HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

DESIGN A RISC STORED-PROGRAM MACHINE

A report on Digital Design Using VHDL Project

Group 2

Students :	Ngô Văn Cảnh	20193204
	Phùng Mạnh Dũng	20193210
	Nguyễn Danh Khuê	20193224

Advisor : Prof. Võ Lê Cường

JANUARY 30, 2023
HUST

Table of Contents

I. Introduction and Overview 3
II. Work distribution 4
III. Design Hierarchy 4
IV. Hardware composition 4
1. Processing Unit 6
a. Arithmetic Logic Unit 6
b. Register Unit 6
c. Multiplexer 6
2. Control Unit 7
a. Function of the control unit 7
b. Control Signals 7
c. Instruction Set 7
d. Controller States 9
e. State transition diagram 10
f. ASM chart 10
3. Memory Unit 16
V. Verilog implementation 16
VI. Design Verification 16
1. Register Unit testbench 17
2. Program Counter testbench 18
3. Arithmetic Logic Unit testbench 19
4. Multiplexer testbench 20
5. Full Module Testbench 21
VII. Conclusion 23
VIII. Reference 23

Table of Figure

Figure 1 Execution time 3
Figure 2 Hierarchy Table 4
Figure 3 Architecture of RISC-SPM 5
Figure 4 State transition diagram 10
Figure 5 ASM chart 11
Figure 6 NOP, ADD, SUB, AND, NOT 12
Figure 7 RD 13
Figure 8 WR 14
Figure 9 BR, BRZ 15
Figure 10 Register testbench waveform 17
Figure 11 Program Counter testbench waveform 18
Figure 12 ALU testbench waveform 19
Figure 13 Mux_3_1_tb waveform 20
Figure 14 Full Module waveform 1 22
Figure 15 Full Module waveform 2 22

I. Introduction and Overview

Reduced instruction-set computers (RISC) are designed to have a **small set of instructions**, which mean a **large number of instructions** per program that execute in **short clock cycles** per instruction. RISC machines are optimized to achieve efficient pipelining of their instruction streams. The machine also serves as a starting point for developing architectural variants and a more robust instruction set.

$$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} X \frac{Cycles}{Instructions} X \frac{Seconds}{Cycle}$$

Figure 1 Execution time

As opposite to RISC is the CISC (Complex Instruction Set Computer) architecture which have a **large set of instructions**, which will **minimize number of instructions** per program but at the cost of an **increase in the number of cycles** per instruction. The designers have to make the tradeoffs to selecting an architecture that serves an application. Table 1 below is all the differences of two architecture. Once an architecture has been selected, a circuit that has sufficient performance (speed) must be synthesized. Hardware description languages (HDLs) play a key role in this process by modeling the system and serving as a descriptive medium that can be used by a synthesis tool.

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

Table 1 RISC vs. CISC

Our design is a modified version from the reference book that can be found at the end of this text. Though the book explained pretty clear about this topic, I'm still going to discuss as manything as posible in my word. The book even contains a full verilog implementation code already, but as we "copy" and try to run it, a few bugs still found. So we have spent quite a lot of time to fix and optimize the code.

Check out this github link for the source code : https://github.com/canh25xp/RISC-SPM

II. Work distribution

	Cảnh	Khuê	Dũng
Leader	X		
Researching	X	X	X
Code Writer and simulation	X		
Testing and Fixing		X	X
Report			X
Presentation	X	X	X

III. Design Hierarchy

We using the Top-Down methodology to design the system. Which mean the top-level block (module) is define first, all the sub-blocks (instances) necessary to build the top-level is define later. The hierarchy tree is shown in the *figure 2*

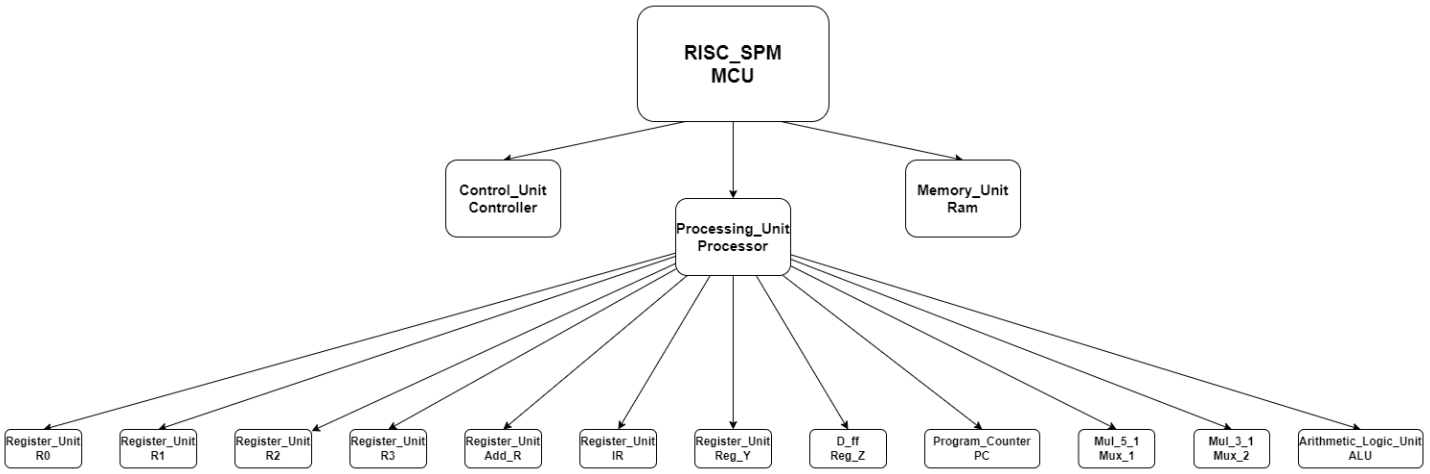

Figure 2 Hierarchy Table

As you can see in the diagram. The top-block is the module RISC-SPM, it called out three sub-block instances, which is the Control Unit, Processing Unit and Memory_Unit. The Processing Unit is then call out 7 Register Unit, a D-FlipFlop, a Program Counter, 2 Multiplexer and a Arithmetic Logic Unit.

IV. Hardware composition

RISC-SPM or Reduced Instruction Set Computer Store Program Machine consists of three functional units :

- 1.Processing Unit (Processor)
- 2.Controll Unit (Controller)
- 3.Memory Unit (RAM)

The Overall Architecture of the RISC-SPM is shown in *figure 3*.

