-type and B-type instructions.

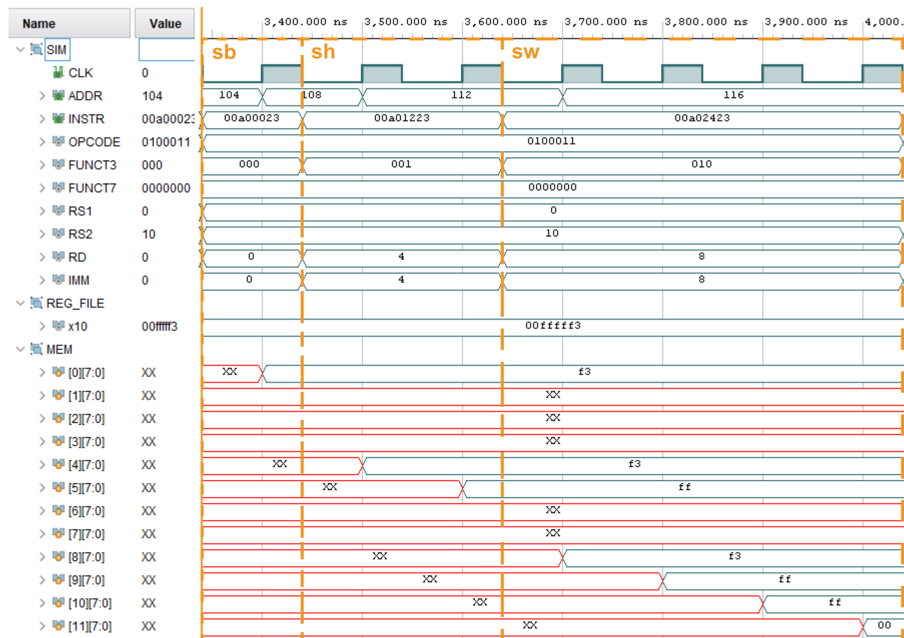

Figure 15. Simulation S type instructions.

Figure 15 depicts the execution of store instructions, which involves storing information in memory. Because of the controller's design, each byte takes one clock cycle to be stored in memory. In this scenario, we take the data 00fffff3 stored in the source register RS2 = 10 and store varying numbers of bytes in memory depending on the instruction. This allows us to verify the proper execution of the store instructions.

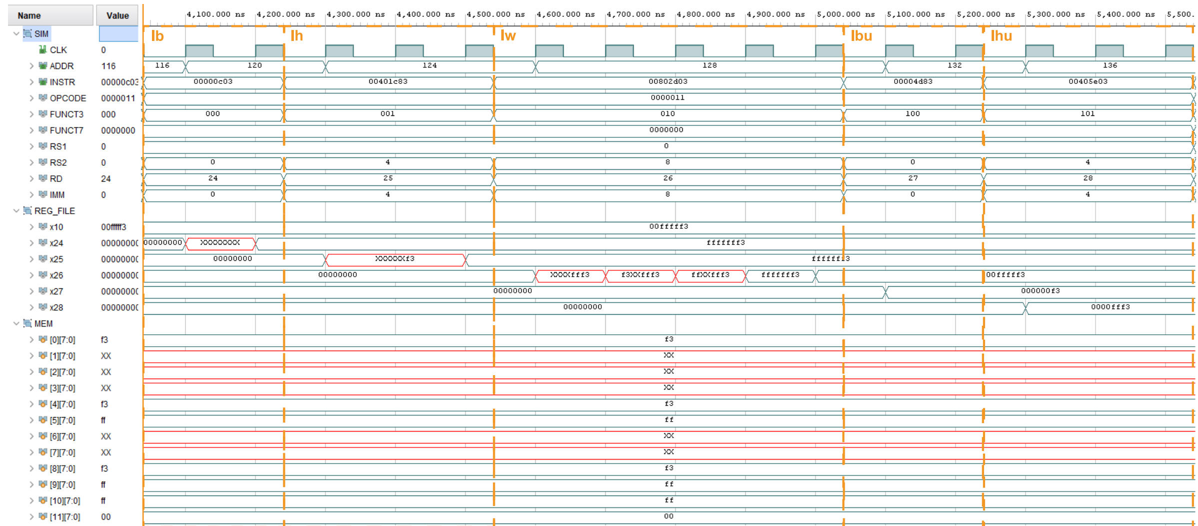

Figure 16. Simulation I type memory instructions.

Figure 16 illustrates the process of reading data from memory to load into a register. These instructions take one clock cycle per byte, with an additional cycle spent. By executing different load instructions using data previously stored in the first three memory addresses, we validate their correct operation and store the data in registers 24-28. By examining the values loaded into the registers, we can confirm the proper functioning of the load instructions.

It's important to mention that the execution time or clock cycle consumption of memory access operations may vary when implementing the micro-architecture in a SoC processor. This can occur due to the use of different types of memory or controllers based on the specific requirements of the implementation.

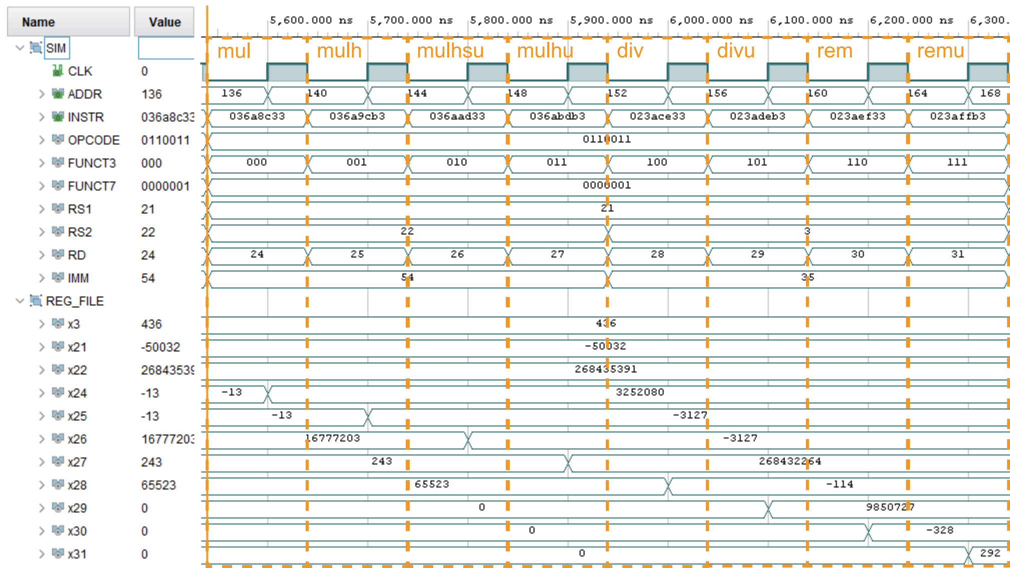

Figure 17. Simulation M extension R type instructions.

Figure 17 illustrates the simulation of M extension operations, where values from source registers RS1 and RS2 are taken, subjected to an operation, and the result is stored in the destination register RD. For instance, in the **mul** operation, the values of registers x21 and x22, -50032 and 268435391 respectively, serve as operands, and the first 32 bits of the product are stored in the destination register x24, resulting in 3252080. By observing the outcomes of all operations, we can confirm the proper functioning of the M extension instructions.

3. MICRO-ARCHITECTURE IMPLEMENTATION

3.1. FPGA

The implementation has opted for the AMD Xilinx KC705 platform, shown in Figure 18. This platform incorporates the XC7K325T-2FFG900C FPGA, a member of Xilinx 7 series FPGAs constructed using a 28 nm process technology from the last generation, providing high performance and low power consumption. The 28 nm FPGA process is the most relevant parameter for this selection because the plan for the future involves implementing the design on-chip using the same technology. Additionally, the KC705 platform features a fixed oscillator of 200 MHz, jitter attenuated clock, onboard JTAG configuration circuitry enabling configuration over USB, UART to USB Bridge, 5x push buttons, 4x DIP switches, 7 I/O pins available through the LCD header, 2x16 LCD, and 8x LEDs. These features help with programming and testing the functionality of the micro-architecture.

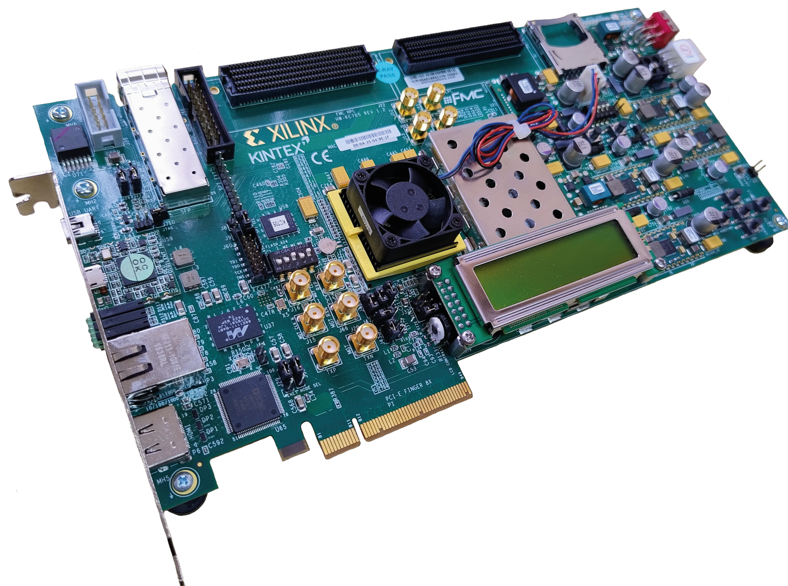

Figure 18. KC705 Platform.

3.2. PERIPHERAL DESIGN

The controllers for the LCD and leds, integrated into the KC705 platform, have been designed, as well as a timer and an interrupt unit for hardware delays.

3.2.1. Interrupt Unit Designing an interrupt unit aims to deactivate the micro-architecture when the data memory controller or timer is active. Keeping only the memory controller or timer active prevents unnecessary power consumption in other parts of the micro-architecture. Figure 19 shows the logic of the interruptions unit.

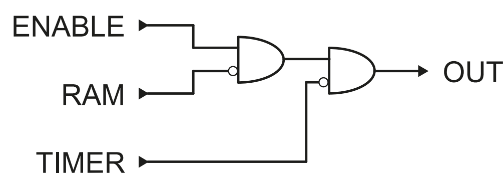

Figure 19. RTL design interrupts unit.

3.2.2. Timer Figure 20 shows the RTL of the timer. If the memory address is 1024 and the WE signal is high, the register saves the input data reduced by two units, setting a set/reset flip-flop. The output signal of the flip-flop triggers the counter. When the count exceeds the value stored in the register, a comparator resets both the flip-flop and the counter.

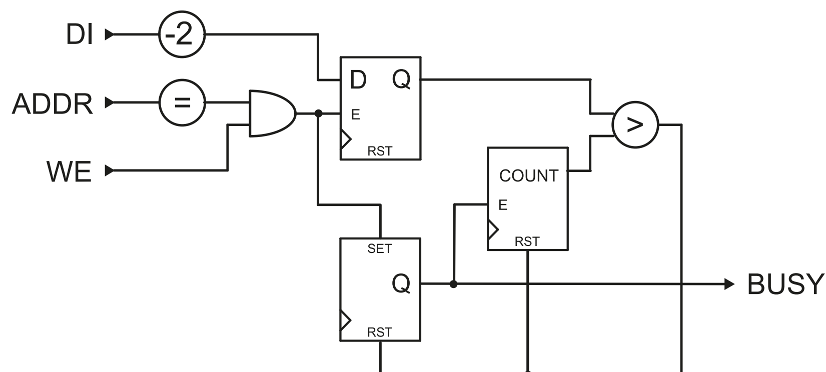

Figure 20. RTL design timer.

3.2.3. LCD and Leds Two controllers are designed to store information from the memory data bus for the leds and the LCD as output peripherals, based on the memory address bus, with 1028 designated for the leds and 1032 assigned for the LCD. Figure 21 illustrates the RTL of the controllers, featuring identical logic but differing in the number of bits: 8 for the leds and 6 for the LCD connections.

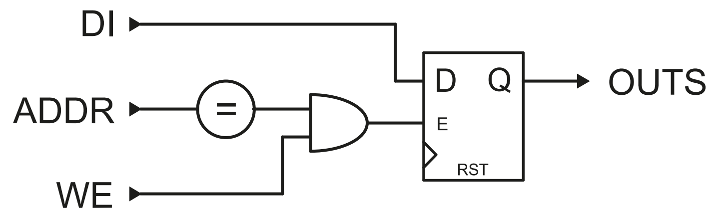

Figure 21. RTL design LCD and leds controllers.

