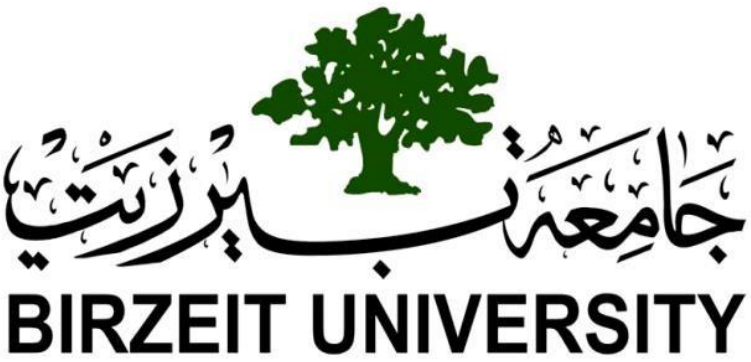

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Advanced Digital – ENCS3310

Project Report – Design Of Multi-Cycle Processor

Prepared by:

Name: Raghad Jamhour. **Number:** 1220212.

Name: Rawaa Hammad. **Number:** 1210927.

Instructor: Ayman Hroub.

Date: 20/08/2024.

Abstract

This project involves the development of a multi-cycle processor in advanced digital design. The used components are 16-bit ALU, Register File with 16 general-purpose registers each 16 bits wide, Data memory and an instruction memory along with a control unit. The processor executes 16-bit machine instructions across five stages: fetch, decode, execute, memory, and write-back. The ALU performs arithmetic and logical operations using a 4-bit opcode, while the Register File provides fast data access. This project offers efficient instruction execution through clock cycles and proper control signal generation. Testing is performed to verify that the processor correctly handles R-type and M-type. It focuses on ensuring proper transitions between different stages and overall processor functionality. The results and simulations, are presented to show how the processor performs.

Table of Contents

Table of figures

List of tables

Brief introduction and background

1- Arithmetic Logic Unit (ALU)

In our multi-cycle processor, the ALU (Arithmetic Logic Unit) is a crucial component that performs arithmetic and logic operations. The ALU processes two 16-bit inputs. Based on a 4-bit opcode sent by the control unit, the ALU can execute various operations including addition, subtraction, bitwise AND, OR, and XOR and the result is the output of the ALU. This design allows for efficient execution of R-Type instructions, ensuring accurate operations within the processor.

2- Register File

The register file is a vital component of the processor since it temporarily stores data during transfers between memory and operational units. The processor accesses this data using two registers, Rs1 and Rs2. Data from Rs1 is accessed using the input address (read_port1 in our design) and the value is assigned to readData1, while data from Rs2 is accessed using the second input address (read_port2 in our design) is assigned to readData2. The register Rd is the destination for writing data, when the control signal writeEn is 1, Rd is accessed using the input address (write_port) then the data on writeData is written into Rd.

3- Data Memory

In our processor, the memory is divided into two separate sections: Instruction Memory and Data Memory. This division helps avoid conflicts that could occur when fetching instructions while simultaneously loading or storing data.

The data memory is connected to the datapath through an address input and two data lines: one as an output for the data and the other is an input for writing on the memory. When the MemRd signal is set to 1, the data memory gets the data stored in the specified address and outputs it. When the MemWrite signal is set to 1, it writes the input data to the specified address.

4- Instruction Memory

The instruction memory is designed to store instructions and is designed only for reading, taking a 16-bit address from the Program Counter (PC) only 8 is used of it in our design since the memory is 512 bytes (256 addresses * 2 Bytes), then providing a corresponding 16-bit instruction.

5- Control Unit

The control unit in our processor is responsible for generating control signals based on the instruction's opcode. It takes the 4-bit opcode and produces appropriate control signals for the operations. The control unit outputs signals for register writing (RegWr), memory reading (MemRd), memory writing (MemWr), and ALU operation (ALUop). The logic is implemented using a combinational always block with a case statement to set these signals according to the opcode received. This ensures that the processor correctly performs arithmetic, logical, load, and store operations by managing the flow of data through the instruction stages.

