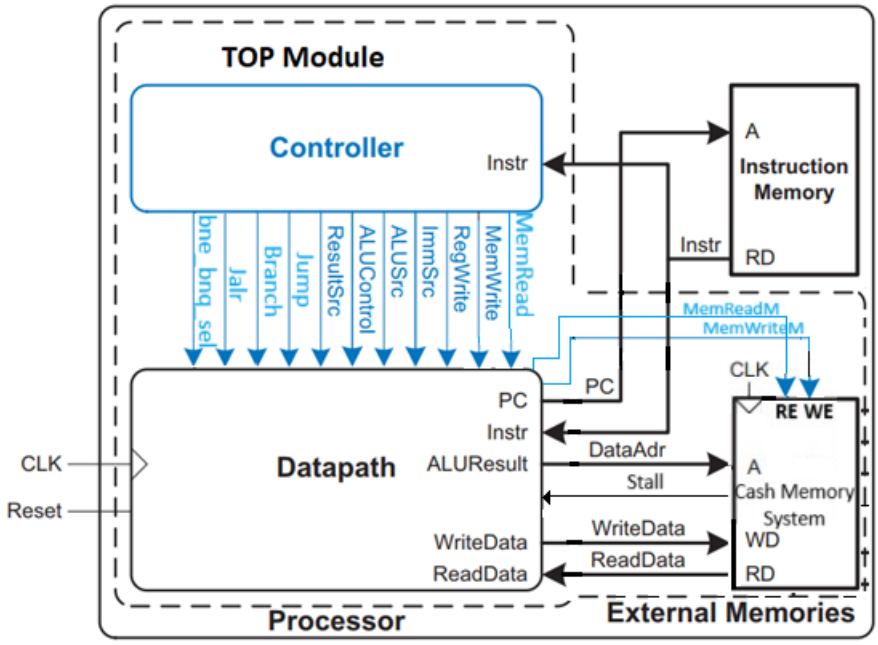
Abdallah Tarek Abdelnaby

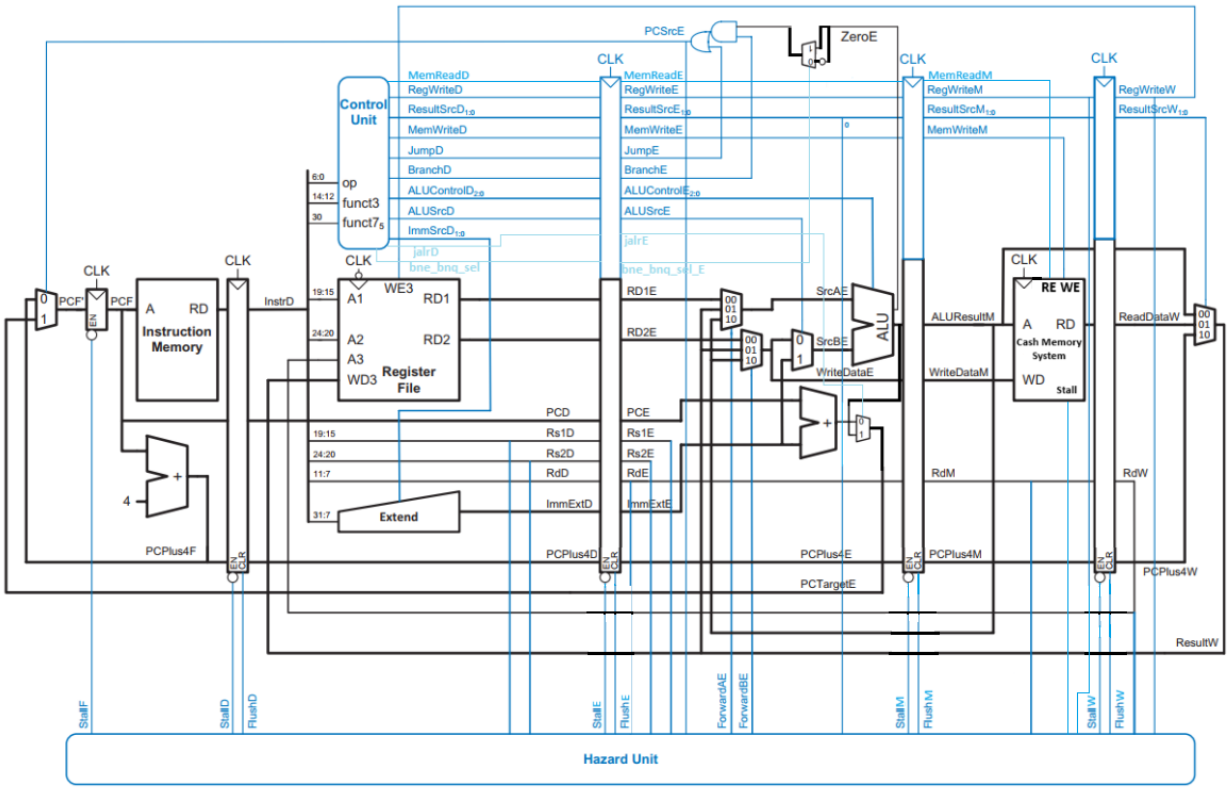
Digital_design

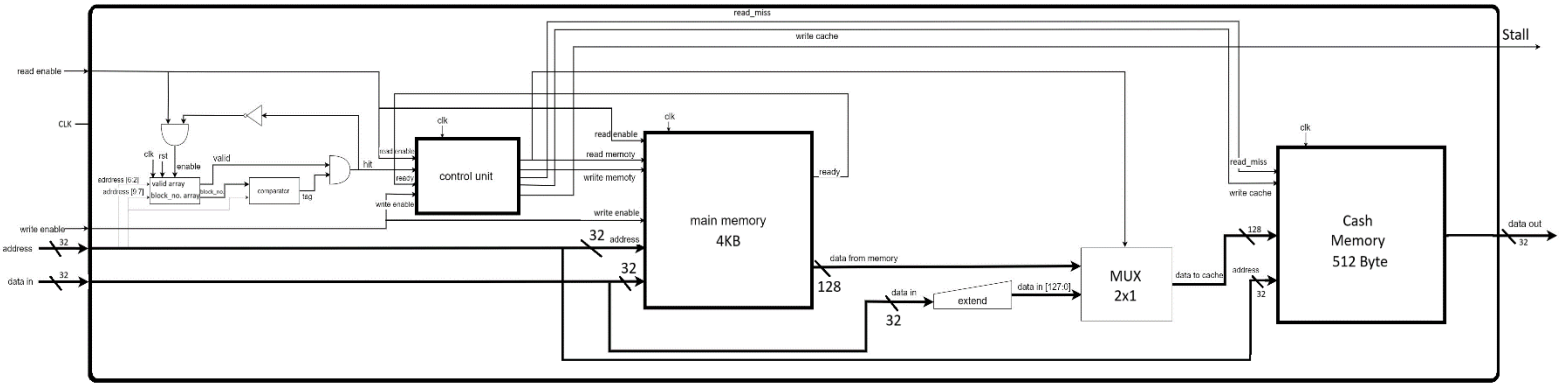
Project 1.3: RISC-V Pipelined Processor With Cash Memory System

