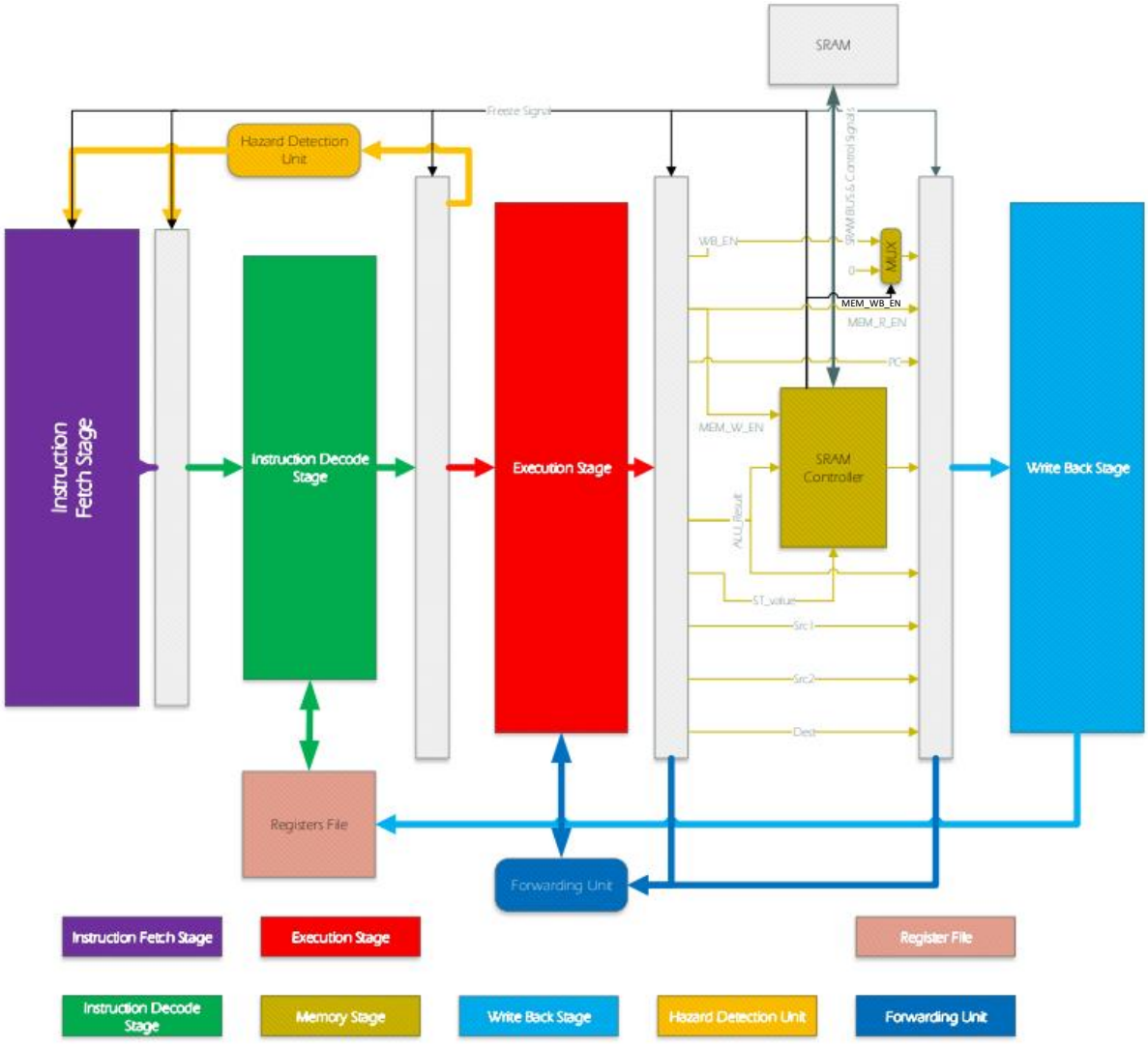
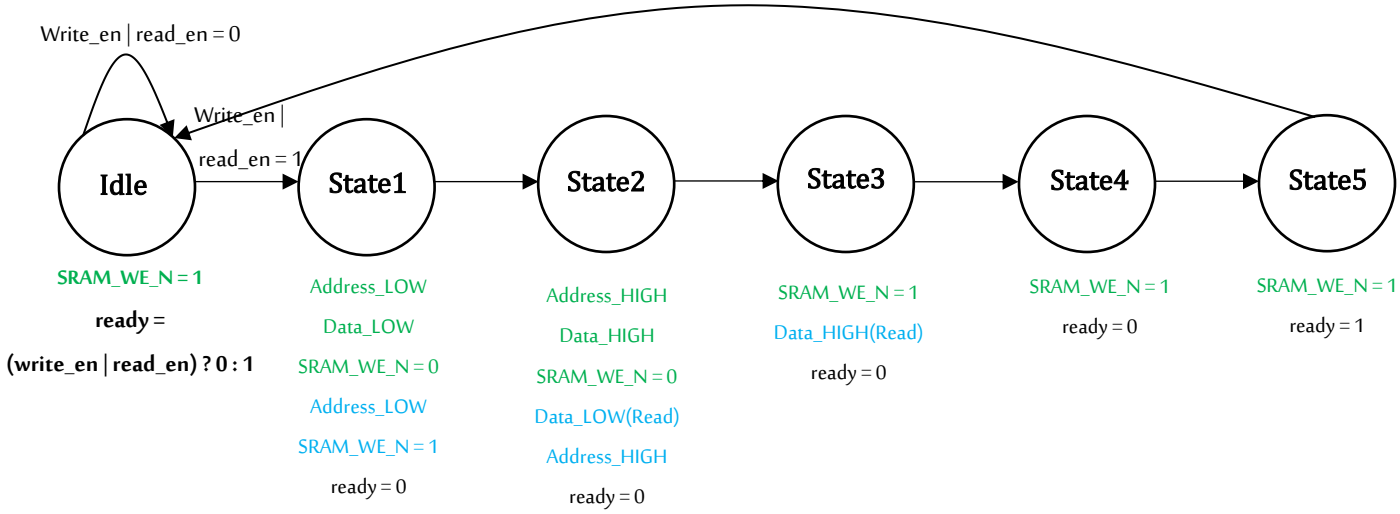
آوا میرمحمدمهدی ۸۱۰۱۹۹۵۰۱ نسا عباسی مقدم ۸۱۰۱۹۹۴۵۷