3.3. UART AND BOOTLOADER DESIGN

The UART unit designed loads programs into memory via USB, allowing external programming without generating a new bitstream. This unit operates at 128 kbaud. As the UART communication transmits 8 bits of data, the bootloader unit processes the 8 bits from each transmission, packs them into 32 bits, completes the instruction's 32 bits, and loads them into memory.

3.4. CLOCK UNIT DESIGN

One unit takes the 200 MHz from the FPGA and generates the two clock domains needed for the micro-architecture to work in the FPGA implementation. The UART unit design requires a 2.083 MHz clock, while the rest of the micro-architecture necessitates a 10 MHz clock.

The first option for developing this unit involved utilizing the Mixed-Mode Clock Manager (MMCM) or Phase-Locked Loop (PLL) units integrated into the FPGA. However, these options have limitations in the lowest frequency they can provide and would require

clock buffers with the ability to divide frequency. As a result, the decision involves exclusively employing the regional clock buffers for generating the desired frequencies.

3.5. SYNTHESIS AND IMPLEMENTATION

During the synthesis and implementation process in Vivado, the focus is on identifying optimal strategies for achieving improved performance.

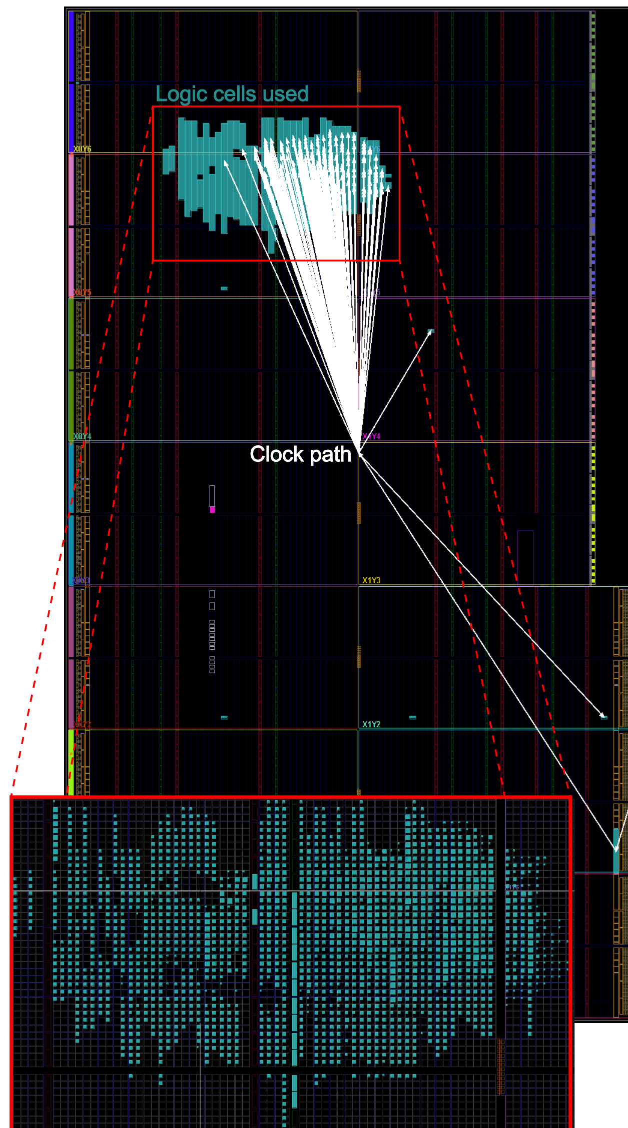

Figure 22. Default implementation.

Figure 22 shows the implementation in the device, with the placement of cells and the clock path obtained from executing the process with default strategies. In this first iteration, the MMCM unit generated the desired frequency for the clock signal, reaching only 8 MHz.

In a second iteration, a Pblock statement was used to restrict the design into the X1Y1 clock region of the device to reduce the clock path, thus, obtaining a higher frequency. In this case, it was necessary to use the BUFH of the device to carry the clock signal from the MMCM to the micro-architecture. Figure 22 shows the results of this iteration.

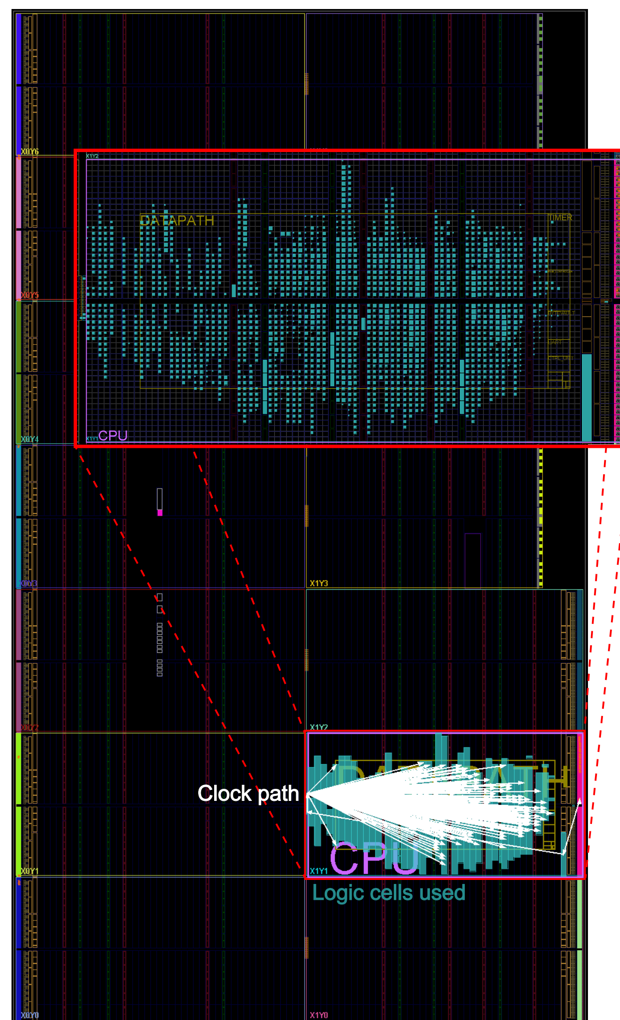

Figure 23. Clock region Implementation.

Finally, replacing the MMCM with the Regional Clock Buffer to generate the clock signal and restricting the Pblock to only one region of slices, we obtain a more compact implementation with a shorter clock path, as illustrated in Figure 24. In addition to this, taking into account the Vivado UG901 and UG904 guides, the synthesis and implementation strategies are configured as 'Flow_PerfOptimized_high' and 'Performance_ExplorePostRoutePhysOpt' respectively, achieving a 25% improvement in frequency up to 10MHz.

Figure 24. Final implementation.

4. RESULTS

The results demonstrate the successful performance of the RISC-V micro-architecture, shown in Figure 25, which can be effectively deployed on an FPGA, as seen in Figure 26, using standard design and implementation tools like Vivado. The operating frequency and power consumption are consistent with low power expectations, and the resource utilization on the FPGA is adequate for the proposed design. Table 10 shows these results, and a comparison with other cores focuses on low-power [14].

This project establishes the groundwork for implementing the forthcoming OnChip microcontroller. In future work, considerations may include adding pipeline stages, enhancing memory specifically for the microcontroller, and following a design process to create the GDS of the processor using this micro-architecture.

---

[14] Ricardo NÚÑEZ-PRIETO, David CASTELLS-RUFAS, and Lluís TERÉS-TERÉS. “RisCO2: SoC Implementation and Performance Evaluation of RISC-V Processors for Low-Power CO2 Concentration Sensing”. In: (2023)

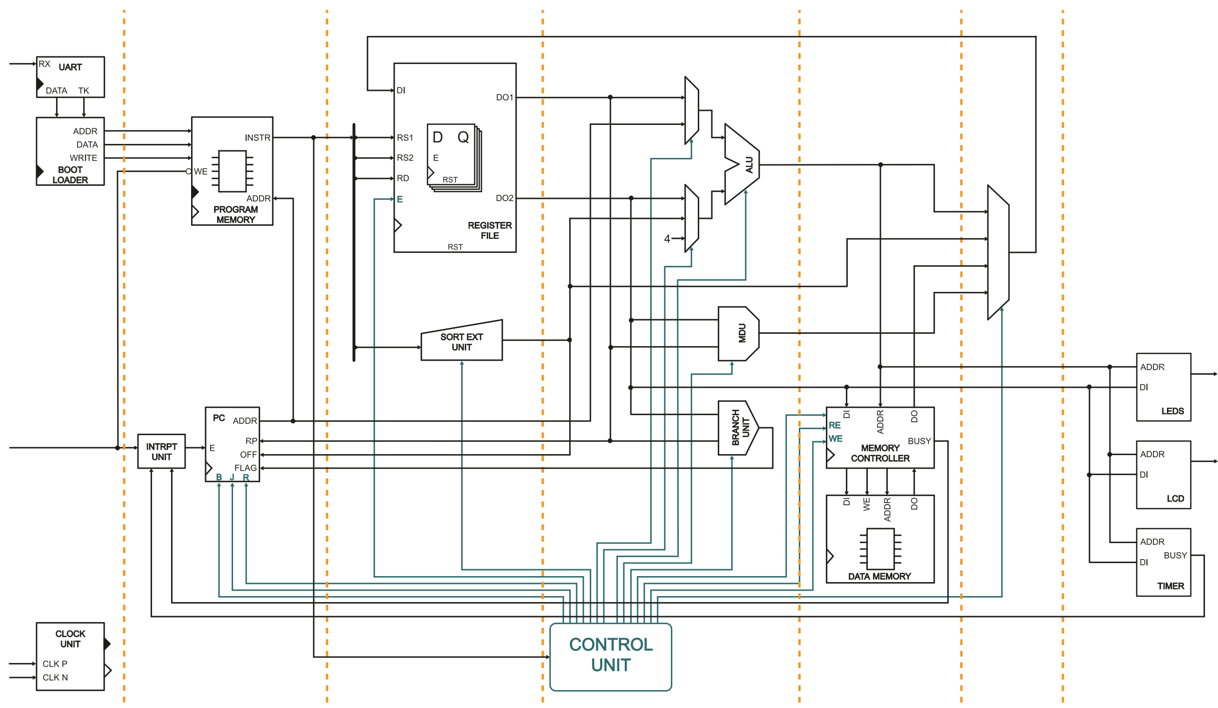

Figure 25. RTL micro-architecture implementation.

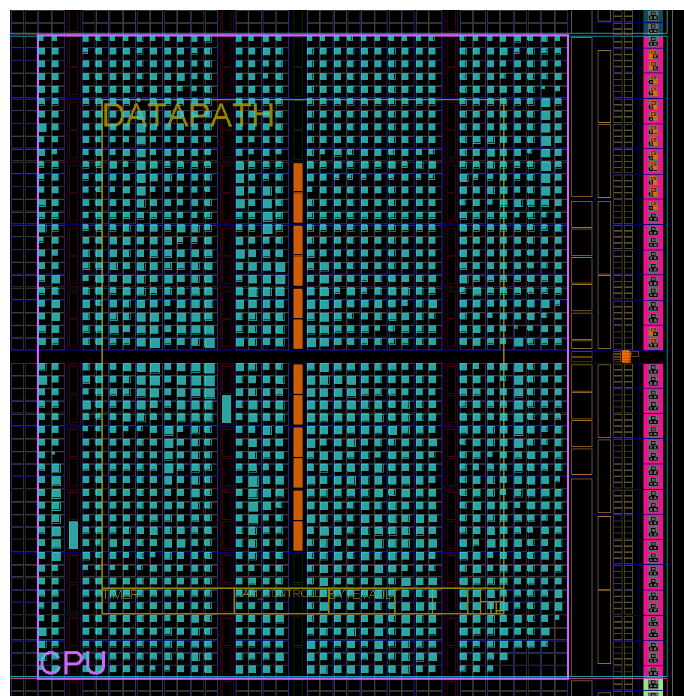

Figure 26. Device implementation.

Table 10. Performance & utilization results.

Feature	This work	Zero-riscy	Ri5cy	CV32E40P
LUT	6306	3171	11912	9072
Flip-Flops	1237	1928	4249	2553
DSP	12	1	8	7
Power [mW]	34	20	52	49
BRAM	1	-	-	-
IO	23	-	-	-
Pipeline stages	0	2	4	4
Platform	Xilinx Kintex7 KC705	Xilinx Artix7 Nexys 4	Xilinx Artix7 Nexys 4	Xilinx Artix7 Nexys 4
Technology [nm]	28	28	28	28
Frequency [MHz]	10	25	25	25

Table 10 compares this work and three others with similar focuses on low power consumption. By analyzing the use of LUTs, Flip-Flops, power, pipeline stages, and frequency, it is observed that, although this work has a lower frequency, it also has fewer Flip-Flops and does not use as many LUTs, which makes it an attractive option for low-power devices in terms of energy and resources. Furthermore, adding pipeline stages could enhance the frequency without significantly raising power consumption compared to the other three studies, thus improving overall performance.