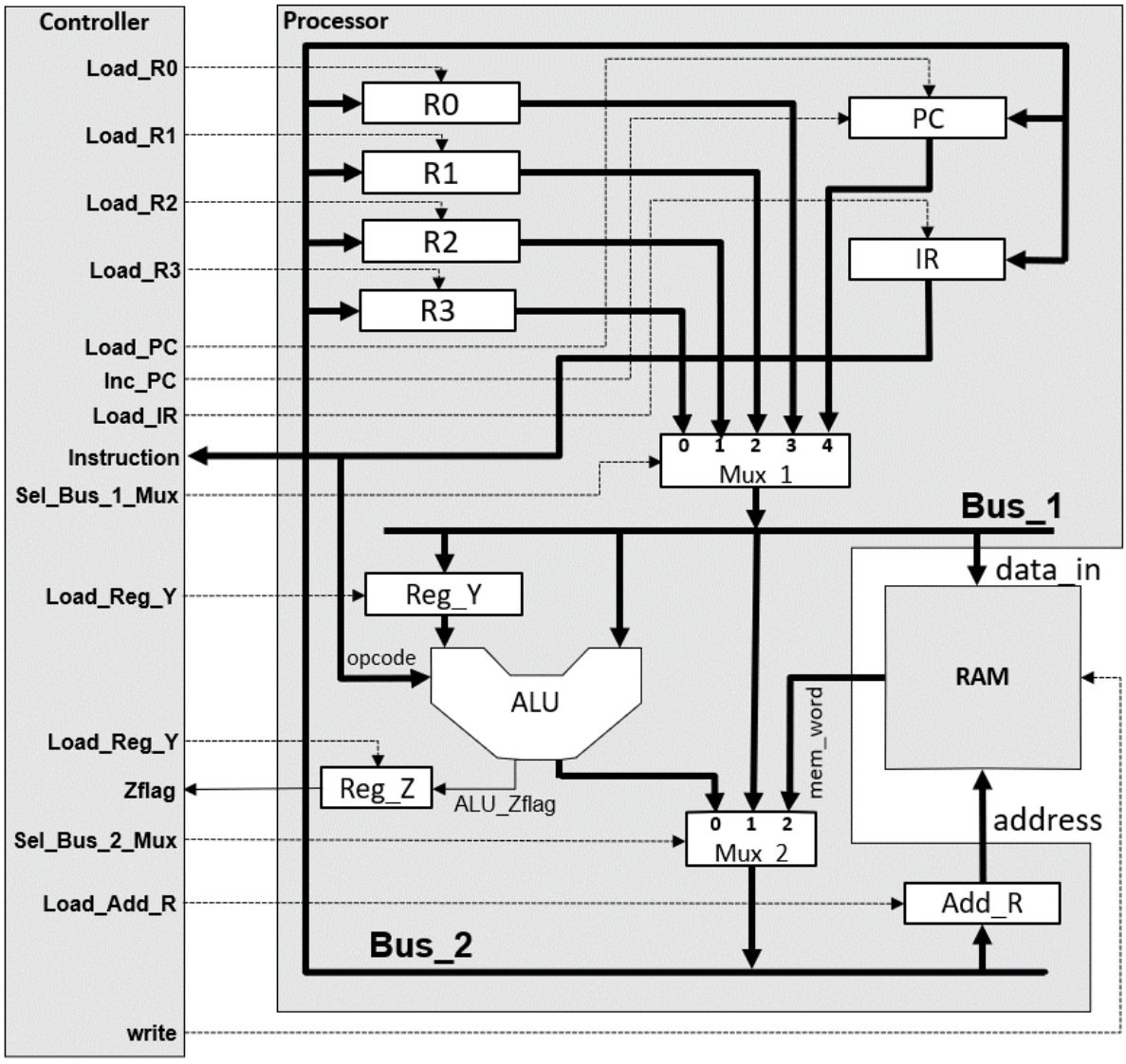

Figure 3 Architecture of RISC-SPM

Program instructions and data are stored in memory

Instructions are fetched from memory synchronously, decoded and executed to :

- Operate on data with ALU
- Change the contents of storage registers
- Change the content of the program counter (PC), instruction register (IR) and the address register (ADD_R)
- Change the content of memory
- Retrieve data and instructions from memory
- Control the movement of data on the system busses

The Program Counter (PC) contains the address of the next instruction to be executed

The Instruction Register (IR) contains the instruction that currently being executed

The address register (Add_R) contains the address of the memory location that will be addressed next by a read or write operation.

1. Processing Unit

The processor includes registers, buses, control lines, and an ALU capable of performing arithmetic and logic operations on its operands depends on the opcode held in the instruction register. Its take control from the Control_Unit, manipulate the Registers and read/write data to Memory_Unit.

a. Arithmetic Logic Unit

For the purposes of this example, the ALU has two operand datapaths, data_1 and data_2, and its instruction set is limited to only 4 instructions, that is :

Opcode	Action
ADD	Adds the datapaths to form data_1 + data_2
SUB	Subtracts the datapaths to form data_1 - data_2
AND	Takes the bitwise and of the datapaths data_1 & data_2
NOT	Takes the bitwise Boolean complement of data_1

The ALU take the output of Reg_y is input for data_in and Bus_1 as input for data_2.

The Output result of the ALU is go to the Mux_2 and the zero flag bit is go to input of the Reg_Z. Note that the zero flag bit is set (bit 1) when the ALU result is equals to 0.

b. Register Unit

There are 9 registers in our design :

- 5 general-purpose registers R0, R1, R2, R3, Reg_y (8-bit)
- 3 special purpose register PC, IR, Add_R (8-bit)
- 1 flag register Reg_Z (1-bit)

All registers have a load signal to store data, a clock signal (clk) to synchronize and a reset signal (rst) to erase data (all bits are set to 0). Note that the **reset signal is active low**, which mean it'll reset when signal is low (0)

The zero flag register Reg_Z is a 1-bit register, so basically it is a D flip flop.

The Program Counter Register (PC) has an additional signal Inc_PC to increase PC by 1 unit.

c. Multiplexer

There are 2 multiplexers in the Processing Unit :

- Mux_1 : it's a 5-1 multiplexer
  - Output : Bus_1
  - Input : R0, R1, R2, R3, PC
  - Control input : Sel_Bus_Mux_1 (3 bits)
- Mux_2 : it's a 3-1 multiplexer
  - Output : Bus_2
  - Input : ALU's output, Bus_1
  - Control input : Sel_Bus_Mux_1 (2 bits)

An instruction can be fetched from memory, placed on Bus_2, and loaded into the instruction register. A word of data can be fetched from memory, and steered to a general-purpose register or to the operand register (Reg_Y) prior to an operation of the ALU. The result of an ALU operation can be placed on Bus_2, loaded into a register, and subsequently transferred to memory. A dedicated register (Reg_Z) holds a flag indicating that the result of an ALU operation is 0.

2. Control Unit

The Control Unit is a FSM (Finite State Machine), or more specifically, a Mealy Machine. Because the output (The control signals, which will be discuss later) of it depends on the state and external input (instruction, Zflag, rst)

a. Function of the control unit

- Determine when and which registers to be load
- Select the path of data through the multiplexers
- Determine when data should be written to memory
- Control the three-state busses in the architecture.

b. Control Signals

There are 13 output signals of the controller, 1 instruction input, 1 zero flag input and a reset signal input. The Action of each control signal is describe as below.

Control Signal	Action
Load_Add_R	Loads the Address Register
Load_PC	Loads Bus_2 to the Program Counter
Load_IR	Loads Bus_2 to the Instruction Register
Inc_PC	Increments the Program Counter
Sel_Bus_1_Mux	Selects among the Program Counter, R0, R1, R2, and R3 to drive Bus_1
Sel_Bus_2_Mux	Selects among ALU_out, Bus_1, and memory to drive Bus_2
Load_R0	Loads general purpose register R0
Load_R1	Loads general purpose register R1
Load_R2	Loads general purpose register R2
Load_R3	Loads general purpose register R3
Load_Reg_Y	Loads Bus_2 to the register Reg_Y
Load_Reg_Z	Stores the zero Flag of ALU in register Reg_Z
write	Loads Bus_1 into the memory

c. Instruction Set

A machine language program consists of a stored sequence of 8-bit words (bytes). The format of an instruction of RISC_SPM can be long or short, depending on the operation :

- Short instructions : requires 1 byte of memory to specifies 4-bit opcode, 2-bit source register address, a 2-bit destination register address.
- Long instruction : requires 2 bytes of memory. The first word of a long instruction contains a 4-bit opcode. The remaining 4 bits of the word can be used to specify addresses of a pair of source and destination registers, depending on the instruction. The second word contains the address of the memory word that holds an operand required by the instruction.

Opcode				Dst		Src	
0	0	1	0	0	1	1	0

Table 2 Short instruction format

Opcode				Dst		Src		Address							
0	1	1	0	x	x	1	0	0	0	0	1	1	1	0	1

Table 3 Long instruction format

The instruction mnemonics and their actions are listed below.

	Action
Short instruction	
NOP	No operation is performed; all registers retain their values. The addresses of the source and destination register are don't-cares, they have no effect.
ADD	Adds the contents of the source and destination registers and stores the result into the destination register.
SUB	Subtracts the content of the source register from the destination register and stores the result into the destination register.
AND	Forms the bitwise and of the contents of the source and destination registers and stores the result into the destination register.
NOT	Forms the bitwise complement of the content of the source register and stores the result into the destination register.
HALT	Halts execution until reset
Long instruction	
RD	Reads a word from the location specified by the second byte and loads the result into the destination register. The source register bits are don't-cares.
WR	Writes the contents of the source register to the word in memory specified by the address held in the second byte. The destination register bits are don't-cares.
BR	Branches the activity flow by loading the program counter with the word at the address specified by the second byte of the instruction. The source and destination bits are don't-cares.
BRZ	Branches if the zero flag register is asserted.

The RISC_SPM instruction set is summarized below.

Ins	Opcode	Dst	Src	Action
NOP	0000	xx	xx	none
ADD	0001	dst	src	dst <= src + dst
SUB	0010	dst	src	dst <= dst - src
AND	0011	dst	src	dst <= src && dst
NOT	0100	dst	src	dst <= ~ src
RD	0101	dst	xx	dst <= memory [Add_R]
WR	0110	xx	src	memory[Add_R] < = src
BR	0111	xx	xx	PC <= memory[Add_R]
BRZ	1000	xx	xx	PC <= memory[Add_R]
HALT	1111	xx	xx	Halts execution until reset (Finish programm)

d. Controller States

Each instruction has three phases : fetch, decode, and execute.

