**EE-475L: Computer Architecture**

**CEP Report**

**Submitted by**

2020ee162

**Submitted to**

Sir Ali Imran

**Department of Electrical Engineering**

**University of Engineering and Technology, Lahore, Pakistan.**

Contents

[RISC\_V 3](#_Toc135509019)

[Datapath 4](#_Toc135509020)

[Controller 7](#_Toc135509021)

[Fetch\_Stage 13](#_Toc135509022)

[Decode/Execute\_Stage 15](#_Toc135509023)

[Writeback\_Stage 17](#_Toc135509024)

[Hazard Detection Unit 19](#_Toc135509025)

[Instruction\_memory 20](#_Toc135509026)

[Register\_File 20](#_Toc135509027)

[ALU 21](#_Toc135509028)

[Branch Detection Unit 22](#_Toc135509029)

[Data\_Memory 23](#_Toc135509030)

[CSR\_Register 25](#_Toc135509031)

[Pipeline Register 27](#_Toc135509032)

[Testbench for Factorial 28](#_Toc135509033)

[Assembly Code 29](#_Toc135509034)

[Results 30](#_Toc135509035)

[Testbench for CSR 30](#_Toc135509036)

[CSR\_Assembly 32](#_Toc135509037)

[Results 33](#_Toc135509038)

# RISC\_V

|  |
| --- |
| `include "datapath.sv"  `include "Controller.sv"  `include "timer.sv"  module RISC\_V (      input logic clk,      input logic rst,      timer\_en,      ext\_inter,      output logic [31:0] result  );    logic [31:0] instruction;    logic [ 3:0] interrupt;    logic        flush,        stall,        ovf,        mem\_read,        reg\_wr,        A\_sel,        PC\_sel,        B\_sel,        br\_taken,        mem\_wr,        csr\_reg\_r,        csr\_reg\_wr,        is\_mret;    logic [1:0] wb\_sel;    logic [3:0] ALUctrl;    Controller Controller (        .clk(clk),        .rst(rst),        .stall(stall),        .instruction(instruction),        .br\_taken(br\_taken),        .flush(flush),        .ALUctrl(ALUctrl),        .mem\_wr\_ppl(mem\_wr),        .mem\_read\_ppl(mem\_read),        .A\_sel(A\_sel),        .B\_sel(B\_sel),        .wb\_sel\_ppl(wb\_sel),        .reg\_wr\_ppl(reg\_wr),        .PC\_sel\_ppl(PC\_sel),        .csr\_reg\_r\_ppl(csr\_reg\_r),        .csr\_reg\_wr\_ppl(csr\_reg\_wr),        .is\_mret\_ppl(is\_mret)    );    datapath datapath (        .clk(clk),        .rst(rst),        .reg\_wr(reg\_wr),        .A\_sel(A\_sel),        .B\_sel(B\_sel),        .mem\_wr(mem\_wr),        .mem\_read(mem\_read),        .PC\_sel(PC\_sel),        .csr\_reg\_r(csr\_reg\_r),        .csr\_reg\_wr(csr\_reg\_wr),        .is\_mret(is\_mret),        .interrupt(interrupt),        .wb\_sel(wb\_sel),        .ALUctrl(ALUctrl),        .instruction(instruction),        .br\_taken(br\_taken),        .main\_flush(flush),        .stall(stall),        .result(result)    );    always\_comb begin      case ({        ext\_inter, ovf      })        2'b00: interrupt = 4'b0000;        2'b01: interrupt = 4'b0001;        2'b10: interrupt = 4'b0010;        2'b11: interrupt = 4'b0001;      endcase    end    timer #(        .WIDTH(4)    ) timer\_instance (        .clk(clk),        .rst(rst),        .en (timer\_en),        .ovf(ovf)    );  endmodule |

