Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS4370 - Computer Architecture

Second Semester 2022/2023

Project No. 2

Prepared by:

Osama Qutait – 1191072

Mahmoud Samara – 1191602

Qays Safa – 1190880

Instructor: Dr. Aziz Qaroush

Section: 2

Date: July-12-2023

Introduction:

This project aims to design and verify a simple RISC processor in Verilog, adhering to specific specifications and instruction formats. The processor is a multi-cycle design with five stages: fetch, decode, execute, memory access, and write back. The processor supports 32-bit instructions and consists of 32 general-purpose registers, a program counter (PC) register, a stack pointer (SP) register, and a control stack for saving return addresses.

The processor's instruction set architecture (ISA) includes four instruction types: R-type (Register Type), I-type (Immediate Type), J-type (Jump Type), and S-type (Shift Type). Each instruction type has a specific format and encoding. The R-type instructions perform operations between registers, the I-type instructions involve immediate values, the J-type instructions handle jump operations, and the S-type instructions perform shifts.

The project involves implementing a subset of the processor's ISA, which includes instructions such as AND, ADD, SUB, CMP, ANDI, ADDI, LW, SW, BEQ, J, and JAL. These instructions enable basic arithmetic and logical operations, memory accesses, branching, and jumps. The processor's ALU generates a "zero" signal indicating whether the result of the last ALU operation is zero.

The verification of the processor's design will involve testing it with various assembly programs to ensure correct functionality and adherence to the given specifications. The Verilog implementation will be simulated and validated against expected results. The successful completion of this project will provide a working implementation of a simple RISC processor, demonstrating its ability to execute a subset of instructions accurately.

Table of Contents

Table of Figures

1. Design and Implementation:

1.1. Processor Specifications:

1. The instruction size is 32 bits

2. 32 32-bit general-purpose registers: from R0 to R31.

3. A special purpose register for the program counter (PC)

4. It has a stack called control stack which saves the return addresses

5. Stack pointer (SP), another special purpose register to point to the top of the control stack. SP holds the address of the empty element on the top of the stack. For simplicity, you can assume a separate on-chip memory for the stack, and the initial value of SP is zero.

6. Four instruction types (R-type, I-type, J-type, and S-type).

7. The processor's ALU has an output signal called "zero" signal, which is asserted when the result of the last ALU operation is zero.

8. Separate data and instructions memories

1.2. Instruction Types and Formats

This ISA has four instruction formats, namely, R-type, I-type, J-type, and S-type. These four types have the following common fields:

a. 2-bit instruction type (00: R-Type, 01: J-Type, 10: I-type, 11: S-type)

b. 5-bit function, to determine the specific operation of the instruction.

c. Stop bit, which is the least significant bit of each instruction binary format, and it is used to mark the end of a function code block. In other words, if the value of this stop bit is "1", this means that this instruction is the last instruction of the function, and hence the execution control should return to the return address which is stored on the top of the control stack.

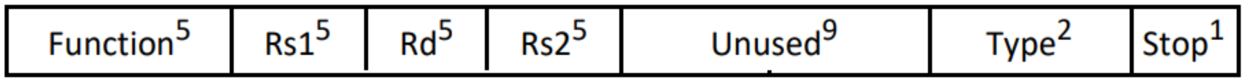
R-Type (Register Type) Format

• 5-bit Rs1: first source register

• 5-bit Rd: destination register •

5-bit Rs2: second source register

• 9-bit unused

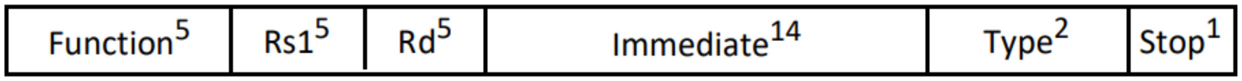


I-Type (Immediate Type) Format

• 5-bit Rs1: first source register

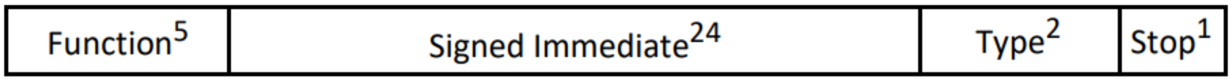
• 5-bit Rd: destination register

• 14-bit immediate: unsigned for logic instructions, and signed otherwise.



J-Type (Jump Type) Format

• 24-bit signed immediate: jump offset



S-Type (Shift Type) Format

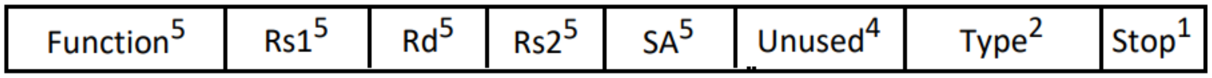
• 5-bit Rs1: first source register

• 5-bit Rd: destination register

• 5-bit Rs2: second source register. This register stores the shift amount in case the shift amount is variable and it is calculated at runtime

• 5-bit SA: the constant shift amount.

• 4-bit unused



1.3. Instructions' Encoding:

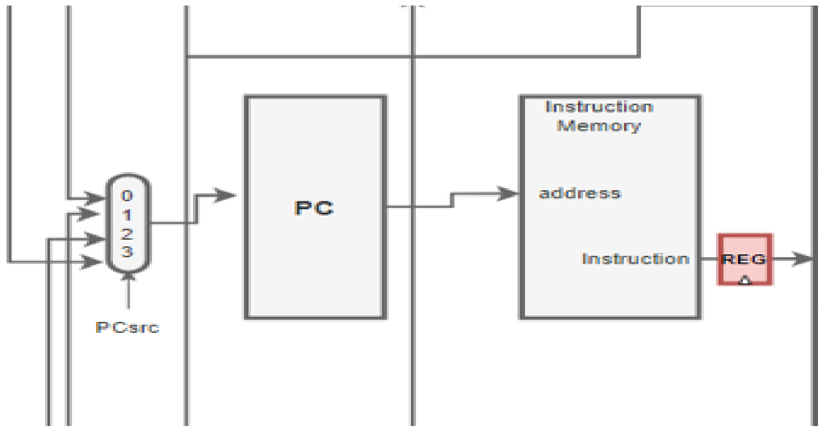
No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & Immediate^{14}	00000
6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

1.4. Component:

1.4.1. PC & Instruction Memory:

The Instruction Memory plays a crucial role in the functioning of the designed RISC processor. It serves as the storage unit for the instructions that need to be fetched and executed by the processor. The Instruction Memory is a separate memory component from the data memory and is responsible for holding the binary representations of the instructions. It is accessed during the fetch stage of the processor's pipeline, where the program counter (PC) is used to retrieve the instruction stored at the corresponding memory address. The fetched instruction is then passed to the decode stage for further processing. The Instruction Memory ensures the availability and retrieval of instructions in a sequential manner, enabling the processor to execute the program's instructions in the desired order. Its efficient operation and accessibility are vital for the overall performance and functionality of the RISC processor.

