Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Computer Architecture – ENCS4370

Final Project Report

Prepared by :

Hajar Salah 1191482

Jana Abu Nasser 1201110

Sary Hammad 1192698

Instructor : Dr. Aziz Qaroush

Section : 1 & 2

Date : 23-6-2023

Introduction

The objective of our project is to design and verify a simple RISC processor using Verilog. A RISC processor is a type of microprocessor architecture that emphasizes simplicity and efficiency by reducing the number of instructions and focusing on a smaller set of basic operations. Our processor follows specific specifications and instruction formats to ensure its functionality and compatibility with software.

The processor's instruction size is 32 bits, allowing for a wide range of operations and data manipulation. It includes 32 general-purpose registers, labeled from R0 to R31, which can be used for storing data during program execution. Additionally, there is a special purpose register called the program counter (PC), which keeps track of the address of the current instruction being executed.

The processor includes a control stack to store return addresses during program execution, aiding in control flow management for function calls and returns. To support stack operations, a stack pointer (SP) is used to reference the top element. With an initial value of zero, the stack pointer ensures efficient control flow and smooth handling of function calls and returns.

Our processor supports four instruction types: R-type, I-type, J-type, and S-type. These instruction types allow for a diverse range of operations, including arithmetic and logical computations, branching, and memory access. The specific operation of each instruction is determined by a 5-bit function field within the instruction.

The processor's ALU (Arithmetic Logic Unit) performs arithmetic and logical operations. It includes an output signal called the "zero" signal, which indicates whether the result of the last ALU operation is zero. This signal is useful for conditional branching and comparisons within the processor.

To ensure efficient and organized execution, our processor includes separate memories for data and instructions. This separation allows for simultaneous access to data and instructions during program execution, enhancing overall performance.

The multi-cycle Datapath in this project consists of five stages, which are similar to the ones presented in the class lectures. The five stages are Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). The IF stage is responsible for fetching instructions from memory, while the ID stage decodes the instruction and reads the necessary registers. The EX stage performs arithmetic and logical operations, the MEM stage accesses memory if necessary, and the WB stage writes the result back to the register file.

In order to effectively manage the multi-cycle Datapath, our project involves designing a control unit. The purpose of this control unit is to generate precise control signals that govern the operation of the Datapath at each stage. These control signals are responsible for controlling various components such as multiplexers, the ALU and the memory interface.

This project provides hands-on experience in designing and testing a simple multi-cycle RISC processor, allowing us to apply theoretical knowledge in processor design, instruction set architecture, and control logic. It offers an opportunity to deepen understanding and practical skills in computer architecture concepts.

Contents

Design Specification 5

1. Motivation 5

2. Instruction Types and Formats 5

2.1 R-type – Register Type 5

2.2 I-type – Immediate Type 6

2.3 J-type – Jump Type 6

2.4 S-type – Shift Type 6

3. Instruction Set 6

Components 7

1. Memory 7

1.1 Instruction Memory 7

1.2 Data Memory 7

2. Register File 8

3. Arithmetic Logic Unit (ALU) 8

4. Stack 9

5. Write Back 9

6. Control Unit 10

6.1 Truth Table 10

6.2 Boolean Expression 11

6.3 Finite State Machine 12

Full Datapath 13

Testing 14

Appendix 15

Appendix A: Stack 15

Appendix B : Stack Test 15

Appendix C : Register 17

Appendix D : Decode 17

Appendix E : Register File 18

Appendix F : Fetch 19

Appendix G : Fetch Test 21

Appendix H : Control Unit 22

Appendix I : Control Unit Test 28

Appendix J : ALU 29

Appendix K : ALU Test 31

Appendix L : Memory 32

Appendix M : Memory Test 33

Appendix N : Write Back 34

Table of Figures

Figure 1: Instruction memory 7
Figure 2: Data Memory 7
Figure 3: Register File 8
Figure 4: Arithmetic Logical Unit 8
Figure 5: Stack 9
Figure 6: Write Back 9
Figure 7: Control Unit 10
Figure 8: Finite State Diagram 12
Figure 9: Full Datapth 13
Figure 10: Full Datapath on Active-HDL program 13
Figure 11: code for testing R-type instructions 14
Figure 12: Code for testing Laod, SLL, and SLLV Instructions 14
Figure 13: Code for testing I-type instructions 14

Design Specification

1. Motivation

In this project, a simple multi-cycle RISC processor is designed according to the following specifications:

- The instruction size is 32 bits
- 32 32-bit general-purpose registers: from R0 to R31.
- A special purpose register for the program counter (PC).
- It has a stack called control stack which saves the return addresses.
- 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.

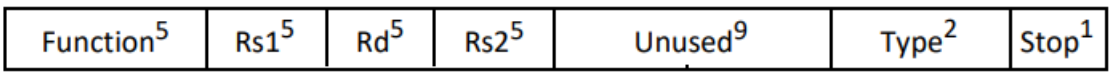
- Four instruction types (R-type, I-type, J-type, and S-type).
- The processor's ALU has an output signal called "zero" signal, which is asserted when the result of the
- last ALU operation is zero.
- Separate data and instructions memories.

2. Instruction Types and Formats

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

- **2-bit instruction type** (00: R-Type, 01: J-Type, 10: I-type, 11: S-type)
- **5-bit function**, to determine the specific operation of the instruction.
- **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.

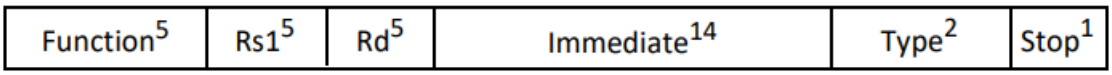
2.1 R-type – Register Type



The R-type format has the following fields:

- **5-bit Rs1:** first source register
- **5-bit Rd:** destination register
- **5-bit Rs2:** second source register
- **9-bit unused**

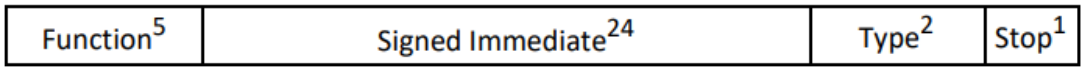
2.2 I-type – Immediate Type



The I-type format has the following fields:

- **5-bit Rs1:** first source register
- **5-bit Rd:** destination register
- **14-bit immediate:** unsigned for logic instructions, and signed otherwise

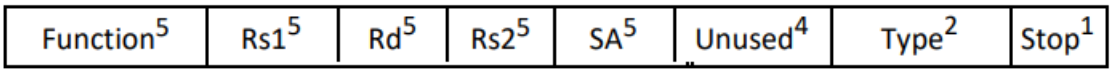
2.3 J-type – Jump Type



The J-type format has the following field:

- **24-bit signed immediate:** jump offset

2.4 S-type – Shift Type



The S-type format has the following fields:

- **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**

3. Instruction Set

The table below shows the different instructions implemented. It shows their type, the function's value, and their meaning in RTN (Register Transfer Notation).

