Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Architecture

Project 2:

SINGLE CYCLE RISC PROCESSOR IMPLEMENTATION

Prepared by:

Student Name: Abdullah Sami Naser **Student ID:** 1201952

Student Name: Majd Rayad Abdeddein **Student ID:** 1202923

Instructors: Dr. Aziz Qaroush & Dr. Ayman Hroub

Sections: 2 & 3

Date: 30/6/2023

1.Abstract

The project aims to design, implement, and verify a simple RISC processor in Verilog. The processor will implement a subset of a given ISA instructions. And then the processor functionality will be verified using Verilog Simulations and Testbenches.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

2. Theory

2.1: Specifications

2.1.1: Processor Specifications

The RISC processor has the following specifications:

1. The instruction size is 32 bits
2. 32 32-bit general-purpose registers: from R0 to R31.
3. A special purpose register for the program counter (PC)
4. It has a stack called control stack which saves the return addresses
5. Stack pointer (SP), another special purpose register to point to the top of the control stack. SP holds the address of the empty element on the top of the stack. For simplicity, you can assume a separate on-chip memory for the stack, and the initial value of SP is zero.
6. Four instruction types (R-type, I-type, J-type, and S-type).
7. The processor's ALU has an output signal called "zero" signal, which is asserted when the result of the last ALU operation is zero.
8. Separate data and instructions memories

2.1.2: ISA Specifications

A. Instruction Formats:

There will be 4 different formats of instructions:

1. **R-Type (00) Format:**

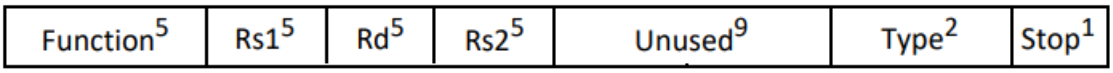


Figure 1: R-Type Format

Where the fields are:

→ 5-bit Function: define the specific operation of the instruction

→ 2-bit type: define the type of this instruction (R-Type, I-Type…. etc.)

→ 5bits Rs1, Rd, Rs2: Rs1 is the first source register, Rs2 is the second source register and Rd is the destination register.

→ 9bit unused: unused bits

→ 1-bit stop: specify if this instruction is the last instruction in the function block of code.

2. **I-Type (01) Format:**

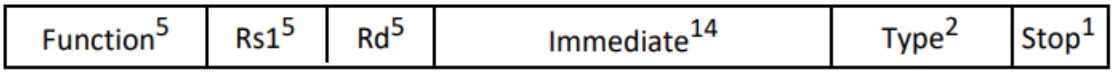


Figure 2: I-Type Format

Where the fields are:

→ 14-bit Immediate: signed/unsigned immediate according logic or other instructions

→ all other fields' descriptions are the same as exists in R-type.

3. **J-Type (10) Format:**

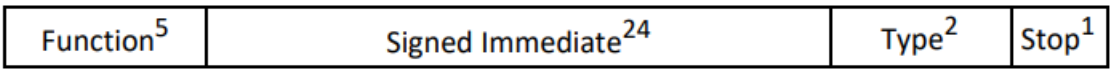


Figure 3: J-Type Format

Where the fields are:

→ 24-bit Immediate: the jump offset

→ all other fields' descriptions are the same as exists in other types.

4. **S-Type (11) Format:**

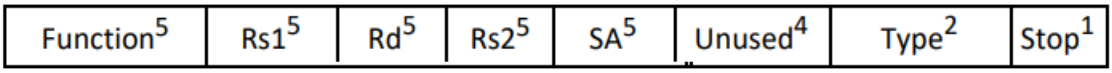


Figure 4: S-Type Format

Where the fields are:

→ 5-bit SA: the shift amount if it is constant

→ all other fields' descriptions are the same as exists in other types.

B. Instructions Encoding:

In this project, only subset of instructions will be implemented. These instructions have the following encoding as shown in table 1:

Table 1: Instructions Encoding

No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & $Immediate^{14}$	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + $Immediate^{14}$	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
J-Type Instructions			
10	J	PC = PC + $Immediate^{24}$	00000
11	JAL	PC = PC + $Immediate^{24}$ Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	Reg(Rd) = Reg(Rs1) << SA^5	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^5	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

2.2: Design Approach

2.2.1: Single Cycle CPU

The selected approach for implementing this processor is the **Single-Cycle** approach. A single-cycle RISC (Reduced Instruction Set Computing) processor design approach follows a simplistic architecture where each instruction is executed within a single clock cycle. The processor design typically consists of a control unit that coordinates the flow of instructions and data, an arithmetic logic unit (ALU) that performs mathematical and logical operations, and a set of registers to store data and instructions.

In this approach, the control unit decodes the instruction fetched from memory and generates control signals that activate the necessary components for executing that specific instruction. All operations required by the instruction, such as fetching operands from registers, performing calculations in the ALU, and storing results back to registers or memory, occur within a single clock cycle. This design approach simplifies the control logic and ensures a uniform execution time for each instruction, allowing for straightforward pipelining and instruction sequencing.

However, the single-cycle RISC processor design approach has limitations. Since each instruction takes exactly one clock cycle, it can result in long clock cycles for complex instructions, leading to decreased processor efficiency. Additionally, resources such as the ALU and registers may be underutilized during certain instructions, further impacting performance. Despite these limitations, the single-cycle RISC processor design approach serves as an introductory and educational model for understanding the basics of processor design and instruction execution in a straightforward and comprehensible manner.[1]

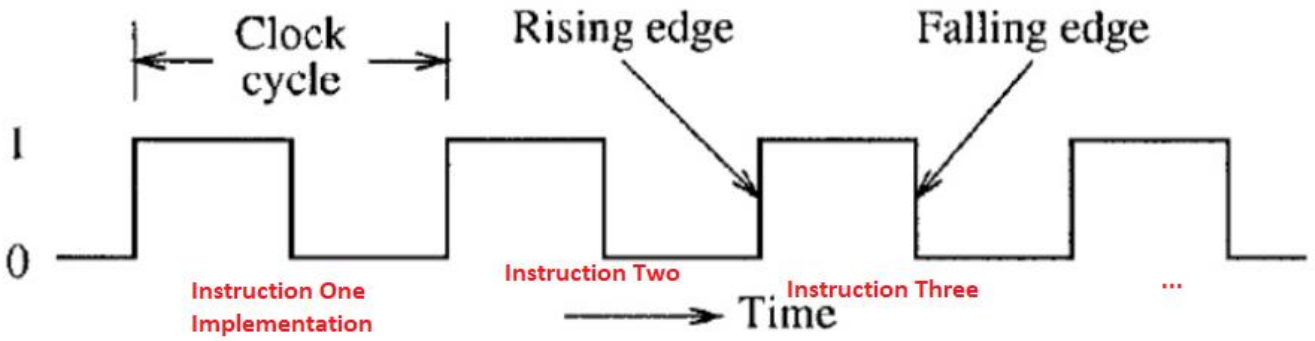


Figure 5: Single Cycle Implementation

2.2.2: Datapath and Control Unit design

Our design will be divided into two parts, the **Datapath design** and the **Control Unit** design. In the Datapath design, all instructions that share the same Datapath will be grouped together. RTL for each group of instructions will be written. Then, a basic group (in our case it is R-type) will be selected and then the Datapath components and connections will be implemented according to the RTL of that group. Other groups will be supported in the Datapath in **Incremental Approach.**

2.3: Datapath Components

2.3.1: PC Register

This is the Program Counter (PC) register which contains the address of the next instruction.

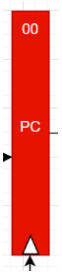


Figure 6: PC Register

2.3.2: Memory Elements

The first memory will be the **Instruction Memory**. This is Read Only Memory **(ROM)** that will carry all instruction of our program. This memory has an input address and 32-bit instruction as output.

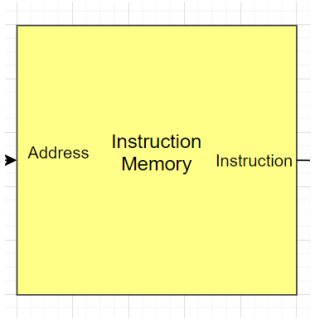


Figure 7: Instruction Memory

The second memory will be **Data Memory**. This memory will store the operands and any other data. It has two input signals **Write** and **Read** to specify the operation. If Read=1 and Write=0 then it will be read operation. If Read=0 and Write=1 then it will be write operation. Anything else will be considered as NOP. It has input **32-bit Data in Bus** and **32-bit Data out Bus**. And it also has **32-bit Address Bus** to specify the location of memory cell. In addition to the clock CLK.

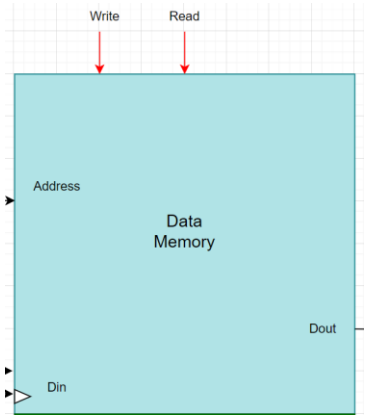


Figure 8: Data Memory

The third memory will be the **Stack Memory**. This memory will be used only for storing the return address. It has a pointer called **Stack Pointer (SP)** refers to the empty location after the top of the stack. It has **an input return address** as 32-bit and **output return address**. In addition to the clock CLK, **push and pop** signals.

