
Exec_Stage:

با افزودن بخش Forwarding کد بخش execution به صورت زیر تغییر کرده است:

```
module Exec_Stage(clk, rst, pc_in, mem_read_in, mem_write_in, S, B, WB_en_in, execute_cmd_in, immediate_in, signed_immediate_24_in, shift_operand_in,
                  val_Rn_in, val_Rm_in, SR_in, dst_in, alu_res_in_MEM, wb_value_WB, alu_mux_src_1_sel, alu_mux_src_2_sel, alu_out, branch_address,
                  status_reg_out, branch_taken_out, WB_en_out, mem_read_out, mem_write_out, dst_out, val_Rm_out);

    input clk, rst;
    input [`WORD_WIDTH-1: 0] pc_in;
    input mem_read_in, mem_write_in, S, B, WB_en_in;
    input [3:0] execute_cmd_in;
    input immediate_in;
    input [`SIGNED_IMM_WIDTH-1:0] signed_immediate_24_in;
    input [`SHIFTER_OPERAND_WIDTH - 1 : 0] shift_operand_in;
    input [`WORD_WIDTH - 1: 0] val_Rn_in, val_Rm_in;
    input [3:0] SR_in;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst_in;
    input [`WORD_WIDTH - 1 : 0] alu_res_in_MEM, wb_value_WB;
    input [1:0] alu_mux_src_1_sel;
    input [1:0] alu_mux_src_2_sel;

    output [`WORD_WIDTH - 1 : 0] alu_out;
    output [`WORD_WIDTH - 1 : 0] branch_address;
    output [3:0] status_reg_out;
    output branch_taken_out, WB_en_out, mem_read_out, mem_write_out;
    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH - 1 : 0] val_Rm_out;


    wire [3:0] status_bits;
    wire [`WORD_WIDTH - 1 : 0] val2out;
    wire [`WORD_WIDTH - 1 : 0] alu_mux_Rn_out, alu_mux_Rm_out;

    ALU alu(.val1(alu_mux_Rn_out), .val2(val2out), .cin(SR_in[2]), .EX_command(execute_cmd_in), .ALU_out(alu_out), .SR(status_bits));
    Mux3To1 #(`WORD_WIDTH) alu_mux_src_1(.in1(val_Rn_in), .in2(alu_res_in_MEM), .in3(wb_value_WB), .sel(alu_mux_src_1_sel), .out(alu_mux_Rn_out));
    Mux3To1 #(`WORD_WIDTH) alu_mux_src_2(.in1(val_Rm_in), .in2(alu_res_in_MEM), .in3(wb_value_WB), .sel(alu_mux_src_2_sel), .out(alu_mux_Rm_out));



    wire is_mem_cmd;
    assign is_mem_cmd = mem_read_in | mem_write_in;

    assign dst_out = dst_in;
    assign branch_taken_out = B;
    assign mem_read_out = mem_read_in;
    assign mem_write_out = mem_write_in;
    assign WB_en_out = WB_en_in;

    Status_Register status_ref(.clk(clk), .rst(rst), .ld(S), .in(status_bits), .out(status_reg_out));

    assign val_Rm_out = alu_mux_Rm_out;
    Val2Generator v2g(.val_Rm(alu_mux_Rm_out), .shift_operand(shift_operand_in), .immediate(immediate_in), .is_mem_cmd(is_mem_cmd), .val2_out(val2out));

    wire [`WORD_WIDTH - 1 : 0] sign_immediate_extended = { {8{signed_immediate_24_in[23]}}, signed_immediate_24_in};
    wire [`WORD_WIDTH - 1 : 0] sign_immediate_extended_xfour = sign_immediate_extended << 2;
    Adder adderPc(.a(pc_in), .b(sign_immediate_extended_xfour), .res(branch_address));

endmodule
```