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) & Immediate14	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + Immediate14	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm14)	00010
8	SW	Mem(Reg(Rs1) + Imm14) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
		J-Type Instructions	
10	J	PC = PC + Immediate24	00000
11	JAL	PC = PC + Immediate24 Stack.Push (PC + 4)	00001
		S-Type Instructions	
12	SLL	Reg(Rd) = Reg(Rs1) << SA5	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA5	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLVR	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

Components

1. Memory

In this project, a memory component is used to store instructions and data separately. The instruction memory and data memory are separated in order to reduce conflicts and streamline the fetching and execution of instructions.

1.1 Instruction Memory

The instruction memory stores the program instructions that need to be executed by the RISC processor. It serves as a dedicated memory unit specifically designed for storing the instructions of a program. It is responsible for providing the instructions to the fetch stage of the processor. It interfaces with the program counter (PC), which holds the address of the next instruction to be fetched. The instruction memory receives this address from the PC and retrieves the corresponding instruction from its storage. It plays a crucial role in the processor's operation by storing and providing the instructions needed to execute a program , and enables the processor to fetch, decode, and execute instructions. Appendix[F]

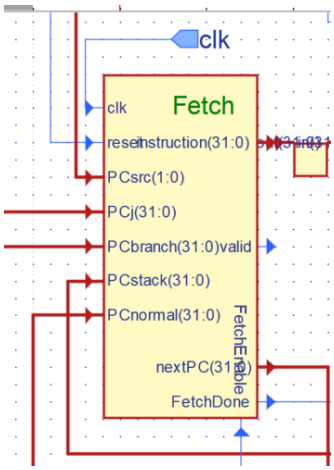


Figure 1: Instruction memory

1.2 Data Memory

The data memory stores and manage the data used by the RISC processor during program execution. It provides a dedicated memory space for storing and retrieving data values, separate from the instruction memory. It enables the processor to read data from and write data to specific memory locations. During the execution of instructions that involve data operations, such as load (LW) and store (SW) instructions, the data memory plays a crucial role. For load instructions, the data memory retrieves the data value from the specified memory address and provides it to the processor for further processing. For store instructions, the data memory stores the data value into the designated memory address. Appendix[L]

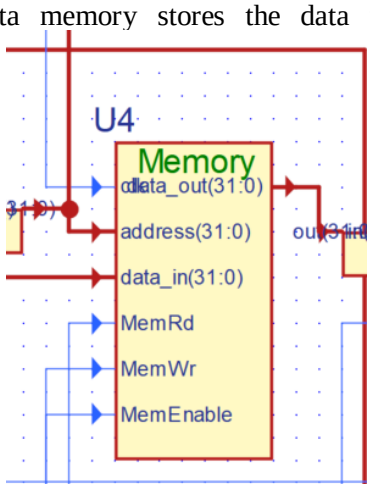


Figure 2: Data Memory

2. Register File

Register File consists of 32 general-purpose registers, each 32 bits wide. It has several input ports, including a clock signal **(clk)** for synchronous operations, control signals for write operations **(RegWR)** and source selection **(RegSrc),** and inputs specifying the source **(Rs, Rt)** and destination **(Rd)** register addresses. It also has an input for enabling the register file **(RegFileEnable).** The output ports include two 32-bit output buses **(BusA and BusB),** which carry the values read from the register file. The register file is an essential component for the execution of instructions that require register operations. The ability to read and write to these registers efficiently is essential for the smooth and effective execution of the processor. Appendix[E].

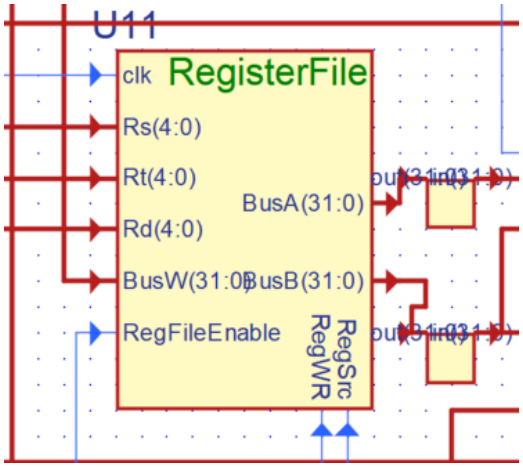


Figure 3: Register File

3. Arithmetic Logic Unit (ALU)

The ALU plays a critical role in performing various arithmetic and logical operations on the input data based on the specified ALU operation **(ALUop).** The module takes in several input signals, including a clock signal **(clk)** for synchronous operations, a reset signal **(reset)** for resetting the ALU, a 4-bit ALU operation signal **(ALUop)** indicating the specific operation to be performed, two 32-bit input signals **(rs1 and rs2)** representing the source operands, a 14-bit immediate value **(imm),** and a 5-bit shift amount **(SA).** it also has output ports that include a 32-bit result **(result)** and a zero flag **(zero).** The ALU implements a combination of arithmetic operations such as AND, ADD, SUB, CMP, ANDI, ADDI, as well as shift operations like SLL, SRL, SLLV, and SRLV. The **reg_rd** register stores the result of the ALU operation, and the **zero_reg** register keeps track of whether the result is zero. The result and zero flag are assigned to the corresponding output ports. Overall, the ALU is responsible for executing arithmetic and logical operations, contributing to the processor's overall computational capabilities. Appendix [J]

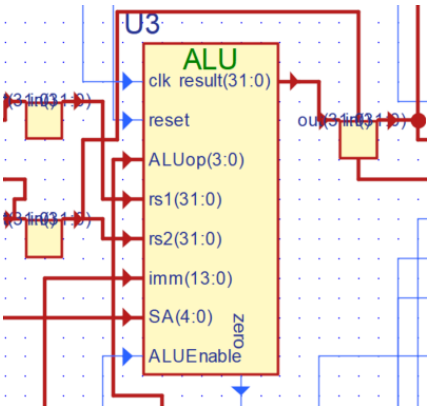


Figure 4: Arithmetic Logical Unit

4. Stack

A stack is a data structure that follows the Last-In-First-Out (LIFO) principle, allowing for temporary storage and retrieval of data. The Stack module features input and output ports for interfacing with other components. Inputs include a clock signal **(clk),** control signals for write **(write)** and read **(read)** operations, and a 32-bit data input **(data_in).** The module's output is a 32-bit data output **(data_out).** Internally, the stack is implemented as an array of 32-bit registers, denoted as stack[0:31], which represents the stack elements. Additionally, there is a 5-bit register called top that keeps track of the current top position of the stack. When the write signal is asserted, the module checks if there is available space in the stack (top < 31) to perform a write operation. If space exists, the 32-bit data_in value is stored at the position indicated by **top**, and top is incremented by 1 (top = top + 5'b00001). It used for temporary storage during program execution. It enables data to be pushed (written) onto the stack and popped (read) from the stack in a LIFO manner. Appendix [A]