Design Philosophy

- ## MCP Philosophy

Inputs and Outputs

The module receives a clock signal (clk) and a reset signal (reset). The operation of the microprocessor occurrs at the rising edge of the clik, while the reset signal resets the system to the fetch stage.

On the other hand the module outputs multipile signals:

- current_state and next_state: These indicate the current and upcoming states of the microprocessor.
- PC (Program Counter): Saves the address of the next instruction to be executed.
- IR (Instruction Register): Holds the current instruction being executed.
- ALUresult: Stores the result of operations performed by the ALU.
- memData: Holds data read from memory.
- writeData: Holds data that will be written back to a register.

States

The microprocessor operates through a finite state machine (FSM) with the following states:

1. **FETCH (000)**: The microprocessor fetches the next instruction from memory using the PC and stores it in the IR.
2. **DECODE (001)**: The instruction in IR is decoded to determine the operation to be performed and the registers involved, it takes the first 4 bits as the Rs2 and the next 4 as Rs1 then another 4 as the Rd and finally the last 4 are the OpCode.
3. **EXECUTE (010)**: The operation specified by the instruction is getting executed in this stage, such as arithmetic operations performed by the ALU which are specified in table1 below.
4. **MEM_ACCESS (011)**: If the instruction involves reading from or writing to memory, this state handles the memory operations.
5. **WRITE_BACK (100)**: The result of the operation (either from the ALU or memory) is written back to the destination register.

State Transitions

- The microprocessor starts in the FETCH state, where it loads the next instruction.
- It then moves to the DECODE state to specify what the instruction does.
- If the instruction is an R-type it will move to EXECUTE to perform an operation and if it's a M-type instruction it will move MEM_ACCESS.
- Finally, the microprocessor moves to the WRITE_BACK state to store the result on Rd taking the value from memData if MemRd is 1 or from the ALUresult is its 0, then it loops back to FETCH to start the process again with the next instruction.

Operation	Opcode
A+B	0
A-B	1
A AND B	2
A OR B	3
A XOR B	4

Table 1: ALU op codes

Code:

```
module MCP(
    input clk,
    input reset,
    output reg [2:0] current_state, // Output current state for waveform
    output reg [2:0] next_state,    // Output next state for waveform
    output reg [15:0] PC,           // Output PC for waveform
    output reg [15:0] IR,           // Output Instruction Register for waveform
    output reg [15:0] ALUresult,    // Output ALU result for waveform
    output reg [15:0] memData,      // Output memory data for waveform
    output reg [15:0] writeData     // Output data to be written back for waveform

);

    // Define states for the FSM using parameters
    parameter FETCH      = 3'b000;
    parameter DECODE     = 3'b001;
    parameter EXECUTE    = 3'b010;
    parameter MEM_ACCESS = 3'b011;
    parameter WRITE_BACK = 3'b100;

    // Control signals
    reg RegWr, MemRd, MemWr;

    // Decode signals
    reg [3:0] opCode;
    reg [3:0] Rd, Rs1, Rs2;

    // Register File connections
    wire [15:0] readData1, readData2;

    // ALU connections
    wire [2:0] ALUop;

    // Instruction Memory Instance
    instructionMem IM(
        .PC(PC),
        .instruction(instruction)
    );
```

```
    // Data Memory Instance
    dataMem dm(
        .in_data(readData2),
        .address(Rd),
        .clk(clk),
        .MemRd(MemRd),
        .MemWr(MemWr)
    );

    // Register File Instance
    RegisterFile RF(
        .clk(clk),
        .read_port1(Rs1),
        .read_port2(Rs2),
        .write_port(Rd),
        .writeEn(RegWr),
        .writeData(writeData),
        .readData1(readData1),
        .readData2(readData2)
    );

    // Control Unit Instance
    controlUnit CU(
        .opCode(opCode),
        .RegWr(RegWr),
        .MemRd(MemRd),
        .MemWr(MemWr),
        .ALUop(ALUop)
    );