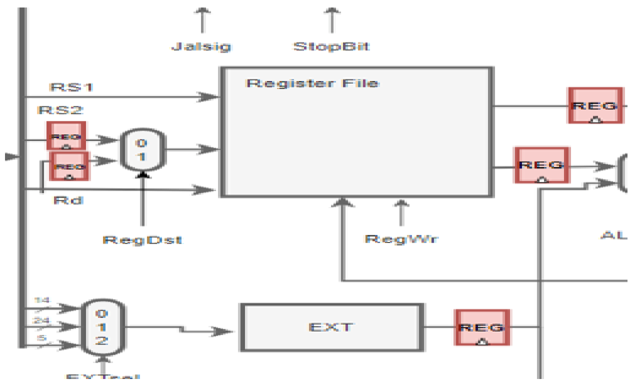
In our project the PC and Instruction memory represents the first stage of our system that is the instruction fetch. In the next PC address it can be previous PC + 4, or the jump address, or the Stack output or the BRANCH operation, then it takes that PC address and returns the instruction at that address. Then, the instruction memory output will be sent to buffer (register bit) to be sure we have multi cycle. Moreover, we have Pcsrc that is the selection for MUX4*1 that is used to choose what will be act as input for the PC. The following figure shows the Instruction fetch stage:



1.4.2. Register File

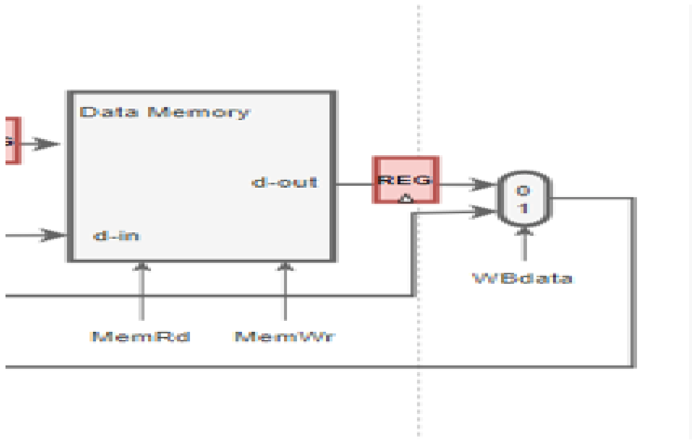
The Register File is a key component of the designed RISC processor that provides a set of general-purpose registers for data storage and manipulation. It serves as a high-speed storage unit for intermediate values and operands during the execution of instructions. The Register File consists of a fixed number of registers, typically 32 in this case, each capable of holding a 32-bit value. The registers are identified by their unique register numbers, ranging from R0 to R31. The Register File facilitates efficient data access and manipulation by allowing instructions to read data from specific registers and write data back to them. It plays a crucial role in register-to-register operations, where instructions involve data transfers and arithmetic or logical operations between registers. The Register File is accessed during the execute stage of the processor's pipeline, enabling the processor to perform operations on the data stored in the registers. It provides fast and low-latency access to data, contributing to the overall performance and functionality of the RISC processor. Proper management and synchronization of the Register File are necessary to ensure correct and efficient data handling during program execution.

In our project the register file was used to read values from the general-purpose registers. Also, there is mux 2*1 to choose between RS2 and Rd to know what it the input of RB depend on what we need. Additionally, we have an immediate (which can be 14-bits in I-Type or 24-bits in J-type, or 5- bits in S-Type) in order to extend it to 32-bits (note that in the S-Type the shift amount will also be used as a 5-bit immediate). The following figure shows what we described:



1.4.3. Data Memory

The Data Memory is a critical component of the designed RISC processor that is responsible for storing and retrieving data during program execution. It serves as a separate memory unit from the instruction memory and is used to hold variables, intermediate results, and other data required by the program. The Data Memory is accessed during the memory access stage of the processor's pipeline, where instructions such as load (LW) and store (SW) interact with the data memory. The memory access is typically performed by providing an address from a register or an immediate value, and the corresponding data is read from or written to the memory location. The Data Memory allows the processor to store and manipulate data efficiently, providing a vital resource for data processing and storage needs of the program being executed. Proper management and utilization of the Data Memory are essential for ensuring correct and efficient operation of the RISC processor. For the Data memory as we will see in the following figure we have memory read that is 1 when we use LW, memory write is 1 in SW, also the Wbdata take its output from the mux2*1 that contain either ALU_out or Data memory_out.



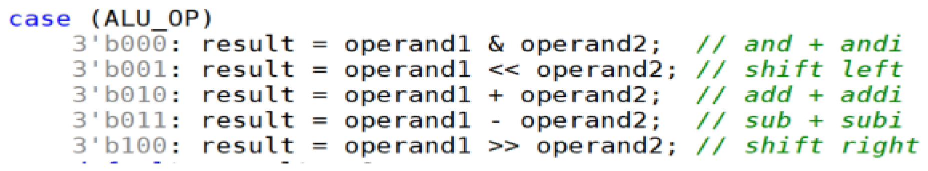
1.4.4. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a fundamental component of the designed RISC processor that performs arithmetic and logical operations on data. It is responsible for executing operations such as addition, subtraction, bitwise logical operations (AND, OR, XOR), and comparisons. The ALU takes inputs from the registers or immediate values, performs the specified operation based on the instruction, and produces the corresponding result. The ALU also generates control signals, such as the zero signal, indicating whether the result of the previous operation is zero. It plays a crucial role in executing arithmetic and logical instructions within the processor. The ALU's design and implementation determine the supported operations, bit widths, and performance characteristics of the processor. It is a key component in achieving the desired functionality and computational capabilities of the RISC processor. Efficient operation and accurate computation by the ALU contribute to the overall performance and reliability of the processor during program execution.

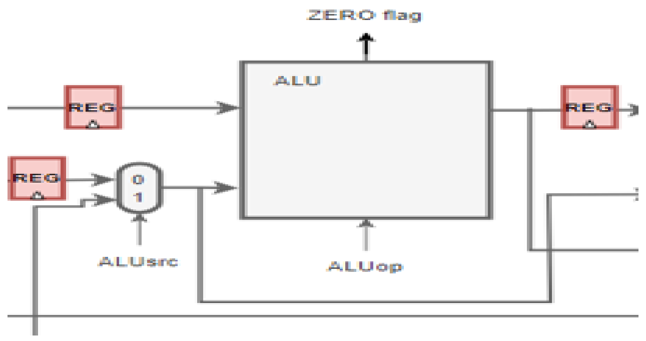
In our project The first input is the ALU Src signal which indicates whether the second input (the first is always BusA) is the BusB or the extended immediate. If the value of the ALU Src is equal to 1 then the second input will be the extended immediate. Otherwise, the second input will be the BusB. The BusA, BusB and the extended immediate are a 32-bit registers that the different arithmetic operations will be performed using them. In order to determine which arithmetic operation to be performed the FUNTION input will be used.

- If binary Value of it is 000 and ALU src value is 0 then it will perform bitwise and operation between BusA and BusB. But, If ALU src value is 1 then it will perform bitwise and operation between BusA and immediate.
- If binary Value of it is 001 and ALU src value is 0 then it will perform shift-left on BusA by the value of BusB. But, If ALU src value is 1 then it will perform shift-left on BusA by the value of immediate.

