



## Design of a Pipelined Mini RISC-V Processor

### **By:**

Mark Hanna                      A2110650

Jad Zaiter                        A2111535

### **Presented to:**

Dr Rafic Ayoubi

A report for CPEN314: Computer  
Architecture

19-12-2023

# Table of Contents

## Contents

Objective: .....	3
Introduction .....	3
Description Of Project.....	4
A-Components Description:.....	5
1-ROM (Read only memory): .....	5
2-ALU: .....	6
3-RAM (Random Access memory): .....	7
4-Register File (RF): .....	8
B-Control units Description: .....	9
1-Main control unit: .....	9
2-ALUcontrol: .....	10
3-Hazard detection Unit (HDU): .....	11
4-Forwarding unit: .....	12
5-Immediate Generation Unit (IGU): .....	13
C-Connections between components: .....	14
1-Fetch Stage:.....	14
2-Decode Stage: .....	15
3-Execution Stage: .....	16
4-Memory Stage:.....	18
5-Write-Back Stage:.....	19
Simulations: .....	20
Conclusion: .....	22
Future Work: .....	22

## Objective:

The project, titled "Design of a Pipelined Mini-RISC-V Processor," involves building a pipeline processor based on a subset of the RISC-V instruction set. It comprises three main parts: designing the ALU (Arithmetic Logic Unit) and Immediate Generation Unit (IGU), designing the pipelined datapath using these units, and designing the control unit, forwarding unit, and hazard detection unit. Students can use schematic or SystemVerilog for the design. The project report should include the objective, introduction, detailed project description, schematics, and alternative solutions considered.

## Introduction

Computer architecture is a set of rules and methods that define the functioning, organization, and implementation of computer systems. It is frequently characterized by the CPU, memory, and I/O interfaces and serves as the blueprint for developing and producing computers.

The history of computer architecture is inextricably linked to the evolution of computer technology. The Von Neumann architecture was one of the earliest and most influential architectural designs, as outlined in John von Neumann's 1945 paper "First Draft of a Report on the EDVAC." This concept specified a system in which a single memory region held both instructions and data, and it has since influenced computer design.

The rise of integrated circuits and microprocessors in the 1960s and 1970s accelerated the evolution of computer architecture. The invention of the Intel 4004, the first microprocessor, in 1971 was a watershed moment. This era saw the introduction of personal computers and well-known architectures such as Intel's x86.

The emphasis on computer architecture changed in the next decades to parallel computing and performance per watt, resulting in the creation of multicore processors and energy-efficient architectures. Today, advances in quantum computing, neuromorphic computing, and other creative architectures designed to suit the needs of modern computing workloads are propelling the industry forward.

## Description Of Project

- The project is a pipelined processor based on a simplified instruction set of RISC-V. The processor should be able to handle the following format:

Instructions Format	Supported Operations
R-type	add, sub, and , or , xor, slt, sltu
I-type	addi, andi, ori, xori, slti, sltiu, lw
S-type	sw
SB-type	beq, bne

- The basic processor incorporates:
  1. ROM unit that resembles the instruction memory
  2. ALU unit responsible for arithmetic and logical (including branching operations a save operation)
  3. RAM unit that resembles the data memory
  4. RF (Register File)
  5. Multiple Control units:
    - Control-unit which provides control signal for the other control units and the overall processor.
    - ALU-control-unit: controls the operations of the ALU.

- HDU (Hazard-Detection-Unit): this unit prevents WAR (write after read) hazards.
- FU (Forwarding-Unit): this unit prevents RAW (read after write) hazards.
- IGU: this unit extracts the immediate from the op code.

## A-Components Description:

### 1-ROM (Read only memory):

The ROM resembles the instruction memory. It has 1 input of 24 bits to reference the memory location. In addition, It has 1 output of 32 bits because each instruction is 32 bits so each location in the ROM must hold 32 bits.

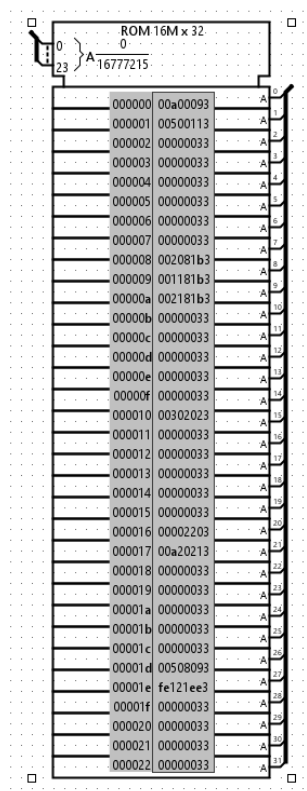


Figure 1: ROM block diagram

## 2-ALU:

The ALU is utilized for arithmetic operations such as PASS B, ADD, SUB, SLT, and SLTU, as well as logic operations including AND, OR, and XOR. The ALU has two 32-bit inputs, A and B, which serve as operands. An input OP is used to select the appropriate operation. The ALU produces two outputs: ALUOUT (32 bits), representing the result of the operation, and Zero, indicating whether the result is zero or not.

