# Computer Organization & Architecture

Project

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

**Group Members:**

| Name | Roll Number |
|---|---|
| Muhammad Miqdad Ahmad | BSCE22001 |
| Ahmad Waleed Akhtar | BSCE22003 |
| Muhammad Arham | BSCE22007 |

**Submitted To: Muhammad Usama Riaz**

## Contents

## 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). We will work on this project with Verilog HDL hardware language.

## Phase I:

In the initial phase of our project, we undertook the construction of a Single Cycle Processor. To achieve this, we developed several distinct components, meticulously designed to fulfill specific functions. These components were then integrated within the main Top module, serving as the central hub that interconnected them seamlessly. This meticulous assembly process effectively brought forth the Single Cycle Processor.

### Components:

Following is the list, explanation & code of all the components being used in this Single Cycle Processor:

#### Program Counter:

In MIPS architecture, the Program Counter (PC) holds the memory address of the next instruction to be fetched and executed. It ensures instructions are retrieved and executed in the correct sequence.

```verilog
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;
    end
endmodule
```

## Instruction Memory:

It stores the program instructions to be executed. However, in a non-pipeline processor, each instruction is fetched, decoded, and executed one at a time, without overlapping stages. The processor fetches an instruction from memory, executes it, and then moves on to the next instruction.

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

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

    initial begin

        memory[0] = 32'h00221820;
        memory[4] = 32'h00161710;
        memory[8] = 32'h00223820;
        memory[12] = 32'h00241820;
        memory[16] = 32'h00161710;
        memory[20] = 32'h00821820;
        memory[24] = 32'h00721820;
        memory[28] = 32'h00621820;
        memory[32] = 32'h00521820;
        memory[36] = 32'h00421820;
        memory[40] = 32'h00321820;

    end

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

## ALU:

The Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU) responsible for performing arithmetic and logic operations on input data. These operations include addition, subtraction, bitwise logical operations (AND, OR, XOR), comparison operations (such as equality and inequality), and shifting operations. The ALU takes input data from the CPU's registers, performs the specified operation, and outputs the result back to the registers. It plays a critical role in executing instructions and performing computations within a computer system.

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

    assign Zero = (ALUResult == 0);

    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;
        endcase
    end
endmodule
```

## ALU Control:

The required process to be performed by the ALU unit is specified and found in this unit.

```
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
```

## Control Unit:

The Control Unit (CU) is a core component of a CPU responsible for coordinating the execution of instructions and controlling the flow of data within the processor. It interprets instructions fetched from memory, decodes them to determine the required operations, and generates control signals to coordinate the activities of other CPU components, such as the ALU, memory, and input/output devices. The Control Unit ensures that instructions are executed in the correct sequence and that data moves smoothly between different parts of the CPU and memory. It plays a pivotal role in the overall operation and performance of a computer system.

```
module ControlUnit (
    input [5:0] Opcode,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
    output reg [1:0] ALUOp
);

    always @(*) begin
        case (Opcode)
            6'b000000: begin // R-type
                RegDst = 1;
                ALUSrc = 0;
                MemtoReg = 0;
                RegWrite = 1;
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                ALUOp = 2'b10;
            end
            6'b100011: begin // LW
                RegDst = 0;
                ALUSrc = 1;
                MemtoReg = 1;
                RegWrite = 1;
                MemRead = 1;
                MemWrite = 0;
                Branch = 0;
                ALUOp = 2'b00;
            end
            6'b101011: begin // SW
                RegDst = 0; // Don't care
                ALUSrc = 1;
                MemtoReg = 0; // Don't care
                RegWrite = 0;
                MemRead = 0;
                MemWrite = 1;
                Branch = 0;
                ALUOp = 2'b00;
            end
            6'b000100: begin // BEQ
                RegDst = 0; // Don't care
                ALUSrc = 0;
                MemtoReg = 0; // Don't care
                RegWrite = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 1;
                ALUOp = 2'b01;
            end
            default: begin
                RegDst = 0;
                ALUSrc = 0;
                MemtoReg = 0;
                RegWrite = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                ALUOp = 2'b00;
            end
        endcase
    end
endmodule
```

## Data Memory:

In a computer architecture, the data memory serves as a storage location for input data, intermediate results, and output data. It allows the CPU to read and write data as needed during program execution. Data memory is typically organized as a collection of memory cells, each capable of storing a fixed amount of data (e.g., 8, 32 bits or more), and each memory cell has a unique address that the CPU uses to access it.