- If binary Value of it is 010 and ALU src value is 0 then it will perform add operation between BusA and BusB. But, If ALU src value is 1 then it will perform add operation between BusA and immediate.
- If binary Value of it is 011 and ALU src value is 0 then it will perform sub operation between BusA and BusB.
- If binary Value of it is 100 and ALU src value is 0 then it will perform shift-right on BusA by the value of BusB. But, If ALU src value is 1 then it will perform shift-right on BusA by the value of immediate.



```
case (ALU_OP)
    3'b000: result = operand1 & operand2;  // and + andi
    3'b001: result = operand1 << operand2; // shift left
    3'b010: result = operand1 + operand2;  // add + addi
    3'b011: result = operand1 - operand2;  // sub + subi
    3'b100: result = operand1 >> operand2; // shift right
```



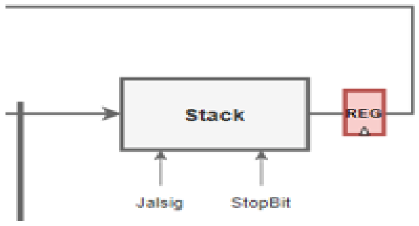
1.4.5. Extender

The Extender is an essential component of the designed RISC processor responsible for extending immediate values to match the register width. Immediate values, which are constants or immediate operands within instruction formats, are often used for immediate calculations or addressing modes. The Extender ensures that these values are properly sign-extended, preserving the sign bit and maintaining their correct interpretation. During the decode stage of the processor's pipeline, the Extender takes the immediate value from the instruction and extends it to the appropriate width, allowing for accurate arithmetic, logical, and memory operations. Its accurate implementation and adherence to sign extension rules are crucial for preserving data integrity and ensuring the proper execution of programs.

1.4.6. Stack

In computer science, a stack is a data structure that follows the Last-In-First-Out (LIFO) principle. It is like a stack of plates, where the last plate placed on top is the first one to be removed. The stack allows operations to add or remove items only from the top. When an item is added, it is pushed onto the stack, and when an item is removed, it is popped from the top. The stack is commonly used to manage function calls and returns in programming languages, storing return addresses and local variables. It provides a way to keep track of the program's execution flow and supports nested function calls by maintaining a record of previous execution points. Additionally, the stack is utilized in various algorithms and data processing tasks for temporary storage or tracking system states.

In our project the stack is used to store the address in the JAL, and at the end of the function it gives me this address.



1.4.7. Control Unit

The Control Unit is a critical component of the designed RISC processor that orchestrates the execution of instructions and manages the flow of data within the processor. It is responsible for interpreting and decoding instructions, generating control signals, and coordinating the activities of other components, such as the ALU, memory units, and registers. The Control Unit takes the opcode and other relevant fields from the instruction and determines the specific operation to be performed. It generates control signals that enable the appropriate components to carry out the instruction, including activating the necessary registers, selecting the ALU operation, initiating memory accesses, and controlling the flow of data between components. The Control Unit plays a pivotal role in ensuring the correct execution and synchronization of instructions, managing the timing and sequencing of operations, and controlling the processor's overall behavior. It serves as the brain of the processor, coordinating the activities of various units to execute instructions accurately and efficiently.

1.5. Datapath and Control Signals

1.5.1. Datapath:

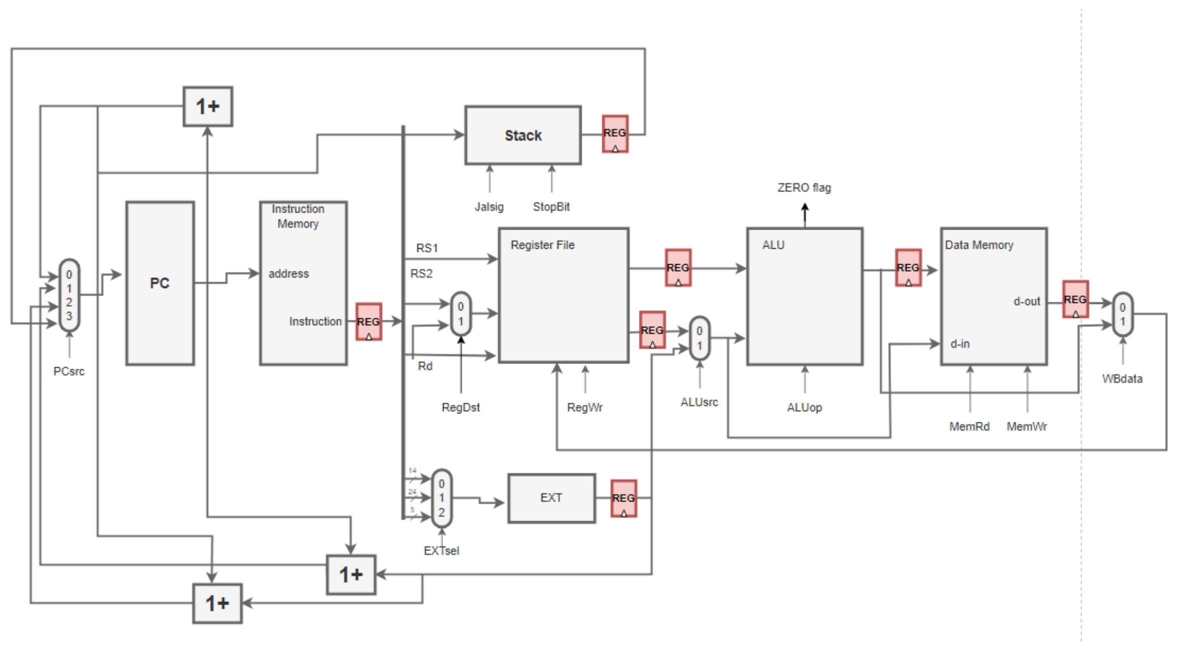


Figure 1-1 Project Datapath

The previous figure shows the full Datapath for our project after we connected all components and wires together. Therefore, the design of the data path and control signals for a Multi-Cycle Processor with Memory had already been completed.

In the instruction fetch stage, the program counter, and an adder by 1 were added to instruction fetch stage. The result was transmitted as input to instruction memory and the instruction memory output is added to buffer, to pass the buffer value to the decode. After that, In the instruction decode stage, the register file was used to read values from the general-purpose registers it has 5 inputs (RA, RB, RD, RegWr, BusW) and two outputs (BusA and BusB), but for RB input can enter to it Rs2 or Rd depend on what we need so we added mux. Also, an extender was used to extend the immediate value to 32 bits, and choose between immediate 14 or 24 or 5 depend on what we need.

Next, 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, shift left, shift right. 8bit status. Also, in this stage we have the Stack is used to store the address in the JAL, and at the end of the function it gives me this address. 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.

1.5.2. Control Signals:

1. JalSig: only occurs during Jal operations, saving the PC + 4 address to the stack.
2. PcSrc: for this control signal it is the selection for the mux that is for the Program Counter. It has 4 cases so two-bit signal, If PcSrc is 0 then PC + 4, if PcSrc is 2 then it is for BEQ, if PcSrc is 3 then it is for Stack output.
3. RegDest: this flag determines if the second or third register address from the instruction is the second source register in the register file
4. AluSrc: The extended immediate or the second source register data will be the second input to the ALU depending on the value of this flag.
5. ALU Op: The ALU Op, which can be (AND-000, ADD-010, SUB-011, Shift-left-001, Shift-right-100), is determined by this three-bit flag.
6. MemRed: This flag is raised whenever memory data has to be read.
7. MemWr: This flag enables the writing of data to the memory.
8. WbData: This flag decides if the data that will be written on the destination register will be the ALU output or the memory output.
9. RegWr: This flag causes the bus W data to be able to be written to the target register.

