PROJECT REPORT

Contents

Title of the Project:

Design, Simulation & Testing of a 5-Stage Pipelined MIPS processor using Verilog HDL

Introduction:

The MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC architecture developed by MIPS Technologies. It was developed in the late 1980s. The MIPS instruction set is used by NEC,Nintendo, Silicon Graphics, Sony PlayStation, PlayStation handheld System and several other computer manufacturers. It has fixed-length 32-bit instructions and 32-bit general-purpose registers.

In this project, we will design a 5-stage MIPS pipelined processor illustrated in the Patterson & Hennessy book Computer Organization & Design: The Hardware Software Interface (chapter 6).

PHASE 1:

Phase I: Build a Single-Cycle processor

REQUIRED MODULES :

- **ALU 32-bit ALU,**
- **Extender sign/zero extender,**
- **Multiplexers m16x2, m32x2, m32x3, m32x5, m5x2: multiplexers with variouswidths and number of inputs,**
- **Register File regfile: register file,**
- **Shifter Arithmetic logical, multi-bit shifting unit.**
- **Instruction Memory 2K-words deep, 32-bit wide, (using "SRAM"), initial content will be loaded from within the Verilog code.**
- **Data Memory 4K-words deep, 32-bit wide, no initial contents.**

EXPLAINATION :

ALU & ALU CONTROL :

ALU working:

It tasked with executing arithmetic and logical operations on input data. Its operation begins by receiving data from processor registers or memory, followed by the determination of the operation to perform based on the instruction opcode. Utilizing combinational logic circuits for arithmetic tasks and bitwise operations for logical tasks, the ALU executes the selected operation, generating the result which could be an arithmetic calculation, logical outcome, or shifted data.

ALU CONTROL working:

ALU control is the process by which a processor determines the appropriate operation for the Arithmetic Logic Unit (ALU) to perform based on the opcode of the instruction being executed. The ALU control unit decodes the instruction opcode, generates control signals accordingly, and synchronizes them with the processor's clock signal. This process ensures that the ALU executes the correct arithmetic or logical operation specified by the instruction.

SHIFTER: Also included in alu that performs task of shifting.

Code:

```
module ALU (  input [31:0] A,   //intializing the  required input and and
output
    input [31:0] B,
    input [3:0] ALUControl,
    output reg [31:0] ALUResult,
    output Zero
);

    assign Zero = (ALUResult == 0); // intially assigning zero to the zero flag

    always @(*) begin
        case (ALUControl)
            4'b0010: ALUResult = A + B;   // ADD
            4'b0110: ALUResult = A - B;   // SUB
            4'b0000: ALUResult = A & B;   // AND
            4'b0001: ALUResult = A | B;   // OR
            4'b0111: ALUResult = (A < B) ? 1 : 0; // SLT
```

```
            default: ALUResult = 0; // default result
        endcase
    end
endmodule
```

ALU CONTROL:

```
module ALUControl (
    input [1:0] ALUOp,
    input [5:0] Funct,
    output reg [3:0] ALUControl
);

    always @(*) begin
        case (ALUOp)
            2'b00: ALUControl = 4'b0010; // lw/sw - add
            2'b01: ALUControl = 4'b0110; // beq - sub
            2'b10: begin
                case (Funct)
                    6'b100000: ALUControl = 4'b0010; // add
                    6'b100010: ALUControl = 4'b0110; // sub
                    6'b100100: ALUControl = 4'b0000; // and
                    6'b100101: ALUControl = 4'b0001; // or
                    6'b101010: ALUControl = 4'b0111; // slt
                    default: ALUControl = 4'b0000;   // default
                endcase
            end
            default: ALUControl = 4'b0000; // default
        endcase
    end
endmodule
```

EXTENDER :

The sign/zero extender is a hardware component in computer architecture that extends the length of binary numbers, typically from 16 to 32 bits. It preserves the sign bit when extending signed numbers, ensuring proper arithmetic operations. For unsigned numbers, it appends zeroes to maintain consistency in calculations. This extender plays a crucial role in maintaining data integrity and accuracy within computer systems

CODE

```
module SignExtender (
    input [15:0] Inst,
    output [31:0] SignImm
);
    assign SignImm = {{16{Inst[15]}}, Inst}; //checking the most significant
bit
endmodule
```

Register reg32: 32-bit REGISTER FILE

The reg32 is a 32-bit register used for storing data .The regfile refers to a register file, a collection of registers typically used for temporary storage of data during computational tasks, offering fast access and manipulation capabilities essential for efficient processing.

CODE

```
module RegisterFile ( // initialing the input and ouputs of the module
    input clk,
    input RegWrite,
    input [4:0] ReadRegister1,
    input [4:0] ReadRegister2,
    input [4:0] WriteRegister,
    input [31:0] WriteData,
    output [31:0] ReadData1,
    output [31:0] ReadData2
);

    reg [31:0] registers [31:0];
    initial begin
    registers[0] = 32'd0;// assigning each of the registers a value of 32 bit
    registers[1] = 32'd4;
    registers[2] = 32'd5;
    registers[3] = 32'd3;
    registers[4] = 32'd1;
    registers[5] = 32'd6;
    registers[6] = 32'd7;
    registers[7] = 32'd2;
    registers[8] = 32'd3;
```

```
    registers[9] = 32'd2;
    registers[10] = 32'd5;
    registers[11] = 32'd4;
    registers[12] = 32'd6;
    registers[13] = 32'd5;
    registers[14] = 32'd4;
    registers[15] = 32'd3;
    registers[16] = 32'd7;
    registers[17] = 32'd3;
    registers[18] = 32'd2;
    registers[19] = 32'd3;
    registers[20] = 32'd1;
    registers[21] = 32'd5;
    registers[22] = 32'd6;
    registers[23] = 32'd7;
    registers[24] = 32'd8;
    registers[25] = 32'd3;
    registers[26] = 32'd8;
    registers[27] = 32'd7;
    registers[28] = 32'd1;
    registers[29] = 32'd5;
    registers[30] = 32'd4;
    registers[31] = 32'd2;
    end

    assign ReadData1 = registers[ReadRegister1]; // Assign data to reg1
    assign ReadData2 = registers[ReadRegister2]; // Assign data to reg1


    always @(posedge clk) begin
        if (RegWrite) begin // if RegWrite is 1
            registers[WriteRegister] <= WriteData; //writw the data to the
register address mentioned
        end
    end
endmodule
```