Group_number: G4

The main block diagrams:







Cash Memory System

The building blocks of the design:

- **Memories**
  - Data_memory_system:
    - Cash_memory:

```
module cash_mem (address,data_in,data_out,clk,we,read_miss);
parameter length = 128;
parameter width = 32;

input clk,we,read_miss;
input [width-1:0]address;
input [(width*4)-1:0]data_in;
output [width-1:0]data_out;

reg [width-1:0] mem [length-1:0];

wire [$clog2(length)-1:0]trunc_add;
function [$clog2(length)-1:0] address_trunc;
input [width-1:0]add;
address_trunc = add;
endfunction

assign trunc_add = address_trunc (address);
assign data_out = mem [trunc_add];

always@(negedge clk)
begin
    if (we)
    begin
        if (read_miss)
        begin
            mem [{trunc_add[$clog2(length)-1:2],2'b00}] <= data_in [width-1:0];
            mem [{trunc_add[$clog2(length)-1:2],2'b01}] <= data_in [(width*2)-1:width];
            mem [{trunc_add[$clog2(length)-1:2],2'b10}] <= data_in [(width*3)-1:(width*2)];
            mem [{trunc_add[$clog2(length)-1:2],2'b11}] <= data_in [(width*4)-1:(width*3)];
        end
        else
        begin
            mem [trunc_add] <= data_in [width-1:0];
        end
    end
    else
    begin
        mem [trunc_add] <= mem [trunc_add];
    end
end
endmodule
```

- Data_memory

```
// 32bit word memory edged write edged read with coounter 4
//to emilate 0.25 speed compared to clk and one block read 16 byte
module data_mem (address,data_in,data_out,clk,mem_read,mem_write,we,re,ready);

parameter length = 1024;
parameter width = 32;

localparam idle      = 3'd0;
localparam rm1       = 3'd1;
localparam rm2       = 3'd2;
localparam rm3       = 3'd3;
localparam rm4       = 3'd4;

input clk,we,re,mem_read,mem_write;
input [width-1:0]address;
input [width-1:0]data_in;

output [(width*4)-1:0]data_out;
output reg ready;


reg [(width*4)-1:0] temp;
reg [width-1:0] mem [length-1:0];

wire [$clog2(length)-1:0]trunc_add;

function [$clog2(length)-1:0] address_trunc;
input [width-1:0]add;

address_trunc = add;
endfunction

assign trunc_add = address_trunc (address);

reg [2:0] current_state,next_state;
wire reset = ~(mem write | mem read);
```

1

```
always@(negedge clk)
begin
    if (reset)
        current_state <= idle;
    else
        current_state <= next_state;
end

always@(current_state,we,re)
begin
    case (current_state)    // synopsys full_case
    idle:begin
            next_state = rm1;
        end
    rm1:begin
            next_state = rm2;
        end
    rm2:begin
            next_state = rm3;
        end
    rm3:begin
            next_state = rm4;
        end
    rm4:begin
            next_state = idle;
        end
    endcase
end
```

2

```
always@(posedge clk)
begin
    if (current_state == rm4)
    begin
        if (we)
        begin
            mem [trunc_add] <= data_in;
            ready = 1'b1;
        end
        else if (re)
        begin
            temp = {mem [{trunc_add[$clog2(length)-1:2],2'b11}],
                    mem [{trunc_add[$clog2(length)-1:2],2'b10}],
                    mem [{trunc_add[$clog2(length)-1:2],2'b01}],
                    mem [{trunc_add[$clog2(length)-1:2],2'b00}]};
            ready = 1'b1;
        end
        else
        begin
            mem [trunc_add] <= mem [trunc_add];
            ready = 1'b0;
        end
    end
    else
    begin
        ready = 1'b0;
        mem [trunc_add] <= mem [trunc_add];
    end
end

assign  data_out = temp;
endmodule
```

3.

- Memory_control_unit

```
module memory_control_unit (clk,reset,add_in,mem_read,mem_write,ready_mem,stall
                            ,we_mem,re_mem,we_cash,read_miss);

parameter data_mem_length = 1024;
parameter cash_mem_length = 128;
parameter cash_mem_block_size = 4; //(cash_mem_block_size)per data_mem_word

localparam mem_block_no_bits = $clog2(data_mem_length) - $clog2(cash_mem_length);

localparam idle            = 4'd0;  //idle and read hit
localparam read_mem        = 4'd1;  //read miss and prepare data from mem
localparam read_miss_cash  = 4'd2;  // read_miss and deliver data from mem to cash
localparam write           = 4'd3;  //write hit or miss

input clk,reset,mem_read,mem_write,ready_mem;
input [31:0] add_in;
output reg stall,we_mem,re_mem,we_cash,read_miss;

reg [0:0] valid [(cash_mem_length/cash_mem_block_size)-1:0];
wire tag;
reg [mem_block_no_bits-1:0] mem_block_no [(cash_mem_length/cash_mem_block_size)-1:0];


wire h_m = (tag) & (valid [add_in [$clog2(cash_mem_length)-1:$clog2(cash_mem_block_size)]]);

reg [2:0] current_state,next_state;
always@(negedge clk)
begin
    if (reset)
        current_state <= idle;
    else
        current_state <= next_state;
end
```

1

```
always@(current_state,h_m,mem_read,mem_write,ready_mem)
begin
    case (current_state)
    idle:begin
            case ({mem_read,mem_write}) // synopsys full_case
            2'b10:begin
                    if (h_m)
                        next_state = idle;
                    else
                        next_state = read_mem;
                  end
            2'b01: next_state = write;
            2'b00: next_state = idle;
            endcase
         end
    read_mem:begin
            if (ready_mem)
                next_state = read_miss_cash;
            else
                next_state = read_mem;
          end
    read_miss_cash:begin
                    case ({mem_read,mem_write}) // synopsys full_case
                    2'b10:begin
                            if (h_m)
                                next_state = idle;
                            else
                                next_state = read_mem;
                          end
                    2'b01: next_state = write;
                    2'b00: next_state = idle;
                    endcase
                   end
    write:begin
            if (ready_mem)
                next_state = idle;
            else
                next_state = write;
          end
    default: next_state = idle;
    endcase
end
```