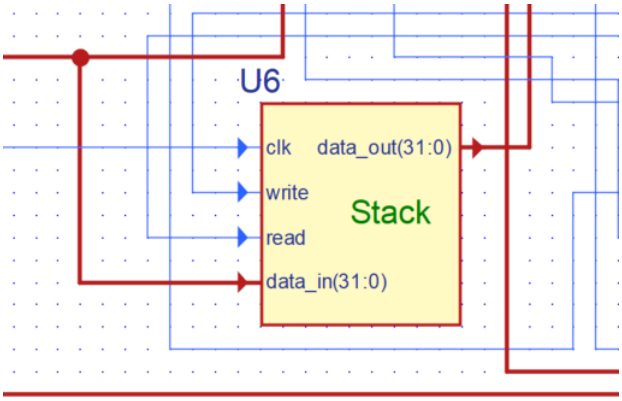


Figure 5: Stack

5. Write Back

The write back stage is to update the value of the destination register with the computed result from the execution stage. During the write back stage, the processor verifies that the instruction being processed does indeed require a write back operation (such as an arithmetic operation that produces a result) and that the destination register is not a special-purpose register. If these conditions are met, the computed result is written back to the destination register, updating its value for future instructions. It is a critical part of the processor's data flow, ensuring that the results of computations are stored correctly in the register file. Appendix[N]

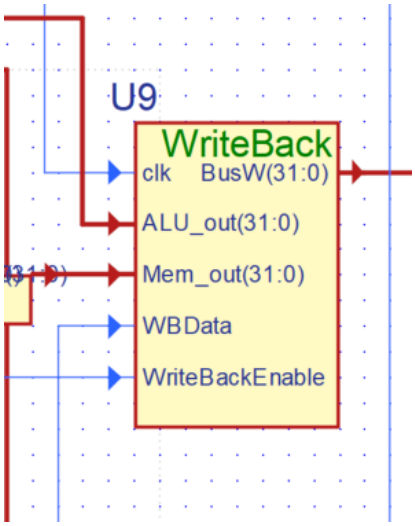


Figure 6: Write Back

6. Control Unit

The control unit plays a crucial role in the RISC processor design project by effectively coordinating and controlling the execution of instructions. One of its key responsibilities is to decode the instructions fetched from memory and generate the necessary control signals to facilitate the smooth operation of the processor. To accomplish this, our control unit utilizes a Finite State Machine, which enables precise control over the various control signals, particularly the enables. By leveraging this approach, the control unit analyzes the instruction's opcode and other pertinent fields, allowing it to determine the specific operation to be performed, such as arithmetic, logical, or memory operations. This analysis enables the control unit to generate the appropriate control signals that enable or disable different functional units within the processor, including the ALU, register file, memory unit, and data path. As the brain of the RISC processor, the control unit acts as the orchestrator, ensuring the accurate and efficient execution of instructions in accordance with the defined Instruction Set Architecture (ISA) while effectively managing the flow of instructions. Appendix [H]

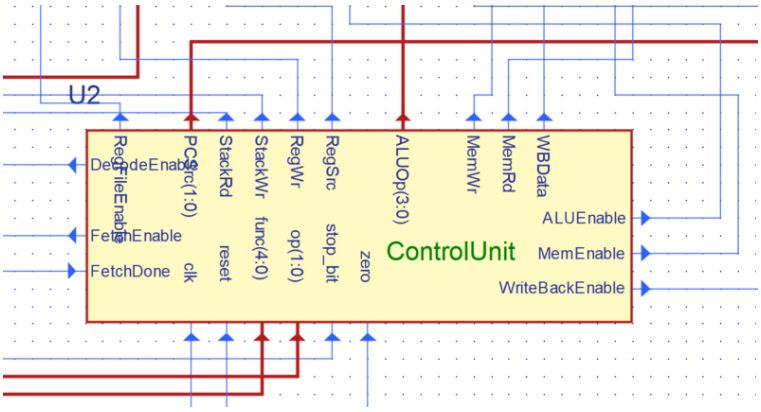


Figure 7: Control Unit

6.1 Truth Table

After building the Datapath, the control signals were derived as shown in Tables below:

Table 1

Stage or Operation / control Signals	*PCSrc*	*StackRd*	*StackWr*	*RegWr*	*RegSRC*	*ALUop*	*ALUSrc*	*MemWr*	*MemRd*	*WBData*
R-type	0	StopBit	0	1	0 = Rt		0	0	0	0-CMP
ANDI	0	StopBit	0	1	x	ANDI	1	0	0	0
ADDI	0	StopBit	0	1	x	ADDI	1	0	0	0
LW	0	StopBit	0	1	x	LW	1	0		1
SW	0	StopBit	0	0	1 = Rd	SW	1	1	0	x
BEQ Z=0	0	StopBit	0	0	1 = Rd	BEQ not taken	0	0	0	0
BEQ Z=1	1	StopBit	0	0	1 = Rd	BEQ taken	0	0	0	0
J	2	StopBit	0	0	x	J	x	0	0	x
JAL	2	StopBit	1	0	x	JAL	x	0	0	x
SLL	0	StopBit	0	1	x	SLL	2	0	0	0
SLR	0	StopBit	0	1	x	SLR	2	0	0	0
SLLV	0	StopBit	0	1	0 = Rt	SLLV	0	0	0	0
SLVR	0	StopBit	0	1	0 = Rt	SLRV	0	0	0	0

Table 2

Opcode	Operation	ALU operation	ALUop
00	R-type	AND	0000
00	R-type	ADD	0001
00	R-type	SUB	0010
00	R-type	CMP	0011
10	ANDI	AND immediate	0100
10	ADDI	ADD immediate	0101
10	LW	Load word	0110
10	SW	Store word	0111
10	BEQ	Branch if Equal	1000
11	SLL	Shift left logical	1011
11	SLR	Shift right logical	1100
11	SLLV	Shift left logical variable	1101
11	SLRV	Shift right logical variable	1110

6.2 Boolean Expression

- StackRd = Stopbit
- StackWr = JAL
- RegWr = Not(SW+BEQ+J+JAL)
- RegSRC = SW + BEQ
- MemWr = SW
- MemRd = LW
- WBData = LW

