Three stage implementation of a custom 32 bit RISCV multicyle integer processor RV32-I

Daniel Andrés Crovo Pérez - Daniel Giovanni Fajardo Lopez
Pontificia Universidad Javeriana
Unconventional architectures

May 30, 2024

Abstract

An optimization of a custom 32-bit integer processor compatible with the RISC-V open-source architecture (RVI32) was achieved through the implementation of a three-stage pipeline. Originally designed by Daniel Fajardo, the processor initially operated in a multi-cycle manner, requiring multiple clock cycles to complete the fetch, decode, and execute stages. The enhancement involved modifying the data path and control logic to allow the instruction fetch, decode, and execute stages to run in parallel.The implementation of the three-stage pipeline allows for the concurrent execution of multiple instructions, thus reducing the overall instruction cycle time and improving the processor's cycles per instruction (CPI). The processor design was implemented in VHDL, compiled using Quartus, and simulated in ModelSim. Simulation results confirmed the correct handling of various instruction types, demonstrating the advantages of the pipelined design.

1 Introduction

RV32-I stands out for being open-source and highly customizable. The open-source RISC-V architecture is free, open. It can be used for any purpose, allowing anyone to design, manufacture, and sell RISC-V chips and software for future applications. RISC-V is a design alternative for free processors, implying that no license needs to be paid to any particular entity. This architecture has a modular design, meaning the core is stable and will not change, while extensions can be added to this core according to the needs and goals of the implementation. Examples include multiplication extensions, floating-point extensions, and double-precision floating-point, among others.

Pipelining is a powerful way to increase processors' and, in general, digital systems' throughput. By subdividing the single-cycle data path into several stages and allowing parallel execution of each stage, throughput can be improved by a factor of N times the number of stages, and ideally, the latency of each instruction is unchanged. In a non-pipelined processor, each instruction must go through all stages before the next instruction can begin. This sequential processing results in longer execution times. In contrast, a pipelined processor overlaps the execution of instructions. For example, while one instruction is being decoded, the next instruction can be fetched, and another can be executed. Since we already have the RVI32 processor implementation and a working knowledge of its design, we decided to use it for this final project. The rest of the document is organised as follows: Section 2 explains the RVI32 architecture and available instructions, Section 3 provides results of the characterisation of the processor

by implementing a bubble sort algorithm, section 4 describes the design steps to reach the pipelined architecture, section 5 provides the results of the implementation an simulation.

2 RVI32 Arquitecture

A processor with a RISC-V architecture understands words of n bits in RISC-V, meaning each instruction must conform to the format specified in tables 2 through 7 for the processor's internal hardware to correctly execute each task. The architecture of a processor resembles a person's language; thus, the hardware of a processor is directly linked to the architecture it is designed with, enabling it to execute each instruction correctly. The RISC-V instruction set is quite extensive;In this custom processor, we will only mention the instructions pertaining to integers. Integer-type instructions comprise R, I, B, J, L, and S instruction types.

Conjunto de instrucciones del RV32I:

Tipo R	funct7	rs2	rs1	funct3	rd	opcode
Add	0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
Sub	0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
and	0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
or	0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
xor	0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
slt	0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
sltu	0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1
sll	0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
srl	0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
sra	0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1

Figure 1: RVI32 R-type instructions.

R-type instructions (refer to figure 1) operate at the register level. They involve three addresses: rs1, rs2, and rsd. An operation is performed between the value at address rs1 and the value at source address rs2, with the result stored at destination address rsd.

Tipo B	Inm(12|10:5)	rs2	rs1	funct3	Inm(4:1|11)	opcode
beq	Inm(12|10:5)	rs2	rs1	0 0 0	Inm(4:1|11)	1 1 0 0 0 1 1
bne	Inm(12|10:5)	rs2	rs1	0 0 1	Inm(4:1|11)	1 1 0 0 0 1 1
blt	Inm(12|10:5)	rs2	rs1	1 0 0	Inm(4:1|11)	1 1 0 0 0 1 1
bge	Inm(12|10:5)	rs2	rs1	1 0 1	Inm(4:1|11)	1 1 0 0 0 1 1
bltu	Inm(12|10:5)	rs2	rs1	1 1 0	Inm(4:1|11)	1 1 0 0 0 1 1
bgeu	Inm(12|10:5)	rs2	rs1	1 1 1	Inm(4:1|11)	1 1 0 0 0 1 1

Figure 2: RVI32 B-type instructions.

B-type instructions (see figure 2) are responsible for executing a jump between instructions of the main program only if the jump condition is met; this value is not stored. An operation is performed between rs1 and rs2, and if the result matches the jump condition, the program counter increases by an offset whose value is contained in the instruction.

Tipo I	Inm(11:0)		rs1	funct3	rd	opcode
Addi	Inm(11:0)		rs1	0 0 0	rd	0 0 1 0 0 1 1
Andi	Inm(11:0)		rs1	1 1 1	rd	0 0 1 0 0 1 1
Ori	Inm(11:0)		rs1	1 1 0	rd	0 0 1 0 0 1 1
Xori	Inm(11:0)		rs1	1 0 0	rd	0 0 1 0 0 1 1
Slti	Inm(11:0)		rs1	0 1 0	rd	0 0 1 0 0 1 1
Sltui	Inm(11:0)		rs1	0 1 1	rd	0 0 1 0 0 1 1
Slli	0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
Srli	0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
srai	0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1

Figure 3: RVI32 I-type instructions.

Unlike R-type instructions, I-type instructions (see figure 3) contain an offset within the instruction with which operations will be performed and subsequently stored in a register. That is, the value at address rs1 must be retrieved and then operated on with the value contained in the instruction.

