BERZIET UNIVERSITY

Faculty of Engineering & Technology – Electrical & Computer Engineering Department

First Semester 2022/2023

COMPUTER ARCHITECTURE ENCS4370

Simple Multi-Cycle RISC processor

Project2 Report

Prepared by:	Yousef Hammad Mohammad AbuJaber Malek Abotouq
ID:	1170625 1190298 1190186
Date:	February 15, 2023
Section:	1

Abstract:

This project aims to design and test a simple Multi-Cycle RISC processor using the Verilog language. The processor has a word size of 24-bits, and all instructions are conditionally executed. It includes eight 24-bit general-purpose registers, a 24-bit program counter, and an 8-bit status register with only the least significant bit being used as a zero flag. The ALU instructions have the option of updating the flag bits, and there are three instruction types (R-type, I-type, and J-type) and five addressing modes. The data path is constructed with a five-stage pipeline and register between stages, and the control logic is designed using a state machine approach. The implementation of the Multi-Cycle Datapath and its control logic will provide a platform for further testing and improvement of the RISC processor design.

Table of Contents

1. Designing the Datapath and Control Signals 5

2. Building the State Machine Diagram 6

3. Testing the Modules 7

3.1. Adder Testbench 8

3.2. Comparator Testbench 8

3.3. Multiplexer Testbench 9

3.4. Extender Testbench 10

3.5. Registers File Testbench 11

3.6. ALU Testbench 12

3.7. Control Unit Testbench 14

3.8. PC Register Testbench 15

3.9. Data Memory Testbench 16

3.10. Machine 17

Conclusion 18

Appendix 19

Table of Figures

Figure 1: Datapath 5
Figure 2: States and Control Signals 7
Figure 3: Adder Code 8
Figure 4: Adder Waveform 8
Figure 5: Comparator Code 8
Figure 6: Comparator Waveform 9
Figure 7: Multiplexer Code 9
Figure 8: Multiplexer Waveform 10
Figure 9: Extender Code 10
Figure 10: Extender Waveform 11
Figure 11: Registers File Code 11
Figure 12: Registers File Waveform 12
Figure 13: ALU Code 12
Figure 14: ALU Waveform (SF = 0) 13
Figure 15: ALU Waveform (SF = 1) 13
Figure 16; Control Unit 14
Figure 17: PC Register Code 15
Figure 18: PC Register Waveform 15
Figure 19: Data Memory Code 16
Figure 20: Data Memory Waveform 16
Figure 21: Machine Waveform 17
Figure 22: LUI Implementation 17
Figure 23: ADDI Implementation 17

1. Designing the Datapath and Control Signals

The datapath was built as shown in the figure below:

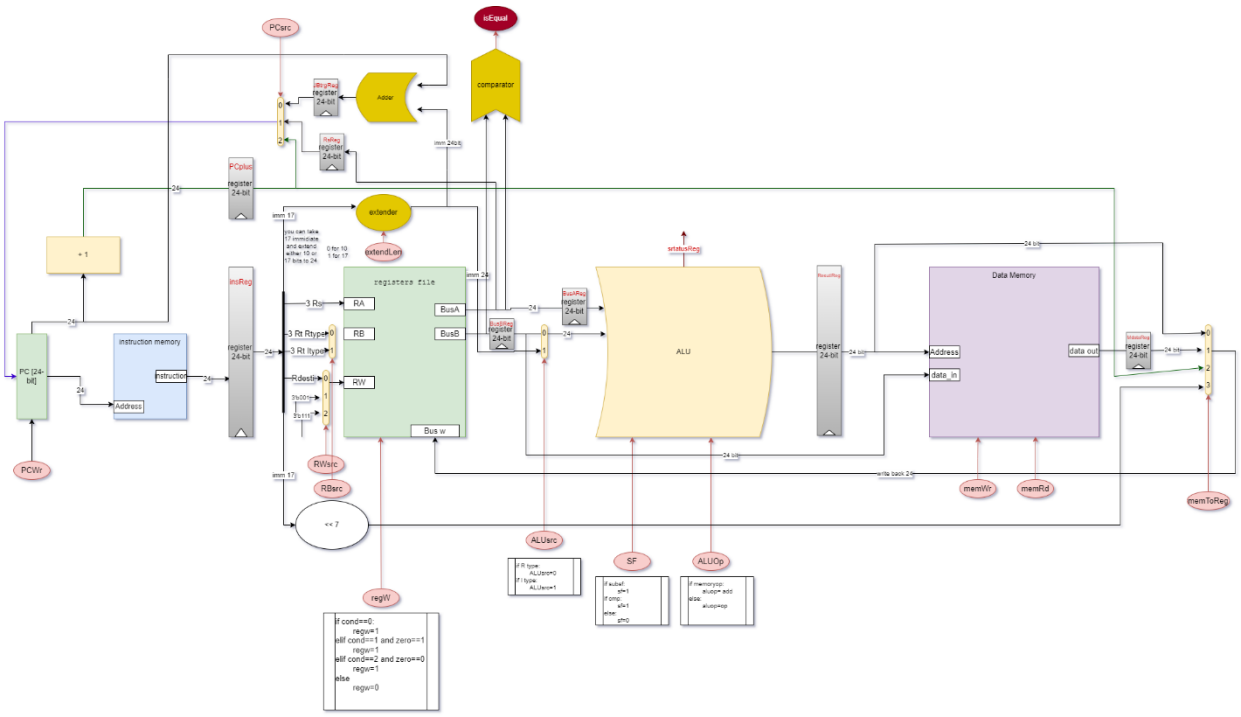

Figure 1: Datapath

The design of the data path and control signals for a Multi-Cycle Processor with Word Accessible Memory had already been completed. In the instruction fetch stage, the program counter, and an incrementor by 1 were added to instruction fetch stage. The result was transmitted as input to instruction memory.

In the instruction decode stage, the register file was used to read values from the general-purpose registers. An extender was used to extend the immediate value to 24 bits, and a multiplexer was used to choose between three PC sources: adding the immediate extended value, BusA output, and PC incremented by 1. A comparator was used to compare BusA and BusB, and a write enable signal was also implemented. Additionally, a multiplexer was used to choose between three registers as the destination register: Rdest, R1, and R7.

The ALU stage had two operands and the second operand was chosen using a multiplexer to choose between the BusB output and the Immediate value. The ALU_OP was used to choose from various instructions such as ADD, ADDI, AND, ANDI, SUB, SUBI, CMP, CA. 8-bit status register was added which will be changed only when using the SUB and CMP instructions (SF signal = 1).

The data memory stage had both a write and read enable signals. The write back stage used a multiplexer to return one of the following values: ALU output, data out, PC+1, or load upper immediate that shifted the immediate 17-bit value by 7.

The design of the data path and control signals for the Multi-Cycle Processor with Word Accessible Memory was a comprehensive solution that covered all the necessary components and stages for efficient processing.

2. Building the State Machine Diagram

To build the state machine diagram for the stages of the multi-cycle processor, a systematic approach had been followed. Firstly, the sequence of operations that needed to be performed during each stage and the conditions under which they were executed were analyzed. Based on this information, the various states that were required in the state machine diagram were identified. Secondly, the transitions between the states were defined, which were triggered by the control signals generated by the processor. For example, the transition from the instruction fetch stage to the instruction decode stage was triggered by the "state number" value, indicating that the instruction had been fetched from memory and was ready for decoding. Furthermore, the actions that would be performed during each state were also defined, such as incrementing the PC or writing to the register file. Finally, the correctness of the state machine diagram was verified by simulating the operation of the processor and comparing the results with the expected behavior.

