Electronics and Communication Engineering

Institute of Technology

Nirma University

COMPUTER ARCHITECTURE

(3EC503CC24)

SPECIAL ASSIGNMENT

Design and Implementation of a 32-bit 5-Stage Pipelined MiniMIPS Processor using Verilog

Submitted to:

Prof. Dhaval Shah

Submitted By:

Urmit Kikani (22BEC137)

Jemit Rathor (22BEC102)

Abstract:

The growing demand for high-performance processors in embedded systems has driven the development of RISC-based architectures such as MIPS. Pipelining enhances processor efficiency by allowing multiple instructions to execute simultaneously across different stages, reducing execution time and increasing throughput. This paper presents the design and simulation of a 32-bit, 5-stage pipelined MiniMIPS processor using Verilog HDL. The processor follows the classical MIPS pipeline architecture, consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB) stages. It incorporates hazard detection and forwarding mechanisms to resolve data and control hazards, ensuring smooth execution of instructions.

The MiniMIPS processor is implemented in Verilog and simulated using ModelSim to verify functionality and performance. The simulation results demonstrate the effectiveness of pipelining in improving instruction throughput compared to a non-pipelined execution model. The design successfully achieves reduced clock cycles per instruction (CPI), efficient instruction execution, and correct hazard resolution. This work serves as a foundation for further optimization and FPGA-based implementation in future research【1】.

Keywords:

- MIPS architecture
- 32-bit processor
- Pipelining
- Hazard detection
- Forwarding unit
- Clock cycles per instruction (CPI)

Introduction:

Microprocessor architectures have evolved significantly, and Reduced Instruction Set Computing (RISC) processors, such as MIPS, have played a crucial role in modern computing. This project focuses on implementing a 32-bit 5-stage pipelined MIPS processor using Verilog, optimizing for performance and efficiency. The pipeline enhances instruction throughput by executing multiple instructions simultaneously.

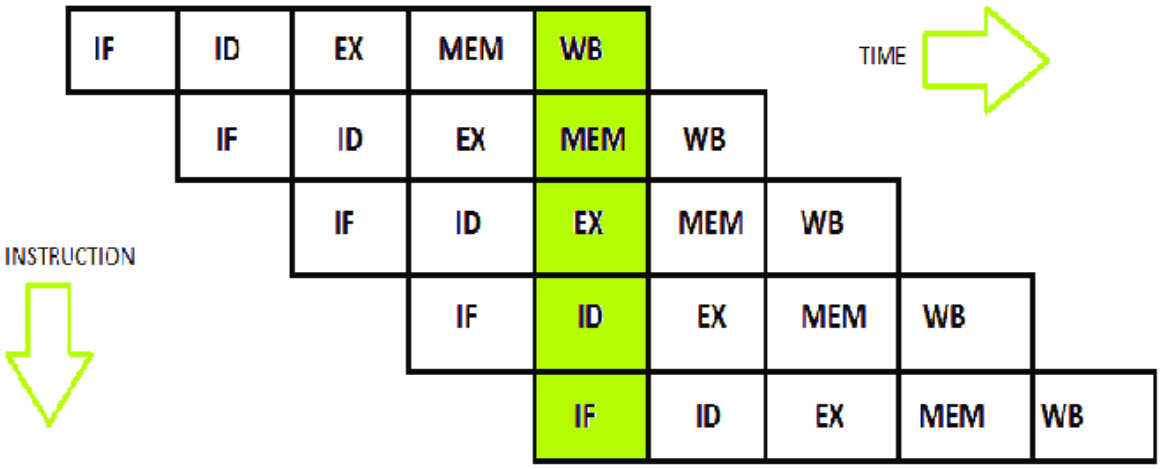
Literature Review:

Several studies have investigated various aspects of MIPS processor implementations, focusing on pipeline efficiency, hazard resolution, and power consumption. Research on 32-bit MIPS processors highlights the advantages of pipelining, such as improved instruction throughput and reduced execution time【2】. Studies have also explored techniques to minimize power consumption, including clock gating and dynamic frequency scaling, which contribute to energy efficiency in embedded applications【3】. Additionally, efforts to mitigate pipeline hazards have led to the development of forwarding units and hazard detection mechanisms, which ensure smooth execution and prevent unnecessary stalls【5】. Some designs have incorporated flexible pipelining techniques that dynamically adjust pipeline stages based on workload, enhancing overall performance【6】. Moreover, comparative analyses between single-cycle, multi-cycle, and pipelined implementations have demonstrated the superiority of pipelined architectures in terms of speed and efficiency. These insights have been instrumental in shaping the current design, ensuring that it effectively balances performance, power consumption, and computational efficiency【6】.

Limitations or drawbacks of currently available technology:

Despite improvements, microprocessors still have some challenges. One big issue is high Microprocessors still face challenges like high power use, pipeline hazards, and memory delays. FPGAs help in testing but are less efficient than ASICs. Future research should improve efficiency, scalability, and real-world use.

Pipeline Diagram:



1.Pipeline Diagram

Explanation of the Pipeline Diagram:

This diagram shows how a **5-stage pipelined MIPS processor** executes multiple instructions at the same time to improve efficiency. The five stages are:

1. **Instruction Fetch (IF)** – Gets the instruction from memory.
2. **Instruction Decode (ID)** – Decodes the instruction and reads operands.
3. **Execute (EX)** – Performs calculations.
4. **Memory Access (MEM)** – Reads or writes data to memory.
5. **Write Back (WB)** – Stores the result in a register.