Tipo S	Inm(11:5)	rs2	rs1	funct3	Inm(4:0)	opcode
sb	Inm(11:5)	rs2	rs1	0 0 0	Inm(4:0)	0 1 0 0 0 1 1
sh	Inm(11:5)	rs2	rs1	0 0 1	Inm(4:0)	0 1 0 0 0 1 1
sw	Inm(11:5)	rs2	rs1	0 1 0	Inm(4:0)	0 1 0 0 0 1 1

Figure 4: RVI32 S-type instructions.

Tipo L	Inm(11:0)	rs1	funct3	rd	opcode
li	Inm(11:0)	rs1	0 0 0	rd	0 0 0 0 0 1 1
lh	Inm(11:0)	rs1	0 0 1	rd	0 0 0 0 0 1 1
lw	Inm(11:0)	rs1	0 1 0	rd	0 0 0 0 0 1 1
lbu	Inm(11:0)	rs1	1 0 0	rd	0 0 0 0 0 1 1
lhu	Inm(11:0)	rs1	1 0 1	rd	0 0 0 0 0 1 1

Figure 5: RVI32 L-type instructions.

S-type and L-type instructions (see figures 4 and 5) are responsible for accessing memory, both for extracting data and storing data. For L-type instructions, an operation is performed between the value at the register address rs1 and the constant contained in the instruction, and its result is the address from which data will be extracted and stored in rd. On the other hand, for storing values in memory (S-type instructions), an operation is performed between the values of addresses rs1 and rs2 to obtain the address where the constant contained in the instruction will be stored.

Each of these instructions are distinguished by the opcode, and among them, they are differentiated by the set of bits referred to as funct7 for R-type instructions and funct3 for B, I, J, S, L-type instructions. The aforementioned instruction set (RISC-V architecture) forms a fundamental and elemental set for designing an RISC-V processor version RV32-I.

3 Characterisation of the processor

3.1 Bubble Sort Algorithm

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list or vector[n] to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order. This process is repeated until the entire list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list with each pass. While not the most efficient sorting algorithm for large datasets, Bubble Sort is easy to understand and implement, making it a common choice for educational purposes and small datasets.

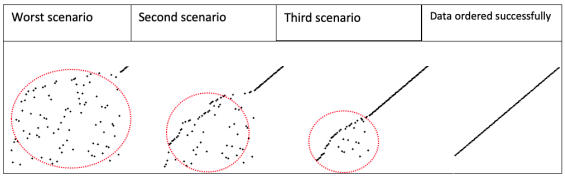

Figure 6: Bubble sort case scenarios.

In figure 6, we can observe three scenarios that may occur in the bubble sort algorithm. In the first scenario, the data is in the opposite order. In the second and third scenario, we can easily observe some type of order on the right side of the vector. In the fourth scenario, we have a successfully ordered vector, where we can observe that it is completely ordered.

- worst_scenario_vector[n=8]={8,7,6,5,4,3,2,1}
- second_scenario_vector[n=8]={1,2,6,5,4,3,7,8}
- first_scenario_vector[n=8]={1,2,3,5,4,6,7,8}
- ordered_vector[n=8]={1,2,3,4,5,6,7,8}

```
1   Function BubbleSort(a0, a1, a2, ..., a(n-1))
2   {
3   for i ← 1 < n - 1 do
4       for j ← 0 < n - i do
5           if a(j) > a(j + 1) then
6               aux ← a(j)
7               a(j) ← a(j + 1)
8               a(j + 1) ← aux
9           end if
10      end for
11  end for
12  }
```

Figure 7: Bubble sort pseudocode.

The algorithm starts by iteratively traversing the array, comparing adjacent elements, and swapping them if they are out of order. This process continues until the array is sorted. Specifically, the outer loop controls the number of passes through the array, while the inner loop performs comparisons and swaps within each pass. The conditional statement within the inner loop dictates the swapping behavior based on the comparison results.In this pseudocode n represents the length of the array being sorted. It is the total number of elements in the array, i is a variable used as a loop counter in the outer loop. It iterates over the indices of the array from 1 to n-1. This loop controls the number of passes through the array, j is a variable used as a loop counter in the inner loop. It iterates over the indices of the array from 0 to n-i. This loop controls the comparisons and swaps within each pass of the algorithm.

3.2 Implementation

Based on the pseudo code depicted in figure 7 we proceeded to implement the algorithm in assembly, based on the Instruction Set Architecture supported by our RVI32 processor. We implemented the loops via branch instructions, and at the end of the code we load the data from RAM to show the sorted array in the simulation. The following code provides the implementation.

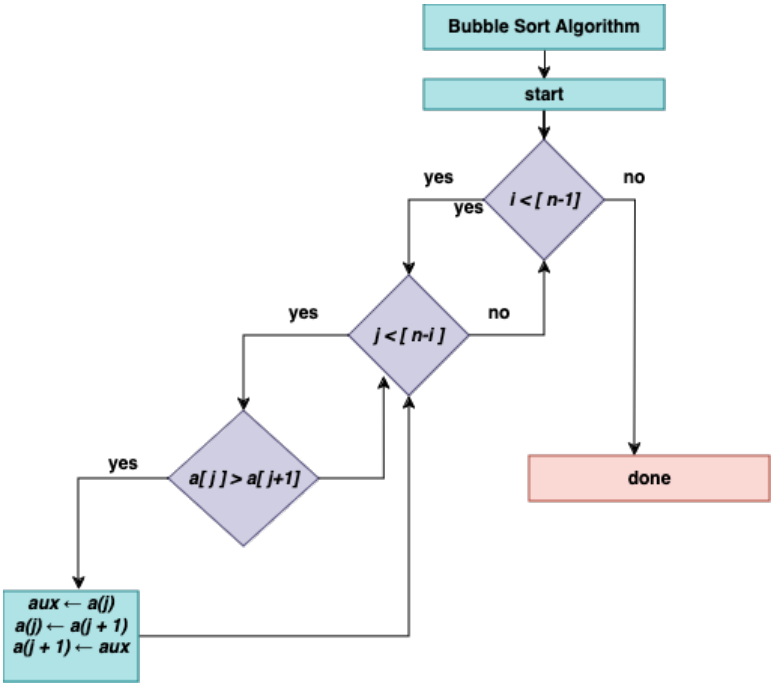

Figure 8: Bubble sort flowchart based on pseudocode.