Each instruction's control signal values are included in the table 1, where some of them will have the value (x) because it isn't particularly important.

instruction	**JalSig**	**ExtSel2**	**ExtSel1**	**RegDest**	**ALUSrc**	**ALUOp**	**MemRd**	**MemWr**	**WbData**	**RegWr**
AND	0	X	X	0	0	000	0	0	1	1
ADD	0	X	X	0	0	010	0	0	1	1
SUB	0	X	X	0	0	011	0	0	1	1
CMP	0	X	X	0	0	011	0	0	X	0
ANDI	0	0	0	0	1	000	0	0	1	1
ADDI	0	0	0	0	1	010	0	0	1	1
LW	0	0	1	0	1	010	1	0	0	1

SW	0	0	1	0	1	010	0	1	0	0
BEQ	0	X	X	1	0	011	0	0	X	0
J	0	0	1	X	X	X	0	0	X	0
JAL	1	0	1	X	X	X	0	0	X	0
SLL	0	1	0	0	1	001	0	0	1	1
SLR	0	1	0	0	1	100	0	0	1	1
SLLV	0	X	X	0	0	001	0	0	1	1
SLRV	0	X	X	0	0	100	0	0	1	1

Expressions for Control Bits:

JalSig = JAL

Extsel2= SLL || SLR

ExtSel1= LW || SW || J || JAL

RedDest BEQ

ALUsrc = ANDI || ADDI || LW || SW ||SLL || SLR

MemRd = LW

MemWr = SW

WbData = AND || ADD || SUB || ANDI || ADDI || SLL || SLR || SLLV || SLRV

RegWr = AND || ADD || SUB || ANDI || ADDI || LW || SLL || SLR || SLLV || SLRV

ALUOP[2] = SLR || SLRV

ALUOP[1] = ADD || SUB || CMP || ADDI || LW || SW || BEQ

ALUOP[0] = SUB || CMP || BEQ || SLL || SLLV

1.6. Building the State Machine Diagram

A methodical technique had been used to construct the state machine diagram for the various stages of the multi-cycle processor. First, the order of operations that had to be carried out during each step and the circumstances in which they were carried out were examined. These details led to the identification of the many states needed in the state machine diagram. The transitions between the states, which were triggered by the control signals produced by the CPU, were defined second. For instance, the "state number" value, which signifies that the instruction has been fetched from memory and is ready for decoding, caused the transition from the instruction fetch stage to the instruction decode stage. The operations that would take place during each state, such as incrementing the PC or writing to the register file, were also outlined. Lastly, the accuracy of the state machine diagram was confirmed by simulating processor activity and contrasting the outcomes with the predicted behavior.

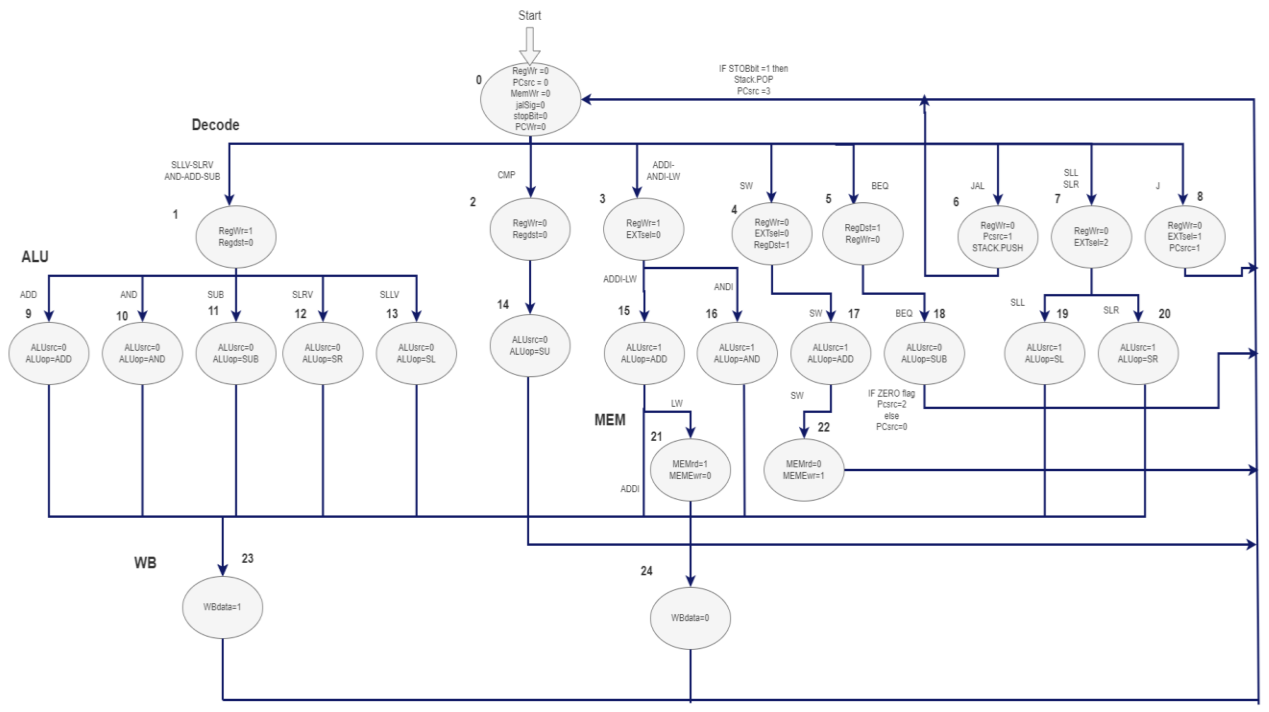
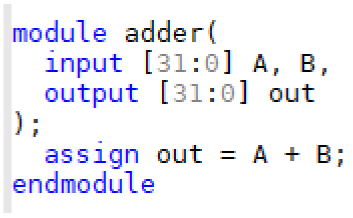


Figure 1-2 State Machine Diagram

2. Testing the Modules:

Testbenches allowed designers to test modules in a controlled environment before integrating them into a larger system, which was essential for confirming that a module's functioning was proper. It was necessary to comprehend the module's requirements, input and output signals, and intended operating conditions in order to build successful testbenches. Designers could make sure that their modules worked as planned and could find and fix mistakes early in the design process by simulating a range of scenarios and stimuli. Testbenches were therefore a vital tool for confirming the accuracy and dependability of Verilog designs. For all modules, test environments were developed to cover a subset of potential scenarios.

