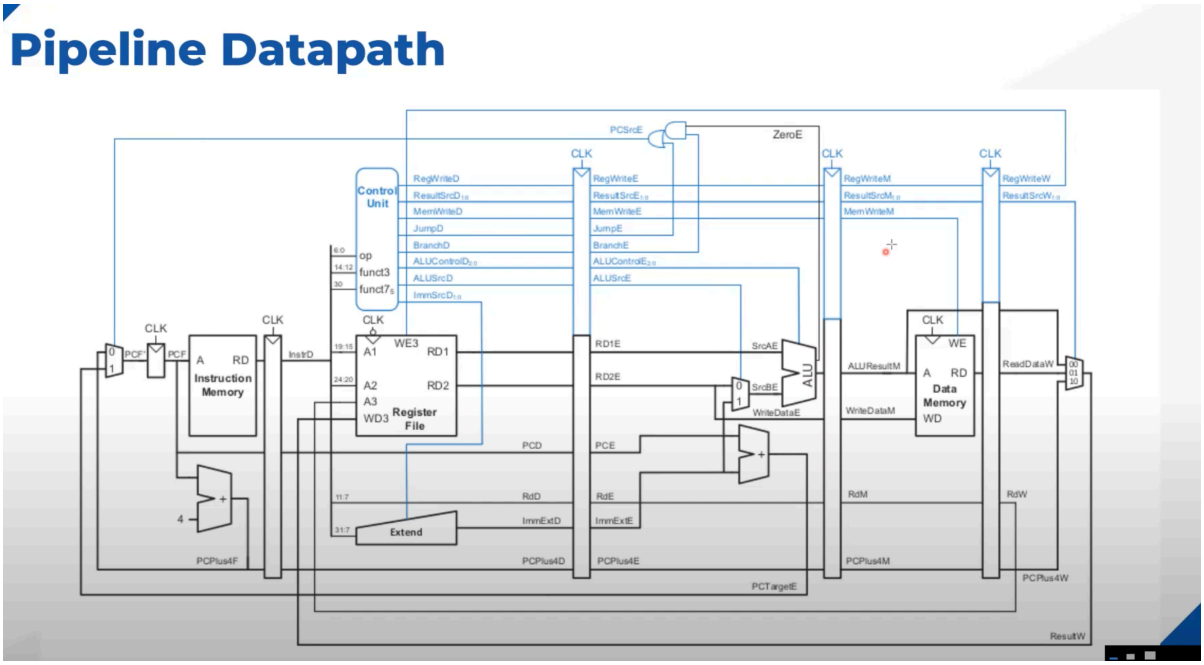
RISC-V 5-stage pipelined processor



Caption

```
`timescale 1ns/1ps

// -------------------- Small libs --------------------
module mux2 #(parameter W=32)(input [W-1:0] a, input [W-1:0] b, input sel, output [W-1:0] y);
  assign y = sel ? b : a;
endmodule

// -------------------- Memories & PC -----------------
module PC_module(input clk, input rst, input [31:0] PC_next, output reg [31:0] PC);
  always @(posedge clk) begin
    if (~rst) PC <= 32'b0; else PC <= PC_next;
  end
endmodule

module instruction_memory(input rst, input [31:0] A, output [31:0] RD);
  reg [31:0] mem[0:1023];
  assign RD = (~rst) ? 32'b0 : mem[A[31:2]];
  initial begin : init_imem
    integer i; for (i=0;i<1024;i=i+1) mem[i]=32'h00000013; // NOPs
    // $readmemh("memfile.hex", mem); // uncomment when you add a file
  end
endmodule

module data_memory(input clk,input rst,input mem_write,input mem_read,
                   input [31:0] addr,input [31:0] wd, output reg [31:0] rd);
  reg [31:0] dmem[0:1023];
  always @(posedge clk) begin
    if (mem_write) dmem[addr[11:2]] <= wd;
    if (mem_read)  rd <= dmem[addr[11:2]];
  end
  integer i; initial for (i=0;i<1024;i=i+1) dmem[i]=32'b0;
endmodule

// -------------------- Register file -----------------
module register_file(input clk,input rst,input WE3,input [4:0] A1,input [4:0] A2,input [4:0] A3,
                     input [31:0] WD3, output [31:0] RD1, output [31:0] RD2);
  reg [31:0] rf[0:31];
  integer i;
  always @(posedge clk) begin
    if (~rst) begin for (i=0;i<32;i=i+1) rf[i]<=32'b0; end
    else if (WE3 && A3!=0) rf[A3] <= WD3;
  end
  assign RD1 = (A1==0) ? 32'b0 : rf[A1];
  assign RD2 = (A2==0) ? 32'b0 : rf[A2];
endmodule

// -------------------- Control, imm, ALU -------------
module control_unit(
  input [6:0] op, input [2:0] funct3, input [6:0] funct7,
  output reg RegWrite, output reg [1:0] ResultSrc, output reg MemWrite,
  output reg Jump, output reg Branch, output reg [2:0] ALUControl,
  output reg ALUSrc, output reg [1:0] ImmSrc
);
  always @* begin
    // defaults
    RegWrite=0; ResultSrc=2'b00; MemWrite=0; Jump=0; Branch=0; ALUControl=3'b000;
ALUSrc=0; ImmSrc=2'b00;
    case (op)
      7'b0110011: begin // R: ADD/SUB/AND/OR
        RegWrite=1; ALUSrc=0; ResultSrc=2'b00;
```

```
      if (funct3==3'b000) ALUControl = (funct7==7'b0100000)?3'b001:3'b000; // SUB/ADD
      else if (funct3==3'b111) ALUControl=3'b010; // AND
      else if (funct3==3'b110) ALUControl=3'b011; // OR
    end
    7'b0010011: begin // ADDI
      RegWrite=1; ALUSrc=1; ImmSrc=2'b00; ALUControl=3'b000; ResultSrc=2'b00;
    end
    7'b0000011: begin // LW
      RegWrite=1; ALUSrc=1; ImmSrc=2'b00; ALUControl=3'b000; ResultSrc=2'b01;
