Faculty of Engineering & Technology

Electrical & Computer Engineering Department

COMPUTER ARCHITECTURE

ENCS4370

Course Project 2

Design and verify a simple RISC processor in Verilog

Prepared by:

Rebal Zabade-1210162

Sondos Farrah-1200905

Mohammad Makhamreh -1200227

Instructor: Dr. Aziz Qaroush.

Date: 20th June 2024

Section: 3

Abstract:

This project involves designing and verifying a multi-cycle RISC processor in Verilog. The 16-bit processor includes 8 general-purpose registers, a program counter, and supports R-type, I-type, J-type, and S-type instructions, encompassing 21 essential operations. The multi-cycle architecture executes instructions over multiple clock cycles, optimizing resource use and simplifying control logic. Verification includes creating a testbench and running code sequences to ensure functionality. The report details the Datapath, control path, control signal generation, and verification results, emphasizing correctness and completeness.

Table of Contents

Table of Figure

Table of Table

1. Theory:

The design and verification of a multi-cycle RISC (Reduced Instruction Set Computing) processor involve several key theoretical concepts that form the foundation of modern computer architecture. This section outlines the essential theoretical aspects underpinning the project.

1.1 Registers:

Function:

- Store data, operands, and intermediate results for instructions.
- Provide fast access to frequently used data, minimizing access to slower main memory.

1.1.1 General-Purpose Registers:

- **Number of Registers**: 8 (R0 to R7).
- **Size**: 16 bits each.
- **R0**: Hardwired to zero. Attempts to write to R0 are discarded.

Operations:

- **Read**: During the decode stage, source register values (e.g., Rs1 and Rs2) are read.
- **Write**: During the write-back stage, the result of an operation is written to the destination register (e.g., Rd).

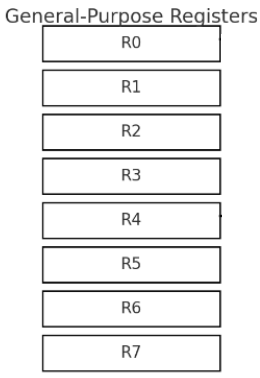

Figure 1: General-Purpose Registers

1.1.2 Special-Purpose Register:

Program Counter (PC):

- **Function**: Holds the address of the next instruction to be fetched.
- **Operations**:
  - **Increment**: Increases by the instruction length (2 bytes for 16-bit instructions) after each fetch.
  - **Update**: Modified by jump and branch instructions to change the control flow.

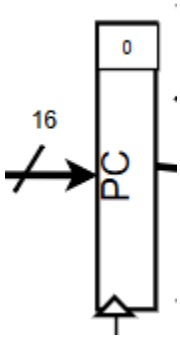

Figure 2: Program Counter (PC)

1.2 Multiplexer (MUX):

Multiplexers are used to select between different data sources. In the data path, they route data based on control signals.

- **Function**: Directs the flow of data to different components based on control signals.
- **Operations**: Selects inputs for the ALU, register file, and other components.
- **Type**: 2*1 MUX and 4*1 MUX

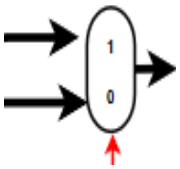

*Figure 3: MUX 2*1*

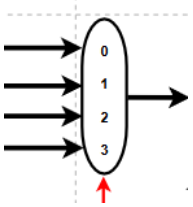

*Figure 4: MUX 4*1*

1.3 Instruction Memory:

Instruction memory stores the program's instructions. It is read during the instruction fetch stage to retrieve the current instruction for execution.

- **Function**: Stores the machine code of the program.
- **Operations**: Read operation based on the address in the PC.

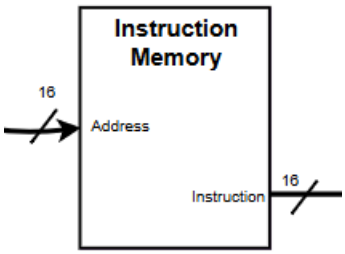

Figure 5: Instruction Memory

1.4 Register File:

The register file consists of multiple general-purpose registers. In this design, there are 8 registers (R0 to R7), with R0 hardwired to zero. It provides fast storage for temporary data and operands.

- **Function**: Stores intermediate data and operands for instructions.
- **Operations**: Read and write operations based on the instruction's register fields.

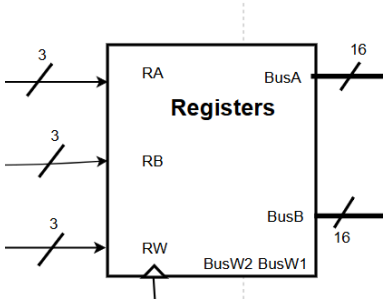

Figure 6: Register File

1.5 Sign-Extend Unit:

The extender unit is used to extend the bit-width of immediate values, ensuring they fit the processor's word size. It is crucial for handling signed and unsigned immediate values correctly.

- **Function**: Extends smaller bit-width immediate values to the full word size.
- **Operations**: Extends the sign of immediate values to ensure correct arithmetic operations.

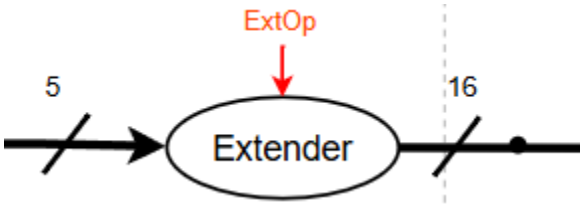

Figure 7: Sign-Extend Unit

1.6 control unit:

The control unit generates control signals that orchestrate the operations of the datapath components. It ensures that each component performs the correct operation at the right time during the instruction execution cycle.

- **Function**: Controls the overall operation of the processor by generating appropriate control signals.
- **Operations**: Decodes the opcode and other fields to produce control signals for the Datapath.

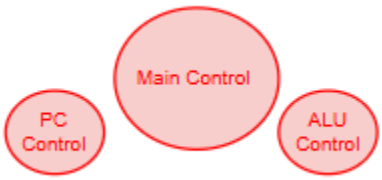

Figure 8:control unit

1.7 Arithmetic Logic Unit (ALU):

The ALU performs arithmetic and logical operations. It is the computational core of the processor, executing operations specified by the instruction.

- **Function**: Performs arithmetic (e.g., addition, subtraction) and logical (e.g., AND, OR) operations.
- **Operations**: Takes inputs from registers or immediate values and produces a result.

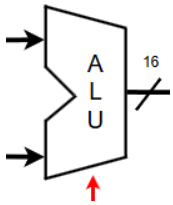

Figure 9: Arithmetic Logic Unit (ALU)

1.8 Comparator:

The Comparator is responsible for comparing register values to determine the outcome of branch instructions. It helps decide the next address of the Program Counter (PC) based on specific conditions.

- **Function**: Compares two register values or a register value with zero to support conditional branch instructions.
- **Operations**:
  - Takes two inputs from the register file (BusA and BusB) or compares a register value with zero.
  - Produces a result of 1 or 0:
    - Result is **1** if the condition is met (branch taken).
    - Result is **0** if the condition is not met (branch not taken).

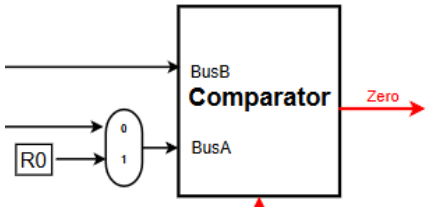

Figure 10: Comparator

1.9 Data Memory:

Data memory is used to store and retrieve data. It is accessed during load (LW, LBu, LBs) and store (Sv) instructions.

