Verilog

```
//==============================//
// 4-STAGE PIPELINED PROCESSOR //
//==============================//

module Pipelined_Processor(clk, reset);
  input clk, reset;

  reg [31:0] PC;
  reg [31:0] InstructionMemory [0:15];
  reg [31:0] DataMemory [0:15];
  reg [31:0] RegisterFile [0:7];

  // Pipeline registers
  reg [31:0] IF_ID_IR;
  reg [31:0] ID_EX_A, ID_EX_B, ID_EX_IR;
  reg [31:0] EX_WB_IR, EX_WB_ALUOUT;
  reg [31:0] EX_WB_LMD;

  wire [5:0] opcode;
  assign opcode = IF_ID_IR[31:26];

  //==========================
  // Stage 1: Instruction Fetch
  //==========================
  always @(posedge clk or posedge reset)
  begin
    if (reset) begin
      PC <= 0;
      IF_ID_IR <= 0;
    end else begin
      IF_ID_IR <= InstructionMemory[PC];
      PC <= PC + 1;
    end
  end

  //==========================
  // Stage 2: Instruction Decode
  //==========================
  always @(posedge clk)
  begin
    ID_EX_IR <= IF_ID_IR;
    ID_EX_A  <= RegisterFile[IF_ID_IR[25:21]];
    ID_EX_B  <= RegisterFile[IF_ID_IR[20:16]];
  end

  //==========================
  // Stage 3: Execute
  //==========================
  reg [31:0] ALUOut;
  always @(posedge clk)
  begin
    EX_WB_IR <= ID_EX_IR;
    case (ID_EX_IR[31:26])
      6'b000000:  // ADD
```

```
        ALUOut <= ID_EX_A + ID_EX_B;
      6'b000001:  // SUB
        ALUOut <= ID_EX_A - ID_EX_B;
      6'b000010:  // LOAD
        ALUOut <= DataMemory[ID_EX_IR[15:0]]; // Load from
memory
      default:
        ALUOut <= 0;
    endcase
    EX_WB_ALUOUT <= ALUOut;
    EX_WB_LMD <= ALUOut;
  end

  //==========================
  // Stage 4: Write Back
  //==========================
  always @(posedge clk)
  begin
    case (EX_WB_IR[31:26])
      6'b000000: RegisterFile[EX_WB_IR[15:11]] <=
EX_WB_ALUOUT; // ADD
      6'b000001: RegisterFile[EX_WB_IR[15:11]] <=
EX_WB_ALUOUT; // SUB
      6'b000010: RegisterFile[EX_WB_IR[20:16]] <=
EX_WB_LMD;    // LOAD
    endcase
  end

  //==========================
  // Program Initialization
  //==========================
  initial begin
    // Data Memory
    DataMemory[0] = 10;
    DataMemory[1] = 20;

    // Register file init
    RegisterFile[1] = 0;
    RegisterFile[2] = 0;
    RegisterFile[3] = 0;
    RegisterFile[4] = 0;

    // Instruction Memory
    // Opcode, Rs, Rt, Rd fields
    // Format: opcode(6) Rs(5) Rt(5) Rd(5) immediate(11)

    InstructionMemory[0] =
32'b000010_00000_00001_00000000000000000; // LOAD R1, 0
    InstructionMemory[1] =
32'b000010_00000_00010_00000000000000001; // LOAD R2, 1
    InstructionMemory[2] =
32'b000000_00001_00010_00011_00000000000; // ADD R3 = R1 +
R2
    InstructionMemory[3] =
32'b000001_00010_00001_00100_00000000000; // SUB R4 = R2 -
R1
  end
```

```
  //==========================
  // Monitor Output
  //==========================
  always @(posedge clk)
  begin
    $display("Time=%0t | PC=%0d | R1=%0d | R2=%0d | R3=%0d
| R4=%0d",
             $time, PC, RegisterFile[1], RegisterFile[2],
RegisterFile[3], RegisterFile[4]);
  end

endmodule


//==========================//
//        TESTBENCH         //
//==========================//

module TB_Pipelined_Processor;
  reg clk, reset;
  Pipelined_Processor cpu(clk, reset);

  initial begin
    clk = 0;
    reset = 1;
    #5 reset = 0;
  end

  always #5 clk = ~clk;

  initial begin
    #80 $finish;
  end
endmodule
```

```
Time=10 | PC=1 | R1=0 | R2=0 | R3=0 | R4=0
Time=20 | PC=2 | R1=10 | R2=0 | R3=0 | R4=0
Time=30 | PC=3 | R1=10 | R2=20 | R3=0 | R4=0
Time=40 | PC=4 | R1=10 | R2=20 | R3=30 | R4=0
Time=50 | PC=5 | R1=10 | R2=20 | R3=30 | R4=10
```

Verilog

```
// pipeline4_debug.v
`timescale 1ns/1ps