ALU Table			
S2	S1	S0	Operation
0	0	0	AND
0	0	1	OR
0	1	0	XOR
0	1	1	PASS B
1	0	0	ADD
1	0	1	SUB
1	1	0	SLT
1	1	1	SLTU

Figure 2: Operations of the ALU based on select line

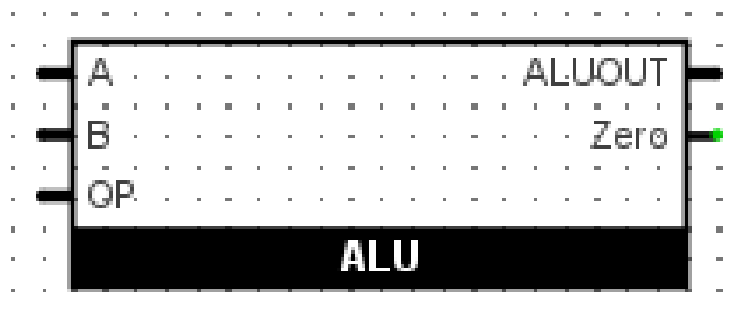


Figure 3: ALU block Diagram

### 3-RAM (Random Access memory):

The RAM, resembling data memory, consists of a 24-bit input to reference memory locations.

It also has one input to enable writing and one input to enable reading, but the output is delayed by 1 cycle. It features one 32-bit output.

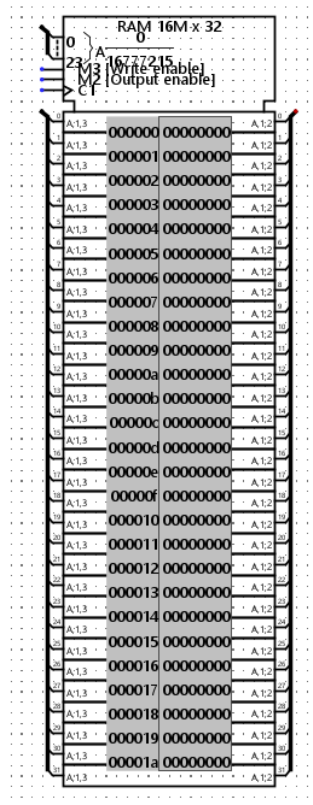


Figure 4: RAM block diagram

#### 4-Register File (RF):

The Register File consists of 32 registers, each 32 bits, that can be used to store data. This is similar to memory but is very fast. The first register is hardwired to 0. It has 2 read ports and 1 write port, each with a 5-bit address. Additionally, it features a write enable line. The write data port is 32 bits, corresponding to the size of each register. (The outputs x1, x2, x4, and x4 are used to monitor the values in those registers for simulation purposes.)

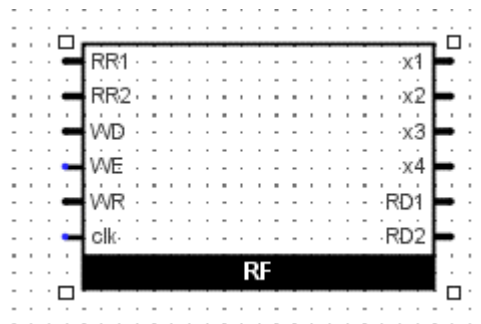


Figure 5: RF block diagram



## B-Control units Description:

### 1-Main control unit:

The main control unit will set the flags that control the 3 stages: execution, memory, and write back. The flags for execution are S[1:0] for the IGU and ALUsrc1 for the select line that chooses either B or immediate as an input to the adder. The flags for memory are MemR, to read from memory, MemW, to write to memory, and Membr, to branch. The flags for the write-back stage are MemtoReg to choose whether MDR or C gets written into the register file and RegWr to enable writing on the register file. This unit takes as input only the op code to determine the type of instruction and as an output the flags (8 bits).

Type/Flags	S[1:0]	ALUsrc1	MemR	MemW	Membr	MemtoReg	RegWr
<b>R</b>	00	0	0	0	0	0	1
<b>I</b>	00	1	0	0	0	0	1
<b>I (only lw)</b>	00	1	1	0	0	1	1
<b>S</b>	01	1	0	1	0	0	0
<b>SB</b>	10	0	0	0	1	0	0

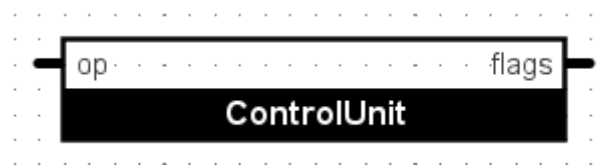


Figure 6: Control unit block diagram

## 2-ALUcontrol:

This unit controls the ALU by providing the select line (S, 3 bits) needed for each instruction.

To identify the operation for each instruction, this unit takes as input the opcode, func3, and func7. With this data, the required operation for each instruction can be identified. The opcode specifies the type of instruction, while func3 combined with function 7 is used to specify the operation needed for R and I types (excluding lw). For type I (for lw) and S, only the add operation is needed, so the select line is 4. For type SB, only the sub operation is needed, so the select line is 5.

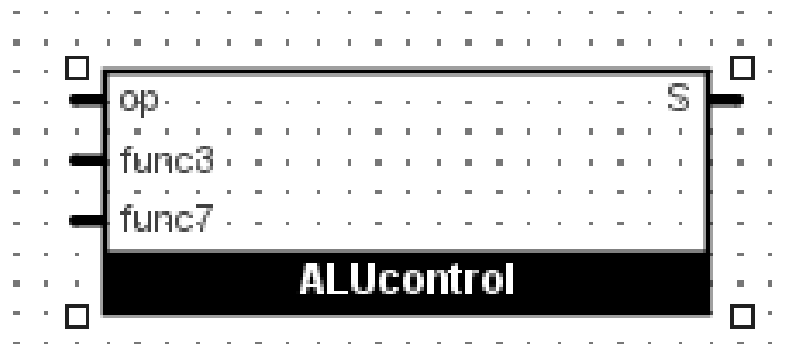


Figure 7: ALUcontrol Block diagram

### 3-Hazard detection Unit (HDU):

The hazard detection unit is used to stall the processor for one cycle when the first instruction is "lw" and has a dependency with the second instruction (one of the source registers of the second instruction is the destination register for "lw"). If the two instructions are consecutive, the "lw" instruction will not be able to forward the new data to the second instruction. This stall allows the "lw" instruction to forward data from the writeback to the execution stage. This function takes as input the source registers from the decode stage, the memory read flag, and the destination register from the execution stage, and checks if there is a dependency. If there is a dependency and the memory read flag is 1, it provides an output of 1.