Instruction Memory:

Instruction memory serves as the source of instructions for the processor. The pipeline stages include instruction fetch, decode, execute, memory access, and write-back. The instruction memory stage fetches the next instruction from memory, aligning it with the pipeline stages for efficient processing. This process enables parallel execution of instructions, enhancing performance by overlapping tasks in different stages of the pipeline.

Code:

```
module InstructionMemory (
    input [31:0] Address,
    output [31:0] Instruction
);

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

    initial begin

            memory[0] = 32'h00221820;  // add x3, x2, x1
            memory[4] = 32'h00253822;  // sub x7,x1,x5     //ALUop of sub=6
            memory[8] = 32'h00329812;  // sub x6,x3,x5
            memory[12] = 32'h000819E5; // or x7,x8,x3     //ALUOP o or=1



    end

    assign Instruction = memory[Address[31:2]];
endmodule
```

DATA MEMORY:

The data memory component is where data is stored and accessed during program execution. It serves as a repository for variables, arrays, and other data structures manipulated by instructions. The pipeline stages involve fetching data from memory, performing calculations or operations, and storing the results back into memory. Efficient access to data memory is crucial for maintaining program correctness and performance. The data memory stage in the pipeline ensures that data operations are seamlessly integrated with instruction execution, enabling the processor to effectively handle data-intensive tasks.

CODE:

```
module DataMemory (
    input clk,
```

```
    input MemRead,
    input MemWrite,
    input [31:0] Address,
    input [31:0] WriteData,
    output [31:0] ReadData
);

    reg [31:0] memory [0:255]; //data memory of 256 bit and width of 32 bit

    assign ReadData = (MemRead) ? memory[Address[31:2]] : 32'b0;//if memory
read is 1 read the data from the memory address given

    always @(posedge clk) begin
        if (MemWrite) begin
            memory[Address[31:2]] <= WriteData;  //write the data to the memory
address
        end
    end
endmodule
```

SINGLE CYCLE MODULES TOP& TEST BENCH :

```
module SingleCycleTop (
    input clk,
    input reset
);
    wire [31:0] PC, NextPC, Instruction, ReadData1, ReadData2, ALUResult,
MemReadData, WriteData;
    wire [4:0] WriteRegister;
    wire [3:0] ALUControl;
    wire [1:0] ALUOp;
    wire [31:0] SignImm, ALUInput2;
    wire Zero, PCSrc, Branch, MemRead, MemWrite, RegDst, ALUSrc, MemtoReg,
RegWrite;
    wire [31:0] BranchTarget, PCPlus4, PCBranch;
    ProgramCounter pc_module(clk, reset, NextPC, PC);
    InstructionMemory imem(PC, Instruction);
    ControlUnit cu(Instruction[31:26], RegDst, ALUSrc, MemtoReg, RegWrite,
MemRead, MemWrite, Branch, ALUOp);
    ALUControl alu_control(ALUOp, Instruction[5:0], ALUControl);
```

```
    MUX2to1 reg_dst_mux(Instruction[20:16], Instruction[15:11], RegDst,
WriteRegister);
    RegisterFile rf(clk, RegWrite, Instruction[25:21], Instruction[20:16],
WriteRegister, WriteData, ReadData1, ReadData2);
    MUX2to1 alu_src_mux(ReadData2, SignImm, ALUSrc, ALUInput2);
    ALU alu(ReadData1, ALUInput2, ALUControl, ALUResult, Zero);
    DataMemory dmem(clk, MemRead, MemWrite, ALUResult, ReadData2, MemReadData);
    SignExtender se(Instruction[15:0], SignImm);
    PCAdder pc_adder(PC, PCPlus4);
    MUX2to1 mem_to_reg_mux(ALUResult, MemReadData, MemtoReg, WriteData);
    assign BranchTarget = SignImm << 2;
    assign PCBranch = PCPlus4 + BranchTarget;
    assign PCSrc = Branch & Zero;
    MUX2to1 pc_mux(PCPlus4, PCBranch, PCSrc, NextPC);
endmodule
```