2

```
always@(current_state)
begin
    case (current_state)    //synopsys full_case
    idle:begin
            we_cash = 1'b0;
            we_mem = 1'b0;
            re_mem = 1'b0;
            read_miss = 1'b1;
            stall = 1'b0;
         end
    read_mem:begin
                we_cash = 1'b1;
                we_mem = 1'b0;
                re_mem = 1'b1;
                read_miss = 1'b1;
                stall = 1'b1;
             end
    read_miss_cash:begin
                    we_cash = 1'b1;
                    we_mem = 1'b0;
                    re_mem = 1'b0;
                    read_miss = 1'b1;
                    stall = 1'b0;
                   end
    write:begin
            if (h_m)
            begin
                we_cash = 1'b1;
                we_mem = 1'b1;
                re_mem = 1'b0;
                read_miss = 1'b0;
                stall = 1'b1;
            end
            else
            begin
                we_cash = 1'b0;
                we_mem = 1'b1;
                re_mem = 1'b0;
                read_miss = 1'b0;
                stall = 1'b1;
            end
          end
    endcase
end
```

3

```
assign tag = (mem_block_no [add_in [$clog2(cash_mem_length)-1:$clog2(cash_mem_block_size)]]
                == add_in[$clog2(data_mem_length)-1:$clog2(cash_mem_length)]) ? 1'b1 : 1'b0;


integer i;
always@(posedge clk)
begin
    if (reset)
    begin
        for (i=0;i<=(cash_mem_length/cash_mem_block_size)-1;i=i+1)
          begin
            valid[i] = 0;
            mem_block_no[i] = 0;
          end
    end
    else if ((~h_m) & mem_read)
    begin
        valid [add_in [$clog2(cash_mem_length)-1:$clog2(cash_mem_block_size)]] = 1'b1;
        mem_block_no [add_in [$clog2(cash_mem_length)-1:$clog2(cash_mem_block_size)]]
                        = add_in[$clog2(data_mem_length)-1:$clog2(cash_mem_length)];
    end
end


endmodule
```

4.

- Memory_System

```
`include "data_mem.v"
`include "cash_mem.v"
`include "memory_control_unit.v"
module memory_system (clk,mem_read,mem_write,reset,add_in,data_in,stall,data_out);

input clk,mem_read,mem_write,reset;
input [31:0] add_in,data_in;

output stall;
output [31:0] data_out;

//memory_control_unit (clk,reset,add_in,mem_read,mem_write,ready_mem,stall,we_mem
//                     ,re_mem,we_cash,read_miss);
wire ready_mem,we_mem,re_mem,we_cash,read_miss;
wire [127:0] data_interm_mem,data_interm_cash;
memory_control_unit mcu(.clk(clk),
                        .reset(reset),
                        .add_in(add_in),
                        .mem_read(mem_read),
                        .mem_write(mem_write),
                        .ready_mem(ready_mem),
                        .stall(stall),
                        .we_mem(we_mem),
                        .re_mem(re_mem),
                        .we_cash(we_cash),
                        .read_miss(read_miss));

//data_mem (address,data_in,data_out,clk,we,re,ready);
data_mem #(1024,32) data_mem1 (.address(add_in),        //length = 1024; width = 32;
                               .data_in(data_in),
                               .data_out(data_interm_mem),
                               .clk(clk),
                               .mem_read(mem_read),
                               .mem_write(mem_write),
                               .we(we_mem),
                               .re(re_mem),
                               .ready(ready_mem));

assign data_interm_cash = read_miss ? data_interm_mem : {96'd0,data_in};
//cash_mem (address,data_in,data_out,clk,we,read_miss)
cash_mem #(128,32) cash_mem1 (.address(add_in),     //length = 128; width = 32;
                              .data_in(data_interm_cash),
                              .data_out(data_out),
                              .clk(clk),
                              .we(we_cash),
                              .read_miss(read_miss));

endmodule
```

- Instruction_memory: 32bit Byte addressable ROM

```
module instruction_memory (address,data_out);

parameter length = 256;
parameter width = 32;

input [31:0]address;
output [31:0]data_out;
reg [width-1:0] mem [length-1:0];

assign data_out = mem [address>>2];

initial
begin
    $readmemh("instruction_mem_for_pipeline_and_cash.txt", mem);
end
endmodule
```

- **Control unit**

```
module control_unit (instr,result_sel,mem_write,alu_sel,imm_sel,mem_read,reg_write,alu_control,jalr_sel,bne_beq_sel,jump,branch);

localparam lw     = 7'b0000011;
localparam sw     = 7'b0100011;
localparam R_type = 7'b0110011;
localparam I_type = 7'b0010011;
localparam jal    = 7'b1101111;
localparam beq    = 7'b1100011;
localparam jalr   = 7'b1100111;

input [31:0] instr;

output mem_read,reg_write,alu_sel,mem_write,jalr_sel,branch,jump;
output reg bne_beq_sel;
output reg [2:0] alu_control;
output [1:0] result_sel,imm_sel;

wire [6:0] op_code = instr [6:0];
wire [2:0] func3 = instr [14:12];
wire func7_5 = instr [30];
wire [1:0] alu_operation;
reg [12:0] op;
assign mem_read = op [12];
assign reg_write = op [11];
assign imm_sel = op [10:9];
assign alu_sel = op [8];
assign mem_write = op [7];
assign result_sel = op [6:5];
assign branch = op [4];
assign jump = op [3];
assign alu_operation = op [2:1];
assign jalr_sel = op [0];

always@(op_code)   //main decoder
begin
    case (op_code)
//              mem_read reg_write  imm_sel  alu_sel   mem_write  result_sel   branch   jump   alu_operation   jalr_sel
    lw: op =     { 1'b1,   1'b1,     2'b00,    1'b1,     1'b0,      2'b01,      1'b0,   1'b0,      2'b00,      1'b0};
    sw: op =     { 1'b0,   1'b0,     2'b01,    1'b1,     1'b1,      2'b00,      1'b0,   1'b0,      2'b00,      1'b0};
    beq: op =    { 1'b0,   1'b0,     2'b10,    1'b0,     1'b0,      2'b00,      1'b1,   1'b0,      2'b01,      1'b0};
    jal: op =    { 1'b0,   1'b1,     2'b11,    1'b0,     1'b0,      2'b10,      1'b0,   1'b1,      2'b00,      1'b0};
    I_type: op = { 1'b0,   1'b1,     2'b00,    1'b1,     1'b0,      2'b00,      1'b0,   1'b0,      2'b10,      1'b0};
    R_type: op = { 1'b0,   1'b1,     2'b00,    1'b0,     1'b0,      2'b00,      1'b0,   1'b0,      2'b10,      1'b0};
    jalr: op =   { 1'b0,   1'b1,     2'b00,    1'b1,     1'b0,      2'b10,      1'b0,   1'b1,      2'b00,      1'b1};
    default : op = 0;
    endcase
end
```

1

```
always@(*)    //alu decoder
begin
    if (func3 == 3'b001)
        bne_beq_sel = 1'b0;
    else
        bne_beq_sel = 1'b1;
    //alu_control = alu_control;
    if (alu_operation == 2'b0)
        alu_control = 3'b000;    //add //lw,sw
    else if (alu_operation == 2'b01)
        alu_control = 3'b001;    //sub  //beq
    else      //alu_operation = 2'b10      //R_type
    begin
        case (func3)
        3'b000:
            begin
                if ({op_code[5],func7_5} == 2'b11)
                    alu_control = 3'b001;      ///sub
                else
                    alu_control = 3'b000;      ///add
            end
        3'b010: alu_control = 3'b101;        //set less than ///slt
        3'b110: alu_control = 3'b011;        ///or
        3'b111: alu_control = 3'b010;        ///and
        default :   alu_control = 3'b000;
        endcase
    end
end
endmodule
```

2.