// ------------------------------
// Simple Register File
// ------------------------------
module regfile(
    input clk,
    input we,
    input [4:0] ra1, ra2, wa,
    input [31:0] wd,
    output [31:0] rd1, rd2
);
    reg [31:0] rf [0:31];
    integer i;
    initial begin
        for (i=0;i<32;i=i+1) rf[i]=0;
    end
    assign rd1 = (ra1==0) ? 32'b0 : rf[ra1];
    assign rd2 = (ra2==0) ? 32'b0 : rf[ra2];

    always @(posedge clk) begin
        if (we && (wa != 0)) rf[wa] <= wd;
    end
endmodule

// ------------------------------
// Instruction Memory (small)
// ------------------------------
module instr_mem(
    input [31:0] addr,
    output [31:0] instr
);
    reg [31:0] rom [0:63];
    initial begin
        // word-addressed: rom[0] -> PC=0, rom[1] -> PC=4
etc (we use PC/4 indexing)
        // Program:
        // lw r1, 0(r0)      // r1 <- 10
        // lw r2, 4(r0)      // r2 <- 20
        // add r3, r1, r2    // r3 = r1 + r2
        // sub r4, r2, r1    // r4 = r2 - r1
        // add r5, r3, r4    // r5 = r3 + r4
        // NOPs...
        rom[0] =
32'b100011_00000_00001_0000000000000000; // lw r1,0(r0)
        rom[1] =
32'b100011_00000_00010_0000000000000100; // lw r2,4(r0)
        rom[2] =
32'b000000_00001_00010_00011_00000_100000; // add r3,r1,r2
        rom[3] =
32'b000000_00010_00001_00100_00000_100010; // sub r4,r2,r1
        rom[4] =
```

```
32'b000000_00011_00100_00101_00000_100000; // add r5,r3,r4
        rom[5] =
32'b000000_00000_00000_00000_00000_000000; // nop
        // rest zeros
        integer i;
        for(i=6;i<64;i=i+1) rom[i]=32'b0;
    end
    assign instr = rom[addr[31:2]]; // assume PC is byte
address, use word index
endmodule

// ------------------------------
// Data Memory (simple read/write)
// ------------------------------
module data_mem(
    input clk,
    input we,
    input [31:0] addr,
    input [31:0] wd,
    output [31:0] rd
);
    reg [31:0] ram [0:63];
    integer i;
    initial begin
        for (i=0;i<64;i=i+1) ram[i]=0;
        ram[0] = 32'd10; // mem[0]
        ram[1] = 32'd20; // mem[1]
    end
    assign rd = ram[addr[31:2]];
    always @(posedge clk) begin
        if (we) ram[addr[31:2]] <= wd;
    end
endmodule

// ------------------------------
// ALU: add/sub
// ------------------------------
module alu(
    input [31:0] a, b,
    input [1:0] op, // 00 add, 01 sub
    output reg [31:0] y
);
    always @(*) begin
        case(op)
            2'b00: y = a + b;
            2'b01: y = a - b;
            default: y = 32'b0;
        endcase
    end
endmodule

// ------------------------------
// 4-stage pipelined CPU
// ------------------------------
module cpu4_debug (
    input clk,
    input rst
```

```
);
    // Program counter (byte address)
    reg [31:0] pc;
    wire [31:0] instr_if;
    instr_mem imem(.addr(pc), .instr(instr_if));

    // IF/ID pipeline registers
    reg [31:0] IFID_instr;
    reg [31:0] IFID_pc;

    // ID stage decode wires
    wire [5:0] id_opcode = IFID_instr[31:26];
    wire [4:0] id_rs = IFID_instr[25:21];
    wire [4:0] id_rt = IFID_instr[20:16];
    wire [4:0] id_rd = IFID_instr[15:11];
    wire [15:0] id_imm = IFID_instr[15:0];

    // Register file
    wire [31:0] rf_rd1, rf_rd2;
    regfile
rf(.clk(clk), .we(wb_reg_write), .ra1(id_rs), .ra2(id_rt),
 .wa(wb_rd), .wd(wb_wd), .rd1(rf_rd1), .rd2(rf_rd2));

    // ID/EX pipeline registers
    reg [31:0] IDEX_pc;
    reg [31:0] IDEX_rd1, IDEX_rd2;
    reg [4:0] IDEX_rs, IDEX_rt, IDEX_rd;
    reg IDEX_isR, IDEX_isLW;
    reg [1:0] IDEX_aluop;
    reg [31:0] IDEX_imm_se;

    // EX stage wires
    wire [31:0] ex_alu_in1;
