Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Computer Architecture ENCS4370

Project #2

Prepared by

Maha Mali 1200746

Lama Nasser 1200190

Basheer Arouri 1201141

Instructor:Dr.Aziz Qaroush

Section:1

Date: 18-6– 2023

Table of Contents

List of Figures

1. Designs and Testing for Stages

1.1. Instruction Memory

The Instruction Memory, or Instruction Cache, is a vital component in a computer system's data path. It stores and provides the instructions for execution, ensuring fast access and efficient processing. It plays a crucial role in program execution.

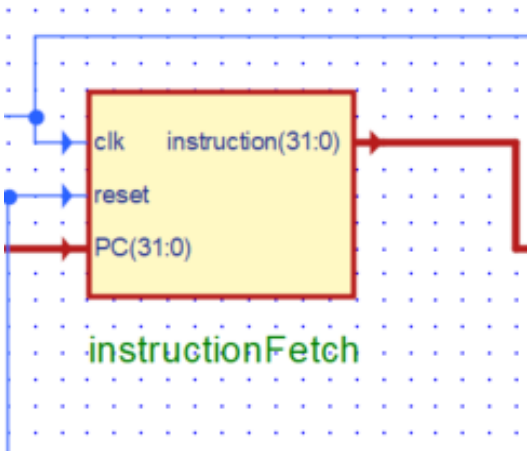


Figure 1: Instruction Memory

1.2. MUX 4X1

The MUX 4x1, or 4-to-1 multiplexer, is a component commonly used in the data path of a computer system. It selects one input signal out of four based on a select signal. In the data path, it is employed for tasks such as instruction decoding and register file access. It allows for signal selection and routing, contributing to the flexibility and functionality of the data path.

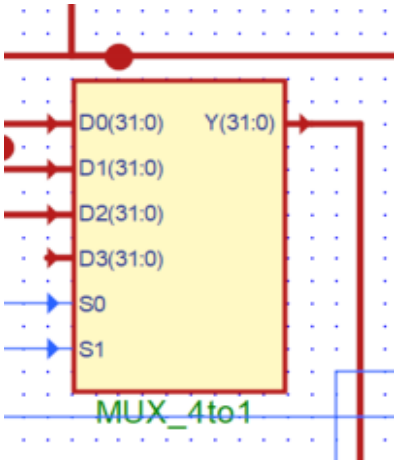


Figure 2: MUX 4x1

1.3. MUX 2X1

The MUX 2x1, or 2-to-1 multiplexer, is a vital component in the data path of a computer system. It selects one input signal out of two based on a select signal. In the data path, it is commonly used for tasks like data selection and routing. It allows for choosing between different data sources based on control signals, enhancing the flexibility and functionality of the data path.

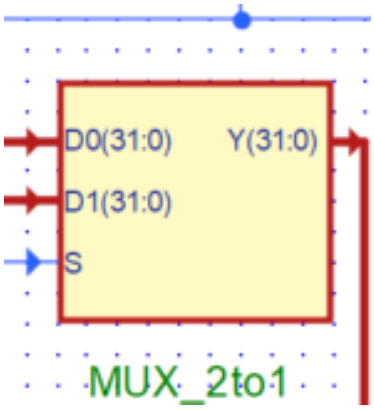


Figure 3:MUX 2x1

1.4. PC Register

The PC (Program Counter) Register is a vital component in the data path of a computer system. It stores the memory address of the next instruction to be fetched and executed. As part of the control flow, it determines the sequence of instructions and is updated during the fetch stage. It allows for branching and jumping instructions, enabling control flow changes within the program. The PC Register plays a crucial role in maintaining the execution flow of the program.

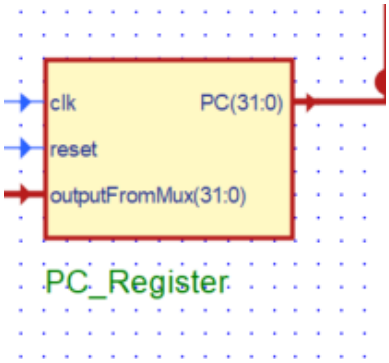


Figure 4: PC Register

1.5. Branch Target Address

The branch target address is the memory address of the instruction to which a branch instruction transfers control during program execution. The branch target address is calculated using a combination of the program counter (PC) and the immediate value provided in the branch instruction.

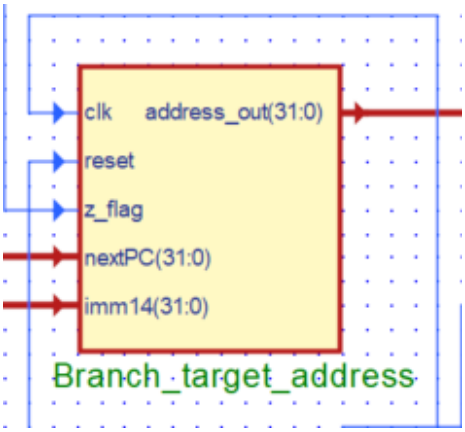


Figure 5: branch target address

1.6. Adder Unite

The adder circuit is a fundamental component of the arithmetic logic unit (ALU), responsible for performing the addition operation on binary numbers. It is a digital circuit that takes two binary inputs and produces a sum output.

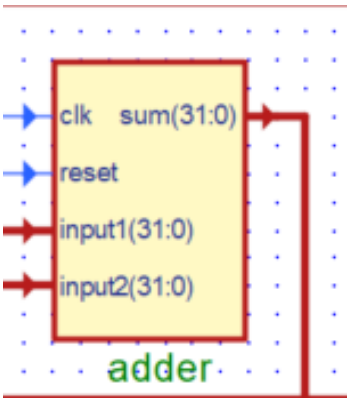


Figure 6:Adder Unite

1.7. Register File

Register file consists of a set of registers which are used to stage data between memory and the functional units, in addition to the circuits that are responsible for writing and reading from these registers. Also, the register file's inputs and outputs, which included the destination registers and the source registers, were already set.

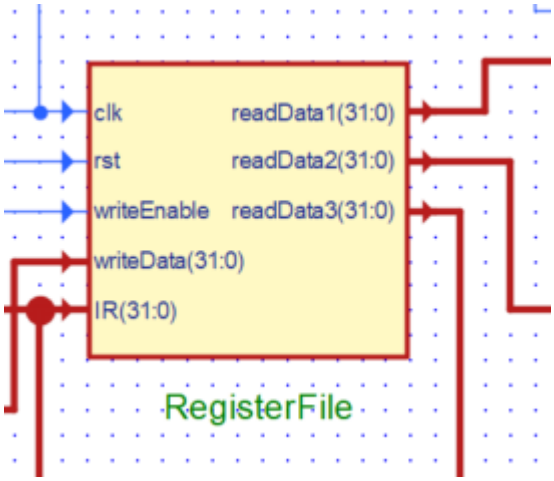


Figure 7: Register file

1.8. ALU Stage

During this phase, the necessary operations for each instruction were executed by incorporating the appropriate opcode and condition bit. Additionally, any necessary extensions were implemented to facilitate the execution of arithmetic or logic operations. As shown in the figure below the ALU Control Unit was designed to choose the required operation in order to execute our instruction.

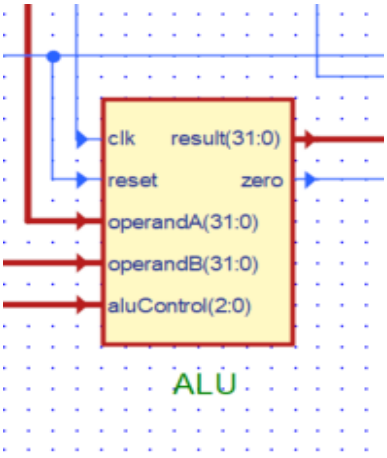


Figure 8:ALU

1.9. Data Memory stage

The Data Memory stage is a crucial component in the data path of a computer system. It facilitates the reading and writing of data to and from the main memory. During this stage, data is transferred between the processor and memory subsystem. Operations such as fetching data, storing data, and performing necessary conversions are performed. The Data Memory stage plays a vital role in executing instructions that involve memory operations, ensuring correct data access and manipulation.

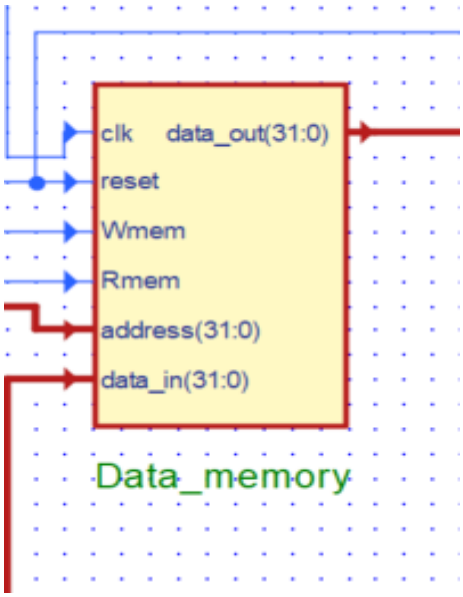


Figure 9:Data Memory

1.10. Stack unit

A stack unit with data_in, data_out, two control signals (push and pop), and a stack pointer. The stack follows the Last-In-First-Out (LIFO) principle, where the most recently pushed item is the first one to be popped. It is used to save the return address.

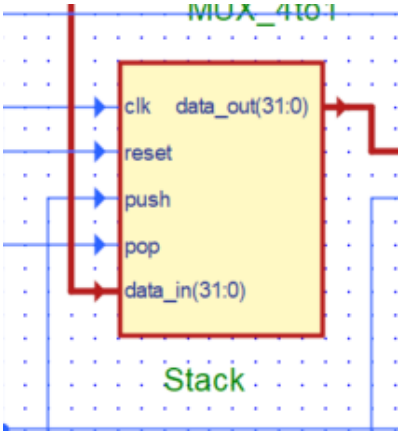


Figure 10: stack unit

1.11. Extender 14

The Extender 14 is an essential component in a data path when there is a need to increase the size of a data signal by 14 bits. It ensures data size compatibility, sign extension, compatibility with wider data paths, and enables increased range or precision in operations. By incorporating the Extender 14, the data path can seamlessly handle larger data sizes, maintain sign integrity, and accommodate complex calculations, enhancing overall efficiency and accuracy in data processing.

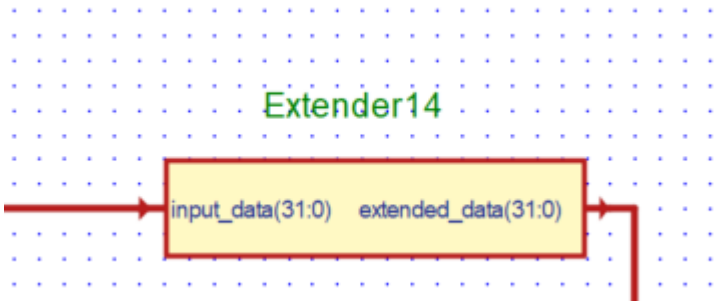


Figure 11: Extender 14

1.12. Extender 24

The Extender 24 is a crucial component in a data path when there is a need to expand the size of a data signal by 24 bits. It is utilized to align data sizes, extend sign bits, ensure compatibility with wider data paths, and enhance the range and precision of operations. By incorporating the Extender 24, the data path can seamlessly handle larger data sizes, preserve sign integrity, accommodate complex calculations, and achieve greater accuracy and precision in data processing.