- **Datapath components:**
    - Regester_file:

```
module reg_file (a1,a2,a3,wd3,rd1,rd2,clk,we3);

input clk,we3;
input [4:0] a1,a2,a3;
input [31:0] wd3;
output [31:0] rd1,rd2;

reg [31:0] mem [31:0];

assign  rd2 = (a2 == 5'd0) ? 32'd0 : mem [a2];
assign  rd1 = (a1 == 5'd0) ? 32'd0 : mem [a1];

always@(negedge clk)
begin
    if (we3)
        mem [a3] <= wd3;
    else
        mem [a3] <= mem [a3];
end
endmodule
```

- 2*1 MUX & 3*1 MUX:

```
module mux2 (mux_out,in0,in1,sel);
    output reg [31:0] mux_out;
    input [31:0] in0,in1;
    input sel;

    assign mux_out = sel ? in1 : in0;
endmodule

module mux3 (mux_out,in0,in1,in2,sel);
    output reg [31:0] mux_out;
    input [31:0] in0,in1,in2;
    input [1:0] sel;

    always@(*)
        case (sel)
        2'b00:  mux_out = in0;
        2'b01:  mux_out = in1;
        2'b10:  mux_out = in2;
        2'b11:  mux_out = 32'd0;
        endcase
endmodule
```

- ALU:

```
module ALU (zero,ALUout,a,b,ALUControl);

    output reg zero;
    output reg signed [31:0] ALUout;
    input [2:0] ALUControl;
    input [31:0] a,b;

    always @(*)
    begin
        case (ALUControl)
        3'b010 : ALUout = a & b;   //bitwise and
        3'b011 : ALUout = a | b;   //bitwise or
        3'b000 : ALUout = a + b;   //addition
        3'b001 : ALUout = a - b;   //subtraction
        3'b101 : ALUout = (a<b)? 1:0;   //compare
        //5 : ALUout = ~(a | b);
        default : ALUout = 0;
        endcase

        if (ALUout == 0)
            zero = 1;
        else
            zero = 0;
    end
endmodule
```

- Adder:

```
module full_adder_behave (f_sum,a,b);

    output reg [31:0] f_sum;
    input [31:0] a,b;

    always @ (*)
        f_sum = a + b ;
endmodule
```

- Sign_extend:

```
module sign_extend (in,out,sel);

input [31:7] in;
input [1:0] sel;
output reg [31:0] out;

always@(*)
    if (sel == 2'b00)    // I_type
        out = {{20{in[31]}}, in[31:20]};
    else if (sel == 2'b01)  // S_type
        out = {{20{in[31]}}, in[31:25], in[11:7]};
    else if (sel == 2'b10)       // B_type
        out = {{20{in[31]}}, in[7], in[30:25], in[11:8], 1'b0};
    else       // sel = 2'b11     // J_type
        out = {{12{in[31]}}, in[19:12], in[20], in[30:21], 1'b0};
endmodule
```

- PC_flip_flop:

```
module d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
input [n-1:0] in;
input clk,reset,en;
output reg [n-1:0] out;

always@(posedge clk)
    if (!en)
    begin
        if (reset)
            out <= 32'd0;
        else
            out <= in;
    end
    else
        out <= out;
endmodule
```

- **Datapath:**

```
`include "MUX.v"
`include "adder.v"
`include "reg_file.v"
`include "sign_extend.v"
`include "ALU.v"
`include "d_flip_flop_32.v"
`include "hazerd_unit.v"
module datapath (instr,instrD,read_data,clk,reset
                ,mem_read,reg_write,alu_selalu_control
                ,result_sel,imm_sel,pc_out,alu_result_out
                ,write_dataM,jalr_sel,bne_beq_sel,jump
                ,branch,mem_write,mem_writeM,mem_readM,stall);

input [31:0] instr;
input [31:0] read_data;
input clk,reset,stall,mem_read,reg_write;
input alu_sel,jalr_sel,bne_beq_sel,jump,branch,mem_write;
input [2:0] alu_control;
input [1:0] result_sel,imm_sel;

output mem_writeM,mem_readM;
output [31:0] pc_out;
output [31:0] alu_result_out;
output [31:0] write_dataM;
output [31:0] instrD;

wire reset_F,reset_D,reset_E,reset_M,reset_W;
wire pc_sel,en_F,en_D,en_E,en_M,en_W;

wire [31:0] pcF;

wire [31:0] pcD,pc_plus4D,rd1,rd2,immexD;
wire [4:0] rs1D,rs2D,rdD;
assign rs1D = instrD [19:15];
assign rs2D = instrD [24:20];
assign rdD = instrD [11:7];

wire mem_readE,reg_writeE,alu_selE,jalr_selE,bne_beq_selE;
wire mem_writeE,jumpE,branchE;
wire [2:0] alu_controlE;
wire [1:0] result_selE;
wire [31:0] pcE,pc_plus4E,rd1E,rd2E,immexE;
wire [4:0] rs1E,rs2E,rdE;
wire [31:0] write_dataE;
```

1

```
    wire reg_writeM;
    wire [1:0] result_selM;
    wire [31:0] pc_plus4M,alu_resultM;
    wire [4:0] rdM;

    wire reg_writeW;
    wire [1:0] result_selW;
    wire [31:0] pc_plus4W,alu_resultW,read_dataW;
    wire [4:0] rdW;

    wire [1:0] forwardAE,forwardBE;
    wire stallF,stallD,stallE,stallM,stallW,flushD,flushE;

    wire [31:0] sourceB,sourceA;
    wire zero,zero_flag;

    //mux2 (mux_out,in0,in1,sel)
    wire pc_sel_real;
    wire [31:0] pc_next,pc_target,pc_plus4;
    wire [31:0] pc_or_reg;    //for jalr selection
    mux2 pc_mux(.mux_out (pc_next),
                .in0 (pc_plus4),
                .in1 (pc_or_reg),
                .sel (pc_sel_real));

    //full_adder_behave (f_sum,a,b)
    full_adder_behave add_plus_4(.f_sum (pc_plus4),
                                 .a (32'd4),
                                 .b (pcF));

    //reg_file (a1,a2,a3,wd3,rd1,rd2,clk,we3)
    wire [31:0] result;
    reg_file reg_file1(.a1 (instrD[19:15]),
                       .a2 (instrD[24:20]),
                       .a3 (rdW),
                       .wd3 (result),
                       .rd1 (rd1),
                       .rd2 (rd2),
                       .clk (clk),
                       .we3 (reg_writeW));

    //sign_extend (in,out,sel)
    sign_extend extend(.in (instrD[31:7]),
                       .out (immexD),
                       .sel (imm_sel));
```