Listing 1: Bubble Sort in RISC-V Assembly

```
# Initialize loop iterators in the register bank
addi 0, rs0, rs31  # Set rs31 to 0, loop iterator for the outer loop
addi 1, rs0, rs30  # Set rs30 to 1, loop iterator for the inner loop
addi 8, rs0, rs29  # rs29 is set to 8, max index
addi 8, rs0, rs28  # rs28 is set to 8, max index
addi 8, rs0, rs27  # rs27 is set to 8, max index
addi 8, rs0, rs26  # is set to 8, max index
addi 8, rs0, rs25  # rs25 is set to 8, max index
addi 8, rs0, rs24  # rs24 is set to 8, max index
addi 1, rs0, rs23  # rs23 is set to 1, used for incrementing

# Load the first two digits to be sorted from RAM
lw 0, rs31, rs22  # Load word from memory at address in rs31 into rs22
lw 0, rs30, rs21  # Load word from memory at address in rs30 into rs21

# Compare and potentially branch to swap
bgt (4), rs21, rs22  # If rs21 > rs22, branch to PC+4 (skip swap)

# Increment loop iterators if no swap is needed
addi 1, rs31, rs31  # Increment rs31 by 1
```

```
addi 1, rs30, rs30  # Increment rs30 by 1

# Branch to outer loop start if needed
bgt -(5), rs29, rs28  # Branch to PC-5 if rs29 > rs28,  loop condition

# Operations if branching did not occur
addi 0, rs31, rs31  # Reset rs31 to 0
sw rs21             # Store rs21 to the address pointed by
addi 0, rs30, rs30  # Reset rs30 to 0
sw rs22             # Store rs22 to the address pointed by


bgt (2), rs31, rs25  # Branch forward by 2 if rs31 > rs25
bgt -(11), rs29, rs28 # Branch backward by 11 if rs29 > rs28, likely to handle anot

# Resetting registers to initial values or constants
addi 0, rs0, rs31   # Set rs31 to 0
addi 1, rs0, rs30   # Set rs30 to 1
addi 1, rs15, rs15  # Set rs15 to 1

# Branching based on comparisons for sorting continuation
bgt (2), rs31, rs15  # If rs31 > rs15, branch forward by 2
bgt -(16), rs29, rs28 # Branch backward by 16 if rs29 > rs28, to restart

# Load sorted array to view order
lw 0, rs0, rs22   # Load word from address in rs0 into rs22
lw 1, rs0, rs21   # Load word from address in rs0+1 into rs21
lw 2, rs0, rs22   # Load word from address in rs0+2 into rs22
lw 3, rs0, rs21   # Load word from address in rs0+3 into rs21
lw 4, rs0, rs22   # Load word from address in rs0+4 into rs22
lw 5, rs0, rs21   # Load word from address in rs0+5 into rs21
lw 6, rs0, rs22   # Load word from address in rs0+6 into rs22
lw 7, rs0, rs21   # Load word from address in rs0+7 into rs21
```

Once the assembly code was developed, we proceeded to translate it into machine code following abovementioned architecture specification in section 2. Below is the resulting machine code, this code was saved into a .mif file and is loaded into the ROM.

```
CONTENT BEGIN
    0       :   00000000000000000000111110010011;
    1       :   00000000000100000000111100010011;
    2       :   00000000100000000000111010010011;
    3       :   00000000100000000000111000010011;
    4       :   00000000100000000000110110010011;
    5       :   00000000100000000000110100010011;
    6       :   00000000100000000000110010010011;
    7       :   00000000100000000000110000010011;
```

```
8       :   0000000000010000000101110010011;
9       :   0000000000001111100010110000011;
10      :   0000000000001111000010101000011;
11      :   0000000101011011010100100110011;
12      :   0000000000011111100011111010011;
13      :   0000000000011111000011111000011;
14      :   1111111110111100101110111100011;
15      :   0000000000001111100011111010011;
16      :   0000000101010000000000000001111111;
17      :   0000000000001111000011111000011;
18      :   0000000101100000000000000001111111;
19      :   0000000110011111110100010110011;
20      :   1111111110111100101101011100011;
21      :   0000000000000000000001111110011;
22      :   0000000000010000000011111000011;
23      :   0000000000010111100001111010011;
24      :   0000000110010111111010001011000011;
25      :   1111111110111100101100001100011;
26      :   0000000000000000000001011000000011;
27      :   0000000000010000000010101000000011;
28      :   0000000000100000000010110000000011;
29      :   0000000000111111000010101000000011;
30      :   0000000001000000000010110000000011;
31      :   0000000001010000000010101000000011;
32      :   0000000001100000000010110000000011;
33      :   0000000001110000000010101000000011;
34      :   0000000010000000000010101000000011;
35      :   0000000010010000000010110000000011;
[36..37]   :   0000000010100000000010101000000011;
38      :   0000000010110000000010101000000011;
39      :   0000000011000000000010101000000011;
40      :   0000000011010000000010110000000011;
41      :   0000000011100000000010101000000011;
42      :   0000000011110000000010101000000011;
[43..32767]   :   0000000000000000000000000000000000;
```

3.3 Simulation

To simulate the behaviour of the algorithm implementation, an array of 8 values was loaded into the RAM, we decided to test the worst case scenario in which the array is completely unsorted, so the algorithm should iterate through the two loops until the end leading to an algorithmic complexity of n^2. The table 1 shows the array loaded into RAM.