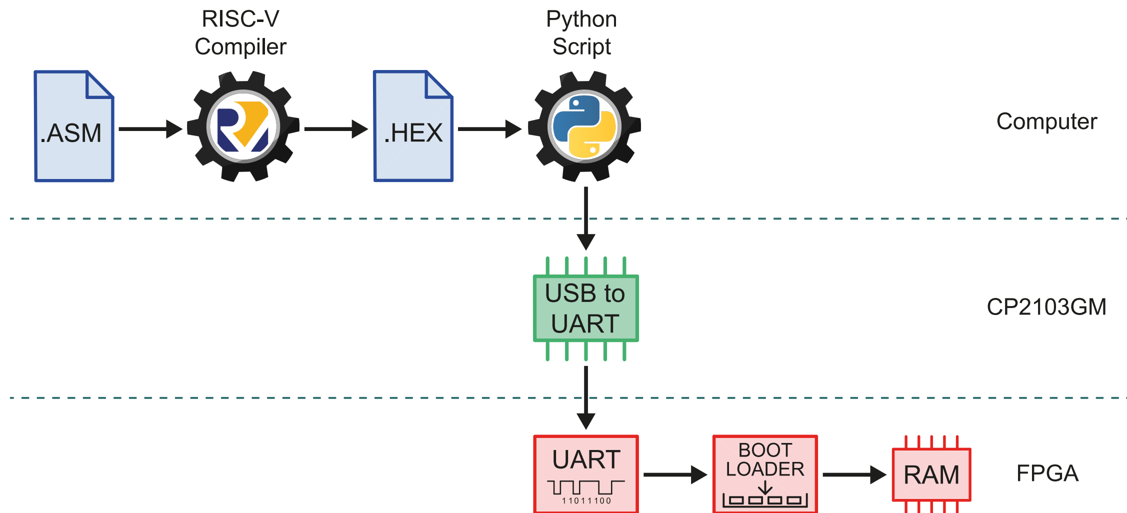

Figure 27. Programming Flow.

Figure 27 illustrates the programming process flow based on the methodology and modules in [15]. The process begins with an assembler program, which is compiled by a RISC-V compiler into a hex file containing the instructions. Next, a Python script transforms these instructions into binary format and sends them to the FPGA using UART. Within the FPGA, the UART and BOOTLOADER modules, as mentioned in section 3.3, are responsible for programming the RAM for further execution.

[15] Hanssel MORALES. “A Verified RISC-V I Based Processor with an External Debugging Capability”. Escuela de Ingeniería Eléctrica, Electrónica y Telecomunicaciones. Undergraduate Work. Bucaramanga, Colombia: Universidad Industrial de Santander, 2021

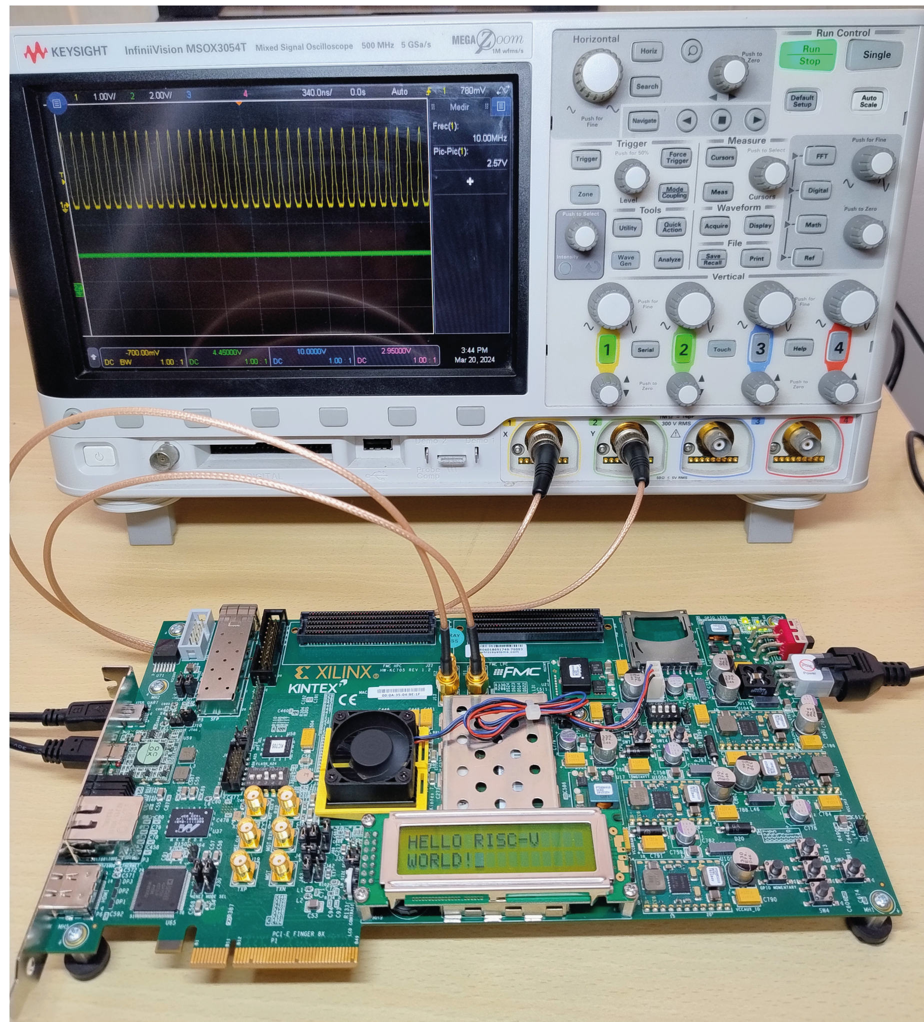

Figure 28. FPGA demo setup.

Performing the setup of Figure 28, and following the programming flow mentioned above, the program of the Appendix 2, which executes the Fibonacci sequence, is compiled and loaded. Figure 29 illustrates the first steps of the execution, the process of programming and execution of the program can be seen in [16].

[16] RISC-V Fibonacci Demo: `https://youtu.be/GehAuWf0Pok`.

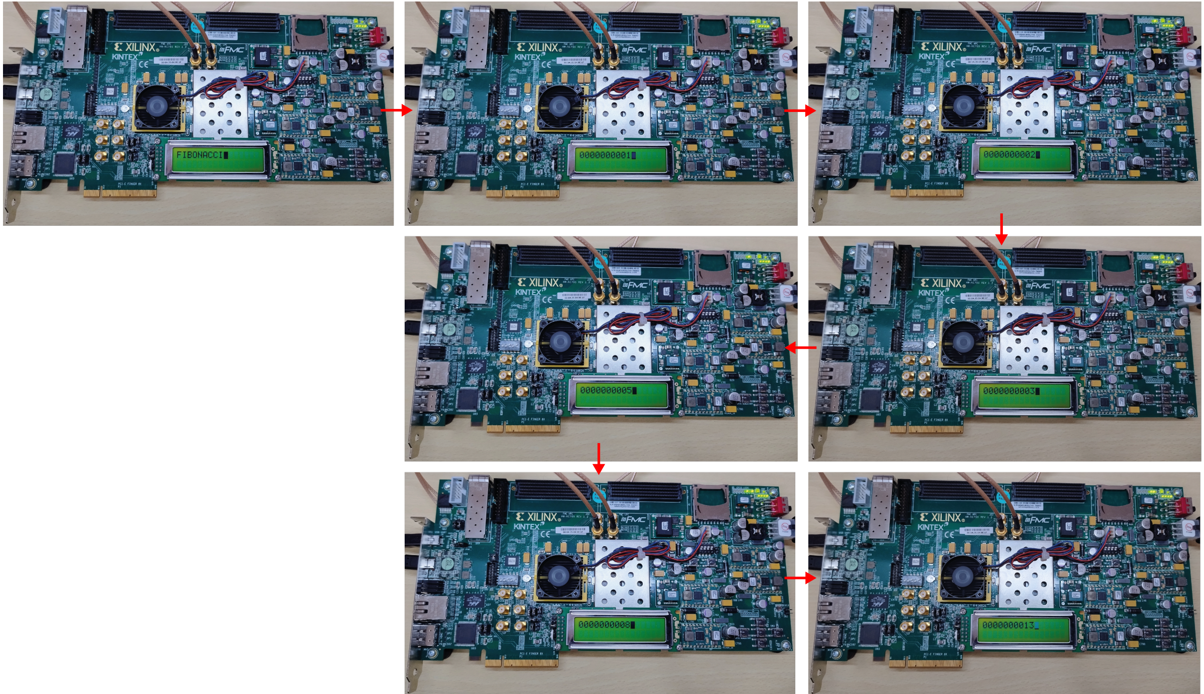

Figure 29. Fibonacci Demo.

The Verilog files for all the units discussed are included in the appendices for a comprehensive understanding of their design and structure. These files are organized hierarchically, making it easier to grasp the entire micro-architecture. Examining them enables a deep dive into the technical aspects of the design, covering everything from register definitions to execution unit implementations, offering a thorough and detailed insight into the development process.

4.1. CONCLUSIONS

- A micro-architecture was designed following the ISA RISC-V RV32IM specification. It operates on one cycle per instruction, except for the memory instructions, which could be upgraded to one cycle per instruction with a change of memories and their controller when implementing the final processor.

- The micro-architecture design was successfully implemented on a state-of-the-art 28 nm FPGA. Choosing this FPGA allowed for sufficient logic and memory resources to accommodate the designed micro-architecture while maintaining a good balance between area occupied, power consumption, and performance, considering other similar work focused on low power consumption.

- A detailed documentation was created covering the whole process of designing and implementing the RISC-V micro-architecture on the FPGA. This documentation includes schematics of the micro-architecture, data flow diagrams, specifications of the hardware used, a description of the design methodology, results of simulations and tests on the FPGA, as well as analysis of the results obtained.

- An area and placement optimization was carried out, following a floor-planning methodology based on the resources available in the FPGA, which resulted in a 25% improvement in the micro-architecture operating frequency.

- Necessary blocks and scripts were designed to allow the reprogramming of the micro-architecture through the UART protocol, avoiding the need to generate a bitstream file at each program change.

BIBLIOGRAPHY

BAI, Chen et al. “BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework”. In: (2021), pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643455 (cit. on p. 15).

COLUCCIO, Andrea et al. “RISC-Vlim, a RISC-V Framework for Logic-in-Memory Architectures”. In: *Electronics* 11.19 (2022). DOI: 10.3390/electronics11192990 (cit. on p. 15).

DURAN, Ckristian et al. “A 32-bit RISC-V AXI4-lite bus-based microcontroller with 10-bit SAR ADC”. In: (2016), pp. 315–318. DOI: 10.1109/LASCAS.2016.7451073 (cit. on p. 13).

DURAN, Ckristian et al. “An Energy-Efficient RISC-V RV32IMAC Microcontroller for Periodical-Driven Sensing Applications”. In: (2020), pp. 1–4. DOI: 10.1109/CICC48029.2020.9075877 (cit. on p. 13).

GRANDVIEWRESEARCH. *Wearable Technology Market Size, Share & Trends Analysis Report By Product (Head & Eyewear, Wristwear), By Application (Consumer Electronics, Healthcare), By Region (Asia Pacific, Europe), And Segment Forecasts, 2023 - 2030*. Tech. rep. (cit. on p. 12).

IMARCGROUP. *Wearable Technology Market: Global Industry Trends, Share, Size, Growth, Opportunity and Forecast 2023-2028*. Tech. rep. (cit. on p. 12).

MARIOTTI, Gianfranco and Roberto GIORGI. “WebRISC-V: A 32/64-bit RISC-V pipeline simulation tool”. In: *SoftwareX* 18 (2022), p. 101105. DOI: https://doi.org/10.1016/j.softx.2022.101105 (cit. on p. 15).

MATTHEWS, Eric and Lesley SHANNON. “TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features”. In: (2017), pp. 1–4. DOI: 10.23919/FPL.2017.8056766 (cit. on p. 15).

MORALES, Hanssel. “A Verified RISC-V I Based Processor with an External Debugging Capability”. Escuela de Ingeniería Eléctrica, Electrónica y Telecomunicaciones. Undergraduate Work. Bucaramanga, Colombia: Universidad Industrial de Santander, 2021 (cit. on p. 45).

NÚÑEZ-PRIETO, Ricardo, David CASTELLS-RUFAS, and Lluís TERÉS-TERÉS. “RisCO2: SoC Implementation and Performance Evaluation of RISC-V Processors for Low-Power CO2 Concentration Sensing”. In: (2023) (cit. on p. 42).

PATTERSON, David and Andrew WATERMAN. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017 (cit. on pp. 18, 21).