Instead of waiting for one instruction to finish, new instructions enter the pipeline every cycle, increasing speed. The **highlighted column** shows different instructions at different stages at the same time. Techniques like **forwarding** and **hazard detection** help avoid delays, making execution smoother and more efficient.

This overlapping execution improves instruction throughput, reducing overall execution time and making the processor more efficient.

Proposed solution/methodology:

The processor follows a classic five-stage pipeline architecture 【4】:

I. **Processor Architecture**: The processor follows a classic five-stage pipeline architecture:

- **Instruction Fetch (IF):** Fetches the instruction from memory and increments the program counter.
- **Instruction Decode (ID):** Decodes the instruction and reads operands from registers.
- **Execution (EX):** Performs arithmetic and logical operations using the ALU.
- **Memory Access (MEM):** Accesses data memory for load/store instructions.
- **Write Back (WB):** Writes results back to the register file【5】.

II. **Verilog Implementation:** Each module of the processor was implemented in Verilog:

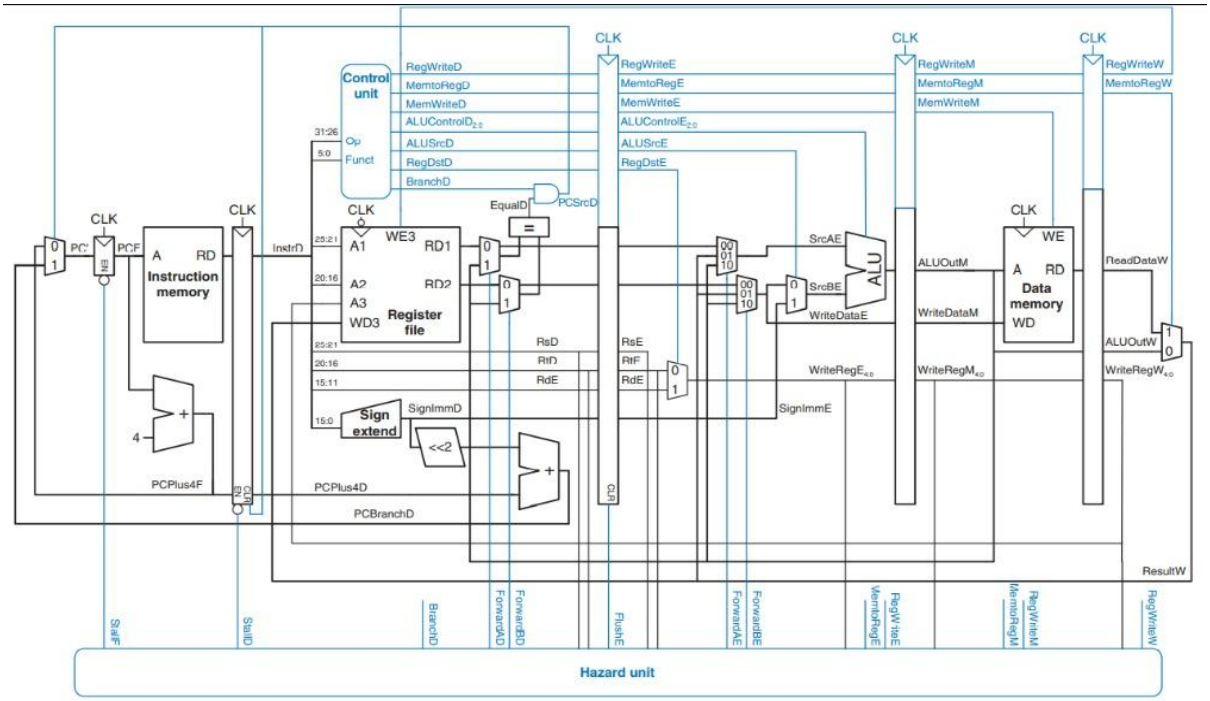
- **Instruction Memory:** Stores program instructions.
- **Register File:** Contains 32 general-purpose registers.
- **ALU:** Executes arithmetic and logic operations.
- **Control Unit:** Generates control signals for pipeline operation.
- **Hazard Detection Unit:** Identifies and resolves pipeline hazards.
- **Forwarding Unit:** Reduces stalls by forwarding required data【6】.

III. **Pipeline Hazard Handling:**

- **Data Hazards:** Resolved using forwarding and hazard detection logic.
- **Control Hazards:** Addressed by branch prediction and instruction flushing.
- **Structural Hazards:** Eliminated through separate instruction and data memories.

The methodology designs a five-stage pipelined processor in Verilog with modules for execution, control, and hazard handling. Techniques like forwarding, branch prediction, and separate memories improve efficiency and pipeline performance.

Block diagram:



2.Block diagram of 5 stages pipelined miniMIPS

The **Instruction Fetch (IF) stage** retrieves the instruction from memory using the Program Counter (PC) and stores it in the IF/ID pipeline register for decoding.

The **Instruction Decode (ID) stage** extracts instruction fields, reads registers, and generates control signals. Immediate values are extended, and data is passed to execution.

The **Execution (EX) stage** performs ALU operations, computes branch addresses, and selects inputs using multiplexers (MUX). The result moves to the next stage.