2

```
  //mux3 (mux_out,in0,in1,in2,sel)
mux3 source_forwardingA(.mux_out (sourceA),
                        .in0 (rd1E),
                        .in1 (result),
                        .in2 (alu_resultM),
                        .sel (forwardAE));

  //mux3 (mux_out,in0,in1,in2,sel)
mux3 source_forwardingB(.mux_out (write_dataE),
                        .in0 (rd2E),
                        .in1 (result),
                        .in2 (alu_resultM),
                        .sel (forwardBE));

  //full_adder_behave (f_sum,a,b)
full_adder_behave add_imm(.f_sum (pc_target),
                          .a (immexE),
                          .b (pcE));

  //mux2 (mux_out,in0,in1,sel)
mux2 reg_out_mux(.mux_out (sourceB),
                 .in0 (write_dataE),
                 .in1 (immexE),
                 .sel (alu_selE));

  wire [31:0] alu_res;
  //ALU (zero,ALUout,a,b,ALUControl)
ALU alu1(.zero (zero),
         .ALUout (alu_res),
         .a (sourceA),
         .b (sourceB),
         .ALUControl (alu_controlE));
  assign zero_flag = bne_beq_selE ? zero : ~zero;
  assign pc_sel_real = pc_sel ? (jumpE | (zero_flag & branchE)) : 1'b0;

  //mux2 (mux_out,in0,in1,sel)
mux2 jalr_mux(.mux_out (pc_or_reg),
              .in0 (pc_target),
              .in1 (alu_res),
              .sel (jalr_selE));

  //mux3 (mux_out,in0,in1,in2,sel)
mux3 result_mux(.mux_out (result),
                .in0 (alu_resultW),
                .in1 (read_dataW),
                .in2 (pc_plus4W),
                .sel (result_selW));
```

3

```
  wire en_F_real = en_F ? stallF : 1'b0;
  //d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
d_flip_flop #(32) featch(.in(pc_next),
                         .out(pcF),
                         .clk(clk),
                         .reset(reset_F),
                         .en(en_F_real));

  wire reset_D_real = reset_D ? 1'b1 : flushD;
  wire en_D_real = en_D ? stallD : 1'b0;
  //d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
  wire [95:0] decode_in = {instr,pcF,pc_plus4};
  wire [95:0] decode_out;
  assign {instrD,pcD,pc_plus4D} = decode_out;
d_flip_flop #(96) decode(.in(decode_in),
                         .out(decode_out),
                         .clk(clk),
                         .reset(reset_D_real),
                         .en(en_D_real));

  //d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
  wire reset_E_real = reset_E ? 1'b1 : flushE;
  wire en_E_real = en_E ? stallE : 1'b0;
wire [187:0] excute_in = {mem_read,reg_write,alu_sel,jalr_sel
                          ,bne_beq_sel,alu_control,result_sel
                          ,mem_write,rd1,rd2,pcD,rs1D,rs2D,rdD
                          ,immexD,pc_plus4D,jump,branch};
  wire [187:0] excute_out;
assign {mem_readE,reg_writeE,alu_selE,jalr_selE,bne_beq_selE
        ,alu_controlE,result_selE,mem_writeE,rd1E,rd2E,pcE
        ,rs1E,rs2E,rdE,immexE,pc_plus4E,jumpE,branchE} = excute_out;
d_flip_flop #(188) excute(.in(excute_in),
                          .out(excute_out),
                          .clk(clk),
                          .reset(reset_E_real),
                          .en(en_E_real));
```

4

```
  //d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
  wire en_M_real = en_M ? stallM : 1'b0;
wire [105:0] mem_in = {mem_readE,reg_writeE,result_selE,mem_writeE
                       ,alu_res,write_dataE,rdE,pc_plus4E};
  wire [105:0] mem_out;
assign {mem_readM,reg_writeM,result_selM,mem_writeM,alu_resultM
        ,write_dataM,rdM,pc_plus4M} = mem_out;
d_flip_flop #(106) memory(.in(mem_in),
                          .out(mem_out),
                          .clk(clk),
                          .reset(reset_M),
                          .en(en_M_real));

  //d_flip_flop #(parameter n = 32)(in,out,clk,reset,en);
  wire en_W_real = en_W ? stallW : 1'b0;
wire [103:0] write_back_in = {reg_writeM,result_selM,alu_resultM
                              ,read_data,rdM,pc_plus4M};
  wire [103:0] write_back_out;
assign {reg_writeW,result_selW,alu_resultW,read_dataW,rdW
        ,pc_plus4W} = write_back_out;
d_flip_flop #(104) write_back(.in(write_back_in),
                              .out(write_back_out),
                              .clk(clk),
                              .reset(reset_W),
                              .en(en_W_real));
```

5

```
//hazerd_unit (rs1D,rs2D,rdE,rs1E,rs2E,pc_sel,result_selE,rdM,reg_writeM
//             ,rdW,reg_writeW,forwardAE,forwardBE,stallF,stallD,stallE
//             ,stallM,stallW,flushD,flushE,stall);
hazerd_unit u0(.rs1D(rs1D),
               .rs2D(rs2D),
               .rdE(rdE),
               .rs1E(rs1E),
               .rs2E(rs2E),
               .pc_sel(pc_sel_real),
               .result_selE(result_selE[0]),
               .rdM(rdM),
               .reg_writeM(reg_writeM),
               .rdW(rdW),
               .reg_writeW(reg_writeW),
               .forwardAE(forwardAE),
               .forwardBE(forwardBE),
               .stallF(stallF),
               .stallD(stallD),
               .stallE(stallE),
               .stallM(stallM),
               .stallW(stallW),
               .flushD(flushD),
               .flushE(flushE),
               .stall(stall));

//output assignment
assign pc_out = pcF;
assign alu_result_out = alu_resultM;
assign {en_F,en_D,en_E,en_M,en_W} = reset ? 5'b0_0_0_0_0 : 5'b1_1_1_1_1;
assign {reset_F,reset_D,reset_E,reset_M,reset_W} = reset ? 5'b1_1_1_1_1 : 5'b0_0_0_0_0;
assign pc_sel = reset ? 1'b0 : 1'b1;

endmodule
```

6.