Figure 10 shows the simulation run in Modelsim. As it can be seen the implementation works, the Program Counter correctly jumps after the comparison between the two adjacent values.

Index	Value
0	16
1	14
2	12
3	10
4	8
5	4
6	2
7	1

Table 1: Index-Value Table

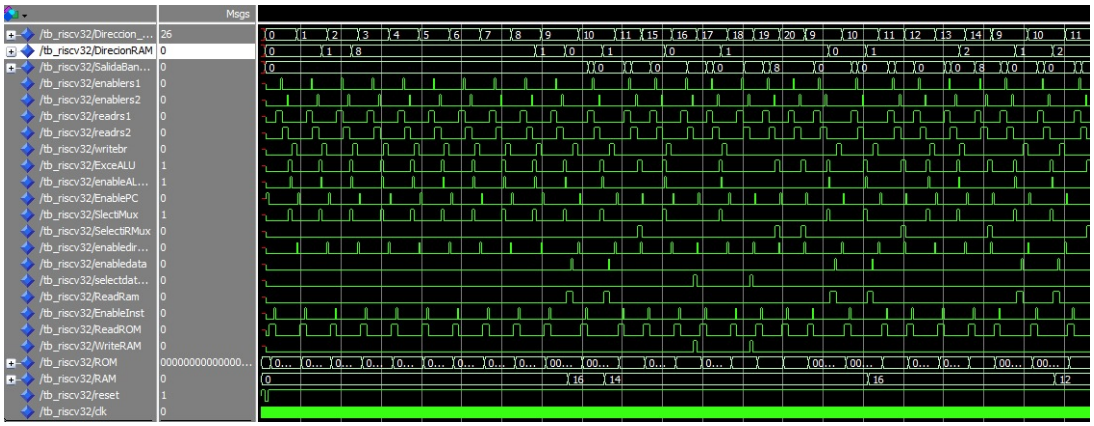

Figure 9: Bubble sort flowchart based on pseudocode.

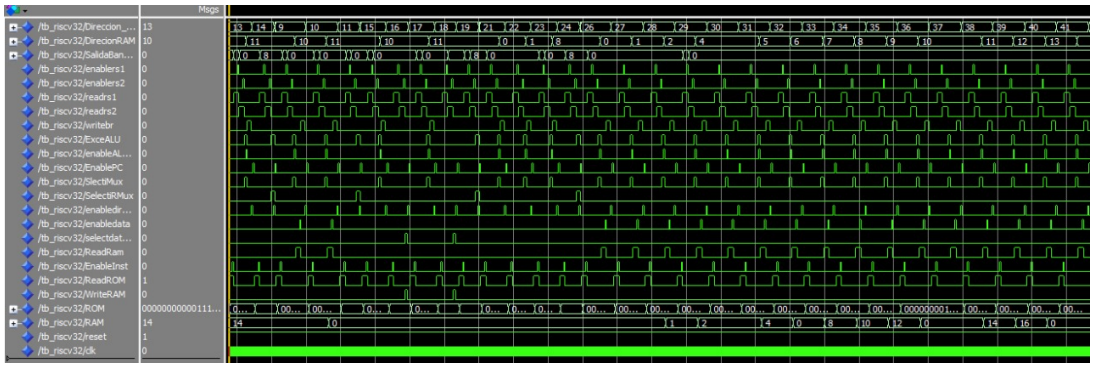

Figure 10: Bubble sort flowchart based on pseudocode.

4 Pipeline design

Figure 11 illustrates the initial microarchitecture design implemented for the RVI32 processor. This multi-cycle implementation includes registers to store intermediate results of the functional units. These

registers are essential to prevent the loss of data, as each instruction takes multiple cycles to complete. For this project, the functionality of these registers was extended to store the results of the instruction fetch and instruction decode stages, enabling the execute stage to properly process previous instructions in the correct order. Additionally, the control unit was redesigned to manage the additional steps required for executing each stage in parallel.

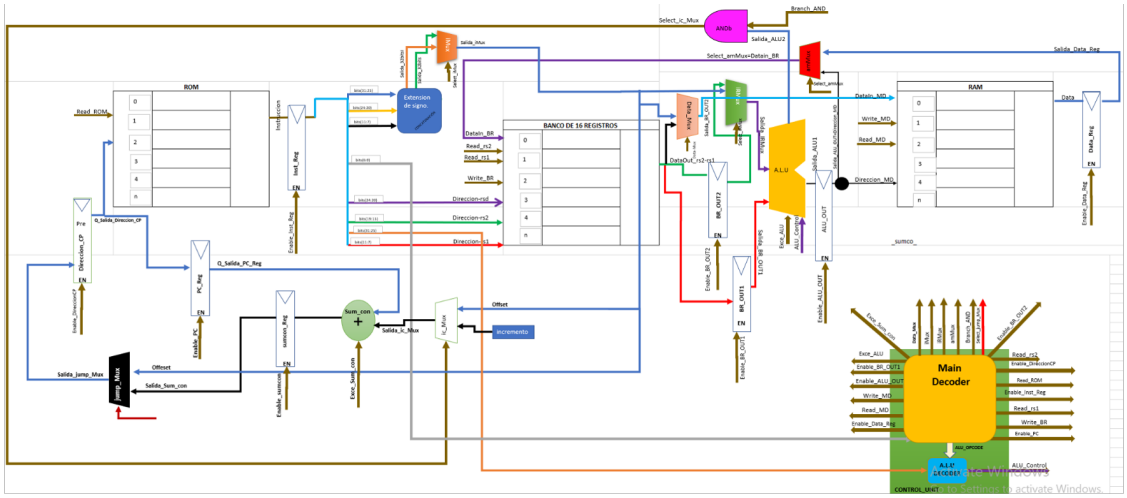

Figure 11: Initial microarchitecture schematic of the RVI32.