```
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];

    assign ReadData = (MemRead) ? memory[Address[31:2]] : 32'b0;

    always @(posedge clk) begin
        if (MemWrite) begin
            memory[Address[31:2]] <= WriteData;
        end
    end
endmodule
```

## 2 to 1 Mux:

These are used in various places to decide on different actions depending upon systems requirement.

```
module MUX2to1 (
    input [31:0] In0,
    input [31:0] In1,
    input Sel,
    output [31:0] Out
);
    assign Out = (Sel) ? In1 : In0;
endmodule
```

## PC Adder:

This module/component increments PC address by 4 bits in each clock cycle.

```
module PCAdder (
    input [31:0] PC,
    output [31:0] NextPC
);
    assign NextPC = PC + 4;
endmodule
```

## Register File:

The register file stores data temporarily during instruction execution. It provides fast access to operands for operations and serves as a staging area for storing intermediate results.

```
module RegisterFile (
    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;    registers[1] = 32'd1;
    registers[2] = 32'd2;    registers[3] = 32'd3;
    registers[4] = 32'd4;    registers[5] = 32'd5;
    registers[6] = 32'd6;    registers[7] = 32'd7;
    registers[8] = 32'd8;    registers[9] = 32'd9;
    registers[10] = 32'd8;    registers[11] = 32'd7;
    registers[12] = 32'd6;    registers[13] = 32'd5;
    registers[14] = 32'd4;    registers[15] = 32'd3;
    registers[16] = 32'd2;    registers[17] = 32'd1;
    registers[18] = 32'd2;    registers[19] = 32'd3;
    registers[20] = 32'd4;    registers[21] = 32'd5;
    registers[22] = 32'd6;    registers[23] = 32'd7;
    registers[24] = 32'd8;    registers[25] = 32'd9;
    registers[26] = 32'd8;    registers[27] = 32'd7;
    registers[28] = 32'd6;    registers[29] = 32'd5;
    registers[30] = 32'd4;    registers[31] = 32'd3;
    end

    assign ReadData1 = registers[ReadRegister1];
    assign ReadData2 = registers[ReadRegister2];

    always @(posedge clk) begin
        if (RegWrite) begin
            registers[WriteRegister] <= WriteData;
        end
    end
endmodule
```

## Sign Extender:

A sign extender is used to preserve the sign of a binary number when it's extended to a wider bit-width, ensuring that the original number's sign is maintained.