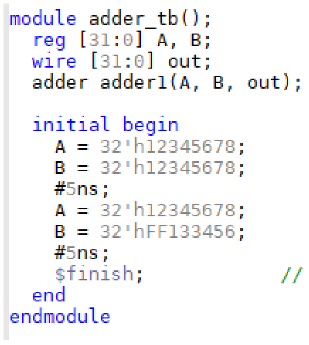
2.1. Adder:



```verilog
module adder(
  input [31:0] A, B,
  output [31:0] out
);
  assign out = A + B;
endmodule
```

Figure 2-1 Adder module code

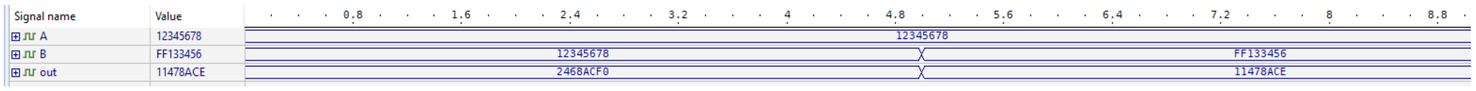
The Verilog module "adder" takes two 32-bit inputs, A and B, and produces a 32-bit output called out. The assign statement performs the addition operation by adding A and B and assigns the result to out.



```verilog
module adder_tb();
  reg [31:0] A, B;
  wire [31:0] out;
  adder adder1(A, B, out);

  initial begin
    A = 32'h12345678;
    B = 32'h12345678;
    #5ns;
    A = 32'h12345678;
    B = 32'hFF133456;
    #5ns;
    $finish;            //
  end
endmodule
```

Figure 2-2 Adder module testbench

The Verilog testbench module "adder_tb" is designed to verify the functionality of the "adder" module. It initializes the input signals A and B with specific values and applies test stimuli to simulate the addition



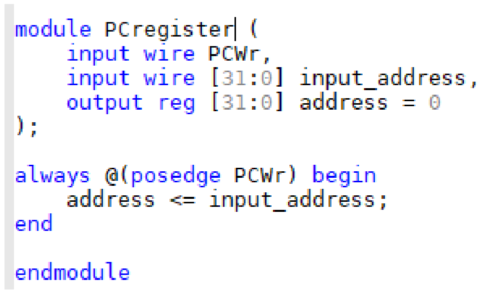
Signal name	Value		
A	12345678	12345678	
B	FF133456	12345678	FF133456
out	11478ACE	2468ACF0	11478ACE

Figure 2-3 Adder result

operation. The simulation progresses in discrete time steps defined by the #5ns; delay. By monitoring the output signal out, the behavior of the "adder" module can be observed and validated.

The output is as expected as 12345678 + 12345678 = 2468ACF0 and 12345678 + FF133456 = 11478ACE.

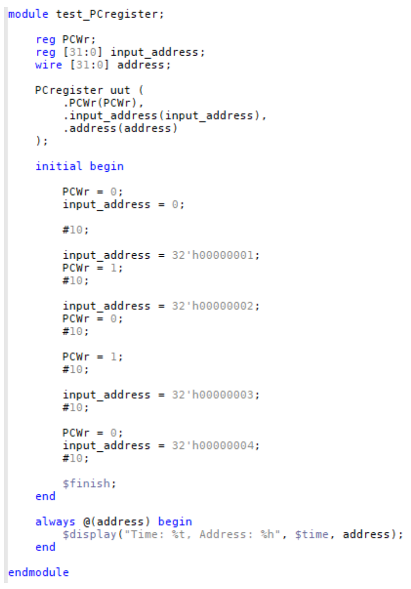
2.2. PC Register:



```
module PCregister (
    input wire PCWr,
    input wire [31:0] input_address,
    output reg [31:0] address = 0
);

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

endmodule
```

Figure 2-4 PCregister code

The Verilog module "PCregister" represents a program counter register. It takes an input signal PCWr to enable writing and an input wire input_address for the new PC address. The output register address holds the current PC address and is initialized to 0. Whenever the PCWr signal experiences a positive edge, the input_address is assigned to the address register, allowing the module to update the current PC address. This module facilitates the storage and retrieval of the program counter value in a synchronous manner.



```
module test_PCregister;

    reg PCWr;
    reg [31:0] input_address;
    wire [31:0] address;

    PCregister uut (
        .PCWr(PCWr),
        .input_address(input_address),
        .address(address)
    );

    initial begin

        PCWr = 0;
        input_address = 0;

        #10;

        input_address = 32'h00000001;
        PCWr = 1;
        #10;

        input_address = 32'h00000002;
        PCWr = 0;
        #10;

        PCWr = 1;
        #10;

        input_address = 32'h00000003;
        #10;

        PCWr = 0;
        input_address = 32'h00000004;
        #10;

        $finish;
    end

    always @(address) begin
        $display("Time: %t, Address: %h", $time, address);
    end

endmodule
```

The Verilog testbench module "test_PCregister" is designed to verify the functionality of the "PCregister" module. It instantiates the "PCregister" module and applies test stimuli to simulate the behavior of the program counter register. The testbench sets values for the PC write enable signal and the new PC address at different time points and observes the current PC address through the "address" signal. The simulation progresses with delays defined by #10;, and the time and address values are displayed using the $display system task. This testbench allows for testing and verification of the program counter register module by monitoring its behavior and output under various conditions and time steps.

Figure 2-5 PCregister testbench

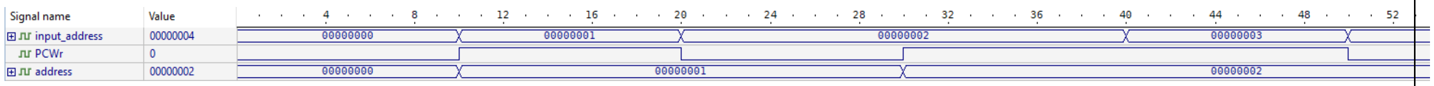
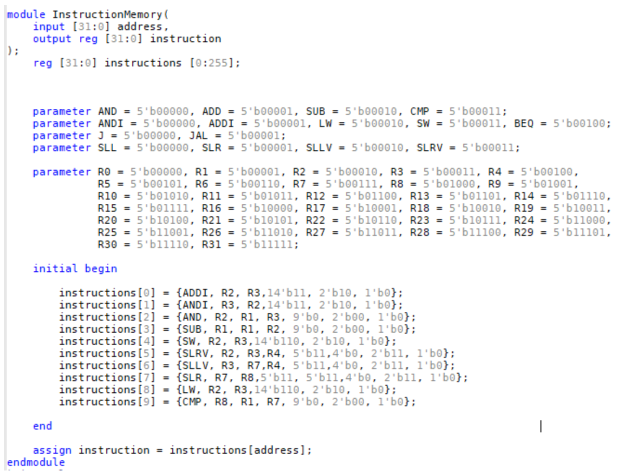