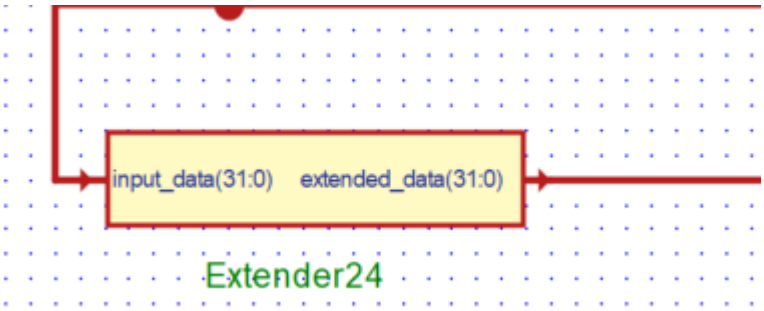


Figure 12: Extender 24

1.13. Register (Buffer Between Stages)

The Register (Buffer Between Stages) is an essential component in a multi-cycle data path for several reasons:

- Synchronization of operations: In a multi-cycle data path, different stages perform operations in separate clock cycles. The Register acts as a buffer between stages, allowing each stage to complete its operation before passing the data to the next stage. It synchronizes the flow of data between stages, ensuring that each stage receives the correct data at the appropriate time.

- Timing control: The Register helps in controlling the timing of data propagation between stages. It allows for proper sequencing of operations, ensuring that data is available and stable when needed by subsequent stages. This helps in avoiding data hazards, such as race conditions or data corruption, by providing a controlled and synchronized transfer of data.

- Preservation of intermediate results: In multi-cycle data paths, intermediate results are generated at various stages. The Register serves as a storage element, holding these intermediate results until they are required by subsequent stages. This allows for proper data dependency management and enables the correct flow of data through the data path.

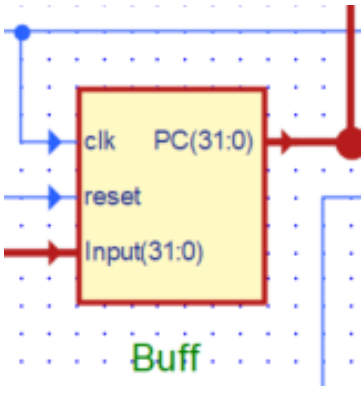


Figure 13:Buffer

1.14. Control Unit

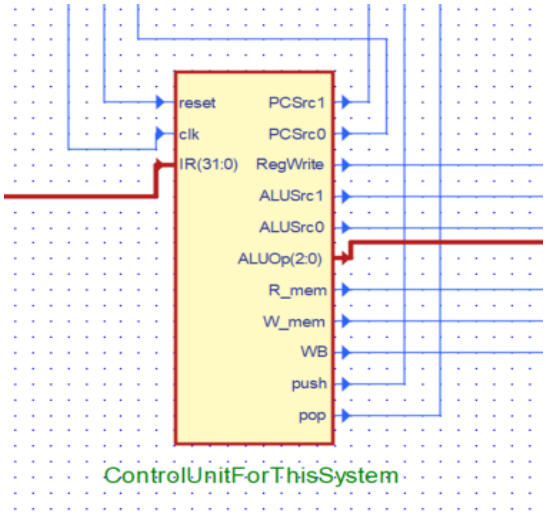


Figure 14:Control Unit

Here is the table for the control unit:

Table 1: control unit

instruction	Type	Function	PC_Cont1	PC_Cont2	push	RegW	Ext14	Ext24	ALUSR1	ALUSR2	ALUOP	Rmem	Wmem	Wdata
AND	00	00000	1	0	0	1	x	x	0	0	010	0	0	1
ADD	00	00001	1	0	0	1	x	x	0	0	000	0	0	1
SUB	00	00010	1	0	0	1	x	x	0	0	001	0	0	1
CMP	00	00011	1	0	0	0	x	x	0	0	001	0	0	x
ANDI	10	00000	1	0	0	1	1	x	0	1	010	0	0	1
ADDI	10	00001	1	0	0	1	1	x	0	1	000	0	0	1
LW	10	00010	1	0	0	1	1	x	0	1	000	1	0	0
SW	10	00011	1	0	0	0	1	x	0	1	000	0	1	x
BEQ	10	00100	0	0	0	0	1	x	1	1	001	0	0	x
J	01	00000	0	1	0	0	x	1	x	x	xxx	0	0	x
JAL	01	00001	0	1	1	0	x	1	x	x	xxx	0	0	x
SLL	11	00000	1	0	0	1	x	x	1	0	101	0	0	1
SLR	11	00001	1	0	0	1	x	x	1	0	110	0	0	1
SLLV	11	00010	1	0	0	1	x	x	0	0	101	0	0	1
SLRV	11	00011	1	0	0	1	x	x	0	0	110	0	0	1

PCcont1, PCcont2: when this control signal is equal to

- 00 then PC = Imm_B + PC
- 01 then PC = Imm_j + PC
- 10 then PC = PC + 4

Push: when this control signal is equal to

- 0 then stack[top] = PC + 4

RegW: when this signal is:

- 0 then writing on register will be performed else not.

Ext14: when this control signal is equal to:

- 0 then it extends the immediate number with zero bits. When it is equal to 1, it extends the number with ones. Otherwise (means X) it will do nothing.

Ext24: when this control signal is equal to:

- 0 then it extends the immediate number with zero bits. When it is equal to 1, it extends the number with ones. Otherwise (means X) it will do nothing

ALUSrc1,ALUSrc2: when the bits are

- 00: The source is Rs2
- 01: The source is the extended immediate value
- 10: The source is the shift amount for the S_Type
- 11: The source is Rd

ALUSrc1,ALUSrc2: when the bits are

- 000: The operation is ‘ADD’
- 001: The operation is ‘SUB’
- 010: The operation is ‘AND’
- 101: The operation is ‘SLL’
- 110: The operation is ‘SLR’
- xxx: it does notdo any operation

Rmem, Wmem: when this control signal is equal to

- 00: No read, no write on data memory
- 01: write on the data memory
- 10: read on the data memory

Wdata: when 0 the output of the data memory will be written back while 1 is for the ALU operations.

1.14.1. Expressions for Control Bits

PCcont1: (instruction == R-type) | (instruction == S-type)

PCcont2: instruction == J | instruction == JAL

Push: Instruction == JAL

RegW: instruction == AND | instruction == ADD | instruction == Sub | Instruction == ADDI | Instruction = ANdI | Instruction == LW | (Instruction == S-typr)

Ext14: (Instruction == I-type)

Ext24: (Instruction == J-type)

ALUsrc1: Instruction == BEQ | Instruction == SLL | Instruction == SLR

ALUsrc2: (Instruction == I-type)

ALU_OP[0]: (Instruction == S-type)

ALU_OP[1]: Instruction == AND | Instruction == ANDI | Instruction == SLR | Instruction == SLRV

ALU_OP[2]: Instruction == SUB | Instruction == CMP | Intruction == BEG | Instruction == SLL | Instruction == SLLV

Rmem: Instruction == LW

Wmem: Instruction == SW

Wdata: Instruction == AND | Instruction == ADD | Instruction == SUB | Instruction == ANDI | Instruction == ADDI | (Instruction == S-type)

1.14.2. The Finite State Machine

In short here are the stages are the states in which:

Instruction Fetch: The processor fetches the instruction from memory and loads it into the instruction register.

Instruction Decode: The processor decodes the instruction, determines the type of instruction and the operands needed, and stores the relevant information in internal registers.

Execution: The processor performs the necessary operations, which may involve multiple clock cycles, to complete the instruction.

Memory Access: If the instruction requires data to be read from or written to memory, the processor accesses the memory to perform the operation

Write Back: The processor writes the result of the operation back to the appropriate internal register or memory location.

The finite state machine below shows the transition between stages from fetch to write back and the behavior of each instruction respectively.

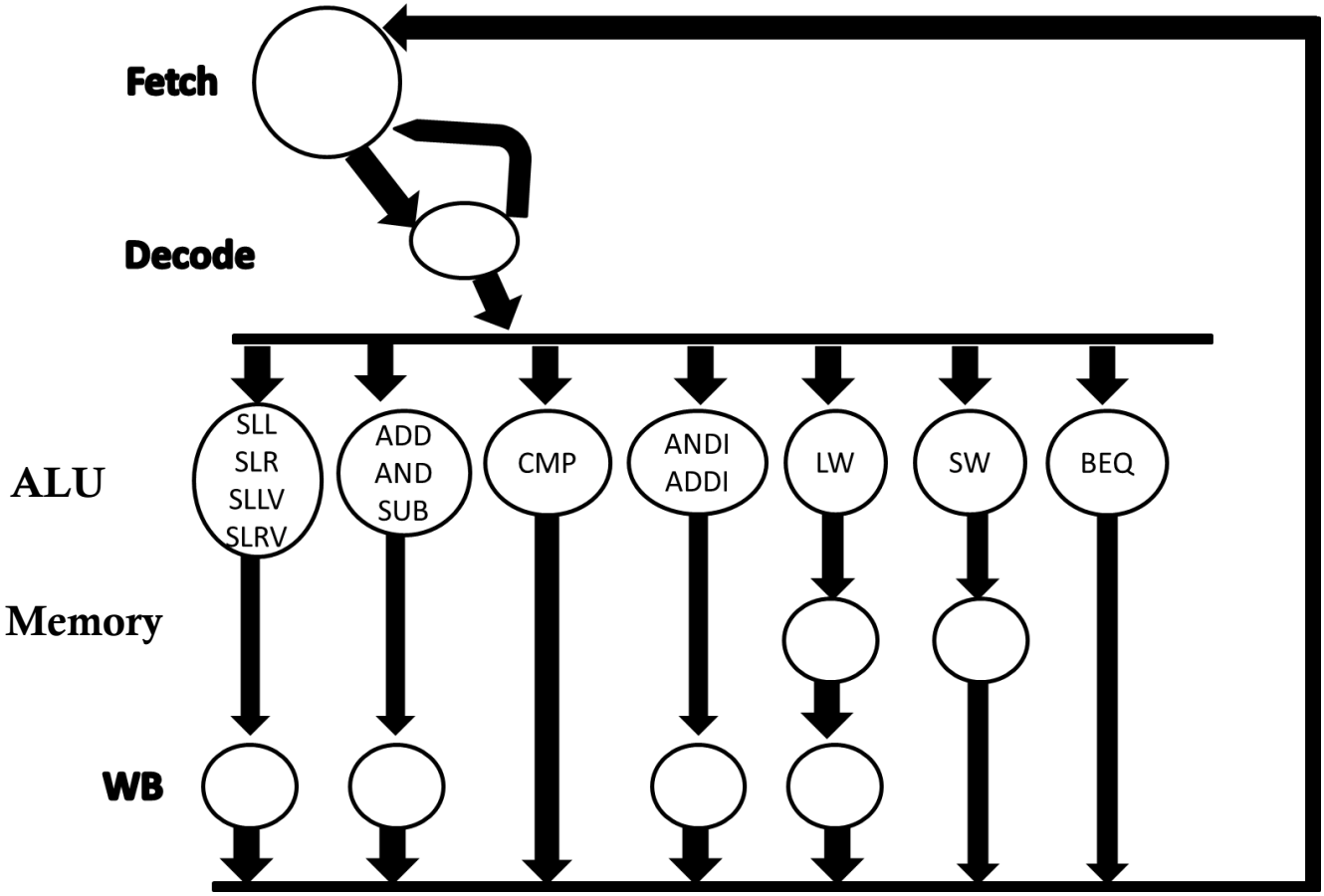


Figure 15: Finite State Machine

The table of control unit shows the corresponding value to control signal and instruction implemented. While the figure above shows the transition per instruction, for example ‘and, add, sub’ instructions being fetch decoded executed and write back to register file then return to fetch state for the next instruction, while ‘j’ is being fetched decoded and then return to fetch again.