The approach began with a timing analysis of different types of instructions. Figure 12 illustrates the typical operation of the RVI32 processor for a type S instruction. After the start signal, the processor initializes, and the fetch stage begins. The three stages are delineated by red lines. As shown, the instructions are processed sequentially, and the stages do not operate in parallel.

To enable the processor to execute instructions in a pipelined manner, we designed the timing for each signal per stage. for instance, by setting the Enable_PC signal to 1 after the execution of the EnableInst_Reg signal, which is the last step in the fetch stage, we allowed the processor to start the fetch stage again to fetch the next instruction.

To enable pipelining in the RVI32 processor, several key changes were made to the control signals and data path. an updated description of each signal is provided:

- **Enable_PC**: The signal is now activated after EnableInst_Reg to allow the program counter to update its value at the beginning of each fetch stage for new instructions.
- **Read_ROM**: This signal is activated to read the instruction from memory at each fetch stage, ensuring that the instruction is available for the subsequent decode stage.
- **EnableInst_Reg**: This signal stores the fetched instruction in the instruction register at the end of each fetch stage, allowing it to be decoded in the next cycle.
- **Exce_Sum_con and Exce_ALU**: These signals are used to execute arithmetic and logical operations. They are activated in the execute stage, ensuring that the ALU performs the required operations on the operands.

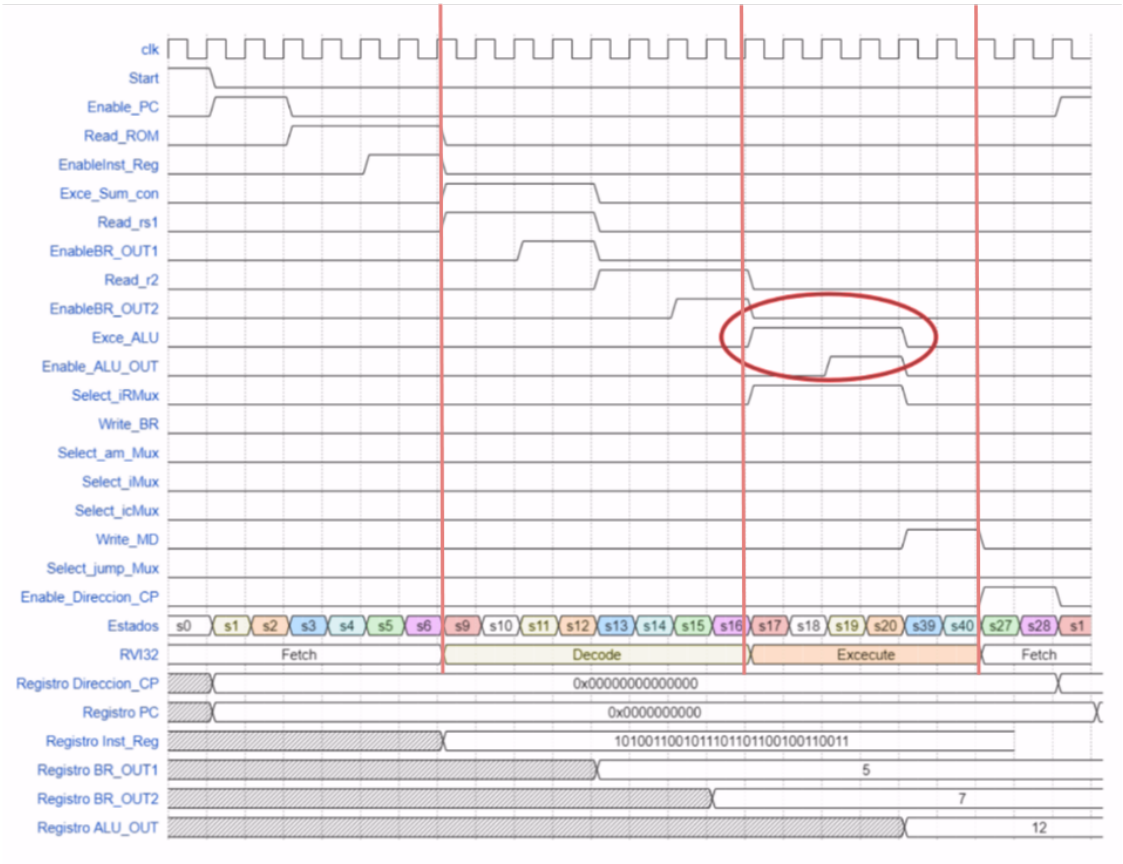

Figure 12: Timing diagram of type s instruction non pipelined.

- **EnableBR_OUT1 and EnableBR_OUT2**: These signals control the enabling of the output registers that store the results of the read operations from the register file. They ensure that the appropriate data is available for the execute stage.
- **Read_rs1 and Read_r2**: These signals initiate the reading of the source registers during the decode stage, making the operand data available for the execute stage.
- **Enable_ALU_OUT**: This signal enables the output register that stores the ALU results, ensuring that the executed data is stored properly for the next instruction.
- **Select_iRMux, Select_am_Mux, Select_iMux, Select_icMux, and Select_jump_Mux**: These multiplexers control various data paths within the processor, routing the correct data to the appropriate stages and components based on the instruction type.
- **Write_MD and Enable_Direccion_CP**: These signals handle writing the final results back to memory or updating the program counter, ensuring that the results of the executed instructions are stored and that the program counter is correctly updated for the next instruction fetch.