POORHOSSEINI, Mehrdad, Wolfgang NEBEL, and Kim GRÜTTNER. “A Compiler Comparison in the RISC-V Ecosystem”. In: (2020), pp. 1–6. DOI: 10.1109/COINS49042.2020.9191411 (cit. on p. 15).

WANG, Lie, Ye ZHANG, and Peter G BRUCE. “Batteries for wearables”. In: *National Science Review* 10.1 (Mar. 2022), nwac062. DOI: 10.1093/nsr/nwac062. eprint: https://academic.oup.com/nsr/article-pdf/10/1/nwac062/48727960/nwac062.pdf (cit. on p. 12).

WATERMAN, Andrew and Krste ASANOVIĆ, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation. May 2017 (cit. on p. 19).

APPENDICES

Appendix 1: ISA Testbench Program

```
#       U-Type
lui     x1, 0xfffff             # x1 = fffff000       = -4096
auipc   x2, 0xfffff             # x2 = fffff004       = -4092
#       I-Type
addi    x1, x1, 0x3bc           # x1 = fffff3bc       = -3140
andi    x2, x1, 0x0F0           # x2 = 000000b0       = 176
ori     x3, x2, 0x104           # x3 = 000001b4       = 436
xori    x4, x3, 0x0FF           # x4 = 0000014b       = 331
slti    x5, x4, 0x100           # x5 = 00000000       = 0
slti    x6, x1, 0x100           # x6 = 00000001       = 1
sltiu   x7, x1, 0x400           # x7 = 00000000       = 0
sltiu   x8, x4, 0x400           # x8 = 00000001       = 1
slli    x9, x3, 0x4             # x9 = 00001b40       = 6976
srli    x10, x1, 0x8            # x10 = 00fffff3      = 16777203
srai    x11, x1, 0x8            # x11 = fffffff3      = -13
#       R-Type
add     x12, x9, x9             # x12 = 00003680      = 13952
sub     x13, x1, x11            # x13 = fffff3c9      = -3127
and     x14, x12, x9            # x14 = 00001200      = 4608
or      x15, x1, x9             # x15 = fffffbfc      = -1028
xor     x16, x3, x2             # x16 = 00000104      = 260
slt     x17, x2, x11            # x17 = 00000000      = 0
slt     x18, x11, x2            # x18 = 00000001      = 1
sltu    x19, x2, x11            # x19 = 00000001      = 1
sltu    x20, x11, x2            # x20 = 00000000      = 0
sll     x21, x13, x16           # x21 = ffff3c90      = -50032
srl     x22, x15, x16           # x22 = 0fffffbf      = 268435391
sra     x23, x15, x16           # x23 = ffffffbf      = -65
#       J-Type
jal     jump
#       S-Type
sb      x10, 0(x0)              # RAM[0] = 000000f3   = 243
sh      x10, 4(x0)              # RAM[4] = 0000fff3   = 65523
sw      x10, 8(x0)              # RAM[8] = 00fffff3   = 16777203
#       IL-Type
```

```
        lb      x24, 0(x0)          # x24 = fffffff3        = -13
        lh      x25, 4(x0)          # x25 = fffffff3        = -13
        lw      x26, 8(x0)          # x26 = 00fffff3        = 16777203
        lbu     x27, 0(x0)          # x27 = 000000f3        = 243
        lhu     x28, 4(x0)          # x28 = 0000fff3        = 65523
        #       RM-Type
        mul     x24, x21, x22       # x24 = 00319f70        = 3252080
        mulh    x25, x21, x22       # x25 = ffff f3c9       = -3127
        mulhsu  x26, x21, x22       # x26 = fffff3c9        = -3127
        mulhu   x27, x21, x22       # x27 = 0ffff388        = 268432264
        div     x28, x21, x3        # x28 = ffffff8e        = -114
        divu    x29, x21, x3        # x29 = 00964f67        = 9850727
        rem     x30, x21, x3        # x30 = fffffeb8        = -328
        remu    x31, x21, x3        # x31 = 00000124        = 292
        #       END
end:    beq     x0, x0, end
        #       B-Type
jump:   beq     x6, x8, next1
        beq     x0, x0, end
next1:  bne     x5, x6, next2
        beq     x0, x0, end
next2:  blt     x5, x6, next3
        beq     x0, x0, end
next3:  bge     x6, x5, next4
        beq     x0, x0, end
next4:  bltu    x5, x11, next5
        beq     x0, x0, end
next5:  bgeu    x11, x5, next6
        beq     x0, x0, end
        #       IJ-Type
next6:  jalr    ra
```

Appendix 2: Fibonacci's Test

```
#       .equ TIMMER     0x400    # 1024
#       .equ LEDS       0x404    # 1028
#       .equ LCD        0x408    # 1032

        .text
        li      s11,    10              # time_HIGH
        li      s10,    1000            # time_LOW
```

```
        li      s9,     50000           # time_CLEAR
        li      s0,     0               # prev
        li      s1,     1               # curr
        li      s2,     0               # fib
        li      s3,     40              # n_term
        li      s4,     0               # count
        li      s5,     0xFFFFFF        # time_SHOW


CONFIG: li      a0,     0x22            # FUNCTION SET
        jal     SEND
        li      a0,     0x28
        jal     SEND
        li      a0,     0x22            # FUNCTION SET
        jal     SEND
        li      a0,     0x28
        jal     SEND
        li      a0,     0x20            # DISPLAY ON/OFF CURSOR Y BLINK
        jal     SEND
        li      a0,     0x2F
        jal     SEND
        jal     CLEAR


START:  li      a0,     0x34            # F
        jal     SEND
        li      a0,     0x36
        jal     SEND
        li      a0,     0x34            # I
        jal     SEND
        li      a0,     0x39
        jal     SEND
        li      a0,     0x34            # B
        jal     SEND
        li      a0,     0x32
        jal     SEND
        li      a0,     0x34            # O
        jal     SEND
        li      a0,     0x3F
        jal     SEND
        li      a0,     0x34            # N
        jal     SEND
```

```
        li      a0,     0x3E
        jal     SEND
        li      a0,     0x34            # A
        jal     SEND
        li      a0,     0x31
        jal     SEND
        li      a0,     0x34            # C
        jal     SEND
        li      a0,     0x33
        jal     SEND
        li      a0,     0x34            # C
        jal     SEND
        li      a0,     0x33
        jal     SEND
        li      a0,     0x34            # I
        jal     SEND
        li      a0,     0x39
        jal     SEND
        sw      s5,     0x400(x0)

FIB:    jal     CLEAR
        add     s2,     s0,     s1      # fib = prev + curr
        mv      s0,     s1              # prev = curr
        mv      s1,     s2              # curr = fib
        addi    s4,     4,      1       # count++
        sw      s4,     0x404(x0)       # leds
        mv      a1,     s2              # split = fib

SPLIT:  li      s6,     1000000000      # length_SPLIT
        li      s7,     10              # step_SPLIT
R:      div     a2,     a1,     s6
        jal     LCD
        rem     a1,     a1,     s6
        div     s6,     s6,     s7
        beq     s6,     x0,     SHOW
        j       R

LCD:    mv      s8,     ra              # save_addr
        li      a0,     0x33
```

```
        jal     SEND
        ori     a0,     a2,     0x30
        jal     SEND
        mv      ra,     s8
        ret


SHOW:   beq     s4,     s3, END
        sw      s5,     0x400(x0)
        j       FIB


SEND:   sw      a0,     0x408(x0)
        sw      s11,    0x400(x0)
        sw      x0,     0x408(x0)
        sw      s10,    0x400(x0)
        ret


CLEAR:  mv      s8,     ra              # save_addr
        li      a0,     0x20            # CLEAR DISPLAY
        jal     SEND
        li      a0,     0x21
        jal     SEND
        sw      s9,     0x400(x0)
        mv      ra,     s8
        ret


END:    j END
```

Appendix 3: TOP

```
`timescale 1ns/1ps

module PROCESSOR (
    input CLOCK_P, CLOCK_N,
    input RESET_CLK,
    input RX_UART,
    input ENABLE, RESET,
    output CLOCK_MAIN, RX_SMA,
```

```
    output [7:0] LED,
    output E, RS, RW, D4, D5, D6, D7
);


wire clock_uart, clock_main;


CLOCKING CLOCKING(
    .CLOCK_P(CLOCK_P),
    .CLOCK_N(CLOCK_N),
    .RESET(RESET_CLK),
    .CLOCK_UART(clock_uart),
    .CLOCK_MAIN(clock_main)
);


wire ram_busy, busy_timer, en_pc;


ENABLER ENABLER(
  .IN(ENABLE),
  .RAM(ram_busy),
  .TIMER(busy_timer),
  .OUT(en_pc)
);


wire tick;
wire [7:0] byte;


assign RX_SMA = RX_UART;


UART UART(
    .CLOCK(clock_uart),
    .RESET(RESET),
    .ENABLE(ENABLE),
    .RX_UART(RX_UART),
    .DATA_OUT(byte),
    .TICK(tick)
```

```
);


wire clock_mem, write_enable;
wire [7:0] addr_write;
wire [31:0] data_write;


BYTEVAULT BYTEVAULT(
    .CLOCK_IN(clock_uart),
    .RESET(RESET),
    .DATA_IN(byte),
    .READ(tick),
    .CLOCK_OUT(clock_mem),
    .ADDRESS(addr_write),
    .DATA_OUT(data_write),
    .WRITE(write_enable)
);


wire [31:0] instr, addr_instr;


MEMORY MEMORY(
    .CLK_WRITE(clock_mem),
    .ADDRESS_WRITE(addr_write),
    .DATA_WRITE(data_write),
    .WRITE_ENABLE(write_enable),
    .CLK_READ(clock_main),
    .ADDRESS_READ(addr_instr[9:2]),
    .READ_ENABLE(en_pc),
    .DATA_READ(instr)
);


wire [2:0] funct3;
wire [6:0] opcode, funct7;
wire j, r, b, reg_wr, mem_wr, mem_rd;
wire [1:0] ctrl_a, ctrl_b, ctrl_s;
wire [2:0] ctrl_ext, ctrl_mdu, ctrl_branch, ctrl_mem;
wire [3:0] ctrl_alu;
```

```
assign funct3 = instr[14:12];
assign funct7 = instr[31:25];
assign opcode = instr[6:0];


CTRL_UNIT CTRL_UNIT(
    .FUNCT3(funct3),
    .OPCODE(opcode),
    .FUNCT7(funct7),
    .J(j),
    .R(r),
    .B(b),
  .MEM_RD(mem_rd),
    .MEM_WR(mem_wr),
    .REG_WR(reg_wr),
    .Ctrl_A(ctrl_a),
    .Ctrl_B(ctrl_b),
    .Ctrl_S(ctrl_s),
    .Ctrl_EXT(ctrl_ext),
    .Ctrl_MDU(ctrl_mdu),
    .Ctrl_BRANCH(ctrl_branch),
    .Ctrl_MEM(ctrl_mem),
    .Ctrl_ALU(ctrl_alu)
);


wire [24:0] data_instr;
wire [31:0] addr_ram_c, data_uA_ram, data_ram_uA;


assign data_instr = instr[31:7];


DATAPATH DATAPATH(
    .CLOCK(clock_main),
    .EN_PC(en_pc),
    .REG_WR(reg_wr),
    .B(b),
    .J(j),
    .R(r),
    .Ctrl_A(ctrl_a),
```

```
    .Ctrl_B(ctrl_b),
    .Ctrl_S(ctrl_s),
    .Ctrl_EXT(ctrl_ext),
    .Ctrl_Branch(ctrl_branch),
    .Ctrl_Mult(ctrl_mdu),
    .Ctrl_ALU(ctrl_alu),
    .ADDR_INSTR(addr_instr),
    .INSTR(data_instr),
    .ADDR_RAM(addr_ram_c),
    .DI_RAM(data_uA_ram),
    .DO_RAM(data_ram_uA),
    .RESET(RESET)
);


wire clock_ram, we_ram;
wire [8:0] addr_ram;
wire [7:0] data_ram_c, data_c_ram;


RAM_CONTROLLER RAM_CONTROLLER(
    .CLOCK(clock_main),
    .RESET(RESET),
    .Ctrl_MEM(ctrl_mem),
    .RE_P(mem_rd),
    .WE_P(mem_wr),
    .ADDR_P(addr_ram_c),
    .DI_P(data_uA_ram),
    .DO_P(data_ram_uA),
    .CLOCK_M(clock_ram),
    .WE_M(we_ram),
    .ADDR_M(addr_ram),
    .DI_M(data_ram_c),
    .DO_M(data_c_ram),
  .BUSY(ram_busy)
);


RAM RAM(
    .CLOCK(clock_ram),
    .WE(we_ram),
    .ADDR(addr_ram),
```

```
    .DI(data_c_ram),
    .DO(data_ram_c)
);


LEDS LEDS(
    .CLOCK(clock_main),
    .WE(mem_wr),
    .RESET(RESET),
    .DI(data_uA_ram[7:0]),
    .ADDR(addr_ram_c),
    .LED(LED)
);


LCD LCD(
  .CLOCK(clock_main),
  .WE(mem_wr),
  .RESET(RESET),
  .ADDR(addr_ram_c),
  .DI(data_uA_ram[5:0]),
  .E(E),
  .RS(RS),
  .RW(RW),
  .D4(D4),
  .D5(D5),
  .D6(D6),
  .D7(D7)
);


TIMER TIMER(
  .CLOCK(clock_main),
  .ENABLE(mem_wr),
  .ADDR(addr_ram_c),
  .ENDING(data_uA_ram),
  .BUSY(busy_timer),
  .RESET(RESET)
);


BUFG BUF_main (
```

```
      .O(CLOCK_MAIN),
      .I(clock_main)
);

endmodule
```