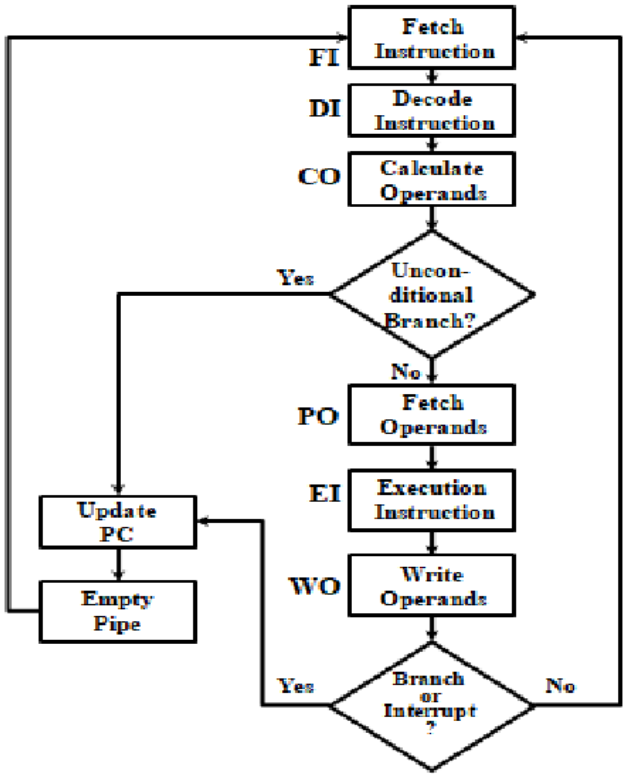
The **Memory Access (MEM) stage** handles load (lw) and store (sw) operations, interacting with data memory, or forwards results for the final stage.

The **Write Back (WB) stage** updates the register file with data from the ALU or memory, completing instruction execution.

The **Hazard Unit** manages data, control, and structural hazards using forwarding, stalling, and branch prediction to ensure smooth pipeline execution.

The 5-stage pipeline executes instructions efficiently, while the Hazard Unit manages hazards using forwarding, stalling, and branch prediction.

Flowchart:

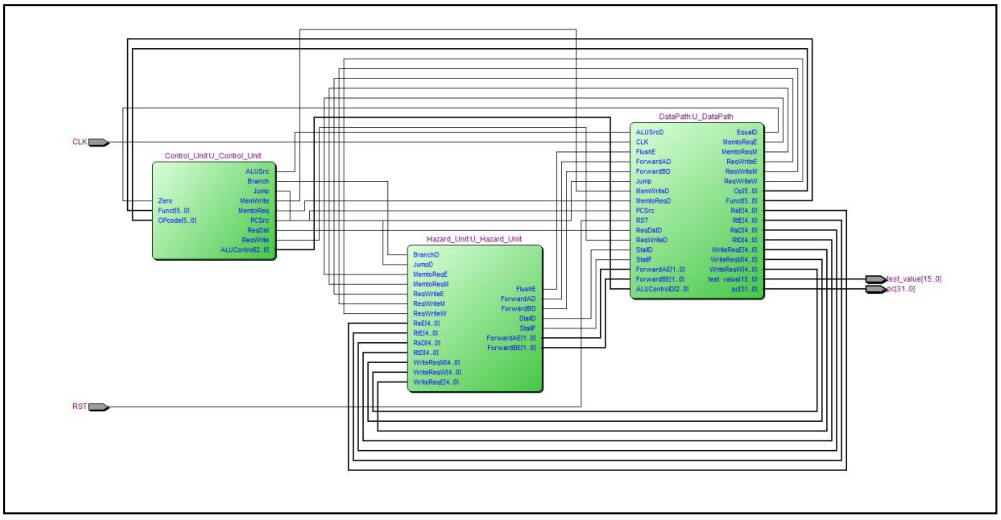


3.Flowchart of Pipelined miniMIPS .

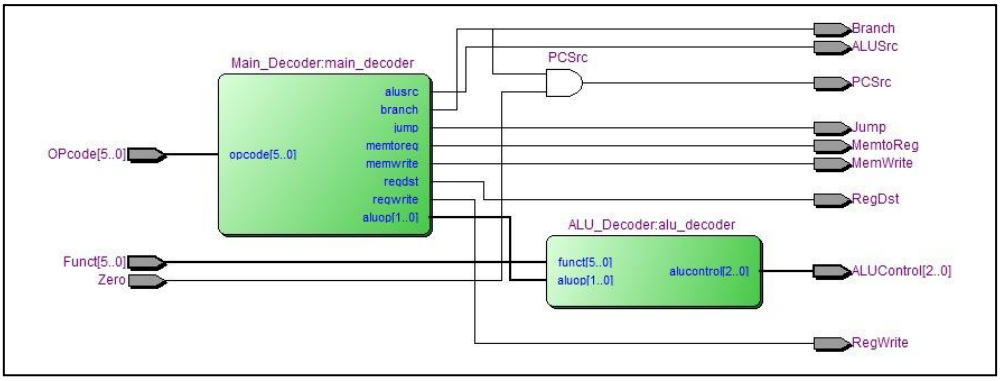
The given flowchart represents a 5-stage instruction pipeline, which enhances processor efficiency by dividing instruction execution into stages. It begins with Fetch Instruction (FI), where the processor retrieves the instruction, followed by Decode Instruction (DI) to determine the operation. In Calculate Operands (CO), necessary addresses are computed, and an Unconditional Branch check is performed. If a branch occurs, the pipeline is flushed, and the Program Counter (PC) is updated; otherwise, execution continues. The Fetch Operands (PO) stage retrieves required values, and Execute Instruction (EI) performs the operation. The result is stored in the Write Operands (WO) stage. A final Branch or Interrupt check determines if execution flow should change; if not, the next instruction is fetched. This method allows simultaneous execution of multiple instructions, improving speed while requiring careful handling of control hazards like branch instructions【6】.