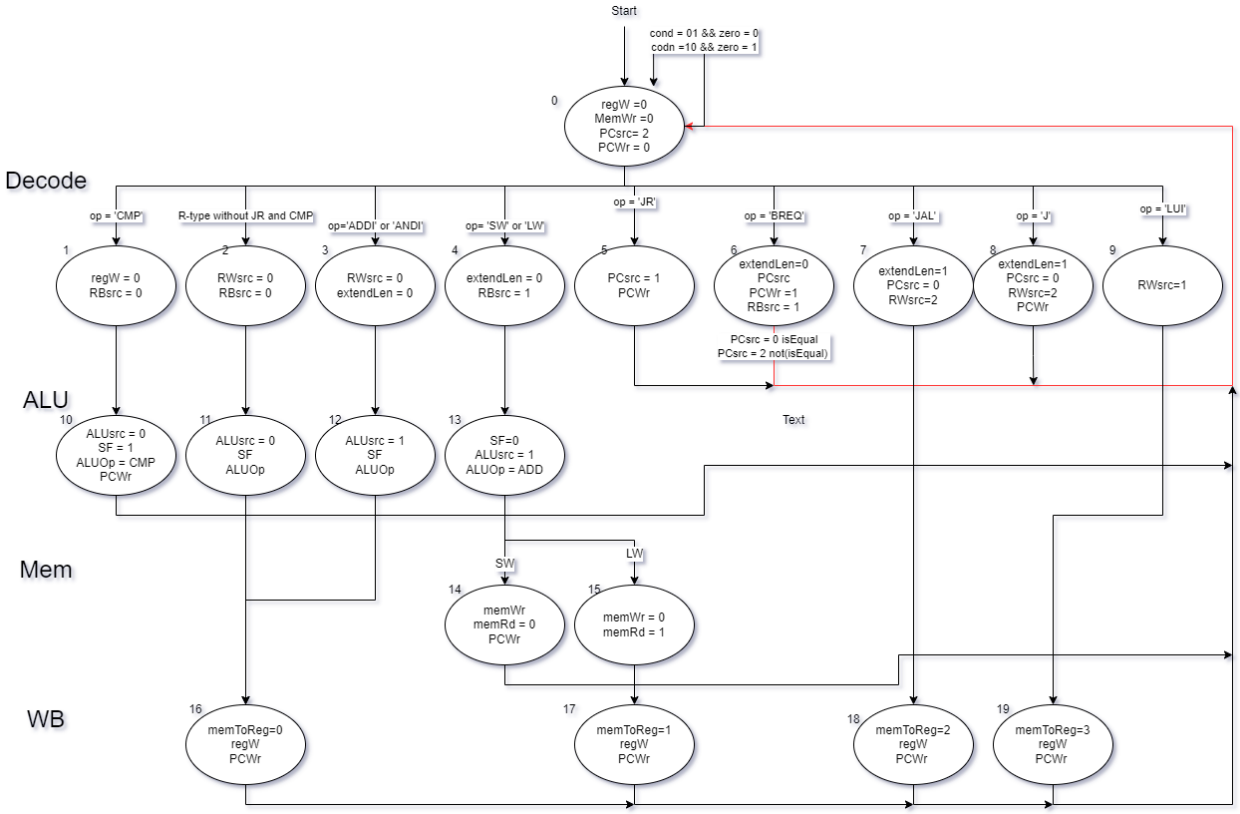

Figure 2: States and Control Signals

3. Testing the Modules

Testbenches were critical for verifying the correctness of a module's functionality, as they allowed designers to test modules in a controlled environment before integrating them into a larger system. Creating effective testbenches required an understanding of the module's specifications, input and output signals, and desired operating conditions. By simulating a variety of scenarios and stimuli, designers could ensure that their modules functioned as intended and could detect and correct errors early in the design process. In this way, testbenches were a crucial tool for ensuring the quality and reliability of Verilog designs. Testbenches were implemented for all modules to cover some certain conditions not all possible scenarios.

3.1. Adder Testbench

```
module adder(A, B, out);
    input[23:0] A, B;
    output[23:0] out;

    assign out = A + B;
endmodule

module adder_tb();
    reg[23:0] A, B;
    wire[23:0] out;
    adder adder1(A, B, out);

    initial begin
        #5ns A = 24'h123456;
        B = 24'h123343;
        #5ns A = 24'h123456;
        B = 24'hFF1334;

    end
endmodule
```

Figure 3: Adder Code

The used code defines a Verilog module called "adder" that performs the addition operation on two 24-bit input signals A & B and produces a 24-bit output signal named "out". The code also defines a testbench module called "adder_tb" that instantiates the "adder" module and provides test stimulus to its inputs A and B. The "initial" block in the testbench assigns initial values to A and B and provides two sets of input stimuli to test the module's functionality. Overall, the code tests the functionality of the "adder" module by simulating two addition operations with different inputs.

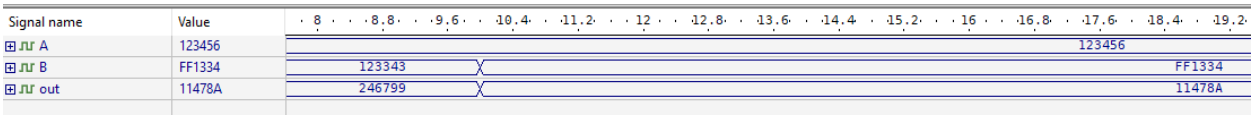

Figure 4: Adder Waveform

The output is as expected as 123456 + 123343 = 246799 and 123456 + FF1334 = 11478A.

3.2. Comparator Testbench

```
module comparator(A, B, eq);
    input[23:0] A, B;
    output eq;

    assign eq = (A==B)? 1:0;

endmodule

module comparator_tb();
    reg[23:0] A, B;
    wire eq;
    comparator c1(A, B, eq);
    initial begin
        #5ns A = 24'h444444;
        B = 24'h444444;
        #5ns A = 24'h444444;
        B = 24'h444344;
        #5ns A = 24'h444444;
        B = 24'h444444;
        #5ns A = 24'h444444;
        B = 24'h444344;
    end

endmodule
```

Figure 5: Comparator Code

The code defines a Verilog module called "comparator" that compares two 24-bit input signals A and B and produces a single-bit output signal "eq", which is 1 if A and B are equal and 0 if they are not. The code also defines a testbench module called "comparator_tb" that instantiates the "comparator" module and provides test stimulus to its inputs A and B. The "initial" block in the testbench assigns initial values to A and B and provides several sets of input stimuli to test the module's functionality. Overall, the code tests the functionality of the "comparator" module by simulating the comparison of different pairs of input values.