**Listing 1. RISC\_V.sv**

## Datapath

|  |
| --- |
| `include "wb\_stage.sv"  `include "Fetch.sv"  `include "Decode.sv"  `include "Hazard\_detection.sv"  module datapath (      input logic clk,      rst,      reg\_wr,      A\_sel,      B\_sel,      mem\_wr,      mem\_read,      PC\_sel,      csr\_reg\_r,      csr\_reg\_wr,      is\_mret,      logic [3:0] interrupt,      input logic [1:0] wb\_sel,      input logic [3:0] ALUctrl,      output logic [31:0] instruction,      output logic br\_taken,      main\_flush,      stall,      output logic [31:0] result  );    logic [31:0] PC\_decode, PC\_wb, ALU\_wb, wdata, data\_to\_mem, instruction\_wb, rdata1\_wb;    logic [4:0] rs2, rs1, rd\_wb;    logic forw\_a, forw\_b, flush;    logic [31:0] epc;    logic epc\_taken, loaded;    assign rs1 = instruction[19:15];    assign rs2 = instruction[24:20];    assign rd\_wb = instruction\_wb[11:7];    assign main\_flush = flush | rst;    Fetch Fetch (        .clk(clk),        .rst(rst),        .PC\_sel(PC\_sel),        .flush(main\_flush),        .stall(stall),        .epc\_taken(epc\_taken),        .epc(epc),        .instruction\_ppl(instruction),        .PC\_ppl(PC\_decode),        .ALU\_o(ALU\_wb)    );    Decode Decode\_instance (        .clk(clk),        .rst(rst),        .flush(main\_flush),        .stall(stall),        .reg\_wr(reg\_wr),        .A\_sel(A\_sel),        .B\_sel(B\_sel),        .forw\_a(forw\_a),        .forw\_b(forw\_b),        .instruction(instruction),        .PC(PC\_decode),        .wdata(wdata),        .ALUctrl(ALUctrl),        .br\_taken(br\_taken),        .PC\_ppl(PC\_wb),        .ALU\_ppl(ALU\_wb),        .rdata2\_ppl(data\_to\_mem),        .rdata1\_ppl(rdata1\_wb),        .instruction\_ppl(instruction\_wb)    );    wb\_stage wb\_stage\_instance (        .clk(clk),        .rst(rst),        .mem\_wr(mem\_wr),        .mem\_read(mem\_read),        .csr\_reg\_wr(csr\_reg\_wr),        .csr\_reg\_r(csr\_reg\_r),        .is\_mret(is\_mret),        .ALU\_o(ALU\_wb),        .wb\_sel(wb\_sel),        .PC(PC\_wb),        .rdata1(rdata1\_wb),        .interrupt(interrupt),        .data\_to\_mem(data\_to\_mem),        .instruction(instruction\_wb),        .wdata(wdata),        .epc(epc),        .epc\_taken(epc\_taken),        .loaded(loaded),        .result(result)    );    Hazard\_detection Hazard\_detection\_instance (        .reg\_wr(reg\_wr),        .mem\_read(mem\_read),        .loaded(loaded),        .PC\_sel(PC\_sel),        .epc\_taken(epc\_taken),        .raddr1(rs1),        .raddr2(rs2),        .rd\_wb(rd\_wb),        .wb\_sel(wb\_sel),        .forw\_a(forw\_a),        .forw\_b(forw\_b),        .flush(flush),        .stall(stall)    );  endmodule |