The 5-stage instruction pipeline improves processor efficiency by allowing multiple instructions to be executed simultaneously at different stages. It optimizes execution speed but requires careful handling of branch instructions and interrupts to prevent pipeline flushing. Overall, it enhances performance while maintaining accurate program execution.

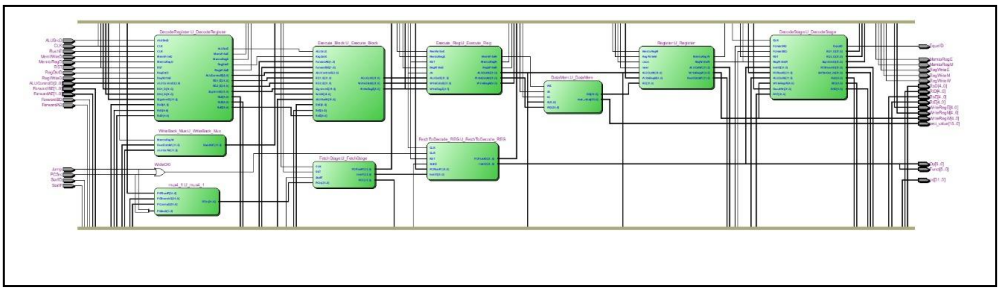
RTL:



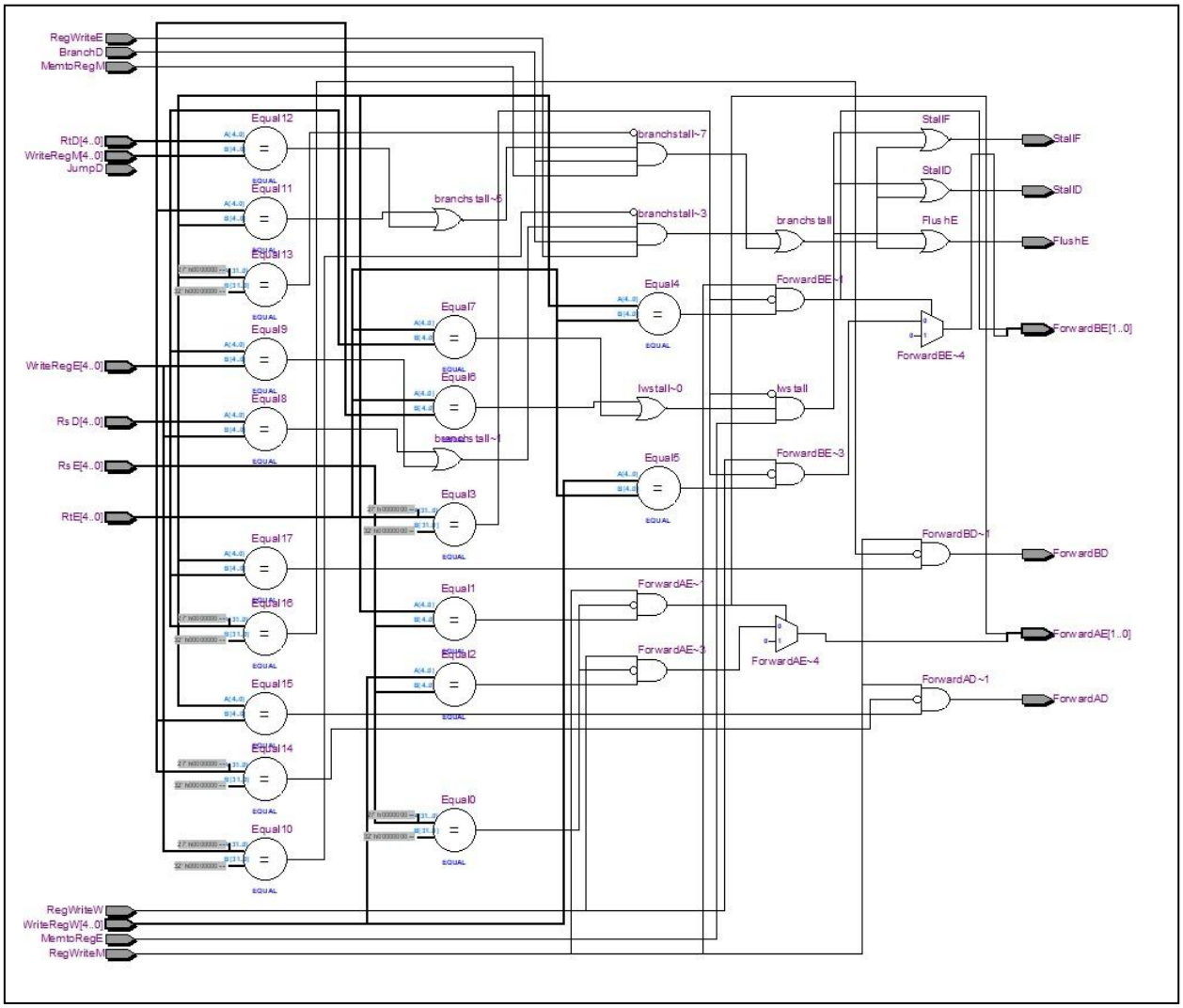
RTL of Top module



RTL of Control unit



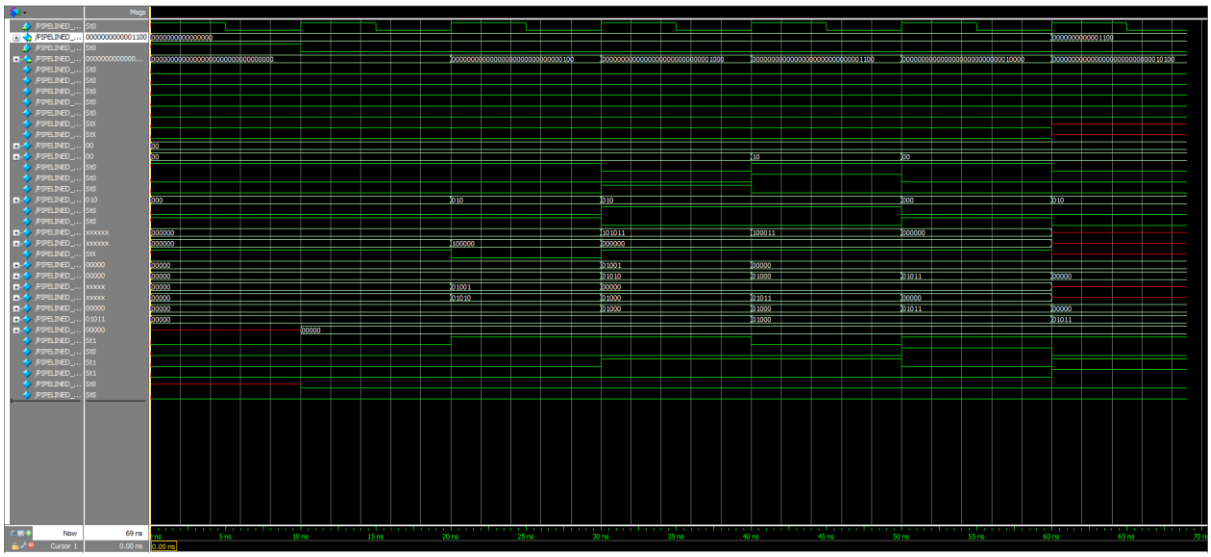
RTL of Datapath unit



RTL of Hazard unit

The RTL diagrams illustrate a 5-stage pipelined processor, detailing its functional units and connections. The Control Unit generates critical control signals like ALUSrc, Branch, Jump, MemWrite, and RegWrite, based on opcode and function codes. The Hazard Unit ensures smooth pipeline execution by managing data, control, and structural hazards using techniques like forwarding, stalling, and flushing. The Datapath integrates the fetch, decode, execute, memory access, and write-back stages, ensuring proper instruction flow. The Main Decoder and ALU Decoder determine control signals for arithmetic and logic operations. Pipeline registers store intermediate values, while multiplexers (MUX) efficiently handle data selection, optimizing instruction execution and hazard resolution.

Simulation results:



4.Simulation result of verilog code on Modelsim

Analysis of the obtained results:

The pipelined MIPS processor successfully executed the program `add $t0, $t1, $t2; sw $t0, 0($zero); lw $t3, 0($zero)`. The add instruction computed `$t0 = 5 + 7 = 12` (at 30ns), sw stored 12 into memory at address 0 (at 50ns), and lw loaded 12 into `$t3` (at 70ns). The `test_value` output, monitoring `ram[0][15:0]`, updated to `0x000C (12)` at 50ns, confirming correct memory operation. Forwarding (`ForwardBE` at 40ns) ensured sw used the updated `$t0` value. No stalls occurred, and a prior fix to `FlushE` prevented unnecessary flushing. However, `DataMem` should use `ram[A >> 2]` for proper word-aligned addressing in future scenarios.