در جلسه ششم، بخش Memory Data را با حافظه SRAM برد FPGA جایگزین نماییم. از این رو باید برای آن کنترلری طراحی کنیم که بتوانیم با سیگنالهای کنترلی write_en و read_en، خواندن و نوشتن با استفاده از آدرس روی BUS را انجام بدهد. استفاده از SRAM مشکل کمبود فضای حافظه را جبران می‌کند اما زمان دسترسی به داده هارا افزایش می‌دهد.



State Diagram کنترلر SRAM به شکل زیر است:



در State Diagram ترسیم شده، عملیات خواندن در ۳ سیکل و عملیات نوشتن در ۲ سیکل انجام شده اما برای شبیه تر شدن به حالت واقعی، زمان را برابر با ۶ کلاک در نظر میگیریم. همچنین به دلیل اینکه داده ها ۳۲ بیتی هستند اما BUS، ۱۶ بیتی است، داده ها در دو سیکل انتقال می‌یابند. State Diagram بالا با دستورکار و ویدیوی آموزشی مطابقت دارد اما از نظر عملی قابل پیاده سازی نیست و نیاز است که هنگام انجام دستور خواندن، در هنگامی که آدرس لود میشود، در همان هنگام داده نیز خوانده شود. بنابراین در کد نوشته شده، عملیات خواندن و نوشتن هر دو در دو سیکل انجام شده است. باید توجه داشت که سیگنال ها active low هستند و یک به معنای غیرفعال بودن و صفر به معنای فعال بودن آنها است. همانطور که در گزارش ذکر شده است، تمامی سیگنال‌های SRAM_UB_N، SRAM_LB_N، SRAM_CE_N و SRAM_OE_N را با مقدار ۰ مقداردهی کردیم و آنها را فعال در نظر گرفتیم. برای بدست آوردن SRAM_ADDR باید آدرس ورودی را منهای ۱۰۲۴ می‌کنیم. هنگامی که می خواهیم از حافظه بخوانیم bus باید خالی شود تا حافظه بتواند در آن بنویسد. به این منظور هنگام خواندن، bus را High Impedance(Z) قرار می‌دهیم. حال به بررسی استیت ها می‌پردازیم. در استیت Idle، سیگنال ready را تعیین می‌کنیم. در State1، در ابتدا آدرس را شیفت داده و سپس با توجه به اینکه دستور write و یا read باشد، سیگنال هارا تعیین می‌کنیم. بصورتیکه اگر دستور نوشتن باشد، سیگنال SRAM_WE_N را برابر صفر(فعال) قرار داده و ۱۶ بیت اول را روی bus می‌گذاریم و در صورتیکه دستور خواندن باشد، سیگنال SRAM_WE_N را برابر یک(غیرفعال) قرار داده و ۱۶ بیت اول داده روی bus را می‌خوانیم. در State2 باید ۱۶ بیت پر ارزش را روی bus گذاشته و یا از آن بخوانیم. در دو استیت بعدی، سیگنال SRAM_WE_N را برابر یک و سیگنال ready را برابر صفر قرار می‌دهیم و در استیت آخر مقدار سیگنال ready را برابر یک قرار می‌دهیم.