همانطور که در کد مشخص است در این بخش دو مولتی پلکسر اضافه کردیم به طوری که alu_mux_src_1 تعیین کننده‌ی ورودی اول ALU است که بیت سه وردی انتخاب می‌کند؛ این ورودی‌ها val_Rn خارج شده از رجیستر بخش ID، نتیجه‌ی ALU که در بخش Memory stage قرار دارد و یا WB_value که از مولتی پلکسر بخش Write Back Stage هستند؛ در حالتی که وابستگی داده‌ای بین یک دستور با دستور قبلی‌اش یا یک دستور با دو دستور قبلش وجود دارد، در واقع داده‌ای که دستور فعلی می‌خواهد از آن استفاده کند آماده است ولی مثلا هنوز در رجیستر مربوطه نوشته نشده است و ما به جای اینکه آن داده را از رجیسترفایل بخوانیم، آن را از یک استیج یا دو استیج بعدی به مرحله‌ی execution فوروارد می‌کنیم تا بتوانیم از مقدار آن استفاده کنیم؛ مثلا وقتی دستور قبلی به صورت

$R_1 = R_2 + 3$ است و دستور فعلی $R_4 = R_1 + 2$ حاصل $R_2 + 3$ را می‌دانیم فقط هنوز آن را در R_1 نریخته‌ایم و می‌توانیم از این حاصل استفاده کنیم؛ یا وقتی در دو دستور قبلی مقداری از حافظه را خوانده‌ایم و قرار است آن را در رجیستر R_1 بنویسیم قبل از اینکه آن را بنویسیم، مقدار خوانده شده از حافظه را می‌دانیم و می‌توانیم از آن استفاده کنیم.

مولتی پلکسر با نام alu_mux_src_2 نیز برای تعیین ورودی val2Generator وجود دارد و ورودی‌های آن val_Rm خارج شده از رجیستر بخش ID، نتیجه‌ی ALU که در بخش Memory stage قرار دارد و یا WB_value که از مولتی پلکسر بخش Write Back Stage هستند و بقیه‌ی توضیحات مااند مولتی پلکسر قبلی است.

به طور کلی با اضافه شدن این بخش، سیگنال ورودی forward_en را به پردازنده‌ی arm اضافه کردیم که ۱ شدن آن نشانه‌ی این است که از forwarding استفاده می‌کنیم و ۰ بودن آن برای عدم استفاده از این آپشن است. این سیگنال همچنین به واحد HazardDetector نیز به عنوان ورودی داده می‌شود و تغییرات زیر در این واحد انجام می‌شود:

Hazard_Detector:

```
module Hazard_Detector (src1, src2, exe_wb_dest, mem_wb_dest, two_src,  exe_wb_enable, mem_wb_enable, forward_en, EXE_mem_read_en, hazard);

    input [`REG_FILE_ADDRESS_LEN - 1:0] src1, src2;
    input [`REG_FILE_ADDRESS_LEN - 1:0] exe_wb_dest;
    input [`REG_FILE_ADDRESS_LEN - 1:0] mem_wb_dest;
    input two_src, exe_wb_enable, mem_wb_enable;
    input forward_en, EXE_mem_read_en;

    output reg hazard;
    always @(*)
    begin
        if (~forward_en) begin
            if ((src1 == exe_wb_dest) && (exe_wb_enable == 1'b1))
                hazard = 1'b1;
            else
            if ((src1 == mem_wb_dest) && (mem_wb_enable == 1'b1))
                hazard = 1'b1;
            else
            if ((src2 == exe_wb_dest) && (exe_wb_enable == 1'b1) && (two_src == 1'b1))
                hazard = 1'b1;
            else
            if ((src2 == mem_wb_dest) && (mem_wb_enable == 1'b1) && (two_src == 1'b1))
                hazard = 1'b1;
            else
                hazard = 1'b0;
        end

        else  begin
            if ((src1 == exe_wb_dest) && EXE_mem_read_en)
                hazard = 1'b1;
            else
            if ((src2 == exe_wb_dest) && (two_src == 1'b1) && EXE_mem_read_en)
                hazard = 1'b1;
            else
                hazard = 1'b0;
        end
    end

endmodule
```

همانطور که در کد دیده می‌شود درحالتی که forward_en برابر با ۰ باشد شروط واحد HazardDetector مانند قبل است و درحالتی که برابر با ۱ باشد نیز در حالتی که mem_read_en در بخش exe برابر با ۱ باشد (یعنی بخواهیم مقداری از حافظه را بخوانیم) و همچنین src1 با مقصد exe_wb یکی باشد یا اینکه در دستور از src2 استفاده شود (یعنی two_src برابر با ۱ باشد و دستور از هر دو مقدار خوانده شده از رجیسترفایل استفاده کند) و همچنین مقدار src2 برابر با مقصد exe_wb باشد، hazard را برابر ۱ قرار می‌دهیم.