Appendix 4: Clock Unit

```
`timescale 1ns/1ps

module CLOCKING (
    input CLOCK_P, CLOCK_N,
    input RESET,
    output CLOCK_UART,
    output CLOCK_MAIN
);


wire clk0;


IBUFDS #(
   .DIFF_TERM("TRUE"),
   .IBUF_LOW_PWR("FALSE"),
   .IOSTANDARD("DEFAULT")
    )
IBUFDS_inst (
   .O(clk0),
   .I(CLOCK_P),
   .IB(CLOCK_N)
);


wire clk1, clk2;


BUFR #(
    .BUFR_DIVIDE("4"),
    .SIM_DEVICE("7SERIES")
    )
BUFR_clk1 (
```

```
      .O(clk1),
      .CE(1'b1),
      .CLR(RESET),
      .I(clk0)
);


BUFR #(
    .BUFR_DIVIDE("3"),
    .SIM_DEVICE("7SERIES")
    )
BUFR_clk21 (
      .O(clk2),
      .CE(1'b1),
      .CLR(RESET),
      .I(clk1)
);


BUFR #(
    .BUFR_DIVIDE("8"),
    .SIM_DEVICE("7SERIES")
    )
BUFR_clk3 (
      .O(CLOCK_UART),
      .CE(1'b1),
      .CLR(RESET),
      .I(clk2)
);


BUFR #(
    .BUFR_DIVIDE("5"),
    .SIM_DEVICE("7SERIES")
    )
BUFR_clk2 (
      .O(CLOCK_MAIN),
      .CE(1'b1),
      .CLR(RESET),
      .I(clk1)
);
```

```
endmodule
```

Appendix 5: Interrupt Unit

```
`timescale 1ns/1ps

module ENABLER (
    input IN,
    input RAM, TIMER,
    output OUT
);

assign OUT = (IN & ~RAM & ~TIMER);

endmodule
```

Appendix 6: UART

```
`timescale 1ns / 1ps

module UART(
    input CLOCK,
    input ENABLE,
    input RESET,
    input RX_UART,
    output [7:0] DATA_OUT,
    output TICK
    );



wire reset;
wire enable;



// negedge detector reg
reg busy = 0;
reg prev = 0;

always @(posedge CLOCK)
```

```
    begin
        if (reset)
            busy <= 0;
        else if (enable)
            busy <= (!RX_UART && prev);
            prev <= RX_UART;
    end



assign enable = (~ENABLE & ~busy);



// count reg
wire c1;
reg [2:0] count = 0;

always @(posedge CLOCK)
    begin
        if (reset)
            count <= 0;
        else if (c1)
            count <= count + 1;
    end



wire b1;
reg end1 = 0;

assign b1 = (count == 3'b111);
assign c1 = (~b1 & ~end1 & ~RX_UART & busy);



// first count reg
always @(posedge CLOCK)
    begin
        if (reset)
            end1 <= 0;
        else if (b1)
```

```
            end1 <= 1;
    end



// count 2 reg
reg [3:0] count2 = 0;

always @(posedge CLOCK)
    begin
        if (reset)
            count2 <= 0;
        else if (end1)
            count2 <= count2 + 1;
    end



wire sample;
assign sample = (count2 == 4'b1111);



// byte count
reg [4:0] countB = 0;

always @(posedge CLOCK)
    begin
        if (reset)
            countB <= 0;
        else if (sample)
            countB <= countB + 1;
    end



wire ran;
assign ran = (countB > 7);



//  bit7 reg
```

```
reg bit7 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit7 <= 0;
        else if (sample)
            bit7 <= RX_UART;
    end



// bit6 reg
reg bit6 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit6 <= 0;
        else if (sample)
            bit6 <= bit7;
    end


// bit5 reg
reg bit5 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit5 <= 0;
        else if (sample)
            bit5 <= bit6;
    end


// bit4 reg
reg bit4 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
```

```
            bit4 <= 0;
        else if (sample)
            bit4 <= bit5;
    end


// bit3 reg
reg bit3 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit3 <= 0;
        else if (sample)
            bit3 <= bit4;
    end


// bit2 reg
reg bit2 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit2 <= 0;
        else if (sample)
            bit2 <= bit3;
    end


// bit1 reg
reg bit1 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit1 <= 0;
        else if (sample)
            bit1 <= bit2;
    end
```

```
// bit0 reg
reg bit0 = 0;

always @(posedge CLOCK)
    begin
        if (RESET)
            bit0 <= 0;
        else if (sample)
            bit0 <= bit1;
    end


wire ending;
assign ending = ((count2 == 4'b1110) & ran );

assign reset = (ending | RESET);

assign DATA_OUT = {bit7,bit6,bit5,bit4,bit3,bit2,bit1,bit0};


assign TICK = ending;

endmodule
```

Appendix 7: Bootloader

```
`timescale 1ns / 1ps

module BYTEVAULT(
    input CLOCK_IN,
    input RESET,
    input [7:0] DATA_IN,
    input READ,
    output CLOCK_OUT,
    output [7:0] ADDRESS,
    output [31:0] DATA_OUT,
    output WRITE
    );
```

```
assign CLOCK_OUT = CLOCK_IN;



// byte3 reg
reg [7:0] byte3 = 0;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            byte3 <= 0;
        else if (READ)
            byte3 <= DATA_IN;
    end



// byte2 reg
reg [7:0] byte2 = 0;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            byte2 <= 0;
        else if (READ)
            byte2 <= byte3;
    end



// byte1 reg
reg [7:0] byte1 = 0;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            byte1 <= 0;
        else if (READ)
            byte1 <= byte2;
    end
```

```
// byte0 reg
reg [7:0] byte0 = 0;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            byte0 <= 0;
        else if (READ)
            byte0 <= byte1;
    end




// bytes count
reg [1:0] count = 2'b00;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            count <= 0;
        else if (READ)
            count <= count + 1;
    end


wire check2;

wire check;
assign check = ( (count == 2'b11) & READ );


// address reg
reg [7:0] count2 = 0;

always @(posedge CLOCK_IN)
    begin
        if (RESET)
            count2 <= 0;
        else if (check2)
            count2 <= count2 + 1;
```

```
    end



wire reset;
// delay on reg
reg don = 0;

always @(posedge CLOCK_IN)
    begin
        if (reset)
            don <= 0;
        else if (check)
            don <= 1;
    end



// delay reg
reg [7:0] count3 = 0;

always @(posedge CLOCK_IN)
    begin
        if (reset)
            count3 <= 0;
        else if (don)
            count3 <= count3 + 1;
    end


assign check2 = (count3 == 8'hbb);

assign reset = (RESET | check2);

assign ADDRESS = count2;
assign DATA_OUT = {byte3,byte2,byte1,byte0};
assign WRITE = check2;


endmodule
```

Appendix 8: Program Memory

```
`timescale 1ns / 1ps

module MEMORY(
  input wire CLK_WRITE,
  input wire [7:0] ADDRESS_WRITE,
  input wire [31:0] DATA_WRITE,
  input wire WRITE_ENABLE,

  input wire CLK_READ,
  input wire [7:0] ADDRESS_READ,
  input wire READ_ENABLE,
  output reg [31:0] DATA_READ
    );

  reg [31:0] memory [0:255];


  always @(posedge CLK_WRITE)
    if (WRITE_ENABLE)
      memory[ADDRESS_WRITE] <= DATA_WRITE;



  always @(negedge CLK_READ)
    if (READ_ENABLE)
      DATA_READ <= memory[ADDRESS_READ];


endmodule
```

Appendix 9: Control Unit

```
`timescale 1ns/1ps

module CTRL_UNIT (
    input [2:0] FUNCT3,
    input [6:0] OPCODE, FUNCT7,
    output J, R, B, REG_WR, MEM_WR, MEM_RD,
    output [1:0] Ctrl_A, Ctrl_B, Ctrl_S,
```

```
    output [2:0] Ctrl_EXT, Ctrl_MDU, Ctrl_BRANCH, Ctrl_MEM,
    output [3:0] Ctrl_ALU
);

DecoMAIN MAIN(
    .OPCODE(OPCODE),
    .FUNCT7(FUNCT7),
    .J(J),
    .R(R),
    .B(B),
  .MEM_RD(MEM_RD),
    .MEM_WR(MEM_WR),
    .REG_WR(REG_WR),
    .Ctrl_A(Ctrl_A),
    .Ctrl_B(Ctrl_B),
    .Ctrl_S(Ctrl_S)
);

DecoALU ALU(
    .FUNCT3(FUNCT3),
    .OPCODE(OPCODE),
    .FUNCT7(FUNCT7),
    .Ctrl_ALU(Ctrl_ALU)
);

DecoEXT SORTEXT(
    .FUNCT3(FUNCT3),
    .OPCODE(OPCODE),
    .Ctrl_EXT(Ctrl_EXT)
);

DecoMDU MDU(
    .FUNCT3(FUNCT3),
    .OPCODE(OPCODE),
    .FUNCT7(FUNCT7),
    .Ctrl_MDU(Ctrl_MDU)
);

DecoBRANCH BRANCH(
    .FUNCT3(FUNCT3),
    .OPCODE(OPCODE),
    .Ctrl_Branch(Ctrl_BRANCH)
```

```
);

DecoMEM MEMORY(
    .FUNCT3(FUNCT3),
    .OPCODE(OPCODE),
    .Ctrl_MEM(Ctrl_MEM)
);

endmodule
```

Appendix 10: Main Decoder

```
`timescale 1ns/1ps

module DecoMAIN (
    input [6:0] OPCODE, FUNCT7,
    output J, R, B, MEM_WR, REG_WR, MEM_RD,
    output [1:0] Ctrl_A, Ctrl_B, Ctrl_S
);

reg [11:0] OUT;

always @(*)
    begin : encode
    casex ({FUNCT7,OPCODE})
    14'b0X00000_0110011:  OUT <= 12'b0_0_0_0_0_1_00_00_00;
    14'bXXXXXXX_0010011:  OUT <= 12'b0_0_0_0_0_1_00_01_00;
    14'bXXXXXXX_1100011:  OUT <= 12'b0_0_1_0_0_0_00_00_00;
    14'bXXXXXXX_0000011:  OUT <= 12'b0_0_0_1_0_1_00_01_10;
    14'bXXXXXXX_0100011:  OUT <= 12'b0_0_0_0_1_0_00_01_00;
    14'bXXXXXXX_0110111:  OUT <= 12'b0_0_0_0_0_1_00_00_01;
    14'bXXXXXXX_0010111:  OUT <= 12'b0_0_0_0_0_1_01_01_00;
    14'bXXXXXXX_1101111:  OUT <= 12'b1_0_0_0_0_1_01_10_00;
    14'bXXXXXXX_1100111:  OUT <= 12'b1_1_0_0_0_1_01_10_00;
    14'b0000001_0110011:  OUT <= 12'b0_0_0_0_0_1_00_00_11;
    default:                OUT <= 12'b0_0_0_0_0_0_00_00_00;
  endcase
  end

assign  Ctrl_S = OUT[1:0];
assign  Ctrl_B = OUT[3:2];
```

```
assign  Ctrl_A = OUT[5:4];
assign  REG_WR = OUT[6];
assign  MEM_WR = OUT[7];
assign  MEM_RD = OUT[8];
assign  B = OUT[9];
assign  R = OUT[10];
assign  J = OUT[11];

endmodule
```

Appendix 11: Sort & Extend Decoder

```
`timescale 1ns/1ps

module DecoEXT (
    input  [2:0] FUNCT3,
    input  [6:0] OPCODE,
    output reg [2:0] Ctrl_EXT
);

always @(*)
    begin : encode
    casex ({FUNCT3,OPCODE})
    10'b011_0010011:  Ctrl_EXT <= 3'b101;
    10'bXXX_0010011:  Ctrl_EXT <= 3'b001;
    10'b11X_1100011:  Ctrl_EXT <= 3'b110;
    10'bXXX_1100011:  Ctrl_EXT <= 3'b010;
    10'b10X_0000011:  Ctrl_EXT <= 3'b101;
    10'bXXX_0000011:  Ctrl_EXT <= 3'b001;
    10'bXXX_0100011:  Ctrl_EXT <= 3'b011;
    10'bXXX_0110111:  Ctrl_EXT <= 3'b000;
    10'bXXX_0010111:  Ctrl_EXT <= 3'b000;
    10'bXXX_1101111:  Ctrl_EXT <= 3'b100;
    10'b000_1100111:  Ctrl_EXT <= 3'b001;
    default:              Ctrl_EXT <= 3'b101;
    endcase
    end

endmodule
```