SRAM Controller:

کد کنترلر SRAM به صورت زیر است:

```
module SRAM_Controller(clk, rst, write_en, read_en, address, write_data, read_data, ready, SRAM_DQ, SRAM_ADDR, SRAM_UB_N, SRAM_LB_N,
                       SRAM_WE_N, SRAM_CE_N, SRAM_OE_N);
    input clk, rst;
    input write_en, read_en;
    input [31:0] address;
    input [31:0] write_data;
    output reg [31:0] read_data;
    output reg ready;

    inout [15:0] SRAM_DQ;
    output reg[17:0] SRAM_ADDR;
    output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N;
    output reg SRAM_WE_N;
    assign SRAM_UB_N = 1'b0;
    assign SRAM_LB_N = 1'b0;
    assign SRAM_CE_N = 1'b0;
    assign SRAM_OE_N = 1'b0;

    wire [31:0] mem_address;
    assign mem_address = address[17:0] - 18'd1024;
    wire [15:0] DataBusRead;
    reg [15:0] data_to_write;

    assign SRAM_DQ = (write_en) ? data_to_write : 16'bz;
    assign DataBusRead = SRAM_DQ;
    reg [2:0] ps, ns;

    parameter [3:0] Idle = 0, State1 = 1, State2 = 2, State3 = 3, State4 = 4, State5 = 5;

    always @(*) begin
        {SRAM_WE_N, ready} = 2'b10;
        case(ps)
            Idle: begin
                ready = (write_en|read_en) ? 1'b0 : 1'b1;
            end

            State1: begin
                SRAM_ADDR = mem_address >> 1;
                if (write_en) begin
                    SRAM_WE_N = 1'b0;
                    data_to_write = write_data[15:0];
                end
                else begin
                    read_data[15:0] <= DataBusRead;
                end
            end

            State2: begin
                SRAM_ADDR = (mem_address + 2) >> 1;
                if (write_en) begin
                    SRAM_WE_N = 1'b0;
                    data_to_write = write_data[31:16];
                end
                else begin
                    read_data[31:16] <= DataBusRead;
                end
            end

            State3: begin
                if (write_en) begin
                    SRAM_WE_N = 1'b1;
                end
            end

            State4: begin
            end

            State5: begin
                ready = 1'b1;
            end
        endcase
    end
```

```
    always @(*)begin
        case(ps)
            Idle: begin
                ns = (write_en|read_en) ? State1 : Idle;
            end

            State1: begin
                ns = State2;
            end

            State2: begin
                ns = State3;
            end

            State3: begin
                ns = State4;
            end

            State4: begin
                ns = State5;
            end

            State5: begin
                ns = Idle;
            end
        endcase
    end

    always @(posedge clk) begin
        if (rst) begin
            ps <= Idle;
        end
        else begin
            ps <= ns;
        end
    end

endmodule
```

Mem_Stage:

کد قسمت Mem_Stage به صورت زیر تغییر می‌کند:

```
module Mem_Stage (clk, rst, dst, ALU_res, val_Rm, mem_read, mem_write, WB_en, dst_out, ALU_res_out, mem_out, mem_read_out, WB_en_out,
                  ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_ADDR, SRAM_DQ);
    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] val_Rm;
    input mem_read, mem_write, WB_en;

    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH-1:0] ALU_res_out;
    output [`WORD_WIDTH-1:0] mem_out;
    output mem_read_out, WB_en_out;
    output ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N;
    output [17:0] SRAM_ADDR;
    inout [15:0] SRAM_DQ;

    assign dst_out = dst;
    assign mem_read_out = mem_read;
    assign ALU_res_out = ALU_res;
    wire freeze;
    assign freeze = ~ready;
    SRAM_Controller sram_controller(.clk(clk), .rst(rst), .write_en(mem_write), .read_en(mem_read), .address(ALU_res), .write_data(val_Rm),
                                    .read_data(mem_out), .ready(ready), .SRAM_DQ(SRAM_DQ), .SRAM_ADDR(SRAM_ADDR), .SRAM_UB_N(SRAM_UB_N), .SRAM_LB_N(SRAM_LB_N),
                                    .SRAM_WE_N(SRAM_WE_N), .SRAM_CE_N(SRAM_CE_N), .SRAM_OE_N(SRAM_OE_N));
    Mux2To1 #(1) MUX_2_to_1_WB_EN (.out(WB_en_out), .in1(WB_en), .in2(1'b0), .sel(freeze));

endmodule
```

در این قسمت یک مولتی پلکسر ۱ بیتی دو به یک تعریف می‌شود که تعیین‌کننده‌ی مقدار خروجی WB_en است، درصورتی که ready برابر با ۰ باشد یعنی freeze برابر با ۱ است و WB_en برابر با ۰ می‌شود تا در رجیستر فایل نوشته نشود چون مقدار خروجی حافظه هنوز مشخص نیست و درصورتی که ready برابر با ۱ باشد یعنی freeze برابر با ۰ است و WB_en ورودی را برابر WB_en خروجی قرار می‌دهیم. در واقع freeze برابر ready~ است. در این مرحله سیگنال‌های کنترلی SRAM و SRAM_ADDR را به عنوان خروجی Mem_Stage و SRAM_DQ را به عنوان inout می‌دهیم.