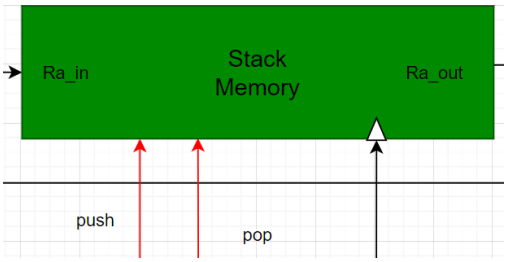


Figure 9: Stack Memory

2.3.4: Register File

The register file is a set of 32-32bit registers from R0 to R31. With two Read buses **BusA** and **BusB.** And one write bus **BusW**. It has also a write signal called **RegWrite** and 3 addresses for two source registers **RA** and **RB** in addition to the destination register **RW**.

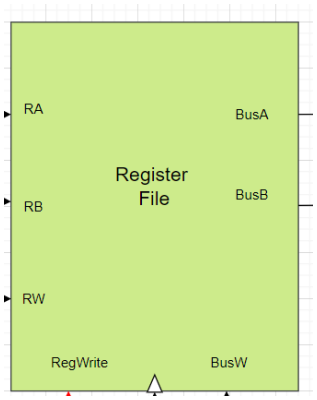


Figure 10: Register File

2.3.5: Extender and Mux's

The extender is a unit that extend a given input into 32-bits by **ANDING** the **MSB** of the input with the signOP (0→zero extension,1→sign extension). It has two inputs **immediate 14** bit and **SA 5** bits. And two signals **signOp** to determine the extension operation and **Imm_sel** signal to select the immediate in which we want to extend.

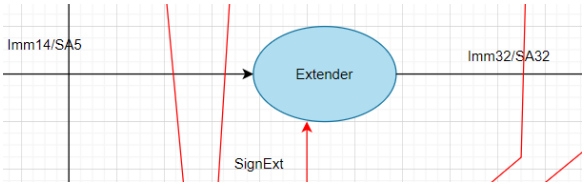


Figure 11: Extender

There will be different Muxes such as 2-to-1 and 4-to-1 Muxes.

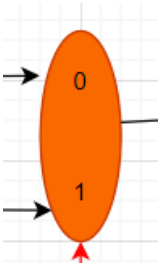


Figure 12: 2-to-1Mux

2.3.6: ALU

The ALU will perform all arithmetic and logical operation. It is 32-capability ALU which **has 2 32-bit inputs** and one **32-bit ALU_Out**. It has a signal **ALU_OP** to determine the specific ALU Operation. In addition to the ALU result there will be signals asserted such as **Zero, Negative and Carry signal**.

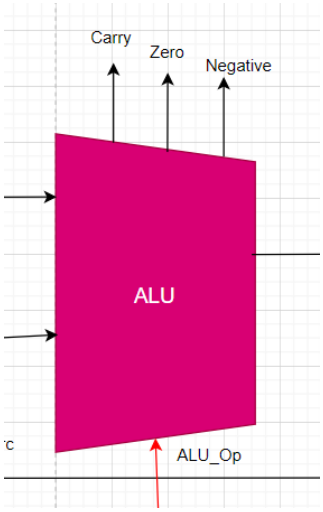


Figure 13: ALU

2.3.7: IR (Instruction Decode Unit)

This unit has an instruction as input. It performs the decode operation. This means that this unit gives as output all needed information from the instruction such as **rs1,rd,function,type,stop bit** and so on.

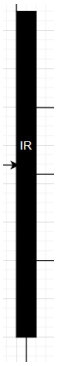


Figure 14: IR

3. Design Procedure

3.1: Datapath Design

According to the components and design methodology described above in the theory part. The final Datapath of our processor was as follows.

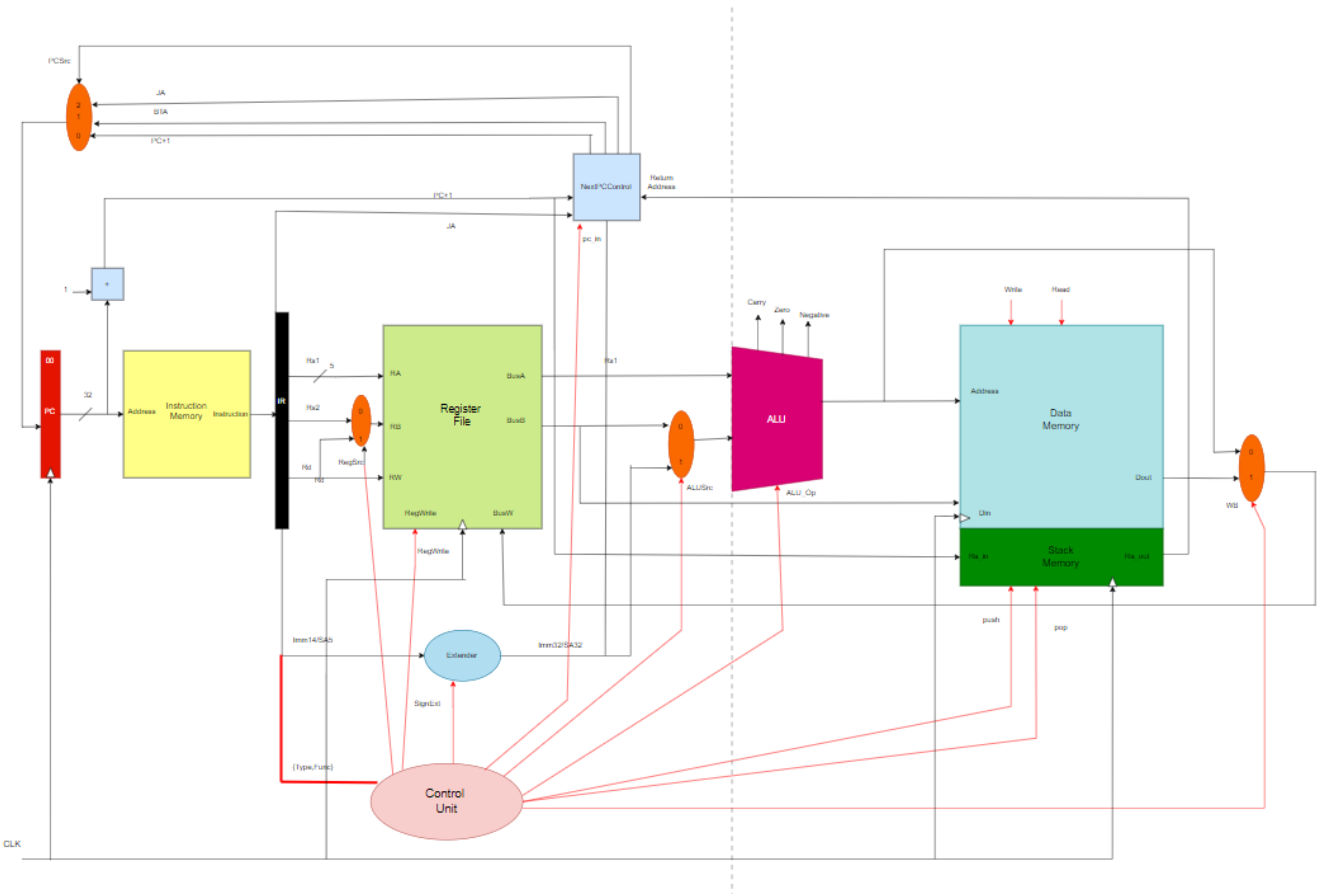


Figure 15: Full Datapath

Each component and connection in the Datapath will be written in the Verilog HDL in order to test the functionality of our processor.

These connections and components support all instructions that we want to implement. The design started by grouping instructions into the same Datapath groups. And then the RTL of each group was written. Moreover, the functional units and Datapath components with transfer between them was extracted from RTL and then the Datapath was built.

3.2: Control Unit Design

The control unit was divided into two parts. The **Main Control Unit** which asserted all needed signals for the proper flow of instructions in the path. And the second one was the **Next PC Control Unit** which determined the next PC input (next instructions). At the first all instructions are listed, and all signals are provided for all instructions as the following table:

Table 2: CU signals

Sig. / Ins.	**Pc_in**	**Reg_src**	**Reg_write**	**Sign_ext**	**Imm_sel**	**ALU_src**	**Mem_rd**	**Mem_wr**	**WB_data**	**ALU_op**	**push**	**pop**
R-Type	0 if stop bit = 0, 3 if stop bit = 1	0	1	x	x	0	0	0	0	According function field	0/1 if stop bit = 1	0
ANDI	0 if stop bit = 0, 3 if stop bit = 1	x	1	0	0	1	0	0	0	000	0/1 if stop bit = 1	0
ADDI	0 if stop bit = 0, 3 if stop bit = 1	x	1	1	0	1	0	0	0	0001	0/1 if stop bit = 1	0
LW	0 if stop bit = 0, 3 if stop bit = 1	x	1	1	0	1	1	0	1	001	0/1 if stop bit = 1	0

SW	0 if stop bit = 0, 3 if stop bit = 1	1	0	1	0	1	0	1	x	001	0/1 if stop bit = 1	0
BEQ	1	1	0	1	0	0	0	0	x	100	0	0
J-Type	2	x	x	x	x	x	x	x	x	x	0 if J, 1 if JAL	0
S-Type with SA	0 if stop bit = 0, 3 if stop bit = 1	x	1	0	1	1	0	0	0	101 SLL, 110 SLR	0/1 if stop bit = 1	0
S-Type with V	0 if stop bit = 0, 3 if stop bit = 1	0	1	x	x	0	0	0	0	101 SLL, 110 SLR	0/1 if stop bit = 1	0

And the **ALU_op** was determined as follows, each distinct operation in ALU was given a unique code:

Table 3: ALU Signals

Operation	Binary Code
AND	000
ADD	001
SUB	010
CMPLS	011
EQ	100
SLL	101
SLR	110

The **NextPCControlUnit** was determined as the following suedocode:

```
If pc_in == 2 then
            If (zero == 1) then
                  PCSrc = 2
            Else
                  PCSrc = 0
      Else
            PCSrc = pc_in
PCin0 = PC+1
PCin1 = BTA
PCin2 = JA
PCin3 = Stack.pop
```

3.3: Programming

All units described in the theory and design parts were implemented in Verilog and tested using Testbenches. Here are the codes of these parts:

3.3.1: PC

```
module PCReg (
    input init,
    input clk,
    input reset,
    input [31:0] input_address,
    output reg [31:0] output_address
);

    always @(posedge clk or negedge reset) begin
        if (!reset)
            output_address <= 32'h0000_0000;
        else begin
            if (!init)
            output_address <= input_address;
            else
                output_address   =0;
        end
    end

endmodule
```

Figure 16: PC Code

This code works exactly as the theory part about PC. Note that the PC works at the positive edge of the clock.