6.3 Finite State Machine

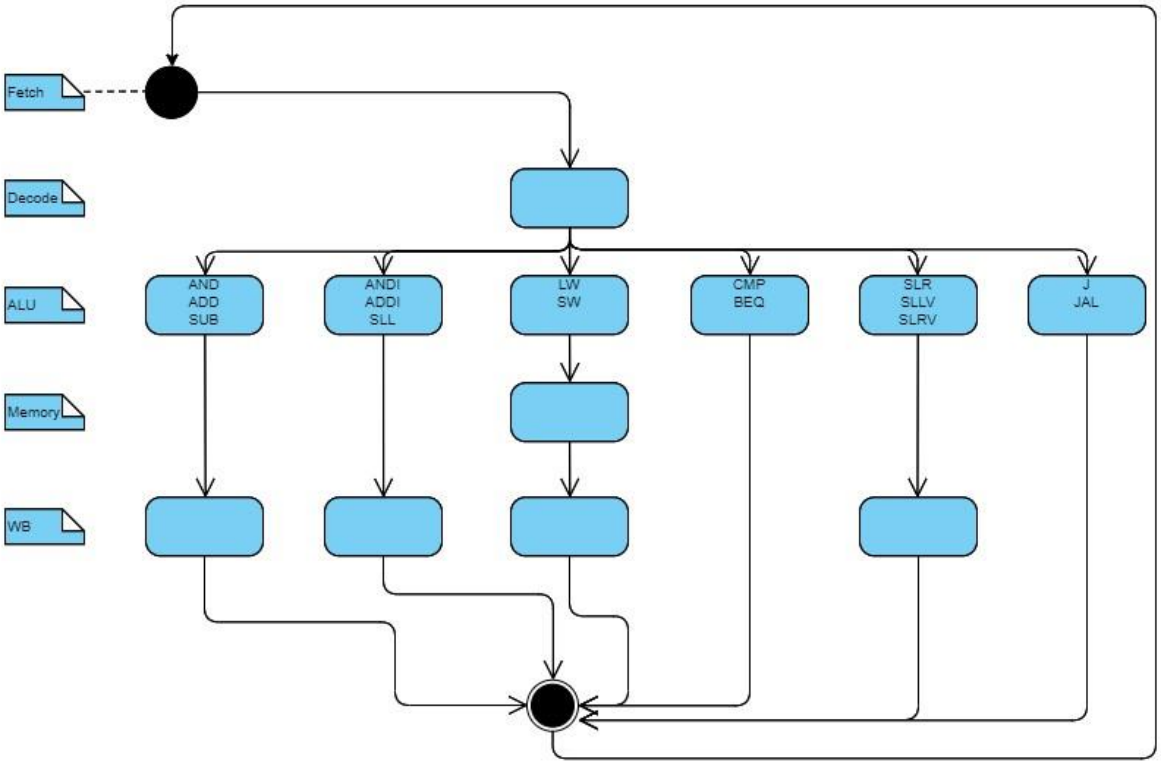


Figure 8: Finite State Diagram

Full Datapath

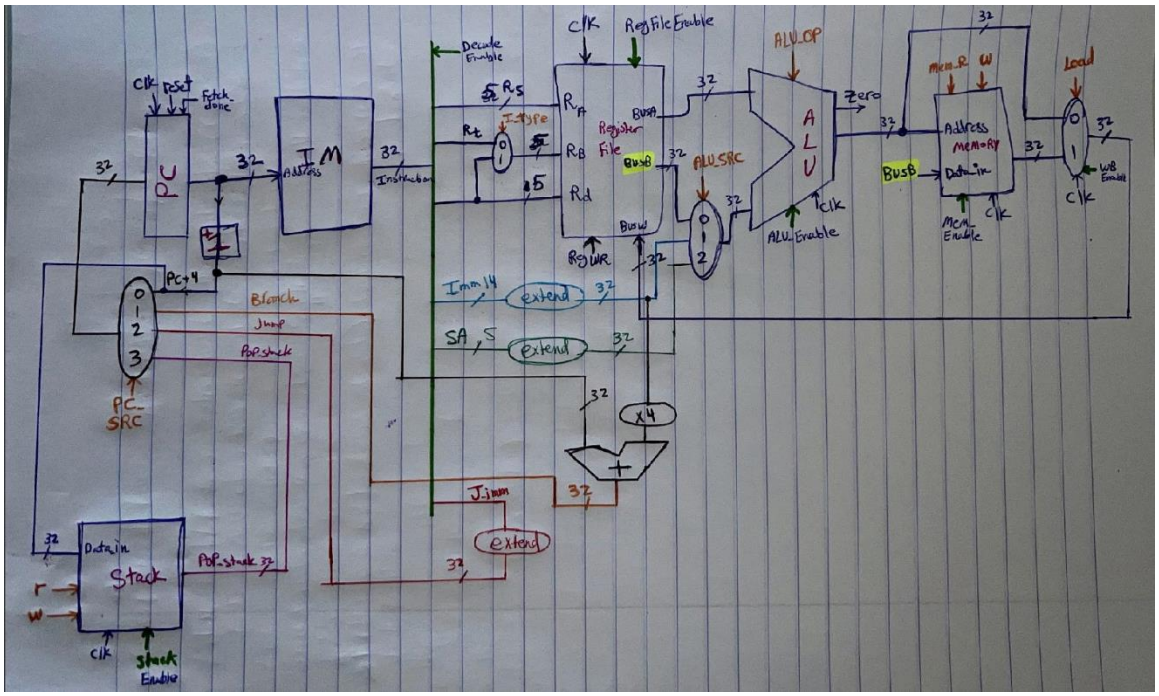


Figure 9: Full Datapth

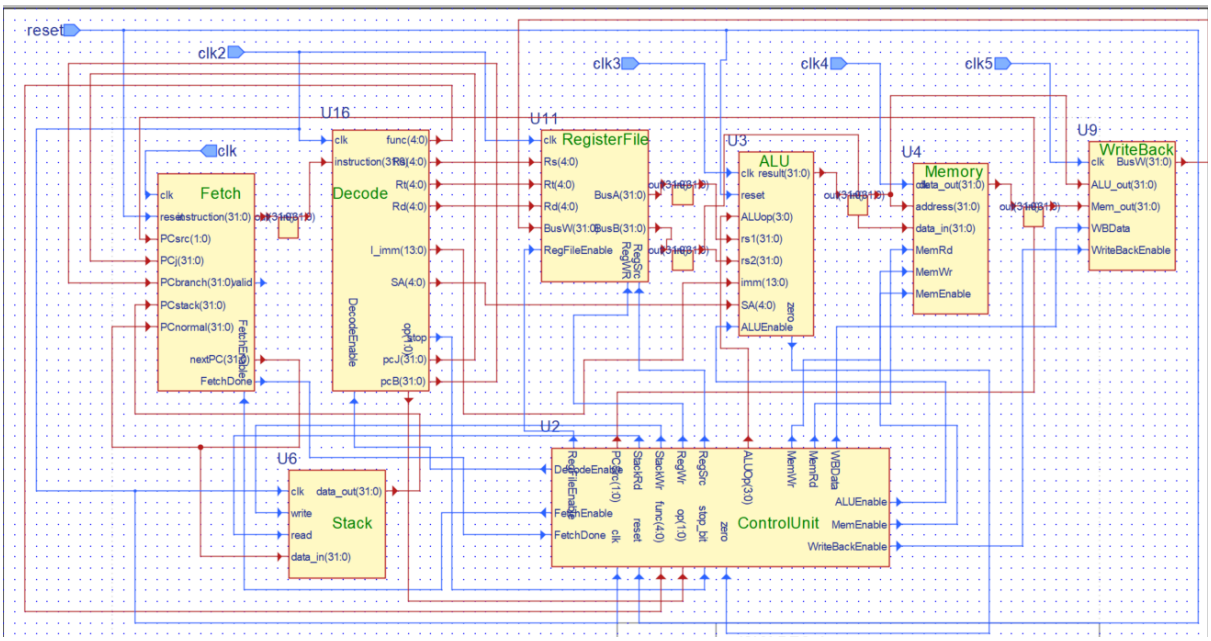


Figure 10: Full Datapath on Active-HDL program

Testing

- R-type :

```
R type test:
IM[0]=32'b00001000010001000000000001000100;//ADDI Rs=00001, Rd=00010, IMM=8
IM[1]=32'b00001000010001100000000100000100;//ADDI Rs=00001, Rd=00011, IMM=32
IM[2]=32'b00001000100011100011000000000000;//ADD Rs1=00010, Rd=00111, Rs2=00011 (Adding the above two results)
```