Appendix 12: ALU Decoder

```
`timescale 1ns/1ps

module DecoALU (
    input [2:0] FUNCT3,
    input [6:0] OPCODE, FUNCT7,
    output reg [3:0] Ctrl_ALU
);

always @(*)
    begin : encode
    casex ({FUNCT3,FUNCT7,OPCODE})
    17'bXXX_0000000_0110011:  Ctrl_ALU <= {1'b0,FUNCT3};
    17'bXXX_0100000_0110011:  Ctrl_ALU <= {1'b1,FUNCT3};
    17'b101_0100000_0010011:  Ctrl_ALU <= {1'b1,FUNCT3};
    17'bXXX_XXXXXXX_0010011:  Ctrl_ALU <= {1'b0,FUNCT3};
    default:            Ctrl_ALU <= 4'b0000;
    endcase
    end

endmodule
```

Appendix 13: Branch Decoder

```
`timescale 1ns/1ps

module DecoBRANCH (
    input [2:0] FUNCT3,
    input [6:0] OPCODE,
    output reg [2:0] Ctrl_Branch
);

always @(*)
    begin : encode
    case (OPCODE)
    7'b1100011: Ctrl_Branch <= FUNCT3;
    default:  Ctrl_Branch <= 3'b010;
    endcase
    end
```

```
endmodule
```

Appendix 14: Memory Decoder

```
`timescale 1ns/1ps

module DecoMEM (
    input [6:0] OPCODE,
    input [2:0] FUNCT3,
    output reg [2:0] Ctrl_MEM
);

always @(*)
    begin : encode
    case (OPCODE)
    7'b0000011: Ctrl_MEM <= FUNCT3;
    7'b0100011: Ctrl_MEM <= FUNCT3;
    default:    Ctrl_MEM <= 3'b111;
    endcase
    end

endmodule
```

Appendix 15: MDU Decoder

```
`timescale 1ns/1ps

module DecoMDU (
    input [2:0] FUNCT3,
    input [6:0] OPCODE, FUNCT7,
    output reg [2:0] Ctrl_MDU
);

always @(*)
    begin : encode
    case ({FUNCT7,OPCODE})
    14'b0000001_0110011:  Ctrl_MDU <= FUNCT3;
    default:        Ctrl_MDU <= 3'b000;
    endcase
    end
```

```
endmodule
```

Appendix 16: Datapath

```
`timescale 1ns/1ps

module DATAPATH (
    input CLOCK, EN_PC, RESET, REG_WR, B, J, R,
    input [1:0] Ctrl_A, Ctrl_B, Ctrl_S,
    input [2:0] Ctrl_EXT, Ctrl_Branch, Ctrl_Mult,
    input [3:0] Ctrl_ALU,
    input [24:0] INSTR,
    input [31:0] DO_RAM,
    output [31:0] ADDR_INSTR, DI_RAM, ADDR_RAM
);

wire flag;
wire [4:0] rs1, rs2, rd;
wire [24:0] imm;
wire [31:0] pc, ext, do1, do2, dsave, sa, sb, dalu, dprod, dload;

assign rs1 = INSTR[12:8];
assign rs2 = INSTR[17:13];
assign rd  = INSTR[4:0];
assign imm = INSTR;
assign dload = DO_RAM;
assign ADDR_RAM = dalu;
assign ADDR_INSTR = pc;
assign DI_RAM = do2;

PC PC(
    .CLOCK(CLOCK),
    .ENABLE(EN_PC),
    .RESET(RESET),
    .B(B),
    .J(J),
    .R(R),
    .FLAG(flag),
    .OFFSET(ext),
    .RP(do1),
    .ADDR(pc)
```

```
);

REG_FILE REG_FILE(
    .CLOCK(CLOCK),
    .ENABLE(REG_WR),
    .RESET(RESET),
    .RS1(rs1),
    .RS2(rs2),
    .RD(rd),
    .DI(dsave),
    .DO1(do1),
    .DO2(do2)
);

SORTEXT SORTEXT(
    .IN(imm),
    .CTRL(Ctrl_EXT),
    .OUT(ext)
);

MUX4_1 MUXA(
    .I0(do1),
    .I1(pc),
    .I2(32'd0),
    .I3(32'd0),
    .CTRL(Ctrl_A),
    .OUT(sa)
);

MUX4_1 MUXB(
    .I0(do2),
    .I1(ext),
    .I2(32'd4),
    .I3(32'd0),
    .CTRL(Ctrl_B),
    .OUT(sb)
);

ALU ALU(
    .A(sa),
    .B(sb),
    .CTRL(Ctrl_ALU),
```

```
    .OUT(dalu)
);

BRANCH BRANCH(
    .A(do1),
    .B(do2),
    .CTRL(Ctrl_Branch),
    .OUT(flag)
);

MDU MDU(
    .A(do1),
    .B(do2),
    .CTRL(Ctrl_Mult),
    .OUT(dprod)
);

MUX4_1 MUXS(
    .I0(dalu),
    .I1(ext),
    .I2(dload),
    .I3(dprod),
    .CTRL(Ctrl_S),
    .OUT(dsave)
);

endmodule
```

Appendix 17: Program Counter

```
`timescale 1ns/1ps

module PC (
    input CLOCK, ENABLE, RESET, B, J, R, FLAG,
    input [31:0] OFFSET, RP,
    output [31:0] ADDR
);

wire jump;
wire [31:0] a1, a2;
wire [31:0] next;
```

```
REG REG(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(ENABLE),
    .DD(next),
    .QQ(ADDR)
);

PC_SUM SUM(
    .IN1(a1),
    .IN2(a2),
    .OUT(next)
);

PC_MUX2_1 MUX_A(
    .I0(32'd4),
    .I1(OFFSET),
    .CTRL(jump),
    .OUT(a1)
);

PC_MUX2_1 MUX_B(
    .I0(ADDR),
    .I1(RP),
    .CTRL(R),
    .OUT(a2)
);

assign jump = J | (FLAG & B);

endmodule
```

Appendix 18: Register

```
`timescale 1ns/1ps

module REG (
    input CLOCK, ENABLE, RESET,
    input  [31:0] DD,
    output [31:0] QQ
```

```
);

reg [31:0] q_int = 32'd0;

// D-type flip flop
always@(posedge CLOCK)
    begin
        if (RESET)
            q_int <= 32'd0;
        else if (ENABLE)
            q_int <= DD;
    end

assign QQ = q_int;

endmodule
```

Appendix 19: Adder

```
`timescale 1ns/1ps

module PC_SUM (
    input [31:0] IN1, IN2,
    output [31:0] OUT
);

assign OUT = IN1 + IN2;

endmodule
```

Appendix 20: 2:1 Multiplexer

```
`timescale 1ns/1ps

module PC_MUX2_1 (
    input [31:0] I0, I1,
    input CTRL,
    output reg [31:0] OUT
);

always @(*)
```

```
    case (CTRL)
    1'b0: OUT <= I0;
    1'b1: OUT <= I1;
    default:  OUT <= I0;
    endcase

endmodule
```

Appendix 21: Register File

```
`timescale 1ns/1ps

module REG_FILE (
    input CLOCK, ENABLE, RESET,
    input [4:0] RS1, RS2, RD,
    input [31:0] DI,
    output [31:0] DO1, DO2
);

wire [31:0] q0, q1, q2, q3, q4, q5, q6, q7;
wire [31:0] q8, q9, q10, q11, q12, q13, q14, q15;
wire [31:0] q16, q17, q18, q19, q20, q21, q22, q23;
wire [31:0] q24, q25, q26, q27, q28, q29, q30, q31;
wire [31:0] e;

RF_EN DM(
    .IN(ENABLE),
    .CTRL(RD),
    .OUT(e)
);

REG X0(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[0]),
    .DD(32'd0),
    .QQ(q0)
);

REG X1(
    .CLOCK(CLOCK),
```

```
    .RESET(RESET),
    .ENABLE(e[1]),
    .DD(DI),
    .QQ(q1)
);

REG X2(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[2]),
    .DD(DI),
    .QQ(q2)
);

REG X3(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[3]),
    .DD(DI),
    .QQ(q3)
);

REG X4(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[4]),
    .DD(DI),
    .QQ(q4)
);

REG X5(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[5]),
    .DD(DI),
    .QQ(q5)
);

REG X6(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[6]),
```

```
    .DD(DI),
    .QQ(q6)
);

REG X7(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[7]),
    .DD(DI),
    .QQ(q7)
);

REG X8(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[8]),
    .DD(DI),
    .QQ(q8)
);

REG X9(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[9]),
    .DD(DI),
    .QQ(q9)
);

REG X10(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[10]),
    .DD(DI),
    .QQ(q10)
);

REG X11(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[11]),
    .DD(DI),
    .QQ(q11)
```

```
);

REG X12(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[12]),
    .DD(DI),
    .QQ(q12)
);

REG X13(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[13]),
    .DD(DI),
    .QQ(q13)
);

REG X14(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[14]),
    .DD(DI),
    .QQ(q14)
);

REG X15(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[15]),
    .DD(DI),
    .QQ(q15)
);

REG X16(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[16]),
    .DD(DI),
    .QQ(q16)
);
```

```
REG X17(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[17]),
    .DD(DI),
    .QQ(q17)
);

REG X18(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[18]),
    .DD(DI),
    .QQ(q18)
);

REG X19(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[19]),
    .DD(DI),
    .QQ(q19)
);

REG X20(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[20]),
    .DD(DI),
    .QQ(q20)
);

REG X21(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[21]),
    .DD(DI),
    .QQ(q21)
);

REG X22(
    .CLOCK(CLOCK),
```

```
    .RESET(RESET),
    .ENABLE(e[22]),
    .DD(DI),
    .QQ(q22)
);

REG X23(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[23]),
    .DD(DI),
    .QQ(q23)
);

REG X24(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[24]),
    .DD(DI),
    .QQ(q24)
);

REG X25(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[25]),
    .DD(DI),
    .QQ(q25)
);

REG X26(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[26]),
    .DD(DI),
    .QQ(q26)
);

REG X27(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[27]),
```

```
    .DD(DI),
    .QQ(q27)
);

REG X28(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[28]),
    .DD(DI),
    .QQ(q28)
);

REG X29(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[29]),
    .DD(DI),
    .QQ(q29)
);

REG X30(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[30]),
    .DD(DI),
    .QQ(q30)
);

REG X31(
    .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[31]),
    .DD(DI),
    .QQ(q31)
);

RF_MUX M1(
    .CTRL(RS1),
    .I0(q0),
    .I1(q1),
    .I2(q2),
    .I3(q3),
```

```
    .I4(q4),
    .I5(q5),
    .I6(q6),
    .I7(q7),
    .I8(q8),
    .I9(q9),
    .I10(q10),
    .I11(q11),
    .I12(q12),
    .I13(q13),
    .I14(q14),
    .I15(q15),
    .I16(q16),
    .I17(q17),
    .I18(q18),
    .I19(q19),
    .I20(q20),
    .I21(q21),
    .I22(q22),
    .I23(q23),
    .I24(q24),
    .I25(q25),
    .I26(q26),
    .I27(q27),
    .I28(q28),
    .I29(q29),
    .I30(q30),
    .I31(q31),
    .OUT(DO1)
);

RF_MUX M2(
    .CTRL(RS2),
    .I0(q0),
    .I1(q1),
    .I2(q2),
    .I3(q3),
    .I4(q4),
    .I5(q5),
    .I6(q6),
    .I7(q7),
    .I8(q8),
```

```
    .I9(q9),
    .I10(q10),
    .I11(q11),
    .I12(q12),
    .I13(q13),
    .I14(q14),
    .I15(q15),
    .I16(q16),
    .I17(q17),
    .I18(q18),
    .I19(q19),
    .I20(q20),
    .I21(q21),
    .I22(q22),
    .I23(q23),
    .I24(q24),
    .I25(q25),
    .I26(q26),
    .I27(q27),
    .I28(q28),
    .I29(q29),
    .I30(q30),
    .I31(q31),
    .OUT(DO2)
);

endmodule
```