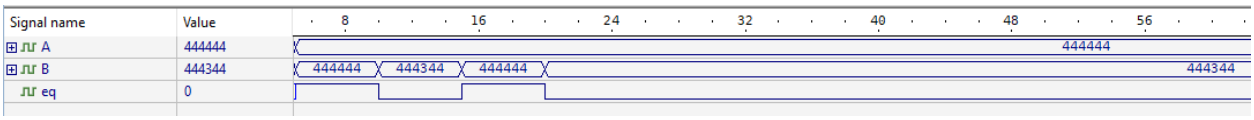

Figure 6: Comparator Waveform

When A = 444444 and B = 444444, the "eq" signal was set to 1. Meanwhile, it was set to 0 when B was set to 444344 which was already expected.

3.3. Multiplexer Testbench

```
module mux4X1(select,I0, I1, I2, I3, out);
    parameter n = 24;
    input[1:0] select;
    input[(n-1):0] I0, I1, I2, I3;
    output[(n-1):0] out;

    assign out = (select[1]) ? ((select[0])? I3: I2):((select[0])? I1: I0);


endmodule

module mux4X1_tb();
    reg[1:0] select;
    reg[23:0] I0, I1, I2, I3;
    wire[23:0] out;
    mux4X1 m41(select,I0, I1,I2, I3, out);

    initial begin
        I0 <= 24'h111222;
        I1 <= 24'h333444;
        I2 <= 24'h555666;
        I3 <= 24'h777888;
        select <= 0;
        repeat(3)
        #5ns select <= select + 1;
    end
endmodule
```

Figure 7: Multiplexer Code

The code defines a Verilog module called "mux4X1" that implements a 4-to-1 multiplexer. The module takes two 2-bit input signals, "select" and "n", and four 24-bit input signals I0, I1, I2, and I3. It outputs a 24-bit signal "out" that carries the selected input. The code also defines a testbench module called "mux4X1_tb" that instantiates the "mux4X1" module and tests its functionality by providing different values to the input signals. The testbench sets the initial values for I0, I1, I2, and I3 and sets the select signal to 0. It then repeats this process with different select inputs at 5 nanoseconds intervals. Overall, the code tests the functionality of the 4-to-1 multiplexer module by simulating different input selections using the testbench.

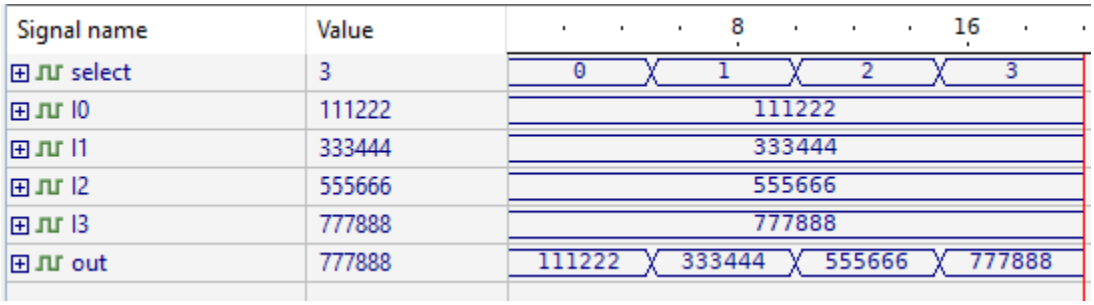

Figure 8: Multiplexer Waveform

The functionality of the multiplexer is like the given code:

When I0 = 111222

I1 = 333444

I2 = 555666

I3 = 777888

3.4. Extender Testbench

```
module extender(imm, extendLen, out);
    input extendLen;
    input[16:0] imm;
    output reg[23:0] out;

    always@* begin
        if (extendLen == 0) begin
            out[9:0] = imm[9:0];
            for (int i=10;i<24;i++) begin
                out[i] = imm[9];
            end
        end
        else begin
            out[16:0] = imm[16:0];
            for (int i=17;i<24;i++) begin
                out[i] = imm[16];
            end
        end
    end

endmodule

module extender_tb();
    reg extendLen;
    reg[16:0] imm;
    wire[23:0] out;
    extender e(imm, extendLen, out);

    initial begin
        #5ns extendLen = 0;
        imm = 17'h01134;
        #5ns extendLen = 0;
        imm = 17'h01334;
        #5ns extendLen = 1;
        imm = 17'h01134;
        #5ns extendLen = 1;
        imm = 17'h11334;

    end

endmodule
```

Figure 9: Extender Code

The first Verilog module is named extender. It extends an input value immediate to a specified length given by extendLen. If extendLen is 0, the lower 10 bits of immediate are sign-extended to 24 bits, and if extendLen is 1, the lower 17 bits of immediate are sign-extended to 24 bits. The second module is a testbench for the extender module. It sets the value of extendLen and immediate for 4 different test cases and checks the out value.