2. Final Data path

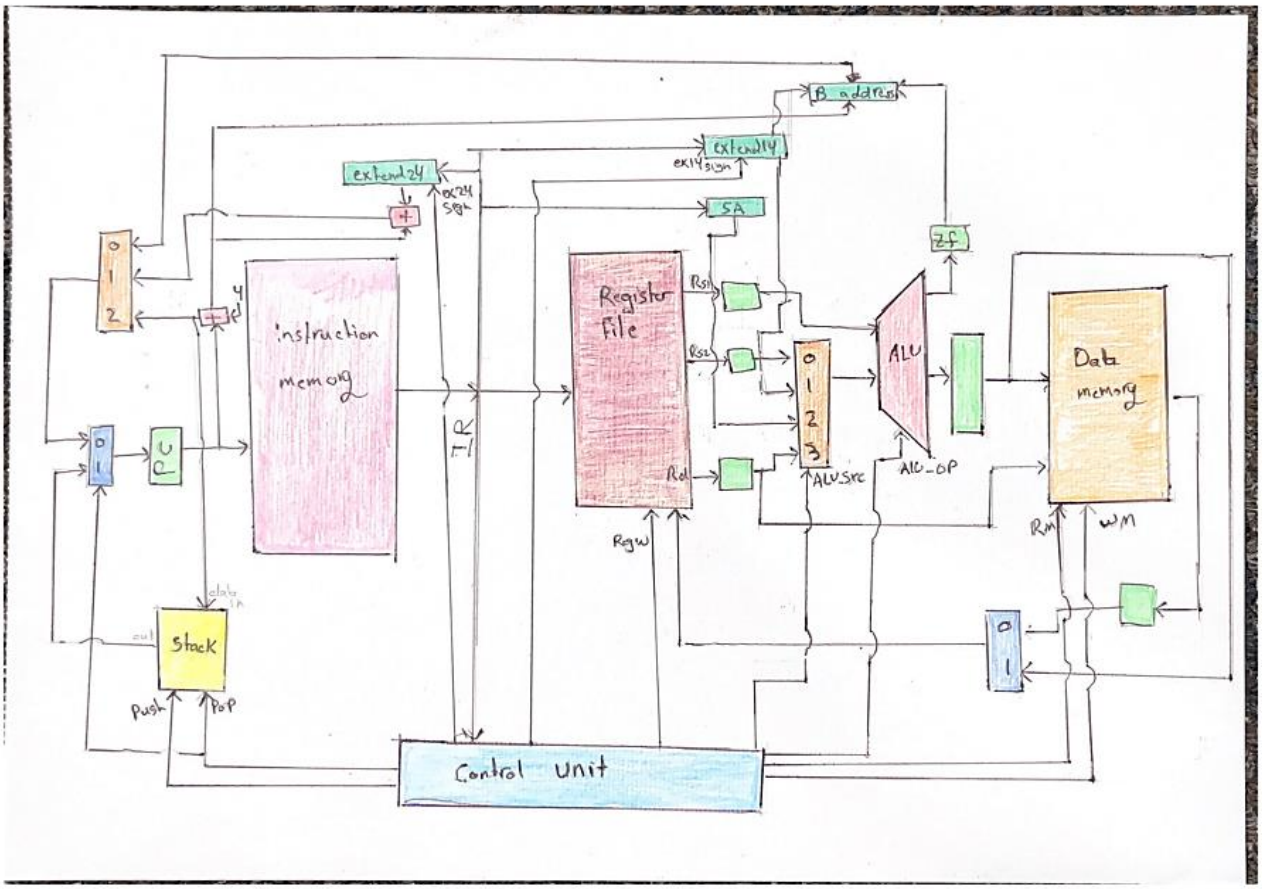
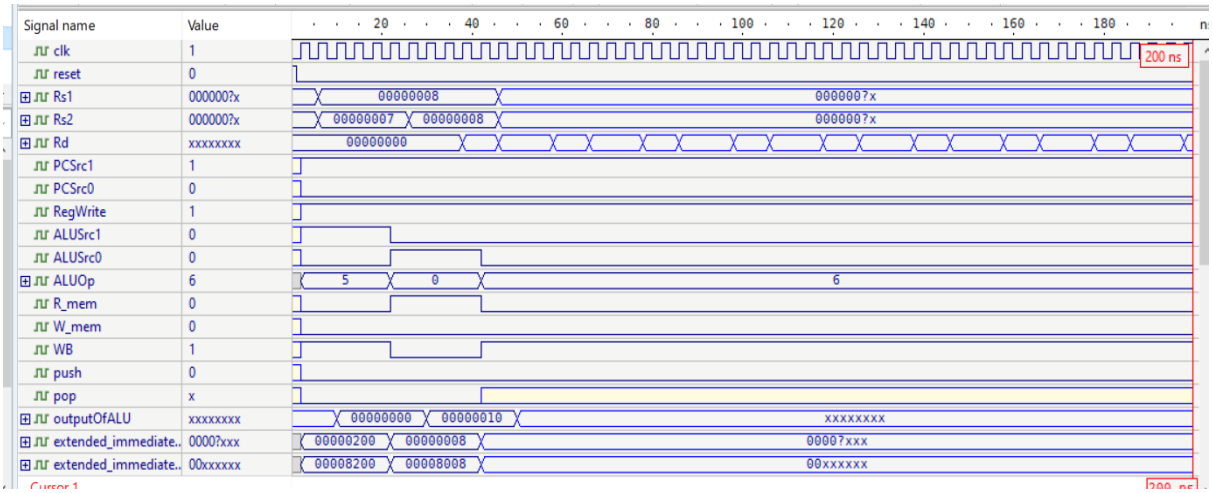
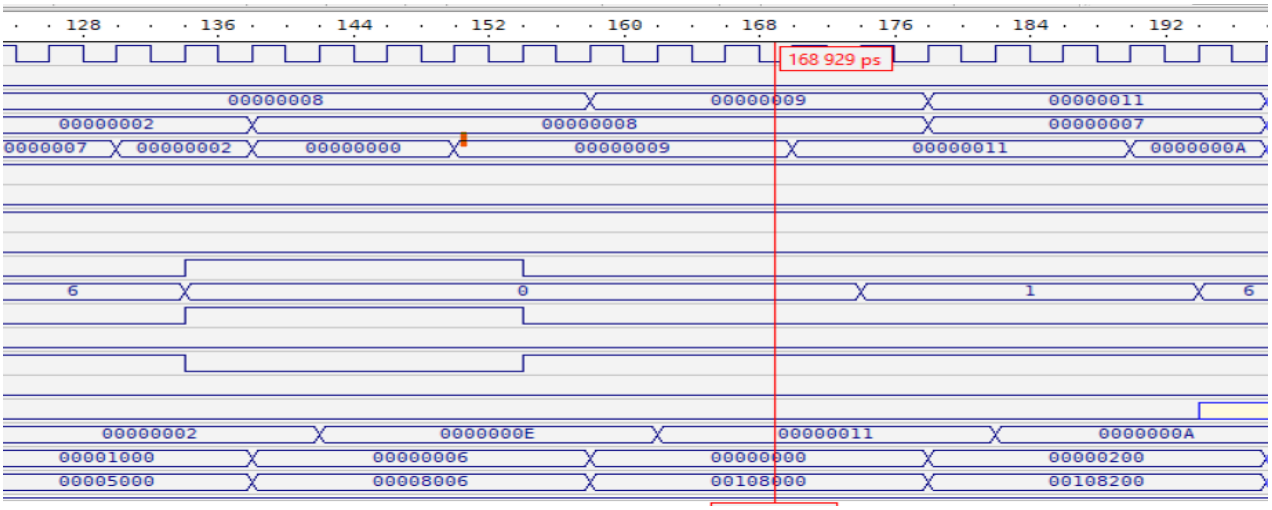


Figure 16: Final Data path

2.1. Final Data Path Testing

```
116
117     memory[0] = 32'h10040034; //LW Rd = 9
118     memory[1] = 32'h08840000; //ADD Rd = Rd + 8 (Rd = 17)
119     memory[2] = 32'h10841000;  // SUB Rd = Rd - 7 (Rd = 10)
120     memory[3] = 32'h18040004;  //SW Rd
121     memory[4] = 32'h00000022; //J to the PC = 8
122
123
124     memory[5] = 32'h10040034; //LW
125     memory[6] = 32'h08840000; //ADD
126     memory[7] = 32'h08840000; //ADD
127
128     memory[8] = 32'h08000062;   //Jal to PC = 20
129     memory[9] = 32'h18028006;   //Rd = SLRV with Shift amount
130
131     memory[10] = 32'h10040034; //LW Rd = 9
132     memory[11] = 32'h08840000; //ADD Rd = Rd + 8    (Rd = 17)
133     memory[12] = 32'h10841000;  // SUB Rd = Rd - 7 (Rd = 10)
134
135     memory[20] = 32'h00840287;  //Rd = SLL with Shift amount  = 5
136
```