    wire [31:0] ex_alu_in2_pre;
    wire [31:0] ex_alu_in2;
    wire [31:0] alu_out;

    // Simple WB->EX forwarding: if WB will write to a reg
used by EX, forward wb_wd
    wire forwardA = (wb_reg_write && (wb_rd != 0) &&
(wb_rd == IDEX_rs));
    wire forwardB = (wb_reg_write && (wb_rd != 0) &&
(wb_rd == IDEX_rt));

    assign ex_alu_in1 = forwardA ? wb_wd : IDEX_rd1;
    assign ex_alu_in2_pre = forwardB ? wb_wd : IDEX_rd2;
    // if LW: use sign-extended immediate as second ALU
input for address calc
    assign ex_alu_in2 = IDEX_isLW ? IDEX_imm_se :
ex_alu_in2_pre;

    alu
alu0(.a(ex_alu_in1), .b(ex_alu_in2), .op(IDEX_aluop), .y(a
lu_out));

    // Data memory (reads in EX stage for simplicity)
    wire [31:0] data_rd;
```

```
    data_mem dmem(.clk(clk), .we(1'b0 /* no store in this
demo
*/), .addr(alu_out), .wd(ex_alu_in2_pre), .rd(data_rd));

    // EX/WB pipeline registers
    reg [31:0] EXWB_alu;
    reg [31:0] EXWB_memrd;
    reg [4:0] EXWB_dest;
    reg EXWB_isLW;
    reg EXWB_reg_write;

    // WB wires
    wire [31:0] wb_wd = EXWB_isLW ? EXWB_memrd : EXWB_alu;
    wire [4:0] wb_rd = EXWB_dest;
    wire wb_reg_write = EXWB_reg_write;

    // Hazard detection: load-use hazard (simple case)
    wire load_use_hazard = IDEX_isLW && ((IDEX_rt ==
id_rs) || (IDEX_rt == id_rt)) && (IFID_instr != 32'b0);

    // Control signals generation (combinational based on
IF/ID.instr)
    reg ctrl_isR, ctrl_isLW;
    reg [1:0] ctrl_aluop;
    always @(*) begin
        ctrl_isR = 0; ctrl_isLW = 0; ctrl_aluop = 2'b00;
        if (IFID_instr == 32'b0) begin
            ctrl_isR = 0; ctrl_isLW = 0; ctrl_aluop =
2'b00;
        end else begin
            case (id_opcode)
                6'b000000: begin // R-type
                    ctrl_isR = 1;
                    case (IFID_instr[5:0])
                        6'b100000: ctrl_aluop = 2'b00; //
ADD
                        6'b100010: ctrl_aluop = 2'b01; //
SUB
                        default: ctrl_aluop = 2'b00;
                    endcase
                end
                6'b100011: begin // lw
                    ctrl_isLW = 1;
                    ctrl_aluop = 2'b00; // add base+imm
                end
                default: begin
                    ctrl_isR = 0; ctrl_isLW = 0;
ctrl_aluop = 2'b00;
                end
            endcase
        end
    end

    // sign-extend immediate (combinational)
    wire [31:0] imm_se = {{16{IFID_instr[15]}},
IFID_instr[15:0]};
```

```
    // ---------------------------
    // Pipeline register updates
    // ---------------------------
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pc <= 0;
            IFID_instr <= 32'b0; IFID_pc <= 32'b0;
            IDEX_pc <= 0; IDEX_rd1 <= 0; IDEX_rd2 <= 0;
            IDEX_rs <= 0; IDEX_rt <= 0; IDEX_rd <= 0;
            IDEX_isR <= 0; IDEX_isLW <= 0; IDEX_aluop <=
0; IDEX_imm_se <= 0;
            EXWB_alu <= 0; EXWB_memrd <= 0; EXWB_dest <=
0; EXWB_isLW <= 0; EXWB_reg_write <= 0;
        end else begin
            // IF stage: if hazard -> stall PC and IF/ID
(insert bubble into ID/EX)
            if (!load_use_hazard) begin
                IFID_instr <= instr_if;
                IFID_pc <= pc;
                pc <= pc + 4;
            end else begin
                // hold IFID and pc (stall)
                IFID_instr <= IFID_instr;
                IFID_pc <= IFID_pc;
                pc <= pc; // hold
            end

            // ID -> IDEX
            if (load_use_hazard) begin
                // insert bubble into IDEX
                IDEX_pc <= 0;
                IDEX_rd1 <= 0; IDEX_rd2 <= 0;
                IDEX_rs <= 0; IDEX_rt <= 0; IDEX_rd <= 0;