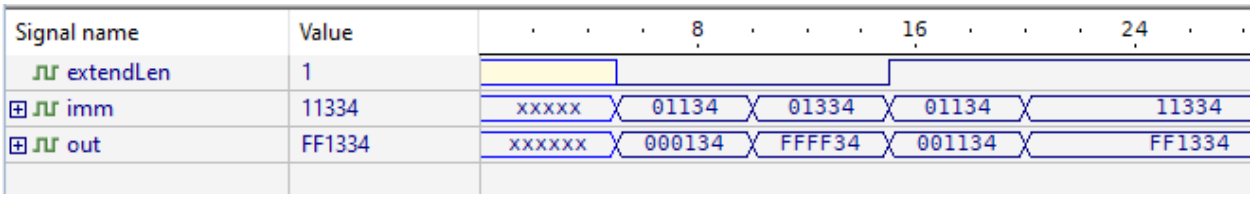

Figure 10: Extender Waveform

3.5. Registers File Testbench

```
module registers_file(RA, RB, RW, enableW, BusW, BusA, BusB);
    input[2:0] RA, RB, RW;
    input[23:0] BusW;
    input enableW;
    output [23:0] BusA, BusB;
    reg [23:0] file[0:7];

    initial begin
        file[0] = 0;
    end
    assign BusA = file[RA];
    assign BusB = file[RB];
    always@* begin
        if ((enableW==1) && (RW != 0)) begin
            file[RW] <= BusW;
        end
    end
endmodule

module rf_tb();
    reg[2:0] RA, RB, RW;
    reg[23:0] BusW;
    reg enableW;
    wire[23:0] BusA, BusB;
    registers_file rf(RA, RB, RW, enableW, BusW, BusA, BusB);
    initial begin
        RA = 1;
        RB = 0;
        RW = 1;
        enableW = 1;
        BusW = 24'h123456;
        repeat(6) begin
            #5ns RA <= RA + 1;
            RB <= RB + 1;
            RW <= RW + 1;
            //enableW <= !enableW;
            BusW <= BusW + 1;
        end
    end
endmodule
```

Figure 11: Registers File Code

The provided code defines a Verilog module to implement a register file with 8 registers, and a testbench module to test it. The register file module has 3 read ports (RA, RB, and RW), 2 write ports (BusW and enableW), and 2 output ports (BusA and BusB) that provide the value of the register read by RA and RB, respectively. The write ports allow writing to a register indicated by the RW input if enableW is asserted. The initial state of the register file has the first register initialized to 0. The testbench initializes the registers and write values to them, then increments the RA, RB, RW, and BusW values for 6 clock cycles, testing that the correct values are read from the correct registers.

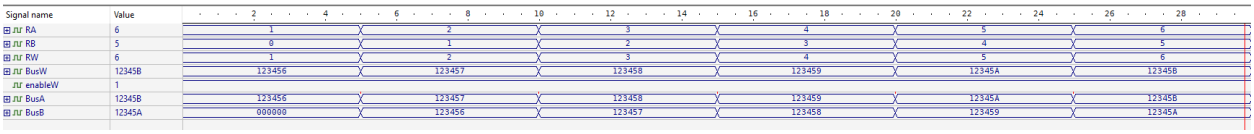

Figure 12: Registers File Waveform

3.6. ALU Testbench

```
module ALU (operand1,operand2,ALU_OP,sfseg,res,statusReg);
    input [23:0] operand1;
    input [23:0] operand2;
    input [2:0] ALU_OP;
    input sfseg;
    output [23:0] res;
    output reg [7:0]statusReg;

    reg [23:0] result;
    initial begin
        statusReg = 8'b00000000;
    end
    always @(*) begin
        case (ALU_OP)
            3'b000: result = operand1 & operand2;//and + andi
            3'b001: result = operand1 < operand2 ? operand2 : operand1;//max
            3'b011: result = operand1 + operand2;//add + addi
            3'b100: result = operand1 - operand2;//sub + subi
            3'b101: result = operand1 >= operand2;//cmp
            default: result = 0;
        endcase
        if (sfseg == 1) begin
             statusReg[0] = (result==0);
            end
    end
    assign res = result;
endmodule

module ALU_tb;
    reg [23:0] operand1, operand2;
    reg [2:0] ALU_OP;
    reg sfseg;
    wire [23:0] res;
    wire [7:0]statusReg;
    ALU uut (
        .operand1(operand1),
        .operand2(operand2),
        .ALU_OP(ALU_OP),
        .sfseg(sfseg),
        .res(res),
        .statusReg(statusReg)
    );
    initial begin
        sfseg = 0;
        operand1 = 24'h012121;
        operand2 = 24'h987654;
        ALU_OP = 0;
        repeat(6)
            #5ns ALU_OP = ALU_OP + 1;
    end
endmodule
```

Figure 13: ALU Code

This code defines a module named "ALU" which performs arithmetic and logic operations on two input operands "operand1" and "operand2". The module takes a 3-bit control input "ALU_OP" which selects the type of operation to be performed. The output of the module is the result of the operation, "res", and an 8-bit register, "statusReg", of which the least significant bit will be used to indicate whether the result is zero or not.