- **Function**: Provides storage for data to be loaded or stored during program execution.
- **Operations**: Read or write based on memory addresses calculated by the ALU.

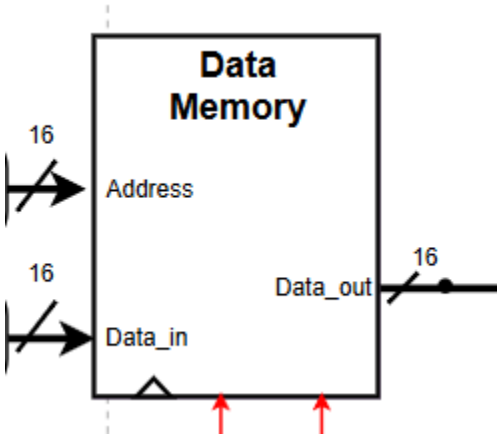

Figure 11: Data Memory

2- Procedure & Discussion:

In this section, we will discuss the procedure followed to generate the control signals for each instruction type (R, I, S) in the multi-cycle RISC processor.

2.1 Control Signals Table(Mohammad, Sondos, Rebal):

The table below provides a detailed mapping of the input opcode to the output control signals for various instructions. This mapping ensures that the processor's Datapath correctly executes each instruction according to its type and specified operation.

Inst	input	Output													
	op	PCSrc	RegSrc2	RegWr1	RegWr2	ALUSrc	ExtOp	ALUOP	AdressSig	DataInSig	MemRd	MemWr	Wbdata	modeSig	RegDst
AND	0000	0	0	1	0	0	0	1	x	x	0	0	0	x	0
ADD	0001	0	0	1	0	0	0	1	x	x	0	0	0	x	0
SUB	0010	0	0	1	0	0	0	1	x	x	0	0	0	x	0
ADDI	0011	0	x	1	0	1	0	1	x	x	0	0	0	x	0
ANDI	0100	0	x	1	0	1	0	1	x	x	0	0	0	x	0
LW	0101	0	x	1	0	1	0	1	0	x	1	0	1	x	0
Lbu	0110	0	x	1	0	1	1	1	0	x	1	0	1	x	0
LBs	0110	0	x	1	0	1	1	1	0	x	1	0	1	x	0
SW	0111	0	1	0	0	1	0	1	0	0	0	1	x	x	0
BGT_T	1000	2	1	0	0	x	1	x	x	x	0	0	x	0	0
BGT_NT	1000	0	1	0	0	x	1	x	x	x	0	0	x	0	0
BGTZ_T	1000	2	1	0	0	x	1	x	x	x	0	0	x	1	0
BGTZ_NT	1000	0	1	0	0	x	1	x	x	x	0	0	x	1	0
BLT_T	1001	2	1	0	0	x	1	x	x	x	0	0	x	0	0
BLT_NT	1001	0	1	0	0	x	1	x	x	x	0	0	x	0	0
BLTZ_T	1001	2	1	0	0	x	1	x	x	x	0	0	x	1	0
BLTZ_NT	1001	0	1	0	0	x	1	x	x	x	0	0	x	1	0
BEQ_T	1010	2	1	0	0	x	1	x	x	x	0	0	x	0	0
BEQ_NT	1010	0	1	0	0	x	1	x	x	x	0	0	x	0	0
BEQZ_T	1010	2	1	0	0	x	1	x	x	x	0	0	x	1	0
BEQZ_NT	1010	0	1	0	0	x	1	x	x	x	0	0	x	1	0
BNE_T	1011	2	1	0	0	x	1	x	x	x	0	0	x	0	0
BNE_NT	1011	0	1	0	0	x	1	x	x	x	0	0	x	0	0
BNEZ_T	1011	2	1	0	0	x	1	x	x	x	0	0	x	1	0
BNEZ_NT	1011	0	1	0	0	x	1	x	x	x	0	0	x	1	0
JMP	1100	1	x	0	0	x	x	x	x	x	0	0	x	x	0
CALL	1101	1	x	0	1	x	x	x	x	x	0	0	x	x	1
RET	1110	3	x	0	0	x	x	x	x	x	0	0	x	x	0
Sv	1111	0	x	x	0	x	0	x	1	1	0	1	x	x	0
		IF	ID	ID	ID	EX	ID	EX	MEM	MEM	MEM	MEM	WB	EX	ID

Table 1: Control Signals

2.1.1 Details about signals(Rebal):

Signal Name	Signal Description	Cases
PCSrc	**Determines the next value of the PC**	**0: PC = PC + 2** **1: PC = {PC[15:10], Immediate}** **2: PC = PC + sign_extended (Imm)** **3: PC = r7**
RegSrc2	**Determines the second register**	**0: RB = Rs2** **1: RB = Rd**
RegWr1	**Determines the value to be written on the register**	**0: Indicates that no write operation should occur** **1: write the value from stage 4 or 5**
RegWr2	**Write the value in the R7 in RF**	**0: Indicates that no write operation should occur** **1: write the value if I have Call instruction**
ALUSrc	**Determines the first input of the ALU**	**0: B = BusB** **1: B = Immediate**
ExtOp	**Determine logical or signed extension**	**0: logical extension** **1: signed extension**
ALUOP	**Determine the operation for the ALU**	**0: AND** **1: ADD** **2: SUB**
AdressSig	**Determines the address for data stored in the memory**	**0: from register file** **1: ALU result**
DataInSig	**Determines if the data stored in the memory from the registerfile or from ALU**	**0: data from register file** **1: Immediate**
MemRd	**Enable read from memory**	**0: disable** **1: enable**
MemWr	**Enable write to memory**	**0: disable** **1: enable**
Wbdata	**Determine the return value from stage 5**	**0: data from ALU** **1: data from memory**
modeSig	**control signal used to specify different modes of operation for(Branches)**	**0: BusA** **1: R0**
RegDst	**Determines on which register to write**	**0: RW = Rd** **1: RW = R7**