3.3.2: PC Mux

```
module PCMux(in0,in1,in2,in3,PCSrc,address);
     input [31:0]in0,in1,in2,in3;
     input [1:0]PCSrc;
     output reg [31:0] address;
     always @(in0 or in1 or in2 or PCSrc) begin
         if(PCSrc == 0)
             address = in0;
         else if (PCSrc == 1)
             address = in1;
         else if (PCSrc == 2)
             address = in2;
         else
             address = in3;
    end
    endmodule
```

Figure 17: PCMux code

This code represents the mux before the PC register. Which chooses the correct input address to the PC.

3.3.3: Instruction Memory

```
//****************Instruction Memory****************
 module InstructionMemory (
  input [31:0] address,output reg [31:0] instruction);

  // Define the instruction memory
  reg [31:0] memory [255:0];

  // Fetch the instruction at the specified address
  always @(address) begin
    instruction = memory[address];
  end

  // Initialize the instruction memory
  initial begin
    // Load instructions into memory
    memory[0] = 32'h00000002;
    memory[1] = 32'h00420002;
    memory[2] = 32'h00840002;
    memory[3] = 32'h00C60002;
    memory[4] = 32'h01080002;
    memory[5] = 32'h014A0002;
    memory[6] = 32'h018C0002;
    memory[7] = 32'h08000012;
    memory[8] = 32'h08420012;
    memory[9] = 32'h0884000A;

    memory[10] = 32'h2002000A;
    memory[11] = 32'h094A000A;
    memory[12] = 32'h18C00002;
    memory[13] = 32'h0800000A;
    memory[14] = 32'h18800002;
    memory[15] = 32'h08000324;
    memory[16] = 32'h110C2000;

    memory[100] = 32'h10880003;

  end


endmodule
```

Figure 18: Instruction Memory

Note that this instruction memory is not synchronized with the clock in the code due to the single cycle correct implementation. It starts its operation when the input address changes. And the input address changes only at the +ve edge of the clk at PC. So, it seems that this instruction memory synchronized with the clock.

Note that the memory is **word addressable** so the next instruction will be after 1 location from the current and this gives simple implementation for other components such as **PCControl**.

3.3.4: Instruction Decode Unit

```
module InstructionDecodeUnit(instruction,typ,stop,func,rs1,rs2,rd,imm14,sa5,imm24);
    input [31:0] instruction;
    output reg stop;
    output reg [1:0]typ;
    output reg [4:0] func,rs1,rs2,rd,sa5;
    output reg [13:0]imm14;
    output reg [23:0]imm24;

    always @(*) begin
        case (instruction[2:1]) //according to type bits
            2'b00: begin // R-type
                func = instruction[31:27];
                rs1 = instruction[26:22];
                rd = instruction[21:17];
                rs2 = instruction[16:12];
                typ = instruction[2:1];
                stop= instruction[0];
            end
            2'b01: begin // I-type
                func = instruction[31:27];
                rs1 = instruction[26:22];
                rd = instruction[21:17];
                imm14 = instruction[16:3];
                typ = instruction[2:1];
                stop = instruction[0];
            end
            2'b10: begin // J-type
                func = instruction[31:27];
                imm24 = instruction[26:3];
                typ = instruction[2:1];
                stop = instruction[0];
            end
            2'b11: begin // S-type
                func = instruction[31:27];
                rs1 = instruction[26:22];
                rd = instruction[21:17];
                rs2 = instruction[16:12];
                sa5 = instruction[11:7];
                typ = instruction[2:1];
                stop= instruction[0];
            end
            default : begin
                typ = 2'bxx;
                end

        endcase
    end
endmodule
```

Figure 19: ID unit

This unit accepts the instruction as input and decodes it. Exactly as described in the theory part. All needed information from instruction extracted here and can be used later such as function field,Rs,immediate,stop bit and so on.

3.3.5: Control Unit

```
module ControlUnit(typ,func,pc_in,reg_src,reg_write,sign_ext,Imm_sel, ALU_src, mem_rd, mem_wr, WB_data,ALU_op,push,pop,stop_bit);
    input [1:0] typ;
    input [4:0] func;
    input stop_bit;
    output reg [1:0]pc_in;
    output reg reg_src,reg_write,sign_ext,Imm_sel,ALU_src,mem_rd,mem_wr,WB_data,push,pop;
    output reg[2:0] ALU_op;

    always @(*)
        begin
            if (typ==2'b00)  //R-type
                begin
                    if (stop_bit == 0)
                        begin
                        pc_in =0;
                        push = 0;
                        pop = 0;
                        end
                    else
                        begin
                        pop = 1;
                        push = 0;
                        pc_in =3;
                        end
                    reg_src = 0;
                    reg_write = 1;
                    sign_ext = 0; //dont care
                    Imm_sel = 0; //dont care
                    ALU_src = 0;
                    mem_rd = 0;
                    mem_wr= 0;
                    WB_data = 0;
                    case(func)
                        5'b00000: ALU_op = 3'b000; //AND
                        5'b00001: ALU_op = 3'b001; //ADD
                        5'b00010: ALU_op = 3'b010; // SUB
                        5'b00011: ALU_op = 3'b011; //CMP
                    endcase
                end
            else if (typ == 2'b01) // I-type
                begin
                case(func)
                        5'b00000 : begin // ANDI
                            if (stop_bit == 0)
                                begin
                                    pc_in =0;
                                    push = 0;
                                    pop = 0;
                                end
                            else
                                begin
                                begin
                                    pop = 1;
                                    push = 0;
                                    pc_in =3;
                                end
                            reg_src = 0; //don't care
                            reg_write = 1;
                            sign_ext = 0;
                            Imm_sel = 0;
                            ALU_src = 1;
                            mem_rd = 0;
                            mem_wr= 0;
                            WB_data = 0;
                            ALU_op = 3'b000; //AND
                        end

                        5'b00001 : // ADDI
                        begin
                            if (stop_bit == 0)
                                begin
                                    pc_in =0;
                                    push = 0;
                                    pop = 0;
                                end
                            else
                                begin
                                    pop = 1;
                                    push = 0;
                                    pc_in =3;
                                end
                            reg_src = 0; //don't care
                            reg_write = 1;
                            sign_ext = 1;
                            Imm_sel = 0;
                            ALU_src = 1;
                            mem_rd = 0;
                            mem_wr= 0;
                            WB_data = 0;
                            ALU_op = 3'b001; //ADD
                        end
```

```
                5'b00010 : // LW
                begin
                     if (stop_bit == 0)
                          begin
                               pc_in =0;
                               push = 0;
                               pop = 0;
                          end
                     else
                          begin
                               pop = 1;
                               push = 0;
                               pc_in =3;
                          end
                     reg_src = 0; //don't care
                     reg_write = 1;
                     sign_ext = 1;
                     Imm_sel = 0;
                     ALU_src = 1;
                     mem_rd = 1;
                     mem_wr= 0;
                     WB_data = 1;
                     ALU_op = 3'b001; //ADD
                end

                5'b00011 : // SW
                begin
                     if (stop_bit == 0)
                          begin
                               pc_in =0;
                               push = 0;
                               pop = 0;
                          end
                     else
                          begin
                               pop = 1;
                               push = 0;
                               pc_in =3;
                          end
                     reg_src = 1; //Rd
                     reg_write = 0;
                     sign_ext = 1;
                     Imm_sel = 0;
                     ALU_src = 1;
                     mem_rd = 0;
                     mem_wr= 1;
                     WB_data = 0; // don't care
                     ALU_op = 3'b001; // ADD
                end


                5'b00100 : // BEQ
                begin
                     pc_in = 1;
                     reg_src = 1;
                     reg_write = 0;
                     sign_ext = 1;
                     Imm_sel = 0;
                     ALU_src = 0;
                     mem_rd = 0;
                     mem_wr= 0;
                     push =0;
                     pop = 0;
                     WB_data = 0; // don't care
                     ALU_op = 3'b100; //BEQ
                end
          endcase
          end
     else if (typ == 2'b10) //J-type
               begin
               pc_in = 2;
               reg_src = 0;//dont care
               reg_write = 0; //dont care
               sign_ext = 0; //dont care
               Imm_sel = 0; //dont care
               ALU_src = 0; //dont care
               mem_rd = 0;    //dont care
               mem_wr= 0;     //dont care
               WB_data = 0; //dont care
               ALU_op = 3'b000;  //don't care
               if (func == 5'b00000) //J
                    begin
                         push=0;
                         pop =0;
                    end
               else
                    begin
                         push = 1;
                         pop = 0;
                    end
                end
     else if (typ == 2'b11) //S-type
               casex(func)
                         5'b0000x :
                         begin
                         if (stop_bit == 0)
                              begin
                                   pc_in =0;
                                   push = 0;
                                   pop = 0;
                              end
                         else
                              begin
                                   pop = 1;
                                   push = 0;
```

```
                    push = 0;
                    pc_in =3;
               end
          reg_src = 0; //dont care
          reg_write = 1;
          sign_ext = 0;
          Imm_sel = 1;
          ALU_src = 1;
          mem_rd = 0;
          mem_wr= 0;
          WB_data = 0;
          case(func[0])
               1'b0 : ALU_op = 3'b101; //SLL
               1'b1 : ALU_op = 3'b110; //SLR
          endcase
          end

          5'b0001x :
          begin
          if (stop_bit == 0)
               begin
                    pc_in =0;
                    push = 0;
                    pop = 0;
               end
          else
               begin
                    pop = 1;
                    push = 0;
                    pc_in =3;
          end
          reg_src = 0;
          reg_write = 1;
          sign_ext = 0; //dont care
          Imm_sel = 0;  //dont care
          ALU_src = 0;
          mem_rd = 0;
          mem_wr= 0;
          WB_data = 0;
          case(func[0])
               1'b0 : ALU_op = 3'b101; //SLL
               1'b1 : ALU_op = 3'b110; //SLR
          endcase
          end
     endcase
else
     begin
          // in order to handle irregular register or memory write and read
          mem_rd = 0;
          mem_wr = 0;
          reg_write = 0;
     end
end

endmodule
```