**Listing 2. Datapath.sv**

## Controller

|  |
| --- |
| `include "LS\_controller.sv"  `include "Pipeline\_reg.sv"  module Controller (      input clk,      input rst,      stall,      input logic [31:0] instruction,      input logic br\_taken,      flush,      output logic [3:0] ALUctrl,      output logic mem\_wr\_ppl,      mem\_read\_ppl,      output logic A\_sel,      B\_sel,      reg\_wr\_ppl,      PC\_sel\_ppl,      is\_mret\_ppl,      csr\_reg\_r\_ppl,      csr\_reg\_wr\_ppl,      output logic [1:0] wb\_sel\_ppl  );    localparam R\_type = 5'b01100;    localparam I\_type = 5'b00100;    localparam Load\_type = 5'b00000;    localparam S\_type = 5'b01000;    localparam B\_type = 5'b11000;    localparam J\_type = 5'b11011;    localparam Jalr\_type = 5'b11001;    localparam lui\_type = 5'b01101;    localparam auipc\_type = 5'b00101;    localparam csr\_type = 5'b11100;    logic [6:0] opcode;    logic func7, func7\_mret, is\_mret, csr\_reg\_r, csr\_reg\_wr;    logic [2:0] func3;    assign func3 = instruction[14:12];    assign opcode = instruction[6:0];    assign func7 = instruction[30];    assign func7\_mret = instruction[29];    logic [1:0] wb\_sel;    logic PC\_sel, reg\_wr, mem\_wr, mem\_read;    always\_comb begin      case (opcode[6:2])        R\_type: begin          casex ({            func7, func3          })            4'b0000: ALUctrl = 4'd0;  //ADD            4'b1000: ALUctrl = 4'd1;  //Sub            4'bX001: ALUctrl = 4'd2;  //SLL            4'bX010: ALUctrl = 4'd3;  //SLT            4'bX100: ALUctrl = 4'd4;  //XOR            4'bX011: ALUctrl = 4'd5;  //SLTU            4'b0101: ALUctrl = 4'd6;  //SRL            4'b1101: ALUctrl = 4'd7;  //SRA            4'bX110: ALUctrl = 4'd8;  //OR            4'bX111: ALUctrl = 4'd9;  //AND            default: ALUctrl = 4'bXXXX;          endcase          A\_sel = 1;          PC\_sel = 0;          mem\_wr = 0;          mem\_read = '0;          B\_sel = 0;          wb\_sel = 2'b01;          reg\_wr = 1;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        I\_type: begin          casex ({            func7, func3          })            4'bX000: ALUctrl = 4'd0;  //ADD            4'bX001: ALUctrl = 4'd2;  //SLL            4'bX010: ALUctrl = 4'd3;  //SLT            4'bX100: ALUctrl = 4'd4;  //XOR            4'bX011: ALUctrl = 4'd5;  //SLTU            4'b0101: ALUctrl = 4'd6;  //SRL            4'b1101: ALUctrl = 4'd7;  //SRA            4'bX110: ALUctrl = 4'd8;  //OR            4'bX111: ALUctrl = 4'd9;  //AND            default: begin              ALUctrl = 4'bXXXX;            end          endcase          mem\_wr = 0;          mem\_read = '0;          A\_sel = 1;          PC\_sel = 0;          B\_sel = 1;          wb\_sel = 2'b01;          reg\_wr = 1;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        Load\_type: begin          mem\_wr = 0;          mem\_read = 1'b1;          A\_sel = 1;          PC\_sel = 0;          B\_sel = 1;          wb\_sel = 2'b10;          reg\_wr = 1;          ALUctrl = 4'd0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        S\_type: begin          mem\_wr = 1;          mem\_read = '0;          A\_sel = 1;          PC\_sel = 0;          B\_sel = 1;          wb\_sel = 'x;          reg\_wr = 0;          ALUctrl = 4'd0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        B\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 0;          B\_sel = 1;          wb\_sel = 'x;          reg\_wr = 0;          ALUctrl = 4'd0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;          case (br\_taken)            0: PC\_sel = 0;            1: PC\_sel = 1;            default: PC\_sel = 'x;          endcase        end        J\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 0;          B\_sel = 1;          wb\_sel = 2'b00;          reg\_wr = 1;          ALUctrl = 4'd0;          PC\_sel = 1'b1;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        Jalr\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 1;          B\_sel = 1;          wb\_sel = 2'b00;          reg\_wr = 1'b1;          ALUctrl = 4'd0;          PC\_sel = 1'b1;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        lui\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 1'bx;          B\_sel = 1;          wb\_sel = 2'b01;          reg\_wr = 1'b1;          ALUctrl = 4'd10;          PC\_sel = 1'b0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        auipc\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 0;          B\_sel = 1;          wb\_sel = 2'b01;          reg\_wr = 1;          ALUctrl = 4'd0;          PC\_sel = 1'b0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end        csr\_type: begin          mem\_wr = 0;          mem\_read = '0;          A\_sel = 1;          B\_sel = 1'bx;          wb\_sel = 2'b11;          reg\_wr = 1'b1;          ALUctrl = 4'd0;          PC\_sel = 1'b0;          casex ({            func7\_mret, func3          })            4'b1000: begin              csr\_reg\_r = 1'b0;              csr\_reg\_wr = 1'b0;              is\_mret = 1'b1;            end            4'bx001: begin              csr\_reg\_r = 1'b1;              csr\_reg\_wr = 1'b1;              is\_mret = 1'b0;            end            default: begin              csr\_reg\_r = 1'b0;              csr\_reg\_wr = 1'b0;              is\_mret = 1'b0;            end          endcase        end        default: begin          mem\_wr = '0;          mem\_read = '0;          B\_sel = 'x;          A\_sel = 'x;          wb\_sel = 'x;          reg\_wr = '0;          ALUctrl = '0;          PC\_sel = '0;          csr\_reg\_r = 1'b0;          csr\_reg\_wr = 1'b0;          is\_mret = 1'b0;        end      endcase    end    //    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipeline1 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(reg\_wr),        .out(reg\_wr\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) pipeline2 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(mem\_wr),        .out(mem\_wr\_ppl)    );    Pipeline\_reg #(        .WIDTH(2),        .reset(0)    ) pipeline3 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(wb\_sel),        .out(wb\_sel\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipieline4 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(PC\_sel),        .out(PC\_sel\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipeline\_reg\_instance (        .clk(clk),        .flush(flush),        .stall(stall),        .in(mem\_read),        .out(mem\_read\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipeline\_ismret (        .clk(clk),        .flush(flush),        .stall(stall),        .in(is\_mret),        .out(is\_mret\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipeline\_csr\_reg\_wr (        .clk(clk),        .flush(flush),        .stall(stall),        .in(csr\_reg\_r),        .out(csr\_reg\_r\_ppl)    );    Pipeline\_reg #(        .WIDTH(1),        .reset(0)    ) Pipeline\_csr\_reg\_r (        .clk(clk),        .flush(flush),        .stall(stall),        .in(csr\_reg\_wr),        .out(csr\_reg\_wr\_ppl)    );  endmodule |

**Listing 3 Controller.sv**

## Fetch\_Stage

|  |
| --- |
| `include "Pipeline\_reg.sv"  `include "Instrmem.sv"  module Fetch (      input clk,      input rst,      PC\_sel,      flush,      stall,      epc\_taken,      input logic [31:0] epc,      input logic [31:0] ALU\_o,      output logic [31:0] instruction\_ppl,      PC\_ppl  );    logic not\_stalled;    logic [31:0] PC, PC\_mux\_o, PC\_ppl\_in;    logic [31:0] instruction;    assign not\_stalled = !stall;    Instrmem Instrmem\_instance (        .addr\_i(PC),        .instruction\_o(instruction)    );    always\_ff @(posedge clk) begin      if (rst) PC <= 32'd0;      else if (not\_stalled) PC <= PC\_mux\_o;    end    assign PC\_mux\_o  = epc\_taken ? epc : (PC\_sel ? ALU\_o : PC + 4);    assign PC\_ppl\_in = flush ? ALU\_o : PC;    Pipeline\_reg Pipieline\_reg\_instance (        .clk(clk),        .flush(rst),        .stall(stall),        .in(PC\_ppl\_in),        .out(PC\_ppl)    );    Pipeline\_reg Pipieline\_reg\_instance2 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(instruction),        .out(instruction\_ppl)    );  endmodule |

