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

Table of Figure

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 of a CPU

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 my 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 Objective

The goal of this project is to design a RISC-SPM or Reduced Instruction Set Computer Store Program Machine using verilog HDL. As the name already suggest, the final design should be able to :

- Read and write data from the memory
- Performing arithmetic and logic operations base on the instruction.
- Execute the program store inside the memory (A program is just a sequence of instructions)

IV. 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*

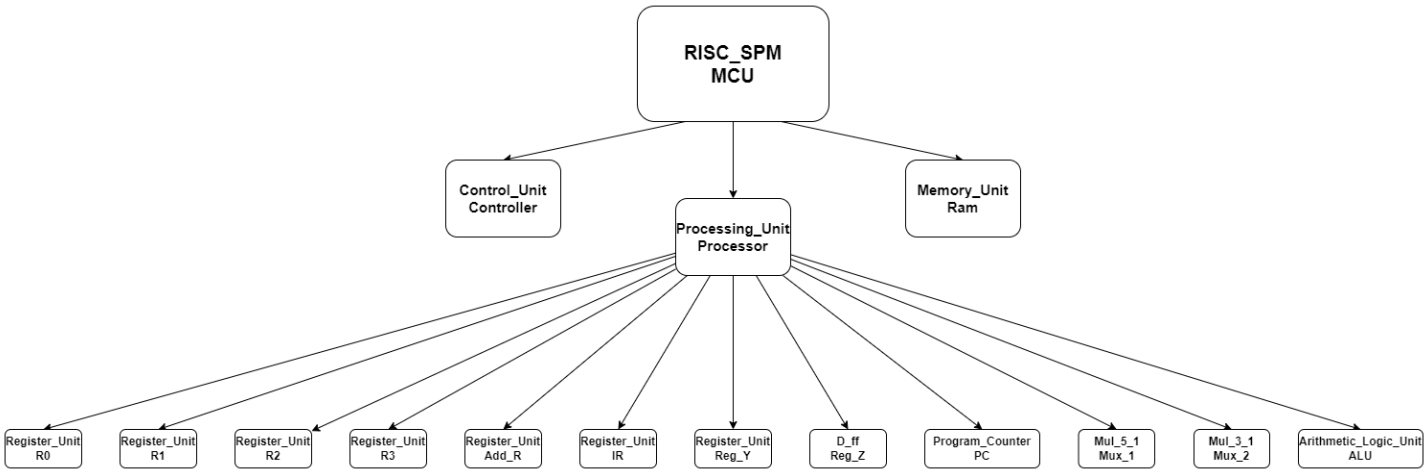


Figure 2 Hierarchy Table

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.

V. Hardware composition

RISC-SPM consists of three functional units :

- Processing Unit (Processor)
- Controll Unit (Controller)
- Memory Unit (RAM)

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

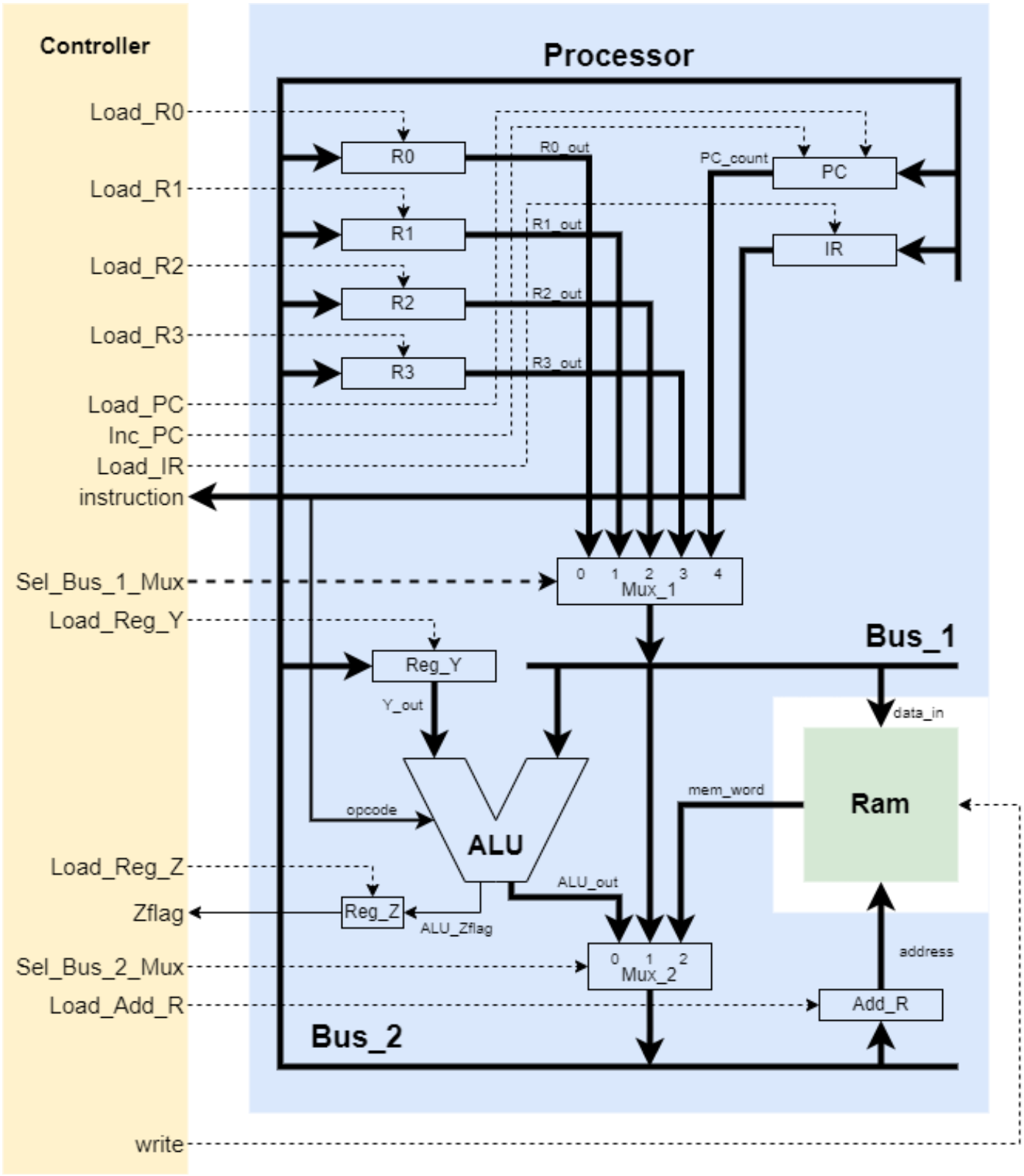


Figure 3 Architecture

Program instructions and data are stored in memory

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

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

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

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) of it depends on both the state and external input (instruction, Zflag)

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 tri-state buffers 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 (opcode) and their actions are listed below.