- Fetching : Retrieves an instruction from memory . Its takes **2 clock cycles**, one to load the address register and one to retrieve the addressed word from memory.
- Decoding : Decodes the instruction, manipulates datapaths ,and loads registers. Its takes **1 clock cycle**
- Execution : Generates the results of the instruction. Its might take 0, 1 or 2 clock cycles, depends on the instruction :
    - NOP : 0 clock
    - ADD, SUB, AND, NOT : 1 clock
    - RD, WR, BR, BRZ : 2 clocks (the BRZ instruction might take 0 instruction if the zero flag not set)

Below is the table of 11 states and the description of each state

State	Action
idle	State entered after reset is asserted. No action.
fet1	Load the Add_R with the contents of the PC. (Note: PC is initialized to the starting address 00H by the reset action.) The state is entered at the first active clock after reset is de-asserted, and is revisited after a NOP instruction is decoded.
fet2	Load the IR with the word addressed by the Add_R, and increment the PC to point to the next location in memory, in anticipation of the next instruction or data fetch.
dec	Decode the IR and assert signals to control datapaths and register transfers.
exe	Execute the ALU operation for a single-byte instruction, conditionally assert the zero flag, and load the destination register.
rd1	Load the Add_R with the second byte of a RD instruction, and increment the PC.
rd2	Load the destination register with the memory word addressed by the byte loaded in rd1.
wr1	Load the Add_R with the second byte of a WR instruction, and increment the PC.
wr2	Load the source register with the memory word addressed by the byte loaded in wr1.
br1	Load the Add_R with the second byte of a BR instruction, and increment the PC.
br2	Load the PC with the memory word addressed by the byte loaded in br1.
halt	Default state to trap failure to decode a valid instruction.

e. State transition diagram

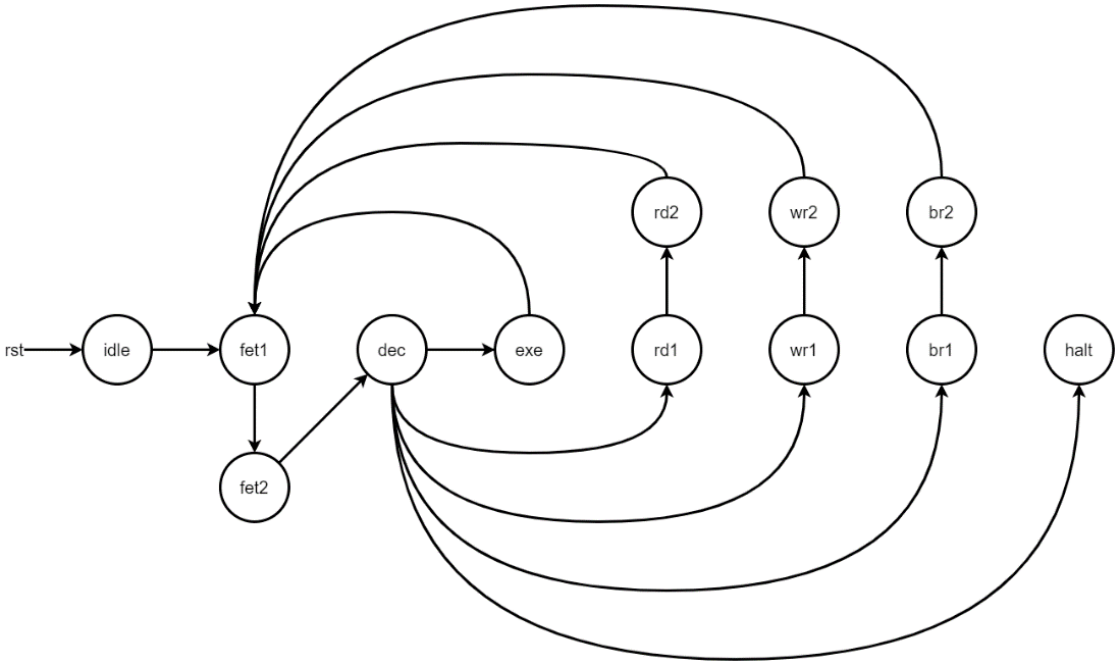

Figure 4 State transition diagram

Figure 4 is a “State transition diagram”, do not misunderstanding it with “State diagram”, though it might looks like it, it is not. It mearly show how many states are there in the fsm and the transition between them. The ASM chart will give us a full detail about the fsm.

f. ASM chart

A full ASM chart is shown in *figure 5*. Later on, I with break down each part of it. Some thing to note here, is that the controller has too many output signals to put on the diagram, so I will only mention about the set (1) signal, other signals is remain as reset (0).