**Listing 4.Fetch.sv**

## Decode/Execute\_Stage

|  |
| --- |
| `include "Pipeline\_reg.sv"  `include "Regfile.sv"  `include "Branch\_block.sv"  `include "ALU.sv"  module Decode (      input clk,      input rst,      flush,      stall,      reg\_wr,      A\_sel,      B\_sel,      forw\_a,      forw\_b,      input logic [31:0] instruction,      PC,      wdata,      input logic [3:0] ALUctrl,      output logic br\_taken,      output logic [31:0] PC\_ppl,      ALU\_ppl,      rdata2\_ppl,      rdata1\_ppl,      instruction\_ppl  );    localparam I\_type = 5'b00100;    localparam Load\_type = 5'b00000;    localparam B\_type = 5'b11000;    localparam S\_type = 5'b01000;    localparam J\_type = 5'b11011;    localparam Jalr\_type = 5'b11001;    localparam lui\_type = 5'b01101;    localparam auipc\_type = 5'b00101;    logic [31:0] rdata1, PC\_ppl\_in, rdata2, rdata1\_, rdata2\_, imm, ALU\_op\_b, ALU\_op\_a, ALU\_o;    logic [4:0] raddr1, raddr2, waddr\_ppl;    logic [2:0] func3;    assign func3 = instruction[14:12];    assign raddr1 = instruction[19:15];    assign raddr2 = instruction[24:20];    assign rdata1\_ = forw\_a ? ALU\_ppl : rdata1;    assign rdata2\_ = forw\_b ? ALU\_ppl : rdata2;    assign ALU\_op\_a = A\_sel ? rdata1\_ : PC;    assign ALU\_op\_b = B\_sel ? imm : rdata2\_;    assign waddr\_ppl = instruction\_ppl[11:7];    assign PC\_ppl\_in = flush ? ALU\_ppl : PC;    Pipeline\_reg Pipeline\_reg\_instance (        .clk(clk),        .flush(rst),        .stall(stall),        .in(PC\_ppl\_in),        .out(PC\_ppl)    );    Pipeline\_reg #(        .reset(0)    ) Pipeline2 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(ALU\_o),        .out(ALU\_ppl)    );    Pipeline\_reg Pipieline3 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(rdata2\_),        .out(rdata2\_ppl)    );    Pipeline\_reg Pipieline5 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(rdata1\_),        .out(rdata1\_ppl)    );    Pipeline\_reg Pipeline4 (        .clk(clk),        .flush(flush),        .stall(stall),        .in(instruction),        .out(instruction\_ppl)    );    Regfile Regfile\_instance (        .rst(1'b0),        .clk(clk),        .write\_en(reg\_wr),        .rs1\_in(raddr1),        .rs2\_in(raddr2),        .rd(waddr\_ppl),        .write\_data(wdata),        .rs1\_out(rdata1),        .rs2\_out(rdata2)    );    ALU ALU\_instance (        .a\_in(ALU\_op\_a),        .b\_in(ALU\_op\_b),        .ALUctrl(ALUctrl),        .result\_o(ALU\_o)    );    Branch\_block Branch\_block\_instance (        .op\_a(rdata1\_),        .op\_b(rdata2\_),        .func3(func3),        .branch\_taken(br\_taken)    );    //Immidiate generation    always\_comb begin      casex (instruction[6:2])        Load\_type, I\_type: imm = {{20{instruction[31]}}, instruction[31:20]};  //load,I        Jalr\_type: imm = {{20{instruction[31]}}, instruction[31:20]};        S\_type: imm = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]};  //save        J\_type:        imm = {{12{instruction[31]}}, instruction[19:12], instruction[20], instruction[30:21], 1'b0};        B\_type:        imm = {{20{instruction[31]}}, instruction[7], instruction[30:25], instruction[11:8], 1'b0};        lui\_type, auipc\_type: imm = {{instruction[31:12]}, {12{1'b0}}};        default: begin          imm = 'x;        end      endcase    end  endmodule |