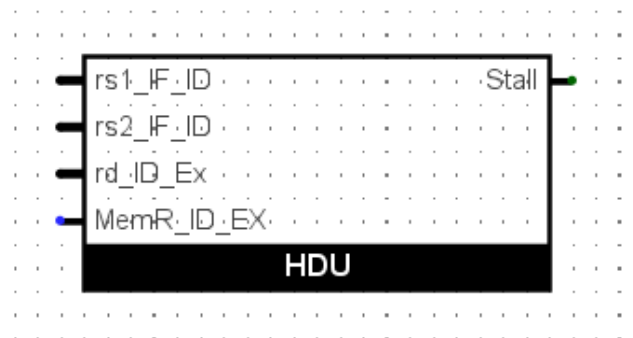


Figure 8: HDU block diagram

#### 4-Forwarding unit:

The Forwarding unit is employed to transmit data backward from the memory or writeback stage to the execution stage when an instruction requires it. This is crucial to avoid dependencies where the source registers in the execution stage might be the destination register from the memory or writeback stage, which has not been written yet in the Register File (RF). Without forwarding, new instructions might use outdated data.

To forward data from the memory stage, the conditions are that the source registers in the execution stage are equal to the destination register in the memory stage, the RegWr flag in the memory stage is 1, and the destination register in the memory stage is not 0. For forwarding from the writeback stage, the conditions are that the source registers in the execution stage are equal to the destination register in the writeback stage, the RegWr flag in the writeback stage is 1, and the destination register in the writeback stage is not 0. If data is needed to be forwarded from memory, the output is 01; if from the writeback stage, then the output is 10. If no forwarding is needed, the output is 0.

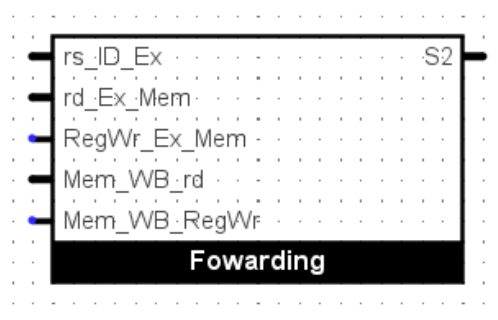


Figure 9: Forwarding block diagram

### 5-Immediate Generation Unit (IGU):

Instructions such as I, S, and SB contain immediate in their instruction. This immediate is distributed in 3 different ways across the instruction. There are 12 bits from the 32 bit instruction which are allocated for the immediate. The input to this unit is the S[1:0] flag and 32 bit instruction. From the S[1:0] flag the unit will know which type of instruction it is. The picture below will demonstrate where the immediate (immed) bits are:

Name (Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format

Figure 10: Distribution of bits for each instruction

IGU Table

S1	S0	Imm Gen
0	0	I
0	1	S
1	0	B

Figure 11: Select line based on type of instruction

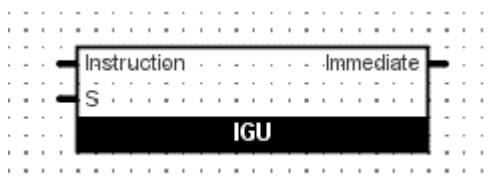
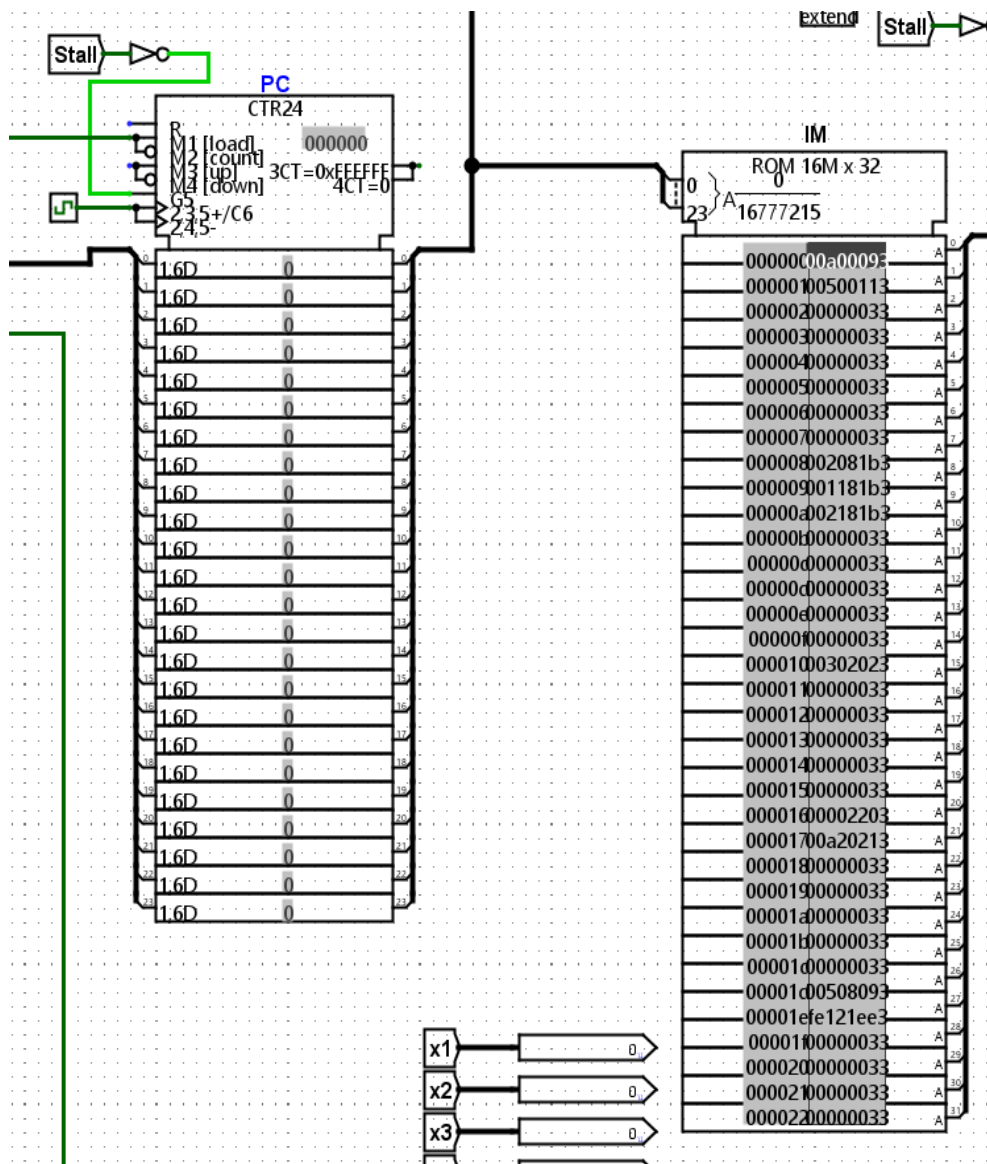