Table 2: Details about signals

2.1.2 Expressions for Control Bits(Rebal):

$mode = LBs + BGTZ + BLTZ + BEQZ + BNEZ$

$ReqSrc2 = \sim(R\_Type)$

$RegWr1 = R - type + ADDI + ANDI + LW + Lbu + LBs$

$ReqWr2 = CALL$

$ALUSrc = R\_Type$

$ExtOP = BGT + BGTZ + BLT + BLTZ + BEQ + BEQZ + BNE + BNEZ + Lbu + LBs$

$ALUOp = R_{Type} + ADDI + ANDI + LW + Lbu + LBS + sw$

$AdressSig = Sv$

$DataInSig = Sv$

$MemRd = LW + Lbu + LBS$

$MemWr = SW + Sv$

$Wbdata = LW + Lbu + LBS$

$modeSig = BGTZ + BLTZ + BEQZ + BNEZ$

$RegDst = CALL$

2.1.3 Final Data path(Mohammad, Sondos, Rebal):

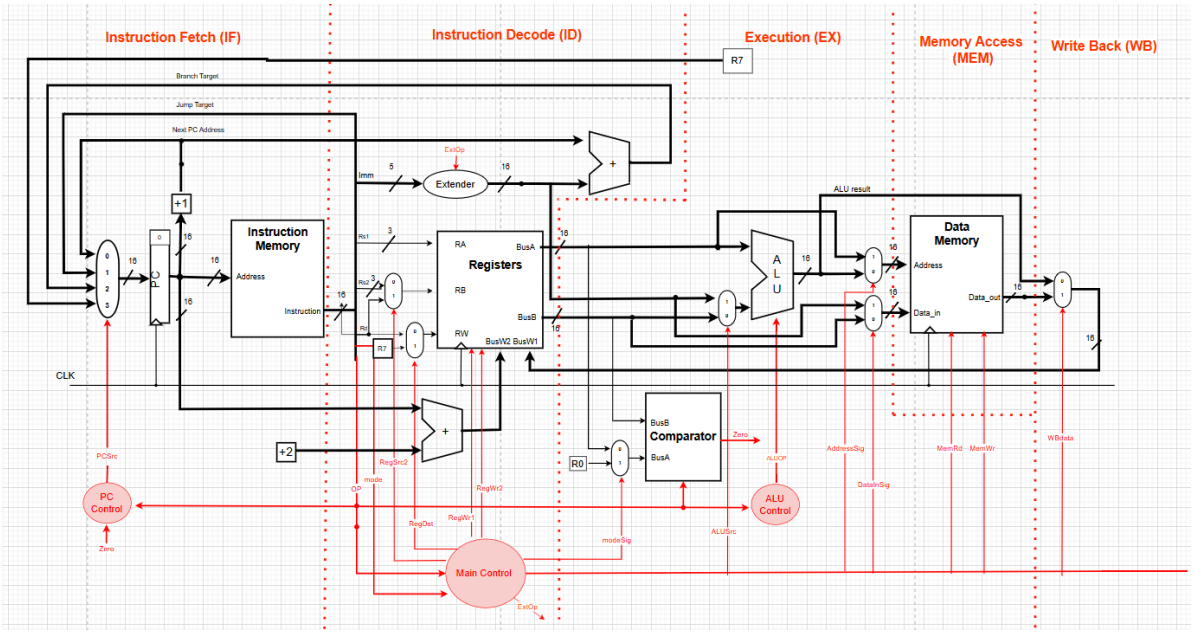

Figure 12: Final Data path

2.2 Modules:

2.2.1 Clock Generator (Mohammad):

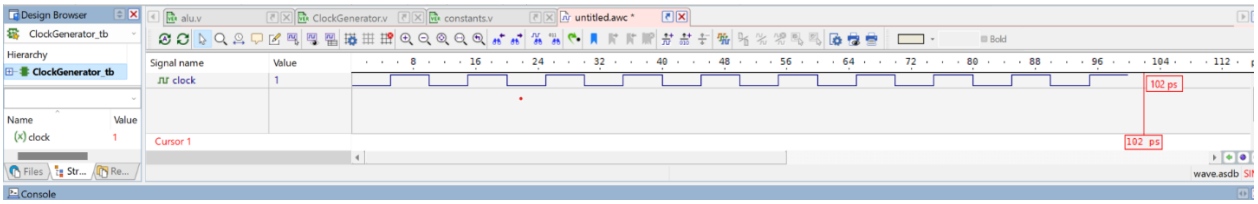

Figure 13: Clock Generator_tb

2.2.2 Instruction Fetch (Sondos):

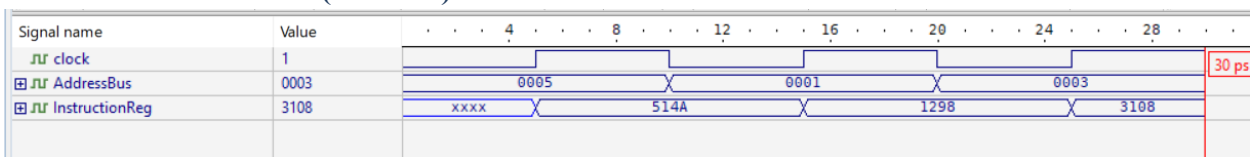

Figure 14: Instruction Fetch_tb

2.2.3 Data memory (Mohammad):

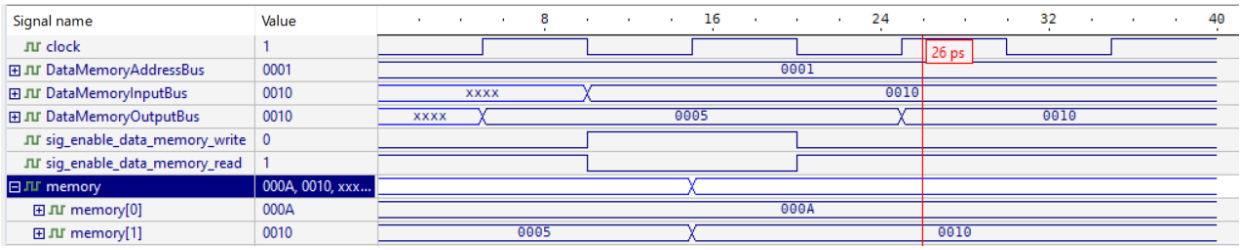

Figure 15: Data memory_tb

2.2.4 ALU(Rebal):

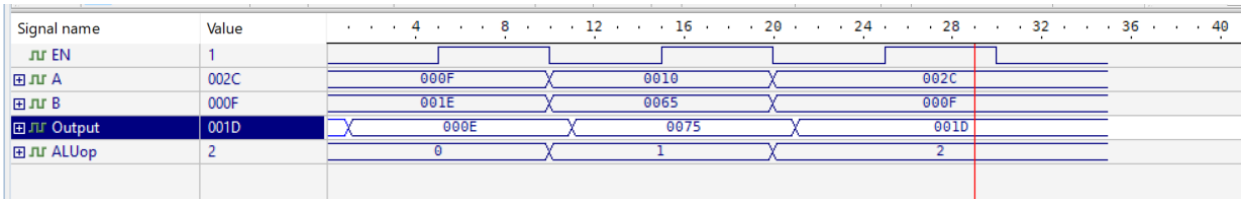

Figure 16: ALU_tb

2.2.5 Control Unit (Rebal):

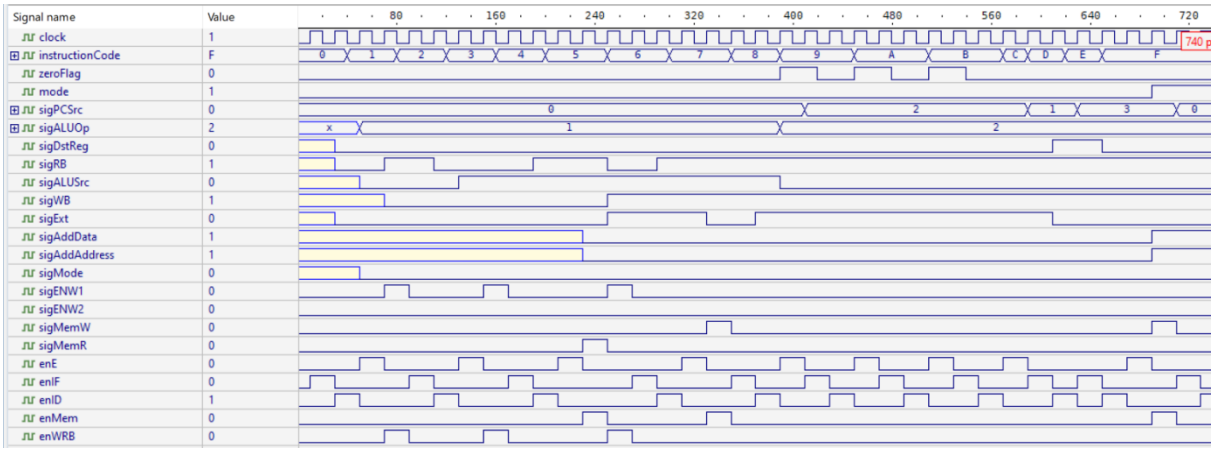

Figure 17: Control Unit_tb

2.2.6 Comparator(Sondos):

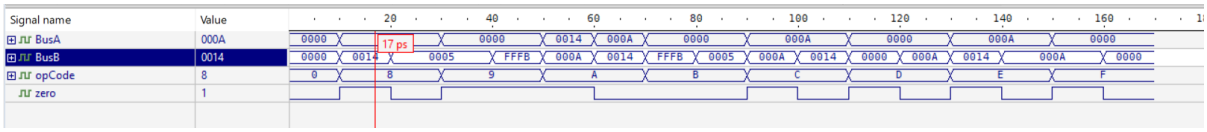

Figure 18: Comparator_tb

2.2.7 PC (Mohammad):

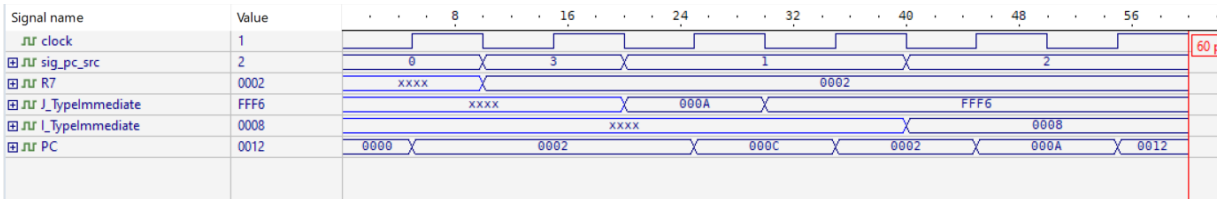

Figure 19: PC_tb

