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

```
`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];
```

```verilog
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

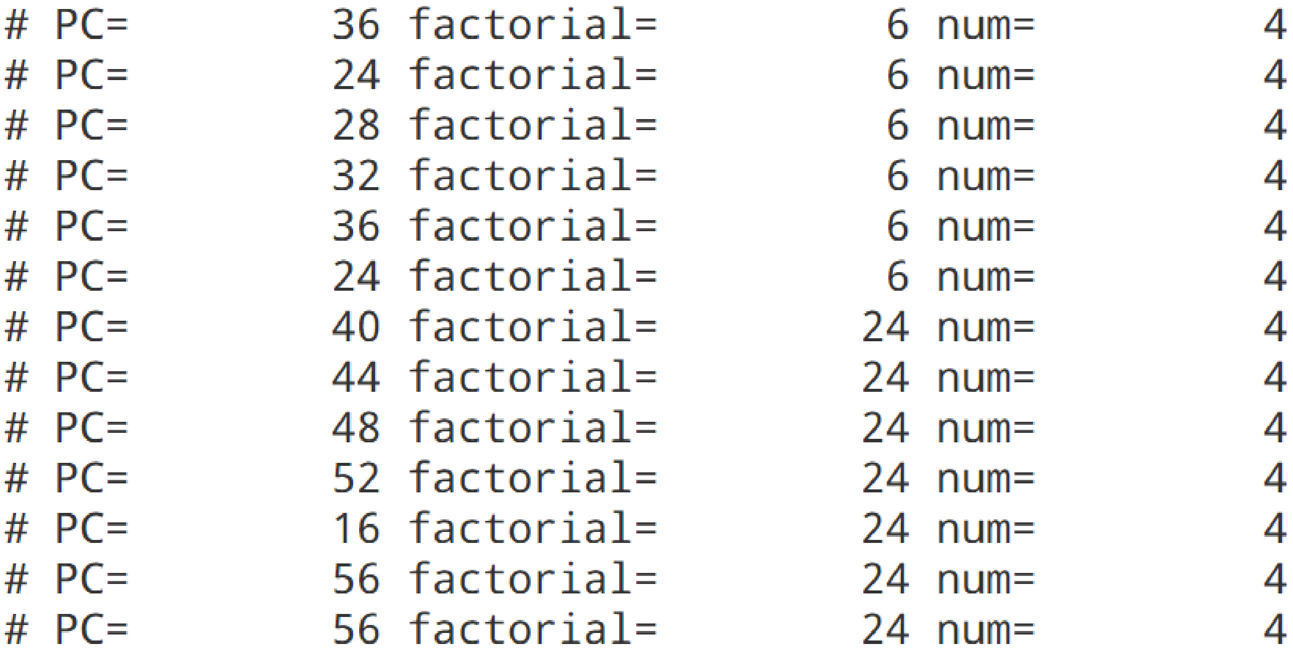
Assembly Code

```
#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.



```
# PC=          36 factorial=          6 num=          4
# PC=          24 factorial=          6 num=          4
# PC=          28 factorial=          6 num=          4
# PC=          32 factorial=          6 num=          4
# PC=          36 factorial=          6 num=          4
# PC=          24 factorial=          6 num=          4
# PC=          40 factorial=         24 num=          4
# PC=          44 factorial=         24 num=          4
# PC=          48 factorial=         24 num=          4
# PC=          52 factorial=         24 num=          4
# PC=          16 factorial=         24 num=          4
# PC=          56 factorial=         24 num=          4
# PC=          56 factorial=         24 num=          4
```