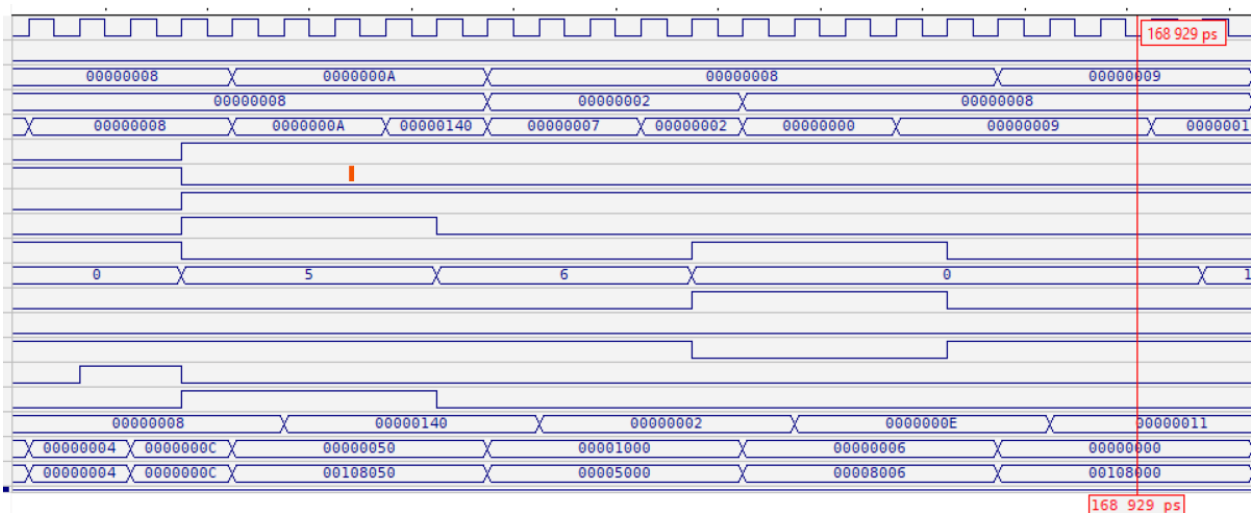


Figure 17: Final Simulation

3. Verification The Component of Data path

3.1. Control unit

3.1.1. R-Type Instructions

3.1.1.1. AND Instruction

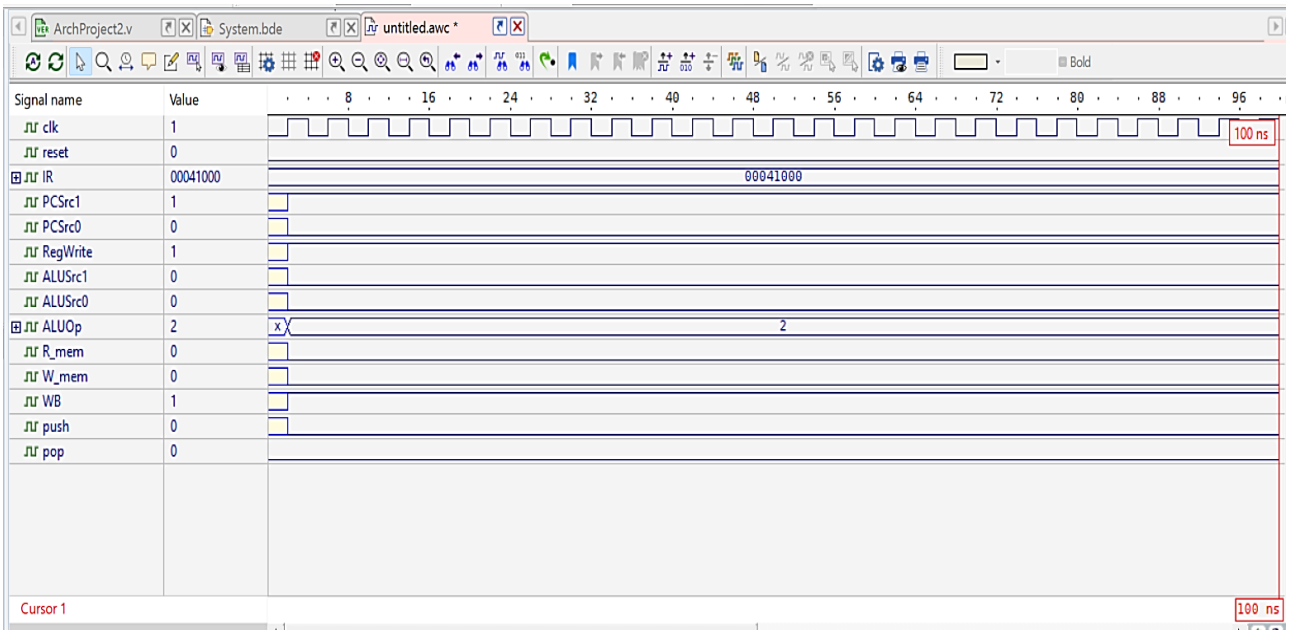


Figure 18:AND Instruction Simulation

3.1.1.2. CMP Instruction

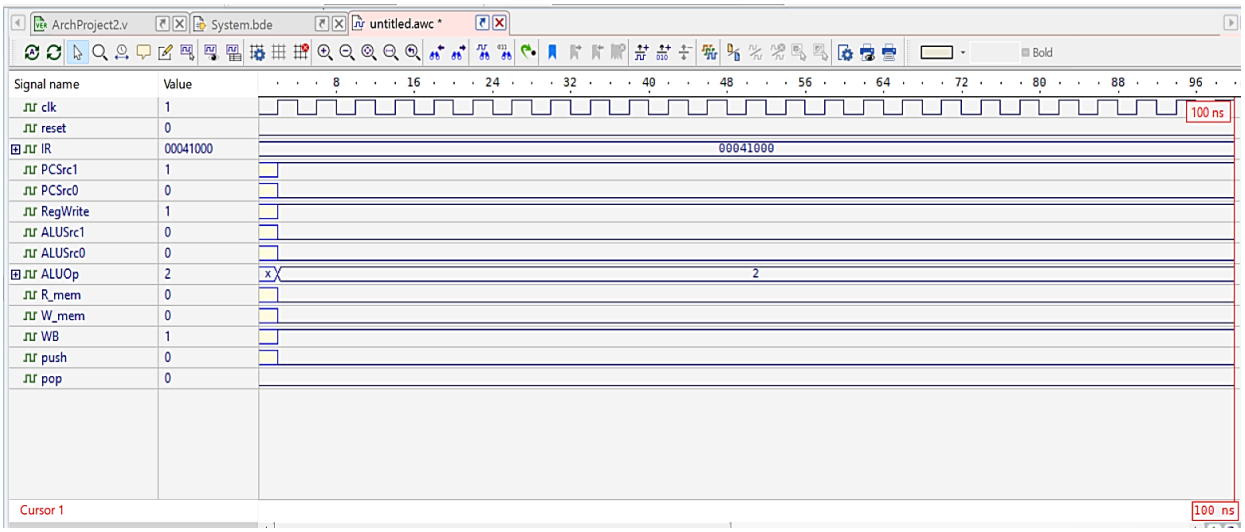


Figure 19:CMP Instruction Simulation

3.1.2. I-Type Instructions

3.1.2.1. LW Instruction

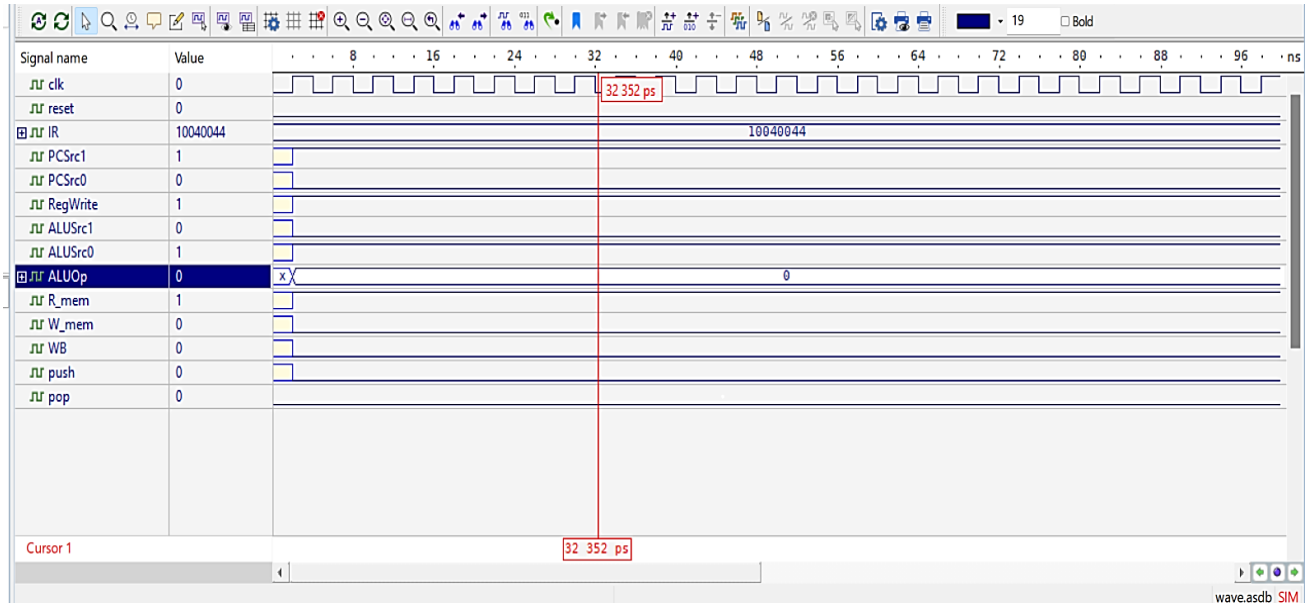


Figure 20:LW Instruction Simulation

3.1.2.2. BEQ Instruction

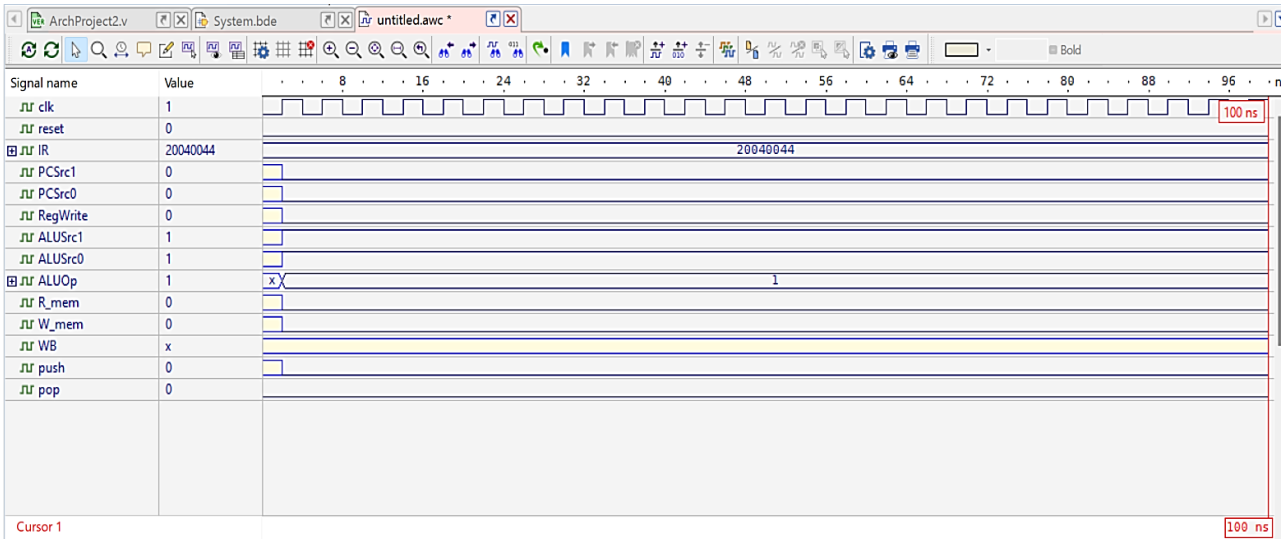


Figure 21:BEQ Instruction *Simulation*

3.1.3. J-Type Instructions

3.1.3.1. J Instruction

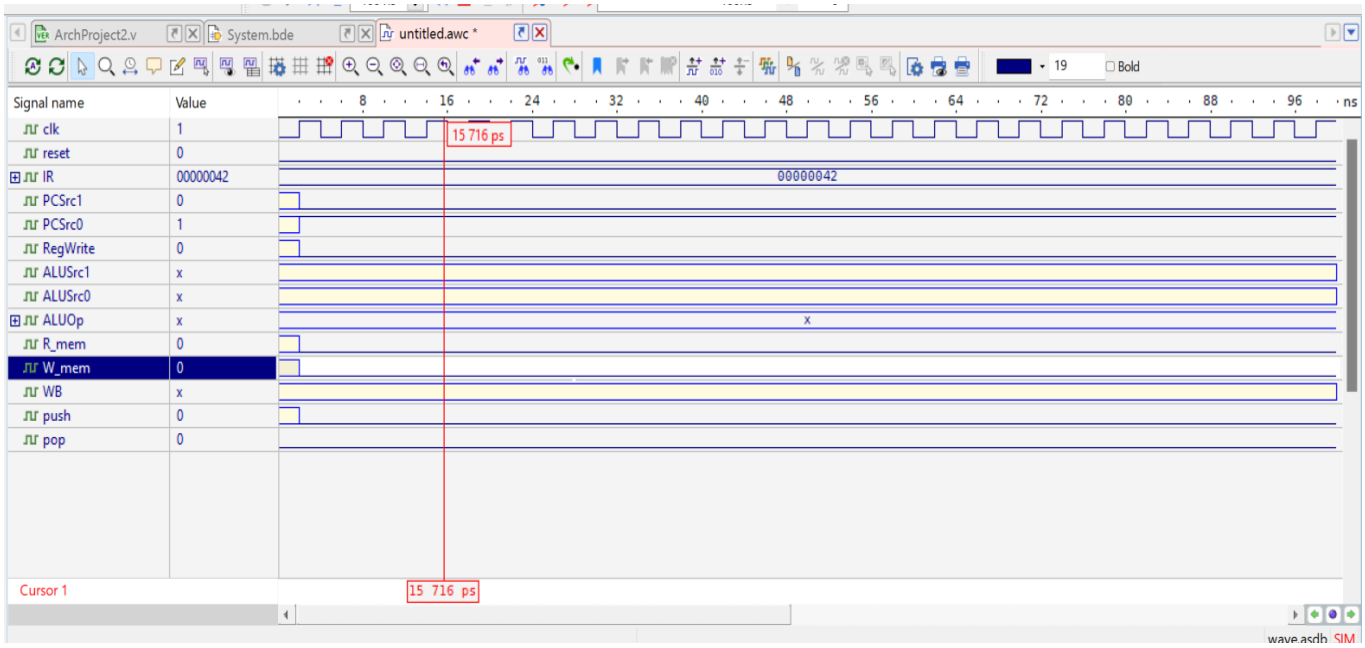


Figure 22: J Instruction Simulation

3.1.3.2. JAL Instruction

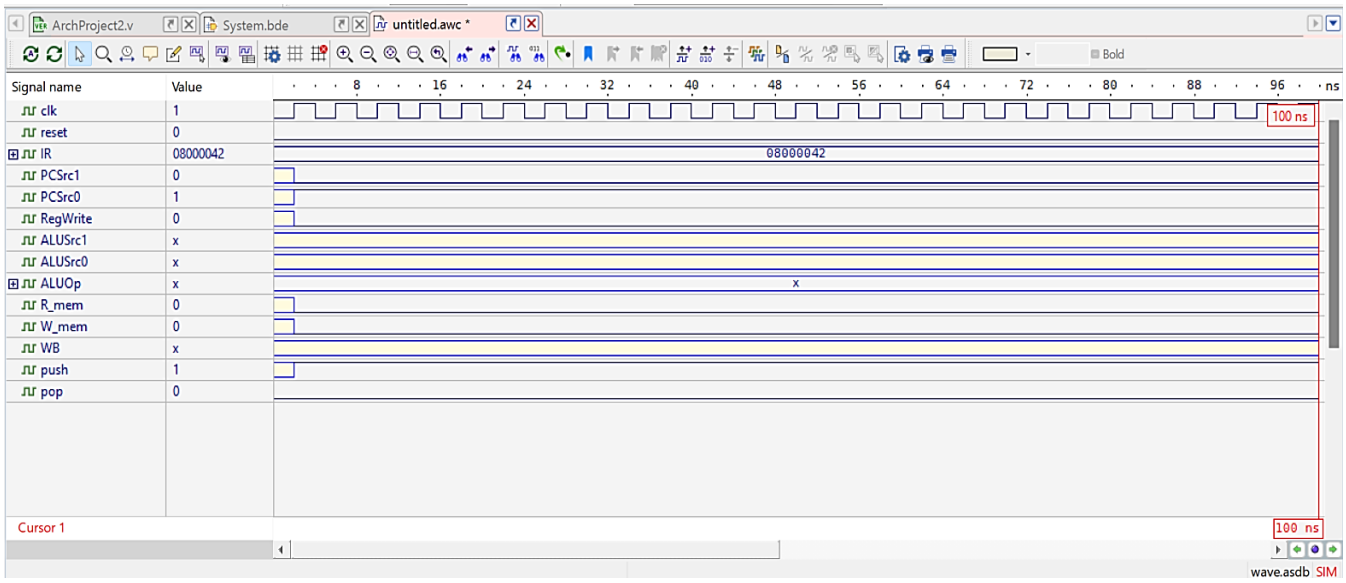