TEST BENCH:

```
module SingleCycleTB();
    reg clk;reg reset;
    wire [31:0] PC, Instruction, ReadData1, ReadData2, ALUResult, MemReadData,
WriteData;
    wire [31:0] NextPC, SignImm, ALUInput2; wire [4:0] WriteRegister; wire
[3:0] ALUControl;
    wire [1:0] ALUOp; wire Zero, PCSrc, Branch, MemRead, MemWrite, RegDst,
ALUSrc, MemtoReg, RegWrite;
    SingleCycleTop top (.clk(clk), .reset(reset));
    assign PC = top.pc_module.PC; assign NextPC = top.NextPC;
    assign Instruction = top.imem.Instruction; assign ReadData1 =
top.rf.ReadData1;
    assign ReadData2 = top.rf.ReadData2; assign ALUResult = top.alu.ALUResult;
    assign MemReadData = top.dmem.ReadData;
    assign WriteData = top.WriteData; assign SignImm = top.se.SignImm;
    assign ALUInput2 = top.ALUInput2; assign WriteRegister = top.WriteRegister;
    assign ALUControl = top.alu_control.ALUControl;
    assign ALUOp = top.cu.ALUOp; assign Zero = top.alu.Zero;
    assign PCSrc = top.PCSrc; assign Branch = top.cu.Branch;
    assign MemRead = top.cu.MemRead; assign MemWrite = top.cu.MemWrite;
    assign RegDst = top.cu.RegDst; assign ALUSrc = top.cu.ALUSrc;
    assign MemtoReg = top.cu.MemtoReg; assign RegWrite = top.cu.RegWrite;
    initial begin
        clk = 0; forever #5 clk = ~clk;
    end
    initial begin
        reset = 1; #10;
        reset = 0; #300;
```

```
        $finish;
    end
endmodule
```

PROGRAM COUNTER: The address of the next instruction to be fetched. It automatically increments unless a branch or jump instruction modifies its value, maintaining the instruction execution sequence.

CODE:

```
module ProgramCounter (
    input clk,
    input reset,
    input [31:0] NextPC,
    output reg [31:0] PC
);

    always @(posedge clk or posedge reset) begin
        if (reset)
            PC <= 32'b0;
        else
            PC <= NextPC; //stores the next address of the instruction to be
executed
    end
endmodule
```

PROGRAM ADDER:

Calculates the address of the next instruction based on the current PC value and the instruction size. It enables the processor to fetch instructions sequentially or adjust the address for branch or jump instructions, ensuring proper program execution flow.

CODE:

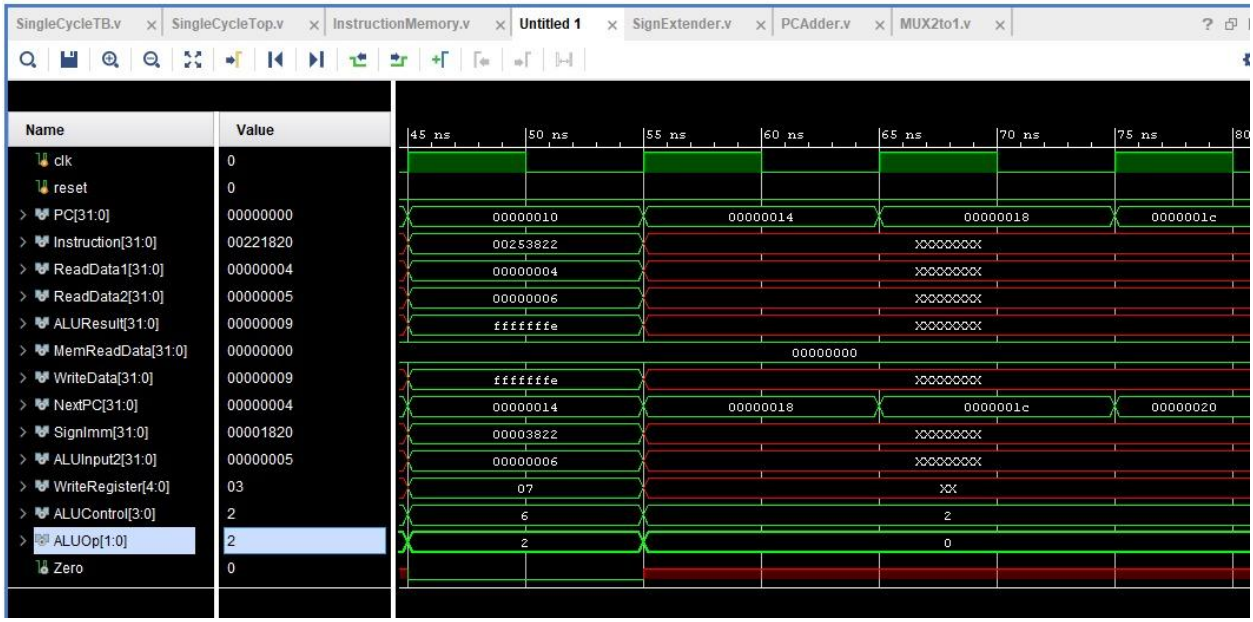
```
module PCAdder (
    input [31:0] PC,
    output [31:0] NextPC
);
    assign NextPC = PC + 4; // increment of 4 at every next instruction
endmodule
```

MUX 2 to1:

CODE:

```
module MUX2to1 (
    input [31:0] In0,
    input [31:0] In1,
    input Sel,
    output [31:0] Out
);
    assign Out = (Sel) ? In1 : In0; // if selection pin then input 1 otherwise input2
endmodule
```

SIMULATION RESULT



Phase II: Pipeline your single-cycle processor of Phase I. Verify that it executes the given subset of the MIPS instruction set.