ARM:

با ایجاد تغییرات ماژول arm به صورت زیر در می‌آید:

```
module ARM(clk, rst, forward_en, SRAM_ADDR, SRAM_DQ, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N);
    input clk, rst, forward_en;
    output SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N;
    output [17:0] SRAM_ADDR;
    inout [15:0] SRAM_DQ;

    wire ready, hazard, freeze, branch_taken;
    //IF Stage
    wire  [`WORD_WIDTH-1:0] branchAddr, IFStagePcOut, IFRegPcOut, IFStageInstructionOut, IFRegInstructionOut;

    //ID Stage
    wire  [`WORD_WIDTH-1:0] ID_pc_out, ID_val_Rn, ID_val_Rm;
    wire [`SIGNED_IMM_WIDTH-1:0] ID_signed_immediate;
    wire [`SHIFTER_OPERAND_WIDTH-1:0] ID_shifter_operand;
    wire [`REG_FILE_ADDRESS_LEN-1:0] ID_regFile_src1, ID_regFile_src2, ID_regFile_dst;
    wire [3:0] ID_execute_cmd_out, status_reg_out;
    wire ID_mem_read_out, ID_mem_write_out, ID_WB_en_out, ID_imm_out, ID_B_out, ID_status_update_out;

    //ID Reg
    wire IDreg_mem_read_out, IDreg_mem_write_out, IDreg_WB_en_out, IDreg_imm_out, IDreg_B_out, IDreg_S_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] IDreg_regFile_dst;
    wire [3:0] IDreg_execute_cmd_out, IDreg_status_reg_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] IDreg_regFile_src1, IDreg_regFile_src2;
    wire [`SHIFTER_OPERAND_WIDTH-1:0] IDreg_shifter_operand;
    wire [`SIGNED_IMM_WIDTH-1:0] IDreg_signed_immediate;
    wire [`WORD_WIDTH-1:0] IDreg_pc_out, IDreg_val_Rn, IDreg_val_Rm;


    //EXE Stage
    wire EXE_wb_en_out, EXE_mem_read_out, EXE_mem_write_out;
    wire [`WORD_WIDTH-1:0] EXE_alu_out, EXE_val_Rm_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] EXE_dst_out;

    //EXE Reg
    wire EXEreg_wb_en_out, EXEreg_mem_read_out, EXEreg_mem_write_out;
    wire [`WORD_WIDTH-1:0] EXEreg_alu_out, EXEreg_val_Rm_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] EXEreg_dst_out;

    //MEM Stage
    wire MEM_wb_en_out, MEM_mem_read_out;
    wire [`WORD_WIDTH-1:0] MEM_alu_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] MEM_dst_out;
    wire [`WORD_WIDTH-1:0] mem_out;
```

```
    //MEM Reg
    wire MEMreg_wb_en_out, MEMreg_mem_read_out;
    wire [`WORD_WIDTH-1:0] MEMreg_alu_out;
    wire [`REG_FILE_ADDRESS_LEN-1:0] MEMreg_dst_out;
    wire [`WORD_WIDTH-1:0] MEMreg_mem_out;


    //WB Stage
    wire WB_WB_en_out;
    wire [`WORD_WIDTH-1:0] WB_value;
    wire [`REG_FILE_ADDRESS_LEN-1:0] WB_dst_out;

    //Hazard Detector
    wire two_src;

    //Forwarding Unit
    wire [1 : 0] FWD_sel_src1, FWD_sel_src2;

    IF_Stage IF_stage(.clk(clk), .rst(rst), .freeze(hazard | freeze), .branch_taken(branch_taken), .branchAddr(branchAddr), .pc(IFStagePcOut),
                      .instruction(IFStageInstructionOut));

    IF_Stage_Reg IF_stage_reg(.clk(clk), .rst(rst), .freeze(hazard | freeze), .flush(branch_taken), .pcIn(IFStagePcOut), .instructionIn(IFStageInstructionOut),
                              .pcOut(IFRegPcOut), .instruction(IFRegInstructionOut));

    ID_Stage ID_stage(.clk(clk), .rst(rst), .freeze(hazard), .reg_file_enable(WB_WB_en_out),  .pc_in(IFRegPcOut), .instruction_in(IFRegInstructionOut),
                      .reg_file_result_WB(WB_value), .reg_file_dest_wb(WB_dst_out), .status_register(status_reg_out), .pcOut(ID_pc_out),
                      .val_Rn(ID_val_Rn), .val_Rm(ID_val_Rm), .signed_immediate(ID_signed_immediate), .shifter_operand(ID_shifter_operand),
                      .reg_file_src1(ID_regFile_src1), .reg_file_src2(ID_regFile_src2), .reg_file_dst(ID_regFile_dst),
                      .execute_cmd_out(ID_execute_cmd_out), .mem_read_out(ID_mem_read_out), .mem_write_out(ID_mem_write_out),
                      .WB_en_out(ID_WB_en_out), .Imm_out(ID_imm_out), .B_out(ID_B_out), .status_update_out(ID_status_update_out));

    ID_Stage_Reg ID_stage_reg(.clk(clk), .rst(rst), .flush(branch_taken), .freeze(freeze), .mem_read_in(ID_mem_read_out), .mem_write_in(ID_mem_write_out),
                              .WB_Enable_in(ID_WB_en_out), .Imm_in(ID_imm_out), .B_in(ID_B_out), .status_update_in(ID_status_update_out),
                              .reg_file_dst_in(ID_regFile_dst), .executeCommand_in(ID_execute_cmd_out), .status_register_in(status_reg_out),
                              .reg_file_src1_in(ID_regFile_src1), .reg_file_src2_in(ID_regFile_src2), .shifter_operand_in(ID_shifter_operand),