Figure 23:JAL Instruction Simulation

3.1.4. S-Type Instructions

3.1.4.1. SLL Instruction

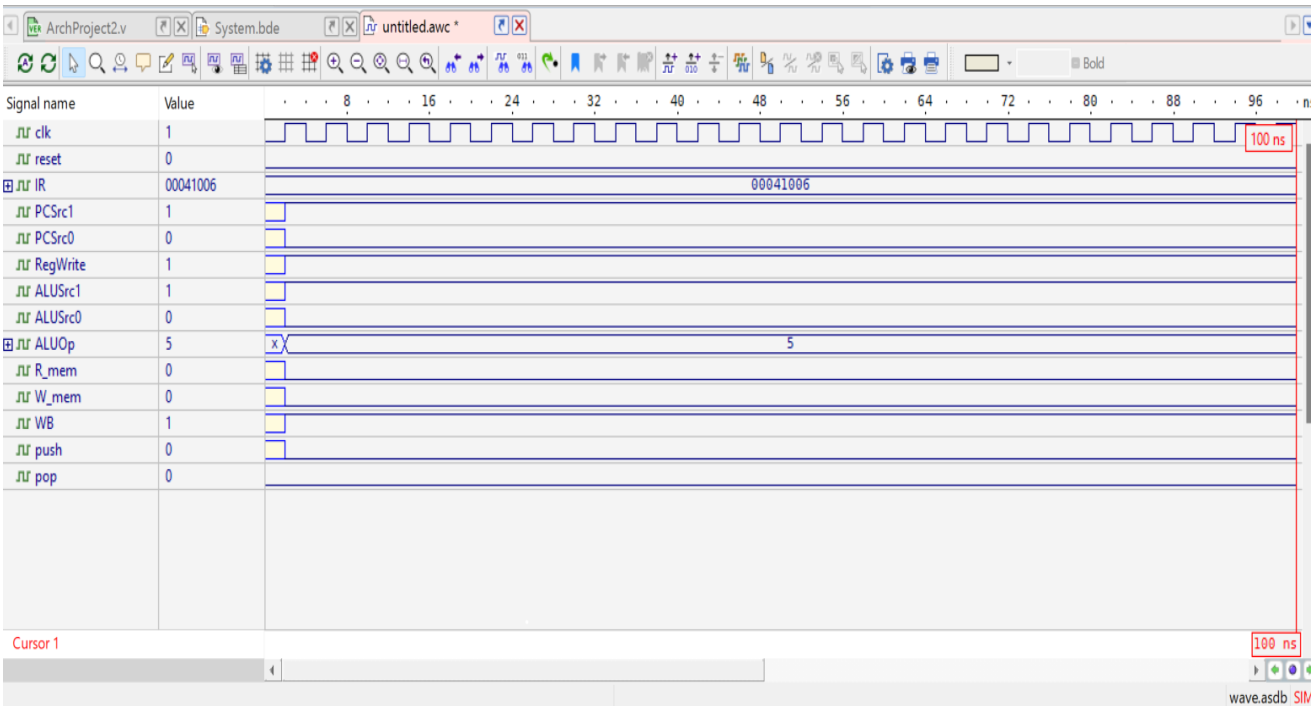


Figure 24:SLL Instruction Simulation

3.1.4.2. SLLV Instruction

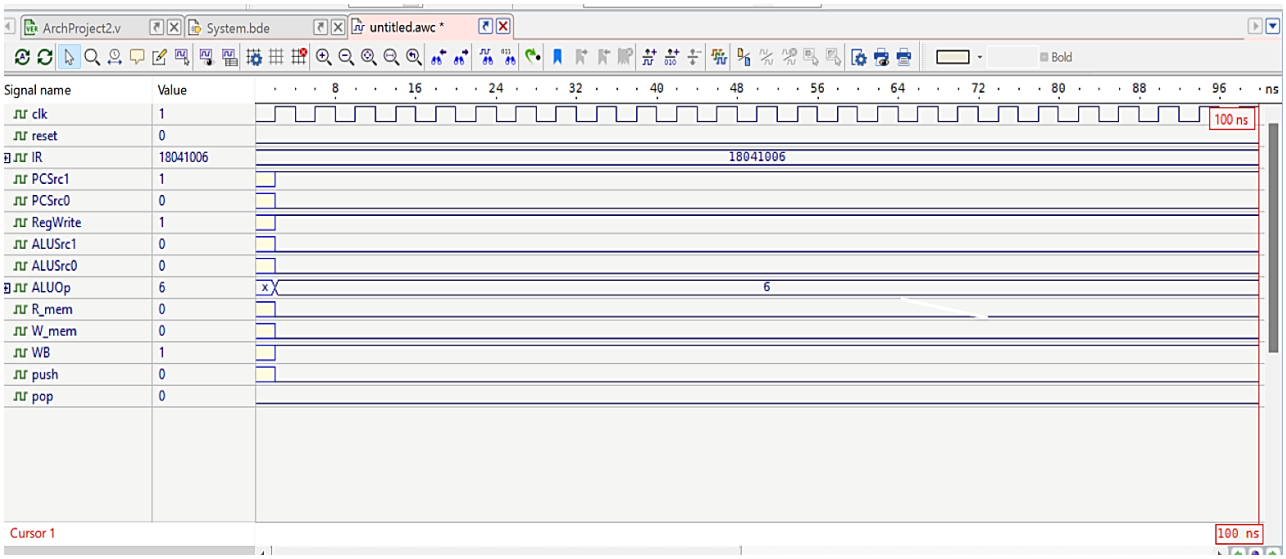


Figure 25: SSLV Instruction *Simulation*

3.2. Instruction Memory

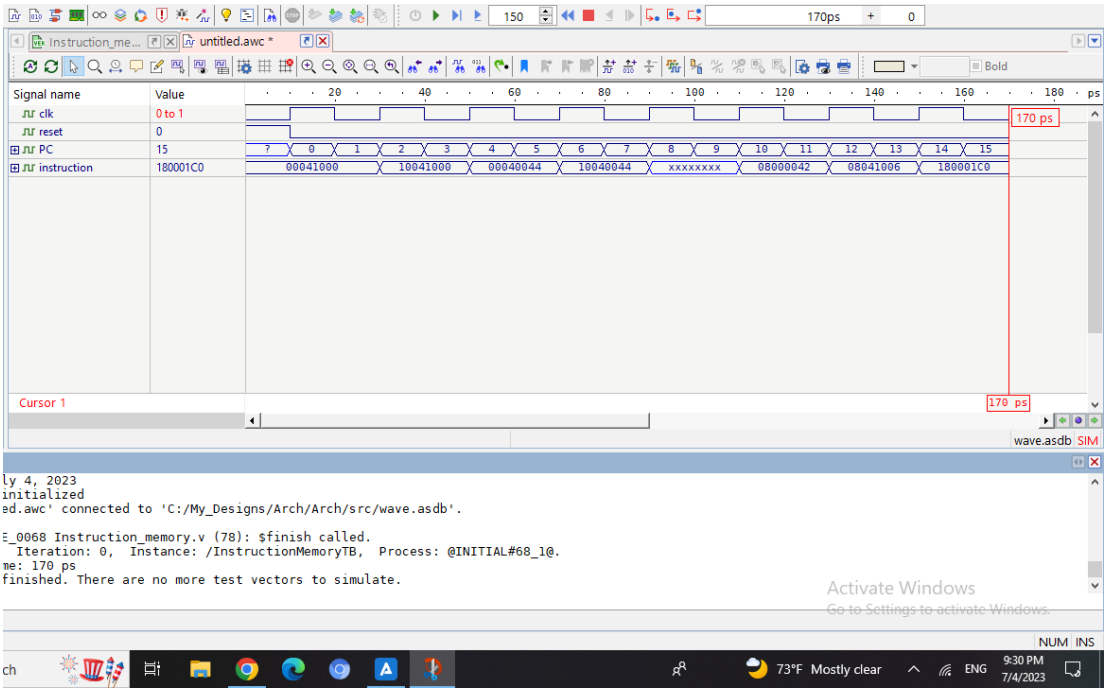


Figure 26: Instruction Memory Simulation

3.3. Register File

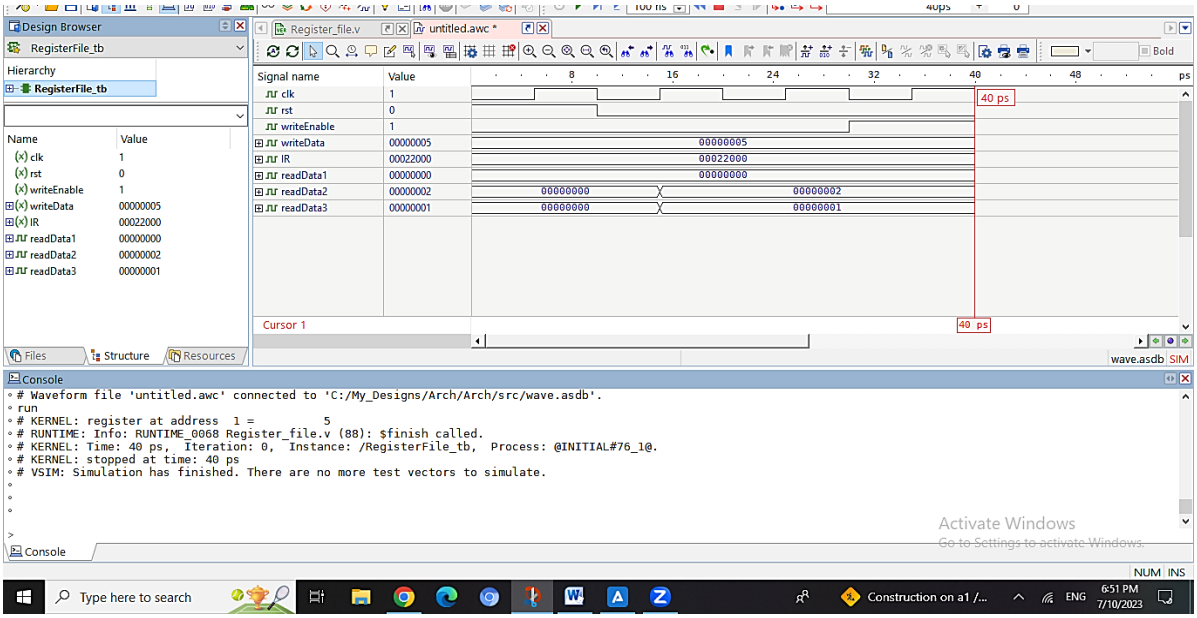


Figure 27: Register File Simulation

3.4. ALU

3.4.1. ALU Addition

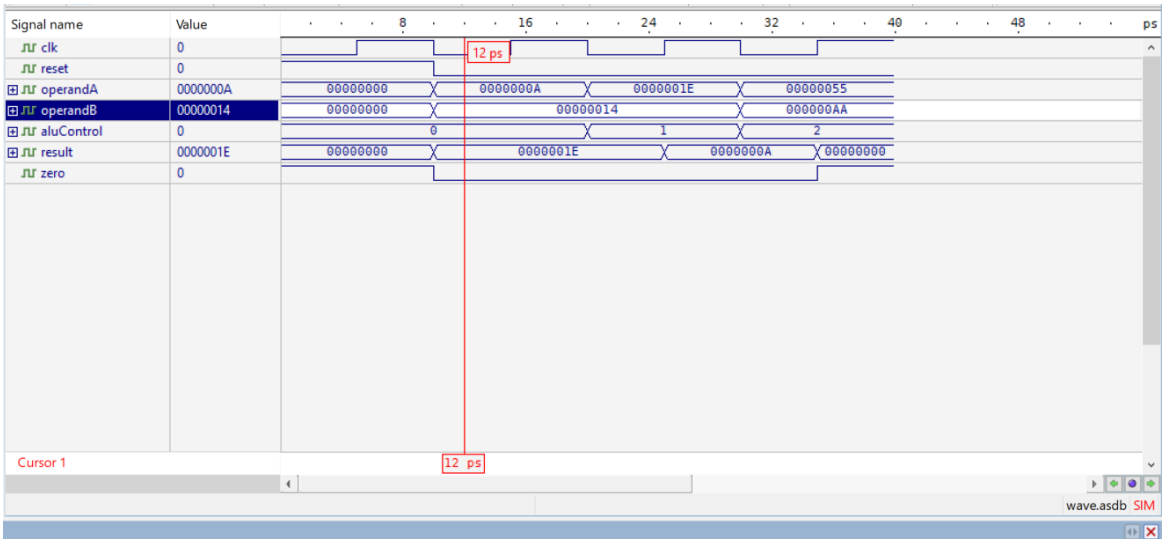


Figure 28: ALU Addition Simulation

3.4.2. ALU Subtraction

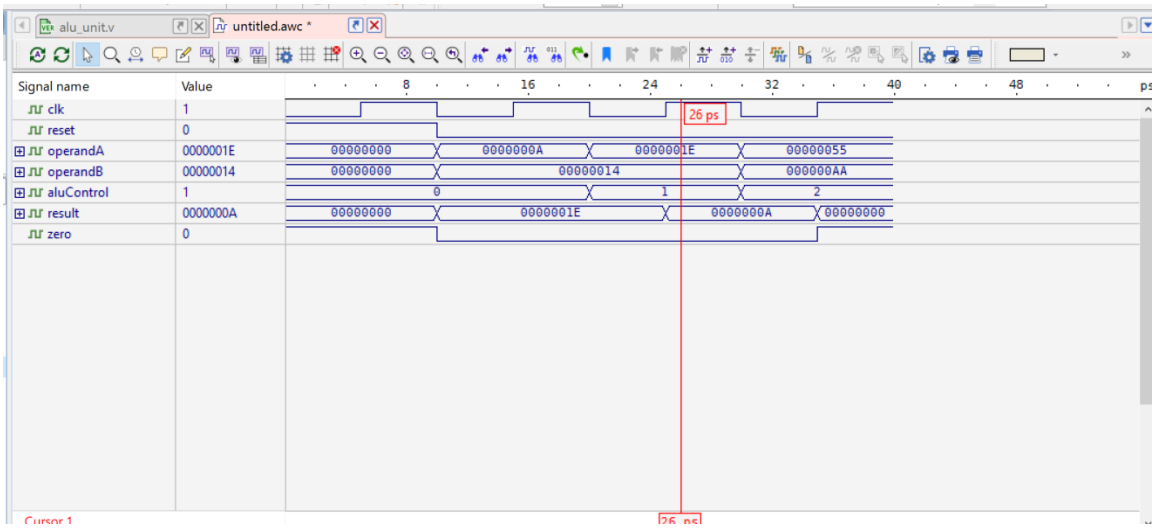


Figure 29: ALU Subtraction Simulation

3.4.3. ALU AND

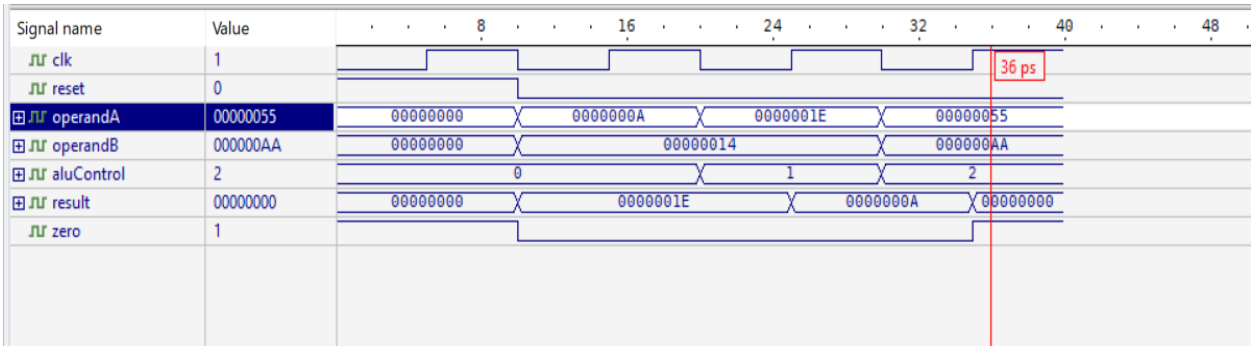