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)

So overall, each instruction take around **3-5 clock cycles**.

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

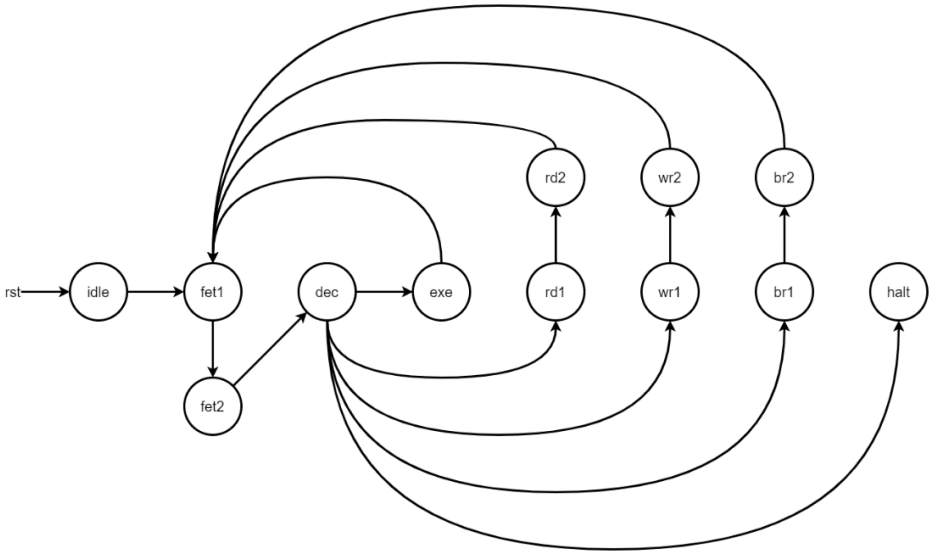


Figure 4 State transition diagram

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*. Some thing to note here, is that because 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).

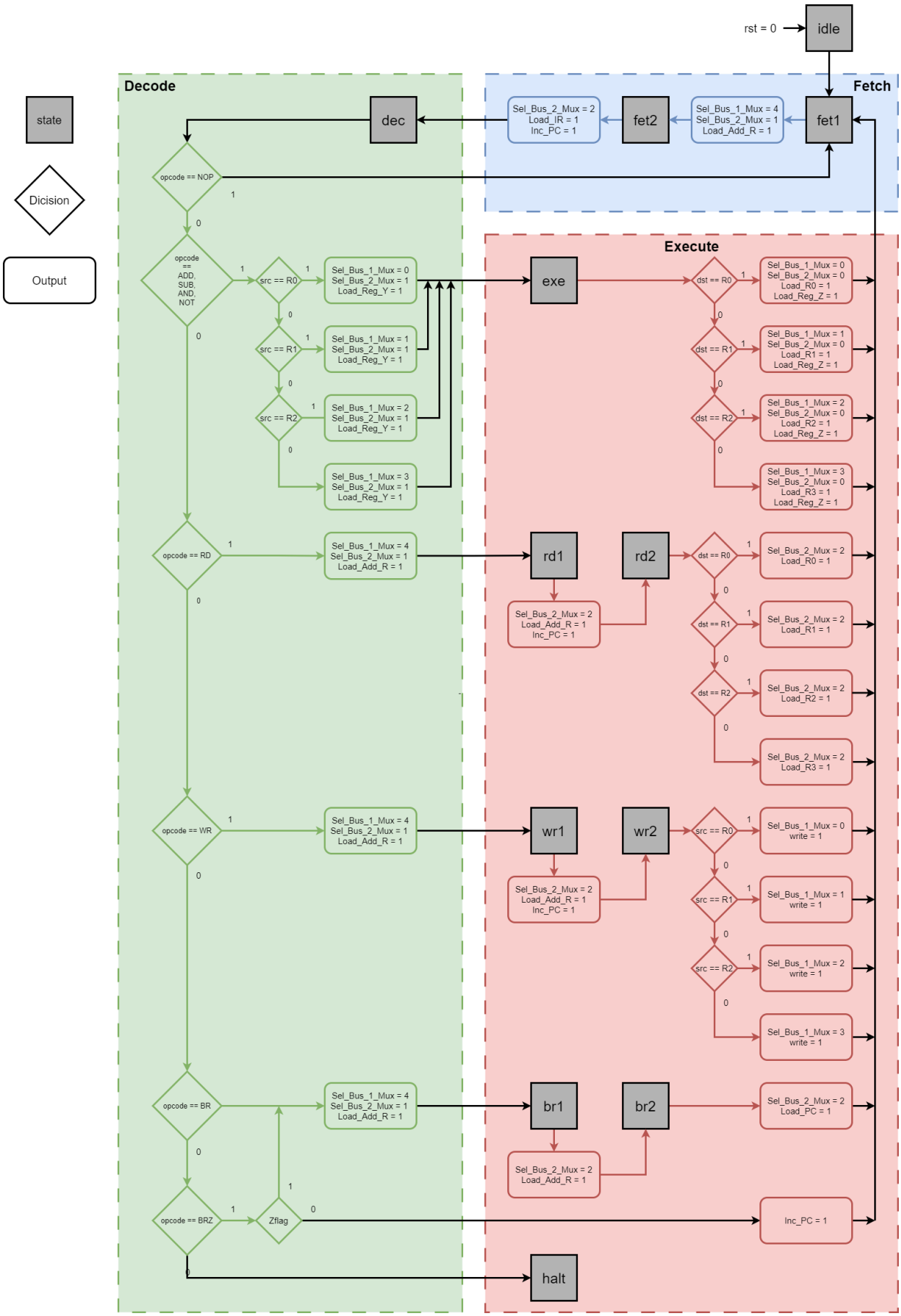


Figure 5 ASM diagram

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.