**Listing 5. Decode.sv**

## Writeback\_Stage

|  |
| --- |
| `include "data\_mem.sv"  `include "CSR\_reg.sv"  module wb\_stage (      input logic clk,      rst,      mem\_wr,      mem\_read,      csr\_reg\_wr,      csr\_reg\_r,      is\_mret,      input logic [31:0] ALU\_o,      input logic [1:0] wb\_sel,      input logic [31:0] PC,      data\_to\_mem,      instruction,      rdata1,      logic [3:0] interrupt,      output logic [31:0] wdata,      epc,      output logic epc\_taken,      loaded,      output logic [31:0] result  );    logic [31:0] mem\_data, csr\_read\_data;    logic [ 2:0] func3;    logic [ 1:0] mem\_col;    logic [11:0] csr\_addr;    localparam I\_type = 5'b00100;    localparam Load\_type = 5'b00000;    localparam B\_type = 5'b11000;    localparam S\_type = 5'b01000;    localparam J\_type = 5'b11011;    localparam Jalr\_type = 5'b11001;    localparam lui\_type = 5'b01101;    localparam auipc\_type = 5'b00101;    assign func3 = instruction[14:12];    assign mem\_col = ALU\_o[1:0];    assign csr\_addr = instruction[31:20];    always\_comb begin      case (wb\_sel)        2'b00:   wdata = PC + 4;        2'b01:   wdata = ALU\_o;        2'b10:   wdata = mem\_data;        2'b11:   wdata = csr\_read\_data;        default: wdata = 'x;      endcase    end    always\_ff @(posedge clk) begin      if (wb\_sel == 2'b01) loaded <= 1'b0;      else loaded <= loaded + 1'b1;    end    data\_mem data\_mem\_instance (        .clk(clk),        .rst(rst),        .mem\_wr(mem\_wr),        .mem\_read(mem\_read),        .addr(ALU\_o),        .data\_wr(data\_to\_mem),        .func3(func3),        .mem\_col(mem\_col),        .mem\_data(mem\_data),        .result(result)    );    CSR\_reg CSR\_reg\_instance (        .clk(clk),        .rst(rst),        .reg\_wr(csr\_reg\_wr),        .reg\_r(csr\_reg\_r),        .PC(PC),        .addr(csr\_addr),        .interrupt(interrupt),        .wdata\_csr(rdata1),        .is\_mret(is\_mret),        .rdata(csr\_read\_data),        .epc(epc),        .epc\_taken(epc\_taken)    );  endmodule |

**Listing 6. wb\_stage.sv**

## Hazard Detection Unit

|  |
| --- |
| module Hazard\_detection (      input logic reg\_wr,      mem\_read,      loaded,      PC\_sel,      epc\_taken,      input logic [4:0] raddr1,      raddr2,      rd\_wb,      input logic [1:0] wb\_sel,      output logic forw\_a,      forw\_b,      flush,      stall  );    logic valid, a1, b1;    assign valid = |rd\_wb;    assign a1 = (raddr1 == rd\_wb);    assign b1 = (raddr2 == rd\_wb);    assign forw\_a = ((raddr1 == rd\_wb) & reg\_wr) & valid & (wb\_sel == 2'b01);    assign forw\_b = ((raddr2 == rd\_wb) & reg\_wr) & valid & (wb\_sel == 2'b01);    assign stall = ((raddr1 == rd\_wb) | (raddr2 == rd\_wb)) & valid & (wb\_sel != 2'b01) & (~loaded);    assign flush = PC\_sel | epc\_taken;  endmodule |

**Listing 7. Hazard\_detection.sv**

## Instruction\_memory

|  |
| --- |
| module Instrmem (      input  logic [31:0] addr\_i,      output logic [31:0] instruction\_o  );    initial begin      $readmemh("instructions.mem", instrmem);    end    logic [31:0] instrmem[0:31];    assign instruction\_o = instrmem[addr\_i[6:2]];  endmodule |

**Listing 8. Instrmem.sv**

## Register\_File

|  |
| --- |
| module Regfile (      input logic rst,      clk,      write\_en,      input logic [4:0] rs1\_in,      rs2\_in,      rd,      input logic [31:0] write\_data,      output logic [31:0] rs1\_out,      rs2\_out  );    logic [31:0] mem[0:31];    logic [31:0] result;    assign result = mem[12];    logic valid\_add1, valid\_add2, valid\_write\_en;    //validations    assign valid\_add1 = |rs1\_in;    assign valid\_add2 = |rs2\_in;    assign valid\_write\_en = |rd & write\_en;    assign rs1\_out = valid\_add1 ? mem[rs1\_in] : '0;    assign rs2\_out = valid\_add2 ? mem[rs2\_in] : '0;    initial begin      $readmemh("registervalues.mem", mem);    end    always\_ff @(negedge clk)      if (rst) begin        mem = '{default: '0};      end else if (write\_en) mem[rd] <= write\_data;  endmodule |

**Listing 9. Regfile.sv**

## ALU

|  |
| --- |
| `include "mux16x1.sv"  module ALU (      input  logic [31:0] a\_in,      b\_in,      input  logic [ 3:0] ALUctrl,      output logic [31:0] result\_o  );    logic [31:0]        and\_res, or\_res, xor\_res, add\_sub\_res, SLT\_res, SLTU\_res, t, SLL\_res, SRL\_res, SRA\_res;    logic [31:0] mux1\_o;    logic C\_out, N, V, W, C;  //Flags    assign N = add\_sub\_res[31];    assign C = C\_out;    assign V = (add\_sub\_res[31] ^ a\_in[31]) & (~(a\_in[31] ^ b\_in[31] ^ ALUctrl[0]));    assign W = add\_sub\_res[31] ^ V;    assign t = ~b\_in;    assign mux1\_o = ALUctrl[0] ? t : b\_in;    // Operations    assign {C\_out, add\_sub\_res} = a\_in + mux1\_o + ALUctrl[0];    assign and\_res = a\_in & b\_in;    assign or\_res = a\_in | b\_in;    assign xor\_res = a\_in ^ b\_in;    assign SLT\_res = {30'd0, W};    assign SLTU\_res = {30'd0, ~C};    assign SRA\_res = a\_in >>> b\_in;    assign SRL\_res = a\_in >> b\_in;    assign SLL\_res = a\_in << b\_in;    mux11x1 ALU\_mux (        .i1 (add\_sub\_res),        .i2 (add\_sub\_res),        .i3 (SLL\_res),        .i4 (SLT\_res),        .i5 (xor\_res),        .i6 (SLTU\_res),        .i7 (SRL\_res),        .i8 (SRA\_res),        .i9 (or\_res),        .i10(and\_res),        .i11(b\_in),        .s  (ALUctrl),        .y  (result\_o)    );  endmodule |