Figure 11: code for testing R-type instructions

The link below clarifies the R-type testing through a screen record :
https://drive.google.com/file/d/1tVdX67_wPwDzrQx8Gm5qSF91_dhCj8LY/view?usp=drive_link

- Load , SLL, and SLLV :

```
//Load test:
IM[0]=32'b00010000010001100000000000000100;//Load from the address stored in Rs1=00001, store it in Rd=00011

//SLL and SLLV test:
IM[0]=32'b00000000010001100100000100000110;//SLL Rs1=00001, Rd=00011, Rs2=00100, SA=00010
IM[1]=32'b00010000010001100100000100000110;//SLLV Rs1=00001, Rd=00011, Rs2=00100, SA=00010
```

Figure 12: Code for testing Laod, SLL, and SLLV Instructions

The link below clarifies the Load , SLL, and SLLV testing through a screen record :

https://drive.google.com/file/d/106fnQOfGccrjBIFEqeoE0jGcREfDnX66/view?usp=drive_link

- I-type , JAL and Stack :

```
//Jump and link + Stop bit (stack pop) test:
IM[0]=32'b00001000010000100000000001000100;//ADDI Rs=00001, Rd=00010, IMM=8
IM[1]=32'b00001000010001000000000100000100;//ADDI Rs=00001, Rd=00010, IMM=32
IM[2]=32'b00001000000000000000000000100010;//Jal go to instruction [4], push PC=2+1 to stack
IM[3]=32'b00001000010001000000000100011100;//ADDI Rs=00001, Rd=00010, IMM=35
IM[4]=32'b00001000010001000000011100000101;//ADDI Rs=00001, Rd=00010, IMM=224, stop bit is 1, go back to 3
```

Figure 13: Code for testing I-type instructions

The link below clarifies the I-type testing through a screen record :

https://drive.google.com/file/d/1YAf5kKSyiC_5o-3GCfHjyCnazDAhdZwT/view?usp=drive_link

Appendix

Appendix A: Stack

```
module Stack(
  input clk,
  input write,
  input read,
  input [31:0] data_in,
  output reg [31:0] data_out
);

  reg [31:0] stack [0:31];
  reg [4:0] top = 5'b00000;

  always @(posedge clk) begin
    if (write) begin
      if (top < 31) begin
        stack[top] = data_in;
                    top = top + 5'b00001;
      end
    end
    else if (read) begin
      if (top > 5'b00000) begin
                    top = top - 1;
        data_out = stack[top];
      end
    end
  end

endmodule
```

Appendix B : Stack Test

```
module StackTest;

  reg clk;
  reg write;
  reg read;
  reg [31:0] data_in;
  wire [31:0] data_out;

  // Instantiate the Stack module
  Stack dut (
    .clk(clk),
    .write(write),
    .read(read),
    .data_in(data_in),
    .data_out(data_out)
  );

  // Clock generation
  always #5 clk = ~clk;

  // Test sequence
  initial begin
```

```
    clk = 0;
    write = 0;
    read = 0;
    data_in = 0;

    // Push data onto the stack
    #10;
    write = 1;
    data_in = 32'h12345678; // Data to be pushed
    #10;
    write = 0;

    #10;
    write = 1;
    data_in = 32'hAABBCCDD; // Data to be pushed
    #10;
    write = 0;

    #10;
    write = 1;
    data_in = 32'h11223344; // Data to be pushed
    #10;
    write = 0;

    // Pop data from the stack
    #20;
    read = 1;
    #10;
    read = 0;
          #220;

    // Verify the popped data
    if (data_out === 32'h11223344) begin
      $display("Popped data is correct!");
    end else begin
      $display("Popped data is incorrect!");
    end

    #10;
    read = 1;
    #10;
    read = 0;
    // Verify the popped data
    if (data_out === 32'hAABBCCDD) begin
      $display("Popped data is correct!");
    end else begin
      $display("Popped data is incorrect!");
    end

    // Perform additional push/pop operations as needed

    #20;
    $finish; // End simulation
  end

endmodule
```

Appendix C : Register

```
module Register(input [31:0]in,
        output reg [31:0] out
        );

always @(*)
        begin
                out<=in;
        end
endmodule
```

Appendix D : Decode

```
module Decode ( clk ,instruction , DecodeEnable, func ,Rs ,Rt ,Rd ,op ,I_imm ,SA ,pcJ ,pcB, stop);

input clk ;
input [31:0] instruction ;
input DecodeEnable;

output reg [4:0] func ;

output reg [4:0] Rs ;

output reg [4:0] Rt ;

output reg [4:0] Rd ;

output reg [1:0] op ;

output reg [13:0] I_imm ;

output reg [4:0] SA ;

output reg [31:0] pcJ ;

output reg [31:0] pcB ;

output reg stop ;
//To store the extended value of the immediate in I or J types
reg[31:0] imm32;
reg [31:0] J_imm;
reg [31:0] PCreg;
reg [31:0] decimal_to_add = 4;

always @(posedge clk)
        begin
                //common fields among the instructions:
                //00: R-Type, 01: J-Type, 10: I-type, 11: S-type)
                if(DecodeEnable == 1'b1) begin

                        op <= instruction[2:1];
                        //2. the Function filed
                        func <= instruction[31:27];
                        // 3. the "stop" bit
                        stop <= instruction[0];
                        I_imm <= instruction[16:3];
                        J_imm = instruction[25:3];
```

```
                        pcJ = J_imm;
                        pcB = I_imm;
                        Rs <= instruction[26:22];
                        Rt <= instruction[16:12];
                        Rd <= instruction[21:17];
                        SA <= instruction[11:7];

                end
        end


endmodule
```

