EECS 112L Lab #4

Pipelined MIPS Processor

Steven Tabilisma
16326339

11/23/22

Objective

Having completed the implementation of new commands for the MIPS32 processor, we are now implementing the feature of pipelining.

Procedure

The procedure of this lab was very long and tedious as it required a good bit of complex wiring in order to properly implement the 5 stage pipelining process. Of the 16 files that we were provided, only 5 actually required real modifications. This does not include any file replacements that the lab guide required us to do. The code for these 5 files is below.

mips_32.v

```
module mips_32(
   input clk, reset,
   output[31:0] result
   );

// define all the wires here. You need to define more wires than the ones you did in Lab2
   wire [9:0]pc_plus4_wire_if;
   wire [31:0]instr_wire_if;

   wire [9:0]if_id_pc_plus4_wire; //check width

   wire [31:0]if_id_instr_wire; //check width

   wire [31:0] reg1_wire_id;
   wire [31:0] reg2_wire_id;
   wire [31:0] imm_value_wire_id;
   wire [9:0] branch_address_wire_id;
   wire [9:0] jump_address_wire_id;
   wire branch_taken_wire_id;
   wire [4:0] destination_reg_wire_id;
   wire mem_to_reg_wire_id;
   wire [1:0] alu_op_wire_id;
   wire mem_read_wire_id;
```

```
    wire mem_write_wire_id;
    wire alu_src_wire_id;
    wire reg_write_wire_id;
    wire jump_wire_id;

    wire [31:0]id_ex_instr_wire;
    wire [31:0]id_ex_reg1_wire;
    wire [31:0]id_ex_reg2_wire;
    wire [31:0]id_ex_imm_value_wire;
    wire [4:0]id_ex_destination_reg_wire;
    wire id_ex_mem_to_reg_wire;
    wire [1:0]id_ex_alu_op_wire;
    wire id_ex_mem_read_wire;
    wire id_ex_mem_write_wire;
    wire id_ex_alu_src_wire;
    wire id_ex_reg_write_wire;

    wire Data_Hazard_wire;
    wire IF_Flush_wire;

    wire [31:0] alu_in2_out_wire_ex;
    wire [31:0] alu_result_wire_ex;

    wire [1:0] Forward_A_wire_fu;
    wire [1:0] Forward_B_wire_fu;

    wire [31:0]ex_mem_instr_wire;
    wire [4:0]ex_mem_destination_reg_wire;
    wire [31:0]ex_mem_alu_result_wire;
    wire [31:0]ex_mem_alu_in2_out_wire;
    wire ex_mem_mem_to_reg_wire;
    wire ex_mem_mem_read_wire;
    wire ex_mem_mem_write_wire;
    wire ex_mem_reg_write_wire;

    wire [31:0]mem_read_data_wire;

    wire [31:0]mem_wb_alu_result_wire;
    wire [31:0]mem_wb_mem_read_data_wire;
    wire mem_wb_mem_to_reg_wire;
    wire mem_wb_reg_write_wire;
    wire [4:0]mem_wb_destination_reg_wire;

    wire [31:0]write_back_data_wire;

// Build the pipeline as indicated in the lab manual

///////////////////////////// Instruction Fetch
    IF_pipe_stage IF_pipe_stage_inst(
        .clk(clk),
        .reset(reset),
        .en(Data_Hazard_wire),
        .branch_address(branch_address_wire_id),
        .jump_address(jump_address_wire_id),
        .branch_taken(branch_taken_wire_id),
        .jump(jump_wire_id),

        .pc_plus4(pc_plus4_wire_if),
        .instr(instr_wire_if)
    );


/////////////////////////////// IF/ID registers
    pipe_reg_en #(.WIDTH(10)) pipe_reg_en_if_id_pc_plus4(//check width
        .clk(clk),
        .reset(reset),
        .en(Data_Hazard_wire),
        .flush(IF_Flush_wire),
        .d(pc_plus4_wire_if),

        .q(if_id_pc_plus4_wire)
    );


    pipe_reg_en #(.WIDTH(32)) pipe_reg_en_if_id_instr(//check width
        .clk(clk),
        .reset(reset),
        .en(Data_Hazard_wire),
        .flush(IF_Flush_wire),
        .d(instr_wire_if),

        .q(if_id_instr_wire)
    );


/////////////////////////////// Instruction Decode
    ID_pipe_stage ID_pipe_stage_inst(
        .clk(clk),
        .reset(reset),