VI. Verilog implementation

All the verilog source codes is located in my github website or at the end of this report.
https://github.com/canh25xp/RISC-SPM

VII. 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
```

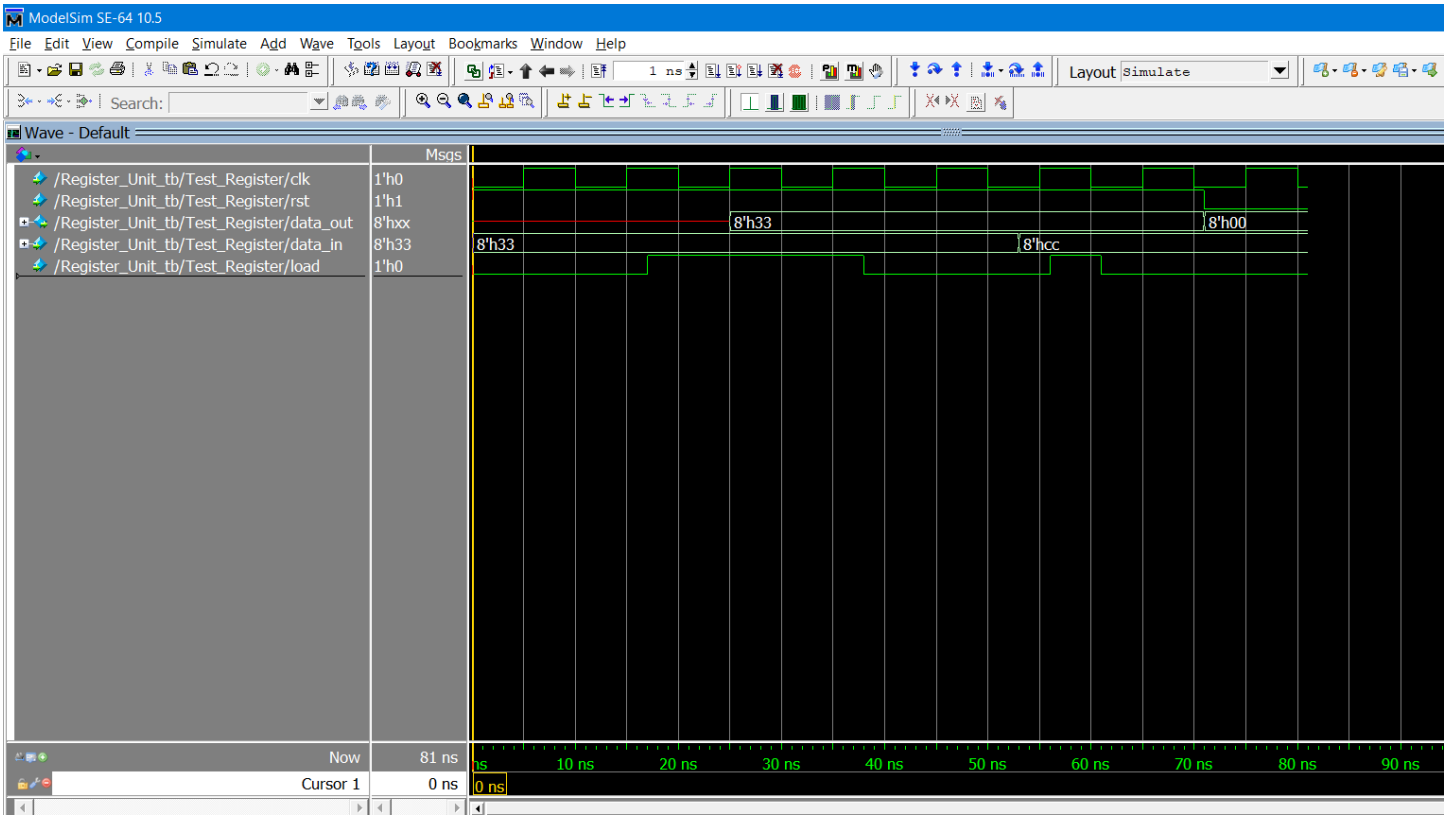


Figure 6 Register testbench waveform

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
```