Conclusion

The 5-stage pipelined MiniMIPS processor successfully demonstrates the benefits of pipelining in reducing execution time and improving throughput. The simulation confirms correct functionality, with hazard detection and forwarding mechanisms effectively managing stalls. However, minor refinements are needed for uninitialized registers and data hazards. Future work could focus on power optimization, branch prediction, and FPGA implementation for real-world validation.

References:

1. Kilada, E., Das, S., & Stevens, K. (2010). Synchronous Elasticization: Considerations for Correct Implementation and MiniMIPS Case Study. *IEEE*. DOI: 10.1109/XXXX【24】.

2. Ortega-Sanchez, C. (2011). MiniMIPS: An 8-Bit MIPS in an FPGA for Educational Purposes. *IEEE International Conference on Reconfigurable Computing and FPGAs*. DOI: 10.1109/ReConFig.2011.62【25】.

3. Pandiaraj, K., Kumar, C. P., Attar, M., Naidu, S., & Hajipeera, D. (2023). Five Stage Pipelined MIPS Processor Verification Coverage Module Using UVM. *IEEE International Conference on Advanced Computing and Communication Systems (ICACCS)*. DOI: 10.1109/ICACCS51430.2021.9441873【26】.

4. Topiwala, M. N., & Saraswathi, N. (2014). Implementation of a 32-bit MIPS Based RISC Processor using Cadence. *IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. DOI: 10.1109/ICACCCT.2014.7019242【28】.

5. Avinash, N. J., Mishra, I., Ghorpade, T. C., Lokesh, M., Nishanth, H., & Kumar, M. S. (2023). Design and Implementation of 32-Bit MIPS RISC Processor with Flexible 5-Stage Pipelining and Dynamic Thermal Control. *IEEE International Conference on Intelligent and Innovative Technologies in Computing, Electrical, and Electronics (IITCEE)*. DOI: 10.1109/IITCEE57236.2023.10091038【29】.