2.2.8 Register file (Sondos):

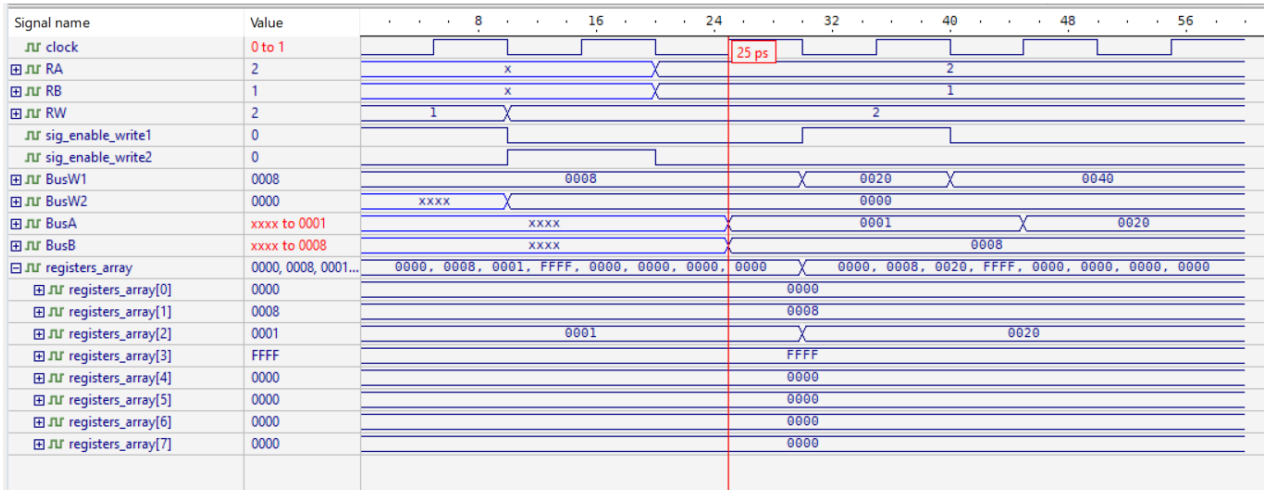

Figure 20: Register file _tb

2.2.9 Constants (Rebal):

```
parameter
// Kept these just in case
// Levels

    LOW = 1'b0,
    HIGH = 1'b1,

// Instruction codes
// 4-bit instruction code

    // R-Type Instructions

    AND = 4'b0000, // Reg(Rd) = Reg(Rs1) & Reg(Rs2)
    ADD = 4'b0001, // Reg(Rd) = Reg(Rs1) + Reg(Rs2)
    SUB = 4'b0010, // Reg(Rd) = Reg(Rs1) - Reg(Rs2)

    // I-Type Instructions

    ADDI= 4'b0011, // Reg(Rd) = Reg(Rs1) & Immediate5
    ANDI = 4'b0100, // Reg(Rd) = Reg(Rs1) + Immediate5
    LW   = 4'b0101, // Reg(Rd) = Mem(Reg(Rs1) + Imm_5)
    LBu  = 4'b0110, //Reg(Rd) = Mem(Reg(Rs1) + Imm_5)
    LBs  = 4'b0110,//   Reg(Rd) = Mem(Reg(Rs1) + Imm_5)
```

Figure 21: Constants_code

2.2.9 CPU (Sondos, Mohammad):

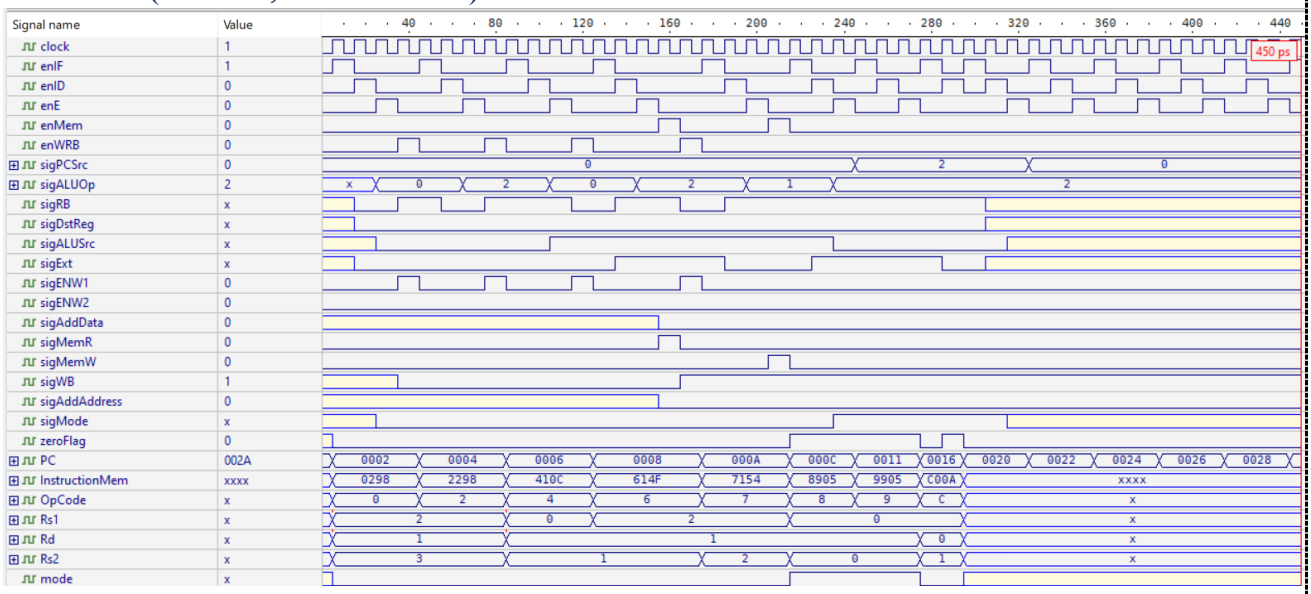

Figure 22: CPU _tb

3. Conclusion:

Designing a multi-cycle RISC processor involves a comprehensive understanding of computer architecture principles, including ISA, data path and control path design, control signal generation, and memory organization. Verification through simulation ensures the correctness and robustness of the processor. This project demonstrates the practical application of these theoretical concepts in developing a functional multi-cycle processor in Verilog.