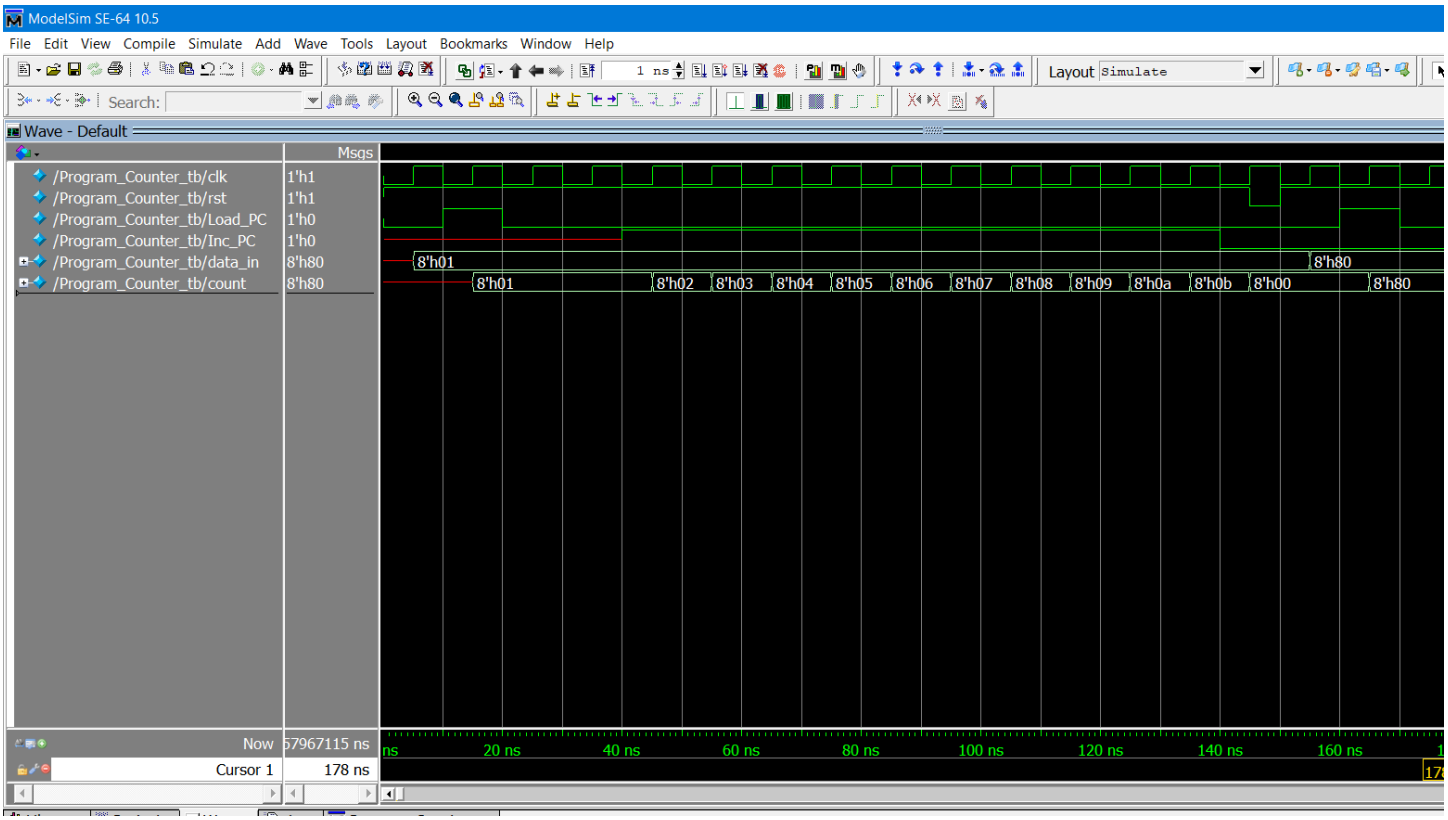


Figure 7 Program Counter testbench waveform

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
```

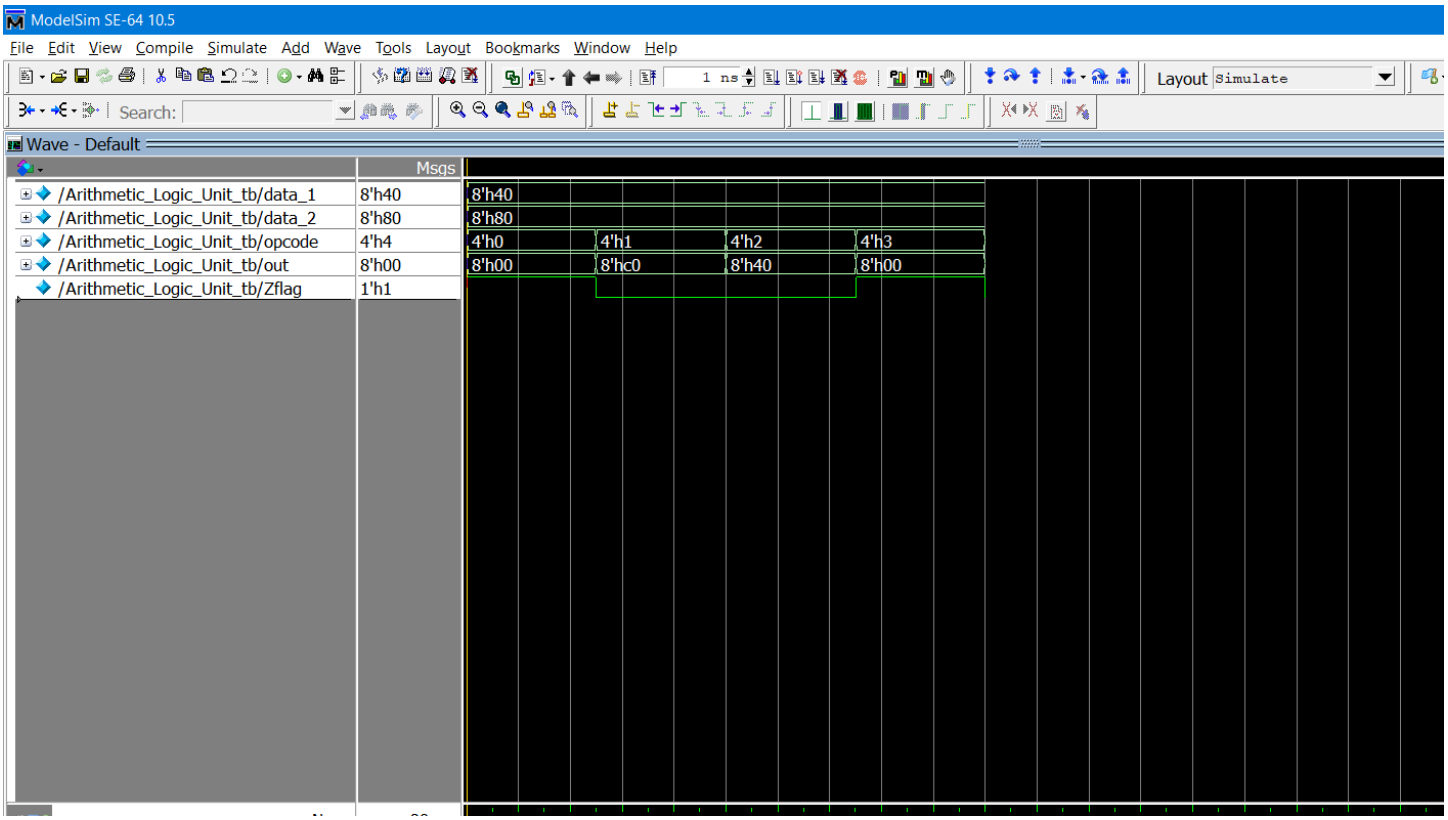


Figure 8 ALU testbench waveform

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
```