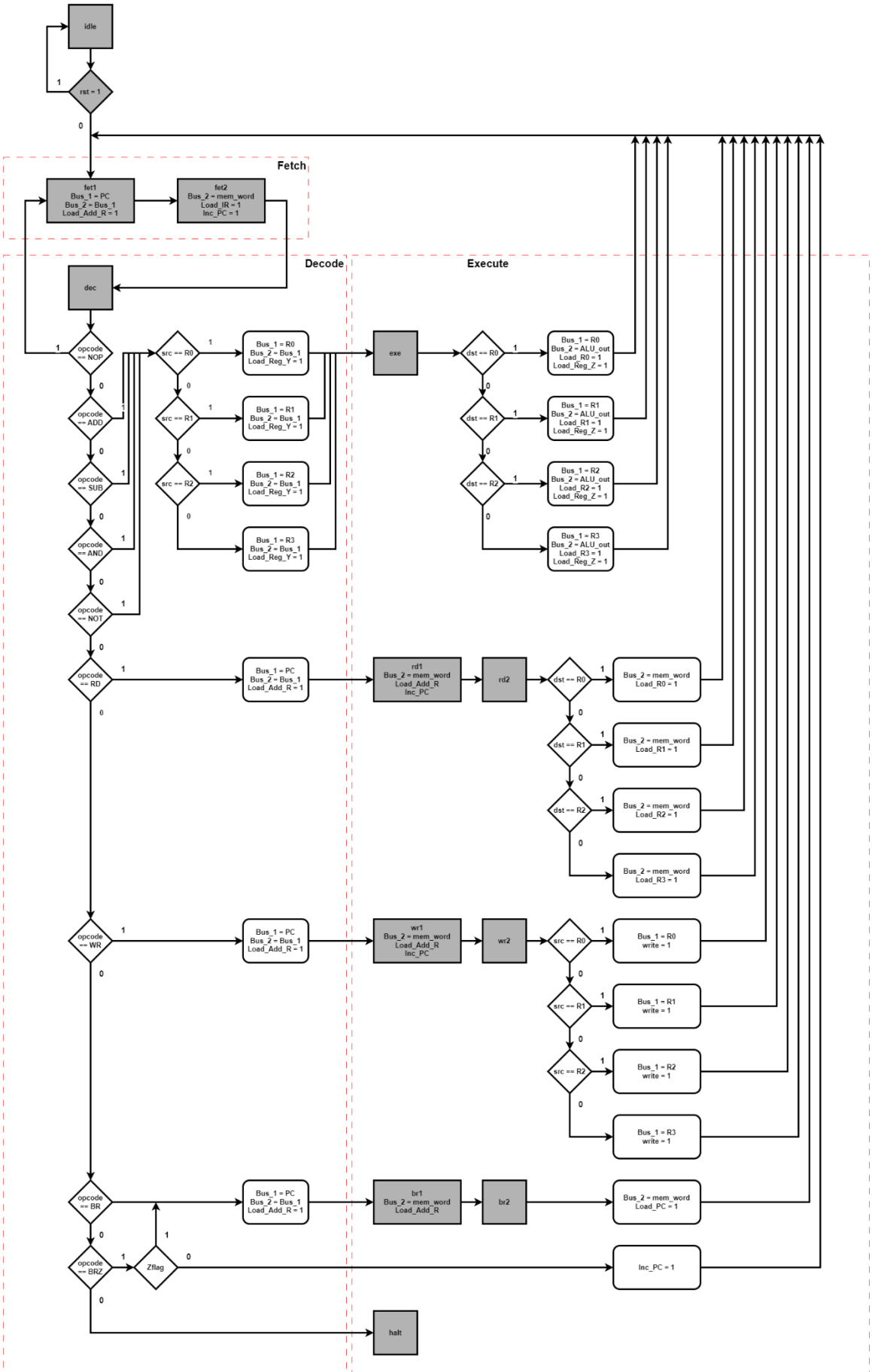

Figure 5 ASM chart

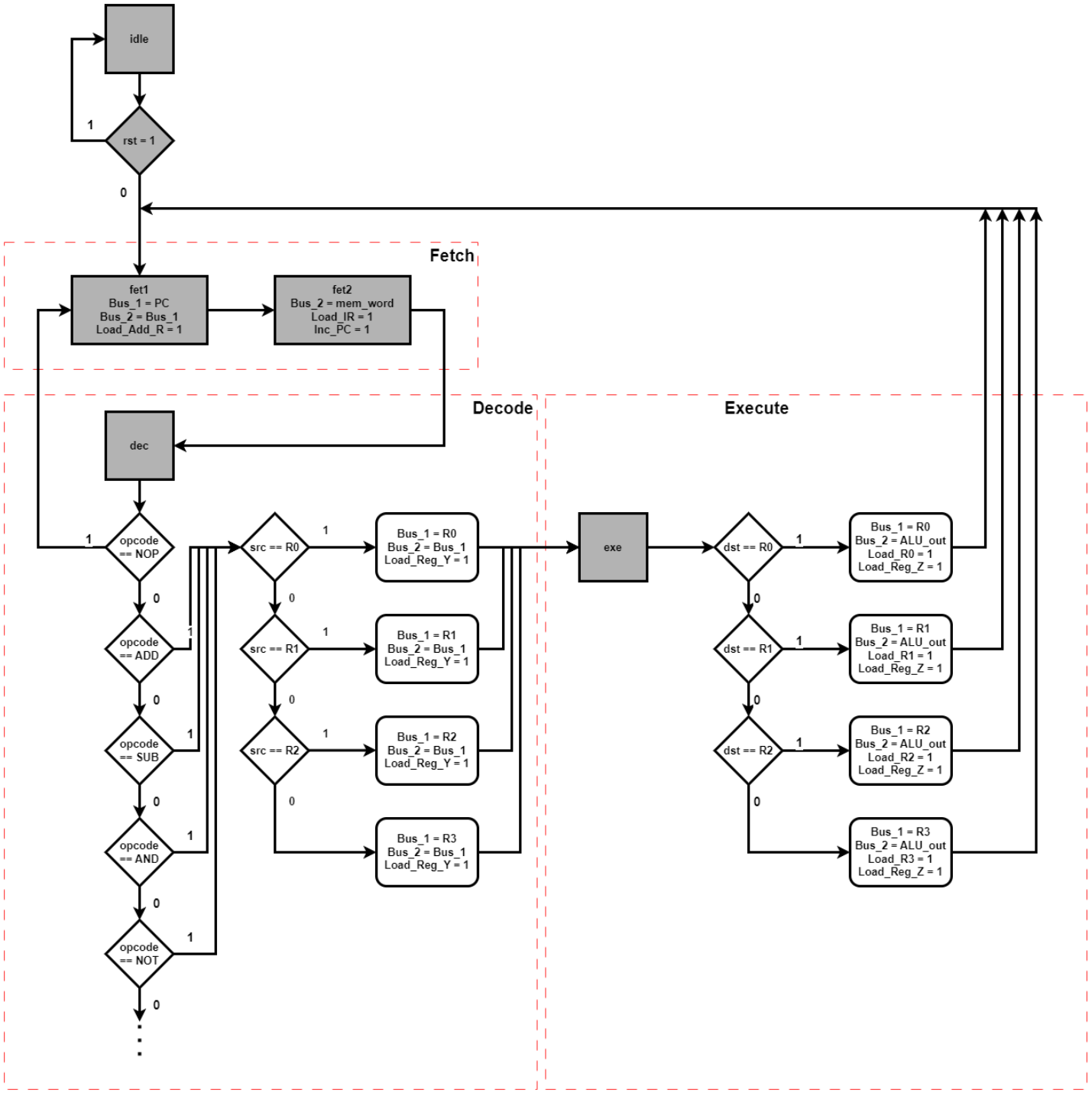

Figure 6 NOP, ADD, SUB, AND, NOT

Figure 6 is the ASM diagram for implementing all the arithmetic operation : ADD, SUB, AND, NOT.

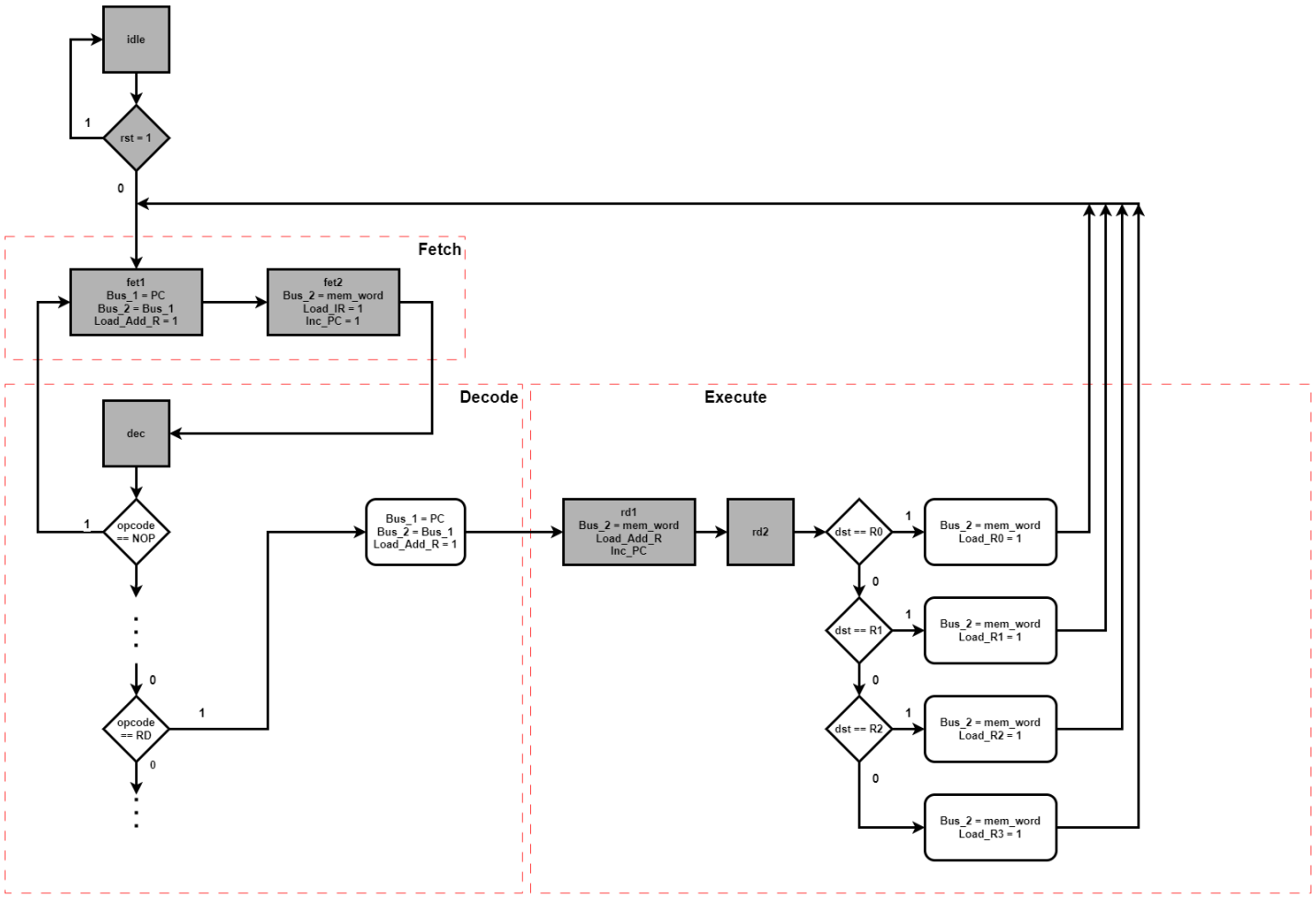

Figure 7 RD

Figure 7 is the ASM diagram for the read instruction.