- **Hazard_unit:**

```
module hazerd_unit (rs1D,rs2D,rdE,rs1E,rs2E,pc_sel,result_selE,rdM,reg_writeM,rdW
                    ,reg_writeW,forwardAE,forwardBE,stallF,stallD,stallE,stallM
                    ,stallW,flushD,flushE,stall);

input pc_sel,result_selE,reg_writeM,reg_writeW,stall;
input [4:0] rdE,rdM,rdW,rs1D,rs2D,rs1E,rs2E;

output reg stallF,stallD,stallE,stallM,stallW,flushD,flushE;
output reg [1:0] forwardAE,forwardBE;

always@(*)
begin
    //------------------forward for data Hazerd-----------------------------
    if (((rs1E == rdM) & reg_writeM) & (rs1E != 0))
        forwardAE = 2'b10;
    else if (((rs1E == rdW) & reg_writeW) & (rs1E != 0))
        forwardAE = 2'b01;
    else
        forwardAE = 2'b00;

    if (((rs2E == rdM) & reg_writeM) & (rs2E != 0))
        forwardBE = 2'b10;
    else if (((rs2E == rdW) & reg_writeW) & (rs2E != 0))
        forwardBE = 2'b01;
    else
        forwardBE = 2'b00;
    //--------------------------------------------------------------------
end
```

1

```
wire lwStall;
assign lwStall = result_selE/*[0]*/ & ((rs1D == rdE) | (rs2D == rdE));
always@(lwStall or pc_sel or stall)
begin
    if (stall)      //memory system stall
        begin
            stallF = 1'b1;
            stallD = 1'b1;
            stallE = 1'b1;
            stallM = 1'b1;
            stallW = 1'b1;
        end
    else
        begin
        //------------------stall for load Hazerd----------------------
        if (lwStall == 1'b1)
            begin
                stallF = lwStall;
                stallD = lwStall;
                stallE = 1'b0;
                stallM = 1'b0;
                stallW = 1'b0;
            end
        else
            begin
                stallF = 1'b0;
                stallD = 1'b0;
                stallE = 1'b0;
                stallM = 1'b0;
                stallW = 1'b0;
            end
        end
    //-------------------------------------------------------------------
    flushE = 1'b0;
    flushD = 1'b0;

    //------------------flush for controls Hazerd----------------------
    if ((lwStall | pc_sel)== 1'b1)
    begin
        flushD = pc_sel;
        flushE = lwStall | pc_sel;
    end
    //-------------------------------------------------------------------
end
endmodule
```

2.

- **Top_module:**

```
`include "datapath.v"
`include "memory_system.v"
`include "control_unit.v"
module top_module (clk,pc_out,instr,reset_ms,reset);

input clk,reset_ms,reset;
input [31:0] instr;
output [31:0] pc_out;

wire [31:0] write_data,read_data,alu_result;
wire write_en,read_en,stall;

//memory_system (clk,mem_read,mem_write,reset,add_in,data_in,stall,data_out);
memory_system ms(.clk(clk),
                 .mem_read(read_en),
                 .mem_write(write_en),
                 .reset(reset_ms),
                 .add_in(alu_result),
                 .data_in(write_data),
                 .stall(stall),
                 .data_out(read_data));

//control_unit (instr,result_sel,mem_write,alu_sel,imm_sel,mem_read,reg_write
//              ,alu_control,jalr_sel,bne_beq_sel,jump,branch);
wire alu_sel,mem_read,reg_write,jalr_sel,bne_beq_sel,jump,branch,mem_write;
wire [1:0] result_sel,imm_sel;
wire [2:0] alu_control;
wire [31:0] instrD;
control_unit c1(.instr(instrD),
                .result_sel(result_sel),
                .mem_write(mem_write),
                .alu_sel(alu_sel),
                .imm_sel(imm_sel),
                .mem_read(mem_read),
                .reg_write(reg_write),
                .alu_control(alu_control),
                .jalr_sel(jalr_sel),
                .bne_beq_sel(bne_beq_sel),
                .jump(jump),
                .branch(branch));

//datapath (instr,instrD,read_data,clk,reset,mem_read,reg_write,alu_sel,alu_control
//          ,result_sel,imm_sel,pc_out,alu_result_out,write_dataM,jalr_sel,bne_beq_sel
//          ,jump,branch,mem_write,mem_writeM,mem_readM,stall);
datapath d1(.instr(instr),
            .instrD(instrD),
            .read_data(read_data),
            .clk(clk),
            .reset(reset),
            .mem_read(mem_read),
            .reg_write(reg_write),
            .alu_sel(alu_sel),
            .alu_control(alu_control),
            .result_sel(result_sel),
            .imm_sel(imm_sel),
            .pc_out(pc_out),
            .alu_result_out(alu_result),
            .write_dataM(write_data),
            .jalr_sel(jalr_sel),
            .bne_beq_sel(bne_beq_sel),
            .jump(jump),
            .branch(branch),
            .mem_write(mem_write),
            .mem_writeM(write_en),
            .mem_readM(read_en),
            .stall(stall));   //remove read_en in case of normal ram or mem
endmodule
```

Test_bench:

```
`include "instruction_memory.v"
`include "top_module.v"
module top_tb ();

localparam t = 25;

reg clk,reset_ms,reset;
wire [31:0] instr,pc_out;

top_module t1(.clk(clk),
              .reset_ms(reset_ms),
              .reset(reset),
              .pc_out(pc_out),
              .instr(instr));

//instruction_memory (address,data_out,clk);
instruction_memory instruct(.address (pc_out),
                            .data_out (instr));

initial
begin
    clk = 0;
    forever #(t/2) clk = ~clk;
end
```

1

```
initial
begin

    $dumpfile ("top_tb.vcd");
    $dumpvars (0,top_tb);
    reset_ms = 1'b1;
    reset = 1'b1;
    #t
    reset_ms = 1'b0;    //to free the memory system reset
    reset = 1'b0;       //to free RISC resets and enables and pc_sel
    #t
    #(t*165)    ///wait for the program to finish
    $dumpoff;
/* add_h  add_d  val_h
    0xe0   224 >> 19
    0x60   96  >> 7
    0x50   80  >> c
    0x2d   45  >> c
    0x28   40  >> a
    0x20   32  >> a
    0x1f   31  >> c
    0x1e   30  >> c
    0x1b   27  >> 7
    0x1a   26  >> 7
    0x18   24  >> e
    0x14   20  >> e4
    0x10   16  >> e
    0xf    15 !>> a0
    0xc    12  >> e
    0x8    8   >> a
    0x6    6   >> c
    0x5    5   >> a
    0x2    2   >> 7
    */
```

2

```
//////////////cash check//////////////////////////
    if (tl.ms.data_meml.mem[8] == 32'ha)
        $display ("success in add 0x8");
    else
        $display ("failure in add 0x8");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[80] == 32'hc)
        $display ("success in add 0x50");
    else
        $display ("failure in add 0x50");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[16] == 32'he)
        $display ("success in add 0x10");
    else
        $display ("failure in add 0x10");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[24] == 32'he)
        $display ("success in add 0x18");
    else
        $display ("failure in add 0x18");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[32] == 32'ha)
        $display ("success in add 0x20");
    else
        $display ("failure in add 0x20");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[6] == 32'd12)
        $display ("success in add 0x6");
    else
        $display ("failure in 0x6");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[5] == 32'd1
        $display ("success in add 0x5");
    else
        $display ("failure in add 0x5");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[12] == 32'd14)
        $display ("success in add 0xc");
    else
        $display ("failure in add 0xc");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[40] == 32'd10)
        $display ("success in add 0x28");
    else
        $display ("failure in add 0x28");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[45] == 32'hc)
        $display ("success in add 0x2d");
    else
        $display ("failure in add 0x2d");
```