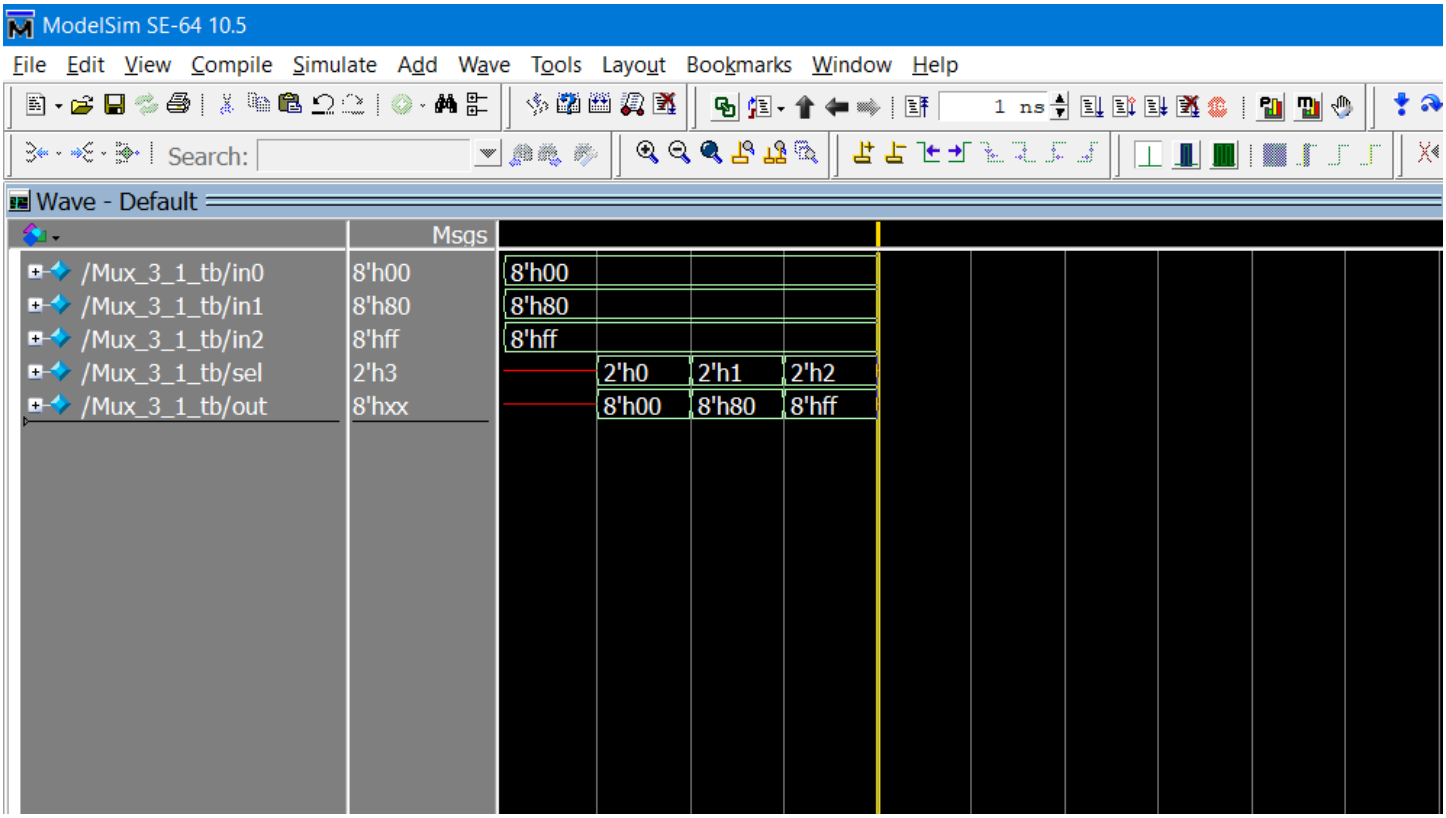


Figure 9 Mux_3_1_tb waveform

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
```

What do we expect the module should behave with this testbench ?

R1 = memory[128] = 6

R0 = memory[129] = 1

R1 = R1 – R0

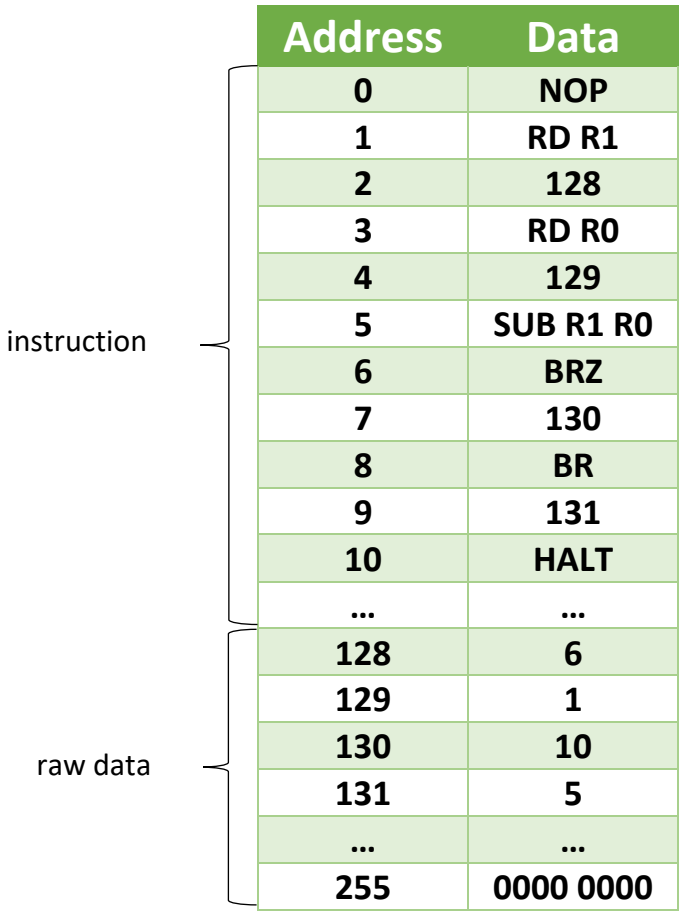
if (Zflag == 1)

PC = memory[130] = 10

HALT

PC = memory[131] = 5

Address	Data
0000 0000	0000 0000
0000 0001	0101 0100
0000 0010	1000 0000
0000 0011	0101 0000
0000 0100	1000 0001
0000 0101	0010 0100
0000 0110	1000 0000
0000 0111	1000 0010
0000 1000	0111 0011
0000 1001	1000 0011
0000 1010	1111 0000
…	…
1000 0000	0000 0110
1000 0001	0000 0001
1000 0010	0000 1010
1000 0011	0000 0101
…	…
1111 1111	0000 0000



Address	Data
0	NOP
1	RD R1
2	128
3	RD R0
4	129
5	SUB R1 R0
6	BRZ
7	130
8	BR
9	131
10	HALT
…	…
128	6
129	1
130	10
131	5
…	…
255	0000 0000