    end
    7'b0100011: begin // SW
      MemWrite=1; ALUSrc=1; ImmSrc=2'b01; ALUControl=3'b000;
    end
    7'b1100011: begin // BEQ
      Branch=1; ALUSrc=0; ImmSrc=2'b10; ALUControl=3'b001;
    end
    default: ; // NOP
   endcase
  end
endmodule

module sign_extend(input [31:0] In, input [1:0] ImmSrc, output reg [31:0] Imm_Ext);
  always @* begin
    case (ImmSrc)
      2'b00: Imm_Ext = {{20{In[31]}}, In[31:20]};                    // I
      2'b01: Imm_Ext = {{20{In[31]}}, In[31:25], In[11:7]};          // S
      2'b10: Imm_Ext = {{19{In[31]}}, In[31], In[7], In[30:25], In[11:8], 1'b0}; // B
      default: Imm_Ext = 32'b0;
    endcase
  end
endmodule

module alu(input [31:0] a,input [31:0] b,input [2:0] alu_ctrl,output reg [31:0] y,output zero);
  always @* begin
    case(alu_ctrl)
      3'b000: y = a + b;
      3'b001: y = a - b;
      3'b010: y = a & b;
      3'b011: y = a | b;
      default:y = a + b;
    endcase
  end
  assign zero = (y==32'b0);
endmodule

// -------------------- [1] FETCH ----------------------
module fetch_cycle(input clk,input rst,input PCsrcE,input [31:0] PCtargetE,
                   output [31:0] InstrD, output [31:0] PCD, output [31:0] PCplus4D);
  wire [31:0] PCF, PCplus4F, InstrF, PC_next;
  reg  [31:0] PCF_reg, InstrF_reg, PCplus4F_reg;

  assign PC_next = PCsrcE ? PCtargetE : (PCF + 32'h4);
  PC_module u_pc(.clk(clk), .rst(rst), .PC_next(PC_next), .PC(PCF));
  instruction_memory u_imem(.rst(rst), .A(PCF), .RD(InstrF));

  always @(posedge clk or negedge rst) begin
    if (!rst) begin
      PCF_reg<=0; InstrF_reg<=0; PCplus4F_reg<=0;
    end else begin
      PCF_reg<=PCF; InstrF_reg<=InstrF; PCplus4F_reg<=PCF + 32'h4;
```

```
    end
  end
  assign InstrD   = InstrF_reg;
  assign PCD      = PCF_reg;
  assign PCplus4D = PCplus4F_reg;
endmodule

// -------------------- [2] DECODE --------------------
module decode_cycle(
  input clk,input rst,
  input [31:0] InstrD,input [31:0] PCD,input [31:0] PCplus4D,
  input RegWriteW,input [4:0] RDW,input [31:0] ResultW,
  output RegWriteE,output [1:0] ResultSrcE,output MemWriteE,
  output JumpE,output BranchE,output [2:0] ALUControlE,output ALUSrcE,
  output [31:0] RD1E,output [31:0] RD2E,output [31:0] ImmExtE,
  output [4:0]  RdE, output [31:0] PCE, output [31:0] PCplus4E
);
  wire RegWriteD, MemWriteD, JumpD, BranchD, ALUSrcD;
  wire [1:0] ResultSrcD, ImmSrcD;
  wire [2:0] ALUControlD;
  wire [31:0] RD1_D, RD2_D, Imm_Ext_D;

  control_unit cu(
    .op(InstrD[6:0]), .funct3(InstrD[14:12]), .funct7(InstrD[31:25]),
    .RegWrite(RegWriteD), .ResultSrc(ResultSrcD), .MemWrite(MemWriteD),
    .Jump(JumpD), .Branch(BranchD), .ALUControl(ALUControlD),
    .ALUSrc(ALUSrcD), .ImmSrc(ImmSrcD)
  );