Figure 12: IGU Block diagram

## C-Connections between components:

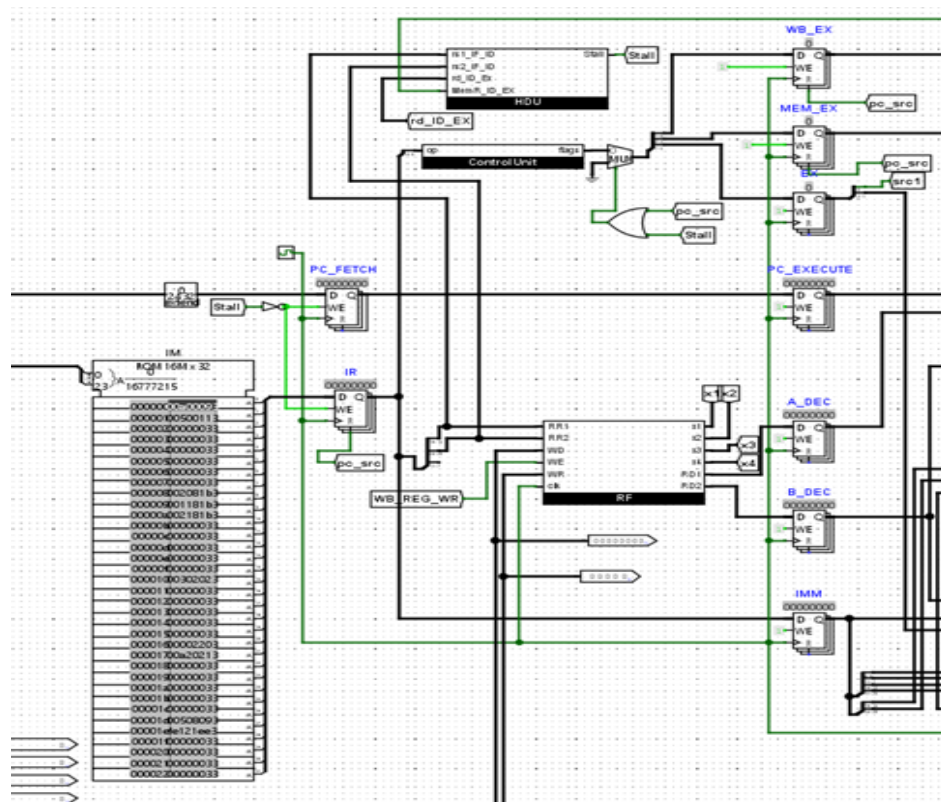
### 1-Fetch Stage:

In our program, the program counter functions as a counter, incrementing each time to reference an address in the ROM and retrieve data. If a stall is required, the counter retains its current value. In case of a branch, the counter is loaded with the new value to redirect the program flow. (The signal that loads the counter is pcsrc to indicate a branch occurred, the input to the counter is always PC + immediate, otherwise it increments 1 by 1)