```

```
    // FSM State transitions
    always @(posedge clk or negedge reset) begin
        if (!reset)
            current_state <= FETCH;
        else
            current_state <= next_state;
    end

    // FSM Control Logic: Determine next state
    always @(*) begin
        case (current_state)
            FETCH:    begin
            next_state = DECODE;
            end
            DECODE: begin
                if (opCode <= 4'b0100) //  R-Type opcode
                    next_state = EXECUTE;
                else if (MemRd || MemWr)
                    next_state = MEM_ACCESS;
                else
                    next_state = FETCH;
            end
            EXECUTE: begin
                if (opCode <= 4'b0100) // R-Type instruction
                    next_state = WRITE_BACK;
                else
                    next_state = FETCH;
            end
            MEM_ACCESS: begin
                if (MemRd)  //load
                    next_state = WRITE_BACK;
                else if (MemWr) //store
                    next_state = FETCH;
                else
                    next_state = FETCH;
            end
            WRITE_BACK: next_state = FETCH;
            default:    next_state = FETCH;
        endcase
    end

```

```
    // FETCH Stage
    always @(posedge clk or negedge reset) begin
        if (!reset)
            begin
            PC <= 16'b0;
           IR <= 16'b0;
            end
        else if (current_state == FETCH)
            begin
            IR <= IM.instruction; // Fetch instruction
           PC <= PC + 16'd2;
           end
    end

    // DECODE Stage
    always @(*) begin
        if (current_state == DECODE) begin
            opCode = IR[15:12];
            Rd = IR[11:8];
            Rs1 = IR[7:4];
            Rs2 = IR[3:0];
        end
    end
```

```
    end

    // EXECUTE Stage
    always @(*) begin
        if (current_state == EXECUTE) begin
            case (ALUop)
                3'b000: ALUresult = readData1 + readData2;
                3'b001: ALUresult = readData1 - readData2;
                3'b010: ALUresult = readData1 & readData2;
                3'b011: ALUresult = readData1 | readData2;
                3'b100: ALUresult = readData1 ^ readData2;
                default: ALUresult = 16'b0;
            endcase
        end
    end
always @(posedge clk) begin
    if (current_state == MEM_ACCESS) begin
        if (MemRd) begin
            memData <= dm.dataMemory[RF.register[Rs1]];  // Load data
            $display("LOAD: Address = %h, Data = %h", RF.register[Rs1], memData);
        end
        if (MemWr) begin
            dm.dataMemory[RF.register[Rs1]] <= RF.register[Rd];  // Store data
            $display("STORE: Address = %h, Data = %h", RF.register[Rs1], RF.register[Rd]);
        end
    end
end
// WRITE-BACK Stage
always @(posedge clk) begin
    if (current_state == WRITE_BACK) begin
        if (RegWr) begin
            if (MemRd) begin
                writeData <= memData;  // Write loaded data to Reg[Rd]
            end else begin
                writeData <= ALUresult;  // Write ALU result to Reg[Rd]
            end
            RF.register[Rd] <= writeData;  // Write-back to register
        end
    end
end


endmodule
```

```
297
298 //*********************************************instruction Memory*************************************************************************************//
299 module instructionMem(
300     input [15:0] PC,
301     output reg [15:0] instruction
302 );
303
304
305     reg [15:0] memory [0:255];
306
307    initial begin
308     memory[0]  = 16'b0110001100100011;
309     memory[1]  = 16'b0001010001100101;