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\t
A=%h\tB=%h\tforw_a=%0d\tforw_b=%0d|\nPC=%0d\tALU_o=%0h\twb_data=%h\tmem=%h\tregwr=%b\twb_se
l=%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\tms
tatus_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
```

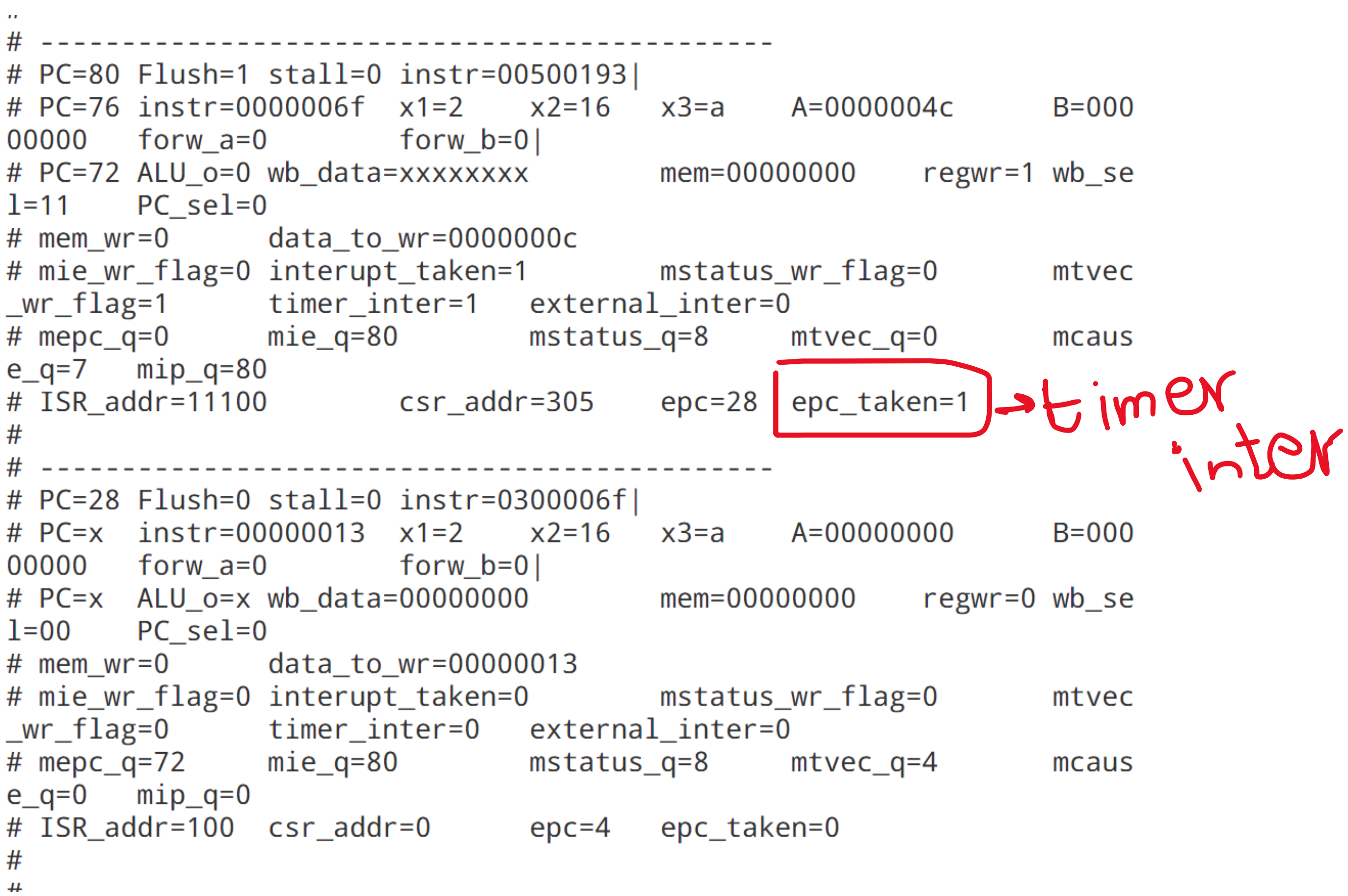
CSR_Testbench

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.



```
..
# --------------------------------------------
# PC=80 Flush=1 stall=0 instr=00500193|
# PC=76 instr=0000006f  x1=2    x2=16   x3=a    A=0000004c      B=000
00000   forw_a=0        forw_b=0|
# PC=72 ALU_o=0 wb_data=xxxxxxxx        mem=00000000    regwr=1 wb_se
l=11    PC_sel=0
# mem_wr=0      data_to_wr=0000000c
# mie_wr_flag=0 interupt_taken=1        mstatus_wr_flag=0       mtvec
_wr_flag=1      timer_inter=1   external_inter=0
# mepc_q=0      mie_q=80        mstatus_q=8     mtvec_q=0       mcaus
e_q=7   mip_q=80
# ISR_addr=11100        csr_addr=305    epc=28  epc_taken=1
#
# --------------------------------------------
# PC=28 Flush=0 stall=0 instr=0300006f|
# PC=x  instr=00000013  x1=2    x2=16   x3=a    A=00000000      B=000
00000   forw_a=0        forw_b=0|
# PC=x  ALU_o=x wb_data=00000000        mem=00000000    regwr=0 wb_se
l=00    PC_sel=0
# mem_wr=0      data_to_wr=00000013
# mie_wr_flag=0 interupt_taken=0        mstatus_wr_flag=0       mtvec
_wr_flag=0      timer_inter=0   external_inter=0
# mepc_q=72     mie_q=80        mstatus_q=8     mtvec_q=4       mcaus
e_q=0   mip_q=0
# ISR_addr=100  csr_addr=0      epc=4   epc_taken=0
#
#
```

→ timer inter