## 2-Decode Stage:

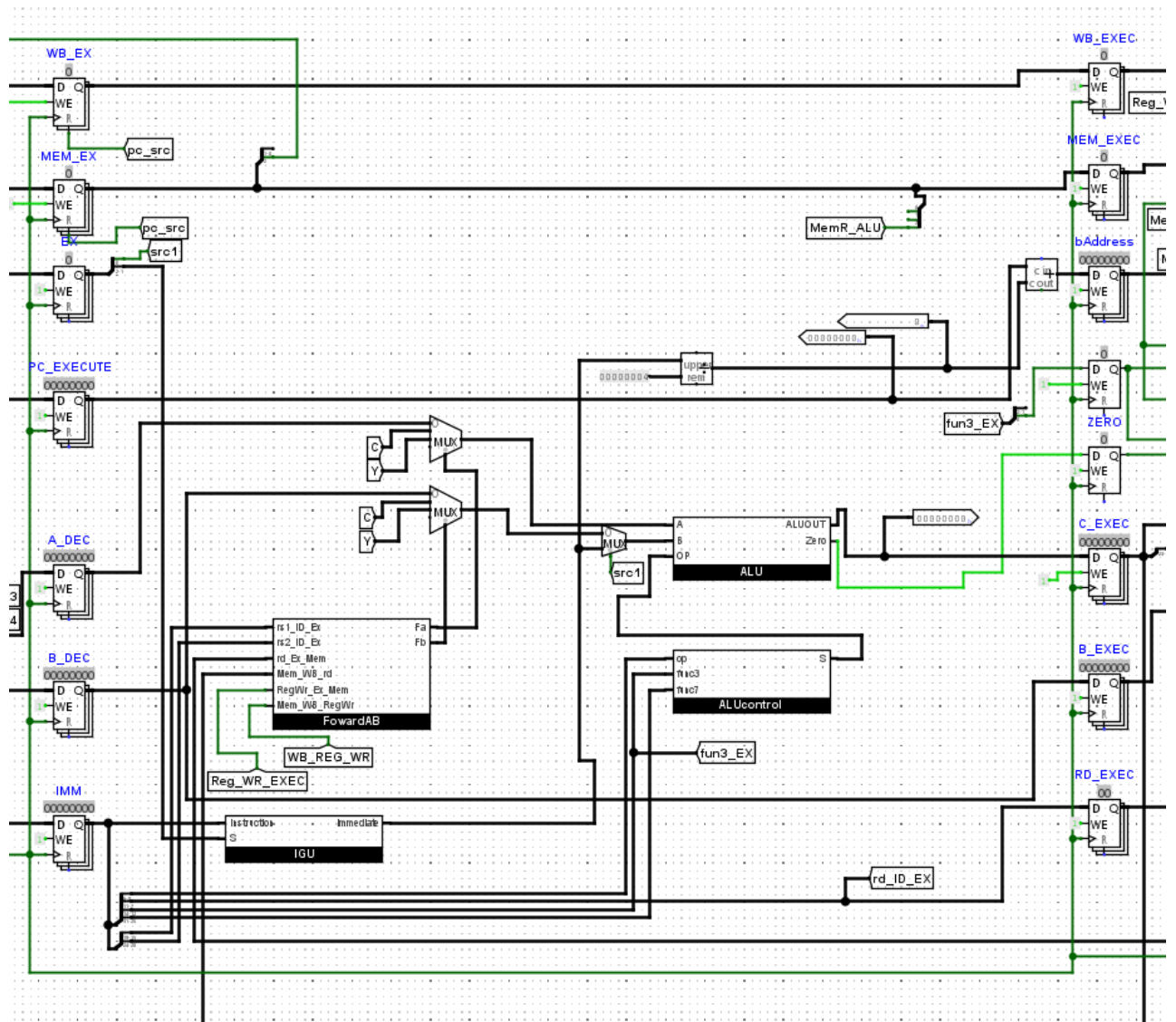
In the decode stage, the instruction fetched from the ROM is stored in the IR register, and the PC is saved in the PC\_FETCH register. From the IR register, rs1 and rs2 are extracted, serving as input for the RF and HDU. The data from the two read ports go into registers A and B. The OP code is extracted and sent as input to the control unit. The IR register is passed into the IMM register for other units. The control unit utilizes the OP code to generate flags, sending them to three registers: WB\_EX for the writeback stage, MEM\_EX for the memory stage, and EX for the execution stage. In the event of a taken branch, we flush the pipeline by clearing the MEM\_EX and WB\_EX registers. To manage the instruction in the decode stage, we clear the IR register and set the select line of the control unit's multiplexer to 1, resulting in all flags being set to 0. If a stall is required, we deactivate the write enable of both PC\_FETCH and IR, ensuring their contents remain unchanged. Additionally, we set all flags to 0 to prevent the current instruction passing from decode to memory from impacting the future.