لازم به ذکر است که اگر آپشن عدم وجود forwarding حذف می‌شد، نیازی نبود تا ورودی‌های mem_wb_dest، exe_wb_enable و mem_wb_enale به عنوان ورودی‌به واحد HazardDetector داده شود، چون در این شرایط فورواردینگ انجام می‌شود و نیازی به فعال کردن hazard نیست.

Forwarding_Unit:

همانطور که در قبل اشاره شد در Forwarding_Unit مقدار select برای دو مولتی پلکسر بخش execution تعیین می‌شود، مقدار پیش‌فرض برای آن برابر با ۰۰ است که درصورتی اتفاق می‌افتد که forwarding وجود نداشته باشد؛ درصورتی که forwarding_en و wb_en مرحله write back فعال باشند، اگر مقصد wb برابر با src1 بود، select مولتی پلکسر اول برابر با 2’b10 می‌شود و درصورتی که برابر با src2 بود، select مولتی پلکسر دوم برابر با 2’b10 می‌شود؛ به طور مشابه درصورتی که forwarding_en و wb_en مرحله memory فعال باشند، اگر مقصد

```
module Forwarding_Unit (forward_en, WB_wb_en, MEM_wb_en, MEM_dst, WB_dst, src1_in, src2_in, sel_src1, sel_src2);
    input forward_en;
    input WB_wb_en, MEM_wb_en;
    input [`REGFILE_ADDRESS_LEN - 1:0] MEM_dst, WB_dst, src1_in, src2_in;

    output reg [1:0] sel_src1, sel_src2;

    always@(*) begin
        {sel_src1, sel_src2} = 4'd0;
        if (forward_en && WB_wb_en) begin
            if (WB_dst == src1_in) begin
                sel_src1 = `FORW_SEL_FROM_WB;
            end

            if (WB_dst == src2_in) begin
                sel_src2 = `FORW_SEL_FROM_WB;
            end
        end
        if (forward_en && MEM_wb_en) begin
            if (MEM_dst == src1_in) begin
                sel_src1 = `FORW_SEL_FROM_MEM;
            end

            if (MEM_dst == src2_in) begin
                sel_src2 = `FORW_SEL_FROM_MEM;
            end
        end
    end
endmodule
```

wb در Memory_Stage برابر با src1 بود، select مولتی پلکسر اول برابر با 2'b01 می‌شود و درصورتی که برابر با src2 بود، select مولتی پلکسر دوم برابر با 2'b01 می‌شود.

در ورودی‌ها و خروجی‌های برخی ماژول‌ها نیز تغییراتی ایجاد کردیم که در ادامه به توضیح آنها می‌پردازیم:

```
module ARM(clk, rst, forward_en);
    input clk, rst, forward_en;
    wire 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(freeze), .branch_taken(branch_taken), .branchAddr(branchAddr), .pc(IFStagePcOut),
                      .instruction(IFStageInstructionOut));

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

    ID_Stage ID_stage(.clk(clk), .rst(rst), .freeze(freeze), .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), .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), .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));

    Mem_Stage_Reg Mem_stage_reg(.clk(clk), .rst(rst), .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(freeze));

    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
```

src1 و src2 به عنوان خروجی در مرحله‌ی ID_Stage اضافه می‌شوند و همچنین در رجیسر بعد از ID نیز ذخیره می‌شوند و خروجی این رجیستر برای این مقادیر به عنوان ورودی src1 و src2 به واحد Forwarding داده می‌شود؛ همچنین خروجی ALU که در مرحله‌ی Memory_Stage قرار دارد را به عنوان ورودی HazardDetection می‌دهیم و دو خروجی واحد Forwarding که select دو مولتی پلکسر بخش execution را مشخص می‌کنند نیز به عنوان wire دو بیتی مشخص شده‌اند.