**Listing 10. ALU.sv**

## Branch Detection Unit

|  |
| --- |
| module Branch\_block (      input logic [31:0] op\_a,      op\_b,      input logic [2:0] func3,      output logic branch\_taken  );    logic [31:0] branch\_sub;    logic overflow, not\_zero, neg, carry;    assign branch\_sub = op\_a - op\_b;    assign not\_zero = |branch\_sub;    assign neg = branch\_sub[31];    assign {carry, overflow} = (op\_a ^ op\_b) && (op\_a ^ branch\_sub[31]);    always\_comb begin      case (func3)        //beq        3'b000:  branch\_taken = ~not\_zero;        //bneq        3'b001:  branch\_taken = not\_zero;        //blt        3'b100:  branch\_taken = branch\_sub[31];        //bge        3'b101:  branch\_taken = ~branch\_sub[31];        //bltu        3'b110:  branch\_taken = ~carry;        //bgeu        3'b111:  branch\_taken = carry;        default: branch\_taken = 1'bx;      endcase    end  endmodule |

**Listing 11. Branch\_block.sv**

## Data\_Memory

|  |
| --- |
| module data\_mem (      input clk,      input rst,      mem\_wr,      mem\_read,      input logic [31:0] addr,      data\_wr,      input logic [2:0] func3,      input logic [1:0] mem\_col,      output logic [31:0] mem\_data,      output logic [31:0] result  );    logic [ 3:0] mask;    logic [ 2:0] load\_ctrl;    logic [31:0] data\_wr\_gen;    LS\_controller LS\_controller\_instance (        .func3(func3),        .address(mem\_col),        .rdata2(data\_wr),        .wdata\_mem(data\_wr\_gen),        .load\_ctrl(load\_ctrl),        .mask(mask)    );    logic [31:0] mem\_data\_read;    logic [31:0] mem[0:31];    assign mem\_data\_read = mem\_read ? mem[addr[6:2]] : '0;    assign result = mem[0];    //Writting data to Memory    initial begin      $readmemh("data\_mem.mem", mem);    end    always\_ff @(posedge clk) begin      if (rst) mem <= '{default: '0};      else if (mem\_wr) begin        if (mask[0]) begin          mem[addr[31:2]][7:0] <= data\_wr\_gen[7:0];        end        if (mask[1]) begin          mem[addr[31:2]][15:8] <= data\_wr\_gen[15:8];        end        if (mask[2]) begin          mem[addr[31:2]][23:16] <= data\_wr\_gen[23:16];        end        if (mask[3]) begin          mem[addr[31:2]][31:24] <= data\_wr\_gen[31:24];        end      end    end    //Reading data\_memory    always\_comb begin      case (load\_ctrl)        3'b000: begin          case (addr[1:0])            2'b00:   mem\_data = {{24{mem\_data\_read[7]}}, mem\_data\_read[7:0]};            2'b01:   mem\_data = {{24{mem\_data\_read[15]}}, mem\_data\_read[15:8]};            2'b10:   mem\_data = {{24{mem\_data\_read[23]}}, mem\_data\_read[23:16]};            2'b11:   mem\_data = {{24{mem\_data\_read[31]}}, mem\_data\_read[31:24]};            default: mem\_data = 'x;          endcase        end        3'b001: begin          case (addr[1])            0: mem\_data = {{16{mem\_data\_read[15]}}, mem\_data\_read[15:0]};            1: mem\_data = {{16{mem\_data\_read[31]}}, mem\_data\_read[31:16]};            default: mem\_data = 'x;          endcase        end        3'b010: begin          mem\_data = mem\_data\_read;        end        3'b011: begin          case (addr[1:0])            2'b00:   mem\_data = {{24{1'b0}}, mem\_data\_read[7:0]};            2'b01:   mem\_data = {{24{1'b0}}, mem\_data\_read[15:8]};            2'b10:   mem\_data = {{24{1'b0}}, mem\_data\_read[23:16]};            2'b11:   mem\_data = {{24{1'b0}}, mem\_data\_read[31:24]};            default: mem\_data = 'x;          endcase        end        3'b100: begin          case (addr[1])            1'b0: mem\_data = {{16{1'b0}}, mem\_data\_read[15:0]};            1'b1: mem\_data = {{16{1'b0}}, mem\_data\_read[31:16]};            default: mem\_data = 'x;          endcase        end        default: mem\_data = 'x;      endcase    end  endmodule |