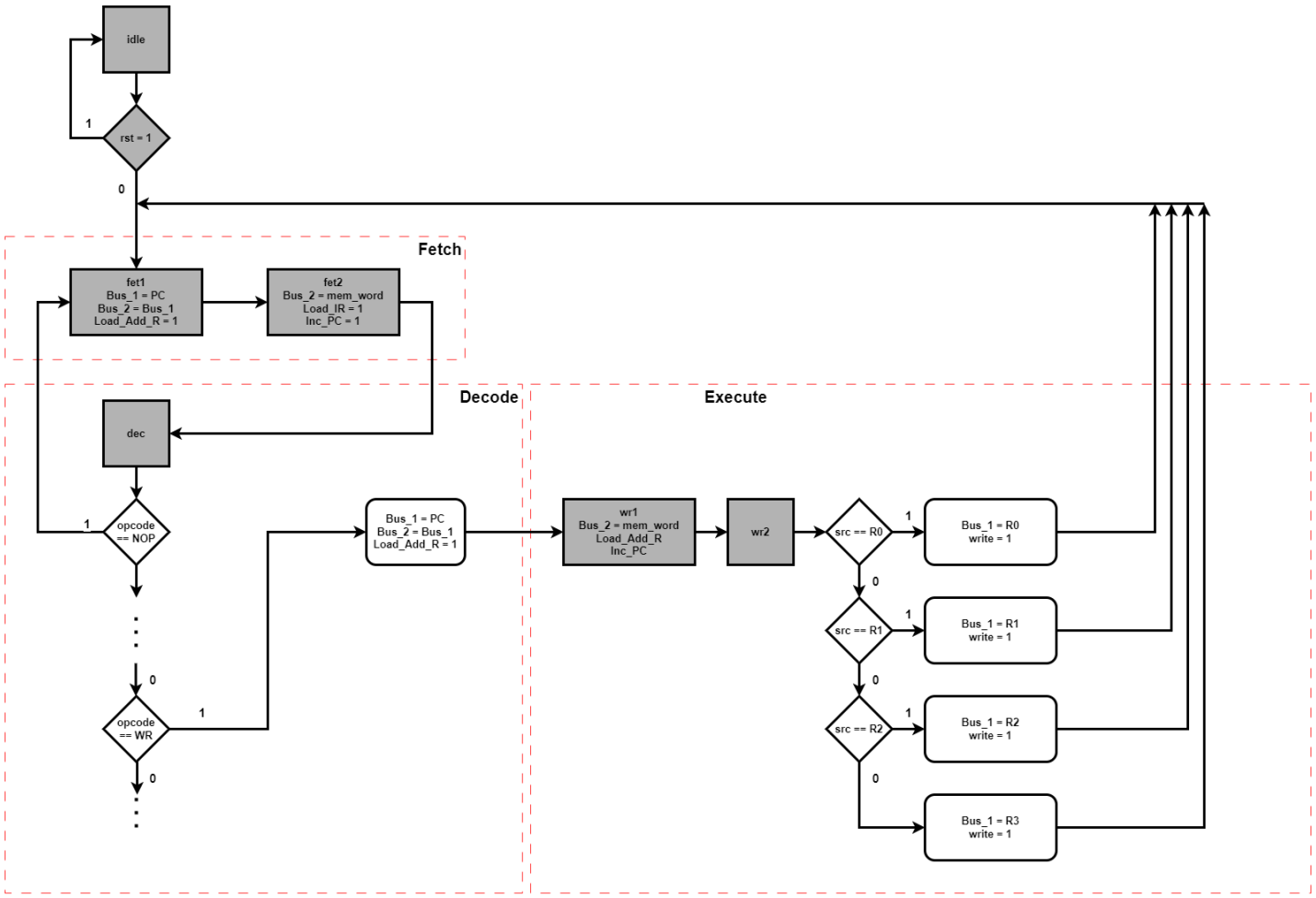

Figure 8 WR

Figure 8 is the ASM diagram for the Write instruction.

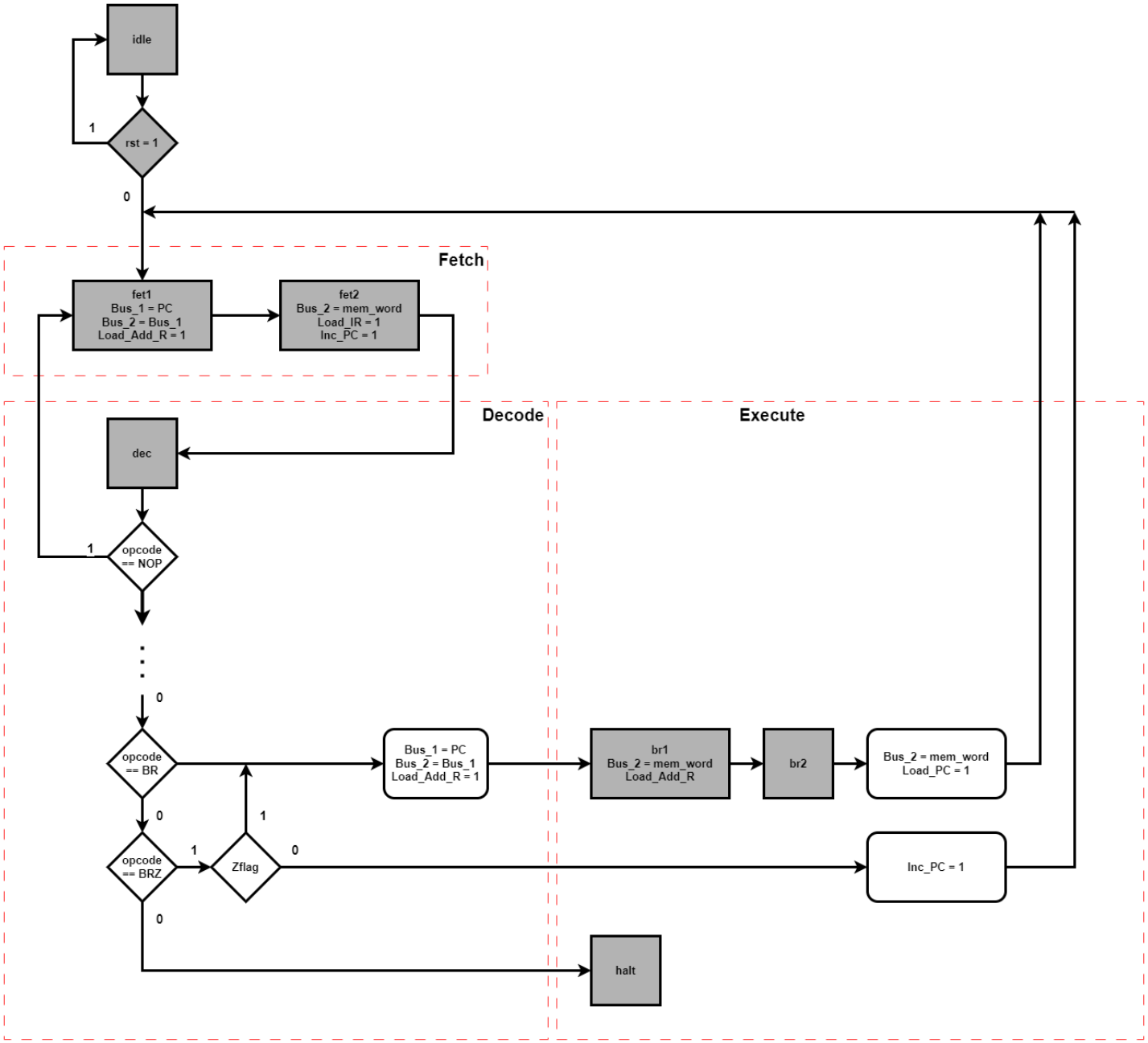

Figure 9 BR, BRZ

Figure 9 is the ASM diagram for the BR and BRZ instruction.

3. Memory Unit

For simplicity, the memory unit of the machine is modeled as an array of D flip-flops that form a **256 bytes** RAM.

This RAM (Random Access Memory) receiving an 8-bit address and output the data stored in the corresponding address. It also have an 8-bit input data. When the rising edge of the write signal is triggered, the input data is written to the corresponding position of the address.

There are no ROM (Read-Only Memory) in this design.

V. Verilog implementation

All the verilog source codes is located in the source code folder or in my github website : https://github.com/canh25xp/RISC-SPM

VI. Design Verification

To ensure the working of the machine, each module has it own testbench : Memory Unit, Control Unit, Register Unit, Arithmetic Logic Unit.

1. Register Unit testbench

```
module Register_Unit_tb;
    reg clk,rst, load;
    reg [7:0] data_in = 8'b00110011;
    wire [7:0] data_out;
    Register_Unit Test_Register(data_out, data_in, load, clk, rst);
    initial clk=1'b0;
    always #5 clk=~clk;
    initial begin
            rst=1'b1; load=0'b0;
        #17 load=1'b1;
        #21 load=0'b0;
        #15 data_in = 8'b11001100;
        #3  load=1'b1;
        #5  load=0'b0;
        #10 rst=0'b0;
        #10 $stop;
    end
endmodule
```