Appendix 22: Enable Register Generator

```
`timescale 1ns/1ps

module RF_EN (
    input IN,
    input [4:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
    case (CTRL)
        5'd0: OUT <= {31'd0, IN};
```

```
        5'd1: OUT <= {30'd0, IN, 1'd0};
        5'd2: OUT <= {29'd0, IN, 2'd0};
        5'd3: OUT <= {28'd0, IN, 3'd0};
        5'd4: OUT <= {27'd0, IN, 4'd0};
        5'd5: OUT <= {26'd0, IN, 5'd0};
        5'd6: OUT <= {25'd0, IN, 6'd0};
        5'd7: OUT <= {24'd0, IN, 7'd0};
        5'd8: OUT <= {23'd0, IN, 8'd0};
        5'd9: OUT <= {22'd0, IN, 9'd0};
        5'd10:  OUT <= {21'd0, IN, 10'd0};
        5'd11:  OUT <= {20'd0, IN, 11'd0};
        5'd12:  OUT <= {19'd0, IN, 12'd0};
        5'd13:  OUT <= {18'd0, IN, 13'd0};
        5'd14:  OUT <= {17'd0, IN, 14'd0};
        5'd15:  OUT <= {16'd0, IN, 15'd0};
        5'd16:  OUT <= {15'd0, IN, 16'd0};
        5'd17:  OUT <= {14'd0, IN, 17'd0};
        5'd18:  OUT <= {13'd0, IN, 18'd0};
        5'd19:  OUT <= {12'd0, IN, 19'd0};
        5'd20:  OUT <= {11'd0, IN, 20'd0};
        5'd21:  OUT <= {10'd0, IN, 21'd0};
        5'd22:  OUT <= {9'd0, IN, 22'd0};
        5'd23:  OUT <= {8'd0, IN, 23'd0};
        5'd24:  OUT <= {7'd0, IN, 24'd0};
        5'd25:  OUT <= {6'd0, IN, 25'd0};
        5'd26:  OUT <= {5'd0, IN, 26'd0};
        5'd27:  OUT <= {4'd0, IN, 27'd0};
        5'd28:  OUT <= {3'd0, IN, 28'd0};
        5'd29:  OUT <= {2'd0, IN, 29'd0};
        5'd30:  OUT <= {1'd0, IN, 30'd0};
        5'd31:  OUT <= {IN, 31'd0};
        default: OUT <= 32'dZ;
    endcase

endmodule
```

Appendix 23: Register File Multiplexer

```
`timescale 1ns/1ps

module RF_MUX (
```

```
    input [31:0] I0, I1, I2, I3, I4, I5, I6, I7,
    input [31:0] I8, I9, I10, I11, I12, I13, I14, I15,
    input [31:0] I16, I17, I18, I19, I20, I21, I22, I23,
    input [31:0] I24, I25, I26, I27, I28, I29, I30, I31,
    input [4:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
    case (CTRL)
        5'd0: OUT <= I0;
        5'd1: OUT <= I1;
        5'd2: OUT <= I2;
        5'd3: OUT <= I3;
        5'd4: OUT <= I4;
        5'd5: OUT <= I5;
        5'd6: OUT <= I6;
        5'd7: OUT <= I7;
        5'd8: OUT <= I8;
        5'd9: OUT <= I9;
        5'd10:  OUT <= I10;
        5'd11:  OUT <= I11;
        5'd12:  OUT <= I12;
        5'd13:  OUT <= I13;
        5'd14:  OUT <= I14;
        5'd15:  OUT <= I15;
        5'd16:  OUT <= I16;
        5'd17:  OUT <= I17;
        5'd18:  OUT <= I18;
        5'd19:  OUT <= I19;
        5'd20:  OUT <= I20;
        5'd21:  OUT <= I21;
        5'd22:  OUT <= I22;
        5'd23:  OUT <= I23;
        5'd24:  OUT <= I24;
        5'd25:  OUT <= I25;
        5'd26:  OUT <= I26;
        5'd27:  OUT <= I27;
        5'd28:  OUT <= I28;
        5'd29:  OUT <= I29;
        5'd30:  OUT <= I30;
        5'd31:  OUT <= I31;
```

```
        default: OUT <= 32'dZ;
    endcase

endmodule
```

Appendix 24: Sort & Extend Unit

```
`timescale 1ns/1ps

module SORTEXT (
    input [24:0] IN,
    input [2:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
    case (CTRL)
        3'b000:   OUT <= { IN[24:5], 12'd0 };
        3'b001:   OUT <= { {20{IN[24]}}, IN[24:13] };
        3'b010:   OUT <= { {20{IN[24]}}, IN[0], IN[23:18], IN[4:1], 1'b0 };
        3'b011:   OUT <= { {20{IN[24]}}, IN[24:18], IN[4:0] };
        3'b100:   OUT <= { {12{IN[24]}}, IN[12:5], IN[13], IN[23:14], 1'b0};
        3'b101:   OUT <= { {20'd0}, IN[24:13] };
        3'b110:   OUT <= { {20'd0}, IN[0], IN[23:18], IN[4:1], 1'b0 };
        default:  OUT <= 32'd0;
    endcase

endmodule
```

Appendix 25: ALU

```
`timescale 1ns/1ps

module ALU (
    input signed [31:0] A,
    input signed [31:0] B,
    input [3:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
```

```
    case (CTRL)
        4'b0_000: OUT <= A + B;
        4'b1_000: OUT <= A - B;
        4'b0_111: OUT <= A & B;
        4'b0_110: OUT <= A | B;
        4'b0_100: OUT <= A ^ B;
        4'b0_010: OUT <= {31'd0 ,A < B};
        4'b0_011: OUT <= {31'd0 ,$unsigned(A) < $unsigned(B)};
        4'b0_001: OUT <= A << B[4:0];
        4'b0_101: OUT <= A >> B[4:0];
        4'b1_101: OUT <= A >>> B[4:0];
        default:  OUT <= A + B;
    endcase

endmodule
```

Appendix 26: Branch Unit

```
`timescale 1ns/1ps

module BRANCH (
    input signed [31:0] A,
    input signed [31:0] B,
    input [2:0] CTRL,
    output reg OUT
);

always @(*)
    case (CTRL)
        3'b000: OUT <= A == B;
        3'b001: OUT <= A != B;
        3'b100: OUT <= A < B;
        3'b101: OUT <= A >= B;
        3'b110: OUT <= $unsigned(A) < $unsigned(B);
        3'b111: OUT <= $unsigned(A) >= $unsigned(B);
        default:OUT <= 1'b0;
    endcase

endmodule
```

Appendix 27: MDU

```
`timescale 1ns/1ps

module MDU (
    input signed [31:0] A,
    input signed [31:0] B,
    input [2:0] CTRL,
    output reg signed [31:0] OUT
);

wire [31:0] D, Du, R, Ru;
wire [63:0] P, Psu, Pu;

M_MULT MULT(
    .A(A),
    .B(B),
    .OUT(P)
);

M_MULTSU MULTSU(
    .A(A),
    .B(B),
    .OUT(Psu)
);

M_MULTU MULTU(
    .A(A),
    .B(B),
    .OUT(Pu)
);

M_DIV DIV(
    .A(A),
    .B(B),
    .OUT(D)
);

M_DIVU DIVU(
    .A(A),
    .B(B),
```

```
    .OUT(Du)
);

M_REM REM(
    .A(A),
    .B(B),
    .OUT(R)
);

M_REMU REMU(
    .A(A),
    .B(B),
    .OUT(Ru)
);

always @(*)
    case (CTRL)
        3'b000: OUT <= P[31:0];
        3'b001: OUT <= P[63:32];
        3'b010: OUT <= Psu[63:32];
        3'b011: OUT <= Pu[63:32];
        3'b100: OUT <= D;
        3'b101: OUT <= Du;
        3'b110: OUT <= R;
        3'b111: OUT <= Ru;
        default:OUT <= 32'd0;
    endcase

endmodule
```

Appendix 28: 4:1 Multiplexer

```
`timescale 1ns/1ps

module MUX4_1 (
    input [31:0] I0, I1, I2, I3,
    input [1:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
```

```
    case (CTRL)
        2'd0:   OUT <= I0;
        2'd1:   OUT <= I1;
        2'd2:   OUT <= I2;
        2'd3:   OUT <= I3;
        default:OUT <= 32'd0;
    endcase

endmodule
```

Appendix 29: Memory Controller

```
`timescale 1ns/1ps

module RAM_CONTROLLER (
    input CLOCK, RESET, RE_P, WE_P,
    input [2:0] Ctrl_MEM,
    input [7:0] DI_M,
    input [31:0] ADDR_P,
  input [31:0] DI_P,
  output BUSY,
  output CLOCK_M, WE_M,
  output [7:0] DO_M,
    output [31:0] DO_P,
  output [8:0] ADDR_M
);

assign CLOCK_M = CLOCK;

wire enable, check;
wire [1:0] ctrl, count;
wire [2:0] vf;
wire [3:0] e;
wire [7:0] do_m, di_m;
wire [31:0] do_p;

wire [31:0] do_r;


assign enable = check & (WE_P | RE_P);
```

```
RAM_COUNTER COUNTER(
  .CLOCK(CLOCK),
  .ENABLE(enable),
  .RESET(RESET),
  .ENDING(vf),
  .BUSY(BUSY),
  .COUNT(count)
);

assign ctrl = Ctrl_MEM[1:0];

RAM_VF FINAL(
  .CTRL(ctrl),
  .RE(RE_P),
  .OUT(vf)
);


RAM_MUX #(.nb(8)) MUX_DATA(
  .I0(DI_P[7:0]),
  .I1(DI_P[15:8]),
  .I2(DI_P[23:16]),
  .I3(DI_P[31:24]),
  .CTRL(count),
  .OUT(do_m)
);


assign DO_M = do_m;


assign ADDR_M = ( ADDR_P[8:0] + count);

assign check = (ADDR_P <= 512 );
assign WE_M = (check & WE_P);


RAM_E ENABLES(
  .CTRL(count),
  .E(e)
);
```

```
assign di_m = DI_M;


RAM_REG REG0(
  .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[0]),
    .DD(di_m),
    .QQ(do_r[7:0])
);

assign do_p[7:0] = e[0] ? di_m : do_r[7:0];

RAM_REG REG1(
  .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[1]),
    .DD(di_m),
    .QQ(do_r[15:8])
);

assign do_p[15:8] = e[1] ? di_m : do_r[15:8];

RAM_REG REG2(
  .CLOCK(CLOCK),
    .RESET(RESET),
    .ENABLE(e[2]),
    .DD(di_m),
    .QQ(do_r[23:16])
);

assign do_p[23:16] = e[2] ? di_m : do_r[23:16];


assign do_p[31:24] = di_m;


RAM_LOAD LOAD(
  .IN(do_p),
  .CTRL(Ctrl_MEM),
  .OUT(DO_P)
```

```
);


endmodule
```

Appendix 30: Counter

```
`timescale 1ns/1ps

module RAM_COUNTER (
  input CLOCK, ENABLE, RESET,
  input [2:0] ENDING,
  output BUSY,
  output [1:0] COUNT
);

wire free;
wire comp;
wire reset;


reg [2:0] q = 0;
reg busy = 0;


//  busy register
always @(posedge CLOCK)
    begin
        if (reset)
            busy <= 0;
        else if (ENABLE)
            busy <= 1;
    end

assign BUSY = busy;


//  count register
always @(posedge CLOCK)
    begin
        if (reset)
```

```
            q <= 0;
        else if (ENABLE)
            q <= q + 1;
    end

assign COUNT = q;


//    free
assign comp = (q >= ENDING);
assign free = (ENABLE & comp);

assign reset = (free | RESET);

endmodule
```

Appendix 31: Final Value

```
`timescale 1ns/1ps

module RAM_VF (
    input [1:0] CTRL,
    input RE,
    output reg [2:0] OUT
);

always @(*)
    case ({RE,CTRL})
        3'b000:  OUT <= 3'd0;
        3'b001:  OUT <= 3'd1;
        3'b010:  OUT <= 3'd3;
        3'b100:  OUT <= 3'd1;
        3'b101:  OUT <= 3'd2;
        3'b110:  OUT <= 3'd4;
        default:OUT <= 3'd0;
    endcase

endmodule
```

Appendix 32: Memory Multiplexer

```
`timescale 1ns/1ps

module RAM_MUX #(parameter nb=0) (
    input [nb-1:0] I0, I1, I2, I3,
    input [1:0] CTRL,
    output reg [nb-1:0] OUT
);

always @(*)
    case (CTRL)
        2'd0:   OUT <= I0;
        2'd1:   OUT <= I1;
        2'd2:   OUT <= I2;
        2'd3:   OUT <= I3;
        default:OUT <= 0;
    endcase

endmodule
```

Appendix 33: Memory Enables

```
`timescale 1ns/1ps

module RAM_E (
    input [1:0] CTRL,
    output reg [3:0] E
);

always @(*)
    case (CTRL)
        2'b00:  E <= 4'b0000;
        2'b01:  E <= 4'b0001;
        2'b10:  E <= 4'b0010;
    2'b11:  E <= 4'b0100;
        default:E <= 4'b0000;
    endcase

endmodule
```