Figure 20: CU

It works exactly as the theory part. At the first, **the type** of the instruction checked and then according to the type, signals asserted as output. These signals are very important for the correct execution of the instruction in the Datapath.

The implementation of ALU Control unit fits exactly the table shown in CU design. All instruction types given the correct signals for the correct execution in the Datapath.

3.3.6: Register File

```
module RegisterFile (ra,rb,rw,busW,regWrite,busA,busB,input clk);
    input regWrite;
    input [4:0]ra,rb,rw;
    input [31:0]busW;
    output [31:0]busA,busB;
   // 32 registers, each 32 bits wide
   reg [31:0] registers [31:0];


   // Write operation on rising edge of the clock
   always @(posedge clk) begin
       if (regWrite)
           registers[rw]  <= busW;
    end

    assign busA = registers[ra];
    assign busB = registers[rb];
    initial begin
        $monitor("At Time =%t ----> R0=%h, R1=%h, R2=%h, R3=%h, R4=%h, R5=%h, R6=%h",$time,registers[0],registers[1],registers[2],registers[3],registers[4],
        registers[5],registers[6]);
        end
    endmodule
```

Figure 21: Register File

The register file implemented as described in theory part. I displayed the contents of **R0-R6**. More registers contents can be displayed. It is synchronized with the clock and performs read or write operations.

3.3.7: ALU

```
module ALU(i1, i2, ALU_op, ALU_out,zero);
    input [31:0]i1,i2;
    input [2:0] ALU_op;
    output reg [31:0] ALU_out;
    output reg zero;

    always @(*) begin
        case (ALU_op)
            5'b000:  ALU_out = i1 & i2;   //AND
            5'b001: ALU_out = i1 + i2;  //ADD
            5'b010: ALU_out = i1-i2;   //SUB
            5'b011: begin       //CMPLS
                if(i1 < i2)   begin
                    ALU_out= i1-i2;      //
                    zero = 1;
                    end
                else   begin
                    ALU_out = i1-i2;
                    zero=0;
                    end
                end
            5'b100: begin  //EQ
                if(i1 == i2)  begin
                    ALU_out= i1-i2;
                    zero = 1;
                    end
                else  begin
                    ALU_out = i1-i2;
                    zero=0;
                    end
                end
            5'b101: ALU_out = i1 << i2; //SLL
            5'b110: ALU_out = i1 >> i2; //SLR
        endcase
    end
endmodule
```

Figure 22: ALU Code

The ALU implemented in behavioral model, according to the **ALU_op** bits that come from the control unit, the operation performed on the inputs **in1** and **in2** and the result placed in **ALU_out**. Note that some operations required asserting signals such as Zero signal. We didn't need the Carry and Negative signals in CMP and BEQ operations. They performed directly using if statement(behavioral).

3.3.8: Extender

```
module Extender (
    input [13:0] imm, // 14-bit immediate
    input [4:0] sa, // 5-bit shift amount
    input imm_sa_sel,
    input signOp,
    output reg [31:0] extended_output
);
    always @(imm, sa, imm_sa_sel, signOp) begin
        if (imm_sa_sel) begin
            extended_output[4:0] = sa;
            if (signOp)
                extended_output[31:5] = {27{sa[4]}};
            else
                extended_output[31:5] = {27'b0};
        end else begin
            extended_output[13:0] = imm;
            if (signOp)
                extended_output[31:14] = {18{imm[13]}};
            else
                extended_output[31:14] = {18'b0};
        end
    end

endmodule
```

Figure 23: Extender Code

3.3.9: NextPCControlUnit

```
module NextPCControlUnit(pc_in,zero,PCSrc,out_pc,PCin0,PCin1,PCin2,PCin3,extended_imm14,imm24,ra);      //must be delayed to be finish after ALU end its operation
    input [1:0]pc_in;
    input zero;
    input [31:0] out_pc;
    input [31:0] ra;
    input [31:0]extended_imm14;
    input [23:0]imm24;
    output reg[1:0]PCSrc;
    output reg [31:0]PCin0,PCin1,PCin2,PCin3;

    // check the pc_in

    always @(*)
        begin
            if(pc_in == 1)            //Branch case
                begin
                    if (zero)
                        PCSrc = pc_in;
                    else
                        PCSrc = 0;
                end
            else
                PCSrc = pc_in;
        end

    assign PCin0 = out_pc + 1,
    PCin1 = out_pc + 1 + extended_imm14,
    PCin2 =  {out_pc[31:24],imm24},
    PCin3 = ra;


endmodule
```

Figure 24: NextPCControlUnit Code

3.3.10: Muxes

```
module mux2to1(i0,i1,s,f);
    input [31:0]i0,i1;
    input s;
    output reg [31:0] f;

    always @(s or i0 or i1) begin
        if(s)
            f = i1;
        else
            f=i0;
    end
endmodule
//------------------------------

module mux2to1_5bit(i0,i1,s,f);
     input [4:0]i0,i1;
    input s;
    output reg [4:0] f;

    always @(s or i0 or i1) begin
        if(s)
            f = i1;
        else
            f=i0;
    end
endmodule
```

Figure 25: Muxes Code

3.3.11: Data Memory

```
module DataMemory (clk,reset,mem_read,mem_write,data_in,address,data_out);
    input mem_read,mem_write,reset,clk;
    input [31:0] address,data_in;
    output reg [31:0] data_out;
    reg [31:0] memory [511:0]; // 512 cell , 32 bit each (smaller memory)

    // perform operations read,write,reset or noop
    always @(negedge clk) begin
        if (!reset)
            memory = '{512{32'd0}};
        else
            begin
                if (mem_read == 1)
                    data_out = memory[address];
                else if (mem_write == 1)
                    memory[address] = data_in;
            end

        end
    initial begin
        $monitor("Time = %t , Mem[0] = %h, Mem[1] = %h",$time,memory[0],memory[1]);
        end
    endmodule
```

Figure 26: Data memory Code

The data memory implemented as described in the theory part about it. I displayed the contents of **mem[0]** and **mem[1],** note that any contents can be displayed as the demands. The memory synchronized at the ***negative edge*** of the clock to make the flow of microoperations correct in the single cycle implementation.

Also, I supposed the memory to be **512 cells**, with **32-bit contents** for each cell for simplicity. So, the memory is not byte addressable, but **word addressable**.

3.3.12: Stack Memory

```
module stack_memory(clk,push,pop,ra_in,ra_out);
    input clk,push,pop;
    input [31:0]ra_in;
    reg [4:0]sp =0 ;
    output reg [31:0]ra_out;

    parameter depth = 32;
    reg [31:0] stack [0:depth-1];
    // perform operations read,write,reset or noop
     always @(negedge clk) begin
       if (push && sp < depth) begin
           // Push data onto the stack
           stack[sp] <= ra_in;
           sp <= sp + 1;
       end else if (pop && sp > 0) begin
           // Pop data from the stack
           sp <= sp - 1;
       end
    end
    assign ra_out = (sp > 0) ? stack[sp-1] : 32'b0;
    endmodule
```

Figure 27: Stack Code

It is implemented exactly as the theory part. Note that the depth of the stack is **32 cells** so at maximum 32 return addresses can be pushed to the stack. This depth is just a parameter and can be increased as needed if data will be pushed to this stack not only the return addresses.