4. Appendices:

4.1 The code for Constants:

```
parameter
// Kept these just in case
// Levels

   LOW = 1'b0,
   HIGH = 1'b1,

// Instruction codes
// 4-bit instruction code

    // R-Type Instructions

   AND = 4'b0000, // Reg(Rd) = Reg(Rs1) & Reg(Rs2)
   ADD = 4'b0001, // Reg(Rd) = Reg(Rs1) + Reg(Rs2)
   SUB = 4'b0010, // Reg(Rd) = Reg(Rs1) - Reg(Rs2)

   // I-Type Instructions

   ADDI = 4'b0011, // Reg(Rd) = Reg(Rs1) & Immediate5
   ANDI = 4'b0100, // Reg(Rd) = Reg(Rs1) + Immediate5
   LW   = 4'b0101, // Reg(Rd) = Mem(Reg(Rs1) + Imm_5)
   LBu  = 4'b0110, //Reg(Rd) = Mem(Reg(Rs1) + Imm_5)
    LBs  = 4'b0110,//          Reg(Rd) = Mem(Reg(Rs1) + Imm_5)

   SW   = 4'b0111, // Mem(Reg(Rs1) + Imm_14) = Reg(Rd)
    BGT  = 4'b1000, /*  if (Reg(Rd) > Reg(Rs1))
                                                Next PC = PC + sign_extended (Imm)
                                          else PC = PC + 2*/
```

```
BGTZ = 4'b1000, /* if (Reg(Rd) > Reg(0))
                                        Next PC = PC + sign_extended (Imm)
                                    else PC = PC + 2*/
BLT  = 4'b1001, /* if (Reg(Rd) < Reg(Rs1))
                                        Next PC = PC + sign_extended (Imm)
                                    else PC = PC + 2*/
BLTZ = 4'b1001,/* if (Reg(Rd) < Reg(R0))
                                        Next PC = PC + sign_extended (Imm)
                                    else PC = PC + 2*/

BEQ  = 4'b1010, /* if (Reg(Rd) == Reg(Rs1))
                                        Next PC = PC + sign_extended (Imm)
                                    else PC = PC + 2*/
BEQZ = 4'b1010,/*if (Reg(Rd) == Reg(R0))
                                        Next PC = PC + sign_extended (Imm)
                                else PC = PC + 2*/
BNE  = 4'b1011, /* if (Reg(Rd) != Reg(Rs1))
                                        Next PC = PC + sign_extended (Imm) */
BNEZ = 4'b1011,/* if (Reg(Rd) != Reg(Rs1))
                                        Next PC = PC + sign_extended (Imm)
                                    else PC = PC + 2*/

// J-Type Instructions

JMP   = 4'b1100, // Next PC = {PC[15:10], Immediate}
CALL  = 4'b1101, /* Next PC = {PC[15:10], Immediate}
                                    PC + 4 is saved on r15 */
RET = 4'b1110, // Next PC = r7

// S-Type Instructions
```

```
    Sv  = 4'b1111, // M[rs] = imm


// *Signals*

// PC src
// 2-bit select to determine next PC value

    pcDefault = 2'b00, // PC = PC + 1
    pcImm = 2'b01, // jump address
    pcSgnImm = 2'b10, // branch target address
    pcRET = 2'b11, // PC = R7(RET)


// Dst Reg
// 1-bit select to determine on which register to write

    //R7 = 1'b0, // RW = R7
    Rd = 1'b1, // RW = Rd

// RB
// 1-bit select to determine second register

    Rs2 = 1'b0, // RB = Rs2
    Rsd = 1'b1, // RB = Rd
    //Rd = 1'b1, // RW = Rd

// En W
// 1-bit select to E/D write on registers
```

```
    WD = 1'b0, // Write enabled
    WE = 1'b1, // Write disabled


// ALU src
// 1-bit source select to determine first input of the ALU

    Imm = 1'b0,      // B = Immediate
    BusB = 1'b1, // B = BusB

// ALU op
// 2-bit select to determine operation of ALU

    ALU_AND = 2'b00,
    ALU_ADD = 2'b01,
    ALU_SUB = 2'b10,

// Mem R
// 1-bit select to enable read from memory

    MRD = 1'b0,     // Memory read disabled
    MRE = 1'b1, // Memory read enabled

// Mem W
// 1-bit select to enable write to memory

    MWD = 1'b0,    // Memory write disabled
    MWE = 1'b1, // Memory write enabled
```

```
// WB
// 1-bit select to determine return value from stage 5

    WBALU = 1'b0, // Data from ALU
    WBMem = 1'b1, // Data from memory

// ext
// 1-bit select to determine logical or signed extension

    logExt = 1'b0, // Logical extension
    sgnExt = 1'b1, // Signed extension

//Data signal



// stack/mem
// 1-bit select to store address of data in memory or push on stack

    //WBALU = 1'b0, // Data from ALU
    BusA = 1'b1, // Calculate address of the memory

// address/data
// 1-bit select to determine source of data to be stored in memory

    //Imm = 1'b0,    // B = Immediate
    //BusB = 1'b1, // B = BusB

// 8 registers

   R0 = 3'd0, // zero register
```

```
    R1 = 3'd1, // general purpose register
    R2 = 3'd2, // general purpose register
    R3 = 3'd3, // general purpose register
    R4 = 3'd4, // general purpose register
    R5 = 3'd5, // general purpose register
    R6 = 3'd6, // general purpose register
    R7 = 3'd7; // general purpose register
```

4.2 The code for comparator:

```
module comparator (
    input [15:0] BusA,    // rs or R0
    input [15:0] BusB,    // rd
    input [3:0] opCode,   // operation code
    output reg zero       // output: 1 if condition is true (branch taken), 0 otherwise
);

always @(*) begin
    case (opCode)
        4'b1000: zero = (BusB > BusA) ? 1 : 0;  // BGT
        4'b1001: zero = (BusB > 16'd0) ? 1 : 0; // BGTZ
        4'b1010: zero = (BusB < BusA) ? 1 : 0;  // BLT
        4'b1011: zero = (BusB < 16'd0) ? 1 : 0; // BLTZ
        4'b1100: zero = (BusB == BusA) ? 1 : 0; // BEQ
        4'b1101: zero = (BusB == 16'd0) ? 1 : 0; // BEQZ
        4'b1110: zero = (BusB != BusA) ? 1 : 0; // BNE
        4'b1111: zero = (BusB != 16'd0) ? 1 : 0; // BNEZ
        default: zero = 0;                  // Default case: no branch
    endcase
end

endmodule
```