The module performs the operation based on the value of "ALU_OP" using a case statement. The module checks the "sfseg" input and sets the first bit of the "statusReg" output to 1 if the result of the operation is 0. The module also initializes "statusReg" to all zeros using an initial block. The module is tested using a testbench named "ALU_tb". The testbench provides input values for "operand1", "operand2", and "ALU_OP" and reads the output values "res" and "statusReg". The testbench initializes "ALU_OP" to 0 and increments it six times after a delay of 5ns in each iteration.

The "sfseg" input was set to 0 in the first testbench as shown:

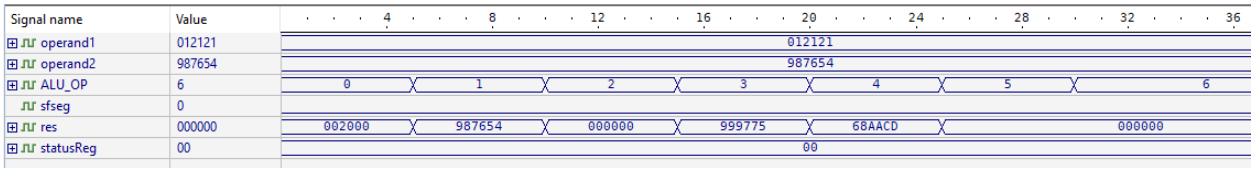

Figure 14: ALU Waveform (SF = 0)

When ALU_OP = 0 ➔ res = 012121 and 987654:

	0000	0001	0010	0001	0010	0001
&	1001	1000	0111	0110	0101	0100
=	0000	0000	0010	0000	0000	0000

When ALU_OP = 1 ➔ res = Max [012121, 987654] = 987654.

When ALU_OP = 3 ➔ res = 012121 + 987654 = 999775.

Then, the "sfseg" input was set to 1 in the second testbench to see the change in "statusReg" output as shown:

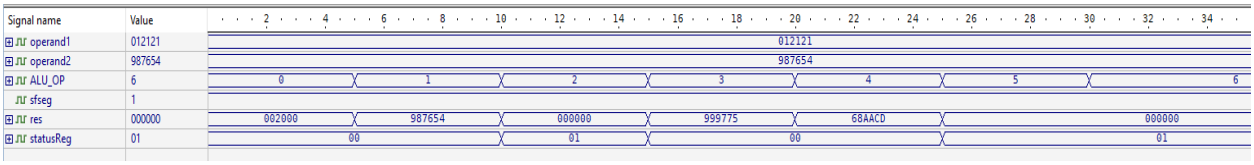

Figure 15: ALU Waveform (SF = 1)

When ALU_OP = 5 ➔ since operand1 < operand2 ➔ statusReg = 1.

3.7. Control Unit Testbench

This code represents a control unit module for a microprocessor. The module takes inputs including a clock signal, condition, operation, various control signals and status registers, and outputs several control signals including ALUOp, read/write source, program counter, memory read/write, and memory-to-register. The module consists of an initial block that sets all output signals to zero and an always block that updates the state of the module based on the current state and the input values. The state variable represents the current state of the module and is updated based on the current opcode and the current state. The always block contains a case statement that determines the next state of the module based on the current state and the opcode. The module includes support for various instructions including arithmetic and logic instructions (AND, ADD, SUB), memory access instructions (LW, SW), control transfer instructions (JR, BEQ, J, JAL), and comparison instruction (CMP).

The module also supports setting various control signals based on the input values, such as setting the ALU operation and memory read/write signals based on the opcode and setting the program counter source signal based on the current state and condition code.

Overall, this module acts as the control unit for the microprocessor, determining the appropriate control signals for each instruction based on the current state and input values.