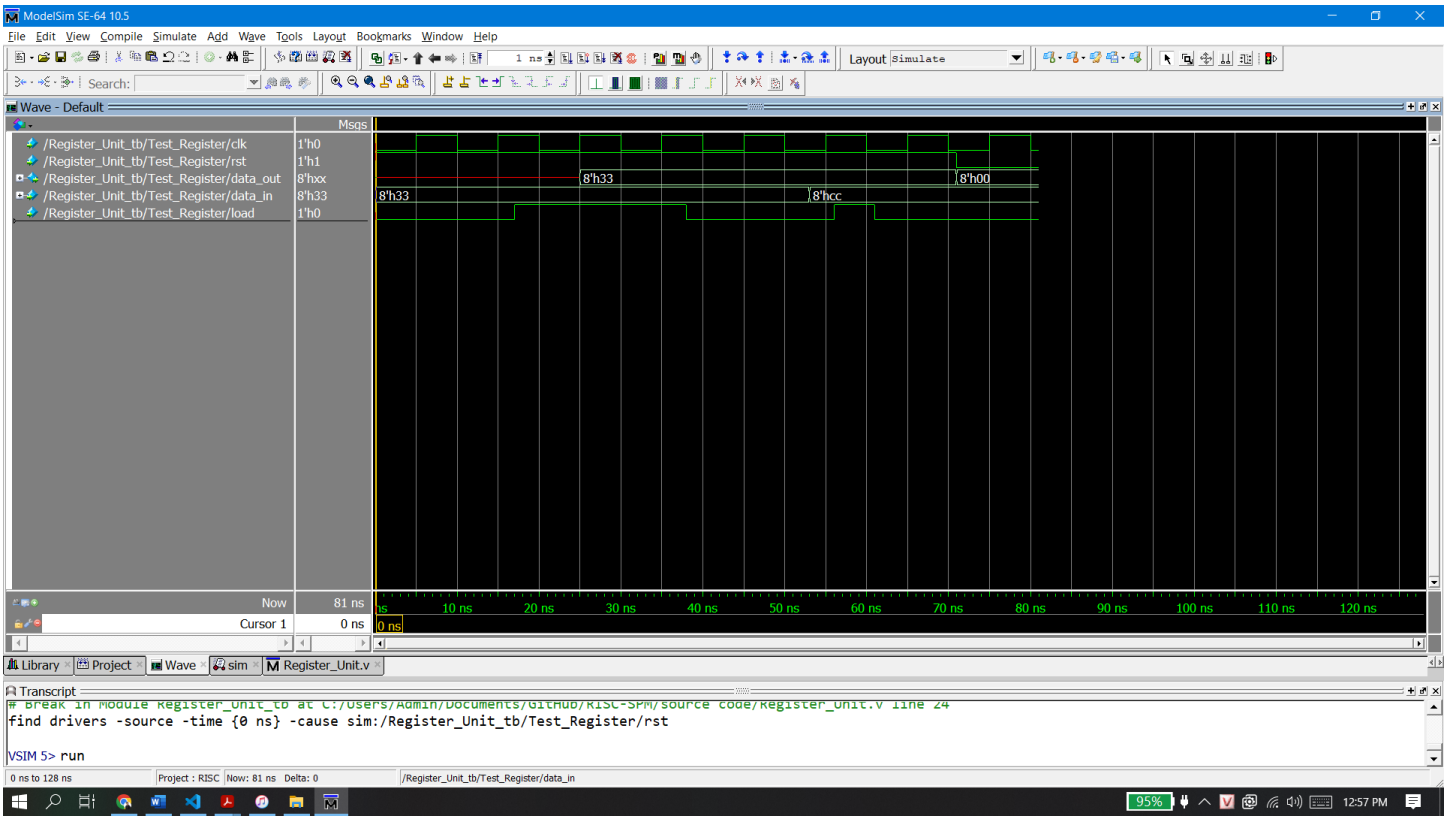

Figure 10 Register testbench waveform

2. Program Counter testbench

```
module Program_Counter_tb;
    reg clk,rst, Load_PC, Inc_PC;
    reg [7:0] data_in;
    wire[7:0] count;
    Program_Counter Test_Counter(count, data_in, Load_PC, Inc_PC, clk, rst);
    initial clk=1'b0;
    always #5 clk=~clk;
    initial begin
                rst=1;Load_PC=0;
        #5      data_in=8'b00000001;
        #5      Load_PC=1;
        #10     Load_PC=0;
        #20     Inc_PC=1;
        #100    Inc_PC=0;
        #5      rst=0;
        #5      rst=1;
        #5      data_in=8'b10000000;
        #5      Load_PC=1;
        #10     Load_PC=0;
    end
endmodule
```

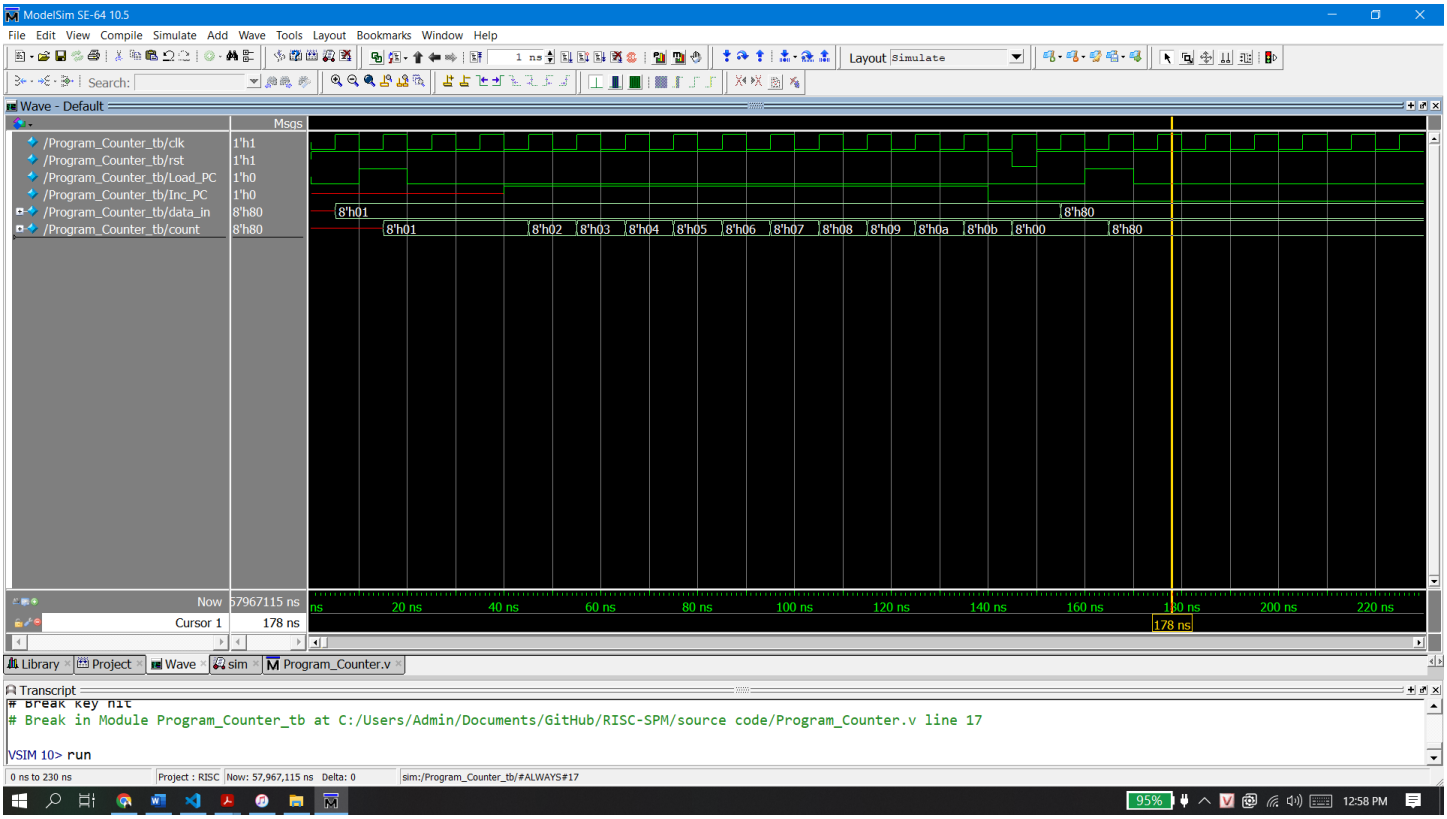

Figure 11 Program Counter testbench waveform

3. Arithmetic Logic Unit testbench

```
module Arithmetic_Logic_Unit_tb;
    reg [7:0] data_1, data_2;
    reg [3:0] opcode;
    wire[7:0] out;
    wire      Zflag;
    Arithmetic_Logic_Unit ALU(out, Zflag, data_1, data_2, opcode);
    initial begin
        #0 data_1 = 64; data_2 = 128; opcode = `NOP;
        // 64 =  01000000
        // 128 = 10000000
        #5 opcode = `ADD; //result should be 192
        #5 opcode = `SUB; //result should be 64
        #5 opcode = `AND; //result should be 0
        #5 opcode = `NOT; //result should be 127
    end
endmodule
```