Components:

Following is the list, explanation & code of all the new modules being used in this Pipelined MIPS processor:

Fetch Cycle:

The module coordinates the fetching of instructions from memory, preparing for the subsequent stages of instruction execution. It comprises several components: a multiplexer (MUX) to select the next program counter value, a program counter (PC) module to manage the current program counter value, an instruction memory (IMEM) to fetch instructions based on the PC value, and an adder to compute the next PC value. Additionally, there are registers to store interim values during clock cycles and logic to handle reset conditions. Ultimately, the module outputs the fetched instruction, the current PC, and the next PC value.

```
module fetch_cycle(clk, rst, PCSrcE, PCTargetE, InstrD, PCD, PCPlus4D);

    // Declare input & outputs
    input clk, rst;
    input PCSrcE;
    input [31:0] PCTargetE;
    output [31:0] InstrD;
    output [31:0] PCD, PCPlus4D;

    // Declaring interim wires
    wire [31:0] PC_F, PCF, PCPlus4F;
    wire [31:0] InstrF;

    // Declaration of Register
    reg [31:0] InstrF_reg;
    reg [31:0] PCF_reg, PCPlus4F_reg;


    // Initiation of Modules
    // Declare PC Mux
    Mux PC_MUX (.a(PCPlus4F),.b(PCTargetE),.s(PCSrcE),.c(PC_F));

    // Declare PC Counter
    PC_Module Program_Counter (.clk(clk),.rst(rst),.PC(PCF),.PC_Next(PC_F));
    // Declare Instruction Memory
    Instruction_Memory IMEM (.rst(rst),.A(PCF),.RD(InstrF));
    // Declare PC adder
```

```
    PC_Adder PC_adder (.a(PCF),.b(32'h00000004),.c(PCPlus4F));
    // Fetch Cycle Register Logic
    always @(posedge clk or negedge rst) begin
        if(rst == 1'b0) begin // if reset then all values set to 0
            InstrF_reg <= 32'h00000000;
            PCF_reg <= 32'h00000000;
            PCPlus4F_reg <= 32'h00000000;
        end
        else begin // store the above obtained values in these registers
            InstrF_reg <= InstrF;
            PCF_reg <= PCF;
            PCPlus4F_reg <= PCPlus4F;
        end
    end
```

Decode Cycle:

The module handles the decoding of fetched instructions and prepares necessary signals for subsequent stages of instruction execution. It includes components such as a control unit, register file, and sign extender. The control unit determines various control signals based on the opcode and function fields of the instruction. The register file facilitates access to registers for operand fetching. Additionally, the sign extender extends immediate values for arithmetic and logical operations. Interim registers are used to store values during clock cycles, and output assignments are made to propagate these values to subsequent stages. Overall, this module ensures that decoded instructions are properly processed and prepared for execution in the pipeline.

```
module decode_cycle(clk, rst, InstrD, PCD, PCPlus4D, RegWriteW, RDW, ResultW,
RegWriteE, ALUSrcE, MemWriteE, ResultSrcE,
    BranchE,  ALUControlE, RD1_E, RD2_E, Imm_Ext_E, RD_E, PCE, PCPlus4E, RS1_E,
RS2_E);

    // Declaring I/O
    input clk, rst, RegWriteW;
    input [4:0] RDW;
    input [31:0] InstrD, PCD, PCPlus4D, ResultW;

    output RegWriteE,ALUSrcE,MemWriteE,ResultSrcE,BranchE;
    output [2:0] ALUControlE;
    output [31:0] RD1_E, RD2_E, Imm_Ext_E;
```

```
    output [4:0] RS1_E, RS2_E, RD_E;
    output [31:0] PCE, PCPlus4E;

    // Declare Interim Wires
    wire RegWriteD,ALUSrcD,MemWriteD,ResultSrcD,BranchD;
    wire [1:0] ImmSrcD;
    wire [2:0] ALUControlD;
    wire [31:0] RD1_D, RD2_D, Imm_Ext_D;

    // Declaration of Interim Register
    reg RegWriteD_r,ALUSrcD_r,MemWriteD_r,ResultSrcD_r,BranchD_r;
    reg [2:0] ALUControlD_r;
    reg [31:0] RD1_D_r, RD2_D_r, Imm_Ext_D_r;
    reg [4:0] RD_D_r, RS1_D_r, RS2_D_r;
    reg [31:0] PCD_r, PCPlus4D_r;


    // Initiate the modules
    // Control Unit
    Control_Unit_Top control
(.Op(InstrD[6:0]),.RegWrite(RegWriteD),.ImmSrc(ImmSrcD),.ALUSrc(ALUSrcD),.MemWr
ite(MemWriteD),.ResultSrc(ResultSrcD),.Branch(BranchD),.funct3(InstrD[14:12]),.
funct7(InstrD[31:25]),.ALUControl(ALUControlD));
    // Register File
    Register_File rf
(.clk(clk),.rst(rst),.WE3(RegWriteW),.WD3(ResultW),.A1(InstrD[19:15]),.A2(Instr
D[24:20]),.A3(RDW),.RD1(RD1_D),.RD2(RD2_D));
    // Sign Extension
    Sign_Extend extension
(.In(InstrD[31:0]),.Imm_Ext(Imm_Ext_D),.ImmSrc(ImmSrcD));

    // Declaring Register Logic
    always @(posedge clk or negedge rst) begin
        if(rst == 1'b0) begin // if reset is zero declare all values to 0
            RegWriteD_r <= 1'b0;
            ALUSrcD_r <= 1'b0;
            MemWriteD_r <= 1'b0;
            ResultSrcD_r <= 1'b0;
            BranchD_r <= 1'b0;
            ALUControlD_r <= 3'b000;
            RD1_D_r <= 32'h00000000;
            RD2_D_r <= 32'h00000000;
            Imm_Ext_D_r <= 32'h00000000;
            RD_D_r <= 5'h00;
            PCD_r <= 32'h00000000;
```

```
            PCPlus4D_r <= 32'h00000000;
            RS1_D_r <= 5'h00;
            RS2_D_r <= 5'h00;
        end
        // instruction after the IF/ID cycle are then come to decode stage
        else begin // assign the corresponding values to the respective
registers
            RegWriteD_r <= RegWriteD;
            ALUSrcD_r <= ALUSrcD;
            MemWriteD_r <= MemWriteD;
            ResultSrcD_r <= ResultSrcD;
            BranchD_r <= BranchD;
            ALUControlD_r <= ALUControlD;
            RD1_D_r <= RD1_D;
            RD2_D_r <= RD2_D;
            Imm_Ext_D_r <= Imm_Ext_D;
            RD_D_r <= InstrD[11:7];
            PCD_r <= PCD;
            PCPlus4D_r <= PCPlus4D;
            RS1_D_r <= InstrD[19:15];
            RS2_D_r <= InstrD[24:20];
        end
    end

    // Output asssign statements
    assign RegWriteE = RegWriteD_r;
    assign ALUSrcE = ALUSrcD_r;
    assign MemWriteE = MemWriteD_r;
    assign ResultSrcE = ResultSrcD_r;
    assign BranchE = BranchD_r;
    assign ALUControlE = ALUControlD_r;
    assign RD1_E = RD1_D_r;
    assign RD2_E = RD2_D_r;
    assign Imm_Ext_E = Imm_Ext_D_r;
    assign RD_E = RD_D_r;
    assign PCE = PCD_r;
    assign PCPlus4E = PCPlus4D_r;
    assign RS1_E = RS1_D_r;
    assign RS2_E = RS2_D_r;

endmodule
```