Figure 2-6 PCregister result

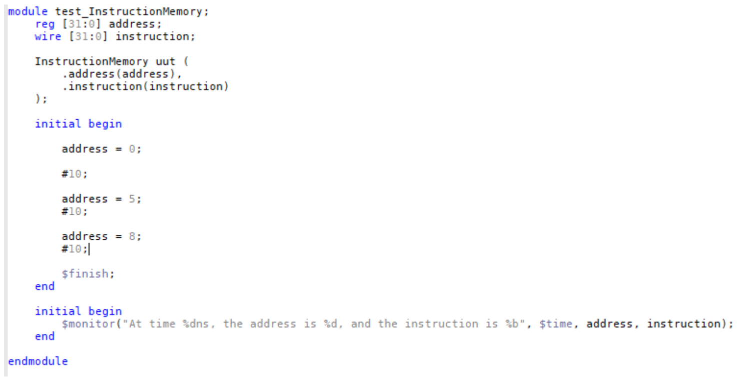
2.3. Instruction Memory:



```
module InstructionMemory(
    input [31:0] address,
    output reg [31:0] instruction
);
    reg [31:0] instructions [0:255];



    parameter AND = 5'b00000, ADD = 5'b00001, SUB = 5'b00010, CMP = 5'b00011;
    parameter ANDI = 5'b00000, ADDI = 5'b00001, LW = 5'b00010, SW = 5'b00011, BEQ = 5'b00100;
    parameter J = 5'b00000, JAL = 5'b00001;
    parameter SLL = 5'b00000, SLR = 5'b00001, SLLV = 5'b00010, SLRV = 5'b00011;

    parameter R0 = 5'b00000, R1 = 5'b00001, R2 = 5'b00010, R3 = 5'b00011, R4 = 5'b00100,
              R5 = 5'b00101, R6 = 5'b00110, R7 = 5'b00111, R8 = 5'b01000, R9 = 5'b01001,
              R10 = 5'b01010, R11 = 5'b01011, R12 = 5'b01100, R13 = 5'b01101, R14 = 5'b01110,
              R15 = 5'b01111, R16 = 5'b10000, R17 = 5'b10001, R18 = 5'b10010, R19 = 5'b10011,
              R20 = 5'b10100, R21 = 5'b10101, R22 = 5'b10110, R23 = 5'b10111, R24 = 5'b11000,
              R25 = 5'b11001, R26 = 5'b11010, R27 = 5'b11011, R28 = 5'b11100, R29 = 5'b11101,
              R30 = 5'b11110, R31 = 5'b11111;

    initial begin

        instructions[0] = {ADDI, R2, R3,14'b11, 2'b10, 1'b0};
        instructions[1] = {ANDI, R3, R2,14'b11, 2'b10, 1'b0};
        instructions[2] = {AND, R2, R1, R3, 9'b0, 2'b00, 1'b0};
        instructions[3] = {SUB, R1, R1, R2, 9'b0, 2'b00, 1'b0};
        instructions[4] = {SW, R2, R3,14'b110, 2'b10, 1'b0};
        instructions[5] = {SLRV, R2, R3,R4, 5'b11,4'b0, 2'b11, 1'b0};
        instructions[6] = {SLLV, R3, R7,R4, 5'b11,4'b0, 2'b11, 1'b0};
        instructions[7] = {SLR, R7, R8,5'b11, 5'b11,4'b0, 2'b11, 1'b0};
        instructions[8] = {LW, R2, R3,14'b110, 2'b10, 1'b0};
        instructions[9] = {CMP, R8, R1, R7, 9'b0, 2'b00, 1'b0};

    end

    assign instruction = instructions[address];
endmodule
```

Figure 2-7 InstructionMemory code

The Verilog module "InstructionMemory" represents an instruction memory that stores and retrieves instructions based on the provided address. It contains a register array called "instructions" that holds 256 32-bit instructions. Each instruction is initialized in the "initial" block using predefined opcodes, register identifiers, and other relevant values. The module takes an input address and assigns the corresponding instruction from the memory array to the output register "instruction" using the "assign" statement. This module is designed to simulate the behavior of an instruction memory component in a processor design, allowing for the retrieval of instructions based on their memory addresses.



```
module test_InstructionMemory;
    reg [31:0] address;
    wire [31:0] instruction;

    InstructionMemory uut (
        .address(address),
        .instruction(instruction)
    );

    initial begin

        address = 0;

        #10;

        address = 5;
        #10;

        address = 8;
        #10;

        $finish;
    end

    initial begin
        $monitor("At time %dns, the address is %d, and the instruction is %b", $time, address, instruction);
    end

endmodule
```

Figure 2-8 InstructionMemory testbench

The Verilog testbench module "test_InstructionMemory" is designed to verify the functionality of the "InstructionMemory" module. It instantiates the "InstructionMemory" module and applies test stimuli to simulate the behavior of the instruction memory. The testbench sets values for the address input at different time points and monitors the address and instruction values during the simulation using the $monitor system task. The simulation progresses with delays defined by #10;, and the time, address, and instruction values are displayed at each simulation cycle. This testbench allows for testing and validation of the instruction memory module by observing its behavior and output under various address inputs and time steps.

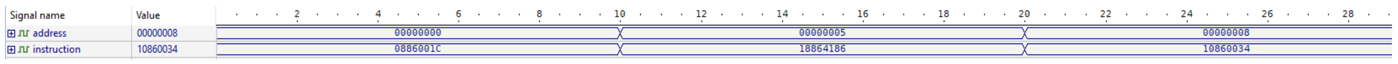
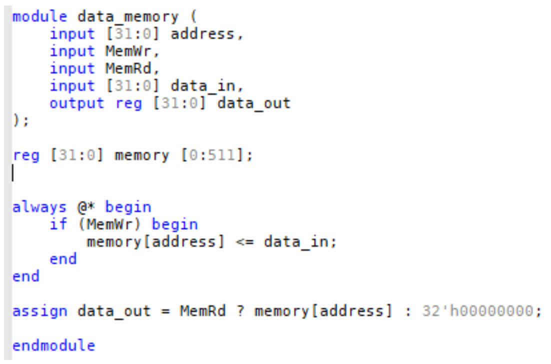


Figure 2-9 InstructionMemory result

2.4. Data memory:

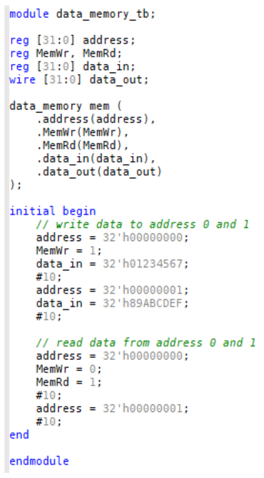


```
module data_memory (
    input [31:0] address,
    input MemWr,
    input MemRd,
    input [31:0] data_in,
    output reg [31:0] data_out
);

reg [31:0] memory [0:511];

always @* begin
    if (MemWr) begin
        memory[address] <= data_in;
    end
end

assign data_out = MemRd ? memory[address] : 32'h00000000;

endmodule
```

Figure 2-10 data_memory code

The Verilog module "data_memory" represents a simple data memory module. It has inputs for address, MemWr (memory write enable), MemRd (memory read enable), and data_in (input data to be written). The module also has an output register data_out, which holds the value read from the memory. Inside the module, there is a memory array called "memory" that can store 512 32-bit values. When a memory write operation is requested (MemWr is true), the input data_in is written to the memory location specified by the address. On a memory read operation (MemRd is true), the value stored at the specified address is assigned to data_out. If no read operation is requested (MemRd is false), data_out is assigned zero. This module provides a basic implementation of a data memory that supports read and write operations.