### 3-Execution Stage:

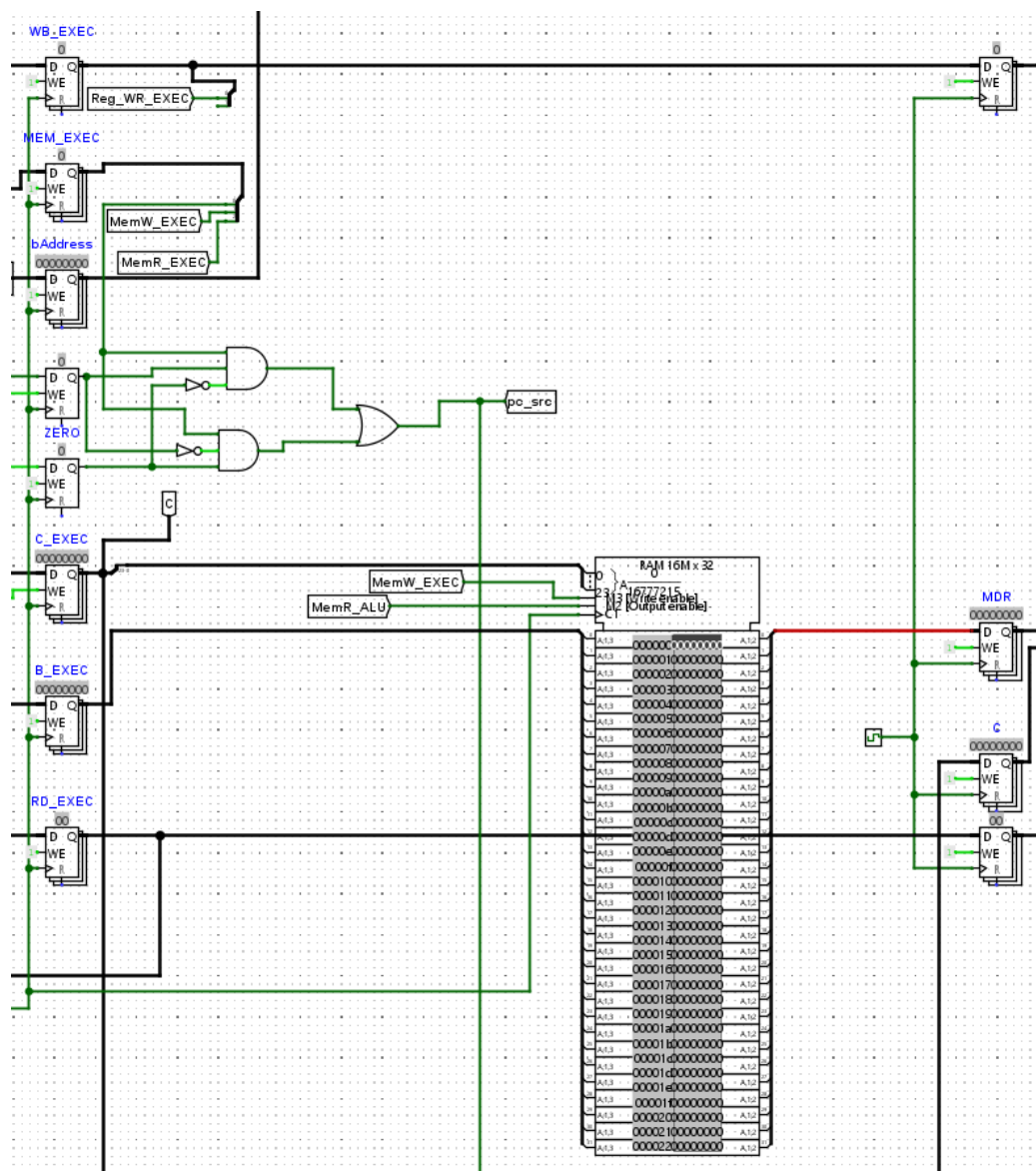
In the execution stage, the WB\_EX and MEM\_EX, and the destination register inside the IMM register are passed to the memory stage registers since they are not utilized here. The EX flags (S) are used in the IGU, along with the IMM register which holds the instruction to extract the immediate. This immediate is then input to a multiplexer controlled by the ALUsrc1 flag, choosing either B or immediate. Additionally, this immediate is added to PC using an adder and stored in the bAddress register, representing  $PC + \text{immediate}$ . The ALU control unit takes op, func3, and func7 from the IMM register and provides the select lines for the ALU. The ALU result is stored in the C\_EXEC register, and the zero bit in the ZERO register. Func3\_EX is also stored in a register since it differentiates between bne and beq. The forwarding unit controls two multiplexers, one in front of A and one in front of B, checking whether it passes A or forwards A from memory or A from writeback. The same is applied to B. The B register is passed on since it will be needed in the next stage. (The immediate for pc is divided by 4 since PC is based on a counter counting 1 by 1)





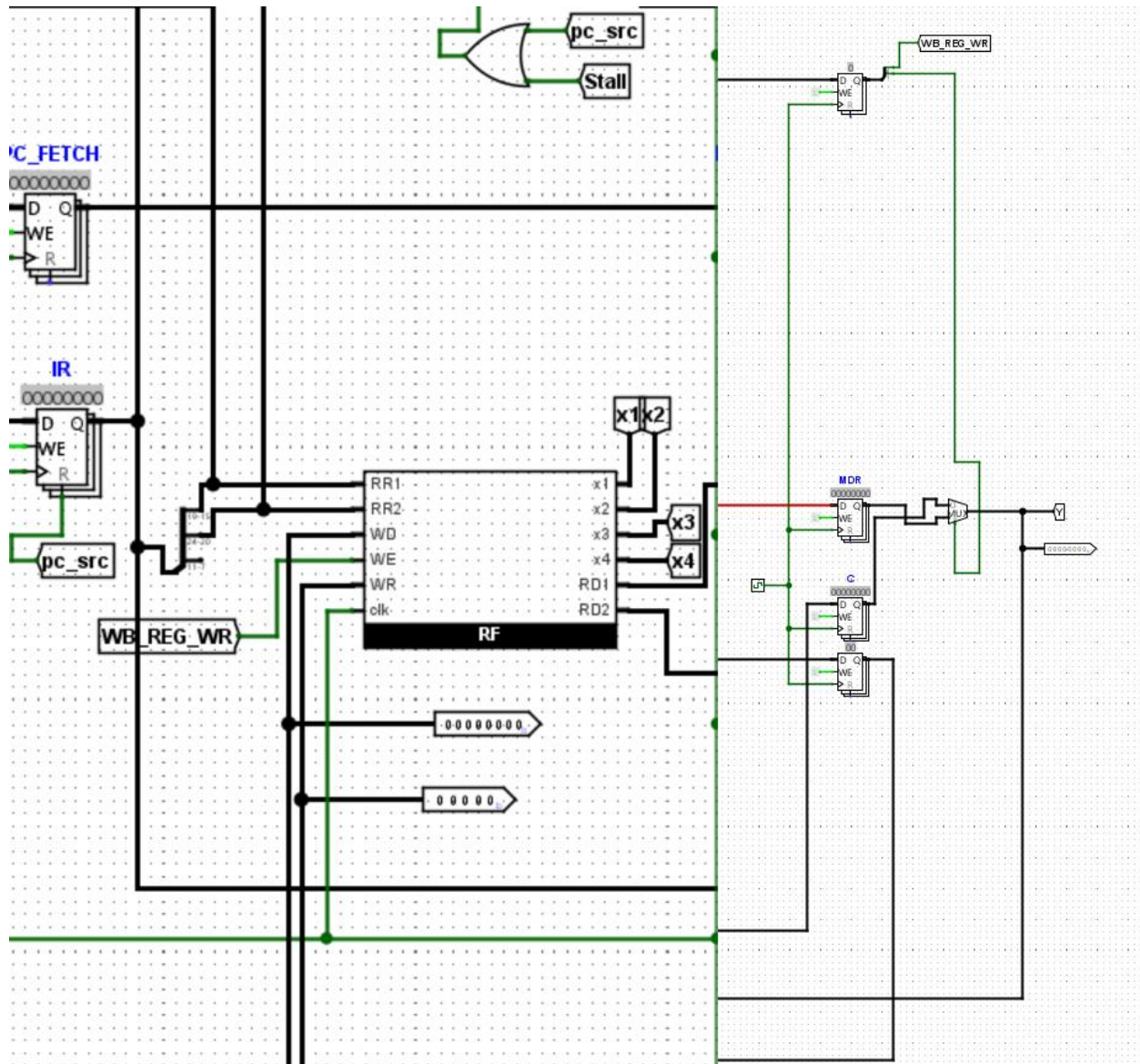
#### 4-Memory Stage:

In the memory stage, the flags MemBr is used along with zero and func3 register to check if we have a branch or not and set the pc\_src flag which will not only flush the pipe but load the new pc in the counter. The memor write and read flags will be used to write the value of B\_EXEC register at the memory location C\_EXEC or read the value from the C\_EXEC memory location and store it in MDR. The C\_EXEC, WB\_EXEC and RD\_EXEC registers are passed on since they are needed in the last stage.



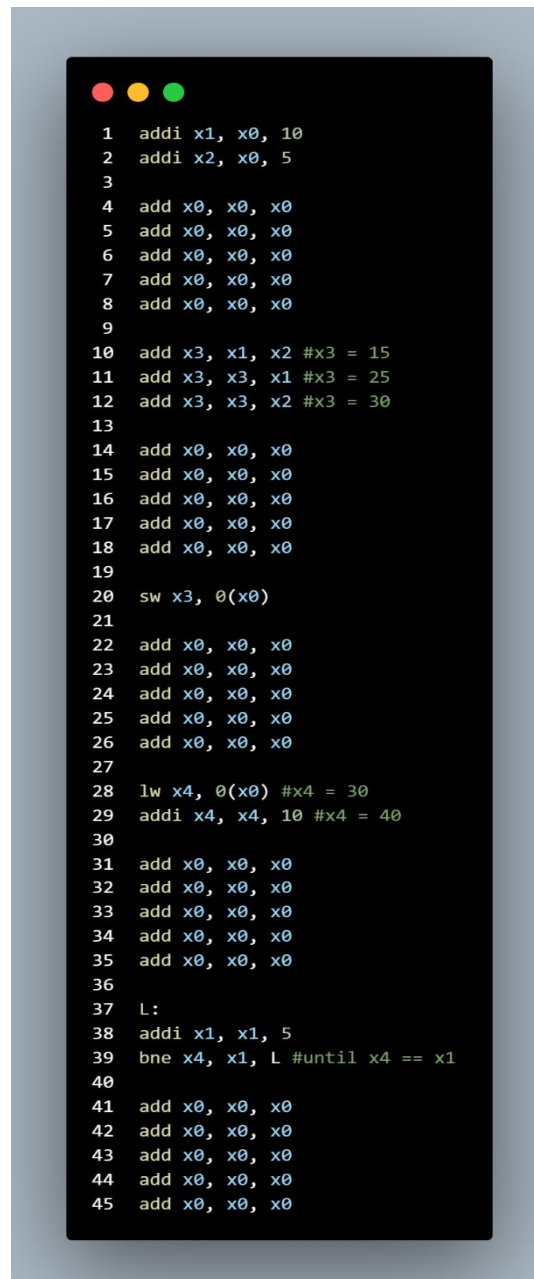
### 5-Write-Back Stage:

In the write-back stage, we use the RegWr flag to enable writing to the register file. The destination register is taken from the register below C. The data to write will be either MDR from the memory or C from the ALU. The MemtoReg flag will choose either MDR or C to be written.