Execute Cycle:

This stage performs arithmetic and logical operations, calculates branch targets, and manages control signals and data forwarding. Inputs include control signals, operand data, and pipeline register data from the decode stage, while outputs provide control signals and data to the memory stage, along with branch target and decision signals. The module uses MUXes for forwarding to select correct operand sources based on data hazards, an ALU for executing specified operations, and a branch adder for computing target addresses. Register logic updates pipeline registers on each clock cycle or reset, and output assignments determine control signals and data for the next pipeline stage. This ensures that the execute stage performs necessary computations efficiently while handling control and data hazards.

```
module execute_cycle(clk, rst, RegWriteE, ALUSrcE, MemWriteE, ResultSrcE,
BranchE, ALUControlE,
    RD1_E, RD2_E, Imm_Ext_E, RD_E, PCE, PCPlus4E, PCSrcE, PCTargetE, RegWriteM,
MemWriteM, ResultSrcM, RD_M, PCPlus4M, WriteDataM, ALU_ResultM, ResultW,
ForwardA_E, ForwardB_E);

    // Declaration I/Os
    input clk, rst, RegWriteE,ALUSrcE,MemWriteE,ResultSrcE,BranchE;
    input [2:0] ALUControlE;
    input [31:0] RD1_E, RD2_E, Imm_Ext_E;
    input [4:0] RD_E;
    input [31:0] PCE, PCPlus4E;
    input [31:0] ResultW;
    input [1:0] ForwardA_E, ForwardB_E;

    output PCSrcE, RegWriteM, MemWriteM, ResultSrcM;
    output [4:0] RD_M;
    output [31:0] PCPlus4M, WriteDataM, ALU_ResultM;
    output [31:0] PCTargetE;

    // Declaration of Interim Wires
    wire [31:0] Src_A, Src_B_interim, Src_B;
    wire [31:0] ResultE;
    wire ZeroE;

    // Declaration of Register
    reg RegWriteE_r, MemWriteE_r, ResultSrcE_r;
    reg [4:0] RD_E_r;
    reg [31:0] PCPlus4E_r, RD2_E_r, ResultE_r;

    // Declaration of Modules
    // 3 by 1 Mux for Source A
```

```
    Mux_3_by_1 srca_mux
(.a(RD1_E),.b(ResultW),.c(ALU_ResultM),.s(ForwardA_E),.d(Src_A));
    // 3 by 1 Mux for Source B
    Mux_3_by_1 srcb_mux
(.a(RD2_E),.b(ResultW),.c(ALU_ResultM),.s(ForwardB_E),.d(Src_B_interim));
     // ALU Src Mux
    Mux alu_src_mux (.a(Src_B_interim),.b(Imm_Ext_E),.s(ALUSrcE),.c(Src_B));
    // ALU Unit
    ALU alu
(.A(Src_A),.B(Src_B),.Result(ResultE),.ALUControl(ALUControlE),.OverFlow(),.Car
ry(),.Zero(ZeroE),.Negative());
    // Adder
    PC_Adder branch_adder (.a(PCE),.b(Imm_Ext_E),.c(PCTargetE));
    // Register Logic
    always @(posedge clk or negedge rst) begin
        if(rst == 1'b0) begin//if the reset is 0 set all to default 0
            RegWriteE_r <= 1'b0;
            MemWriteE_r <= 1'b0;
            ResultSrcE_r <= 1'b0;
            RD_E_r <= 5'h00;
            PCPlus4E_r <= 32'h00000000;
            RD2_E_r <= 32'h00000000;
            ResultE_r <= 32'h00000000;
        end
        else begin  //storing registers value
            RegWriteE_r <= RegWriteE;
            MemWriteE_r <= MemWriteE;
            ResultSrcE_r <= ResultSrcE;
            RD_E_r <= RD_E;
            PCPlus4E_r <= PCPlus4E;
            RD2_E_r <= Src_B_interim;
            ResultE_r <= ResultE;
        end
    end

    // Output Assignments
    assign PCSrcE = ZeroE &  BranchE;
    assign RegWriteM = RegWriteE_r;
    assign MemWriteM = MemWriteE_r;
    assign ResultSrcM = ResultSrcE_r;
    assign RD_M = RD_E_r;
    assign PCPlus4M = PCPlus4E_r;
    assign WriteDataM = RD2_E_r;
    assign ALU_ResultM = ResultE_r;
endmodule
```