                              .signed_immediate_in(ID_signed_immediate), .pc_in(ID_pc_out), .val_Rn_in(ID_val_Rn), .val_Rm_in(ID_val_Rm),
                              .mem_read_out(IDreg_mem_read_out), .mem_write_out(IDreg_mem_write_out), .WB_Enable_out(IDreg_WB_en_out),
                              .Imm_out(IDreg_imm_out), .B_out(IDreg_B_out), .status_update_out(IDreg_S_out),
                              .reg_file_dst_out(IDreg_regFile_dst), .execute_cmd_out(IDreg_execute_cmd_out), .status_register_out(IDreg_status_reg_out),
                              .reg_file_src1_out(IDreg_regFile_src1), .reg_file_src2_out(IDreg_regFile_src2), .shifter_operand_out(IDreg_shifter_operand),
                              .signed_immediate_out(IDreg_signed_immediate), .pc_out(IDreg_pc_out),
                              .val_Rn_out(IDreg_val_Rn), .val_Rm_out(IDreg_val_Rm));

    Exec_Stage exec_stage(.clk(clk), .rst(rst), .pc_in(IDreg_pc_out), .mem_read_in(IDreg_mem_read_out), .mem_write_in(IDreg_mem_write_out), .S(IDreg_S_out), .B(IDreg_B_out),
                          .WB_en_in(IDreg_WB_en_out), .execute_cmd_in(IDreg_execute_cmd_out), .immediate_in(IDreg_imm_out), .signed_immediate_24_in(IDreg_signed_immediate),
                          .shift_operand_in(IDreg_shifter_operand), .val_Rn_in(IDreg_val_Rn), .val_Rm_in(IDreg_val_Rm), .SR_in(IDreg_status_reg_out), .dst_in(IDreg_regFile_dst),
                          .alu_res_in_MEM(MEM_alu_out), .wb_value_WB(WB_value), .alu_mux_src_1_sel(FWD_sel_src1), .alu_mux_src_2_sel(FWD_sel_src2),
                          .alu_out(EXE_alu_out), .branch_address(branchAddr), .status_reg_out(status_reg_out), .branch_taken_out(branch_taken), .WB_en_out(EXE_wb_en_out),
                          .mem_read_out(EXE_mem_read_out), .mem_write_out(EXE_mem_write_out), .dst_out(EXE_dst_out), .val_Rm_out(EXE_val_Rm_out));

    Exec_Stage_Reg Exec_stage_reg(.clk(clk), .rst(rst), .freeze(freeze), .dst_in(EXE_dst_out), .mem_read_in(EXE_mem_read_out), .mem_write_in(EXE_mem_write_out), .WB_en_in(EXE_wb_en_out),
                                  .val_Rm_in(EXE_val_Rm_out), .ALU_res_in(EXE_alu_out), .dst_out(EXEreg_dst_out), .ALU_res_out(EXEreg_alu_out), .val_Rm_out(EXEreg_val_Rm_out),
                                  .mem_read_out(EXEreg_mem_read_out), .mem_write_out(EXEreg_mem_write_out), .WB_en_out(EXEreg_wb_en_out));

    Mem_Stage Mem_stage(.clk(clk), .rst(rst), .dst(EXEreg_dst_out), .ALU_res(EXEreg_alu_out), .val_Rm(EXEreg_val_Rm_out), .mem_read(EXEreg_mem_read_out), .mem_write(EXEreg_mem_write_out),
                        .WB_en(EXEreg_wb_en_out), .dst_out(MEM_dst_out), .ALU_res_out(MEM_alu_out), .mem_out(mem_out), .mem_read_out(MEM_mem_read_out), .WB_en_out(MEM_wb_en_out),
                        .ready(ready), .SRAM_UB_N(SRAM_UB_N), .SRAM_LB_N(SRAM_LB_N), .SRAM_WE_N(SRAM_WE_N), .SRAM_CE_N(SRAM_CE_N), .SRAM_OE_N(SRAM_OE_N),
                        .SRAM_ADDR(SRAM_ADDR), .SRAM_DQ(SRAM_DQ));

    assign freeze = ~ready;

    Mem_Stage_Reg Mem_stage_reg(.clk(clk), .rst(rst), .freeze(freeze), .dst(MEM_dst_out), .ALU_res(MEM_alu_out), .mem_data(mem_out), .mem_read(MEM_mem_read_out), .WB_en(MEM_wb_en_out),
                                .dst_out(MEMreg_dst_out), .ALU_res_out(MEMreg_alu_out), .mem_data_out(MEMreg_mem_out), .mem_read_out(MEMreg_mem_read_out), .WB_en_out(MEMreg_wb_en_out));

    WB_Stage wb_stage(.clk(clk), .rst(rst), .dst(MEMreg_dst_out), .ALU_res(MEMreg_alu_out), .mem_data(MEMreg_mem_out), .mem_read(MEMreg_mem_read_out), .WB_en(MEMreg_wb_en_out),
                      .WB_dst(WB_dst_out), .WB_en_out(WB_WB_en_out), .WB_value(WB_value));

    assign two_src = ID_mem_write_out | ~ID_imm_out;
    Hazard_Detector hazard_detector(.src1(ID_regFile_src1), .src2(ID_regFile_src2), .exe_wb_dest(EXE_dst_out), .mem_wb_dest(MEM_dst_out), .two_src(two_src),  .exe_wb_enable(EXE_wb_en_out),
                                    .mem_wb_enable(MEM_wb_en_out), .forward_en(forward_en), .EXE_mem_read_en(IDreg_mem_read_out), .hazard(hazard));

    Forwarding_Unit forwarding_unit(.forward_en(forward_en), .WB_wb_en(WB_WB_en_out), .MEM_wb_en(MEM_wb_en_out), .MEM_dst(MEM_dst_out), .WB_dst(WB_dst_out), .src1_in(IDreg_regFile_src1),
                                    .src2_in(IDreg_regFile_src2), .sel_src1(FWD_sel_src1), .sel_src2(FWD_sel_src2));
endmodule
```