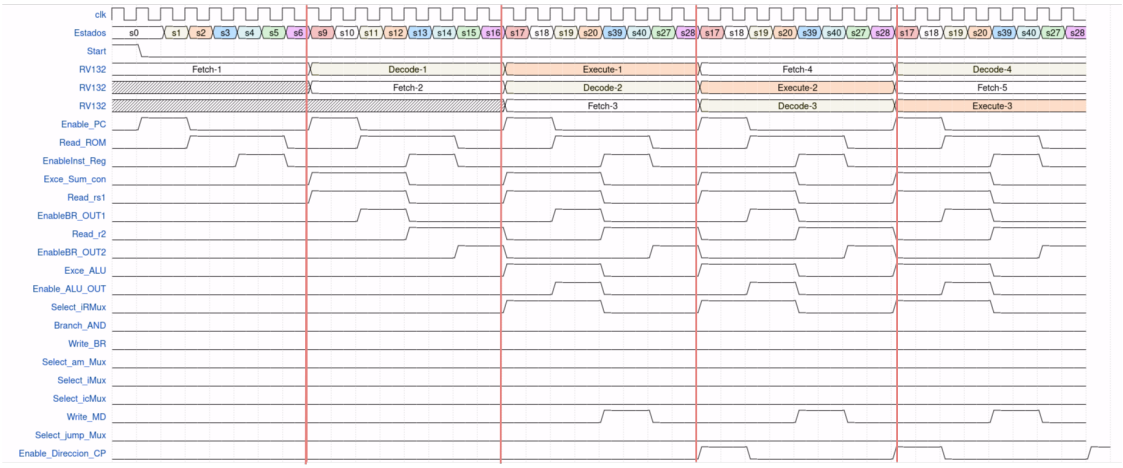

Figure 13: Timing diagram of type s instruction with pipeline.

The resulting timing analysis is illustrated int figure 13 this represents the ideal solution an behaviour of the type s instruction when implementing pipelining, at the third fetch run, the pipeline is full so there are 3 instructions executing in parallel.

4.1 Data path pipelining

4.2 Control pipelining

5 Implementation and simulation

The design was implemented in VHDL and compiled with quartus.

6 Conclusions

Referencias

- Referencia 1.RVI32

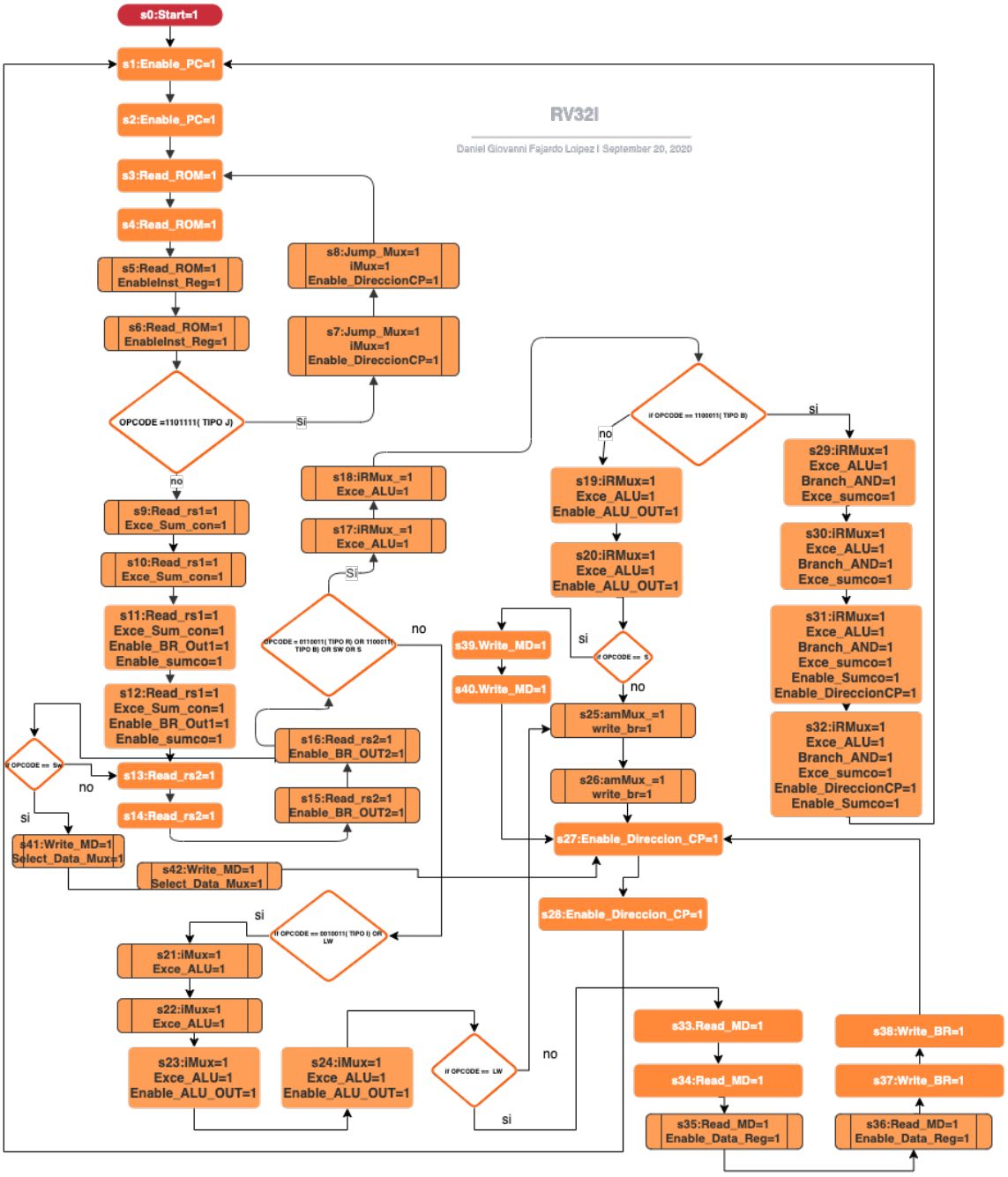

Figure 14: Control dataflow.