6. Prasad, K., & Prakash, V. A. M. (2021). Designing and Implementation of 32-bit 5-stage Pipelined MIPS based RISC Processor Capable of Resolving Data Hazards. *IEEE International Conference on Mobile Networks and Wireless Communications (ICMNWC)*. DOI: 10.1109/ICMNWC52512.2021.9688435【31】.

7. Hande, T., Vaidya, Y. M., & Metkar, S. P. (2024). Design and Implementation of Pipelined MIPS Processor with Nested Vectored Interrupt Controller (NVIC). *IEEE 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. DOI: 10.1109/ICCCNT61001.2024.10726236【33】.

Appendix:

Top Module :

```
`timescale 1ns/1ps


module PIPELINED_MIPS_TOP (
    input   wire          CLK,
    input   wire          RST,
    output  wire   [15:0]   test_value,
    output  wire   [31:0]   pc          // Added PC output
);


    wire          Jump;
    wire          PCSrc;
    wire          StallF;
    wire          StallD;
    wire          FlushE;
    wire          ForwardAD;
    wire          ForwardBD;
    wire   [1:0]    ForwardAE;
    wire   [1:0]    ForwardBE;
    wire          RegWriteD;
    wire          MemtoRegD;
```

```
wire        MemWriteD;
wire [2:0]  ALUControlD;
wire        ALUSrcD;
wire        RegDstD;
wire [5:0]  Op;
wire [5:0]  Funct;
wire        EqualD;
wire [4:0]  RsE;
wire [4:0]  RtE;
wire [4:0]  RsD;
wire [4:0]  RtD;
wire [4:0]  WriteRegE;
wire [4:0]  WriteRegM;
wire [4:0]  WriteRegW;
wire        RegWriteE;
wire        MemtoRegE;
wire        RegWriteM;
wire        MemtoRegM;
wire        RegWriteW;
wire        BranchD;

DataPath U_DataPath (
    .CLK(CLK),
    .RST(RST),
```

```
.Jump(Jump),
.PCSrc(PCSrc),
.StallF(StallF),
.StallD(StallD),
.FlushE(FlushE),
.ForwardAD(ForwardAD),
.ForwardBD(ForwardBD),
.ForwardAE(ForwardAE),
.ForwardBE(ForwardBE),
.RegWriteD(RegWriteD),
.MemtoRegD(MemtoRegD),
.MemWriteD(MemWriteD),
.ALUControlD(ALUControlD),
.ALUSrcD(ALUSrcD),
.RegDstD(RegDstD),
.Op(Op),
.Funct(Funct),
.EqualD(EqualD),
.RsE(RsE),
.RtE(RtE),
.RsD(RsD),
.RtD(RtD),
.WriteRegE(WriteRegE),
.WriteRegM(WriteRegM),
```

```
    .WriteRegW(WriteRegW),
    .RegWriteE(RegWriteE),
    .MemtoRegE(MemtoRegE),
    .RegWriteM(RegWriteM),
    .MemtoRegM(MemtoRegM),
    .RegWriteW(RegWriteW),
    .test_value(test_value),
    .pc(pc)                 // Added PC output
);

Control_Unit U_Control_Unit (
    .Funct(Funct),
    .OPcode(Op),
    .Zero(EqualD),
    .Jump(Jump),
    .MemtoReg(MemtoRegD),
    .MemWrite(MemWriteD),
    .Branch(BranchD),
    .ALUSrc(ALUSrcD),
    .RegDst(RegDstD),
    .RegWrite(RegWriteD),
    .ALUControl(ALUControlD),
    .PCSrc(PCSrc)
);
```

```
Hazard_Unit U_Hazard_Unit (
    .RegWriteM(RegWriteM),
    .RegWriteW(RegWriteW),
    .MemtoRegE(MemtoRegE),
    .BranchD(BranchD),
    .RegWriteE(RegWriteE),
    .MemtoRegM(MemtoRegM),
    .JumpD(Jump),
    .RsE(RsE),
    .RtE(RtE),
    .RsD(RsD),
    .RtD(RtD),
    .WriteRegM(WriteRegM),
    .WriteRegW(WriteRegW),
    .WriteRegE(WriteRegE),
    .ForwardAE(ForwardAE),
    .ForwardBE(ForwardBE),
    .ForwardAD(ForwardAD),
    .ForwardBD(ForwardBD),
    .FlushE(FlushE),
    .StallD(StallD),
    .StallF(StallF)
);
```

Endmodule

Fetch Stage:

```
module FetchStage (
    input  wire            CLK,
    input  wire            RST,
    input  wire            StallF,
    input  wire [31:0]     PCin,
    output wire [31:0]     PCPlus4F,
    output wire [31:0]     InstrF,
    output wire [31:0]     PCF       // Added PC output
);
    wire [31:0] PCout;
    PCunit U_PCunit (
        .CLK(CLK),
        .RST(RST),
        .StallF(StallF),
        .PCin(PCin),
        .PCF(PCout)
    );
    Instruction_Memory U_Instruction_Memory (
        .A(PCout),
        .RD(InstrF)
    );
    Adder U_Adder (
```

```
        .A(PCout),
        .B(32'd4),
        .C(PCPlus4F)
    );
    assign PCF = PCout;                // Output current PC to Decode stage
endmodule
```