```
module SignExtender (
    input [15:0] Inst,
    output [31:0] SignImm
);
    assign SignImm = {{16{Inst[15]}}, Inst};
endmodule
```

## Top Module:

This module connects all the smaller modules together to create the Single Cycle MIPS processor.

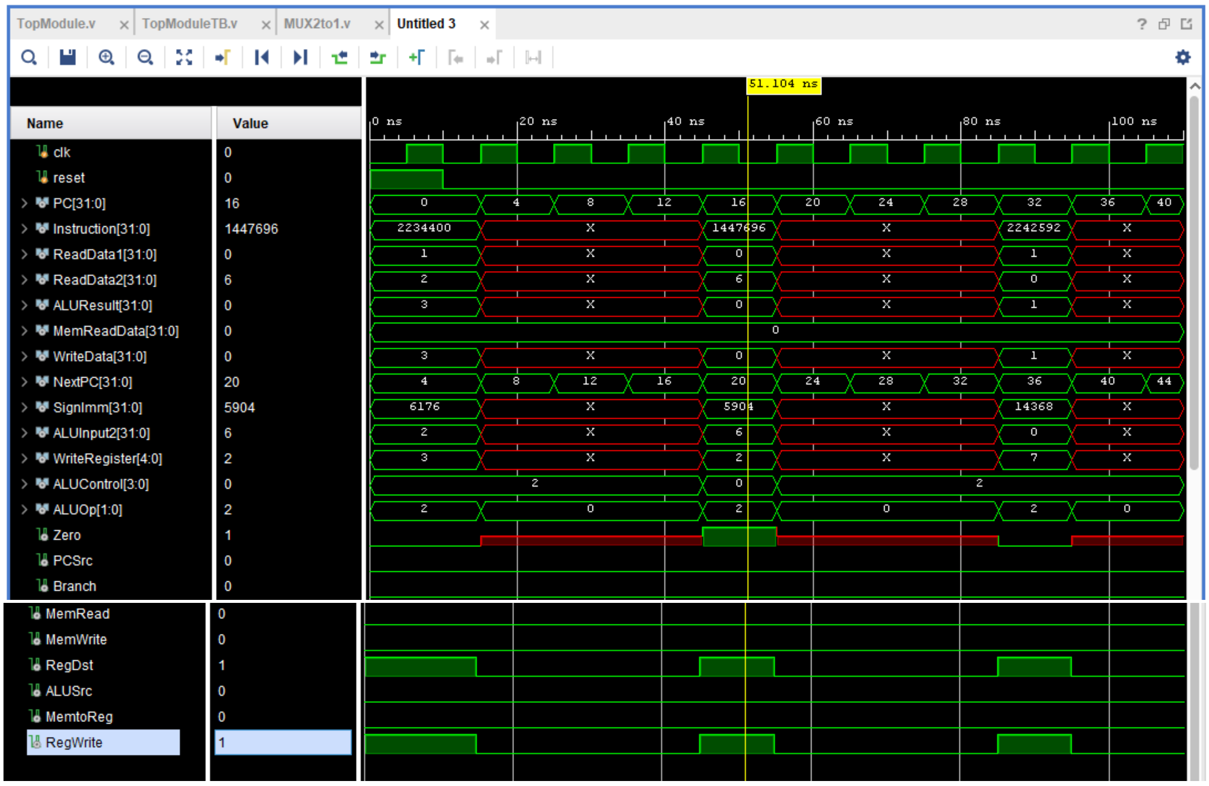
```
module TopModule (
    input clk,
    input reset
);
    wire [31:0] PC, NextPC, Instruction, ReadData1, ReadData2, ALUResult,
MemReadData, WriteData;
    wire [4:0] WriteRegister;
    wire [3:0] ALUControl;
    wire [1:0] ALUOp;
    wire [31:0] SignImm, ALUInput2;
    wire Zero, PCSrc, Branch, MemRead, 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 & Simulation (Single Cycle):

```
module TopModuleTB;
    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;
    TopModule topModule (.clk(clk), .reset(reset));
    assign PC = topModule.pc_module.PC; assign NextPC = topModule.NextPC;
    assign Instruction = topModule.imem.Instruction; assign ReadData1 =
topModule.rf.ReadData1;
    assign ReadData2 = topModule.rf.ReadData2; assign ALUResult =
topModule.alu.ALUResult;
    assign MemReadData = topModule.dmem.ReadData;
    assign WriteData = topModule.WriteData;
    assign SignImm = topModule.se.SignImm;
    assign ALUInput2 = topModule.ALUInput2; assign WriteRegister =
topModule.WriteRegister;
    assign ALUControl = topModule.alu_control.ALUControl;
    assign ALUOp = topModule.cu.ALUOp; assign Zero = topModule.alu.Zero;
    assign PCSrc = topModule.PCSrc; assign Branch = topModule.cu.Branch;
    assign MemRead = topModule.cu.MemRead; assign MemWrite =
topModule.cu.MemWrite;
    assign RegDst = topModule.cu.RegDst; assign ALUSrc = topModule.cu.ALUSrc;
    assign MemtoReg = topModule.cu.MemtoReg; assign RegWrite =
topModule.cu.RegWrite;
    initial begin
        clk = 0; forever #5 clk = ~clk;
    end
    initial begin
        reset = 1; #10;
        reset = 0; #100;
        $finish;
    end
endmodule
```

### *Simulation:*



## Phase II:

In Phase II we will be using the modules (with slight modifications) created for single cycle processor & some new additional modules to effectively create a pipelined MIPS processor.