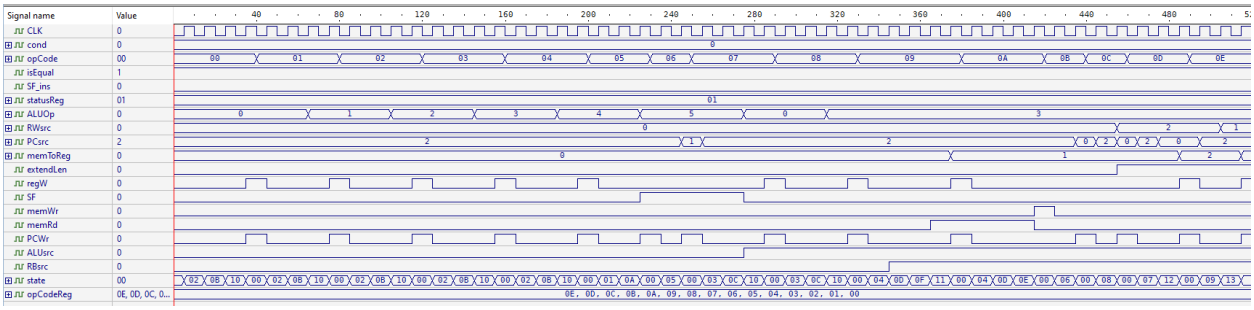

Figure 16; Control Unit

The waveform shows the change in the state corresponding to the change in control signals.

3.8. PC Register Testbench

```
module PCregister (
    input PCWr,
    input [23:0] input_address,
    output reg [23:0] address,

);

reg [23:0] address_reg;

always @* begin
    if (PCWr) begin
        address <= input_address;
    end
end

endmodule

module PCregister_tb;
  reg PCWr;
  reg [23:0] input_address;

  wire [23:0] address;
  PCregister pcr (
    .PCWr(PCWr),
    .input_address(input_address),
    .address(address)
  );

  initial begin
    PCWr = 0;
    input_address = 0;
    #5;
    PCWr = 1;
    input_address = 123;
    #5;
    PCWr = 0;
    input_address = 456;
    #5;
    PCWr = 1;
    input_address = 789;
    #5;
    PCWr = 0;
    #5;
  end
endmodule
```

Figure 17: PC Register Code

The Verilog code defines two modules: PCregister and PCregister_tb. The PCregister module is a 24-bit register that holds an address. It has three ports: PCWr, input_address, and address. When PCWr is asserted (1), the input value input_address is written to the register and the value is stored in the address output port. The PCregister_tb module is a testbench module that instantiates the PCregister module and provides input stimulus to it. It sets PCWr and input_address to specific values at certain times to simulate write operations.

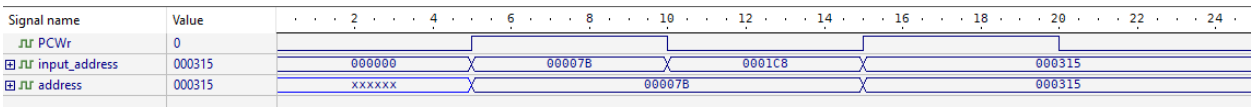

Figure 18: PC Register Waveform

It was noticed that the PC value don't change until the PCWr was 1.

3.9. Data Memory Testbench

```
module data_memory (input [23:0] address, input memWr, input memRd, input [23:0] data_in,
    output reg [23:0] data_out);

reg [23:0] memory [0:511];

always @* begin
    if (memWr) begin
        memory[address] <= data_in;
    end
end
assign data_out = memRd ? memory[address] : 24'h000000; // if memRd = 1 ==> data_out = mem[address] else ==> 0

endmodule

module data_memory_tb;

reg [23:0] address;
reg memWr, memRd;
reg [23:0] data_in;
wire [23:0] data_out;

data_memory mem (
    .address(address),
    .memWr(memWr),
    .memRd(memRd),
    .data_in(data_in),
    .data_out(data_out)
);

initial begin
    // write data to address 0 and 1
    address = 0;
    memWr = 1;
    data_in = 24'h012345;
    #10;
    address = 1;
    data_in = 24'h678901;
    #10;

    // read data from address 0 and 1
    address = 0;
    memWr = 0;
    memRd = 1;
    #10;
    address = 1;
    #10;
end

endmodule
```

Figure 19: Data Memory Code

The Verilog code defines two modules: data_memory and data_memory_tb. The data_memory module is a simple 24-bit data memory that holds 512 words of data. It has five ports: address, memWr, memRd, data_in, and data_out. When memWr is asserted (1), the value of data_in is written to the memory at the address specified by address. When memRd is asserted (1), the value stored in the memory at the address specified by address is output at data_out. If memRd is not asserted (0), data_out is set to 0. The data_memory_tb module is a testbench module that instantiates the data_memory module and provides input stimuli to it. It writes values to memory locations 0 and 1 using the memWr and data_in inputs, and then reads the values back using the memRd input and data_out output. The delays between the stimulus values are controlled using the #10 delay operator, which simulates 10 units of time.

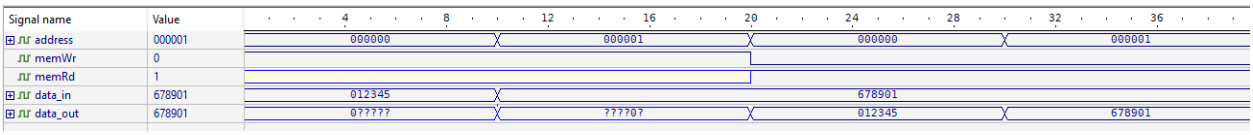

Figure 20: Data Memory Waveform

memWr = 1, memRd = 0 ➔ 012345 was stored into 000000 and 678901 was stored into 000001.

memWr = 0, memRd = 1 ➔ 012345 was read from 000000 and 678901 was read from 000001.