**Listing 12. Data\_mem.sv**

## CSR\_Register

|  |
| --- |
| module CSR\_reg (      input clk,      input rst,      reg\_wr,      reg\_r,      input logic [31:0] PC,      input logic [11:0] addr,      input logic [3:0] interrupt,      input logic [31:0] wdata\_csr,      input logic is\_mret,      output logic [31:0] rdata,      epc,      output logic epc\_taken  );    logic mie\_wr\_flag, interupt\_taken, mstatus\_wr\_flag, mtvec\_wr\_flag, timer\_inter, external\_inter;    logic [31:0] mepc\_q, mie\_q, mstatus\_q, mtvec\_q, mcause\_q, mip\_q, mcause\_d, mip\_d, ISR\_addr;    assign timer\_inter\_occur = mie\_q[7] & mip\_q[7];    localparam mepc\_addr = 12'h341;    localparam mcause\_addr = 12'h342;    localparam mip\_addr = 12'h344;    localparam mtvec\_addr = 12'h305;    localparam mie\_addr = 12'h304;    localparam mstatus\_addr = 12'h300;    assign timer\_inter = mip\_q[7] & mie\_q[7];    assign external\_inter = mip\_q[11] & mie\_q[11];    assign interupt\_taken = (timer\_inter | external\_inter) & mstatus\_q[3];    assign epc\_taken = is\_mret | interupt\_taken;    assign ISR\_addr = mtvec\_q[0] ? {mtvec\_q[31:2], 2'b00} : {mtvec\_q[31:2], 2'b00} + {mcause\_q[29:0], 2'b00};    assign epc = is\_mret ? mepc\_q : ISR\_addr;    //csr\_registers    always\_ff @(posedge clk) begin      if (rst) mepc\_q <= '0;      else if (interupt\_taken) mepc\_q <= PC;    end    always\_ff @(posedge clk) begin      if (rst) mie\_q <= '0;      else if (mie\_wr\_flag) mie\_q <= wdata\_csr;    end    always\_ff @(posedge clk) begin      if (rst) mstatus\_q <= '0;      else if (mstatus\_wr\_flag) mstatus\_q <= wdata\_csr;    end    always\_ff @(posedge clk) begin      if (rst) mtvec\_q <= '0;      else if (mtvec\_wr\_flag) mtvec\_q <= wdata\_csr;    end    //    always\_ff @(posedge clk) begin      if (rst) mcause\_q <= '0;      else mcause\_q <= mcause\_d;    end    always\_ff @(posedge clk) begin      if (rst) mip\_q <= '0;      else if (interupt\_taken) mip\_q <= '0;      else mip\_q <= mip\_d;    end    //    always\_comb begin      case (interrupt)        4'd1: begin          mip\_d = 32'h80;          mcause\_d = 32'd7;        end        4'd2: begin          mip\_d = 32'h800;          mcause\_d = 32'd11;        end        default: begin          mcause\_d = '0;          mip\_d = '0;        end      endcase    end    //    always\_comb begin      rdata = '0;      if (reg\_r) begin        case (addr)          mip\_addr: begin            rdata = mip\_q;          end          mie\_addr: begin            rdata = mie\_q;          end          mstatus\_addr: begin            rdata = mstatus\_q;          end          mcause\_addr: begin            rdata = mcause\_q;          end          mtvec\_addr: begin            rdata = mtvec\_q;          end          mepc\_addr: begin            rdata = mepc\_q;          end          default: rdata = '0;        endcase      end    end    always\_comb begin      mie\_wr\_flag = 1'b0;      mstatus\_wr\_flag = 1'b0;      mtvec\_wr\_flag = 1'b0;      if (reg\_wr) begin        case (addr)          mie\_addr: begin            mie\_wr\_flag = 1'b1;          end          mstatus\_addr: begin            mstatus\_wr\_flag = 1'b1;          end          mtvec\_addr: begin            mtvec\_wr\_flag = 1'b1;          end        endcase      end    end  endmodule |

**Listing 13. CSR\_reg.sv**

## Pipeline Register

|  |
| --- |
| module Pipeline\_reg #(      parameter WIDTH = 32,      reset = 32'h13  ) (      input logic clk,      input logic flush,      stall,      input logic [WIDTH-1:0] in,      output logic [WIDTH-1:0] out  );    logic not\_stalled;    assign not\_stalled = !stall;    always\_ff @(posedge clk) begin      if (flush) out <= WIDTH'(reset);      else if (not\_stalled) out <= in;    end  endmodule |

**Listing 14.** **Pipeline\_reg.sv**

# Testbench for Factorial

Following testbench has been created for our design simulation.

|  |
| --- |
| `include "RISC\_V.sv"  module RISC\_V\_tb;    logic rst, clk, ext\_inter, timer\_en;    logic [3:0] interrupt;    RISC\_V DUT (        .clk(clk),        .rst(rst),        .ext\_inter(ext\_inter),        .timer\_en(timer\_en)    );    //clock generation    localparam CLK\_PERIOD = 2;    initial begin      clk = 0;      forever begin        #(CLK\_PERIOD / 2);        clk = ~clk;      end    end    //Testbench    initial begin      rst = 1;      timer\_en = 0;      ext\_inter = 0;      @(posedge clk);      rst = 0;      repeat (200) @(posedge clk);      $finish;    end    //Monitor values at posedge    always @(posedge clk) begin      $strobe("PC=%d factorial=%d num=%d", DUT.datapath.Fetch.PC,              DUT.datapath.Decode\_instance.Regfile\_instance.mem[10],              DUT.datapath.Decode\_instance.Regfile\_instance.mem[11],);    end    initial begin      $dumpfile("RISC\_R\_dump.vcd");      $dumpvars;    end  endmodule |