  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_extend se(.In(InstrD), .ImmSrc(ImmSrcD), .Imm_Ext(Imm_Ext_D));

  reg RegWriteD_r, MemWriteD_r, JumpD_r, BranchD_r, ALUSrcD_r;
  reg [1:0] ResultSrcD_r;
  reg [2:0] ALUControlD_r;
  reg [31:0] RD1_r, RD2_r, Imm_r, PC_r, PC4_r;
  reg [4:0]  Rd_r;

  always @(posedge clk or negedge rst) begin
    if (!rst) begin
      RegWriteD_r<=0; ResultSrcD_r<=0; MemWriteD_r<=0; JumpD_r<=0; BranchD_r<=0;
ALUControlD_r<=0; ALUSrcD_r<=0;
      RD1_r<=0; RD2_r<=0; Imm_r<=0; PC_r<=0; PC4_r<=0; Rd_r<=0;
    end else begin
      RegWriteD_r<=RegWriteD; ResultSrcD_r<=ResultSrcD; MemWriteD_r<=MemWriteD;
      JumpD_r<=JumpD; BranchD_r<=BranchD; ALUControlD_r<=ALUControlD;
ALUSrcD_r<=ALUSrcD;
      RD1_r<=RD1_D; RD2_r<=RD2_D; Imm_r<=Imm_Ext_D; PC_r<=PCD; PC4_r<=PCplus4D;
Rd_r<=InstrD[11:7];
    end
  end

  assign RegWriteE  = RegWriteD_r;
  assign ResultSrcE = ResultSrcD_r;
  assign MemWriteE  = MemWriteD_r;
```

```
  assign JumpE      = JumpD_r;
  assign BranchE    = BranchD_r;
  assign ALUControlE= ALUControlD_r;
  assign ALUSrcE    = ALUSrcD_r;
  assign RD1E       = RD1_r;
  assign RD2E       = RD2_r;
  assign ImmExtE    = Imm_r;
  assign RdE        = Rd_r;
  assign PCE        = PC_r;
  assign PCplus4E   = PC4_r;
endmodule

// -------------------- [3] EXECUTE -------------------
module execute_cycle(
  input clk,input rst,
  input RegWriteE,input [1:0] ResultSrcE,input MemWriteE,input JumpE,input BranchE,
  input [2:0] ALUControlE,input ALUSrcE,
  input [31:0] RD1E,input [31:0] RD2E,input [31:0] ImmExtE,input [31:0] PCE,input [31:0] PCplus4E,
  input [4:0] RdE,
  output RegWriteM,output [1:0] ResultSrcM,output MemWriteM,
  output [31:0] ALUResultM,output [31:0] WriteDataM,
  output [4:0] RdM,output [31:0] PCplus4M,
  output PCsrcE,output [31:0] PCtargetE
);
  wire [31:0] srcB = ALUSrcE ? ImmExtE : RD2E;
  wire [31:0] alu_y; wire zero;
  alu u_alu(.a(RD1E), .b(srcB), .alu_ctrl(ALUControlE), .y(alu_y), .zero(zero));

  assign PCtargetE = PCE + ImmExtE;
  assign PCsrcE    = (BranchE & zero) | (1'b0); // JumpE kept 0 in this minimal core

  reg RegWrite_r; reg [1:0] ResultSrc_r; reg MemWrite_r;
  reg [31:0] ALUOut_r, WD_r; reg [4:0] Rd_r; reg [31:0] PC4_r;

  always @(posedge clk or negedge rst) begin
    if (!rst) begin
      RegWrite_r<=0; ResultSrc_r<=0; MemWrite_r<=0; ALUOut_r<=0; WD_r<=0; Rd_r<=0;
PC4_r<=0;
    end else begin
      RegWrite_r<=RegWriteE; ResultSrc_r<=ResultSrcE; MemWrite_r<=MemWriteE;
      ALUOut_r<=alu_y; WD_r<=RD2E; Rd_r<=RdE; PC4_r<=PCplus4E;
    end
  end

  assign RegWriteM  = RegWrite_r;
  assign ResultSrcM = ResultSrc_r;
  assign MemWriteM  = MemWrite_r;
  assign ALUResultM = ALUOut_r;
  assign WriteDataM = WD_r;