310     memory[2]  = 16'b0010000100110010;
311     memory[3]  = 16'b0110000100100110;
312     memory[4]  = 16'b0100000101100101;
313     memory[5]  = 16'b0110001000110000;
314     memory[6]  = 16'b0110001010000011;
315     memory[7]  = 16'b0111000000010001;
316     memory[8]  = 16'b0000000000101000;
317     memory[9]  = 16'b0110000000110111;
318     memory[10] = 16'b0010000001000100;
319     memory[11] = 16'b0011000001010001;
320     memory[12] = 16'b0100000010001100;
321     memory[13] = 16'b0101000010010110;
322     memory[14] = 16'b0110000010100001;
323     memory[15] = 16'b0110000010110110;
324 end
325
326
327     always @ (PC) begin
328         instruction = memory[PC[7:0]];
329     end
330
331 endmodule
332
333
```

```
181
182 module RegisterFile(
183     input clk,
184     input[3:0] read_port1, read_port2, write_port,
185     input writeEn,
186     input [15:0] writeData,
187     output [15:0] readData1, readData2
188     );
189     reg [15:0] register [15:0] = '{0, 1, 2, 3, 7,
190                                    6, 4, 9, 8 , 1 , 5 , 6,3 ,4 ,6,9};
191
192     assign readData1 = register[read_port1];
193     assign readData2 = register[read_port2];
194
195     always @(posedge clk)
196     begin
197         if(writeEn)
198             register[write_port] <= writeData;
199     end
200     endmodule
201
202
203 //*********************************************data Memory*************************************************************************************//
204
205 module dataMem(
206     input [15:0] in_data,
207     input [7:0] address,
208     input clk,
209     input MemRd,
210     input MemWr
211     );
212     integer i;
213
214     reg [15:0] dataMemory [0:255];
215     initial begin
216     for(i=0; i<=255; i++)
217                 dataMemory[i] = 16'b0;
218 end
219
220     always @ (posedge clk)
221         begin
222             if(MemWr)
223                 dataMemory[address] <= in_data;
224         end
225         endmodule
```

```
//*****************************************control Unit*****************************************************************//
module controlUnit (
    input [3:0] opCode,
    output reg RegWr, MemRd, MemWr,
    output reg [2:0] ALUop
);

    always @(*)
    begin
        case(opCode)
            4'b0000:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b000;
            end
            4'b0001:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b001;
            end
            4'b0010:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b010;
            end
            4'b0011:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b011;
            end
            4'b0100: begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b100;
            end
```

```
            4'b0010:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b010;
            end
            4'b0011:
            begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b011;
            end
            4'b0100: begin
                RegWr = 1;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b100;
            end
            4'b0101: begin
                RegWr = 1;
                MemRd = 1;
                MemWr = 0;
                ALUop = 3'bxxx;
            end
            4'b0110:
            begin
                RegWr = 0;
                MemRd = 0;
                MemWr = 1;
                ALUop = 3'bxxx;
            end
            default:
            begin
                RegWr = 0;
                MemRd = 0;
                MemWr = 0;
                ALUop = 3'b000;
            end
            endcase
            end

endmodule
```

Design:

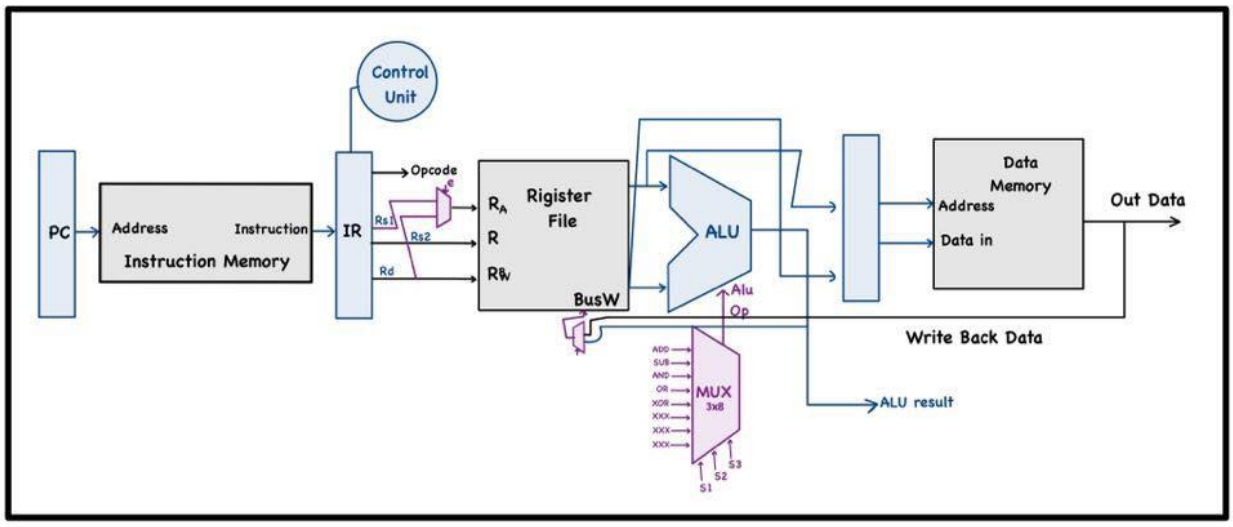

Figure 1: Design

Test cases and Simulation results

Test Bench:

```
//**********************************************Test Bench**********************************************************************************//
module test_MCP;

    // Inputs
    reg clk;
    reg reset;

    // Outputs
    wire [2:0] current_state;
    wire [2:0] next_state;
    wire [15:0] PC;
    wire [15:0] IR;
    wire [15:0] ALUresult;
    wire [15:0] memData;
    wire [15:0] writeData;

    // Instantiate the MCP module
    MCP uut (
        .clk(clk),
        .reset(reset),
        .current_state(current_state),
        .next_state(next_state),
        .PC(PC),
        .IR(IR),
        .ALUresult(ALUresult),
        .memData(memData),
        .writeData(writeData)
    );

    // Clock generation
    always #5 clk = ~clk;  // 10 ns clock period

    // Testbench procedure
    initial begin
        // Initialize inputs
        clk = 0;
        reset = 0;

        // Apply reset
        #10 reset = 1;
```

```
        // Initialize data memory with test values
        uut.dm.dataMemory[0] = 16'h0001;
        uut.dm.dataMemory[1] = 16'h0002;
        uut.dm.dataMemory[2] = 16'h0003;
        uut.dm.dataMemory[3] = 16'h0004;
        uut.dm.dataMemory[4] = 16'h0005;
        uut.dm.dataMemory[5] = 16'h0006;
        uut.dm.dataMemory[6] = 16'h0007;
        uut.dm.dataMemory[7] = 16'h0008;
        // Add more initial values if needed


            // Populate the instruction memory with more memory operations
            uut.IM.memory[0] = 16'b0101000100100001;  // LOAD R2, [R1] (Load from address in R1)
            uut.IM.memory[1] = 16'b0110001000110010;  // STORE R3, [R2] (Store to address in R2)
            uut.IM.memory[2] = 16'b0101001100100011;  // LOAD R4, [R3] (Load from address in R3)
            uut.IM.memory[3] = 16'b0110001100110100;  // STORE R5, [R4] (Store to address in R4)
            uut.IM.memory[4] = 16'b0101010000100101;  // LOAD R6, [R5] (Load from address in R5)
            uut.IM.memory[5] = 16'b0110010000110110;  // STORE R7, [R6] (Store to address in R6)
            uut.IM.memory[6] = 16'b0101010100100111;  // LOAD R8, [R7] (Load from address in R7)
            uut.IM.memory[7] = 16'b0110010100111000;  // STORE R9, [R8] (Store to address in R8)

            // Allow some time for the MCP module to execute
            #400;

            // Finish the simulation
            $stop;
        end

        // Monitor signals for debugging
        initial begin
            $monitor("Time: %0d | State: %b | PC: %h | IR: %h | ALUresult: %h | memData: %h | writeData: %h",
                     $time, current_state, PC, IR, ALUresult, memData, writeData);
        end

endmodule
```