Appendix E : Register File

```
module RegisterFile(
  input clk,
  input RegWR,
  input RegSrc,//RegSrc: 0 -> rt (normal flow ), 1 -> rd (in case of Store operation)
  input [4:0] Rs,
  input [4:0] Rt,
  input [4:0] Rd,
  input RegFileEnable,
  wire [31:0] BusW,
  output reg [31:0] BusA,
  output reg [31:0] BusB
);

  reg [31:0] registers [31:0];
  integer i;
  reg counter;
  reg [31:0] BusWReg; // Local register variable for BusW

  initial begin
          counter =0;
    for (i = 0; i < 32; i = i + 1)
      registers[i] = i;
  end

  always @(posedge clk) begin
            if(RegFileEnable==1'b1) begin
    BusA = registers[Rs];
    // Normal Flow
    if (RegSrc == 0 && Rt != 0) begin
      BusB = registers[Rt];
    end
    // In case of SW
    else if (RegSrc == 1 && Rd != 0) begin
      BusB = registers[Rd];
    end
          end
  end

  always @(negedge clk) begin
            if(RegFileEnable==1'b1) begin
    // If there is a "write to register command," then rd is always the destination
```

```
                if (RegWR && Rd==Rs && counter!=2 && Rd != 0) begin
                        counter = counter+1;
                end

                else if(RegWR && Rd==Rs && counter==2 && Rd != 0) begin
                        BusWReg = BusW; // Assign BusW to the local register variable
                registers[Rd] = BusWReg;
                        counter = 0;
            end

                else if (RegWR && Rd !=Rs && Rd != 0) begin
              BusWReg = BusW; // Assign BusW to the local register variable
              registers[Rd] = BusWReg;
            end
          end
  end

endmodule
```

Appendix F : Fetch

```
module Fetch (
        input clk,
        input reset,
        input [1:0] PCsrc,//0:PC+4 1:PCbranch 2:PCj 3:PCstack
        input [31:0] PCnormal,
        input [31:0] PCj,
        input [31:0] PCbranch,
        input [31:0] PCstack,
        input FetchEnable,
        output reg [31:0] nextPC,
        output reg [31:0] instruction,
        output reg valid,
        output reg FetchDone
);

reg [31:0] IM [4:0];//instruction memory
reg counter = 1'b0;

initial begin
        //Jump and link + Stop bit (stack pop) test:
        IM[0]=32'b00001000010000100000000001000100;//ADDI Rs=00001, Rd=00010, IMM=8
        IM[1]=32'b00001000010001000000000100000100;//ADDI Rs=00001, Rd=00010, IMM=32
        IM[2]=32'b00001000000000000000000000100010;//Jal go to instruction [4], push PC=2+1
to stack
        IM[3]=32'b00001000010001000000000100011100;//ADDI Rs=00001, Rd=00010, IMM=35
        IM[4]=32'b00001000010001000000011100000101;//ADDI Rs=00001, Rd=00010,
IMM=224, stop bit is 1, go back to 3

        //R type test:
        //IM[0]=32'b00001000010001000000000001000100;//ADDI Rs=00001, Rd=00010, IMM=8
        //IM[1]=32'b00001000010001100000000100000100;//ADDI Rs=00001, Rd=00011,
IMM=32
        //IM[2]=32'b00001000100011100011000000000000;//ADD Rs1=00010, Rd=00111,
Rs2=00011 (Adding the above two results)

        //Load test:
```

```
	//IM[0]=32'b00010000010001100000000000000100;//Load from the address stored in
Rs1=00001, store it in Rd=00011

	//SLL and SLLV test:
	//IM[0]=32'b00000000010001100100000100000110;//SLL Rs1=00001, Rd=00011,
Rs2=00100, SA=00010
	//IM[1]=32'b00010000010001100100000100000110;//SLLV Rs1=00001, Rd=00011,
Rs2=00100, SA=00010

	FetchDone <= 1'b0;
	instruction = 32'd0;
	PCreg = 32'd0;
	//nextPC = PCreg + 1'b1;
end

reg fetchValidReg;
reg done;
reg [31:0] PCreg;
reg [31:0] mem_address;

always @(posedge clk) begin

	if(FetchEnable == 1'b0)
		begin
			FetchDone <= 1'b0;
			instruction = 32'd0;
		end
	else begin
		if (PCsrc == 2'b00) begin
			PCreg = PCnormal;
		end
		else if (PCsrc==2'b01) begin
			PCreg = PCbranch;
		end
		else if (PCsrc==2'b10) begin
			PCreg = PCj;
		end
		else if (PCsrc==2'b11) begin
			PCreg = PCstack;
		end
			mem_address <= {nextPC[31:2],2'b00};
			instruction <= IM[PCreg];
			//FetchDone <= 1'b1;
			//PCreg <= nextPC;
			if (instruction != 32'd0 && FetchDone != 1'b1) begin
				FetchDone <= 1'b1;
				nextPC <= PCreg + 1'b1;
				PCreg <= PCreg + 1'b1;
			end //else begin
				//FetchDone = 1'b0;
			//end
	end


end
endmodule
```

Appendix G : Fetch Test

```
`timescale 1ns/1ps

module Fetch_tb;

  reg clk;
  reg reset;
  reg [1:0] PCsrc;
  reg [31:0] PCdecode;
  reg [31:0] PCj;
  reg [31:0] PCbranch;
  reg [31:0] PCstack;
  wire [31:0] nextPC;
  wire [31:0] instruction;
  wire valid;

  Fetch uut (
    .clk(clk),
    .reset(reset),
    .PCsrc(PCsrc),
          .PCdecode(PCdecode),
    .PCj(PCj),
    .PCbranch(PCbranch),
    .PCstack(PCstack),
    .nextPC(nextPC),
    .instruction(instruction),
    .valid(valid)
  );

  initial begin
    clk = 0;
    reset = 1;
    PCsrc = 0;
    PCj = 0;
    PCbranch = 0;
    PCstack = 0;
    #10
          reset = 0; // Assert reset for 10 time units
    #20;
    $finish;
  end

  always #5 clk = ~clk;

endmodule
```