  assign RdM        = Rd_r;
  assign PCplus4M   = PC4_r;
endmodule

// -------------------- [4] MEMORY --------------------
module memory_cycle(
  input clk,input rst,
  input RegWriteM,input [1:0] ResultSrcM,input MemWriteM,
  input [31:0] ALUResultM,input [31:0] WriteDataM,input [4:0] RdM,input [31:0] PCplus4M,
  output RegWriteW,output [1:0] ResultSrcW,
  output [31:0] ReadDataW,output [31:0] ALUResultW,output [4:0] RdW,output [31:0] PCplus4W
```

```
);
  wire [31:0] read_data;
  data_memory
dmem(.clk(clk), .rst(rst), .mem_write(MemWriteM), .mem_read(ResultSrcM==2'b01),
                .addr(ALUResultM), .wd(WriteDataM), .rd(read_data));

  reg RegWrite_r; reg [1:0] ResultSrc_r; reg [31:0] RD_r, ALU_r, PC4_r; reg [4:0] Rd_r;
  always @(posedge clk or negedge rst) begin
    if (!rst) begin
      RegWrite_r<=0; ResultSrc_r<=0; RD_r<=0; ALU_r<=0; Rd_r<=0; PC4_r<=0;
    end else begin
      RegWrite_r<=RegWriteM; ResultSrc_r<=ResultSrcM; RD_r<=read_data; ALU_r<=ALUResultM;
Rd_r<=RdM; PC4_r<=PCplus4M;
    end
  end

  assign RegWriteW  = RegWrite_r;
  assign ResultSrcW = ResultSrc_r;
  assign ReadDataW  = RD_r;
  assign ALUResultW = ALU_r;
  assign RdW        = Rd_r;
  assign PCplus4W   = PC4_r;
endmodule

// -------------------- [5] WRITEBACK -----------------
module writeback_cycle(
  input RegWriteW,input [1:0] ResultSrcW,
  input [31:0] ReadDataW,input [31:0] ALUResultW,input [31:0] PCplus4W,input [4:0] RdW,
  output RegWrite_out, output [4:0] RDW, output [31:0] ResultW
);
  wire [31:0] m01 = (ResultSrcW==2'b01) ? ReadDataW  : ALUResultW;
  assign ResultW  = (ResultSrcW==2'b10) ? PCplus4W : m01;
  assign RDW      = RdW;
  assign RegWrite_out = RegWriteW;
endmodule

// -------------------- TOP: ties everything -----------
module riscv5_pipeline(
  input clk, input rst
);
  wire [31:0] InstrD, PCD, PCplus4D;

  wire RegWriteW_out; wire [4:0] RDW_out; wire [31:0] ResultW_out;

  wire RegWriteE; wire [1:0] ResultSrcE; wire MemWriteE; wire JumpE; wire BranchE; wire [2:0]
ALUControlE; wire ALUSrcE;
  wire [31:0] RD1E, RD2E, ImmExtE; wire [4:0] RdE; wire [31:0] PCE; wire [31:0] PCplus4E;

  wire RegWriteM; wire [1:0] ResultSrcM; wire MemWriteM; wire [31:0] ALUResultM; wire [31:0]
WriteDataM; wire [4:0] RdM; wire [31:0] PCplus4M;

  wire RegWriteW; wire [1:0] ResultSrcW; wire [31:0] ReadDataW, ALUResultW, PCplus4W; wire
[4:0] RdW;

  wire PCsrcE; wire [31:0] PCtargetE;

  fetch_cycle   IF (.clk(clk), .rst(rst), .PCsrcE(PCsrcE), .PCtargetE(PCtargetE),
                .InstrD(InstrD), .PCD(PCD), .PCplus4D(PCplus4D));

  decode_cycle  ID (.clk(clk), .rst(rst), .InstrD(InstrD), .PCD(PCD), .PCplus4D(PCplus4D),
```

```
                .RegWriteW(RegWriteW_out), .RDW(RDW_out), .ResultW(ResultW_out),