3.3.13: Datapath Connections

Now after each component written in Verilog. All components were connected as shown in Datapath design part. The connection was done with three stages, stage one of the connection was preparing the wires and reg variables to make the connection. The second stage was connecting the components together, and the final stage was making the correct synchronization of elements at the edge of the clock for correct single cycle CPU implementation.

```
//------------------------------Connect Datapath --------------------------------
module DataPath;
    //clock and reset (basic inputs for the datapath)
    reg clk;
    reg reset;
    reg init;

// ********** DECODE *****************
    // pc input address and output addressed
    wire [31:0] input_address;
    wire [31:0] output_address;

    // input for Instruction memory and output
       // input is pc_out
    wire [31:0] instruction;

    // input and output for instruction decode unit
       // input is instruction
    wire stop;
    wire [1:0]typ;
    wire [4:0] func,rs1,rs2,rd,sa5;
    wire [13:0]imm14;
    wire [23:0]imm24;

    // input and output of control unit -----> DELAY MUST BE AFTER HERE
         // input for cu is type and func,stop
    wire [1:0]pc_in;
    wire reg_src,reg_write,sign_ext,Imm_sel,ALU_src,mem_rd,mem_wr,WB_data,push,pop;
    wire [2:0] ALU_op;


    // call the stack
          //input of stack push ,pop ,output_address +1,clk
    wire [31:0] ra_out;
```

```
    // input and outputs for extender
        // input is sa5,imm14,sign_ext,Imm_sel
    wire [31:0] extended_output;

    // input and outputs for mux before rb
        //inputs are rd,rs2 and reg_src
    wire [4:0] rb; //output of mux

    // input and outputs for RegFile
        // inputs are ra,rb,rw,busW,regWrite
        wire [31:0]busW;
    wire [31:0]busA,busB;    //outputs of reg file


    // input and output of mux before ALU
         // input is extended output,busB,ALU_src
    wire [31:0]ALU_in2; //output of mux

    // input and output of ALU
          //inputs are busA and ALU_in2 and ALU_op
    wire [31:0] ALU_out;
    wire zero;

    // input for NextPCcontrolunit and output
        // inputs are zero,pc_in,output_address,imm24,extended_output,ra_out
    wire[1:0]PCSrc;
    wire [31:0]PCin0,PCin1,PCin2,PCin3;;

    //inputs for PCMux
        //inputs are PCSrc,PCin0,PCin1,PCin2,PCin3
        // output is the input address for pc


    // inputs for Data Memory and outputs
        //inputs are ALU_out and mem_rd,mem_wr and busB
    wire [31:0] data_out;
```

```
  // inputs for WBMux and outputs
      // inputs are data_out,ALU_out,WB_data
      // output is busW
```

```
PCReg pc1(
    .init(init),
    .clk(clk),
    .reset(reset),
    .input_address(input_address),
    .output_address(output_address)
);


InstructionMemory im1(
         .address(output_address),
         .instruction(instruction));


InstructionDecodeUnit id1(
        .instruction(instruction),.typ(typ),
        .stop(stop),.func(func),
        .rs1(rs1),.rs2(rs2),
        .rd(rd),.imm14(imm14),
        .sa5(sa5),.imm24(imm24));


ControlUnit cu1(
        .typ(typ),.func(func),
        .pc_in(pc_in),.reg_src(reg_src),
        .reg_write(reg_write),.sign_ext(sign_ext),
        .Imm_sel(Imm_sel), .ALU_src(ALU_src),
        .mem_rd(mem_rd), .mem_wr(mem_wr),
        .WB_data(WB_data),
        .ALU_op(ALU_op),
        .pop(pop),
        .push(push),
        .stop_bit(stop));
```

```
stack_memory  st1(.clk(clk),
.push(push),.pop(pop),
.ra_in(output_address + 1),
.ra_out(ra_out));

Extender ex1(
    .imm(imm14),
    .sa(sa5),
    .imm_sa_sel(Imm_sel),
    .signOp(sign_ext),
     .extended_output(extended_output)
);


mux2to1_5bit m1(
.i0(rs2),
.i1(rd),
.s(reg_src),
.f(rb));


RegisterFile rf1(
.clk(clk),
.ra(rs1),.rb(rb),
.rw(rd),.busW(busW),
.regWrite(reg_write),.busA(busA),
.busB(busB));



mux2to1 m2(
.i0(busB),
.i1(extended_output),
.s(ALU_src),
.f(ALU_in2));
```

```
ALU  al1(
.i1(busA),
.i2(ALU_in2),
.ALU_op(ALU_op),
.ALU_out(ALU_out),
.zero(zero));


NextPCControlUnit npc1(
.pc_in(pc_in),.zero(zero),
.PCSrc(PCSrc),.out_pc(output_address),
.PCin0(PCin0),.PCin1(PCin1),
.PCin2(PCin2),.extended_imm14(extended_output),
.imm24(imm24),.PCin3(PCin3),.ra(ra_out));


PCMux  pcm1(
.in0(PCin0),.in1(PCin1),
.in2(PCin2),.PCSrc(PCSrc),
.in3(PCin3),
.address(input_address));

DataMemory  dm1(
.clk(clk),
.reset(reset),.mem_read(mem_rd),
.mem_write(mem_wr),.data_in(busB),
.address(ALU_out),.data_out(data_out));


mux2to1 m3(
.i0(ALU_out),
.i1(data_out),
.s(WB_data),
.f(busW));
```

```
initial begin
    clk = 0 ;
reset = 1;
init=1;


#400ns
init = 0;

#10000ns $finish;
end
always begin
    #200ns begin
        clk = ~clk;
end
end
endmodule
```

Figure 28: Datapath connection Code

As shown before, at the first stage all wires and registers were prepared for the inputs and outputs of the components. After that, all connections were made according to the Datapath design part. Finally, the synchronization process was done, all components were synchronized on the clock as follows:

+ve Edge CLK:

→ PC Register

→ Register File (Write Operation Only, Done on the next cycle pos edge)

-ve Edge CLK:

→ Data Memory

→ Stack Memory

All other components synchronized using the correct sequence of inputs and outputs.

4. Testing the Processor

Our processor was tested using groups of instructions that verify the functionality of whole datapath. Note that the Testing process was done by converting the instructions into binary and then hexadecimal, and the put these instructions in the Instruction Memory. After that the correct time needed was assigned for the simulation. Finally, the Test process was done, and the results printed on the terminal.

4.1: Important Modifications

After some testing, we noticed that:

- The Active HDL program puts the initial value of reg variable to be don't care(x) not zero(0), so the registers will be at the first (x).

- **The first problem** to solve this problem is to ANDING the register that we want to use with zero, since x AND 0 = 0. This works correctly but not practical.

- **The second solution** to solve this problem was that we used a variable in the Datapath module called **regfile_init** and gives it to the register file to indicate that this is the first time the program run, so initialize all registers with zero. This was done as the following:

```
module DataPath;
    //clock and reset (basic inputs for the dat
    reg clk;
    reg reset;
    reg init;
    reg regfile_init;
```

```
RegisterFile rf1(
.init(regfile_init),
.clk(clk),
.ra(rs1),.rb(rb),
.rw(rd),.busW(busW),
.regWrite(reg_write),.busA(busA),
.busB(busB));
```

```
module RegisterFile (ra,rb,rw,busW,regWrite,busA,busB,input clk,input init);
    input regWrite;
    input [4:0]ra,rb,rw;
    input [31:0]busW;
    output [31:0]busA,busB;
    // 32 registers, each 32 bits wide
    reg [31:0] registers [31:0];
    int i;

    // Write operation on rising edge of the clock
    always @(posedge clk) begin
        if (regWrite)
            registers[rw]  <= busW;
    end

    always @(init) begin
        if(init)
            for (i=0; i<32 ; i=i+1)
                registers[i] = 32'd0;
    end

    assign busA = registers[ra];
    assign busB = registers[rb];
    initial begin
        $monitor("At Time =%t ----> R0=%h, R1=%h, R2=%h, R3=%h, R4=%h, R5=%h,
```

Figure 29:RegFile change

4.2: Testing Groups

4.2.1: Testing Instructions One

The following subset of instructions was used at the first to verify the functionality of Datapath.

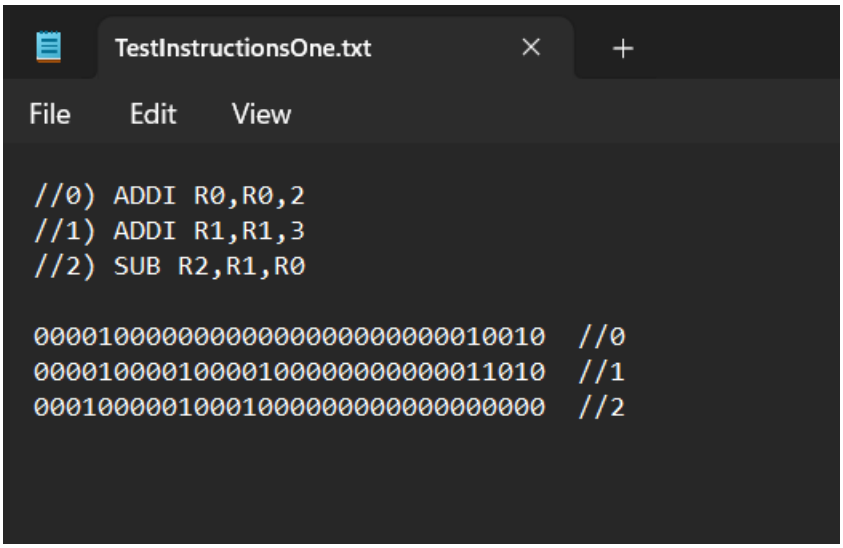


Figure 30: Test Instructions One

These instructions were put in the instruction memory at memory cells 0,1 and 2 and then the code was Compiled and the simulation Initialized.

```
//****************Instruction Memory***************
 module InstructionMemory (
  input [31:0] address,output reg [31:0] instruction);

  // Define the instruction memory
  reg [31:0] memory [255:0];

  // Fetch the instruction at the specified address
  always @(address) begin
    instruction = memory[address];
  end

  // Initialize the instruction memory
  initial begin
    // Load instructions into memory
     memory[0] = 32'b00001000000000000000000000010010;
      memory[1] = 32'b00001000010000100000000000011010;
     memory[2] = 32'b00010000010001000000000000000000;
```