Appendix 34: Memory Register

```
`timescale 1ns/1ps

module RAM_REG (
    input CLOCK, ENABLE, RESET,
    input  [7:0] DD,
    output [7:0] QQ
);

reg [7:0] q_int = 0;

// D-type flip flop
always@(posedge CLOCK)
    begin
        if (RESET)
            q_int <= 0;
        else if (ENABLE)
            q_int <= DD;
    end

assign QQ = q_int;

endmodule
```

Appendix 35: Data Loader

```
`timescale 1ns/1ps

module RAM_LOAD (
    input [31:0] IN,
    input [2:0] CTRL,
    output reg [31:0] OUT
);

always @(*)
    case (CTRL)
        3'b000: OUT <= { {24{IN[7]}}, IN[7:0] };
        3'b001: OUT <= { {16{IN[15]}}, IN[15:0] };
        3'b010: OUT <= { IN };
        3'b100: OUT <= { 24'b0, IN[7:0] };
        3'b101: OUT <= { 16'b0, IN[15:0] };
        default:OUT <= { IN };
```

```
  endcase

endmodule
```

Appendix 36: Data Memory

```
`timescale 1ns/1ps

module RAM (
    input CLOCK,
    input WE,
    input [8:0] ADDR,
    input [7:0] DI,
    output reg [7:0] DO
);


reg [7:0] memory [0:511];


always@(posedge CLOCK)
    if (WE)
        memory[ADDR] <= DI;
    else
        DO <= memory[ADDR];


endmodule
```

Appendix 37: Leds Controller

```
`timescale 1ns/1ps

module LEDS (
    input CLOCK, WE, RESET,
    input [7:0] DI,
    input [31:0] ADDR,
    output [7:0] LED
);

wire enable, check;
```

```
wire [7:0] led;

assign check = (ADDR == 32'd1028);
assign enable = (check & WE);

reg [7:0] q_int = 0;

// D-type flip flop
always@(posedge CLOCK)
  begin
    if (RESET)
        q_int <= 0;
    else if (enable)
        q_int <= DI;
  end

assign led = q_int;

assign LED = led;

endmodule
```

Appendix 38: LCD Controller

```
`timescale 1ns/1ps

module LCD (
    input CLOCK, WE, RESET,
    input [5:0] DI,
    input [31:0] ADDR,
    output E, RS, RW, D4, D5, D6, D7
);


wire enable, check;
wire [5:0] lcd;

wire [3:0] HEXA;

wire e, block;
wire [3:0] dato;
```

```
wire [5:0] out;

wire enable_df;

assign check = (ADDR == 32'd1032);
assign enable = (check & WE);


reg [5:0] q_int = 0;

// D-type flip flop
always@(posedge CLOCK)
  begin
    if (RESET)
        q_int <= 0;
    else if (enable)
        q_int <= DI;
  end

assign lcd = q_int;


assign HEXA = out[3:0];


assign E = lcd[5];
assign RS = lcd[4];
assign D7 = lcd[3];
assign D6 = lcd[2];
assign D5 = lcd[1];
assign D4 = lcd[0];


OBUF #(
      .DRIVE(12),
      .IOSTANDARD("DEFAULT"),
      .SLEW("SLOW")
   ) OBUF_rw (
      .O(RW),
      .I(1'b0)
);
```

```
endmodule
```

Appendix 39: Timer

```
`timescale 1ns/1ps

module TIMER (
    input CLOCK, RESET, ENABLE,
  input [31:0] ENDING, ADDR,
    output BUSY
);


wire free, check, enable, reset;
wire comp;
wire [31:0] plus;
wire [31:0] d;
reg  [31:0] q = 0;
reg busy = 0;
reg [31:0] ending = 32'd0;


assign check = (ADDR == 32'd1024);
assign enable = (check & ENABLE);



// D-type flip flop
always@(posedge CLOCK)
    begin
        if (RESET)
            ending <= 32'd0;
        else if (enable)
            ending <= (ENDING - 2);
    end



// comparator
assign comp = (q > ending);
```

```
// reg
always@(posedge CLOCK)
  begin
    if (free)
        q <= 0;
    else if (busy)
        q <= q + 1;
  end


// busy
assign free = (comp | RESET);

always @(negedge CLOCK)
    begin
        if (free)
            busy <= 0;
        else if (enable)
            busy <= 1;
    end

assign BUSY = busy;

endmodule
```

Appendix 40: Constrains File

```
########################################################################################################
##  File name :         KC705.xdc
##
##  Details :           Constraints file
##                              FPGA family:    kintex7
##                              FPGA:           xc7k325t-2ffg900
##                              Speedgrade:     -2
##
########################################################################################################



########################################################################################################
## DESIGN CONSTRAINS
########################################################################################################

set_property CFGBVS GND [current_design]
```

```
set_property CONFIG_VOLTAGE 1.8 [current_design]


##################################################################################################
## PIN ASSIGNMENTS & I/O STANDARDS
##################################################################################################
## CLOCK

set_property -dict {PACKAGE_PIN AD11 IOSTANDARD LVDS} [get_ports CLOCK_N]
set_property -dict {PACKAGE_PIN AD12 IOSTANDARD LVDS} [get_ports CLOCK_P]


##################################################################################################
## BUTTONs

set_property -dict {PACKAGE_PIN AA12 IOSTANDARD LVCMOS18} [get_ports RESET_CLK]
set_property -dict {PACKAGE_PIN AB12 IOSTANDARD LVCMOS18} [get_ports RESET]


##################################################################################################
## LEDs

set_property -dict {PACKAGE_PIN AB8 IOSTANDARD LVCMOS18} [get_ports {LED[0]}]
set_property -dict {PACKAGE_PIN AA8 IOSTANDARD LVCMOS18} [get_ports {LED[1]}]
set_property -dict {PACKAGE_PIN AC9 IOSTANDARD LVCMOS18} [get_ports {LED[2]}]
set_property -dict {PACKAGE_PIN AB9 IOSTANDARD LVCMOS18} [get_ports {LED[3]}]
set_property -dict {PACKAGE_PIN AE26 IOSTANDARD LVCMOS18} [get_ports {LED[4]}]
set_property -dict {PACKAGE_PIN G19 IOSTANDARD LVCMOS18} [get_ports {LED[5]}]
set_property -dict {PACKAGE_PIN E18 IOSTANDARD LVCMOS18} [get_ports {LED[6]}]
set_property -dict {PACKAGE_PIN F16 IOSTANDARD LVCMOS18} [get_ports {LED[7]}]


##################################################################################################
## DIP SWITCHes

set_property -dict {PACKAGE_PIN Y29 IOSTANDARD LVCMOS18} [get_ports ENABLE]


##################################################################################################
## LCD

set_property -dict {PACKAGE_PIN AB10 IOSTANDARD LVCMOS18} [get_ports E]
set_property -dict {PACKAGE_PIN Y11 IOSTANDARD LVCMOS18} [get_ports RS]
set_property -dict {PACKAGE_PIN AB13 IOSTANDARD LVCMOS18} [get_ports RW]

set_property -dict {PACKAGE_PIN AA13 IOSTANDARD LVCMOS18} [get_ports D4]
set_property -dict {PACKAGE_PIN AA10 IOSTANDARD LVCMOS18} [get_ports D5]
set_property -dict {PACKAGE_PIN AA11 IOSTANDARD LVCMOS18} [get_ports D6]
set_property -dict {PACKAGE_PIN Y10 IOSTANDARD LVCMOS18} [get_ports D7]


##################################################################################################
## USB-UART

set_property -dict {PACKAGE_PIN M19 IOSTANDARD LVCMOS18} [get_ports RX_UART]
```

```
####################################################################################################
## SMA

set_property -dict {PACKAGE_PIN Y23 IOSTANDARD LVCMOS18} [get_ports CLOCK_MAIN]
set_property -dict {PACKAGE_PIN Y24 IOSTANDARD LVCMOS18} [get_ports RX_SMA]



####################################################################################################
## PLACEMENT CONSTRAINS
####################################################################################################

set_property DONT_TOUCH true [get_cells BYTEVAULT]
set_property DONT_TOUCH true [get_cells CLOCKING]
set_property DONT_TOUCH true [get_cells ENABLER]
set_property DONT_TOUCH true [get_cells CTRL_UNIT]
set_property DONT_TOUCH true [get_cells DATAPATH]
set_property DONT_TOUCH true [get_cells RAM_CONTROLLER]
set_property DONT_TOUCH true [get_cells RAM]
set_property DONT_TOUCH true [get_cells LEDS]
set_property DONT_TOUCH true [get_cells MEMORY]
set_property DONT_TOUCH true [get_cells LCD]
set_property DONT_TOUCH true [get_cells TIMER]
set_property DONT_TOUCH true [get_cells UART]

create_pblock CPU
add_cells_to_pblock [get_pblocks CPU] [get_cells -quiet [list \
          BYTEVAULT \
          CLOCKING \
          CTRL_UNIT \
          DATAPATH \
          ENABLER \
          LCD \
          LEDS \
          MEMORY \
          RAM \
          RAM_CONTROLLER \
          TIMER \
          UART]]


resize_pblock [get_pblocks CPU] -add {SLICE_X120Y50:SLICE_X153Y99}
##resize_pblock [get_pblocks CPU] -add {CLOCKREGION_X1Y1:CLOCKREGION_X1Y1}

set_property IS_SOFT FALSE [get_pblocks CPU]


set_property RAM_STYLE block [get_cells MEMORY]





set_property BEL BUFR [get_cells CLOCKING/BUFR_clk1]
```

```
set_property LOC BUFR_X1Y7 [get_cells CLOCKING/BUFR_clk1]
set_property BEL BUFR [get_cells CLOCKING/BUFR_clk3]
set_property LOC BUFR_X1Y4 [get_cells CLOCKING/BUFR_clk3]
set_property BEL BUFR [get_cells CLOCKING/BUFR_clk2]
set_property LOC BUFR_X1Y6 [get_cells CLOCKING/BUFR_clk2]
set_property BEL BUFR [get_cells CLOCKING/BUFR_clk21]
set_property LOC BUFR_X1Y5 [get_cells CLOCKING/BUFR_clk21]



set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTSU/OUT__1]
set_property LOC DSP48_X5Y31 [get_cells DATAPATH/MDU/MULTSU/OUT__1]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULT/OUT__1]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULT/OUT__0]
set_property LOC DSP48_X5Y33 [get_cells DATAPATH/MDU/MULT/OUT__1]
set_property LOC DSP48_X5Y32 [get_cells DATAPATH/MDU/MULT/OUT__0]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULT/OUT__2]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULT/OUT__3]
set_property LOC DSP48_X5Y34 [get_cells DATAPATH/MDU/MULT/OUT__2]
set_property LOC DSP48_X5Y35 [get_cells DATAPATH/MDU/MULT/OUT__3]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTSU/OUT__3]
set_property LOC DSP48_X5Y29 [get_cells DATAPATH/MDU/MULTSU/OUT__3]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTU/OUT__1]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTU/OUT__0]
set_property LOC DSP48_X5Y27 [get_cells DATAPATH/MDU/MULTU/OUT__1]
set_property LOC DSP48_X5Y26 [get_cells DATAPATH/MDU/MULTU/OUT__0]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTU/OUT__3]
set_property BEL DSP48E1 [get_cells DATAPATH/MDU/MULTU/OUT__2]
set_property LOC DSP48_X5Y25 [get_cells DATAPATH/MDU/MULTU/OUT__3]
set_property LOC DSP48_X5Y24 [get_cells DATAPATH/MDU/MULTU/OUT__2]



####################################################################################################
## TIME CONSTRAINS
####################################################################################################
## CLOCK

create_clock -period 5.000 -name sys_clk -add [get_ports CLOCK_P]


####################################################################################################
## DELAY

set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk3/O]] -max 2.000 [get_ports RX_UART]
set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk3/O]] -min 1.000 [get_ports RX_UART]

set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports ENABLE]
set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports RESET]
set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports RESET_CLK]

set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports ENABLE]
set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports RESET]
set_input_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports RESET_CLK]

set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[0]}]
```

```
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[1]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[2]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[3]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[4]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[5]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[6]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports {LED[7]}]

set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports E]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports RS]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports RW]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports D4]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports D5]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports D6]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -max 2.000 [get_ports D7]

set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[0]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[1]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[2]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[3]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[4]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[5]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[6]}]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports {LED[7]}]

set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports E]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports RS]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports RW]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports D4]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports D5]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports D6]
set_output_delay -clock [get_clocks -of_objects [get_pins CLOCKING/BUFR_clk2/O]] -min 1.000 [get_ports D7]
```