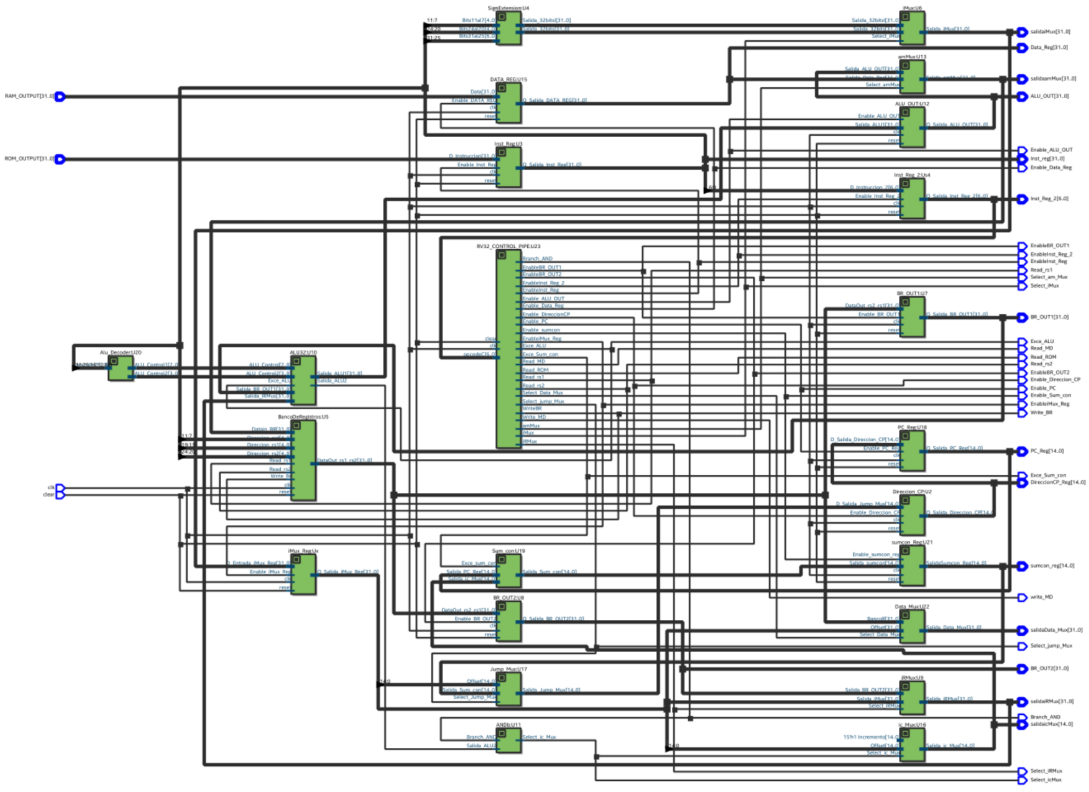

Figure 15: Compiled design in quartus.

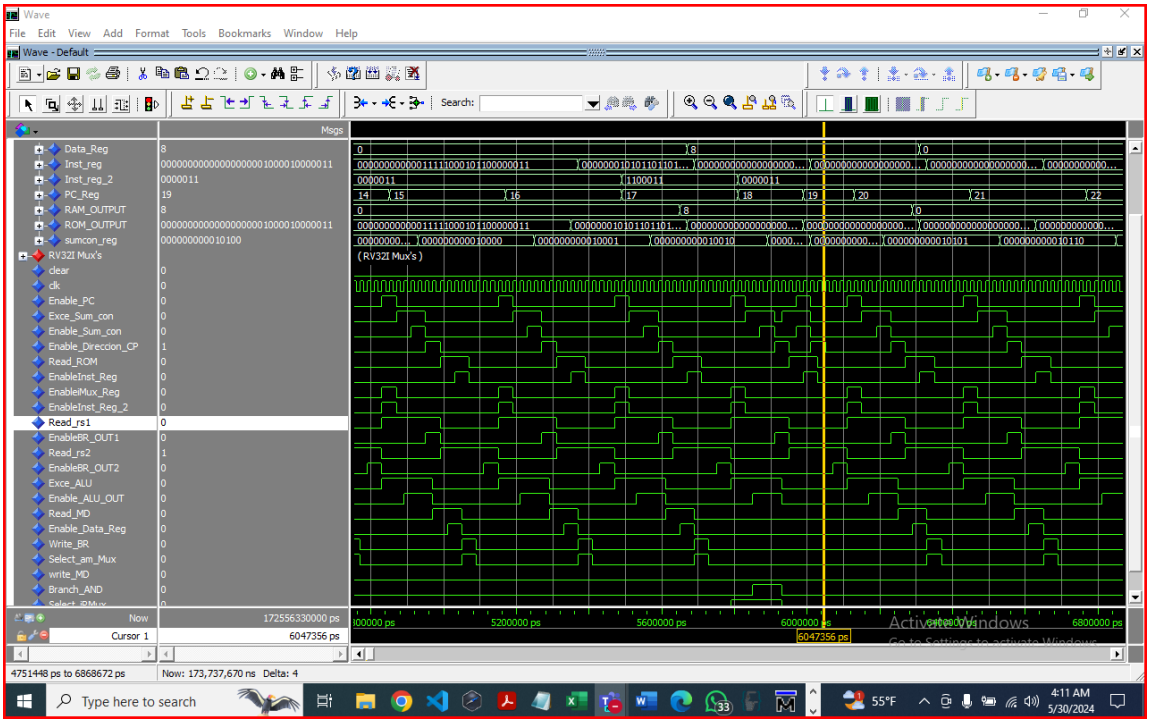

Figure 16: Simulation type b instruction .