3

```
//////////////pipline processor check//////////
    if (tl.ms.data_meml.mem[96] == 32'h7)
        $display ("success in add 0x60");
    else
        $display ("failure in 0x60");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[224] == 32'h19)
        $display ("success in add 0xe0");
    else
        $display ("failure in add 0xe0");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[2] == 32'h7)
        $display ("success in add 0x2");
    else
        $display ("failure in add 0x2");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[15] != 32'hc0)
        $display ("success jalr jumping");
    else
        $display ("failure jalr jumping");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[20] == 32'he4)
        $display ("success in add 0x14");
    else
        $display ("failure in add 0x14");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[30] == 32'hc)
        $display ("success in add 0x1e");
    else
        $display ("failure in add 0x1e");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[31] == 32'hc)
        $display ("success in add 0x1f");
    else
        $display ("failure in add 0x1f");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[26] == 32'h7)
        $display ("success in add 0x1a");
    else
        $display ("failure in add 0x1a");
//////////////////////////////////////////
    if (tl.ms.data_meml.mem[27] == 32'h7)
        $display ("success in add 0x1b");
    else
        $display ("failure in add 0x1b");
//////////////////////////////////////////
    $stop;

end
endmodule
```

4.

The program loaded in the instruction memory:

```
1   addi x5, x0, 10 # x5 = 10                          (0x0)   0x00A00293
2   addi x6, x0, 12 # x6 = 12                          (0x4)   0x00C00313
3   addi x7, x0, 14 # x7 = 14                          (0x8)   0x00E00393
4   sw x5, 5(x0)    #write miss around                 (0xc)   0x005022A3
5   lw x8, 5(x0)    #read miss                         (0x10)  0x00502403
6   sw x6, 6(x0)    #write hit throgh                  (0x14)  0x00602323
7   lw x9, 6(x0)    #read hit                          (0x18)  0x00602483
8   sw x7, 12(x0)   #write miss around                 (0x1c)  0x00702623
9   lw x10, 12(x0)  #read miss                         (0x20)  0x00C02503
10  lw x12, 12(x0)  #read hit                          (0x24)  0x00C02603
11  add x3, x12, x5 #                                  (0x28)  0x005601B3
12  add x2, x12, x6 #                                  (0x2c)  0x00660133
13  sw x5, 40(x0)   #write miss around                 (0x30)  0x00502F23
14  lw x13, 40(x0)  #read miss                         (0x34)  0x01E02683
15  sw x6, 45(x0)   #write miss around                 (0x38)  0x026021A3
16  addi x5, x0, 10 # x5 = 10                          (0x3c)  0x00A00293
17  addi x6, x0, 12 # x6 = 12                          (0x40)  0x00C00313
18  addi x7, x0, 14 # x7 = 14                          (0x44)  0x00E00393
19  sw x8, 8(x0)    #                                  (0x48)  0x00802423
20  sw x9, 80(x0)   #                                  (0x4c)  0x04902823
21  sw x10, 16(x0)  #                                  (0x50)  0x00A02823
22  sw x12, 24(x0)  #                                  (0x54)  0x00C02C23
23  sw x13, 32(x0)  #                                  (0x58)  0x02D02023
24  main: addi x2, x0, 5 # x2 = 5                      (0x5c)  0x00500113
25  addi x3, x0, 12 # x3 = 12                          (0x60)  0x00C00193
26  addi x7, x3, -9 # x7 = (12 - 9) = 3                (0x64)  0xFF718393
27  or x4, x7, x2 # x4 = (3 OR 5) = 7                  (0x68)  0x0023E233
28  and x5, x3, x4 # x5 = (12 AND 7) = 4               (0x6c)  0x0041F2B3
29  add x5, x5, x4 # x5 = 4 + 7 = 11                   (0x70)  0x004282B3
30  beq x5, x7, end # shouldn't be taken               (0x74)  0x02728863
31  slt x4, x3, x4 # x4 = (12 < 7) = 0                 (0x78)  0x0041A233
32  beq x4, x0, around # should be taken               (0x7c)  0x00020463
33  addi x5, x0, 0 # shouldn't execute                 (0x80)  0x00000293
34  around: slt x4, x7, x2 # x4 = (3 < 5) = 1          (0x84)  0x0023A233
35  add x7, x4, x5 # x7 = (1 + 11) = 12                (0x88)  0x005203B3
36  add x30, x7, x0 # x30 = 12 =! 3 #test forwarding   (0x8c)  0x00038F33
37  add x31, x0, x7 # x31 = 12 =! 3 #test forwarding   (0x90)  0x00700FB3
38  sw x30, 30(x0) # [30] = 12 =! 3 #test forwarding   (0x94)  0x01E02F23
39  sw x31, 31(x0) # [31] = 12 =! 3 #test forwarding   (0x98)  0x01F02FA3
40  sub x7, x7, x2 # x7 = (12 - 5) = 7                 (0x9c)  0x402383B3
41  sw x7, 84(x3) # [96] = 7                           (0xa0)  0x0471AA23
42  lw x2, 96(x0) # x2 = [96] = 7                      (0xa4)  0x06002103
43  add x26, x2, x0 # x26 = 7 =! 5 #test stall         (0xa8)  0x00010D33
```

```
44  add x27, x0, x2 # x27 = 7 =! 5 #test stall       (0xac)   0x00200DB3
45  sw x26, 26(x0) # [26] = 7 =! 5 #test stall       (0xb0)   0x01A02D23
46  sw x27, 27(x0) # [27] = 7 =! 5 #test stall       (0xb4)   0x01B02DA3
47  add x9, x2, x5 # x9 = (7 + 11) = 18              (0xb8)   0x005104B3
48  jal x3, end # jump to end, x3 = 0xc0             (0xbc)   0x008001EF
49  addi x2, x0, 1 # shouldn't execute               (0xc0)   0x00100113
50  end: add x2, x2, x9 # x2 = (7 + 18) = 25         (0xc4)   0x00910133
51  sw x2, 0x20(x3) # [224] = 25                     (0xc8)   0x0221A023
52  bne x7,x5,test_bne # should be taken             (0xcc)   0x00539463
53  done: beq x2, x2, done # infinite loop           (0xd0)   0x00210063
54  test_bne: sw x7, 2(x0) # [2] = 7                 (0xd4)   0x00702123
55  addi x11, x0, 8 # x11 = 8                        (0xd8)   0x00800593
56  sw x11, 15(x0) # [15] = 8                        (0xdc)   0x00B027A3
57  my_place: jalr x7, x11, my_place #               (0xe0)   0x060583E7
58  sw x3, 15(x0) # [15] != 0xc0                     (0xe4)   0x003027A3
59  test_jlr: sw x7, 20(x0) # [20] = 0xe8            (0xe8)   0x00702A23
60  bne x7,x5,done # should be taken                 (0xec)   0xFE5394E3
```