        .pc_plus4(if_id_pc_plus4_wire),
        .instr(if_id_instr_wire),
        .mem_wb_reg_write(mem_wb_reg_write_wire),
        .mem_wb_write_reg_addr(mem_wb_destination_reg_wire),
        .mem_wb_write_back_data(write_back_data_wire),
        .Data_Hazard(Data_Hazard_wire),
        .Control_Hazard(IF_Flush_wire),

        .reg1(reg1_wire_id),
        .reg2(reg2_wire_id),
        .imm_value(imm_value_wire_id),
        .branch_address(branch_address_wire_id),
        .jump_address(jump_address_wire_id),
        .branch_taken(branch_taken_wire_id),
        .destination_reg(destination_reg_wire_id),
        .mem_to_reg(mem_to_reg_wire_id),
        .alu_op(alu_op_wire_id),
        .mem_read(mem_read_wire_id),
        .mem_write(mem_write_wire_id),
        .alu_src(alu_src_wire_id),
        .reg_write(reg_write_wire_id),
        .jump(jump_wire_id)
    );


//////////////////////////// ID/EX registers
    pipe_reg #(.WIDTH(32)) pipe_reg_id_ex_instr(
        .clk(clk),
        .reset(reset),
        .d(if_id_instr_wire),
        .q(id_ex_instr_wire)
    );


    pipe_reg #(.WIDTH(32)) pipe_reg_id_ex_reg1(
        .clk(clk),
        .reset(reset),
        .d(reg1_wire_id),
        .q(id_ex_reg1_wire)
    );


    pipe_reg #(.WIDTH(32)) pipe_reg_id_ex_reg2(
        .clk(clk),
        .reset(reset),
        .d(reg2_wire_id),
        .q(id_ex_reg2_wire)
    );


    pipe_reg #(.WIDTH(32)) pipe_reg_id_ex_imm_value(
        .clk(clk),
        .reset(reset),
        .d(imm_value_wire_id),
        .q(id_ex_imm_value_wire)
    );


    pipe_reg #(.WIDTH(5)) pipe_reg_id_ex_destination_reg(
        .clk(clk),
        .reset(reset),
        .d(destination_reg_wire_id),
        .q(id_ex_destination_reg_wire)
    );


    pipe_reg #(.WIDTH(1)) pipe_reg_id_ex_mem_to_reg(
        .clk(clk),
        .reset(reset),
        .d(mem_to_reg_wire_id),
        .q(id_ex_mem_to_reg_wire)
    );
    pipe_reg #(.WIDTH(2)) pipe_reg_id_ex_alu_op(
        .clk(clk),
        .reset(reset),
        .d(alu_op_wire_id),
        .q(id_ex_alu_op_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_id_ex_mem_read(
        .clk(clk),
        .reset(reset),
        .d(mem_read_wire_id),
        .q(id_ex_mem_read_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_id_ex_mem_write(
        .clk(clk),
        .reset(reset),
        .d(mem_write_wire_id),
        .q(id_ex_mem_write_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_id_ex_alu_src(
        .clk(clk),
        .reset(reset),
        .d(alu_src_wire_id),
        .q(id_ex_alu_src_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_id_ex_reg_write(
        .clk(clk),
        .reset(reset),
        .d(reg_write_wire_id),
        .q(id_ex_reg_write_wire)
    );



//////////////////////////// Hazard_detection unit
    Hazard_detection Hazard_detection_inst (
        .id_ex_mem_read(id_ex_mem_read_wire),
        .id_ex_destination_reg(id_ex_destination_reg_wire),
        .if_id_rs(if_id_instr_wire[25:21]),
        .if_id_rt(if_id_instr_wire[20:16]),
        .branch_taken(branch_taken_wire_id),
        .jump(jump_wire_id),

        .Data_Hazard(Data_Hazard_wire),
        .IF_Flush(IF_Flush_wire)
    );

//////////////////////////// Execution
    EX_pipe_stage EX_pipe_stage_inst(
        .id_ex_instr(id_ex_instr_wire),
        .reg1(id_ex_reg1_wire),
        .reg2(id_ex_reg2_wire),
        .id_ex_imm_value(id_ex_imm_value_wire),
        .ex_mem_alu_result(ex_mem_alu_result_wire),
        .mem_wb_write_back_result(write_back_data_wire),
        .id_ex_alu_src(id_ex_alu_src_wire),
        .id_ex_alu_op(id_ex_alu_op_wire),
        .Forward_A(Forward_A_wire_fu),