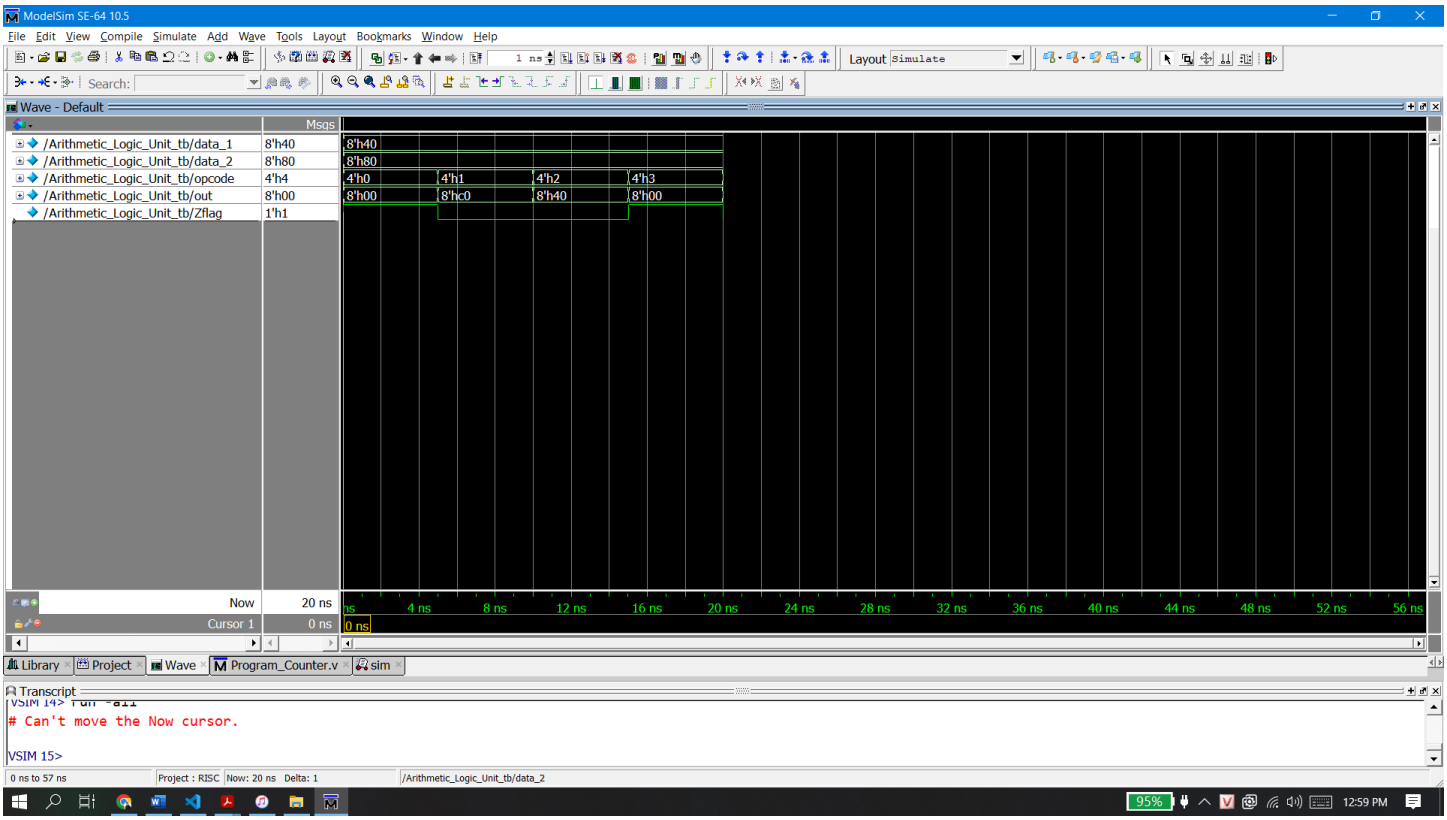

Figure 12 ALU testbench waveform

4. Multiplexer testbench

```
module Mux_3_1_tb;
    reg [7:0] in0, in1, in2;
    reg [1:0] sel;
    wire [7:0] out;
    Mux_3_1 Test_Mux (out, in0, in1, in2, sel);
    initial begin
        in0 = 0; in1 = 128; in2 = 255;
        #5 sel = 2'b00;
        #5 sel = 2'b01;
        #5 sel = 2'b10;
        #5 sel = 2'b11;
    end
endmodule
```

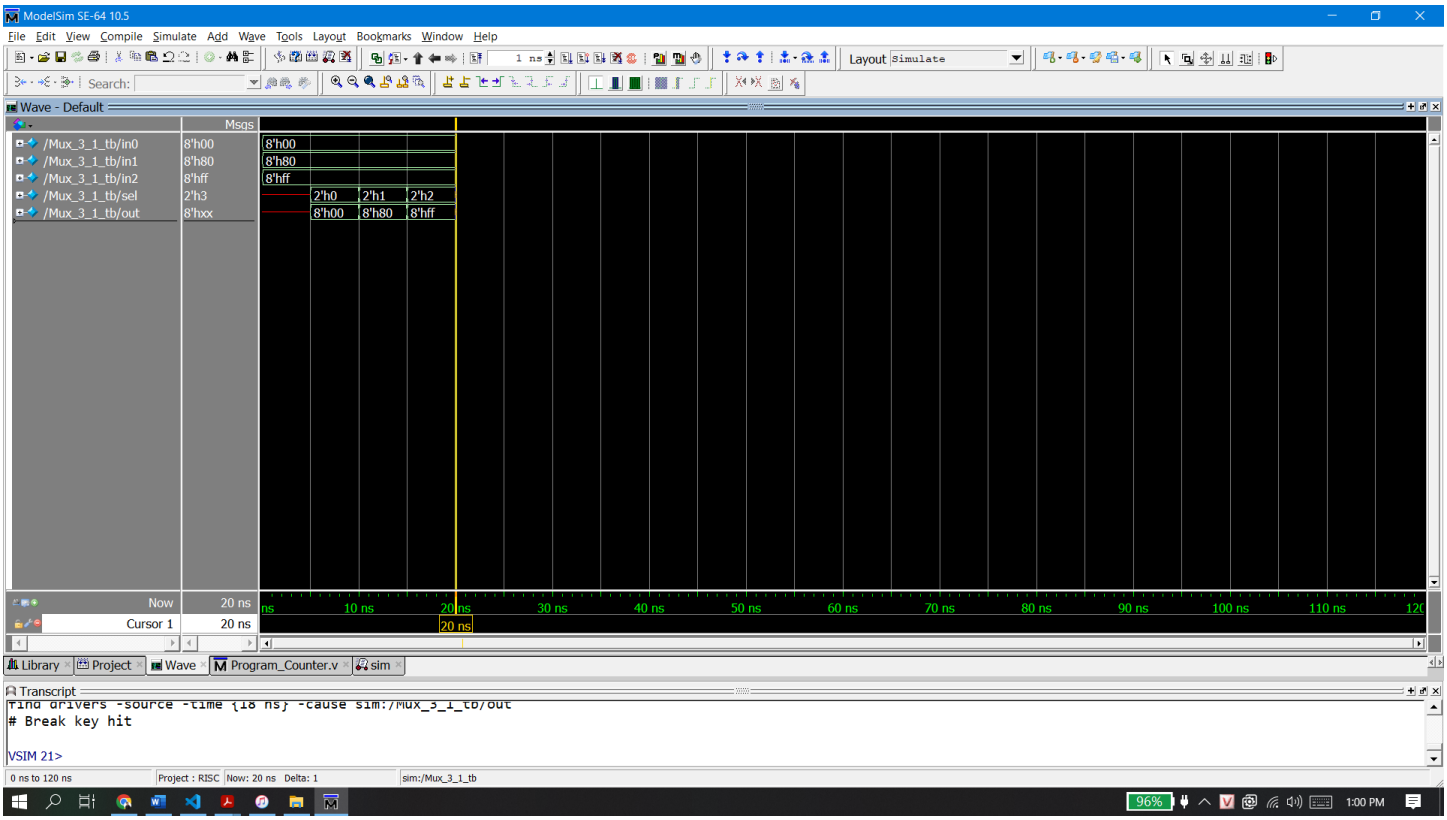

Figure 13 Mux_3_1_tb waveform

5. Full Module Testbench

```
module RISC_SPM_tb;
    reg rst, clk;
    reg[8:0]i;

    RISC_SPM MCU (clk,rst);

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        #0 rst=0;
            for(i=0;i<=255;i=i+1)
                MCU.Ram.memory[i]=0;
        #10 rst=1;
    end

    initial begin
        #5
        MCU.Ram.memory[0]= 8'b0000_00_00;  //NOP
        MCU.Ram.memory[1]= 8'b0101_01_00;  //R1 = memory[128] = 6
        MCU.Ram.memory[2]= 128;
        MCU.Ram.memory[3]= 8'b0101_00_00;  //R0 = memory[129] = 1
        MCU.Ram.memory[4]= 129;
        MCU.Ram.memory[5]= 8'b0010_01_00;   //R1 = R1 - R0
        MCU.Ram.memory[6]=8'b1000_00_00;   //BRZ to memory[130] = 10 ( HALT )
        MCU.Ram.memory[7]=130;
        MCU.Ram.memory[8]= 8'b0111_00_11;  //BR to memory[131] = 5
        MCU.Ram.memory[9]= 131;
        MCU.Ram.memory[10]=8'b1111_00_00;  //HALT

        //Load data
        MCU.Ram.memory[128]=6;
        MCU.Ram.memory[129]=1;
        MCU.Ram.memory[130]=10;
        MCU.Ram.memory[131]=5;
    end

    initial #2800 $stop;
endmodule
```