Figure 30: ALU AND Simulation

3.4.4. ALU OR

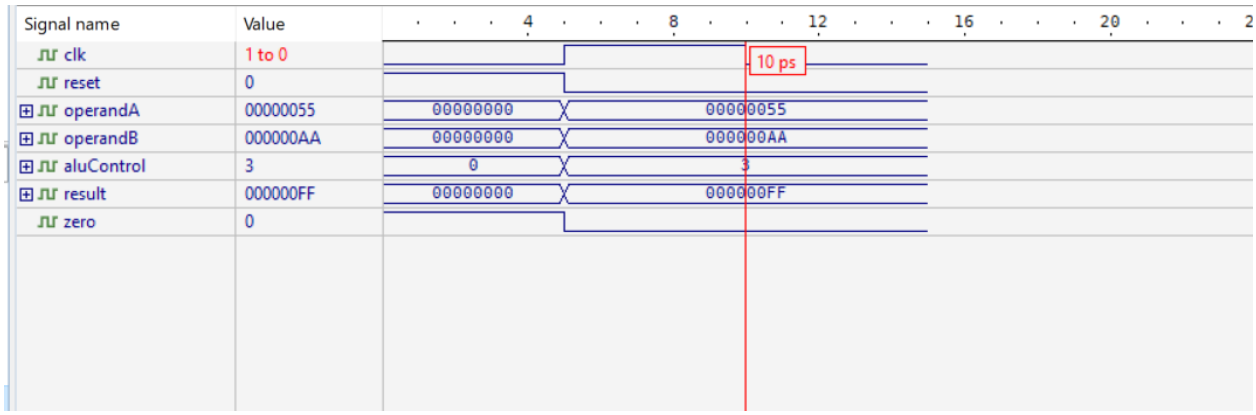


Figure 31:ALU OR

3.4.5. ALU XOR

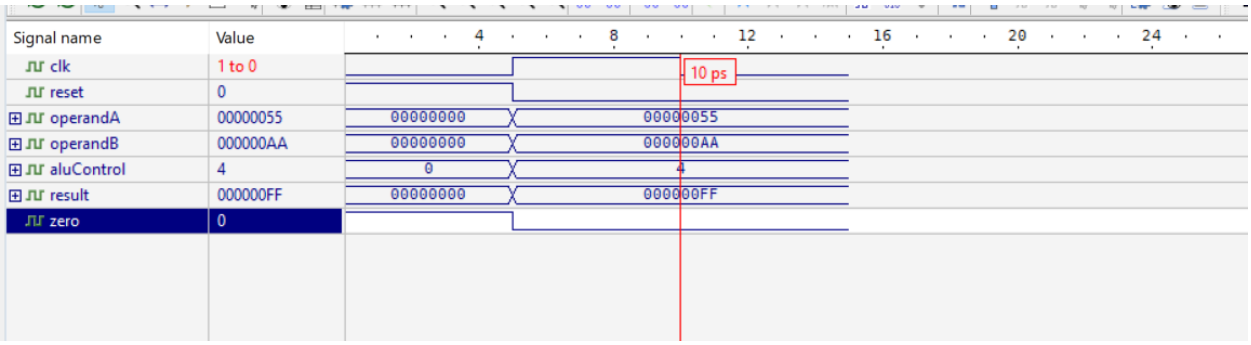


Figure 32:ALU XOR

3.4.6. SLL

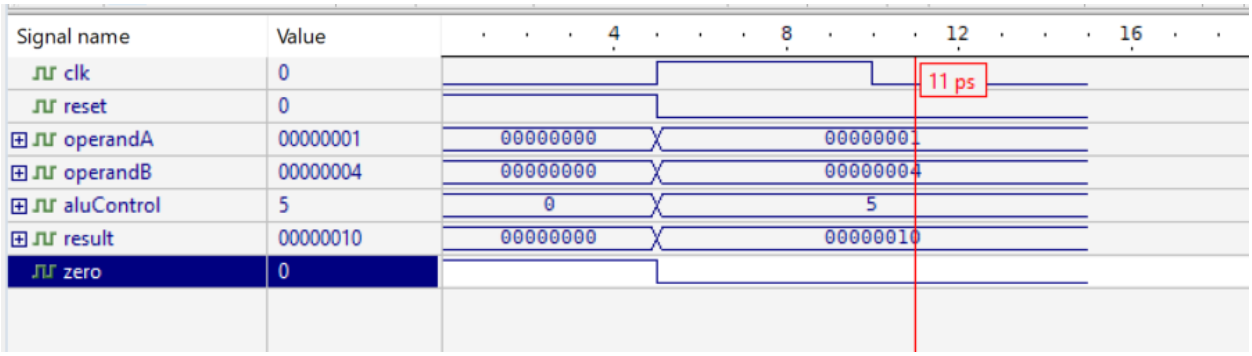


Figure 33:SSL Simulation

3.4.7. SLR

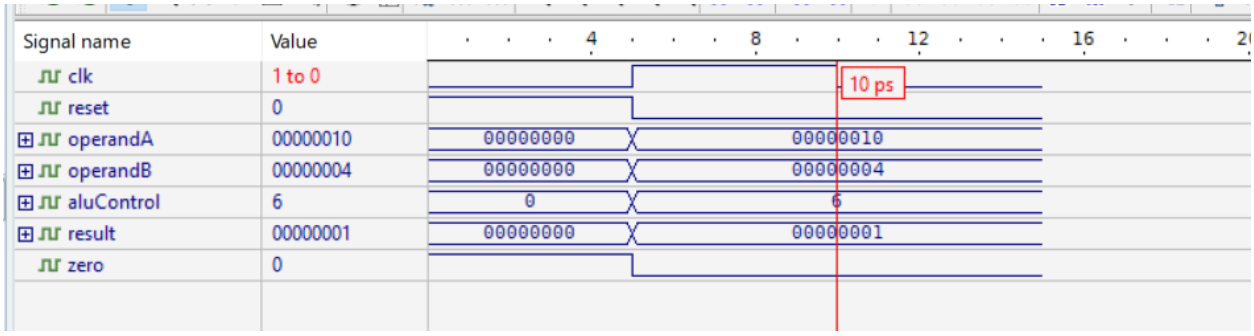


Figure 34: SLR Simulation

3.5. Data Memory

3.5.1. Write on memory

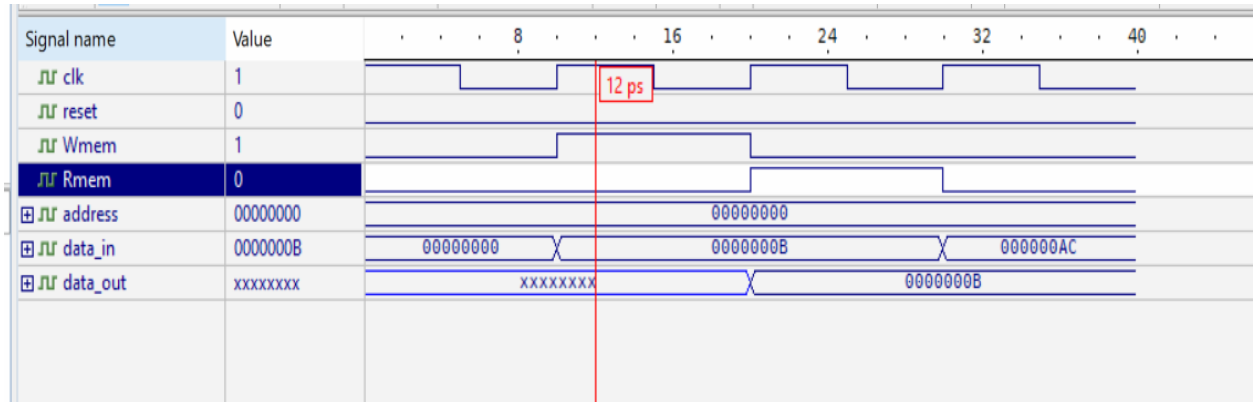


Figure 35:Write on Memory Simulation

3.5.2. Read from memory

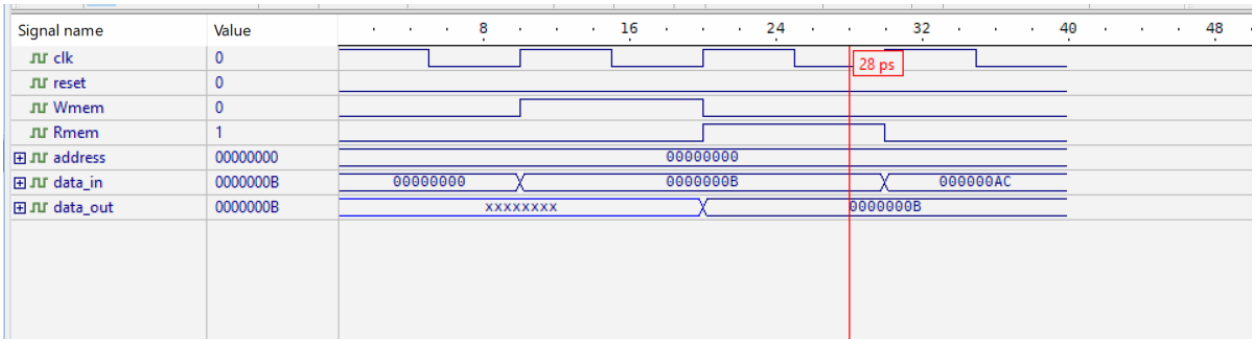


Figure 36:Read From Memory Simulation

4. Conclusion

The project involved demonstrating the complete construction of the data path based on specified instructions. Each stage was developed with the necessary control signals to regulate their operations. A simulation was performed for every step, along with the design work for creating the required data path and the final state machine, all of which are detailed in the report.

5. Teamwork

As a team, we cooperated together to make this project, first we start with thinking of the design for each component, by sharing ideas and correct each other, then we agreed on the design and put the control unit signals table. After that, one of us built and test PC, IF/ID and register file, one built and test Decode Condition, sub-Condition, add Condition and ALU control unit and the last one built ALU unit, Control unit and the extenders, finally after all units were ready and worked in the right way, we all put our effort, and built the data path together then testing it to make sure that it works as required