Memory Cycle:

This stage handles memory read and write operations, and it prepares the data for the write-backstage. The module receives inputs such as control signals, data to be written to memory, and addresses. It outputs control signals and data for the write-backstage. The key components include a data memory module that performs memory read and write operations based on control signals and addresses provided. The module also features register logic to store values across clock cycles and reset conditions, ensuring proper synchronization of data. Outputs are assigned from these registers to propagate the necessary signals and data to the next pipeline stage, ensuring smooth and efficient data processing through the memory cycle.

```
module memory_cycle(clk, rst, RegWriteM, MemWriteM, ResultSrcM, RD_M, PCPlus4M,
WriteDataM,
    ALU_ResultM, RegWriteW, ResultSrcW, RD_W, PCPlus4W, ALU_ResultW,
ReadDataW);

    // Declaration of I/Os
    input clk, rst, RegWriteM, MemWriteM, ResultSrcM;
    input [4:0] RD_M;
    input [31:0] PCPlus4M, WriteDataM, ALU_ResultM;

    output RegWriteW, ResultSrcW;
    output [4:0] RD_W;
    output [31:0] PCPlus4W, ALU_ResultW, ReadDataW;

    // Declaration of Interim Wires
    wire [31:0] ReadDataM;

    // Declaration of Interim Registers
    reg RegWriteM_r, ResultSrcM_r;
    reg [4:0] RD_M_r;
    reg [31:0] PCPlus4M_r, ALU_ResultM_r, ReadDataM_r;

    // Declaration of Module Initiation
    Data_Memory dmem
(.clk(clk),.rst(rst),.WE(MemWriteM),.WD(WriteDataM),.A(ALU_ResultM),.RD(ReadDat
aM));

    // Memory Stage Register Logic
    always @(posedge clk or negedge rst) begin
        if (rst == 1'b0) begin
```

```
            RegWriteM_r <= 1'b0;
            ResultSrcM_r <= 1'b0;
            RD_M_r <= 5'h00;
            PCPlus4M_r <= 32'h00000000;
            ALU_ResultM_r <= 32'h00000000;
            ReadDataM_r <= 32'h00000000;
        end
        else begin
            RegWriteM_r <= RegWriteM;
            ResultSrcM_r <= ResultSrcM;
            RD_M_r <= RD_M;
            PCPlus4M_r <= PCPlus4M;
            ALU_ResultM_r <= ALU_ResultM;
            ReadDataM_r <= ReadDataM;
        end
    end

    // Declaration of output assignments
    assign RegWriteW = RegWriteM_r;
    assign ResultSrcW = ResultSrcM_r;
    assign RD_W = RD_M_r;
    assign PCPlus4W = PCPlus4M_r;
    assign ALU_ResultW = ALU_ResultM_r;
    assign ReadDataW = ReadDataM_r;

endmodule
```

Write Back Cycle:

This module (if needed) ensures that the correct result of instruction execution is selected and prepared for writing back into the register file, maintaining the integrity of the data flow in the processor's pipeline.

```
module writeback_cycle(clk, rst, ResultSrcW, PCPlus4W, ALU_ResultW, ReadDataW,
ResultW);

// Declaration of IOs
input clk, rst, ResultSrcW;
input [31:0] PCPlus4W, ALU_ResultW, ReadDataW;

output [31:0] ResultW;

// Declaration of Module
Mux result_mux (.a(ALU_ResultW),.b(ReadDataW),.s(ResultSrcW),.c(ResultW));
Endmodule
```

Hazard Detection Unit:

Hazard detection unit check the instruction in the pipeline for any dependency, if founded it forwards the proper data to proper location in the execution of instructions making sure the pipeline don't create any faulty data.

```
module hazard_unit(rst, RegWriteM, RegWriteW, RD_M, RD_W, Rs1_E, Rs2_E,
ForwardAE, ForwardBE);

    // Declaration of I/Os
    input rst, RegWriteM, RegWriteW;
    input [4:0] RD_M, RD_W, Rs1_E, Rs2_E;
    output [1:0] ForwardAE, ForwardBE;
    //if the register in write back stage matches with the register in execute
stage hazard detects
    //if hazard is detected then data is forwarded to the correct stage to be
used.
    assign ForwardAE = (rst == 1'b0) ? 2'b00 :
                       ((RegWriteM == 1'b1) & (RD_M != 5'h00) & (RD_M ==
Rs1_E)) ? 2'b10 :
                       ((RegWriteW == 1'b1) & (RD_W != 5'h00) & (RD_W ==
Rs1_E)) ? 2'b01 : 2'b00;

    assign ForwardBE = (rst == 1'b0) ? 2'b00 :
                       ((RegWriteM == 1'b1) & (RD_M != 5'h00) & (RD_M ==
Rs2_E)) ? 2'b10 :
                       ((RegWriteW == 1'b1) & (RD_W != 5'h00) & (RD_W ==
Rs2_E)) ? 2'b01 : 2'b00;

endmodule
```