4.3 The code for Control Unit:

```
`include "constants.v"


module controlUnit(
    clock,


     // signal outputs
    sigPCSrc,
    sigDstReg,
    sigRB,
    sigENW1,
    sigENW2,
    sigALUSrc,
    sigALUOp,
    sigExt,
    sigAddData,
    sigMemR,
    sigMemW,
    sigWB,
    sigAddAddress,
    sigMode,


    // Enable signals to be outputted by the CU
    enIF, // enable instruction fetch
    enID, // enable instruction decode
    enE,  // enable execute
    enMem,// enable memory
    enWRB,
```

```
    // inputs
    zeroFlag,
    mode,

    // Opcode to determine signals
    instructionCode
);

  // ----------------- INPUTS -----------------

  // clock
input wire clock;

// mode
input wire mode;

  // zero flag and negative Flag
input wire zeroFlag;

  // function code
input wire [3:0] instructionCode;

  // ----------------- OUTPUTS -----------------

output reg [3:0] sigPCSrc = pcDefault;
output reg [1:0] sigALUOp;
output reg sigDstReg, sigRB, sigALUSrc, sigWB, sigExt, sigAddData, sigAddAddress, sigMode;

output reg  sigENW1 = LOW,
            sigENW2 = LOW,
```

```
            sigMemW = LOW,
            sigMemR = LOW;

output reg  enE = LOW,
            enIF = LOW,
            enID = LOW,
            enMem = LOW,
            enWRB = LOW;

   // Code to determine stages and manage stage paths
`define STG_FTCH 3'b000
`define STG_DCDE 3'b001
`define STG_EXEC 3'b010
`define STG_MEM 3'b011
`define STG_WRB 3'b100
`define STG_INIT 3'b101

  reg [2:0] currentStage = `STG_INIT;
reg [2:0] nextStage = `STG_FTCH;


  always@(posedge clock) begin

    currentStage = nextStage;

end

  always@(posedge clock) begin

    case (currentStage)
```

```
        `STG_INIT: begin
            enIF = LOW;
            nextStage <= `STG_FTCH;
             end

      `STG_FTCH: begin
            // Disable all previous stages leading up to IF
            enID = LOW;
            enE = LOW;
                        enMem = LOW;
                        enWRB = LOW;

            // Disable signals
            sigENW1 = LOW;
                  sigENW2 = LOW;
            sigMemW = LOW;
            sigMemR = LOW;

            // Enable IF
            enIF = HIGH; // Fetch after finding PC src

            // Determine next stage
            nextStage <= `STG_DCDE;

                  // Determine next PC through testing of opcodes
            if (instructionCode == BGT && zeroFlag || instructionCode == BLT && zeroFlag ||
instructionCode == BEQ && zeroFlag || instructionCode == BNE && zeroFlag ||instructionCode ==
BNEZ && zeroFlag || BGTZ && zeroFlag || BLTZ && zeroFlag ) begin

               sigPCSrc = pcSgnImm;
```

```
            end else if (instructionCode == JMP || instructionCode == CALL) begin

               sigPCSrc = pcImm;

            end else if (instructionCode == RET) begin

               sigPCSrc = pcRET;                    //R7

            end else begin

               sigPCSrc = pcDefault;

            end

         end


         `STG_DCDE: begin
            // Disable all previous stages leading up to ID
            enIF = LOW;

            // Enable IF
            enID = HIGH;

            // Next stage is determined by opcode
                              if (instructionCode == CALL || instructionCode == RET ||instructionCode
== JMP) begin
                                       nextStage <= `STG_FTCH;
                              end
```

```
                else begin
                        nextStage <= `STG_EXEC;
                end

                sigRB =(instructionCode == AND || instructionCode == ADD ||
instructionCode == SUB) ? LOW : HIGH;
                sigENW1= LOW;
              sigENW2 =LOW;

    sigExt=(instructionCode==BNEZ||instructionCode==BNE||instructionCode==BEQZ||instructionCo
de==BEQ||instructionCode==BLTZ||instructionCode==BLT||instructionCode==BGTZ||instructionCode
==BGT||instructionCode==LBs||instructionCode==LBu)  ? HIGH : LOW;
                sigDstReg =(instructionCode == CALL)? HIGH : LOW;



        end

            `STG_EXEC: begin


        // Disable all previous stages leading up to execute stage
        enID = LOW;
                //sigENW1 = !sigENW1;
        // Enable execute stage
        enE = HIGH;

                sigALUSrc=(instructionCode == ADDI || instructionCode == ANDI ||
instructionCode == LW|| instructionCode == LBu || instructionCode == LBs|| instructionCode == SW)?
HIGH : LOW;

                sigMode=(instructionCode==BGTZ &&
mode==1||instructionCode==BLTZ && mode==1||instructionCode==BEQZ &&
```

```
mode==1||instructionCode==BNEZ && mode==1 ||instructionCode==LBs && mode==1)     ? HIGH :
LOW;



        // Next stage is determined by opcode

        if  (instructionCode == LW || instructionCode == LBu || instructionCode == SW ||
instructionCode == LBs|| instructionCode ==Sv ) begin


            nextStage <= `STG_MEM;


        end else if (instructionCode == AND || instructionCode == ADD || instructionCode == SUB ||
instructionCode == ANDI || instructionCode == ADDI) begin


            nextStage <= `STG_WRB;


        end else begin


            nextStage <= `STG_FTCH;


        end


        // Set ALUOp signal based on the opcodes

        if (instructionCode == AND || instructionCode == ANDI) begin


            sigALUOp = ALU_AND;


        end else if (instructionCode == ADD || instructionCode == ADDI || instructionCode == LW ||
instructionCode == SW) begin


            sigALUOp = ALU_ADD;
```

```
        end else begin

            sigALUOp = ALU_SUB;

        end
                        sigENW1 = LOW;
        sigENW2 = LOW;
    end

            `STG_MEM: begin
                        enMem = HIGH;
        // Disable all previous stages leading up to memory stage
                        enE = LOW;
                        enID = LOW;

        // Enable memory stage


        // Next stage determined by opcodes
        nextStage <= (instructionCode == LW ||instructionCode == LBu ||instructionCode == LBs) ?
`STG_WRB : `STG_FTCH;

        // Memory write is determined by the SW instruction
                        sigMemW         =(instructionCode==Sv||instructionCode==SW)? HIGH :
LOW;

        // Memory Read is determined by Load instructions
         sigMemR=(instructionCode==LBs||instructionCode==LBu||instructionCode==LW)  ? HIGH
: LOW;

        // Signals determined by opcodes
```

```
                    sigAddData=(instructionCode==Sv)? HIGH : LOW;

                        sigAddAddress=(instructionCode==Sv)? HIGH : LOW;

    sigExt=(instructionCode==BNEZ||instructionCode==BNE||instructionCode==BEQZ||instructionCo
de==BEQ||instructionCode==BLTZ||instructionCode==BLT||instructionCode==BGTZ||instructionCode
==BGT||instructionCode==LBs||instructionCode==LBu)  ? HIGH : LOW;

        end

            `STG_WRB: begin
                    enWRB=HIGH;
        // Disable all previous stages leading up to WRB
        sigMemW = LOW;
        sigMemR = LOW;
        enE = LOW;
                    enMem = LOW;

        // Enable WRB stage
                    sigRB =~(instructionCode == AND || instructionCode == ADD ||
instructionCode == SUB) ?LOW : HIGH;

        // Next stage following WRB is IF
        nextStage <= `STG_FTCH;

        // Enable writing to register filw
                    sigENW1=(instructionCode == AND || instructionCode == ADD ||
instructionCode ==
SUB||instructionCode==LBs||instructionCode==LBu||instructionCode==LW||instructionCode==ADDI||i
nstructionCode==ANDI)? HIGH : LOW;
```

```
                                sigENW2=(instructionCode ==CALL)? HIGH : LOW;
            // Determine the register to write to
                                sigWB
=(instructionCode==LBs||instructionCode==LBu||instructionCode==LW)  ? HIGH : LOW;
            // Set to 1 for all instructions
                                sigDstReg =(instructionCode == CALL)? HIGH : LOW;


    sigExt=(instructionCode==BNEZ||instructionCode==BNE||instructionCode==BEQZ||instructionCo
de==BEQ||instructionCode==BLTZ||instructionCode==BLT||instructionCode==BGTZ||instructionCode
==BGT||instructionCode==LBs||instructionCode==LBu)  ? HIGH : LOW;


          end


      endcase
   end



endmodule
```