Appendix H : Control Unit

```
module ControlUnit(
        input clk,
        input reset,
        input [4:0] func,
        input [1:0] op,
        input stop_bit,
        input zero,
        input FetchDone,
        output reg [1:0] PCSrc,
        output reg StackRd,
        output reg StackWr,
        output reg RegWr,
        output reg RegSrc,
        output reg [3:0] ALUOp,
        output reg MemWr,
        output reg MemRd,
        output reg WBData,
        output reg FetchEnable,
        output reg DecodeEnable,
        output reg RegFileEnable,
        output reg ALUEnable,
        output reg MemEnable,
        output reg WriteBackEnable
);

        //reg FETCH;
        //reg DECODE;
        //reg EXECUTE;
        //reg MEM_ACCESS;
        //reg WRITE_BACK;

        parameter [2:0] FETCH = 3'b000;
   parameter [2:0] DECODE = 3'b001;
   parameter [2:0] EXECUTE = 3'b010;
   parameter [2:0] MEM_ACCESS = 3'b011;
   parameter [2:0] WRITE_BACK = 3'b100;

   reg [2:0] STATE;
   reg [2:0] NEXT_STATE;

   initial begin
      STATE = FETCH;
                FetchEnable = 1'b1;
                DecodeEnable = 1'b1;
                RegFileEnable = 1'b1;
   end

        always @(posedge clk) begin
      if (reset) begin
         STATE <= FETCH;
      end
      else begin
         STATE <= NEXT_STATE;
      end
   end
```

```
always_comb begin
   case (STATE)
      FETCH:
         NEXT_STATE = DECODE;
      DECODE:
         NEXT_STATE = EXECUTE;
      EXECUTE:
         NEXT_STATE = MEM_ACCESS;
      MEM_ACCESS:
         NEXT_STATE = WRITE_BACK;
      default:
         NEXT_STATE = FETCH;
   endcase
end

       always @(posedge clk) begin
   case (STATE)
      FETCH:
         begin
            if(FetchDone==1'b1) begin
                                                FetchEnable<=1'b0;
                                          end
                                          else begin
                                                FetchEnable <= 1'b1;
                                          end
                                          DecodeEnable <= 1'b1;
                                          ALUEnable <= 1'b0;
                                          MemEnable <= 1'b0;
                                          WriteBackEnable <= 1'b0;

         end
      DECODE:
         begin
            // Control signals assignment for DECODE stage
            FetchEnable <= 1'b0;
                                          DecodeEnable <= 1'b0;
                                          RegFileEnable <= 1'b1;
                                          ALUEnable <= 1'b1;
                                          MemEnable <= 1'b0;
                                          WriteBackEnable <= 1'b0;
         end
      EXECUTE:
         begin
            // Control signals assignment for EXECUTE stage
            FetchEnable <= 1'b0;
                                          DecodeEnable <= 1'b0;
                                          RegFileEnable <= 1'b0;
                                          ALUEnable <= 1'b1;
                                          MemEnable <= 1'b0;
                                          WriteBackEnable <= 1'b0;
         end
      MEM_ACCESS:
         begin
            // Control signals assignment for MEM_ACCESS stage
            FetchEnable <= 1'b0;
```

```
                                        DecodeEnable <= 1'b0;
                                        ALUEnable <= 1'b0;
                                        MemEnable <= 1'b1;
                                        WriteBackEnable <= 1'b0;
        end
      WRITE_BACK:
        begin
          // Control signals assignment for WRITE_BACK stage
          FetchEnable <= 1'b1;
                                        DecodeEnable <= 1'b0;
                                        ALUEnable <= 1'b0;
                                        MemEnable <= 1'b0;
                                        WriteBackEnable <= 1'b1;
                                        RegFileEnable <= 1'b1;
        end
    endcase
end

      always @(posedge clk) begin

            //R type
            if(op==2'b00) begin
                  PCSrc <= 2'b00;
                  StackRd <= stop_bit;
                  StackWr <= 1'b0;
                  RegWr <= 1'b1;
                  RegSrc <= 1'b0;
                  MemWr <= 1'b0;
                  MemRd <= 1'b0;
                  WBData <= 1'b0;
                  if(func==5'b00000) begin

                        ALUOp <= 4'b0000;//AND

                  end
                  else if(func==5'b00001) begin

                        ALUOp <= 4'b0001;//ADD

                  end
                  else if(func==5'b00010) begin

                        ALUOp <= 4'b0010;//SUB

                  end
                  else if(func==5'b00011) begin

                        ALUOp <= 4'b0011;//CMP

                  end


            end//end of R type

            else if((op==2'b10) && (func==5'b00000)) begin
                  //ANDI
```

```
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b1;
            RegSrc <= 1'b0;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b0100;//ANDI

        end
        else if((op==2'b10) && (func==5'b00001)) begin
            //ADDI
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b1;
            RegSrc <= 1'b0;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b0101;//ADDI

        end
        else if((op==2'b10) && (func==5'b00010)) begin
            //LW
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b1;
            RegSrc <= 1'b0;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b1;
            WBData <= 1'b1;
            ALUOp <= 4'b0110;//LW

        end
        else if((op==2'b10) && (func==5'b00011)) begin
            //SW
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b0;
            RegSrc <= 1'b0;//we dont care
            MemWr <= 1'b1;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b0111;//SW

        end
        else if((op==2'b10) && (func==5'b00100) && (zero==1'b0)) begin
            //BEQ NOT TAKEN
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b0;
```

```
            RegSrc <= 1'b1;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b1000;//BEQ

        end
        else if((op==2'b10) && (func==5'b00100) && (zero==1'b1)) begin
            //BEQ TAKEN
            PCSrc <= 2'b01;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b0;
            RegSrc <= 1'b1;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b1000;//BEQ

        end
        else if((op==2'b01) && (func==5'b00000)) begin
            //Jump
            PCSrc <= 2'b10;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b0;
            RegSrc <= 1'b1;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b1001;//J

        end
        else if((op==2'b01) && (func==5'b00001)) begin
            //Jump and link
            PCSrc <= 2'b10;
            StackRd <= stop_bit;
            StackWr <= 1'b1;
            RegWr <= 1'b0;
            RegSrc <= 1'b1;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
            ALUOp <= 4'b1010;//Jal

        end
        else if((op==2'b11) && (func==5'b00000)) begin
            //SLL
            PCSrc <= 2'b00;
            StackRd <= stop_bit;
            StackWr <= 1'b0;
            RegWr <= 1'b1;
            RegSrc <= 1'b1;//we dont care
            MemWr <= 1'b0;
            MemRd <= 1'b0;
            WBData <= 1'b0;
```

```
                ALUOp <= 4'b1011;//SLL

            end
            else if((op==2'b11) && (func==5'b00001)) begin
                //SLR
                PCSrc <= 2'b00;
                StackRd <= stop_bit;
                StackWr <= 1'b0;
                RegWr <= 1'b1;
                RegSrc <= 1'b1;//we dont care
                MemWr <= 1'b0;
                MemRd <= 1'b0;
                WBData <= 1'b0;
                ALUOp <= 4'b1100;//SLR

            end
            else if((op==2'b11) && (func==5'b00010)) begin
                //SLLV
                PCSrc <= 2'b00;
                StackRd <= stop_bit;
                StackWr <= 1'b0;
                RegWr <= 1'b1;
                RegSrc <= 1'b0;
                MemWr <= 1'b0;
                MemRd <= 1'b0;
                WBData <= 1'b0;
                ALUOp <= 4'b1101;//SLLV

            end
            else if((op==2'b11) && (func==5'b00011)) begin
                //SLRV
                PCSrc <= 2'b00;
                StackRd <= stop_bit;
                StackWr <= 1'b0;
                RegWr <= 1'b1;
                RegSrc <= 1'b0;
                MemWr <= 1'b0;
                MemRd <= 1'b0;
                WBData <= 1'b0;
                ALUOp <= 4'b1110;//SLRV

            end

            if(stop_bit==1'b1) begin//if stop bit is 1

                PCSrc <= 2'b11;

            end



        end

endmodule
```