First, it will fill the memory with data like the table above.

Using the instruction set table, we can then decode each data line. We can also convert the numeral system to decimal to make the debug process esier. But how can the machine tell if it is instruction or just raw data ? Well, it cannot, it is the programmer's job to determine and foreseen the bahaviour of the machine. Oneway to do, it's that we can split the memory space into 2 half. Like here, I use the first 128 bytes (0 to 127) to store instructions and the last 128 bytes (128 to 255) to store data, so the program don't misunderstand any data with instruction.

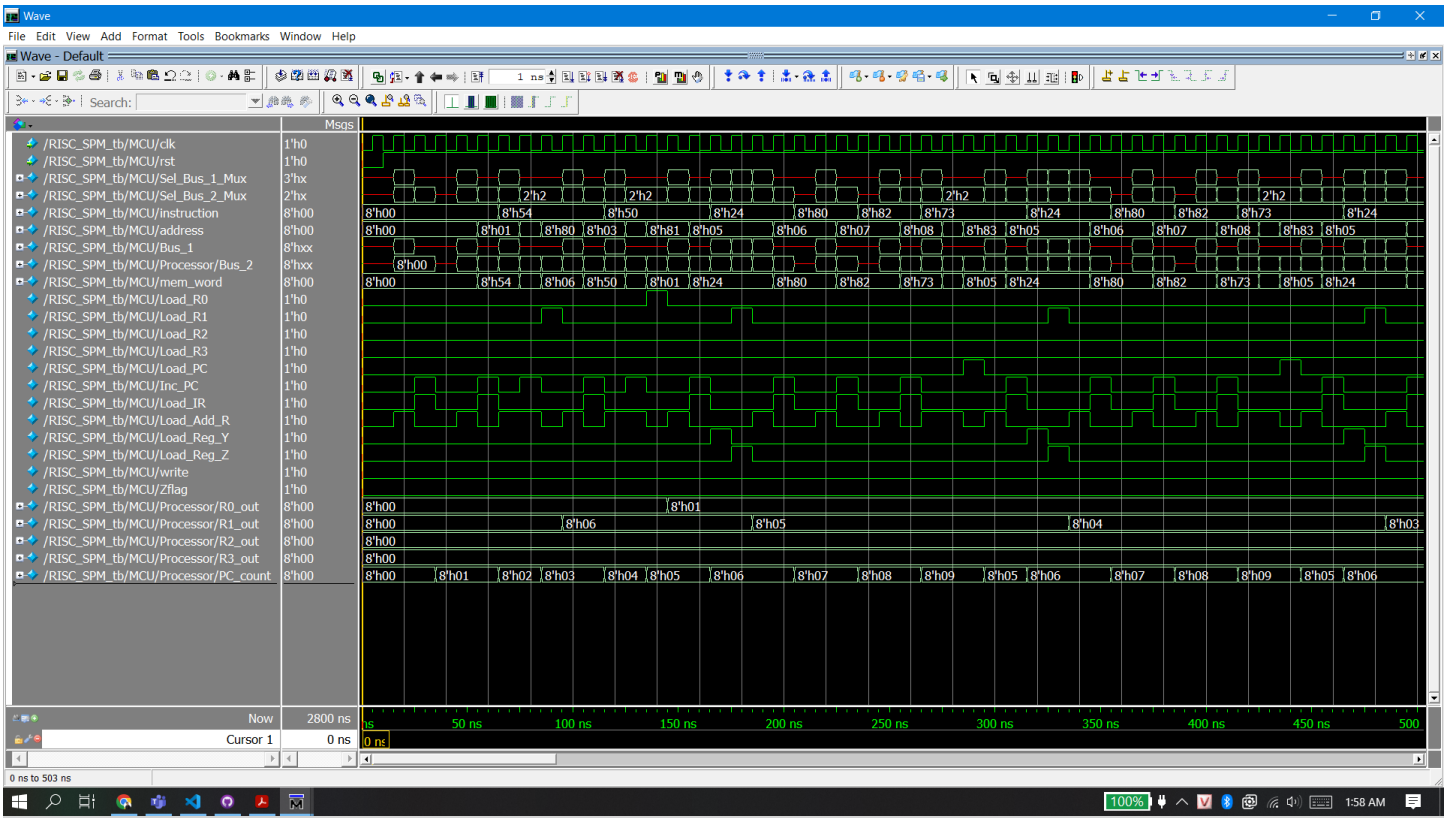


Figure 10 Full Module waveform 1

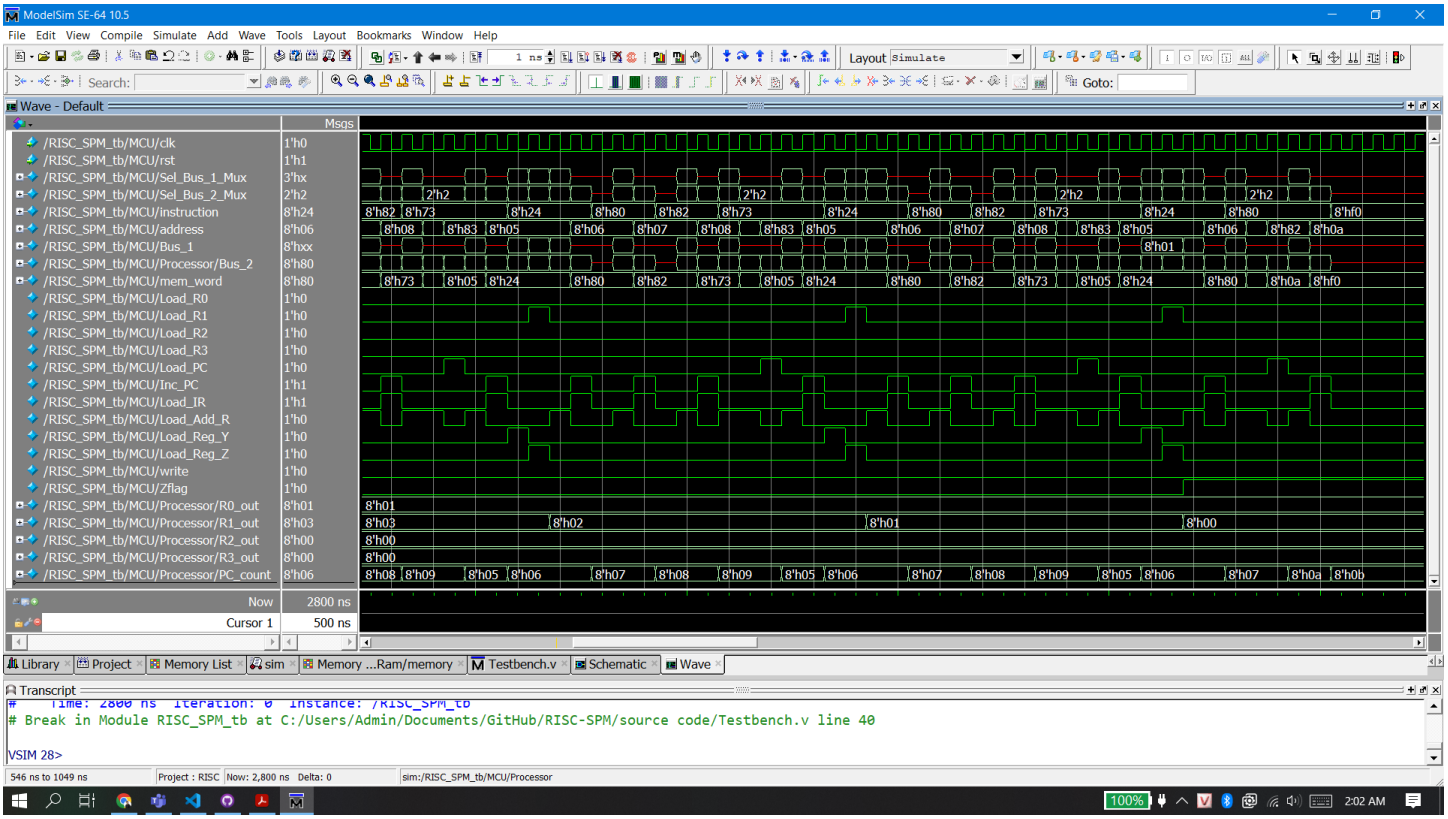


Figure 11 Full Module waveform 2

VIII. 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.

IX. 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