Figure 31: initialize instruction Memory – One

And the results were as follows:

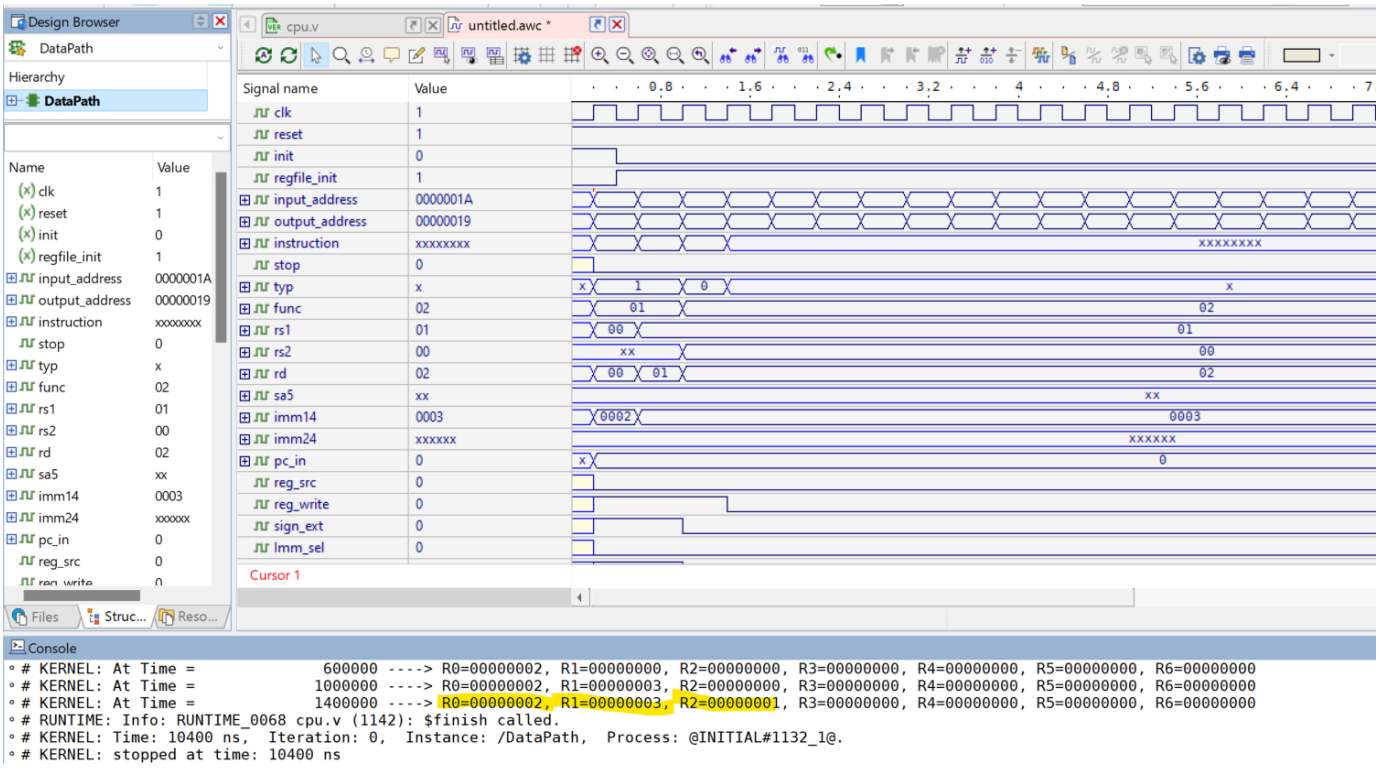


Figure 32: Result One

We noticed that the instructions worked correctly. **R0=2,R1=3 and R2=R1-R2=3-2=1.**

4.2.2: Testing Instructions Two

At this group, a subset of R-type, I-type and S-type instruction was tested together (with Store and Load).

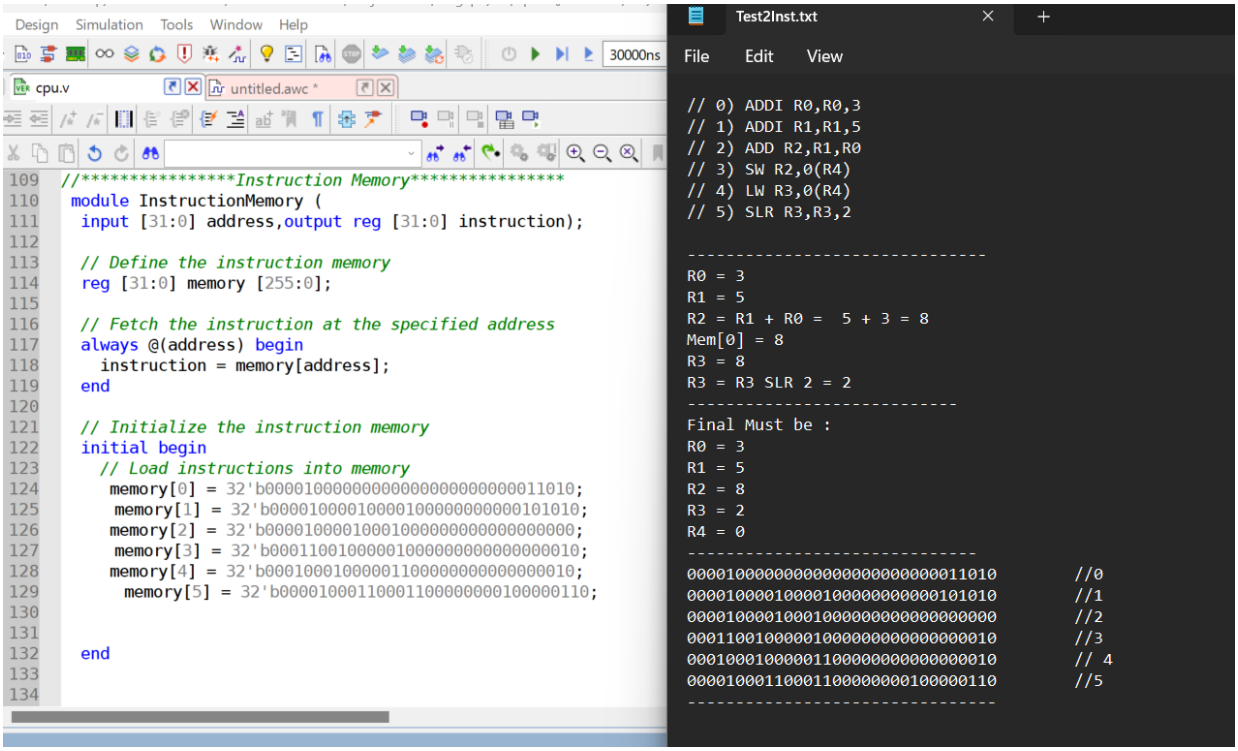


Figure 33: Test Instructions Two

And the result was correct as shown in the figure below:

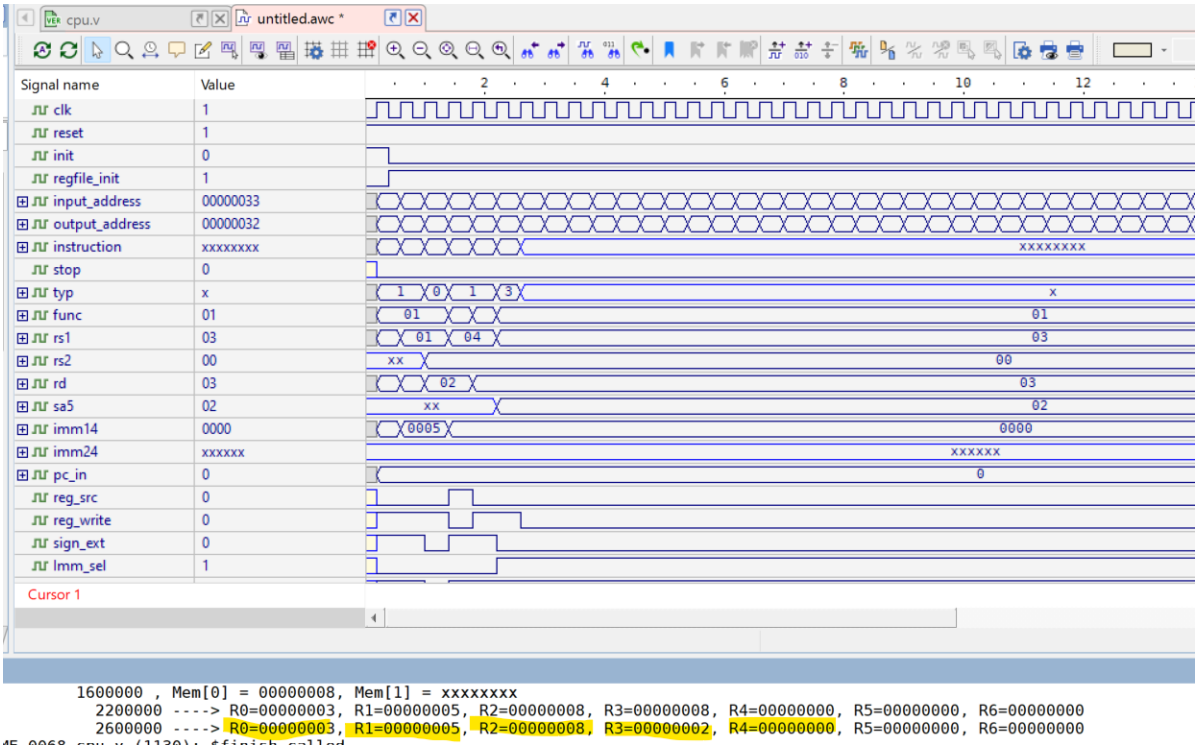


Figure 34: Result Two

4.2.3: Testing Instructions Three

In this group we tested the BEQ instruction with R-Type and I-Type instructions.