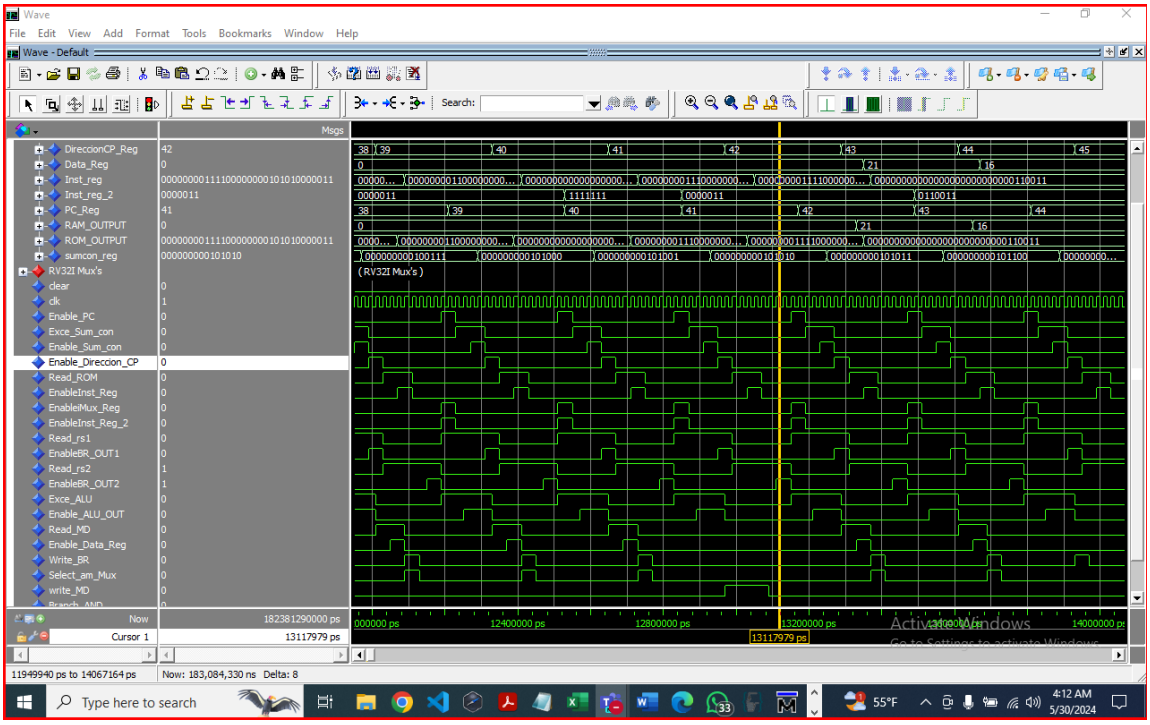

Figure 17: Simulation type s instruction .

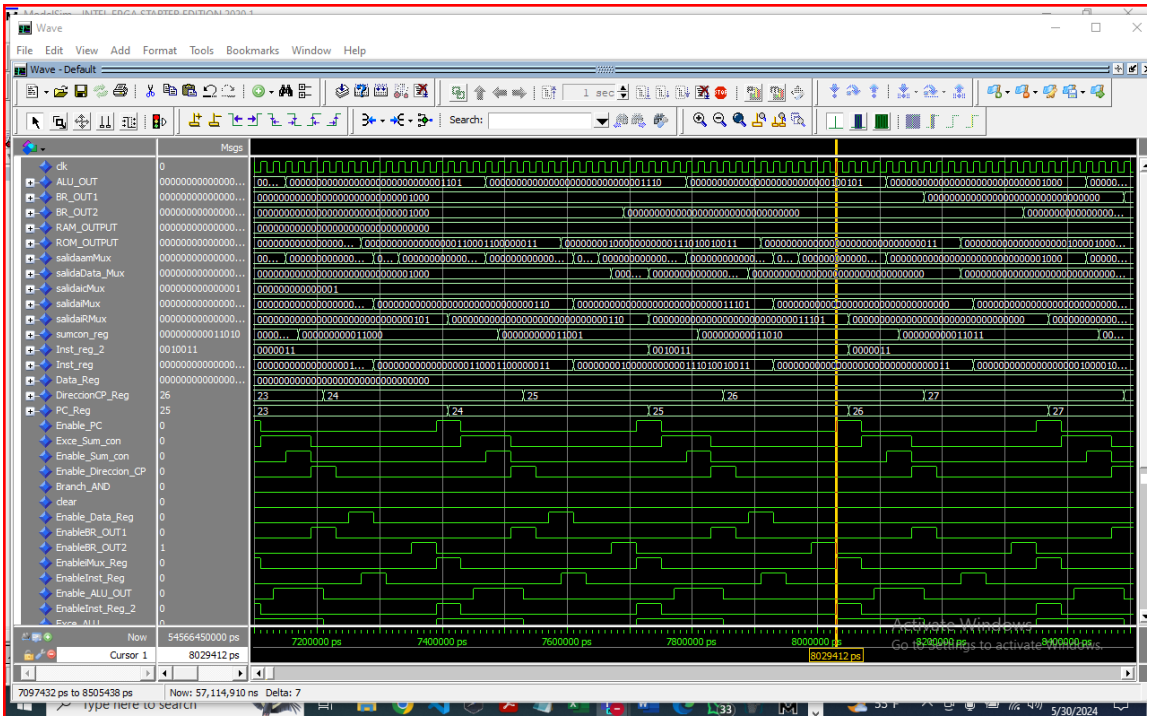

Figure 18: Simulation type I instruction .