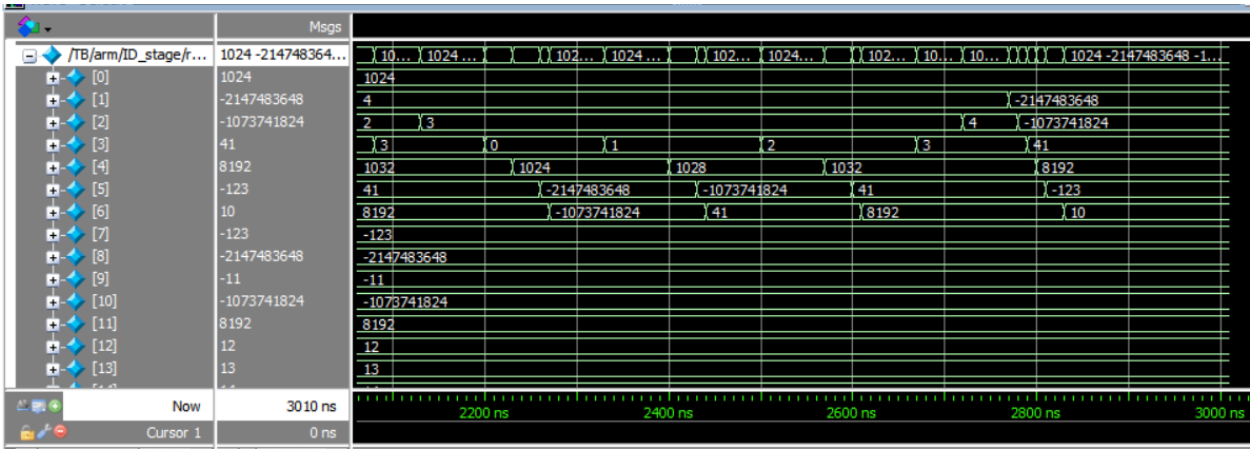
Test Bench:

```
module TB();
    reg clk = 1'b0, rst = 1'b0, forward_en = 1'b0;
    ARM arm(clk, rst, forward_en);
    always #5 clk = ~clk;
    initial begin
        rst = 1'b1;
        #10 rst = 1'b0;
       #3000 $stop;
    end
endmodule
```

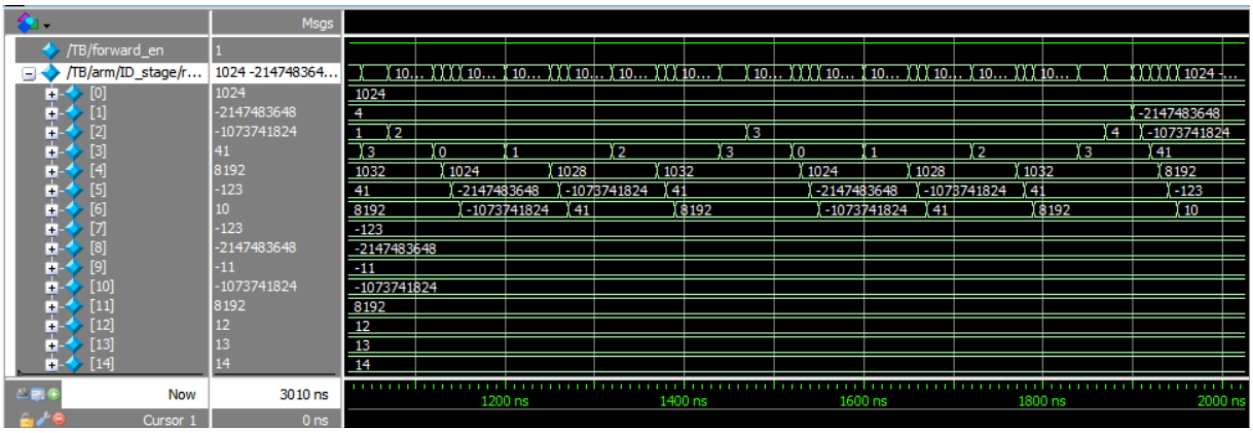
سیگنال forward_en مشخص می‌کند که از روش forwarding (ارسال رو به جلو) استفاده می‌کنیم یا خیر. این سیگنال را به سیگنال‌های ورودی پردازنده اضافه می‌کنیم و سیگنال‌های ورودی برابر rst, clk, forward_en خواهد شد.

دو خروجی waveform را به ازای فعال بودن و نبودن سیگنال forward_en بررسی می‌کنم:

حالت forward_en = 0:



حالت forward_en = 1:



همانطور که مشخص است، خروجی های در هر دو حالت صحیح است که عملکرد صحیح پردازنده را نشان می‌دهد. همچنین باید توجه کرد که در حالت دوم و با استفاده از forwarding سریعتر به انتهای برنامه می‌رسیم که نشان دهنده افزایش کارایی سیستم با اضافه شدن این حالت است.