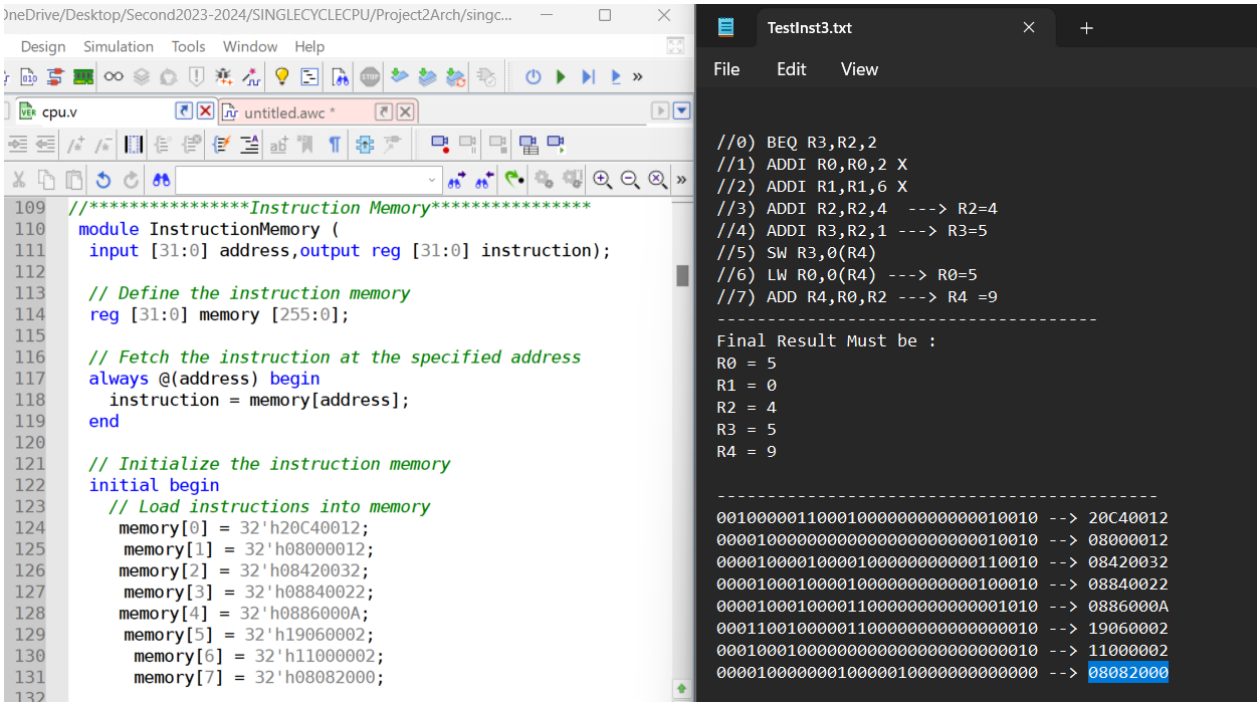


Figure 35: Test Instruction Three

And the result was correct as follows:

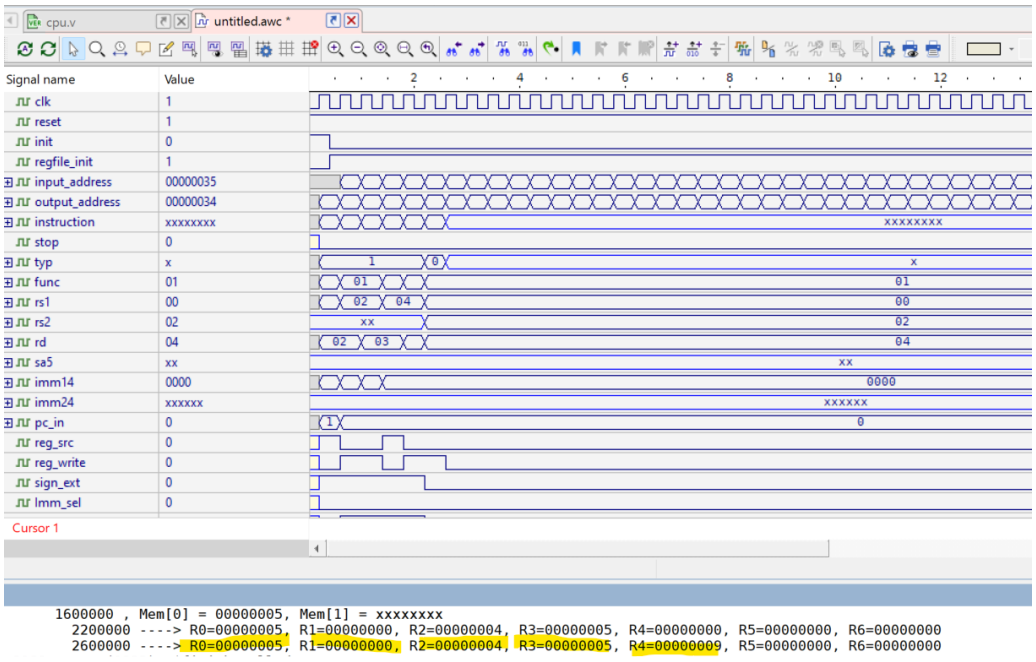


Figure 36: Result Three

4.2.4: Testing Instructions Four

In the following group we tested the functionality of **Stack Memory and Stop bit** by calling a function100 and then calling another function200 inside 100. The results were correct as the figures below show:

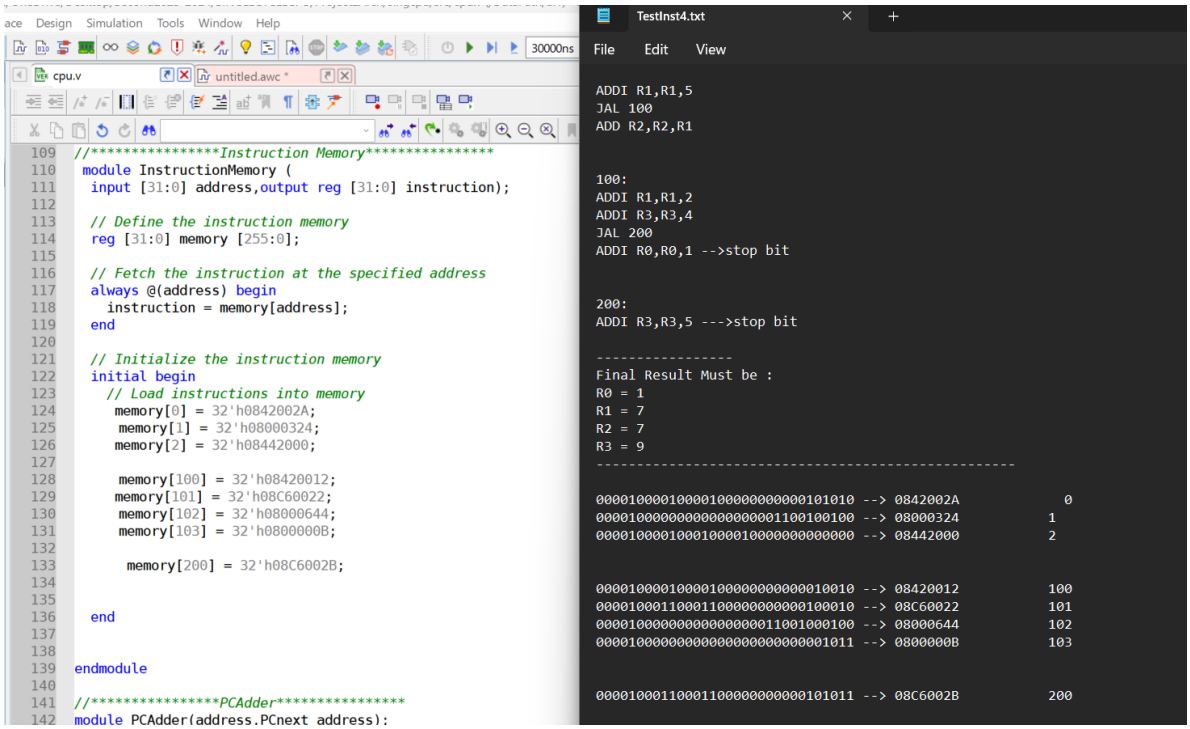


Figure 37: Test Instruction Four

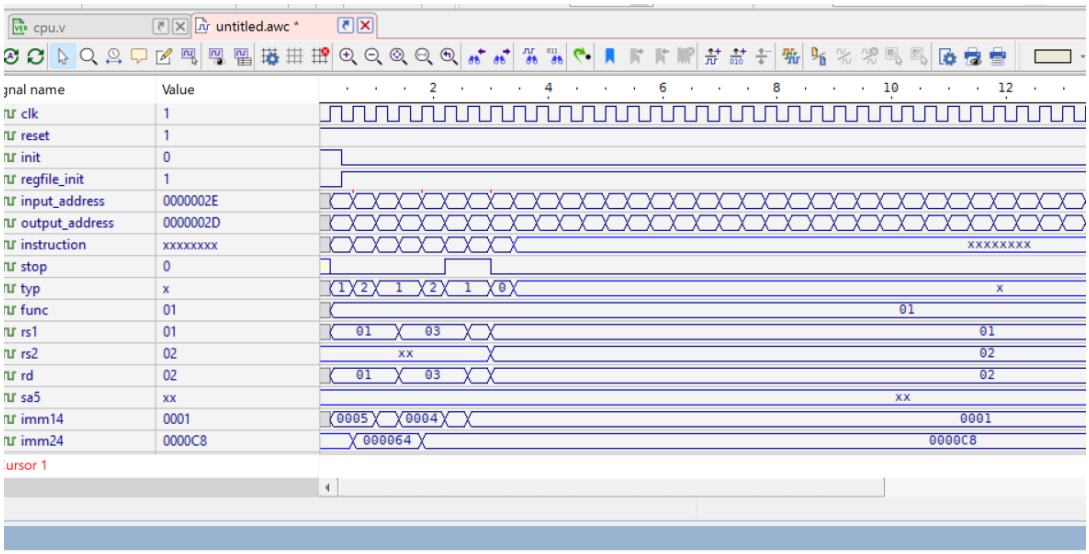
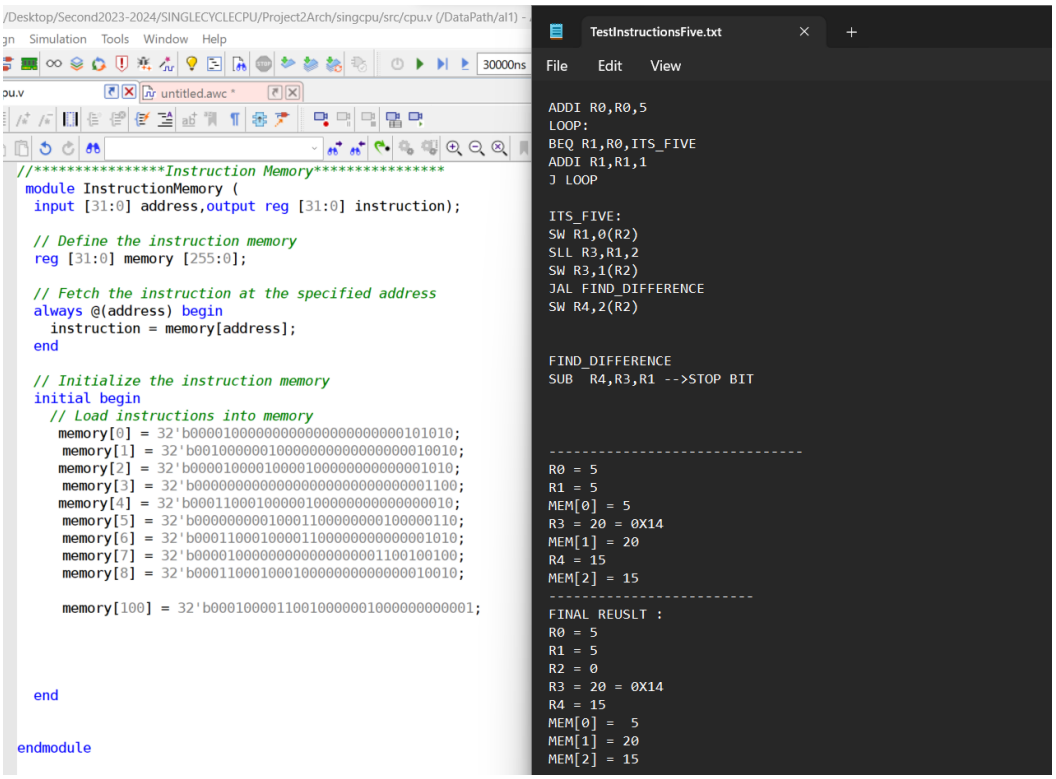


Figure 38: Result Four

4.2.5: Testing Instructions Five

In this group we tested every instruction format (R-type, I-type, J-type, and S-type). And the result was correct as shown in the figures below:



```
//****************Instruction Memory****************
module InstructionMemory (
 input [31:0] address,output reg [31:0] instruction);

 // Define the instruction memory
 reg [31:0] memory [255:0];

 // Fetch the instruction at the specified address
 always @(address) begin
   instruction = memory[address];
 end

 // Initialize the instruction memory
 initial begin
   // Load instructions into memory
    memory[0] = 32'b00001000000000000000000000101010;
    memory[1] = 32'b00100000100000000000000000010010;
    memory[2] = 32'b00001000010000100000000000001010;
    memory[3] = 32'b00000000000000000000000000001100;
    memory[4] = 32'b00011000100000100000000000000010;
    memory[5] = 32'b00000000001000110000000100000110;
    memory[6] = 32'b00011000100001100000000000001010;
    memory[7] = 32'b00001000000000000000000001100100;
    memory[8] = 32'b00011000100010000000000000010010;

    memory[100] = 32'b00010000110010000001000000000001;



 end


endmodule
```

```
ADDI R0,R0,5
LOOP:
BEQ R1,R0,ITS_FIVE
ADDI R1,R1,1
J LOOP

ITS_FIVE:
SW R1,0(R2)
SLL R3,R1,2
SW R3,1(R2)
JAL FIND_DIFFERENCE
SW R4,2(R2)


FIND_DIFFERENCE
SUB  R4,R3,R1 -->STOP BIT



------------------------------
R0 = 5
R1 = 5
MEM[0] = 5
R3 = 20 = 0X14
MEM[1] = 20
R4 = 15
MEM[2] = 15
-------------------------
FINAL REUSLT :
R0 = 5
R1 = 5
R2 = 0
R3 = 20 = 0X14
R4 = 15
MEM[0] =  5
MEM[1] = 20
MEM[2] = 15
```

Figure 39: Test Instructions Five

```
: At Time =                  0 ----> R0=xxxxxxxx, R1=xxxxxxxx, R2=xxxxxxxx, R3=xxxxxxxx, R4=xxxxxxxx, R5=xxxxxxxx, R6=xxxxxxxx
: Time =                     0 , Mem[0] = xxxxxxxx, Mem[1] = xxxxxxxx , Mem[2] = xxxxxxxx
: At Time =             400000 ----> R0=00000000, R1=00000000, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =             600000 ----> R0=00000005, R1=00000000, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =            1400000 ----> R0=00000005, R1=00000001, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =            2600000 ----> R0=00000005, R1=00000002, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =            3800000 ----> R0=00000005, R1=00000003, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =            5000000 ----> R0=00000005, R1=00000004, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: At Time =            6200000 ----> R0=00000005, R1=00000005, R2=00000000, R3=00000000, R4=00000000, R5=00000000, R6=00000000
: Time =               7200000 , Mem[0] = 00000005, Mem[1] = xxxxxxxx , Mem[2] = xxxxxxxx
: At Time =            7800000 ----> R0=00000005, R1=00000005, R2=00000000, R3=00000014, R4=00000000, R5=00000000, R6=00000000
: Time =               8000000 , Mem[0] = 00000005, Mem[1] = 00000014 , Mem[2] = xxxxxxxx
: At Time =            9000000 ----> R0=00000005, R1=00000005, R2=00000000, R3=00000014, R4=0000000f, R5=00000000, R6=00000000
: Time =               9200000 , Mem[0] = 00000005, Mem[1] = 00000014 , Mem[2] = 0000000f
E: Info: RUNTIME_0068 cpu.v (1137): $finish called.
: Time: 20400 ns,  Iteration: 0,  Instance: /DataPath,  Process: @INITIAL#1127_1@.
```

Figure 40: Result Five

NOTE:

As you can see, the implementation of single cycle CPU is correct. All testing groups of instructions worked correctly without any error. Notice that the synchronization process also made the execution correct and met the requirements of single cycle CPU, where at the positive edge the address read from PC and then instruction read from Instruction Memory. After that, the instruction decoded in IR and then the other stages of its execution. Note that the register file write operation worked at the positive edge of the next cycle, this because we may need the data from memory, so the memory finished its operation at the negative edge of Cycle (i) and then the write done on positive edge of Cycle(i+1)

5.Teamwork

We worked as an integrated team on all parts of the project, from designing to writing the code and finally testing.

Where the first member built the first part of the data path and verified its validity, and then the second member built the other part and verified it also. Hence ideas were proposed on the construction of the control unit and then the design process started.

At the stage of writing the code, the roles were divided among each member of the team, and finally the testing and verification process took place.

6.Comments and Conclusion

The aim of the project is already done. We designed, implemented, and verified a simple RISC processor using VERILOG HDL. Our processor was implemented using a single cycle CPU approach. We started grouping the instructions share the same Datapath. Then, we wrote the RTL for each group. After that, the components and transfer of data was taken from RTL, and we mentioned all needed components of the Datapath. After that, we designed the Datapath and Control Unit for the correct execution of the instructions we have.

After that, we implemented all design parts in VERILOG, each component tested alone and then all connected.

Finally, we made some testing group instructions for testing the whole functionality and verify our design. All groups worked correctly without any error. All things correct.

We learned more about how single cycle CPU is actually implemented and how to derive and connect all components together in order to implement a specific ISA.

7.References

[1] Computer Architecture Single Cycle CPU Slides, BZU.