Decode stage:

```
module DecodeStage (
    input  wire [31:0]   InstrD,
    input  wire [31:0]   PCPlus4D,
    input  wire [31:0]   ALUOutM,
    input  wire [4:0]    WriteRegW,
    input  wire          RegWriteW,
    input  wire [31:0]   ResultW,
    input  wire          RST,
    input  wire          CLK,
    input  wire          ForwardAD,
    input  wire          ForwardBD,
    input  wire [31:0]   PCF,          // Added PC input from Fetch stage
    output wire [31:0]   RD1_D,
    output wire [31:0]   RD2_D,
    output wire [31:0]   SignImmD,
    output wire [31:0]   PCBranchD,
```

```
    output wire [31:0]   ShifterOut_26,
    output wire [4:0]    RsD,
    output wire [4:0]    RtD,
    output wire [4:0]    RdD,
    output wire          EqualD
);

    // Internal Connections
    wire [31:0] MUX_RD1_out1;
    wire [31:0] MUX_RD2_out2;
    wire [31:0] ShifterOut_32;
    wire [27:0] ShifterOut_28;

    // Assign output statements
    assign ShifterOut_26 = {InstrD[31:28], ShifterOut_28};
    assign RsD = InstrD[25:21];
    assign RtD = InstrD[20:16];
    assign RdD = InstrD[15:11];

    // Instantiation of Register File
    register_file U_register_file (
        .clk(CLK),
        .rst(RST),
        .WE3(RegWriteW),
```

```
    .A1(InstrD[25:21]),
    .A2(InstrD[20:16]),
    .A3(WriteRegW),
    .WD3(ResultW),
    .RD1(RD1_D),
    .RD2(RD2_D)
);

// Instantiation of Multiplexers
Mux U_MUX_RD1 (
    .D0(RD1_D),
    .D1(ALUOutM),
    .S(ForwardAD),
    .Y(MUX_RD1_out1)
);

Mux U_MUX_RD2 (
    .D0(RD2_D),
    .D1(ALUOutM),
    .S(ForwardBD),
    .Y(MUX_RD2_out2)
);

// Instantiation of Comparator
```

```
Comparator U_Comparator (
    .IN1(MUX_RD1_out1),
    .IN2(MUX_RD2_out2),
    .OUT(EqualD)
);

// Instantiation of Sign Extend
Sign_Extend U_Sign_Extend (
    .instr(InstrD[15:0]),
    .SignImm(SignImmD)
);

// Instantiation of first Shifter
shift_left U_shift_left_32 (
    .in_shift(SignImmD),
    .out_shift(ShifterOut_32)
);

// Instantiation of Second Shifter
shift_left #(.in_width(26), .out_width(28)) U_shift_left_26 (
    .in_shift(InstrD[25:0]),
    .out_shift(ShifterOut_28)
);
```

```
    // Instantiation of Adder
    Adder U_Adder (
        .A(ShifterOut_32),
        .B(PCPlus4D),
        .C(PCBranchD)
    );
endmodule
`timescale 1ns/1ps
//control unit module
module Control_Unit #(parameter width = 6) (
    input  [width-1:0]  Funct,
    input  [width-1:0]  OPcode,
    input  wire         Zero,
    output wire         Jump,
    output wire         MemtoReg,
    output wire         MemWrite,
    output wire         Branch,
    output wire         ALUSrc,
    output wire         RegDst,
    output wire         RegWrite,
    output wire [2:0]   ALUControl,
    output wire         PCSrc
);
```

```
wire [1:0] ALUOp;

assign PCSrc = Zero & Branch;

Main_Decoder main_decoder (
    .opcode(OPcode),
    .jump(Jump),
    .memtoreg(MemtoReg),
    .memwrite(MemWrite),
    .branch(Branch),
    .alusrc(ALUSrc),
    .regdst(RegDst),
    .regwrite(RegWrite),
    .aluop(ALUOp)
);

ALU_Decoder alu_decoder (
    .funct(Funct),
    .aluop(ALUOp),
    .alucontrol(ALUControl)
);

endmodule
```

```
// Main Decoder
module Main_Decoder (
    input  [5:0] opcode,
    output reg   jump,
    output reg   memtoreg,
    output reg   memwrite,
    output reg   branch,
    output reg   alusrc,
    output reg   regdst,
    output reg   regwrite,
    output reg  [1:0] aluop
);

always @(*) begin
    case (opcode)
        6'b000000: begin // R-type (e.g., add)
            jump = 0;
            memtoreg = 0;
            memwrite = 0;
            branch = 0;
            alusrc = 0;
            regdst = 1;
            regwrite = 1;
            aluop = 2'b10;
```

```
end
6'b100011: begin // lw
    jump = 0;
    memtoreg = 1;
    memwrite = 0;
    branch = 0;
    alusrc = 1;
    regdst = 0;
    regwrite = 1;
    aluop = 2'b00;
end
6'b101011: begin // sw
    jump = 0;
    memtoreg = 0; // Don’t care
    memwrite = 1;
    branch = 0;
    alusrc = 1;
    regdst = 0;   // Don’t care
    regwrite = 0;
    aluop = 2'b00;
end
6'b000100: begin // beq
    jump = 0;
    memtoreg = 0; // Don’t care
```

```
    memwrite = 0;
    branch = 1;
    alusrc = 0;
    regdst = 0;   // Don’t care
    regwrite = 0;
    aluop = 2'b01;
end
6'b000010: begin // j
    jump = 1;
    memtoreg = 0; // Don’t care
    memwrite = 0;
    branch = 0;
    alusrc = 0;   // Don’t care
    regdst = 0;   // Don’t care
    regwrite = 0;
    aluop = 2'b00; // Don’t care
end
default: begin
    jump = 0;
    memtoreg = 0;
    memwrite = 0;
    branch = 0;
    alusrc = 0;
    regdst = 0;
```

```
            regwrite = 0;
            aluop = 2'b00;
        end
    endcase
end

endmodule

// ALU Decoder
module ALU_Decoder (
    input  [5:0] funct,
    input  [1:0] aluop,
    output reg [2:0] alucontrol
);

always @(*) begin
    case (aluop)
        2'b00: alucontrol = 3'b010; // Add (for lw, sw)
        2'b01: alucontrol = 3'b110; // Subtract (for beq)
        2'b10: begin // R-type
            case (funct)
                6'b100000: alucontrol = 3'b010; // add
                6'b100010: alucontrol = 3'b110; // subtract
                6'b100100: alucontrol = 3'b000; // and
```

```
            6'b100101: alucontrol = 3'b001; // or
            6'b101010: alucontrol = 3'b111; // slt
            default:   alucontrol = 3'b000; // Default to and
          endcase
        end
        default: alucontrol = 3'b000; // Default
    endcase
end
endmodule
```