                IDEX_isR <= 0; IDEX_isLW <= 0; IDEX_aluop
<= 0;
                IDEX_imm_se <= 0;
            end else begin
                IDEX_pc <= IFID_pc;
                IDEX_rd1 <= rf_rd1;
                IDEX_rd2 <= rf_rd2;
                IDEX_rs <= id_rs;
                IDEX_rt <= id_rt;
                IDEX_rd <= id_rd;
                IDEX_isR <= ctrl_isR;
                IDEX_isLW <= ctrl_isLW;
                IDEX_aluop <= ctrl_aluop;
                IDEX_imm_se <= imm_se;
            end

            // EX -> EXWB
            EXWB_alu <= alu_out;
            EXWB_memrd <= data_rd;
            EXWB_isLW <= IDEX_isLW;
            EXWB_dest <= (IDEX_isR ? IDEX_rd : IDEX_rt);
            EXWB_reg_write <= (IDEX_isR | IDEX_isLW);
        end
    end
```

```
    // --------------------------
    // For observability: print per-cycle status (after
posedge updates)
    // Use small delay (#1) to show new latched values
    // --------------------------
    always @(posedge clk) begin
        #1;

$display("------------------------------------------------
--------------------");
        $display("Time=%0t | PC=%0d", $time, pc);
        $display("IF stage: fetched instr = 0x%h",
instr_if);
        $display("IF/ID: instr=0x%h  pc=%0d", IFID_instr,
IFID_pc);

        $display("ID stage:  opcode=0x%0h rs=%0d rt=%0d
rd=%0d imm=0x%h", id_opcode, id_rs, id_rt, id_rd, id_imm);
        $display("ID: rf_rd1=%0d rf_rd2=%0d ctrl_isR=%0b
ctrl_isLW=%0b aluop=%0b", rf_rd1, rf_rd2, ctrl_isR,
ctrl_isLW, ctrl_aluop);

        $display("ID/EX: pc=%0d rs=%0d rt=%0d rd=%0d
rd1=%0d rd2=%0d isR=%0b isLW=%0b aluop=%0b imm_se=%0d",
                 IDEX_pc, IDEX_rs, IDEX_rt, IDEX_rd,
IDEX_rd1, IDEX_rd2, IDEX_isR, IDEX_isLW, IDEX_aluop,
IDEX_imm_se);

        $display("EX stage: alu_in1=%0d alu_in2=%0d
alu_out=%0d (forwardA=%0b forwardB=%0b)",
                 ex_alu_in1, ex_alu_in2, alu_out,
forwardA, forwardB);
        $display("Data mem read (if load) = %0d",
data_rd);

        $display("EX/WB: alu=%0d memrd=%0d dest=%0d
isLW=%0b will_write=%0b", EXWB_alu, EXWB_memrd, EXWB_dest,
EXWB_isLW, EXWB_reg_write);

        $display("WB stage: will write reg %0d with data
%0d (if write=%0b)", wb_rd, wb_wd, wb_reg_write);

        // Print a few registers for convenience
        $display("Registers r0..r6: r0=%0d r1=%0d r2=%0d
r3=%0d r4=%0d r5=%0d r6=%0d",
                 rf.rf[0], rf.rf[1], rf.rf[2], rf.rf[3],
rf.rf[4], rf.rf[5], rf.rf[6]);
    end

endmodule
```

```
// ------------------------------
// Testbench
// ------------------------------
module tb;
    reg clk, rst;
    cpu4_debug cpu(.clk(clk), .rst(rst));

    initial begin
        clk = 0; rst = 1;
        #12 rst = 0; // release reset after a few ns
        #400 $finish; // run long enough
    end

    always #5 clk = ~clk; // 10 ns period

    // Waveform dump
    initial begin
        $dumpfile("pipeline4_debug.vcd");
        $dumpvars(0, tb);
    end
endmodule
```

```
------------------------------------------------------------
----------
Time=21 | PC=4
IF stage: fetched instr = 0x8c010000
IF/ID: instr=0x8c010000  pc=0
ID stage:  opcode=0x23 rs=0 rt=1 rd=0 imm=0x0000
ID: rf_rd1=0 rf_rd2=0 ctrl_isR=0 ctrl_isLW=1 aluop=00
ID/EX: pc=0 rs=0 rt=1 rd=0 rd1=0 rd2=0 isR=0 isLW=1
aluop=00 imm_se=0
EX stage: alu_in1=0 alu_in2=0 alu_out=10 (forwardA=0
forwardB=0)
Data mem read (if load) = 10
EX/WB: alu=10 memrd=10 dest=1 isLW=1 will_write=1
WB stage: will write reg 1 with data 10 (if write=1)
Registers r0..r6: r0=0 r1=0 r2=0 r3=0 r4=0 r5=0 r6=0
------------------------------------------------------------
----------
Time=31 | PC=8
...
```