## 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
            InstrF_reg <= 32'h00000000;
            PCF_reg <= 32'h00000000;
            PCPlus4F_reg <= 32'h00000000;
        end
        else begin
            InstrF_reg <= InstrF;
            PCF_reg <= PCF;
            PCPlus4F_reg <= PCPlus4F;
        end
    end


    // Assigning Registers Value to the Output port
    assign  InstrD = (rst == 1'b0) ? 32'h00000000 : InstrF_reg;
    assign  PCD = (rst == 1'b0) ? 32'h00000000 : PCF_reg;
    assign  PCPlus4D = (rst == 1'b0) ? 32'h00000000 : PCPlus4F_reg;


endmodule
```

## 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),
                                .MemWrite(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(InstrD[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
            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
        else begin
            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(),
            .Carry(),
            .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
            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
            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;
```

```verilog
    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(ReadDataM)
                    );

    // 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;

//assigning values according to control signals
    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
```

## Two Bit Predictor:

It is used for the prediction of branches.

```
module two_bit_predictor (
    input wire clk,
    input wire reset,
    input wire branch,    // Branch instruction signal (1 if a branch
instruction is present, 0 otherwise)
    input wire taken,     // Actual outcome of the branch (1 if taken, 0 if
not taken)
    output reg predict    // Prediction output (1 if predicted taken, 0 if
predicted not taken)
);

    // State encoding for the 2-bit predictor
    parameter STRONGLY_NOT_TAKEN = 2'b00;
    parameter WEAKLY_NOT_TAKEN   = 2'b01;
    parameter WEAKLY_TAKEN       = 2'b10;
    parameter STRONGLY_TAKEN     = 2'b11;

    reg [1:0] state, next_state;

    // State transition and output logic
    always @(*) begin
        case (state)
            STRONGLY_NOT_TAKEN: begin
                predict = 1'b0;
                if (branch) begin
                    if (taken)
                        next_state = WEAKLY_NOT_TAKEN;
                    else
                        next_state = STRONGLY_NOT_TAKEN;
                end else begin
                    next_state = STRONGLY_NOT_TAKEN;
                end
            end
            WEAKLY_NOT_TAKEN: begin
                predict = 1'b0;
                if (branch) begin
                    if (taken)
                        next_state = WEAKLY_TAKEN;
                    else
                        next_state = STRONGLY_NOT_TAKEN;
                end else begin
                    next_state = WEAKLY_NOT_TAKEN;
                end
            end
            WEAKLY_TAKEN: begin
                predict = 1'b1;
                if (branch) begin
                    if (taken)
                        next_state = STRONGLY_TAKEN;
                    else
                        next_state = WEAKLY_NOT_TAKEN;
                end else begin
                    next_state = WEAKLY_TAKEN;
                end
            end
            STRONGLY_TAKEN: begin
                predict = 1'b1;
                if (branch) begin
                    if (taken)
                        next_state = STRONGLY_TAKEN;
                    else
                        next_state = WEAKLY_TAKEN;
                end else begin
                    next_state = STRONGLY_TAKEN;
                end
            end
            default: begin
                predict = 1'b0;
                next_state = STRONGLY_NOT_TAKEN;
            end
        endcase
    end

    // State register
    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= STRONGLY_NOT_TAKEN;
        else
            state <= next_state;
    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);
```