**Listing 15. Testbench for Factorial algo**

## Assembly Code

|  |
| --- |
| #factorial.s  li a1, 4 # number =4  li a0, 1 # factorial  li t0, 0 #product  li t1, 2 #i  loop:  bgt t1, a1, end  addi t2 , t1 , 0 #t2=j  innerloop\_start:  beq t2, x0, innerloop\_end  add t0,t0,a0  addi t2,t2,-1  j innerloop\_start  innerloop\_end:  addi a0,t0,0  li t0,0  addi t1,t1,1  j loop  end:  j end |

**Assembly code for Factorial**

## Results

For the above testbench we get the following output.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 1. Monitor Results**

# Testbench for CSR

Following testbench has been created for our design simulation.

|  |
| --- |
| `include "RISC\_V.sv"  module RISC\_V\_tb;    logic rst, clk, ext\_inter, timer\_en;    logic [3:0] interrupt;    RISC\_V DUT (        .clk(clk),        .rst(rst),        .ext\_inter(ext\_inter),        .timer\_en(timer\_en)    );    //clock generation    localparam CLK\_PERIOD = 2;    initial begin      clk = 0;      forever begin        #(CLK\_PERIOD / 2);        clk = ~clk;      end    end    //Testbench    initial begin      rst = 1;      timer\_en = 0;      ext\_inter = 0;      @(posedge clk);      rst = 0;      repeat (4) @(posedge clk);      timer\_en = 1'b1;      repeat (7) @(posedge clk);      timer\_en = 1'b0;      repeat (5) @(posedge clk);      $finish;    end    //Monitor values at posedge    always @(posedge clk) begin      $strobe(          "PC=%0d\tFlush=%0d\tstall=%d\tinstr=%h|\nPC=%0d\tinstr=%h\tx1=%0h\tx2=%0h\tx3=%0h\tA=%h\tB=%h\tforw\_a=%0d\tforw\_b=%0d|\nPC=%0d\tALU\_o=%0h\twb\_data=%h\tmem=%h\tregwr=%b\twb\_sel=%b\tPC\_sel=%b\nmem\_wr=%b\tdata\_to\_wr=%h\nmie\_wr\_flag=%0h\tinterupt\_taken=%0h\tmstatus\_wr\_flag=%0h\tmtvec\_wr\_flag=%0h\ttimer\_inter=%0h\texternal\_inter=%0h\nmepc\_q=%0d\tmie\_q=%0h\tmstatus\_q=%0h\tmtvec\_q=%0h\tmcause\_q=%0h\tmip\_q=%0h\nISR\_addr=%0b\tcsr\_addr=%0h\tepc=%0d\tepc\_taken=%0h",          DUT.datapath.Fetch.PC, DUT.datapath.flush, DUT.datapath.stall,          DUT.datapath.Fetch.instruction, DUT.datapath.Decode\_instance.PC,          DUT.datapath.Decode\_instance.instruction,          DUT.datapath.Decode\_instance.Regfile\_instance.mem[1],          DUT.datapath.Decode\_instance.Regfile\_instance.mem[2],          DUT.datapath.Decode\_instance.Regfile\_instance.mem[3], DUT.datapath.Decode\_instance.ALU\_op\_a,          DUT.datapath.Decode\_instance.ALU\_op\_b, DUT.datapath.Decode\_instance.forw\_a,          DUT.datapath.Decode\_instance.forw\_b, DUT.datapath.wb\_stage\_instance.PC, DUT.datapath.wdata,          DUT.datapath.ALU\_wb, DUT.datapath.wb\_stage\_instance.data\_mem\_instance.mem[0], DUT.reg\_wr,          DUT.wb\_sel, DUT.PC\_sel, DUT.mem\_wr, DUT.datapath.wb\_stage\_instance.data\_to\_mem,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mie\_wr\_flag,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.interupt\_taken,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mstatus\_wr\_flag,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mtvec\_wr\_flag,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.timer\_inter,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.external\_inter,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mepc\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mie\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mstatus\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mtvec\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mcause\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.mip\_q,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.ISR\_addr,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.addr,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.epc,          DUT.datapath.wb\_stage\_instance.CSR\_reg\_instance.epc\_taken);      $display("\n---------------------------------------------");    end    initial begin      $dumpfile("RISC\_R\_dump.vcd");      $dumpvars;    end  endmodule |

**CSR\_Testbench**

## CSR\_Assembly

|  |
| --- |
| j main  j end  j end  j end  j end  j end  j end  j end  j timer\_handler  j end  j end  j end  j external\_handler  main:  li x11,0x80  li x12,0x8  li x13,0x4  csrrw x0,mie,x11  csrrw x0,mstatus,x12  csrrw x0,mtvec,x13  end:  j end  timer\_handler:  li x3,0x5  mret  external\_handler:  li x3,0x6  mret |

## Results

For the above testbench we get the following output.