        .Forward_B(Forward_B_wire_fu),

        .alu_in2_out(alu_in2_out_wire_ex),
        .alu_result(alu_result_wire_ex)
    );

//////////////////////////// Forwarding unit
    EX_Forwarding_unit EX_Forwarding_unit_inst(
        .ex_mem_reg_write(ex_mem_reg_write_wire),
        .ex_mem_write_reg_addr(),
        .id_ex_instr_rs(id_ex_instr_wire[25:21]),
        .id_ex_instr_rt(id_ex_instr_wire[20:16]),
        .mem_wb_reg_write(mem_wb_reg_write_wire),
        .mem_wb_write_reg_addr(mem_wb_destination_reg_wire),

        .Forward_A(Forward_A_wire_fu),
        .Forward_B(Forward_B_wire_fu)
    );


//////////////////////////// EX/MEM registers
            pipe_reg #(.WIDTH(32)) pipe_reg_ex_mem_instr(
        .clk(clk),
        .reset(reset),
        .d(id_ex_instr_wire),
        .q(ex_mem_instr_wire)
    );

    pipe_reg #(.WIDTH(5)) pipe_reg_ex_mem_destination_reg(
        .clk(clk),
        .reset(reset),
        .d(id_ex_destination_reg_wire),
        .q(ex_mem_destination_reg_wire)
    );

    pipe_reg #(.WIDTH(32)) pipe_reg_ex_mem_alu_result(
        .clk(clk),
        .reset(reset),
        .d(alu_result_wire_ex),
        .q(ex_mem_alu_result_wire)
    );

    pipe_reg #(.WIDTH(32)) pipe_reg_ex_mem_alu_in2_out(
        .clk(clk),
        .reset(reset),
        .d(alu_result_wire_ex),
        .q(ex_mem_alu_in2_out_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_ex_mem_mem_to_reg(
        .clk(clk),
        .reset(reset),
        .d(id_ex_mem_to_reg_wire),
        .q(ex_mem_mem_to_reg_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_ex_mem_mem_read(
        .clk(clk),
        .reset(reset),
        .d(id_ex_mem_read_wire),
        .q(ex_mem_mem_read_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_ex_mem__mem_write(
        .clk(clk),
        .reset(reset),
        .d(id_ex_mem_write_wire),
        .q(ex_mem_mem_write_wire)
    );

    pipe_reg #(.WIDTH(1)) pipe_reg_ex_mem_reg_write(
        .clk(clk),
        .reset(reset),
        .d(id_ex_reg_write_wire),
        .q(ex_mem_reg_write_wire)
    );



//////////////////////////// memory
    data_memory data_mem(
        .clk(clk),
        .mem_access_addr(ex_mem_alu_in2_out_wire),
        .mem_write_data(ex_mem_alu_result_wire),
        .mem_write_en(ex_mem_mem_write_wire),
        .mem_read_en(ex_mem_mem_read_wire),

        .mem_read_data(mem_read_data_wire)
    );


//////////////////////////// MEM/WB registers
            pipe_reg #(.WIDTH(32)) mem_wb_alu_result(
        .clk(clk),
        .reset(reset),
        .d(ex_mem_alu_result_wire),
        .q(mem_wb_alu_result_wire)
    );

    pipe_reg #(.WIDTH(32)) mem_wb_mem_read_data(
        .clk(clk),
        .reset(reset),
        .d(mem_read_data_wire),
        .q(mem_wb_mem_read_data_wire)
    );

    pipe_reg #(.WIDTH(1)) mem_wb_mem_to_reg(
        .clk(clk),
        .reset(reset),
        .d(ex_mem_mem_to_reg_wire),
        .q(mem_wb_mem_to_reg_wire)
    );

    pipe_reg #(.WIDTH(1)) mem_wb_reg_write(
        .clk(clk),
        .reset(reset),
        .d(ex_mem_reg_write_wire),
        .q(mem_wb_reg_write_wire)
    );

    pipe_reg #(.WIDTH(5)) mem_wb_destination_reg(
        .clk(clk),
        .reset(reset),
        .d(ex_mem_destination_reg_wire),
        .q(mem_wb_destination_reg_wire)
    );


//////////////////////////////// writeback
              mux2 #(.mux_width(32)) write_back_mux(
                  .a(mem_wb_alu_result_wire),
        .b(mem_wb_mem_read_data_wire),
        .sel(mem_wb_mem_to_reg_wire),
        .y(write_back_data_wire));

assign result = write_back_data_wire;
endmodule
```