Figure 2: Test Bench

Results:

```
Console
run
# KERNEL: Time: 0 | State: 000 | PC: 0000 | IR: 0000 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: Time: 15 | State: 001 | PC: 0002 | IR: 6323 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: Time: 25 | State: 011 | PC: 0002 | IR: 6323 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: STORE: Address = 0004, Data = 0003
# KERNEL: Time: 35 | State: 000 | PC: 0002 | IR: 6323 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: Time: 45 | State: 001 | PC: 0004 | IR: 5323 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: Time: 55 | State: 011 | PC: 0004 | IR: 5323 | ALUresult: xxxx | memData: xxxx | writeData: xxxx
# KERNEL: LOAD: Address = 0004, Data = xxxx
# KERNEL: Time: 65 | State: 100 | PC: 0004 | IR: 5323 | ALUresult: xxxx | memData: 0003 | writeData: xxxx
# KERNEL: Time: 75 | State: 000 | PC: 0004 | IR: 5323 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 85 | State: 001 | PC: 0006 | IR: 5425 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 95 | State: 011 | PC: 0006 | IR: 5425 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: LOAD: Address = 0004, Data = 0003
# KERNEL: Time: 105 | State: 100 | PC: 0006 | IR: 5425 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 115 | State: 000 | PC: 0006 | IR: 5425 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 125 | State: 001 | PC: 0008 | IR: 5527 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 135 | State: 011 | PC: 0008 | IR: 5527 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: LOAD: Address = 0004, Data = 0003
# KERNEL: Time: 145 | State: 100 | PC: 0008 | IR: 5527 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 155 | State: 000 | PC: 0008 | IR: 5527 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 165 | State: 001 | PC: 000a | IR: 0028 | ALUresult: xxxx | memData: 0003 | writeData: 0003
# KERNEL: Time: 175 | State: 010 | PC: 000a | IR: 0028 | ALUresult: 000d | memData: 0003 | writeData: 0003
# KERNEL: Time: 185 | State: 100 | PC: 000a | IR: 0028 | ALUresult: 000d | memData: 0003 | writeData: 0003
# KERNEL: Time: 195 | State: 000 | PC: 000a | IR: 0028 | ALUresult: 000d | memData: 0003 | writeData: 000d
# KERNEL: Time: 205 | State: 001 | PC: 000c | IR: 2044 | ALUresult: 000d | memData: 0003 | writeData: 000d
# KERNEL: Time: 215 | State: 010 | PC: 000c | IR: 2044 | ALUresult: 0003 | memData: 0003 | writeData: 000d
# KERNEL: Time: 225 | State: 100 | PC: 000c | IR: 2044 | ALUresult: 0003 | memData: 0003 | writeData: 000d
# KERNEL: Time: 235 | State: 000 | PC: 000c | IR: 2044 | ALUresult: 0003 | memData: 0003 | writeData: 0003
# KERNEL: Time: 245 | State: 001 | PC: 000e | IR: 408c | ALUresult: 0003 | memData: 0003 | writeData: 0003
# KERNEL: Time: 255 | State: 010 | PC: 000e | IR: 408c | ALUresult: 000a | memData: 0003 | writeData: 0003
# KERNEL: Time: 265 | State: 100 | PC: 000e | IR: 408c | ALUresult: 000a | memData: 0003 | writeData: 0003
# KERNEL: Time: 275 | State: 000 | PC: 000e | IR: 408c | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 285 | State: 001 | PC: 0010 | IR: 60a1 | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 295 | State: 011 | PC: 0010 | IR: 60a1 | ALUresult: 000a | memData: 0003 | writeData: 000a
> restart
```

```
# KERNEL: STORE: Address = 0006, Data = 000a
# KERNEL: Time: 305 | State: 000 | PC: 0010 | IR: 60a1 | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 315 | State: 001 | PC: 0012 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 325 | State: 000 | PC: 0012 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 335 | State: 001 | PC: 0014 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 345 | State: 000 | PC: 0014 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 355 | State: 001 | PC: 0016 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 365 | State: 000 | PC: 0016 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 375 | State: 001 | PC: 0018 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 385 | State: 000 | PC: 0018 | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 395 | State: 001 | PC: 001a | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# KERNEL: Time: 405 | State: 000 | PC: 001a | IR: xxxx | ALUresult: 000a | memData: 0003 | writeData: 000a
# RUNTIME: Info: RUNTIME_0070 MCP.v (402): $stop called.
```