```
`define idle    0
`define fet1    1
`define fet2    2
`define dec     3
`define exe     4
`define rd1     5
`define rd2     6
`define wr1     7
`define wr2     8
`define br1     9
`define br2     10
`define halt    11
`define NOP     4'b0000
`define ADD     4'b0001
`define SUB     4'b0010
`define AND     4'b0011
`define NOT     4'b0100
`define RD      4'b0101
`define WR      4'b0110
`define BR      4'b0111
`define BRZ     4'b1000

module RISC_SPM(input clk,rst);
    wire [2:0] Sel_Bus_1_Mux; //5 to 1 multiplexer
    wire [1:0] Sel_Bus_2_Mux; //3 to 1 multiplexer
    wire [7:0] instruction, address, Bus_1, mem_word;
    wire Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R, Load_Reg_Y,
Load_Reg_Z, write, Zflag;

    Control_Unit Controller(
        Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R, Load_Reg_Y,
Load_Reg_Z, write,
        Sel_Bus_1_Mux,
        Sel_Bus_2_Mux,
        instruction,
        Zflag, clk, rst
    );

    Processing_Unit Processor(
        instruction, address, Bus_1,
        Zflag,
        mem_word,
        Sel_Bus_1_Mux,
        Sel_Bus_2_Mux,
        Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R, Load_Reg_Y,
Load_Reg_Z, clk, rst
    );

    Memory_Unit Ram(
        mem_word,
        Bus_1, address,
        write, clk
    );
endmodule
```

```
module Control_Unit(
    output reg Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R,
Load_Reg_Y, Load_Reg_Z, write,
    output reg [2:0] Sel_Bus_1_Mux,
    output reg [1:0] Sel_Bus_2_Mux,
    input [7:0] instruction,
    input Zflag, clk, rst
);

    reg [3:0] state, next_state;
    reg err_flag; // error flag to debug

    wire [3:0] opcode = instruction[7:4];
    wire [1:0] dst = instruction [3:2]; //destination
    wire [1:0] src = instruction [1:0]; //source

    //state register
    always @ (posedge clk or negedge rst) begin
        if(!rst)
            state <= `idle; // reset active low
        else
            state <= next_state;
    end

    //next state logic
    always @ (state or opcode or src or dst or Zflag) begin
        Sel_Bus_1_Mux = 3'bx ; Sel_Bus_2_Mux = 2'bx;
        Load_R0 = 0; Load_R1 = 0; Load_R2 = 0; Load_R3 = 0; Load_Reg_Y = 0; Load_Reg_Z = 0;
        Load_PC = 0; Inc_PC = 0; Load_IR = 0; Load_Add_R = 0;
        write = 0;
        err_flag = 0;
        next_state = state;
        case (state)
            `idle : next_state = `fet1;
            `fet1 :begin
                next_state = `fet2;
                Sel_Bus_1_Mux = 4; // Bus_1 = PC
                Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                Load_Add_R = 1;
            end
            `fet2 :begin
                next_state = `dec;
                Sel_Bus_2_Mux = 2; // Bus_2 = memory
                Load_IR = 1;
                Inc_PC = 1;
            end
            `dec :begin
                case(opcode)
                    `NOP : next_state = `fet1;
                    `ADD, `SUB, `AND, `NOT : begin
                        next_state = `exe;
                        case (src)
                            0 : Sel_Bus_1_Mux = 0; //Bus 1 = R0
                            1 : Sel_Bus_1_Mux = 1; //Bus 1 = R1
                            2 : Sel_Bus_1_Mux = 2; //Bus 1 = R2
                            3 : Sel_Bus_1_Mux = 3; //Bus 1 = R3
                            default : err_flag = 1;
                        endcase
                        Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                        Load_Reg_Y = 1; // Reg_Y = Bus_2 = Bus_1 = src
                    end
```

```
            `RD : begin
                next_state = `rd1;
                Sel_Bus_1_Mux = 4; // Bus_1 = PC
                Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                Load_Add_R = 1;
            end
            `WR : begin
                next_state = `wr1;
                Sel_Bus_1_Mux = 4; // Bus_1 = PC
                Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                Load_Add_R = 1;
            end
            `BR : begin
                next_state = `br1;
                Sel_Bus_1_Mux = 4; // Bus_1 = PC
                Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                Load_Add_R = 1;
            end
            `BRZ : begin
                next_state = `br1;
                if (Zflag == 1) begin
                    Sel_Bus_1_Mux = 4; // Bus_1 = PC
                    Sel_Bus_2_Mux = 1; // Bus_2 = Bus_1
                    Load_Add_R = 1;
                end
                else begin
                    next_state = `fet1;
                end
            end
            default : next_state = `halt;
        endcase
    end
    `exe :begin
        next_state = `fet1;
        case (dst)
            0: begin Sel_Bus_1_Mux = 0; Load_R0 = 1; end //R0 = ALU_out
            1: begin Sel_Bus_1_Mux = 1; Load_R1 = 1; end //R1 = ALU_out
            2: begin Sel_Bus_1_Mux = 2; Load_R2 = 1; end //R2 = ALU_out
            3: begin Sel_Bus_1_Mux = 3; Load_R3 = 1; end //R3 = ALU_out
            default : err_flag = 1;
        endcase
        Sel_Bus_2_Mux = 0; //Bus_2 = ALU_out
        Load_Reg_Z = 1;
    end
    `rd1 :begin
        next_state = `rd2;
        Sel_Bus_2_Mux = 2; // Bus_2 = mem_word
        Load_Add_R = 1;
        Inc_PC = 1;
    end
    `rd2 :begin
        next_state = `fet1;
        Sel_Bus_2_Mux = 2;
        case (dst)
            0 : Load_R0 = 1;
            1 : Load_R1 = 1;
            2 : Load_R2 = 1;
            3 : Load_R3 = 1;
            default : err_flag = 1;
        endcase
    end
```