IF_pipe_stage.v

```
module IF_pipe_stage(
    input clk, reset,
    input en,
    input [9:0] branch_address,
    input [9:0] jump_address,
    input branch_taken,
    input jump,
    output [9:0] pc_plus4,
    output [31:0] instr
    );


//Regs and Wires
reg [9:0] pc;
wire [9:0]pc_plus4_wire;
wire [9:0]branch_jump_wire;
wire [9:0]jump_pc_wire;

//PC Gen
assign pc_plus4_wire = pc + 4;
always @(posedge clk  or posedge reset)
begin
    if(reset)
       pc <= 10'b0000000000;
    else if (en)
       pc <= jump_pc_wire;
end

//Modules
mux2 #(.mux_width(10)) branch_taken_mux //DONE
    (   .a(pc_plus4_wire),
        .b(branch_address),
        .sel(branch_taken),
        .y(branch_jump_wire));

mux2 #(.mux_width(10)) jump_mux  //Where does this output into?
    (   .a(branch_jump_wire),
        .b(jump_address),
        .sel(jump),
        .y(jump_pc_wire));

instruction_mem inst_mem (
    .read_addr(pc),
    .data(instr));

assign pc_plus4 = pc_plus4_wire;

endmodule
```

ID_pipe_stage.v

```
module ID_pipe_stage(
    input  clk, reset,
    input  [9:0] pc_plus4,
    input  [31:0] instr,
    input  mem_wb_reg_write,
    input  [4:0] mem_wb_write_reg_addr,
    input  [31:0] mem_wb_write_back_data,
    input  Data_Hazard,
    input  Control_Hazard,
    output [31:0] reg1, reg2,
    output [31:0] imm_value,
    output [9:0] branch_address,
    output [9:0] jump_address,
    output branch_taken,
    output [4:0] destination_reg,
    output mem_to_reg,
    output [1:0] alu_op,
    output mem_read,
    output mem_write,
    output alu_src,
    output reg_write,
    output jump
    );


    // Remember that we test if the branch is taken or not in the decode stage.

    //Regs and Wires
    //mem_to_reg, alu_op, mem_read, mem_write, alu_src, reg_write
    wire reg_dst_wire;
    wire branch_wire;

    //branch taken
    wire eq_test_wire;

    //reg1 and reg2
    wire [31:0]reg1_wire;
    wire [31:0]reg2_wire;
    //imm_value
    wire [31:0]imm_value_wire;


    //Assigns
    //mem_to_reg, alu_op, mem_read, mem_write, alu_src, reg_write
    wire mem_to_reg_wire;
    wire [1:0]alu_op_wire;
    wire mem_read_wire;
    wire mem_write_wire;
    wire alu_src_wire;
    wire reg_write_wire;

    control control_inst(
        .reset(reset),
        .opcode(instr[31:26]),
        .reg_dst(reg_dst_wire),
        .mem_to_reg(mem_to_reg_wire),
        .alu_op(alu_op_wire),
        .mem_read(mem_read_wire),