6. Appendices

6.1. The Code for the data Path

```
// Define global variables
reg [2:0]current_state;
reg [2:0]next_state;
reg [31:0]IR;


module AllTheSystem (

        input clk,
        input reset,

        // Observing the values of the generated signals
        output reg PCSrc1,
        output reg PCSrc0,
        output reg RegWrite,
        output reg ALUSrc1,
        output reg ALUSrc0,
        output reg [2:0] ALUOp,
        output reg  R_mem,
        output reg W_mem,
        output reg WB,
        output reg push,
        output reg pop,


        ///Instruction Decode Part
        output reg [31:0] Rs1, // (Bus A )Output data from register 1
    output reg [31:0] Rs2, // (Bus B) Output data from register 2
    output reg [31:0] Rd, // (Bus W) Output data from register 3
```

```
        ///ALU Part
        output reg [31:0] outputOfALU, // Just to verify that the ALU works properly

        ///immediate14 Part
        output reg [31:0]extended_immediate14,
        ///immediate24 Part
        output reg [31:0]extended_immediate24,


        ///just for depugging the code for store
        output reg [31:0]outputOfTheStore,

        ///Shift amount constant
        output reg [4:0]shiftAmount

);

        /// Instruction Fetch
        reg [31:0] memory [0:31];
        reg [31:0] PC;

        ///Instruction Decode
        reg [31:0] writeData;
        reg [31:0]D0;
        reg [31:0]D1;
    reg [31:0]D2;
        reg [31:0]D3;
    reg [31:0]Y;
        reg [4:0] Registers[0:31];


        reg [4:0] AddressOfRd;  // Address for Rd
        reg [4:0]AddressOfRs1;  // Address for RS1
        reg [4:0]AddressOfRs2;  // Address for RS2
```

```
        ///Control Unit Signals Generated
        reg [1:0]Type;
        reg [4:0]Function;

        ///ALU Part
         reg [31:0] operandA;    // Operand A
 reg [31:0] operandB;    // Operand B
 reg zero;               // Zero flag indicating if the result is zero


        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ///Data Memory
        reg [31:0] address,data_in;
reg [31:0] data_out;

        ///Data Memory
        reg [31:0] data_memory [31:0];

        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        ///Write back stage
reg [31:0]outOfDownMux;

        //Extenders 14 and 24
        reg [13:0] immediate14;
        reg [23:0] immediate24;

        //Up adder
        reg [31:0] outputOfUpAdder;

        ///Stack Part
        reg [31:0] outputOfStack;
```

```
        reg empty,full;
        parameter STACK_DEPTH = 8;    // This is the maximum # of inner functions for this
stack
    reg [31:0] stack [STACK_DEPTH-1:0];
    reg [2:0] top;  //Stack Pointer

        //Branch Target Address
        reg [31:0] outputOfBTA;

        ///outputOfMuxAfterBTA
        reg [31:0]outputOfMuxAfterBTA;

        ///outputOfMuxAfterStack
        reg [31:0]outputOfMuxAfterStack;

        initial begin



        memory[0] = 32'h10040034; //LW Rd = 9
        memory[1] = 32'h08840000; //ADD Rd = Rd + 8          (Rd = 17)
        memory[2] = 32'h10841000;  // SUB Rd = Rd - 7 (Rd = 10)
        memory[3] = 32'h18040004;  //SW Rd
        memory[4] = 32'h00000022; //J to the PC = 8



        memory[5] = 32'h10040034; //LW
        memory[6] = 32'h08840000; //ADD
        memory[7] = 32'h08840000; //ADD

        memory[8] =     32'h08000062;   //Jal to PC = 20
        memory[9] = 32'h18028006;      //Rd = SLRV with Shift amount

        memory[10] = 32'h10040034; //LW Rd = 9
        memory[11] = 32'h08840000; //ADD Rd = Rd + 8          (Rd = 17)
        memory[12] = 32'h10841000;  // SUB Rd = Rd - 7 (Rd = 10)
```

```
memory[20] = 32'h00840287;    //Rd = SLL with Shift amount  = 5

end

initial top = 3'b000;


integer i;
always @(posedge clk)
        case (current_state)
                0:
                        next_state = 1;
                1:
                        next_state = 2;
                2:
                        next_state = 3;
                3:
                        next_state = 4;
                4:
                        next_state = 0;
        endcase


always @(posedge clk)
        current_state = next_state;


always @(posedge clk, posedge reset)

        if (reset)          begin

                current_state = 4;
                PC = 32'h00000000;
```

```
                Rs1 = 32'd0;
                Rs2 = 32'd0;
                Rd = 32'd0;
                Y = 32'h00000000;



                for (i = 0; i < 31; i = i + 1)
                 Registers[i] <= 32'h00000000;
            end

    else if (current_state == 0) begin

            //instruction fetch
            IR = memory[PC];

            //$display ("%x", IR);

            //grep the type and the function to determine the kind of the inctruction
            Type = IR[2:1];
        Function = IR[31:27];



            //$display ("%b", Type);

            ///Grep the constant for the Shift process
            shiftAmount = IR [11:7];

            //pop signal
            pop = IR[0];



            /// R-type
            if (Type == 2'b00)
                    if (Function == 5'b00000 || Function == 5'b00001 || Function ==
5'b00010) begin ///ADD|| AND || SUB
```

```
                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
        RegWrite = 1'b1;
ALUSrc1 = 1'b0;
                ALUSrc0 = 1'b0;

                if (Function == 5'b00000)
                        ALUOp = 3'b010;
                else if (Function == 5'b00001)
                        ALUOp = 3'b000;
                else
                        ALUOp = 3'b001;

R_mem = 1'b0;
W_mem = 1'b0;
WB = 1'b1;
                push = 1'b0;
    end

    else begin

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
        RegWrite = 1'b0;
ALUSrc1 = 1'b0;
                ALUSrc0 = 1'b0;
                ALUOp = 3'b001;
R_mem = 1'b0;
W_mem = 1'b0;
                push = 1'b0;
    end

/// I-Type
```

```
else if (Type == 2'b10) begin

    if (Function == 5'b00000 || Function == 5'b00001) begin    ///ANDI    ||
ADDI

        PCSrc1 = 1'b1;
        PCSrc0 = 1'b0;
    RegWrite = 1'b1;
ALUSrc1 = 1'b0;
        ALUSrc0 = 1'b1;

        if (Function == 5'b00000)
            ALUOp = 3'b010;
        else
            ALUOp = 3'b000;


R_mem = 1'b0;
W_mem = 1'b0;
WB = 1'b1;
        push = 1'b0;
    end

    else if (Function == 5'b00010) begin //LW

        PCSrc1 = 1'b1;
        PCSrc0 = 1'b0;
    RegWrite = 1'b1;
ALUSrc1 = 1'b0;
        ALUSrc0 = 1'b1;
        ALUOp = 3'b000;
R_mem = 1'b1;
W_mem = 1'b0;
        push = 1'b0;
        WB = 1'b0;
```

```
                push = 1'b0;
            end

            else if (Function == 5'b00011) begin //SW

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
              RegWrite = 1'b0;
        ALUSrc1 = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 3'b000;
        R_mem = 1'b0;
        W_mem = 1'b1;
                push = 1'b0;
                push = 1'b0;
            end

            else if (Function == 5'b00100) begin //BEQ

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
              RegWrite = 1'b0;
        ALUSrc1 = 1'b1;
                ALUSrc0 = 1'b1;
                ALUOp = 3'b001;
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;
            end
    end


    /// J-Type
    else if (Type == 2'b01) begin
```

```
            if (Function == 5'b00000) begin  // J

                    PCSrc1 = 1'b0;
                    PCSrc0 = 1'b1;
                RegWrite = 1'b0;
R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;
            end

            else begin          // JAL

                    PCSrc1 = 1'b0;
                    PCSrc0 = 1'b1;
                RegWrite = 1'b0;
        R_mem = 1'b0;
        W_mem = 1'b0;
                    push = 1'b1;
            end

end

else begin/// S-Type

            if (Function == 5'b00000 || Function == 5'b00001) begin    ///SLL || SLR

                    PCSrc1 = 1'b1;
                    PCSrc0 = 1'b0;
                RegWrite = 1'b1;
        ALUSrc1 = 1'b1;
                    ALUSrc0 = 1'b0;

                    if (Function == 5'b00000)
```

```
                    ALUOp = 3'b101;
                else
                    ALUOp = 3'b110;

            R_mem = 1'b0;
            W_mem = 1'b0;
            WB = 1'b1;
                push = 1'b0;
            end

            else begin

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
                RegWrite = 1'b1;
            ALUSrc1 = 1'b0;
                ALUSrc0 = 1'b0;

                if (Function == 5'b00010)
                    ALUOp = 3'b101;
                else
                    ALUOp = 3'b110;

            R_mem = 1'b0;
            W_mem = 1'b0;
            WB = 1'b1;
                push = 1'b0;
            end
        end
    end



    else if (current_state == 1) begin
```

```
///Getting the values of the extenders
immediate14 = IR[16:3];
extended_immediate14 = {18'b000000000000000000, immediate14}; ///Temp, we want to change these 0's later on

//$display ("extended_immediate14 = %b", extended_immediate14);

immediate24 = IR[26:3];
extended_immediate24 = {8'b00000000, immediate24}; ///Temp, we want to change these 0's later on

Registers[0] = 32'h00000008;                    //0001 1000 0000 0010 1000 0000 0000 0110
Registers[1] = 32'h00000007;
Registers[8] = 32'h00000002;

///Getting the addresses for the registers
AddressOfRs1 = IR[26:22]; // (Rs1) Register 1 address for read operation
AddressOfRs2 = IR[16:12]; // (Rs2) Register 2 address for read operation
AddressOfRd = IR[21:17]; // (Rd) Register address for write operation

$display (PC);
//$display ("%b", IR);
//$display ("\n");

Rs1 = Registers[AddressOfRs1];
Rs2 = Registers[AddressOfRs2];
Rd = Registers[AddressOfRd];
```

```
        ///Choose a value for the mux

case ({ALUSrc1, ALUSrc0})
  2'b00: Y = Rs2;
  2'b01: Y = extended_immediate14; //Output of the Extender 14
  2'b10: Y = shiftAmount; //Output of the shift amount
  2'b11: Y = Rd;
endcase

        //$display(Y);


        /// Check if the Current instruction is Jump (J)
        if (Type == 2'b01 && Function == 5'b00000) begin

                current_state = 4;

                ///Modify the PC to the Jump address
                outputOfUpAdder = PC + extended_immediate24;
                PC =  outputOfUpAdder;

        end


        /// Check if the Current instruction is Jal
        if (Type == 2'b01 && Function == 5'b00001) begin

                current_state = 4;

           empty = (top == 0);
           full = (top == STACK_DEPTH);


                 // Push operation
```

```
                if (push && !pop && !full) begin
                   stack[top] = PC;
                                top = top + 1;
                end

                    ///Modify the PC
                    PC = PC +  extended_immediate24;
  end

 // $display (stack[0]);

end


else if (current_state == 2)  begin

          operandA = Rs1;
          operandB = Y; //Output of the previous mux4x1

          //$display ("extended_immediate14 = %b", extended_immediate14);

case(ALUOp)
   3'b000: outputOfALU = operandA + operandB;     // Addition
   3'b001: outputOfALU = operandA - operandB;     // Subtraction
   3'b010: outputOfALU = operandA & operandB;     // Bitwise AND
   3'b011: outputOfALU = operandA | operandB;     // Bitwise OR
   3'b100: outputOfALU = operandA ^ operandB;     // Bitwise XOR
   3'b101: outputOfALU = operandA << operandB;    // Shift left (logical)
   3'b110: outputOfALU = operandA >> operandB;    // Shift right (logical)
   3'b111: outputOfALU = operandA >>> operandB;   // Shift right (arithmetic)
   default: outputOfALU = 32'b0;                  // Default case: result is 0
 endcase
```

```
 ///We want later on to check if the last instruction is BEQ or CMP~~~~~~~~~~~~~~~~~
//--------------------------------------------------



zero = (outputOfALU == 32'b0);  // Set zero flag if the result is zero
//$display (zero);

///Check the current instruction is CMP or not
if (Type == 2'b00 && Function == 5'b00011)  begin
        PC = PC + 1;
        current_state = 4;
end

///We want here to check the BEQ
if (Type == 2'b10 && Function == 5'b00100)  begin
        if (zero == 1) begin
                //Modify the BTA

                current_state = 4;

                 ///Output of the Branch Target Address
                 outputOfBTA = PC + extended_immediate14;

                 ///Modify the PC to the Branch Target Address
                 PC =  outputOfBTA;

        end

        else begin
                ///The two numbers are not equal
                PC = PC + 1;
```

```
                current_state = 4;
            end

        end

end


        else if (current_state == 3) begin


            address = outputOfALU; /// Address of the memory
            data_in = Rd; // Store Instruction

            if(W_mem && !R_mem) begin

                /// Store Instruction
                data_memory[address] = data_in;
                outputOfTheStore = data_memory[address];
                //$display ("data_memory[%d] = %d", address, outputOfTheStore);

            end

            else if(!W_mem && R_mem) begin

                /// Load Instruction
                data_memory[14] = 32'h00000009;
                data_out = data_memory[address];

            end

            ///Check the current instruction is Store or not
            if (Type == 2'b10 && Function == 5'b00011) begin
```

```
                    current_state = 4;

                    ///check if the current instruction is the last instruction in the function

                        ///Here is the code to choose the correct value for the PC


                ///Stack Part
                 empty = (top == 0);
                 full = (top == STACK_DEPTH);

                 // Pop operation
                 if (pop && !push && !empty) begin

              top = top - 1;  // Move the top pointer down
                        outputOfStack = stack[top];

                        PC = outputOfStack;
                    PC = PC + 1;
            end

                 else PC = PC + 1;

        end
end


      else if (current_state == 4) begin

                 //$display ("Data out = %d", data_out);
                 if (WB == 1)
                        outOfDownMux = outputOfALU;
                 else
                        outOfDownMux = data_out;
```

```
                /// Write back to the destination Register
                if (RegWrite) begin
                        Rd = outOfDownMux;
                        Registers [AddressOfRd] = Rd;
                end

                ///Here is the code to choose the correct value for the PC
                //$display(PC);
                ///Stack Part
                 empty = (top == 0);
                 full = (top == STACK_DEPTH);

                 // Pop operation
                 if (pop && !push && !empty) begin

              top = top - 1;  // Move the top pointer down
                          outputOfStack = stack[top];

                          PC = outputOfStack;
                    PC = PC + 1;
          end

                 else PC = PC + 1;

 end


endmodule
```

6.2. The Test Bench For The System

```
module TestBench ();


        reg clk;

        reg reset;


        ///Instruction Decode Part

        wire [31:0] Rs1; // (Bus A )Output data from register 1

    wire [31:0] Rs2; // (Bus B) Output data from register 2

    wire [31:0] Rd; // Rd


        ///Control Unit Part

        wire PCSrc1;

        wire PCSrc0;

        wire RegWrite;

        wire ALUSrc1;

        wire ALUSrc0;

        wire [2:0]ALUOp;

        wire R_mem;

        wire W_mem;

        wire WB;

        wire push;

        wire pop;

        wire [31:0]outputOfALU;

        wire [31:0]extended_immediate14;

        wire [31:0]extended_immediate24;

        wire [31:0]outputOfTheStore;
```

AllTheSystem WS (clk, reset, PCSrc1, PCSrc0, RegWrite, ALUSrc1, ALUSrc0, ALUOp,R_mem, W_mem, WB, push, pop, Rs1, Rs2, Rd, outputOfALU, extended_immediate14, extended_immediate24, outputOfTheStore);

initial begin current_state = 0; clk = 0; reset = 1; #1ns reset = 0; end

//always @ (posedge clk) $display (current_state);

always #2ns clk = ~clk;

initial #200ns $finish;

endmodule