                .RegWriteE(RegWriteE), .ResultSrcE(ResultSrcE), .MemWriteE(MemWriteE),
                .JumpE(JumpE), .BranchE(BranchE), .ALUControlE(ALUControlE), .ALUSrcE(ALUSrcE
),
                .RD1E(RD1E), .RD2E(RD2E), .ImmExtE(ImmExtE), .RdE(RdE), .PCE(PCE), .PCplus4E(
PCplus4E));

  execute_cycle EX (.clk(clk), .rst(rst),
                .RegWriteE(RegWriteE), .ResultSrcE(ResultSrcE), .MemWriteE(MemWriteE), .JumpE(J
umpE), .BranchE(BranchE),
                .ALUControlE(ALUControlE), .ALUSrcE(ALUSrcE),
                .RD1E(RD1E), .RD2E(RD2E), .ImmExtE(ImmExtE), .PCE(PCE), .PCplus4E(PCplus4E), .
RdE(RdE),
                .RegWriteM(RegWriteM), .ResultSrcM(ResultSrcM), .MemWriteM(MemWriteM),
                .ALUResultM(ALUResultM), .WriteDataM(WriteDataM), .RdM(RdM), .PCplus4M(PCplu
s4M),
                .PCsrcE(PCsrcE), .PCtargetE(PCtargetE));

  memory_cycle  MEM(.clk(clk), .rst(rst),
                .RegWriteM(RegWriteM), .ResultSrcM(ResultSrcM), .MemWriteM(MemWriteM),
                .ALUResultM(ALUResultM), .WriteDataM(WriteDataM), .RdM(RdM), .PCplus4M(PCplu
s4M),
                .RegWriteW(RegWriteW), .ResultSrcW(ResultSrcW),
                .ReadDataW(ReadDataW), .ALUResultW(ALUResultW), .RdW(RdW), .PCplus4W(PCpl
us4W));

  writeback_cycle WB(.RegWriteW(RegWriteW), .ResultSrcW(ResultSrcW),
                 .ReadDataW(ReadDataW), .ALUResultW(ALUResultW), .PCplus4W(PCplus4W), .Rd
W(RdW),
                 .RegWrite_out(RegWriteW_out), .RDW(RDW_out), .ResultW(ResultW_out));
endmodule
```

testbench

```
                                                `timescale 1ns/1ps

module tb_riscv5_pipeline;
  reg clk;
  reg rst;

  // Instantiate top
  riscv5_pipeline dut(.clk(clk), .rst(rst));

  // Clock: 10ns period
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Reset sequence
  initial begin
    rst = 0;
    #12 rst = 1;   // release reset after some cycles
  end

  // Run for N cycles then stop
  initial begin
    #500 $finish;   // run simulation for 500ns
  end

  // Monitor important signals in console
  initial begin
    $display("Time | PC | Instr | RegWrite | Rd | ResultW");
    $monitor("%4t | %h | %h | %b | %0d | %h",
             $time,
             dut.IF.PCD,           // PC in Decode stage
             dut.IF.InstrD,        // Instruction in Decode stage
             dut.WB.RegWrite_out,  // Writeback enable
             dut.WB.RDW,           // Destination reg
             dut.WB.ResultW        // Value written back
    );
  end

  // Dump signals for waveform (GTKWave)
  initial begin
    $dumpfile("riscv5_pipeline.vcd");  // name of dump file
    $dumpvars(0, tb_riscv5_pipeline);  // dump everything in testbench + DUT
  end

endmodule
```