The machine code loaded in the instruction memory:

00A00293

00C00313

00E00393

005022A3

00502403

00602323

00602483

00702623

00C02503

00C02603

005601B3

00660133

02502423

02802683

026026A3

00A00293

00C00313

00E00393

00802423

04902823

00A02823

00C02C23

02D02023

00500113

00C00193

FF718393

0023E233

0041F2B3

004282B3

04728863

0041A233

00020463

00000293

0023A233

005203B3

00038F33

00700FB3

01E02F23

01F02FA3

402383B3

0471AA23

06002103

00010D33

00200DB3

01A02D23

01B02DA3

005104B3

008001EF

00100113

00910133

0221A023

00539463

00210063

00702123

00800593

00B027A3

0E0583E7

003027A3

00702A23

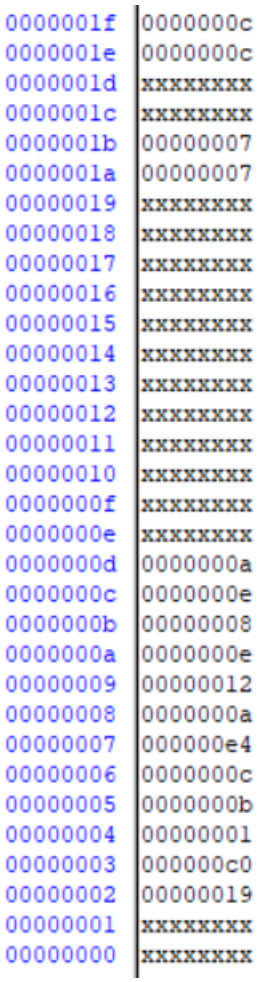
FE5392E3

The result from test bench:

```
VSIM 2> run -all
# success in add 0x8
# success in add 0x50
# success in add 0x10
# success in add 0x18
# success in add 0x20
# success in add 0x6
# success in add 0x5
# success in add 0xc
# success in add 0x28
# success in add 0x2d
# success in add 0x60
# success in add 0xe0
# success in add 0x2
# success jalr jumping
# success in add 0x14
# success in add 0x1e
# success in add 0x1f
# success in add 0x1a
# success in add 0x1b
# ** Note: $stop    : F:/ITI/RISC-V/pipline/ser/top_tb.v(157)
```

mem_add_h	mem_add_d		val_h
0xe0	224	>>	0x19
0x60	96	>>	0x7
0x50	80	>>	0xc
0x2d	45	>>	0xc
0x28	40	>>	0xa
0x20	32	>>	0xa
0x1f	31	>>	0xc
0x1e	30	>>	0xc
0x1b	27	>>	0x7
0x1a	26	>>	0x7
0x18	24	>>	0xe
0x14	20	>>	0xe4
0x10	16	>>	0xe
0xf	15	!>>	0xa0
0xc	12	>>	0xe
0x8	8	>>	0xa
0x6	6	>>	0xc
0x5	5	>>	0xa
0x2	2	>>	0x7

Reg_file



Comments:

○ RISC-V TEST Comments:

- It stored in the address 0x60 (0d96) 0x7 which is the value of 0x7 register of the register file -at run time- which verify the following instructions (as 0x7 can't have 0x7 value unless the success of the instructions) :

  Sw, add, sub, beq, slt, addi, or, and.

- It stored in the address 0xe0 (0d224) 0x19 (0d25) which is the value of 0x2 register in the register file which verify the following instructions (as 0x2 can't have 0x19 (0d25) value unless the success of the instructions) :

  lw, jal.

- It stored in the address 0x2 (0d2) 0x7 (0d7) which is the value of 0x7 register in the register file which verify the following instructions (as this instruction won't be executed if it wasn't for bne (bnq jumped is infinite loop)):

  bne.

- It didn't store in the address 0x3 (0d3) 0xc0 (0d192) which is the value of 0x3 register in the register file which verify the jumping part of the following instructions (as this instruction jumps the sw instruction successfully):

  jalr.

- It didn't store in the address 0x14 (0d20) 0xe4 (0d228) which is the value of 0x7 register in the register file which verify the storing of the return (pc+4) part of the following instructions (as this instruction stores the next instruction address (pc+4) in the 0x7 (destination register) successfully):

  jalr.

- It stored in the address 0x1e (0d30) 0xc (0d12) which is the value of 0x30 register in the register file which verify the successful forwarding from the memory stage of an instruction to the execution of dependent instruction.

- It stored in the address 0x1f (0d31) 0xc (0d12) which is the value of 0x31 register in the register file which verify the successful forwarding from the write back stage of an instruction to the execution of dependent instruction.

- It stored in the address 0x1a (0d26) 0x7 (0d7) which is the value of 0x26 register in the register file which verify the successful stalling from lw instruction to a dependent instruction till the write back stage of lw and forwarding to the execution stage to the dependent instruction.

- It stored in the address 0x1b (0d27) 0x7 (0d7) which is the value of 0x27 register in the register file which verify the successful stalling from lw instruction to a dependent instruction till the write back stage of lw and read it directly from the register file in the decode stage of the dependent instruction.

- Every B or J type instruction successfully executed proves the control hazard solution is successful.

- **Cash Memory System TEST Comments:**

- It stored in the address 0x5 (0d5) 0xa which is the value of 0x5 register of the register file
  -at run time- which verify write miss operation.

- - It stored in the address 0x6 (0d6) 0xc which is the value of 0x6 register of the register file
  -at run time- which verify write hit after read miss operation.

- - It stored in the address 0xc (0d12) 0xe which is the value of 0x7 register of the register file
  -at run time- which verify write miss after read hit operation.

- - It stored in the address 0x28 (0d40) 0xa which is the value of 0x5 register of the register file
  -at run time- which verify write miss after read hit with load stall operation.

- - It stored in the address 0x2d (0d45) 0xc which is the value of 0x6 register of the register file
  -at run time- which verify write miss after read miss operation.

- - It stored in the address 0x8 (0d8) 0xa which is the value of 0x8 register of the register file
  -at run time- which verify read miss after write miss operation on different blocks.

- - It stored in the address 0x50 (0d80) 0xc which is the value of 0x9 register of the register file
  -at run time- which verify read hit after write hit operation.

- - It stored in the address 0x18 (0d24) 0xe which is the value of 0x12 register of the register file
  -at run time- which verify read hit of an address which is the same of the instruction before which is does read miss operation.

- - It stored in the address 0x20 (0d32) 0xa which is the value of 0x13 register of the register file
  -at run time- which verify read miss after write miss operation on the same data block.
  .