```
            `wr1 :begin
                next_state = `wr2;
                Sel_Bus_2_Mux = 2; // Bus_2 = mem_word
                Load_Add_R = 1;
                Inc_PC = 1;
            end
            `wr2 :begin
                next_state = `fet1;
                case (src)
                    0 : Sel_Bus_1_Mux = 0; // Bus_1 = R0
                    1 : Sel_Bus_1_Mux = 1; // Bus_1 = R1
                    2 : Sel_Bus_1_Mux = 2; // Bus_1 = R2
                    3 : Sel_Bus_1_Mux = 3; // Bus_1 = R3
                    default : err_flag = 1;
                endcase
                write = 1;
            end
            `br1 :begin
                next_state = `br2;
                Sel_Bus_2_Mux = 2; // Bus_2 = mem_word
                Load_Add_R = 1;
            end
            `br2 :begin
                next_state = `fet1;
                Sel_Bus_2_Mux = 2; // Bus_2 = mem_word
                Load_PC = 1;
            end
            `halt :begin
                next_state = `halt;
            end
            default : next_state = `idle;
        endcase
    end

endmodule

module Processing_Unit(
    output [7:0] instruction, address, Bus_1,
    output Zflag,
    input [7:0] mem_word,
    input [2:0] Sel_Bus_1_Mux,
    input [1:0] Sel_Bus_2_Mux,
    input Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R, Load_Reg_Y,
Load_Reg_Z, clk, rst
);
    wire [7:0] Bus_2, R0_out, R1_out, R2_out, R3_out, PC_count, Y_out, ALU_out;
    wire ALU_Zflag;
    wire [3:0] opcode = instruction [7:4];

    Register_Unit R0(R0_out, Bus_2,Load_R0, clk,rst);
    Register_Unit R1(R1_out, Bus_2,Load_R1, clk,rst);
    Register_Unit R2(R2_out, Bus_2,Load_R2, clk,rst);
    Register_Unit R3(R3_out, Bus_2,Load_R3, clk,rst);
    Register_Unit Reg_Y(Y_out, Bus_2, Load_Reg_Y, clk, rst);
    Register_Unit Add_R(address, Bus_2, Load_Add_R, clk, rst);
    Register_Unit IR(instruction, Bus_2, Load_IR, clk, rst);

    D_ff Reg_Z(Zflag, ALU_Zflag, Load_Reg_Z, clk,rst);

    Program_Counter PC(PC_count, Bus_2, Load_PC, Inc_PC, clk, rst);
```

```
    Mux_5_1 Mux_1(Bus_1, R0_out, R1_out, R2_out,  R3_out, PC_count, Sel_Bus_1_Mux);
    Mux_3_1 Mux_2(Bus_2, ALU_out, Bus_1, mem_word, Sel_Bus_2_Mux);

    Arithmetic_Logic_Unit ALU(ALU_out, ALU_Zflag, Y_out, Bus_1, opcode);
endmodule

module Memory_Unit(
    output [7:0] data_out,
    input [7:0] data_in, address,
    input write, clk
);
    reg [7:0] memory [255:0]; //256 bytes ram

    always @(posedge clk) begin
        if(write)
            memory[address] <= data_in;
    end
    assign data_out = memory[address];
endmodule

module Arithmetic_Logic_Unit(
    output reg [7:0] ALU_out,
    output ALU_Zflag,
    input [7:0] data_1, data_2,
    input [3:0] opcode
);

    assign ALU_Zflag = ~|ALU_out; //reduction nor
    always @(opcode or data_1 or data_2) begin
        case(opcode)
            `ADD :      ALU_out = data_2 + data_1;
            `SUB :      ALU_out = data_2 - data_1;
            `AND :      ALU_out = data_1 & data_2;
            `NOT :      ALU_out = ~ data_1;
            default :   ALU_out = 0;
        endcase
    end
endmodule

module Register_Unit(output reg [7:0] data_out, input [7:0] data_in, input load, clk, rst);
    always @(posedge clk or negedge rst)
    if (!rst)
        data_out <= 0;
    else if (load)
        data_out <= data_in;
endmodule

module Program_Counter(output reg [7:0] count, input [7:0] data_in, input Load_PC, Inc_PC, clk,
rst);
    always @(posedge clk or negedge rst)
    if (!rst)
        count <= 0;
    else if (Load_PC)
        count <= data_in;
    else if (Inc_PC)
        count <= count+1;
endmodule
```

```
module D_ff(output reg data_out, input data_in, load, clk, rst);
    always @(posedge clk or negedge rst)
    if (!rst)
        data_out <= 0;
    else if (load)
        data_out <= data_in;
endmodule

module Mux_3_1 (output reg [7:0] out, input [7:0] in0, in1, in2, input [1:0] sel);
    always @(in0 or in1 or in2 or sel)
    case(sel)
        2'b00 : out = in0;
        2'b01 : out = in1;
        2'b10 : out = in2;
        default : out = 'bx;
    endcase
endmodule

module Mux_5_1 (output reg [7:0] out, input [7:0] in0, in1, in2, in3, in4, input [2:0] sel);
    always @(in0 or in1 or in2 or in3 or in4 or sel)
    case(sel)
        3'b000 : out = in0;
        3'b001 : out = in1;
        3'b010 : out = in2;
        3'b011 : out = in3;
        3'b100 : out = in4;
        default : out = 'bx;
    endcase
endmodule
```