4.4 The code for alu:

```
 `include "constants.v"


 module ALU(A, B, Output, ALUop,EN);


     // ----------------- SIGNALS & INPUTS -----------------


    // chip select for ALU operation
     input wire [3:0] ALUop;


     //Enable flag
```

```
input wire EN;

// operands
input wire [15:0] A, B;

// ----------------- OUTPUTS -----------------

output reg        [15:0]   Output;


// ----------------- LOGIC -----------------


always @(EN) begin
        //#1 // To wait for ALU source mux to select operands
        case (ALUop)
                AND:  Output <= A & B;
                ANDI:  Output <= A & B;
                ADD:  Output <= A + B;
                ADDI:  Output <= A + B;
                LW:  Output <= A + B;
                LBu:  Output <= A + B;
                LBs:  Output <= A + B;
                SW:  Output <= A + B;
                SUB:  Output <= A - B;
                default: Output <= 0;
        endcase
end

initial begin
```

```
            if (EN == LOW)begin
                Output = 16'd0;
            end
    end

 endmodule
```

4.5 The code for ClockGenerator:

```
// generates clock square wave with 10ns period

module ClockGenerator (
   clock
);

initial begin
   $display("(%0t) > initializing clock generator ...", $time);
end

output reg clock=0; // starting LOW is important for first instruction fetch

always #5 begin
   clock=~clock;
end


endmodule
```

4.6 The code for data Memory:

```
// data memory with 256 cells of 16 bits each
module dataMemory(clock, AddressBus, InputBus, OutputBus, sig_enable_write,
sig_enable_read,memory);

   // ----------------- SIGNALS -----------------
```

```
// clock
input wire clock;

// enable write signal
  input wire sig_enable_write;

// enable write signal
  input wire sig_enable_read;

// ----------------- INPUTS -----------------

// address bus
input wire [15:0] AddressBus;

// MBR - in
input wire [15:0] InputBus;

// ----------------- OUTPUTS -----------------

// MBR - out
output reg [15:0] OutputBus;

// ----------------- INTERNALS -----------------

// memory
output reg [15:0] memory [0:255];

// ----------------- LOGIC -----------------
```

```
    // read instruction at positive edge of clock
    always  @(posedge clock) begin
        if (sig_enable_read) begin
            OutputBus <= memory[AddressBus];
        end else if (sig_enable_write) begin
            memory[AddressBus] <= InputBus;
        end
    end

    // ----------------- INITIALIZATION -----------------

    initial begin

        // store some initial data
        memory[0] = 16'd10;
        memory[1] = 16'd5;
                memory[16] = 16'd9;

    end


endmodule
```

4.7 The code for instruction Memory:

```
module instructionMemory(clock, AddressBus, InstructionReg);

    // ----------------- INPUTS -----------------

    // clock
    input wire clock;
```

```
    // address bus
    input wire [15:0] AddressBus;

    // ----------------- OUTPUTS -----------------

    // instruction register
    output reg [15:0] InstructionReg;



    // ----------------- INTERNALS -----------------

    // instruction memory
    reg [15:0] instruction_memory [0:255]; // Each instruction is 16 bits, memory size can be adjusted as
needed

    // ----------------- LOGIC -----------------

    always @(posedge clock) begin
        InstructionReg <= instruction_memory[AddressBus];
    end

    // ----------------- INITIALIZATION -----------------

    initial begin

                // instruction formats:
        // R-Type Instruction Format
        // Opcode_4, Rd_3, Rs1_3, Rs2_3, Unused_3

        // I-Type Instruction Format
```

```
// Opcode_4, m_1, Rd_3, Rs1_3, Immediate_5

// J-Type Instruction Format
// Opcode_4, Jump_Offset_12

// S-Type Instruction Format
// Opcode_4, Rs_3, Immediate_9
        /*****************************************************/

// R-Type Instructions
// AND R1, R2, R3
instruction_memory[0] = {4'b0000, 3'b001, 3'b010, 3'b011, 3'b000}; // AND R1, R2, R3
// ADD R1, R2, R3
instruction_memory[1] = {4'b0001, 3'b001, 3'b010, 3'b011, 3'b000}; // ADD R1, R2, R3
// SUB R1, R2, R3
instruction_memory[2] = {4'b0010, 3'b001, 3'b010, 3'b011, 3'b000}; // SUB R1, R2, R3

// I-Type Instructions
// ADDI R1, R0, 8
instruction_memory[3] = {4'b0011, 1'b0, 3'b001, 3'b000, 5'd8}; // ADDI R1, R0, 8
// ANDI R1, R0, 12
instruction_memory[4] = {4'b0100, 1'b0, 3'b001, 3'b000, 5'd12}; // ANDI R1, R0, 12
// LW R1, 10(R2)
instruction_memory[5] = {4'b0101, 1'b0, 3'b001, 3'b010, 5'd10}; // LW R1, 10(R2)
// LBu R1, 15(R2)
instruction_memory[6] = {4'b0110, 1'b0, 3'b001, 3'b010, 5'd15}; // LBu R1, 15(R2)
// LBs R1, 15(R2)
instruction_memory[7] = {4'b0110, 1'b1, 3'b001, 3'b010, 5'd15}; // LBs R1, 15(R2)
// SW R1, 20(R2)
instruction_memory[8] = {4'b0111, 1'b0, 3'b001, 3'b010, 5'd20}; // SW R1, 20(R2)
```

```
    // BGT R1, R2, 5
    instruction_memory[9] = {4'b1000, 1'b0, 3'b001, 3'b010, 5'd5}; // BGT R1, R2, 5
    // BGTZ R1, 5
    instruction_memory[10] = {4'b1000, 1'b1, 3'b001, 3'b000, 5'd5}; // BGTZ R1, 5
    // BLT R1, R2, 5
    instruction_memory[11] = {4'b1001, 1'b0, 3'b001, 3'b010, 5'd5}; // BLT R1, R2, 5
    // BLTZ R1, 5
    instruction_memory[12] = {4'b1001, 1'b1, 3'b001, 3'b000, 5'd5}; // BLTZ R1, 5
    // BEQ R1, R2, 5
    instruction_memory[13] = {4'b1010, 1'b0, 3'b001, 3'b010, 5'd5}; // BEQ R1, R2, 5
    // BEQZ R1, 5
    instruction_memory[14] = {4'b1010, 1'b1, 3'b001, 3'b000, 5'd5}; // BEQZ R1, 5
    // BNE R1, R2, 5
    instruction_memory[15] = {4'b1011, 1'b0, 3'b001, 3'b010, 5'd5}; // BNE R1, R2, 5
    // BNEZ R1, 5
    instruction_memory[16] = {4'b1011, 1'b1, 3'b001, 3'b000, 5'd5}; // BNEZ R1, 5

    // J-Type Instructions
    // JMP 10
    instruction_memory[17] = {4'b1100, 12'd10}; // JMP 10
    // CALL 20
    instruction_memory[18] = {4'b1101, 12'd20}; // CALL 20
    // RET
    instruction_memory[19] = {4'b1110, 12'b0}; // RET

    // S-Type Instructions
    // Sv R2, 255
    instruction_memory[20] = {4'b1111, 3'b010, 9'd255}; // Sv R2, 255
end
```

```
endmodule
```