از بیت ready برای متوقف کردن پردازنده در زمان نوشتن یا خواندن از حافظه استفاده می شود. بنابراین، همانطور که می‌بینیم سیگنال freeze را برابر با ready~ قرار می‌دهیم و آن را به تمامی رجیسترها به جز رجیستر بعد از IF می‌دهیم، درصورتی که freeze برابر با ۰ باشد مقدار ورودی رجیستر در خروجی‌اش ریخته می‌شود و درصورتی که برابر با ۱ باشد، کاری انجام نمی‌شود. مقدار freeze رجیستر بعد از IF_Stage و IF_Stage را برابر hazard | freeze قرار می‌دهیم تا درصورتی برابر با ۱ شود که هم freeze و هم hazard برابر با ۱ باشند.

Exec_Stage_Reg:

مقدار freeze در رجیستر بعد از ID همچنان برابر با مقدار hazard است.

```
module Mem_Stage (clk, rst, dst, ALU_res, val_Rm, mem_read, mem_write, WB_en, dst_out, ALU_res_out, mem_out, mem_read_out, WB_en_out,
                  ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_ADDR, SRAM_DQ);
    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] val_Rm;
    input mem_read, mem_write, WB_en;

    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH-1:0] ALU_res_out;
    output [`WORD_WIDTH-1:0] mem_out;
    output mem_read_out, WB_en_out;
    output ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N;
    output [17:0] SRAM_ADDR;
    inout [15:0] SRAM_DQ;

    assign dst_out = dst;
    assign mem_read_out = mem_read;
    assign ALU_res_out = ALU_res;
    wire freeze;
    assign freeze = ~ready;
    SRAM_Controller sram_controller(.clk(clk), .rst(rst), .write_en(mem_write), .read_en(mem_read), .address(ALU_res), .write_data(val_Rm),
                                    .read_data(mem_out), .ready(ready), .SRAM_DQ(SRAM_DQ), .SRAM_ADDR(SRAM_ADDR), .SRAM_UB_N(SRAM_UB_N), .SRAM_LB_N(SRAM_LB_N),
                                    .SRAM_WE_N(SRAM_WE_N), .SRAM_CE_N(SRAM_CE_N), .SRAM_OE_N(SRAM_OE_N));
    SRAM sram(clk, rst, SRAM_WE_N, SRAM_ADDR, SRAM_DQ);
    Mux2To1 #(1) MUX_2_to_1_WB_EN (.out(WB_en_out), .in1(WB_en), .in2(1'b0), .sel(freeze));

endmodule
```