نتایج حاصل از پیاده سازی در Quartus را در هر دو حالت بررسی می‌کنیم:

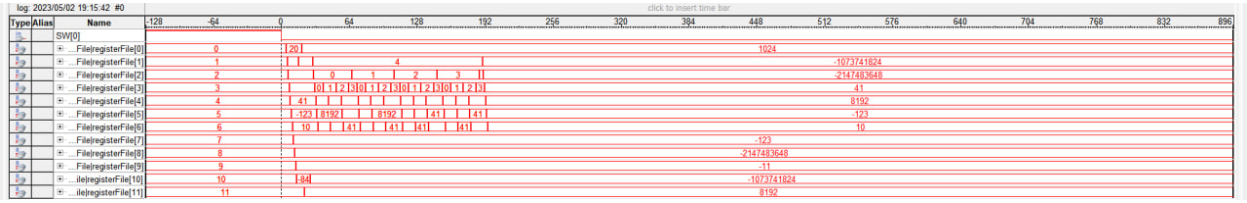
Flow Summary	
Flow Status	Successful - Thu May 04 23:28:17 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	MIPS
Top-level Entity Name	MIPS
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,675 / 33,216 (23 %)
Total combinational functions	3,985 / 33,216 (12 %)
Dedicated logic registers	5,853 / 33,216 (18 %)
Total registers	5853
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	396,288 / 483,840 (82 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Flow Summary	
Flow Status	Successful - Fri May 05 02:06:56 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	7,928 / 33,216 (24 %)
Total combinational functions	4,272 / 33,216 (13 %)
Dedicated logic registers	5,857 / 33,216 (18 %)
Total registers	5857
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	396,288 / 483,840 (82 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

همانطور که مشخص است، با اضافه شدن قسمت forwarding، تعداد المان های منطقی نیز افزایش یافته است.

حالت forward_en = 0:

حالت forward_en = 1:



میزان افزایش کارایی:

با توجه به دو تصویر بالا زمانیکه پردازنده در حالت forward_en = 0 تأخیر در حدود 284 کلاک است. همچنین وقتی پردازنده در در حالت forward_en = 1 تأخیر در حدود 195 کلاک است. در نتیجه میزان بهبود کارایی با تقریب به صورت زیر بدست می آید:

$$performance1 = \frac{1}{Execution\ time1} = \frac{1}{284}$$
$$performance2 = \frac{1}{Excuarion\ time2} = \frac{1}{195}$$

$$speedUp = \frac{p1}{p2} = \frac{284}{195} \approx 1.45 \rightarrow 45\%$$

میزان هزینه سخت‌افزاری:

$$\frac{Logic\ Elements2}{Logic\ Elements1} = \frac{7{,}928}{7{,}675} \approx 1.032 \rightarrow 3.2\%$$

میزان کارایی بر هزینه:

$$Cost\ per\ performance1 = \frac{7{,}928}{\frac{1}{284}}$$
$$Cost\ per\ performance2 = \frac{7{,}675}{\frac{1}{195}}$$

$$\frac{7{,}675 \times 195}{7{,}928 \times 284} \approx 0.66 \rightarrow 66\%$$

با توجه به محاسبات بالا نتیجه می‌گیریم که با اضافه شدن forwarding unit، در کنار افزایش هزینه، بهبود کارایی نیز داشتیم. نسبت هزینه اضافه شده به افزایش کارایی نشان میدهد که افزودن forwarding unit مجموعا باعث بهبود عملکرد سیستم شده است.

بخش امتیازی:

بغیر از forwarding، برای جلوگیری از رخ دادن خطا ناشی از وابستگی داده، روش نرمافزاری یا Code Reordering نیز وجود دارد. این روش به اینصورت عمل می‌کند که بدون ایجاد تغییر در سخت افزار و فقط با جابجایی خط‌های

کد برنامه، سعی می‌کنیم بدون تغییر در ماهیت عملکرد، دستورات دارای وابستگی داده‌ای را با فاصله بیشتر از هم قرار دهیم و به اینصورت از ایجاد خطا جلوگیری کنیم.