The Verilog testbench module "data_memory_tb" is designed to verify the functionality of the "data_memory" module. It instantiates the "data_memory" module and applies test stimuli to simulate write and read operations. The testbench sets values for the address, MemWr, MemRd, and data_in signals at different time points to simulate write



```
module data_memory_tb;

reg [31:0] address;
reg MemWr, MemRd;
reg [31:0] data_in;
wire [31:0] 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 = 32'h00000000;
    MemWr = 1;
    data_in = 32'h01234567;
    #10;
    address = 32'h00000001;
    data_in = 32'h89ABCDEF;
    #10;

    // read data from address 0 and 1
    address = 32'h00000000;
    MemWr = 0;
    MemRd = 1;
    #10;
    address = 32'h00000001;
    #10;
end

endmodule
```

Figure 2-11 data_memory testbench

operations. It also performs read operations by setting MemWr to 0 and MemRd to 1. The simulation progresses with delays defined by #10;, allowing time for the memory operations to take place. The behavior and output of the "data_memory" module, including data_out, can be observed and validated during the simulation. This testbench enables testing and verification of the data memory module by simulating various write and read scenarios.

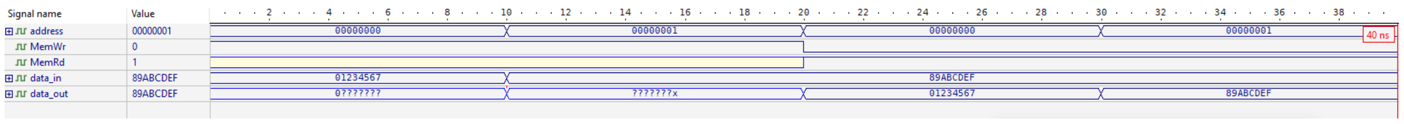
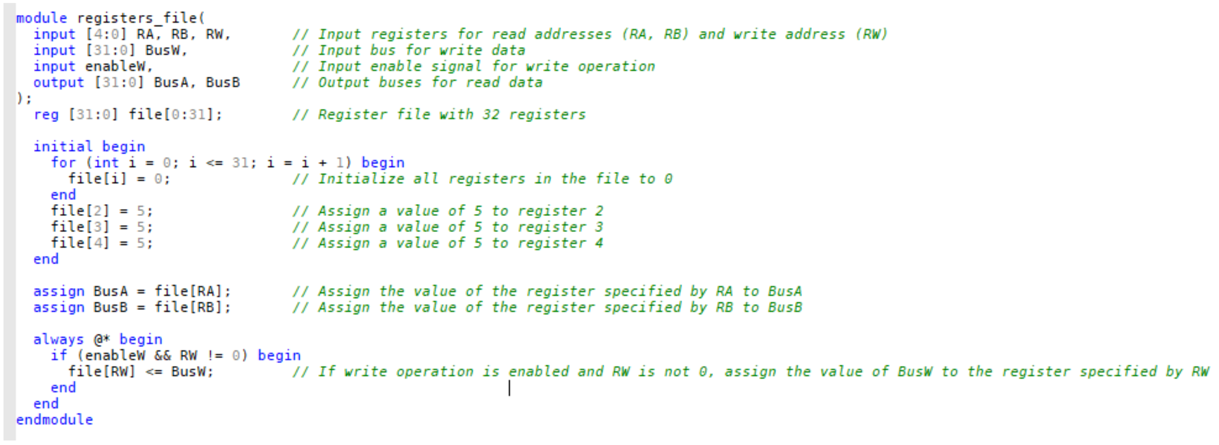


Figure 2-12 data_memory result

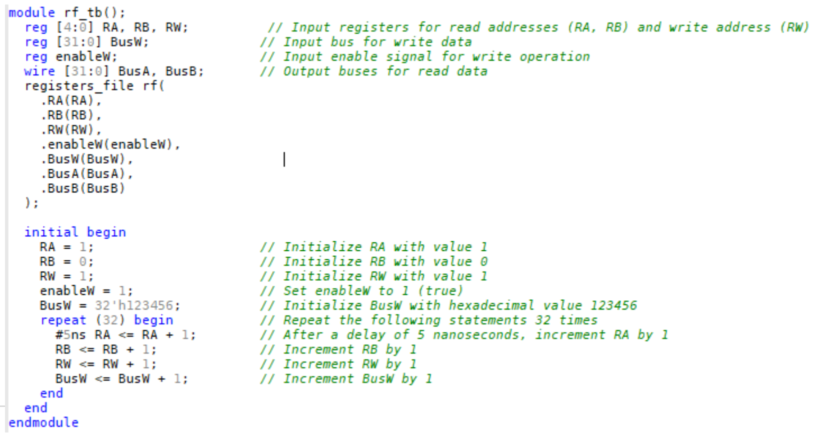
2.5. Registers file:



```
module registers_file(
  input [4:0] RA, RB, RW,       // Input registers for read addresses (RA, RB) and write address (RW)
  input [31:0] BusW,            // Input bus for write data
  input enableW,                // Input enable signal for write operation
  output [31:0] BusA, BusB      // Output buses for read data
);
  reg [31:0] file[0:31];        // Register file with 32 registers

  initial begin
    for (int i = 0; i <= 31; i = i + 1) begin
      file[i] = 0;              // Initialize all registers in the file to 0
    end
    file[2] = 5;                // Assign a value of 5 to register 2
    file[3] = 5;                // Assign a value of 5 to register 3
    file[4] = 5;                // Assign a value of 5 to register 4
  end

  assign BusA = file[RA];       // Assign the value of the register specified by RA to BusA
  assign BusB = file[RB];       // Assign the value of the register specified by RB to BusB

  always @* begin
    if (enableW && RW != 0) begin
      file[RW] <= BusW;         // If write operation is enabled and RW is not 0, assign the value of BusW to the register specified by RW
    end
  end
endmodule
```

Figure 2-13 registers_file code

The Verilog module "registers_file" represents a register file that can be read from and written to. It consists of an array of 32 registers, initialized with zeros except for registers 2, 3, and 4 which are set to 5. The module takes input read addresses (RA and RB) and a write address (RW), as well as an input write data (BusW) and an enable signal (enableW). It provides output buses (BusA and BusB) for reading data from the register file. The values of the registers specified by RA and RB are assigned to BusA and BusB, respectively. When the write operation is enabled and the write address is non-zero, the value from BusW is written to the register specified by RW. This module enables the manipulation and retrieval of data stored in registers within a register file.



```
module rf_tb();
  reg [4:0] RA, RB, RW;          // Input registers for read addresses (RA, RB) and write address (RW)
  reg [31:0] BusW;              // Input bus for write data
  reg enableW;                  // Input enable signal for write operation
  wire [31:0] BusA, BusB;       // Output buses for read data
  registers_file rf(
    .RA(RA),
    .RB(RB),
    .RW(RW),
    .enableW(enableW),
    .BusW(BusW),
    .BusA(BusA),
    .BusB(BusB)
  );

  initial begin
    RA = 1;                     // Initialize RA with value 1
    RB = 0;                     // Initialize RB with value 0
    RW = 1;                     // Initialize RW with value 1
    enableW = 1;                // Set enableW to 1 (true)
    BusW = 32'h123456;          // Initialize BusW with hexadecimal value 123456
    repeat (32) begin           // Repeat the following statements 32 times
      #5ns RA <= RA + 1;        // After a delay of 5 nanoseconds, increment RA by 1
      RB <= RB + 1;             // Increment RB by 1
      RW <= RW + 1;             // Increment RW by 1
      BusW <= BusW + 1;         // Increment BusW by 1
    end
  end
endmodule
```

Figure 2-14 registers_file testbench

The Verilog testbench module "rf_tb" is designed to verify the functionality of the "registers_file" module. It instantiates the "registers_file" module and applies test stimuli to simulate the behavior of the register file. The testbench initializes the read addresses (RA and RB), the write address (RW), the write enable signal (enableW), and the write data (BusW) to specific values. It then repeats a set of instructions 32 times, incrementing the values of RA, RB, RW, and BusW in each iteration. The simulation progresses with delays of 5 nanoseconds between each iteration. This testbench allows for testing and validation of the register file module by observing the behavior of the registers and outputs over time and ensuring they exhibit the expected behavior.