## Simulations:

For simulation the IM in the project contains the following code. This code will test the forwarding unit for dependency, storing and loading from data memory, hazard detection unit for stalling the processor, branching then flushing the processor. (The add x0, x0, x0 instructions serve as a nop instruction)



```

1  addi x1, x0, 10
2  addi x2, x0, 5
3
4  add x0, x0, x0
5  add x0, x0, x0
6  add x0, x0, x0
7  add x0, x0, x0
8  add x0, x0, x0
9
10 add x3, x1, x2 #x3 = 15
11 add x3, x3, x1 #x3 = 25
12 add x3, x3, x2 #x3 = 30
13
14 add x0, x0, x0
15 add x0, x0, x0
16 add x0, x0, x0
17 add x0, x0, x0
18 add x0, x0, x0
19
20 sw x3, 0(x0)
21
22 add x0, x0, x0
23 add x0, x0, x0
24 add x0, x0, x0
25 add x0, x0, x0
26 add x0, x0, x0
27
28 lw x4, 0(x0) #x4 = 30
29 addi x4, x4, 10 #x4 = 40
30
31 add x0, x0, x0
32 add x0, x0, x0
33 add x0, x0, x0
34 add x0, x0, x0
35 add x0, x0, x0
36
37 L:
38 addi x1, x1, 5
39 bne x4, x1, L #until x4 == x1
40
41 add x0, x0, x0
42 add x0, x0, x0
43 add x0, x0, x0
44 add x0, x0, x0
45 add x0, x0, x0

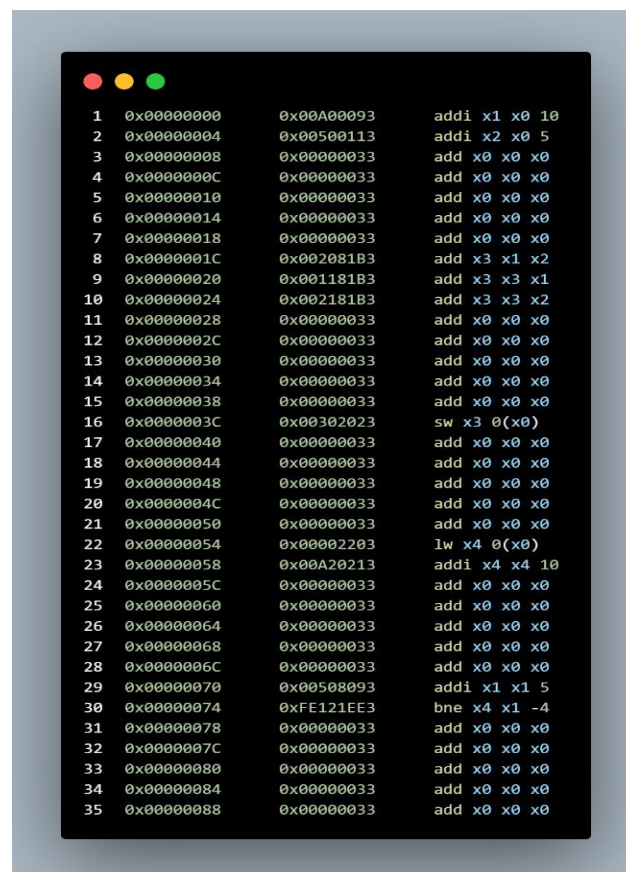
```

To execute this code, each instruction must be converted into its corresponding 32-bit binary representation, following the format of RISC-V instructions.

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Figure 13: RISC-V instruction format

To simplify this process, we utilized a Visual Studio Code extension called "RISC-V Venus Simulator." This extension establishes an environment resembling a RISC-V processor, allowing us to monitor various elements such as assembly code, program counter (PC), registers, memory, and more. When utilizing this extension and navigating to the assembly section, we obtain the following result:



```

1 0x00000000 0x00A00093 addi x1 x0 10
2 0x00000004 0x00500113 addi x2 x0 5
3 0x00000008 0x00000033 add x0 x0 x0
4 0x0000000C 0x00000033 add x0 x0 x0
5 0x00000010 0x00000033 add x0 x0 x0
6 0x00000014 0x00000033 add x0 x0 x0
7 0x00000018 0x00000033 add x0 x0 x0
8 0x0000001C 0x002081B3 add x3 x1 x2
9 0x00000020 0x001181B3 add x3 x3 x1
10 0x00000024 0x002181B3 add x3 x3 x2
11 0x00000028 0x00000033 add x0 x0 x0
12 0x0000002C 0x00000033 add x0 x0 x0
13 0x00000030 0x00000033 add x0 x0 x0
14 0x00000034 0x00000033 add x0 x0 x0
15 0x00000038 0x00000033 add x0 x0 x0
16 0x0000003C 0x00302023 sw x3 0(x0)
17 0x00000040 0x00000033 add x0 x0 x0
18 0x00000044 0x00000033 add x0 x0 x0
19 0x00000048 0x00000033 add x0 x0 x0
20 0x0000004C 0x00000033 add x0 x0 x0
21 0x00000050 0x00000033 add x0 x0 x0
22 0x00000054 0x00002203 lw x4 0(x0)
23 0x00000058 0x00A20213 addi x4 x4 10
24 0x0000005C 0x00000033 add x0 x0 x0
25 0x00000060 0x00000033 add x0 x0 x0
26 0x00000064 0x00000033 add x0 x0 x0
27 0x00000068 0x00000033 add x0 x0 x0
28 0x0000006C 0x00000033 add x0 x0 x0
29 0x00000070 0x00508093 addi x1 x1 5
30 0x00000074 0xFE121EE3 bne x4 x1 -4
31 0x00000078 0x00000033 add x0 x0 x0
32 0x0000007C 0x00000033 add x0 x0 x0
33 0x00000080 0x00000033 add x0 x0 x0
34 0x00000084 0x00000033 add x0 x0 x0
35 0x00000088 0x00000033 add x0 x0 x0

```

This includes the program counter, the 32-bit binary representation of the instruction, and its corresponding instruction in assembly language. This streamlined the process of populating

our instruction memory, as we could effortlessly convert any instruction into its binary equivalent.

## Conclusion:

In summary, the processor crafted in this project stands as a fully functional and practical solution ready for implementation in diverse projects. Its successful design and functionality underscore the project's effectiveness in contributing a viable processor solution for various applications.

## Future Work:

- Instead of using a counter for the program counter (PC) and a divider to divide the branching immediate, we can employ a register and an adder that increments the PC by 4 each cycle. Now, for branching, a multiplexer chooses between  $PC + 4$  or  $PC + \text{immediate}$ . This solution is less hardware-intensive and represents a more efficient implementation.
- Instead of utilizing a RAM for data memory with a 1-cycle delay for reading, we can opt to create our own RAM from registers or leverage a ROM. This alternative solution will eliminate the 1-cycle delay for reading, offering a more efficient implementation.
- Instead of deferring the check for a taken branch until the memory stage, we can incorporate a counter that increments every time a branch is taken and decrements otherwise. This counter can have four states, ranging from 00 (strongly not taken) to 11 (strongly taken). Given that branches are often employed for loops, this counter can expedite the decision-making process, allowing us to determine whether to take the branch directly from the decode stage. This approach not only enhances efficiency but also eliminates the need for pipeline flushing, saving two cycles per occurrence.