SRAM:

برای تست پردازنده در ModelSim یک ماژول SRAM طراحی می کنیم.

```
`timescale 1ns/1ns
module SRAM(
    input clk, rst,
    input SRAM_WE_N,
    input [17:0] SRAM_ADDR,
    inout [15:0] SRAM_DQ
);
    reg [15:0] memory [0:511];
    assign #1 SRAM_DQ = (SRAM_WE_N == 1'b1) ? memory[SRAM_ADDR] : 16'dz;

    always @(negedge clk) begin
        if (SRAM_WE_N == 1'b0) begin
            memory[SRAM_ADDR] = SRAM_DQ;
        end
    end
endmodule
```

Freqeuncy Divider:

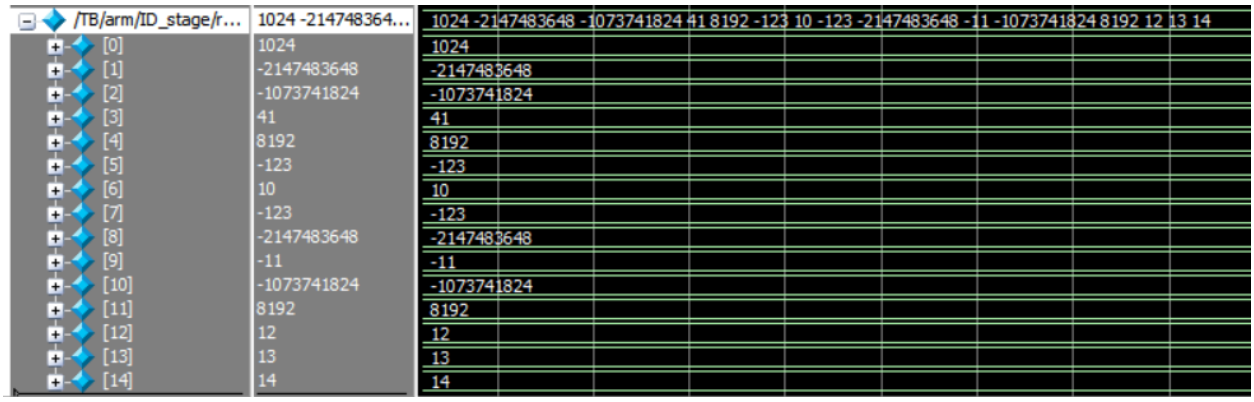
بعد از اجرای کد در حالت فعال بودن forwarding، مشاهده شد که نتایج درست نیست و با بررسی روش های مختلف در نهایت به این نتیجه رسیدیم که دلیل این مشکل، سرعت بالای کلاک است، بنابراین با نوشتن کد زیر، مشکل را حل کردیم و در ماژول toplevel(Mips) از آن instance گرفتیم.

```
module FreqDiv #(parameter Bits = 1)(input clk, rst, en,output co);
    reg [Bits-1:0] q;

    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= {Bits{1'b0}};
        else if (en) begin
            if (co)
                q <= {Bits{1'b0}};
            else
                q <= q + 1;
        end
    end

    assign co = &q;
endmodule
```

نمایش خروجی waveform در modelsim:

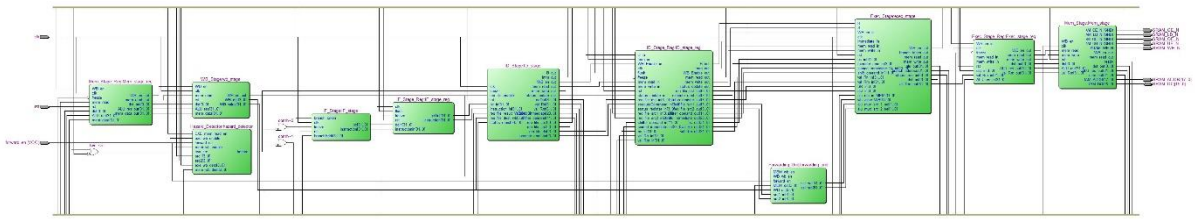


خروجی برنامه Quartus:

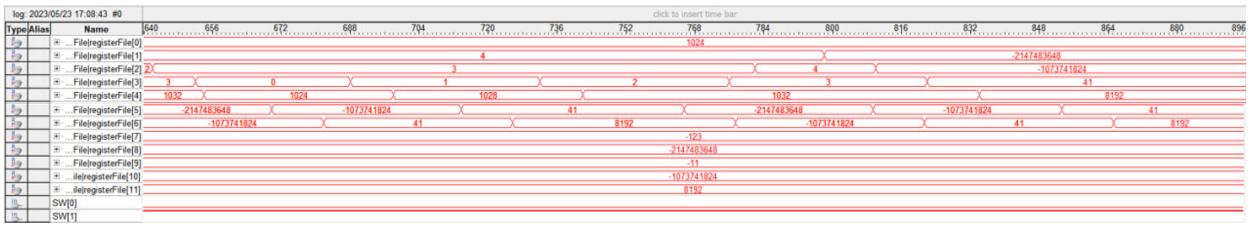
خروجی Flow Summary:

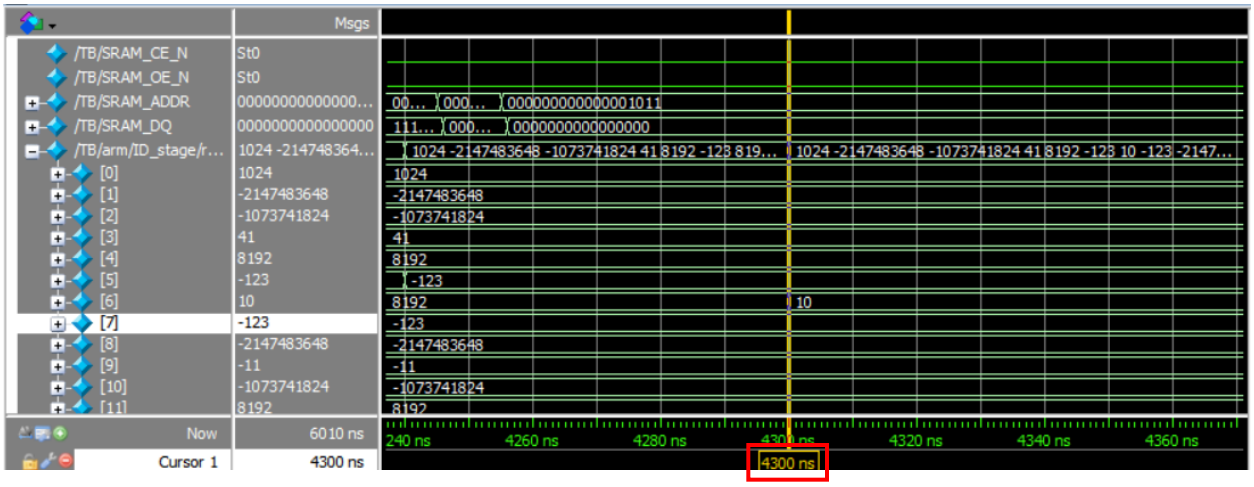
Flow Summary	
Flow Status	Successful - Tue May 23 17:18:21 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	MIPS
Top-level Entity Name	MIPS
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	5,177 / 33,216 (16 %)
Total combinational functions	3,537 / 33,216 (11 %)
Dedicated logic registers	3,154 / 33,216 (9 %)
Total registers	3154
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	165,888 / 483,840 (34 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

نمایش RTL:



خروجی Signal Tab:





میزان افت کارایی:

$$performance1 = \frac{1}{Execution\ time1} = \frac{1}{195}$$
$$performance2 = \frac{1}{Excuarion\ time2} = \frac{1}{430}$$

$$speedUp = \frac{p2}{p1} = \frac{195}{430} \approx 0.45 \rightarrow 45\%$$

میزان هزینه سخت‌افزاری:

$$\frac{Logic\ Elements2}{Logic\ Elements1} = \frac{5{,}177}{7{,}928} \approx 0.65 \rightarrow 35\%$$

میزان کارایی بر هزینه:

$$Cost\ per\ performance1 = \frac{7{,}928}{\frac{1}{195}}$$
$$Cost\ per\ performance2 = \frac{5{,}177}{\frac{1}{430}}$$

$$\frac{5{,}177 \times 430}{7{,}928 \times 195} \approx 1.44 \rightarrow 44\%$$

با توجه به محاسبات بالا نتیجه می‌گیریم که با اضافه شدن SRAM، هزینه سخت‌افزاری کاهش می‌یابد اما کارایی نیز افت کرده است. نسبت هزینه به میزان کارایی نشان میدهد که افزودن SRAM باعث افت عملکرد سیستم به نسبت هزینه شده که این نتیجه مطابق با انتظار اما برخلاف خواسته ما است.

روش های پیشنهادی برای بهبود عملکرد سیستم:

Random Replacement

Last Recently used

Most Recently used

Least Frequently used

Least Frequent recently used

Processor’s Registers