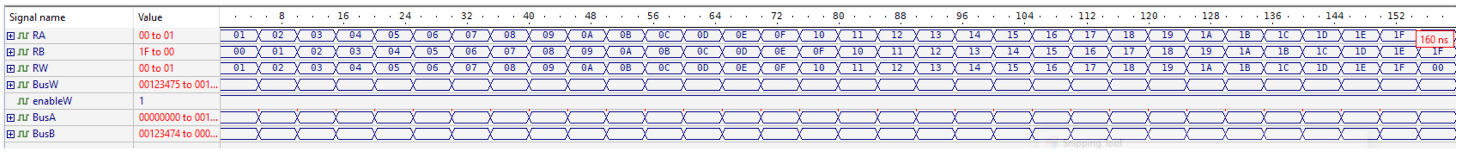
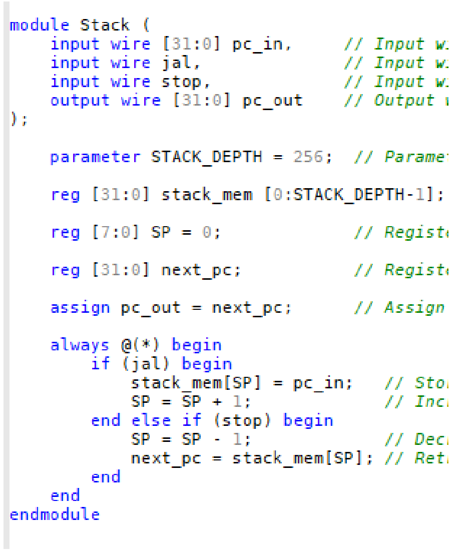


Figure 2-15 rf_tb result

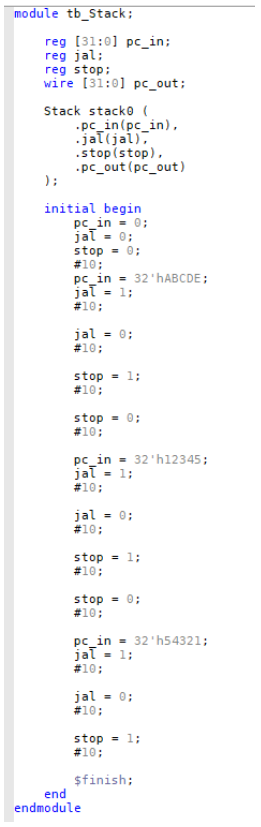
2.6. Stack:



```
module Stack (
    input wire [31:0] pc_in,     // Input w
    input wire jal,              // Input w
    input wire stop,             // Input w
    output wire [31:0] pc_out    // Output
);

    parameter STACK_DEPTH = 256;  // Parame

    reg [31:0] stack_mem [0:STACK_DEPTH-1];

    reg [7:0] SP = 0;             // Regist

    reg [31:0] next_pc;           // Regist

    assign pc_out = next_pc;      // Assign

    always @(*) begin
        if (jal) begin
            stack_mem[SP] = pc_in;   // Sto
            SP = SP + 1;             // Inc
        end else if (stop) begin
            SP = SP - 1;             // Dec
            next_pc = stack_mem[SP]; // Ret
        end
    end
endmodule
```

Figure 2-16 Stack code

The Verilog module "Stack" represents a stack implementation that stores program counter values. It has inputs for pc_in, indicating a program counter value to be stored in the stack, jal for a jump and store operation, and stop for a return and pop operation. The output pc_out retrieves the program counter value from the stack. The stack is implemented as a register array named "stack_mem" with a specified depth. The stack pointer (SP) keeps track of the current position in the stack. When jal is true, pc_in is stored in the stack at the current SP position, and SP is incremented. When stop is true, SP is decremented, and the value from the stack at the updated SP position is retrieved and assigned to pc_out. This module provides a simple stack functionality, enabling storing and retrieving program counter values in a last-in-first-out manner.



```
module tb_Stack;

    reg [31:0] pc_in;
    reg jal;
    reg stop;
    wire [31:0] pc_out;

    Stack stack0 (
        .pc_in(pc_in),
        .jal(jal),
        .stop(stop),
        .pc_out(pc_out)
    );

    initial begin
        pc_in = 0;
        jal = 0;
        stop = 0;
        #10;
        pc_in = 32'hABCDE;
        jal = 1;
        #10;

        jal = 0;
        #10;

        stop = 1;
        #10;

        stop = 0;
        #10;

        pc_in = 32'h12345;
        jal = 1;
        #10;

        jal = 0;
        #10;

        stop = 1;
        #10;

        stop = 0;
        #10;

        pc_in = 32'h54321;
        jal = 1;
        #10;

        jal = 0;
        #10;

        stop = 1;
        #10;

        $finish;
    end
endmodule
```

The Verilog testbench module "tb_Stack" is designed to verify the functionality of the "Stack" module. It instantiates the "Stack" module and applies test stimuli to simulate the behavior of the stack. The testbench sets values for the pc_in, jal, and stop signals at different time points to simulate storing and retrieving program counter values from the stack. The simulation progresses with delays defined by #10;, allowing time for the stack operations to take place. The behavior and output of the "Stack" module, including the pc_out signal, can be observed and validated during the simulation. This testbench enables testing and verification of the stack module by simulating various push and pop operations and ensuring they exhibit the expected behavior.

Figure 2-17 Stack testbench

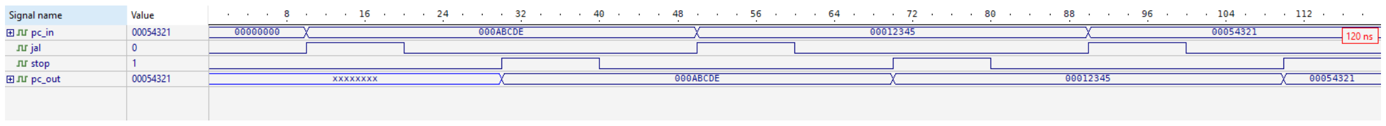


Figure 2-18 Stack result

Note that the testing results for each instruction and all instructions together with code sequences are attached in separate pdf file.

3. Conclusion

In the conclusion, we can say that all our work depends on the tests we did is accepted with the theoretical theorems for each instruction. Additionally, we can say that the Multi-Cycle Processor architecture is a strong and adaptable processor design that can divide complex instructions into numerous cycles for more effective and flexible execution. This enables faster instruction execution and more effective use of hardware resources. Moreover, the entire data path construction was shown according to the required instructions. Where the stages were built with the control signals required to control each of them, and we made a state diagram to see the way for each instruction and it helped us to write the control unit code. Finally, we can say that at the end of this project we learnt a more in-depth understanding about how exactly the MIPS CPU works and we learnt the best way to connect the modules using code and testing them.