1 BIT PREDICTOR:

It is used for the prediction of branches.

```
module predictor(
    input wire clk,
    input wire rst,
    input wire branch,      // Signal indicating a branch instruction
    input wire taken,       // Actual outcome of the branch
    output reg prediction   // Prediction made by the predictor
);

    // State: 1 for taken, 0 for not taken
    reg state;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Reset the predictor to a known state, e.g., predict not taken
            state <= 1'b0;
            prediction <= 1'b0;
        end else if (branch) begin
            // Make a prediction
            prediction <= state;

            // Update the state based on the actual outcome
            state <= taken;
        end
    end

endmodule
```

Top Module (Pipelined Processor):

This module integrates all the previously defined components to form a complete pipelined MIPS processor. It manages the seamless flow of data and control signals across the fetch, decode, execute, memory, and write-back stages, ensuring efficient instruction execution within the pipeline

```
module Pipeline_top(clk, rst);

    // Declaration of I/O
    input clk, rst;

    // Declaration of Interim Wires
    wire PCSrcE, RegWriteW, RegWriteE, ALUSrcE, MemWriteE, ResultSrcE, BranchE,
RegWriteM, MemWriteM, ResultSrcM, ResultSrcW;
    wire [2:0] ALUControlE;
    wire [4:0] RD_E, RD_M, RDW;
    wire [31:0] PCTargetE, instructions, PCD, PCPlus4D, ResultW, RD1_E, RD2_E,
Imm_Ext_E, PCE, PCPlus4E, PCPlus4M, WriteDataM, ALU_ResultM;
    wire [31:0] PCPlus4W, ALU_ResultW, ReadDataW;
    wire [4:0] RS1_E, RS2_E;
    wire [1:0] ForwardBE, ForwardAE;


    // Module Initiation
    // Fetch Stage
    fetch_cycle Fetch (
                        .clk(clk),
                        .rst(rst),
                        .PCSrcE(PCSrcE),
                        .PCTargetE(PCTargetE),
                        .InstrD(instructions),
                        .PCD(PCD),
                        .PCPlus4D(PCPlus4D)
                    );

    // Decode Stage
    decode_cycle Decode (
                        .clk(clk),
                        .rst(rst),
                        .InstrD(instructions),
                        .PCD(PCD),
                        .PCPlus4D(PCPlus4D),
                        .RegWriteW(RegWriteW),
                        .RDW(RDW),
                        .ResultW(ResultW),
                        .RegWriteE(RegWriteE),
                        .ALUSrcE(ALUSrcE),
                        .MemWriteE(MemWriteE),
                        .ResultSrcE(ResultSrcE),
                        .BranchE(BranchE),
```

```
                    .ALUControlE(ALUControlE),
                    .RD1_E(RD1_E),
                    .RD2_E(RD2_E),
                    .Imm_Ext_E(Imm_Ext_E),
                    .RD_E(RD_E),
                    .PCE(PCE),
                    .PCPlus4E(PCPlus4E),
                    .RS1_E(RS1_E),
                    .RS2_E(RS2_E)
                );

    // Execute Stage
    execute_cycle Execute (
                    .clk(clk),
                    .rst(rst),
                    .RegWriteE(RegWriteE),
                    .ALUSrcE(ALUSrcE),
                    .MemWriteE(MemWriteE),
                    .ResultSrcE(ResultSrcE),
                    .BranchE(BranchE),
                    .ALUControlE(ALUControlE),
                    .RD1_E(RD1_E),
                    .RD2_E(RD2_E),
                    .Imm_Ext_E(Imm_Ext_E),
                    .RD_E(RD_E),
                    .PCE(PCE),
                    .PCPlus4E(PCPlus4E),
                    .PCSrcE(PCSrcE),
                    .PCTargetE(PCTargetE),
                    .RegWriteM(RegWriteM),
                    .MemWriteM(MemWriteM),
                    .ResultSrcM(ResultSrcM),
                    .RD_M(RD_M),
                    .PCPlus4M(PCPlus4M),
                    .WriteDataM(WriteDataM),
                    .ALU_ResultM(ALU_ResultM),
                    .ResultW(ResultW),
                    .ForwardA_E(ForwardAE),
                    .ForwardB_E(ForwardBE)
                );

    // Memory Stage
    memory_cycle Memory (
                    .clk(clk),
                    .rst(rst),
```