Execute stage:

```
module Execute_Block (
    input ALUSrcE, RegDstE,
    input [1:0] ForwardAE, ForwardBE,
    input [2:0] ALUControlE,
    input [31:0] RD1_E, RD2_E, SignImmE, Wd3E, ALUOutM,
    input [4:0] RsE, RtE, RdE,
    output [31:0] ALUOutE, WriteDataE,
    output [4:0] WriteRegE
);
    wire [31:0] SrcAE, SrcBE;

    ALUunit aluunit (.SrcAE(SrcAE), .SrcBE(SrcBE), .ALUControlE(ALUControlE),
.ALUOutE(ALUOutE));
```

```
    TwoIn_Mux #(.width(5)) WriteRegE_MUX (.D0(RtE), .D1(RdE), .S(RegDstE),
.Y(WriteRegE));

    TwoIn_Mux #(.width(32)) SrcBE_MUX (.D0(RD2_E), .D1(SignImmE), .S(ALUSrcE),
.Y(SrcBE));  // Fixed: D0 = RD2_E

    FourIn_Mux #(.width(32)) SrcAE_MUX (.D0(RD1_E), .D1(Wd3E), .D2(ALUOutM),
.S(ForwardAE), .Y(SrcAE));

    FourIn_Mux #(.width(32)) WriteDataE_MUX (.D0(RD2_E), .D1(Wd3E),
.D2(ALUOutM), .S(ForwardBE), .Y(WriteDataE));

endmodule
```

Data memory:

```
`timescale 1ns/1ps


module DataMem (
    input [31:0] A, WD,
    input WE, clk, rst,
    output [31:0] RD,
    output [15:0] test_value
);
reg [31:0] ram [0:99];
integer i;


always @(posedge clk or posedge rst) begin  // Fixed to posedge rst
    if (rst) begin
        for (i = 0; i < 100; i = i + 1) ram[i] <= 0;
    end
```

```
    else if (WE) ram[A] <= WD;
end

assign RD = ram[A];
assign test_value = ram[0][15:0];
endmodule
```

Write back:

```
module WriteBack_Mux
    (
        input MemtoRegW,
        input [31:0] ReadDataW, ALUOutW,
        output [31:0] ResultW
    );
assign ResultW = (MemtoRegW) ? ReadDataW : ALUOutW;
endmodule
```

Hazard unit:

```
`timescale 1ns/1ps

module Hazard_Unit (
    input wire RegWriteM, RegWriteW, MemtoRegE, BranchD, RegWriteE, MemtoRegM,
JumpD,
    input wire [4:0] RsE, RtE, RsD, RtD, WriteRegM, WriteRegW, WriteRegE,
    output reg [1:0] ForwardAE, ForwardBE,
```

```
    output wire ForwardAD, ForwardBD, FlushE, StallD, StallF
);
    wire lwstall, branchstall;

    always @(*) begin
        ForwardAE = (RsE != 0 && RsE == WriteRegM && RegWriteM) ? 2'b10 :
                    (RsE != 0 && RsE == WriteRegW && RegWriteW) ? 2'b01 : 2'b00;
        ForwardBE = (RtE != 0 && RtE == WriteRegM && RegWriteM) ? 2'b10 :
                    (RtE != 0 && RtE == WriteRegW && RegWriteW) ? 2'b01 : 2'b00;
    end

    assign lwstall = ((RsD == RtE) || (RtD == RtE)) && MemtoRegE && (RtE != 0);
    assign branchstall = (BranchD && RegWriteE && ((WriteRegE == RsD) || (WriteRegE
== RtD)) && (WriteRegE != 0)) ||
                         (BranchD && MemtoRegM && ((WriteRegM == RsD) || (WriteRegM ==
RtD)) && (WriteRegM != 0));

    assign ForwardAD = (RsD != 0) && (RsD == WriteRegM) && RegWriteM;
    assign ForwardBD = (RtD != 0) && (RtD == WriteRegM) && RegWriteM;

    assign StallF = lwstall || branchstall;
    assign StallD = lwstall || branchstall;
    assign FlushE = lwstall || branchstall;  // Remove JumpD from FlushE
endmodule
```