Appendix I : Control Unit Test

```
module ControlUnit_TB;

  // Inputs
  reg clk;
  reg reset;
  reg [4:0] func;
  reg [1:0] op;
  reg stop_bit;
  reg zero;

  // Outputs
  wire [1:0] PCSrc;
  wire StackRd;
  wire StackWr;
  wire RegWr;
  wire RegSrc;
  wire ExtOp;
  wire [3:0] ALUOp;
  wire [1:0] ALUSrc;
  wire MemWr;
  wire MemRd;
  wire WBData;

  // Instantiate the unit under test (UUT)
  ControlUnit dut (
    .clk(clk),
    .reset(reset),
    .func(func),
    .op(op),
    .stop_bit(stop_bit),
    .zero(zero),
    .PCSrc(PCSrc),
    .StackRd(StackRd),
    .StackWr(StackWr),
    .RegWr(RegWr),
    .RegSrc(RegSrc),
    .ALUOp(ALUOp),
    .MemWr(MemWr),
    .MemRd(MemRd),
    .WBData(WBData)
  );

  // Clock initialization
  initial
    clk = 0;

  always #5
    clk = ~clk;

  // Reset initialization
  initial
    reset = 1;

  always #10
    reset = 0;
```

```
  // Stimulus
  initial begin
    // Set initial input values

    // Test case 1
    #10;
    func = 5'b00000;
    op = 2'b01;
    stop_bit = 1'b0;
    zero = 1'b0;


    #100;
    $finish;
  end

endmodule
```

Appendix J : ALU

```
module ALU (
  input wire clk,
  input wire reset,
  input wire [3:0] ALUop,
  input wire [31:0] rs1,
  input wire [31:0] rs2,
  input wire [13:0] imm,
  input wire [4:0] SA,
  input wire ALUEnable,
  output wire [31:0] result,
  output wire zero
);

  reg [31:0] reg_rd;
  reg zero_reg;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      reg_rd <= 0;
      zero_reg <= 0;
    end
  else
            begin
                      if(ALUEnable == 1'b1) begin
      case (ALUop)
        4'b0000: // AND
          reg_rd <= rs1 & rs2;

        4'b0001: // ADD
                                  begin
                                  reg_rd <= rs1 + rs2;
                                  end

        4'b0010: // SUB
                                  begin
                                reg_rd <= rs1 - rs2;
```

```
                        end

    4'b0011: // CMP
                zero_reg <= (rs1 < rs2);
                // No need to assign reg_rd, carry, or overflow in CMP case

    4'b0100: // ANDI
                reg_rd <= rs1 & { {18{imm[13]}}, imm }; // Sign-extend 14-bit immediate to 32-bit

    4'b0101: // ADDI
                        begin
                    reg_rd <= rs1 + { {18{imm[13]}}, imm }; // Sign-extend 14-bit immediate to
32-bit
                        end

    4'b0110: // LW
        reg_rd <= rs1 + imm;

    4'b0111: // SW
        reg_rd <= rs1 + imm;

    4'b1000: // BEQ
                        begin
                    zero_reg <= (rs1 == rs2);
                    // Reset carry and overflow flags for BEQ
                        end
    4'b1011: // SLL
        reg_rd <= rs1 <<< SA;

    4'b1100: // SRL
        reg_rd <= rs1 >>> SA;

    4'b1101: // SLLV
        reg_rd <= rs1 <<< rs2;

    4'b1110: // SRLV
                reg_rd <= rs1 >>> rs2;

    default:
                 begin
        reg_rd <= 0;
                 end

    endcase//of case

    zero_reg <= (reg_rd == 0);
  end   //of else
 end
 end

 assign result = reg_rd;
 assign zero = zero_reg;

endmodule
```

Appendix K : ALU Test

```
module ALUTest;

  reg clk;
  reg reset;
  reg [3:0] ALUop;
  reg [31:0] rs1;
  reg [31:0] rs2;
  reg [13:0] imm;
  reg [4:0] SA;
  wire [31:0] result;
  wire zero;
  wire carry;
  wire overflow;

  ALU dut (
    .clk(clk),
    .reset(reset),
    .ALUop(ALUop),
    .rs1(rs1),
    .rs2(rs2),
    .imm(imm),
    .SA(SA),
    .result(result),
    .zero(zero),
    .carry(carry),
    .overflow(overflow)
  );

  initial begin
    clk = 0;
    reset = 1;
    ALUop = 4'b0000;
    rs1 = 32'b00000000000000000000000000000100;
    rs2 = 32'd2;
    imm = 14'b00000000100100;
    SA = 5'b00101;

    #10 reset = 0;

    // Test case 10: SLL
    ALUop = 4'b0100;
    #50;


    $finish;
  end

  always #5 clk = ~clk;

endmodule
```

Appendix L : Memory

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

reg [31:0] memory [255:0];

initial begin
  integer i;
  reg [31:0] lfsr;

  // Initialize the LFSR register
  lfsr = 32'hACE1B5ED;

  // Loop through each element of the array
  for (i = 0; i < 256; i = i + 1) begin
    memory[i] = lfsr;
    lfsr = {lfsr[0] ^ lfsr[1] ^ lfsr[3] ^ lfsr[4], lfsr[31:1]};
  end
end

always @(posedge clk) begin

        if(MemEnable == 1'b1) begin

        if(MemWr == 1'b1) begin

                memory[address] = data_in;

        end
        else if(MemRd == 1'b1) begin

                data_out = memory[address];

        end
        end


end

endmodule
```

Appendix M : Memory Test

```
module MemTest;

  reg clk;
  reg [31:0] address;
  reg [31:0] data_in;
  reg MemRd;
  reg MemWr;
  reg MemEnable;
  wire [31:0] data_out;

  // Instantiate the Memory module
  Memory dut (
    .clk(clk),
    .address(address),
    .data_in(data_in),
    .MemRd(MemRd),
    .MemWr(MemWr),
          .MemEnable(MemEnable),
    .data_out(data_out)
  );

  // Clock generation
  always #5 clk = ~clk;

  // Test sequence
  initial begin
          MemEnable = 1'b1;
    clk = 0;
    address = 0;
    data_in = 0;
    MemRd = 0;
    MemWr = 0;

    // Perform write operation
    #10;
    MemWr = 1;
    address = 0; // Address to write to
    data_in = 32'h12345678; // Data to be written
    #10;
    MemWr = 0;
          #10

    // Perform read operation
    #10;
    MemRd = 1;
    address = 0; // Address to read from
    #10;
    MemRd = 0;

    // Verify the read data
    if (data_out === 32'h12345678) begin
      $display("Read data is correct!");
    end else begin
      $display("Read data is incorrect!");
    end
```

```
    // Perform additional read/write operations as needed

    #20;
    $finish; // End simulation
  end

endmodule
```

Appendix N : Write Back

```
module WriteBack(
        input clk,
        input [31:0] ALU_out,
        input [31:0] Mem_out,
        input WBData,
        input WriteBackEnable,
        output reg [31:0] BusW
);

always @(posedge clk) begin

        if(WriteBackEnable == 1'b1) begin

        if(WBData == 1'b0) begin

                BusW <= ALU_out;

        end
        else if(WBData == 1'b1) begin

                BusW <= Mem_out;

        end
        end

end

endmodule
```