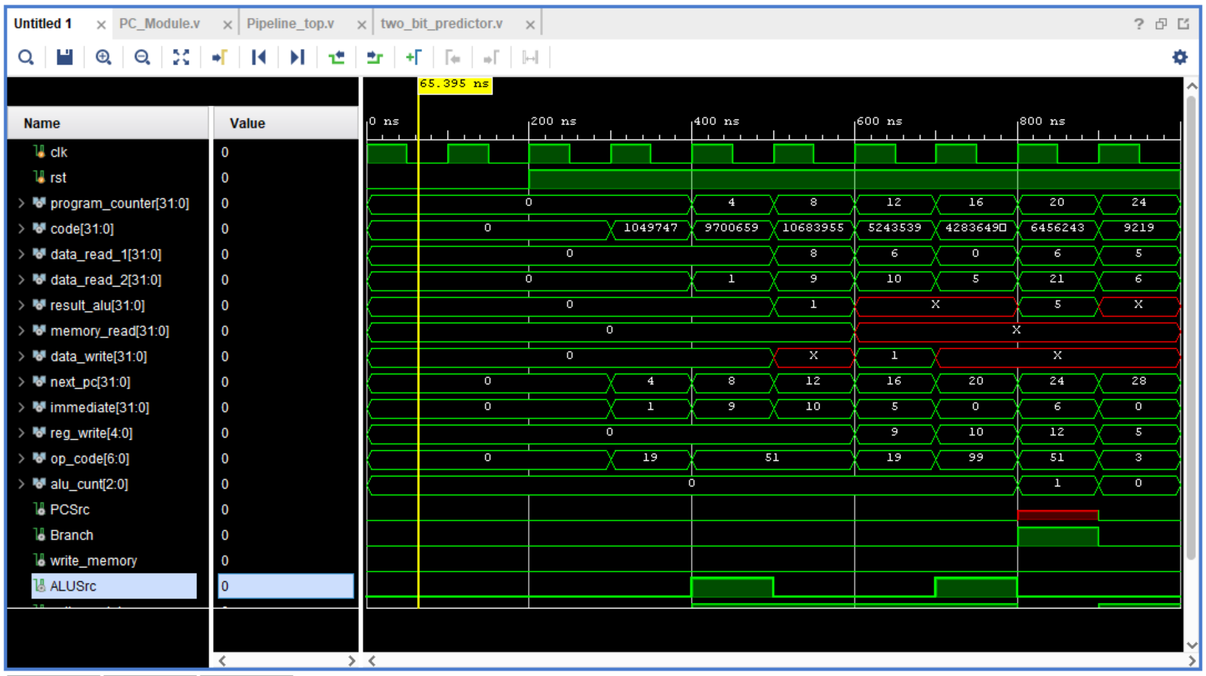
```
    // Declaration of I/O
    input clk, rst;

    // 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] program_counter, code, data_read_1, data_read_2, result_alu,
memory_read, data_write;
    wire [31:0] next_pc, immediate;
    wire [4:0] reg_write;
    wire [6:0]op_code;
    wire [2:0]alu_cunt;
    // wire Zero;
    wire PCSrc, Branch, write_memory, ALUSrc, write_register;

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

    assign program_counter = pipeline.PCD;
```

```verilog
    assign next_pc = pipeline.PCPlus4D;
    assign code = pipeline.instructions;
    assign data_read_1 = pipeline.RS1_E;
    assign data_read_2 = pipeline.RS2_E;
    assign result_alu = pipeline.ALU_ResultM;
    assign memory_read = pipeline.ReadDataW;
    assign data_write = pipeline.WriteDataM;
    assign immediate = pipeline.Decode.Imm_Ext_D;
    assign reg_write = pipeline.RDW;
    // assign Zero = pipeline.Execute.ZeroE;
    assign alu_cunt = pipeline.ALUControlE;
    assign op_code = pipeline.Decode.control.Op;
    assign PCSrc = pipeline.PCSrcE;
    assign Branch = pipeline.BranchE;
    assign write_memory = pipeline.MemWriteE;
    assign ALUSrc = pipeline.ALUSrcE;
    assign write_register = pipeline.RegWriteE;
    always begin
        clk = ~clk;
        #50;
    end

    initial begin
        rst <= 1'b0;
        #200;
        rst <= 1'b1;
        #1500;
        // $finish;
    end

endmodule
```

*Simulation:*



## CPI & Latency 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:

LambdaMamba. "Building a MIPS 5-Stage Pipeline Processor in Verilog," *Medium*, Feb. 23, 2020. [Online]. Available: Link

FPGA4Student. "32-bit Pipelined MIPS Processor in Verilog (Part-3)," *FPGA4Student*, Jun. 20, 2017. [Online]. Available: Link

Hassan Nazeer, "Lecture 3a, ASMD based design," *YouTube*, Mar. 27, 2018. [Online]. Available: Link

Hassan Nazeer, "Lecture 3a, Part-2, ASMD based design," *YouTube*, Apr. 06, 2018. [Online]. Available: Link