3.10. Machine

In machine module, all modules were connected to each other by calling other modules inside it and taking the output of each one using wires as inputs to others.

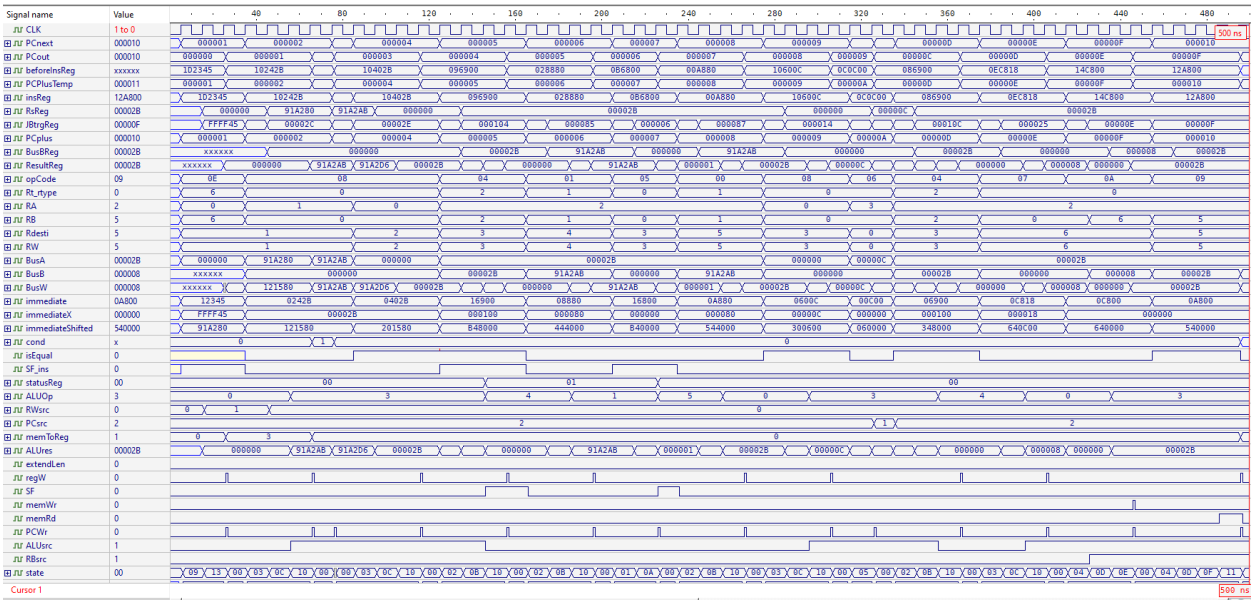

Figure 21: Machine Waveform

instructions[0] = {2'b00, LUI, 17'h12345}; // LUI 17'h12345

Figure 22: LUI Implementation

instructions[1] = {2'b00, ADDI,1'b0, R1, R1,10'b0000101011}; // ADDI R1,R1, 2B

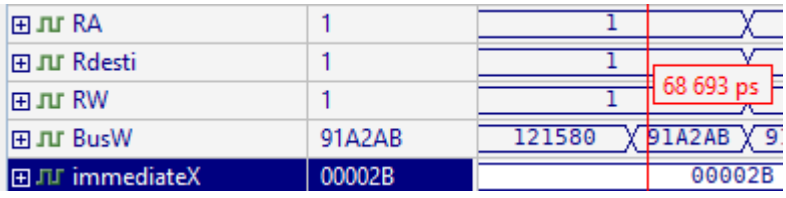

Figure 23: ADDI Implementation

Conclusion

In conclusion, this project successfully achieved its objective of designing and testing a Multi-Cycle RISC processor using Verilog. The processor has a 24-bit word size and supports conditional execution of all instructions. It features eight general-purpose registers, a program counter, and a status register with a zero flag. The processor's ALU instructions can update flag bits, and there are three instruction types and five addressing modes. The data path is constructed with a five-stage pipeline and register between stages, while the control logic is designed using a state machine approach. The implementation of this Multi-Cycle Datapath and control logic provides a foundation for further testing and improvement of the RISC processor design. Overall, this project provides an essential demonstration of the principles and techniques involved in designing and testing a RISC processor using Verilog, which can be applied to more complex processor designs in the future.

Appendix

All the work is attached below:

https://drive.google.com/drive/folders/1esav_-K5RmpW2tZnDnR53-bqpfz-KXMC