4.8 The code for register File:

```
module registerFile(clock, RA, RB, RW, sig_enable_write1,sig_enable_write2, BusW1, BusW2,
BusA, BusB,registers_array);

    input wire clock;

    input wire sig_enable_write1 , sig_enable_write2;

    input wire [2:0] RA, RB, RW;

    output reg [15:0] BusA, BusB;



    input wire [15:0] BusW1,BusW2;

     output reg [15:0] registers_array [0:7];

    always @(posedge clock) begin
            if(sig_enable_write1==0 && sig_enable_write2==0) begin
                    BusB = registers_array[RB];
                    BusA = registers_array[RA];

            end
    end

    always @(posedge (sig_enable_write1 ||  sig_enable_write2) ) begin

            if ( RW != 3'b0 && sig_enable_write1) begin
```

```
                registers_array[RW] = BusW1;
        end
        if ( RW != 3'b0 && sig_enable_write2) begin
                registers_array[RW] = BusW2;
        end

    end

    initial begin
        // Initialize registers to be zeros
        registers_array[0] <= 16'h0000;
        registers_array[1] <= 16'h4444;
        registers_array[2] <= 16'h0001;
        registers_array[3] <= 16'hFFFF;
        registers_array[4] <= 16'h0000;
        registers_array[5] <= 16'h0000;
        registers_array[6] <= 16'h0000;
        registers_array[7] <= 16'h0000;

    end
endmodule
```

4.9 The code for riscProcessor:

```
`include "constants.v"
module riscProcessor();

   initial begin
        #0
      $display("(%0t) > initializing processor ...", $time);

      #450 $finish;
```

```
end



// clock generator wires/registers
 wire clock;
 wire enIF, enID, enE, enMem, enWRB;

// ---------- Control Unit ----------

 wire [3:0] sigPCSrc;
wire [1:0] sigALUOp;
 wire sigRB, sigDstReg, sigALUSrc, sigExt, sigENW1,sigENW2,  sigAddData, sigMemR, sigMemW, sigWB,sigAddAddress,sigMode;
 wire zeroFlag;



// ---------- Instrution Memory ----------

// instruction memory wires/registers
wire [15:0] PC; // output of PC Module input to instruction memory
reg [15:0] InstructionMem; // output if instruction memory, input to other modules

// Instruction Parts
wire [3:0] OpCode; // function code

// R-Type
wire [2:0] Rs1, Rd, Rs2; // register selection
 wire  mode;
```

```
    // ----------------- Register File -----------------
    wire [2:0] RA, RB, RW;
    wire [15:0] BusA, BusB, BusW1 , BusW2,BusAC;

    // ----------------- ALU -----------------
    wire [15:0] A, B;
    wire [15:0] ALURes;


    // ----------------- Data Memory -----------------
    wire [15:0]        AddressBus, InputBus,OutputBus;
    assign AddressBus = (sigAddAddress == LOW) ? ALURes : BusA;
    assign InputBus = (sigAddData == LOW ) ? BusB :  extendedImm5;

   // ----------------- Assignment -----------------

   // Function Code
   assign OpCode = InstructionMem[15:12];

   // R-Type

   assign Rd = (OpCode==ADD ||OpCode== AND ||OpCode==SUB
||OpCode==Sv)?InstructionMem[11:9] :InstructionMem[10:8] ;
   assign Rs1 = (OpCode==ADD ||OpCode== AND ||OpCode==SUB ||OpCode==Sv)?
InstructionMem[8:6] : InstructionMem[7:5];
   assign Rs2 = InstructionMem[5:3];

   // I-Type
   wire signed [4:0] Imm5;
   assign Imm5 = InstructionMem[4:0];
    assign mode = InstructionMem[11];
```

```
// J-Type
wire [11:0]        Imm12;
assign Imm12 = InstructionMem[11:0];


// S-Type
wire [8:0]         Imm9;
assign Imm9 = InstructionMem[8:0];


// ----------------- Register File -----------------
wire [15:0] WBOutput;
wire EnReg;
assign RA = Rs1;
assign RB = (sigRB == LOW) ? Rs2 : Rd;
assign RW = (sigDstReg == LOW) ? Rd : 3'b111;
assign EnReg = (enID || enWRB);
assign BusW1 =  WBOutput;
assign BusW2 =  (PC + 2'd2);


// ----------------- ALU -----------------
wire [15:0] extendedImm5 ,extendedImm12 ;
wire [15:0] signExtendedImm, zeroExtendedImm;
assign signExtendedImm = {{11{Imm5[4]}}, Imm5};
assign zeroExtendedImm = {{11{1'b0}}, Imm5};
assign extendedImm5 = (sigExt == HIGH) ?  signExtendedImm: zeroExtendedImm;
assign A = BusA;
assign B = (sigALUSrc == LOW) ? BusB : extendedImm5;
```

```
 assign BusAC = (sigMode==LOW) ? BusA : 3'b000;
assign extendedImm12 = {{4{Imm12[11]}}, Imm12};

 ClockGenerator clock_generator(.clock(clock) );

// ------------------------------------------------------------------
// ----------------- Control Unit -----------------
// ------------------------------------------------------------------


controlUnit cu(
    clock,

     // signal outputs
    sigPCSrc,
    sigDstReg,
    sigRB,
    sigENW1,
    sigENW2,
    sigALUSrc,
    sigALUOp,
    sigExt,
    sigAddData,
    sigMemR,
    sigMemW,
    sigWB,
    sigAddAddress,
    sigMode,

    // Enable signals to be outputted by the CU
```

```
    enIF, // enable instruction fetch
    enID, // enable instruction decode
    enE,  // enable execute
    enMem,// enable memory
    enWRB,

    // inputs
    zeroFlag,
    mode,

    // Opcode to determine signals
    OpCode
);



// ---------------------------------------------------------------
// ----------------- PC Module -----------------
// ---------------------------------------------------------------

pcModule pcMod(enIF, PC, extendedImm5, extendedImm12,r7, sigPCSrc);

 // ---------------------------------------------------------------
// ----------------- Instruction Memory -----------------
// ---------------------------------------------------------------

instructionMemory insMem(enIF, PC, InstructionMem);

// ---------------------------------------------------------------
// ----------------- Register File -----------------
// ---------------------------------------------------------------
```

```
    wire [15:0] registers_array [0:7];
    wire [15:0] r7 = registers_array[7];
   registerFile regFile(clock, RA, RB, RW, sigENW1,sigENW2, BusW1, BusW2, BusA,
BusB,registers_array);

   // ----------------------------------------------------------------
   // ----------------- ALU -----------------
   // ----------------------------------------------------------------

   ALU alu(A, B, ALURes, OpCode, enE);



    // ----------------------------------------------------------------
   // ----------------- Comparator -----------------
   // ----------------------------------------------------------------
    comparator comp(BusAC,BusB,OpCode,zeroFlag);

   // ----------------------------------------------------------------
   // ----------------- Data Memory -----------------
   // ----------------------------------------------------------------
    wire [15:0] memory [0:255];
   dataMemory dataMem(enMem, AddressBus, InputBus, OutputBus, sigMemW, sigMemR,memory);

   // ----------------------------------------------------------------
   // ----------------- Write Back -----------------
   // ----------------------------------------------------------------

   assign WBOutput = (sigWB == 0) ? ALURes : OutputBus;
endmodule
```