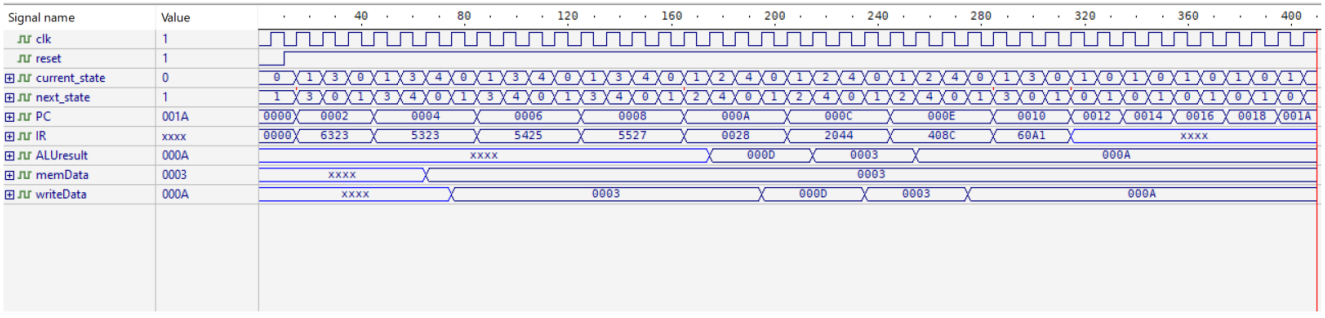

Figure 3: Results

Initially the pc starts at 0 and the IR stores the instruction in the 0 address which is 6323 and stores the value of R3 which is 3 on the data memory with as address of 4 since this is value loaded into R1 and as shown in the wave form the processor has successfully done this instruction since writeData is 3 which is the data will be written back on the memory.

Then, when pc is 2 the IR is 5323 which is supposed to load the value of memory[4] since 4 is the value inside R2, now the value of memory[4] is 3 since this is what we stored in it in the previous instruction so the memData should be 3 which is correct as shown in the wave form.

The results in ALU operations are correctly performed note that when the pc is 8 the IR is 0028 (with a bit of delay) which is the value we initially stored in the instruction memory this instruction is ADD R0 R2 R8 and the values in R2, R8 are 4,9 respectively so the ALUresult is expected to be 10 which is true and the result was successfully loaded into the writeData to be loaded into the result register (Rd), proving the ability of the processor in performing arithmetic operations.

The remaining instructions are handled similarly, proving the efficiency of the processor.

Conclusion and future work

In this project, we successfully designed and verified a multi-cycle processor that proved its ability of handling instructions over several clock cycles. The design was implemented using Verilog designing multiple modules such as Memory, Data path, Control unit and a top module combining them all, each stage functioned correctly from FETCH to WRITE-BACK ensuring correct execution. For future improvements, the processor could be improved by adding additional ALU operations for wider arithmetic functions, as well as providing conditional instructions such as jump and branch. These improvements would increase the processor's capability to handle complex tasks and increase its efficiency and adaptability for more varied tasks.