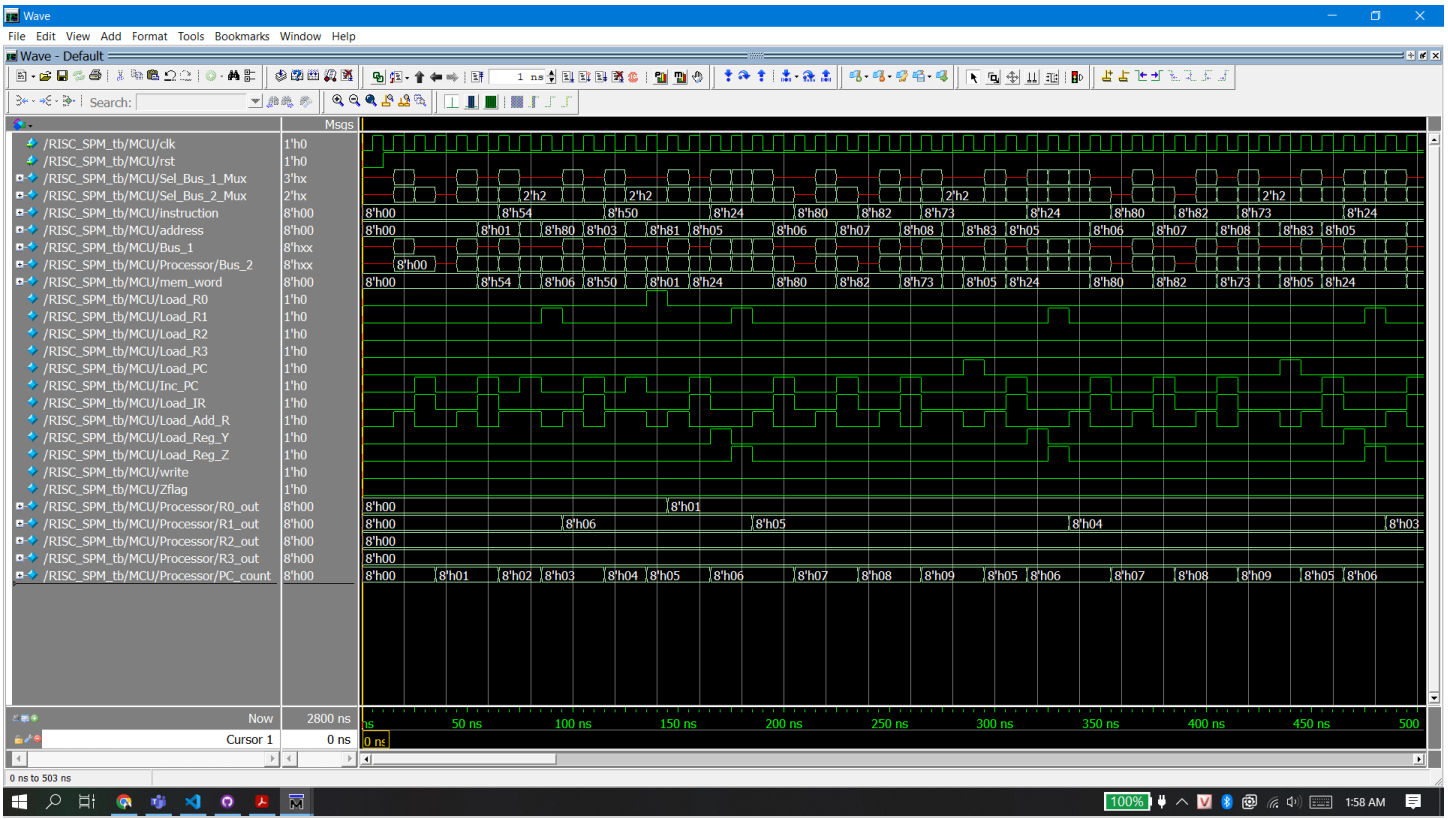

Figure 14 Full Module waveform 1

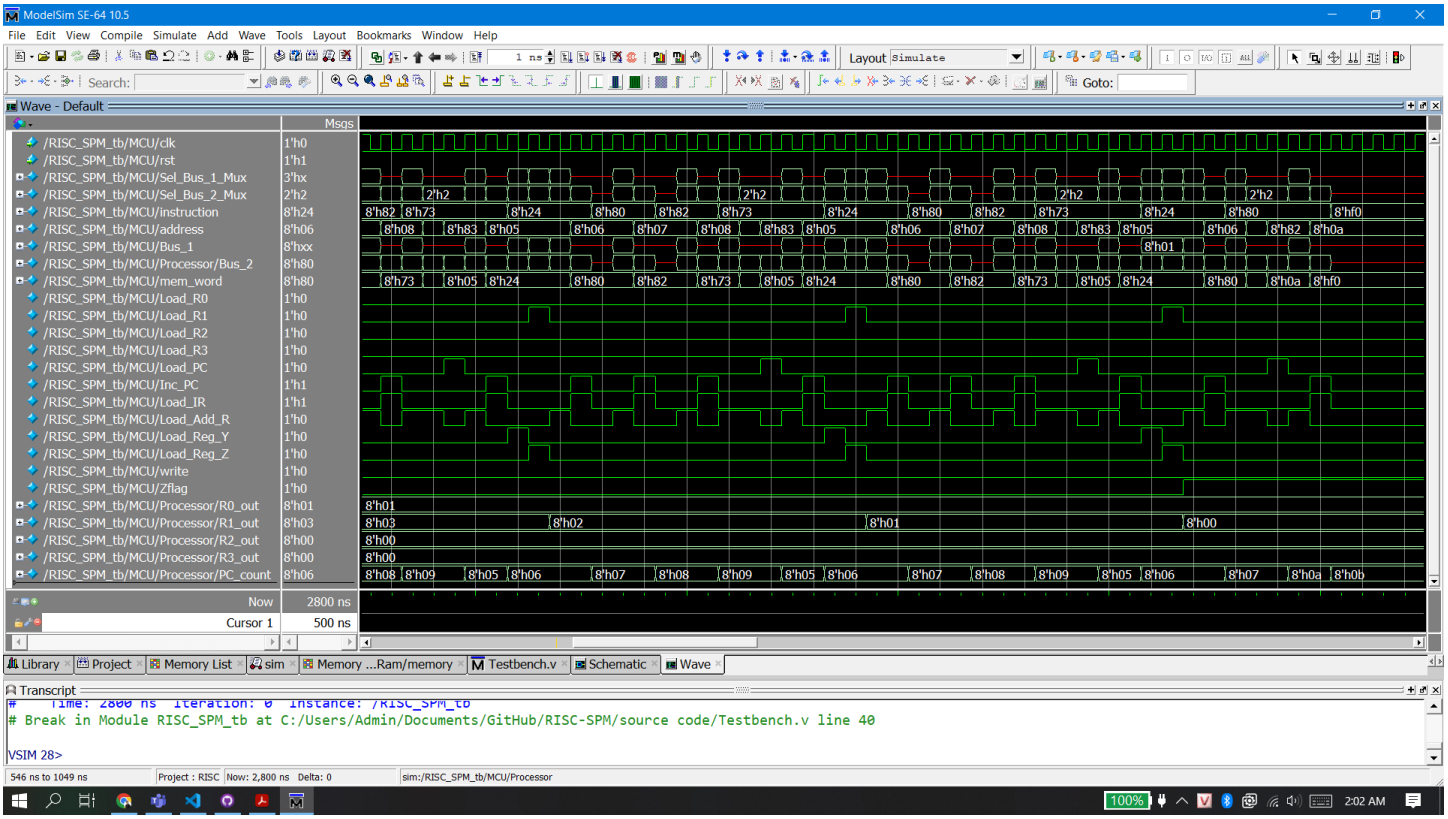

Figure 15 Full Module waveform 2

VII. Conclusion

This article builds an 8-bit RISC CPU, introduces the design process and experimental test in detail, including: hardware composition, instruction set system, etc. The focus is on the design of the controller. Based on the finite state machine, the correspondence and transfer between instructions and states has been realized, and a detailed simulation experiment has been carried out. The results prove that the CPU functions normally and meets expectations.

VIII. Reference

[1] Michael D. Ciletti's Advanced Digital Design with the Verilog HDL, 2005

[2] Samir Palnitkar's Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition, 2003