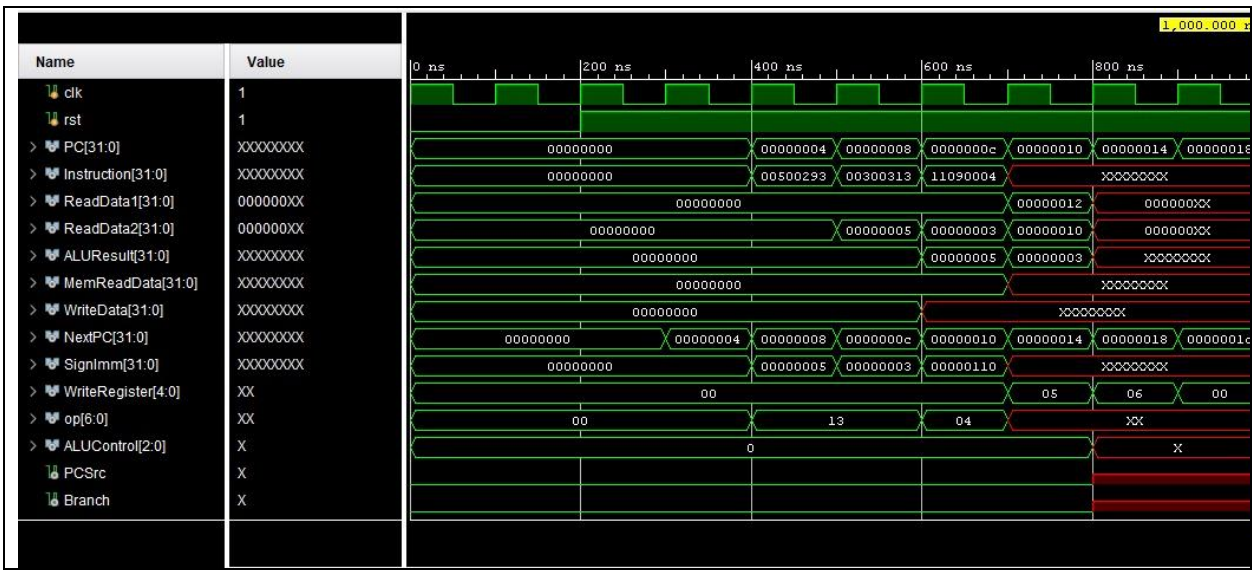
```
                        .RegWriteM(RegWriteM),
                        .MemWriteM(MemWriteM),
                        .ResultSrcM(ResultSrcM),
                        .RD_M(RD_M),
                        .PCPlus4M(PCPlus4M),
                        .WriteDataM(WriteDataM),
                        .ALU_ResultM(ALU_ResultM),
                        .RegWriteW(RegWriteW),
                        .ResultSrcW(ResultSrcW),
                        .RD_W(RDW),
                        .PCPlus4W(PCPlus4W),
                        .ALU_ResultW(ALU_ResultW),
                        .ReadDataW(ReadDataW)
                    );

    // Write Back Stage
    writeback_cycle WriteBack (
                        .clk(clk),
                        .rst(rst),
                        .ResultSrcW(ResultSrcW),
                        .PCPlus4W(PCPlus4W),
                        .ALU_ResultW(ALU_ResultW),
                        .ReadDataW(ReadDataW),
                        .ResultW(ResultW)
                    );

    // Hazard Unit
    hazard_unit Forwarding_block (
                        .rst(rst),
                        .RegWriteM(RegWriteM),
                        .RegWriteW(RegWriteW),
                        .RD_M(RD_M),
                        .RD_W(RDW),
                        .Rs1_E(RS1_E),
                        .Rs2_E(RS2_E),
                        .ForwardAE(ForwardAE),
                        .ForwardBE(ForwardBE)
                        );
endmodule
```

Test Bench & Simulation (Pipelined Processor):

```
module testbench();
    reg clk=0, rst;
    wire [31:0] PC, Instruction, ReadData1, ReadData2, ALUResult, MemReadData,
WriteData;
    wire [31:0] NextPC, SignImm; wire [4:0] WriteRegister;
    wire [6:0]op;
    wire [2:0]ALUControl;
    // wire Zero;
    wire PCSrc, Branch, MemWrite, ALUSrc, RegWrite;

Pipeline_top pipeline (.clk(clk), .rst(rst));

    assign PC = pipeline.PCD;
    assign NextPC = pipeline.PCPlus4D;
    assign Instruction = pipeline.instructions;
    assign ReadData1 = pipeline.RS1_E;
    assign ReadData2 = pipeline.RS2_E;
    assign ALUResult = pipeline.ALU_ResultM;
    assign MemReadData = pipeline.ReadDataW;
    assign WriteData = pipeline.WriteDataM;
    assign SignImm = pipeline.Decode.Imm_Ext_D;
    assign WriteRegister = pipeline.RDW;
    // assign Zero = pipeline.Execute.ZeroE;
    assign ALUControl = pipeline.ALUControlE;
    assign op = pipeline.Decode.control.Op;
    assign PCSrc = pipeline.PCSrcE;
    assign Branch = pipeline.BranchE;
    assign MemWrite = pipeline.MemWriteE;
    assign ALUSrc = pipeline.ALUSrcE;
    assign RegWrite = pipeline.RegWriteE;
    always begin
        clk = ~clk;
        #50;
    end

    initial begin
        rst <= 1'b0;
        #200;
        rst <= 1'b1;
```

```
        #1000;
        $finish;
    end

endmodule
```

SIMULATION:



SPEEDUP AND GAIN:

CPI & Throughput of Code:

Single Cycle:

$$Clock\ Rate = 10\ GHz$$

$$CPI = 3$$

$$Execution\ Time = \frac{CPI * Instruction\ Count}{Clock\ Rate}$$

$$Execution\ Time = \frac{3 * 6}{10 * 10^6}$$

$$Execution\ Time = 1.8\ ns$$

$$Latency = 3$$

Pipelined:

$$Clock\ Rate = 10\ GHz$$

$$CPI = 1$$

$$Execution\ Time = \frac{CPI * Instruction\ Count}{Clock\ Rate}$$

$$Execution\ Time = \frac{1 * 7}{10 * 10^6}$$

$$Execution\ Time = 0.7\ ns$$

$$Latency = 1$$

REFERENCES:

https://medium.com/@LambdaMamba/building-a-mips-5-stage-pipeline-processor-in-verilog-6d627a31127c

https://www.fpga4student.com/2017/06/32-bit-pipelined-mips-processor-in-verilog-3.html