        .mem_write(mem_write_wire),
        .alu_src(alu_src_wire),
        .reg_write(reg_write_wire),
        .branch(branch_wire),
        .jump(jump)
    );

    mux2 #(.mux_width(1)) mem_to_reg_mux
    (   .a(mem_to_reg_wire),
        .b(1'b0),
        .sel((!Data_Hazard & Control_Hazard)),
        .y(mem_to_reg));

    mux2 #(.mux_width(2)) alu_op_mux
```

```
(   .a(alu_op_wire),
    .b(2'b00),
    .sel((!Data_Hazard & Control_Hazard)),
    .y(alu_op));

mux2 #(.mux_width(1)) mem_read_mux
(   .a(mem_read_wire),
    .b(1'b0),
    .sel((!Data_Hazard & Control_Hazard)),
    .y(mem_read));

mux2 #(.mux_width(1)) mem_write_mux
(   .a(mem_write_wire),
    .b(1'b0),
    .sel((!Data_Hazard & Control_Hazard)),
    .y(mem_write));

mux2 #(.mux_width(1)) alu_src_mux
(   .a(alu_src_wire),
    .b(1'b0),
    .sel((!Data_Hazard & Control_Hazard)),
    .y(alu_src));

mux2 #(.mux_width(1)) reg_write_mux
(   .a(reg_write_wire),
    .b(1'b1),
    .sel((!Data_Hazard & Control_Hazard)),
    .y(reg_write));

//jump_address
assign jump_address = instr[25:0] << 2;

//branch address
assign branch_address = pc_plus4 + (imm_value_wire << 2); //where does pc_plus4 come from.

//branch taken
assign eq_test_wire = ((reg1_wire^reg2_wire)==32'd0) ? 1'b1: 1'b0;
assign branch_taken = branch_wire && eq_test_wire;

//reg1 and reg2
register_file register_file_inst (
    .clk(clk),
    .reset(reset),
    .reg_write_en(mem_wb_reg_write),
    .reg_write_dest(mem_wb_write_reg_addr),
    .reg_write_data(mem_wb_write_back_data),
    .reg_read_addr_1(instr[25:21]), //not sure which is bound to which
    .reg_read_addr_2(instr[20:16]), //not sure which is bound to which
    .reg_read_data_1(reg1_wire),
    .reg_read_data_2(reg2_wire));
assign reg1 = reg1_wire;
assign reg2 = reg2_wire;

//imm_value
```

```
    sign_extend sign_ex_inst (
        .sign_ex_in(instr[15:0]),
        .sign_ex_out(imm_value_wire));
    assign imm_value = imm_value_wire;

    //destination_reg
    mux2 #(.mux_width(5)) jump_mux
    (   .a(instr[20:16]),
        .b(instr[15:11]),
        .sel(reg_dst_wire),
        .y(destination_reg));

endmodule
```

EX_pipe_stage.v

```
module EX_pipe_stage(
    input [31:0] id_ex_instr,
    input [31:0] reg1, reg2,
    input [31:0] id_ex_imm_value,
    input [31:0] ex_mem_alu_result,
    input [31:0] mem_wb_write_back_result,
    input id_ex_alu_src,
    input [1:0] id_ex_alu_op,
    input [1:0] Forward_A, Forward_B,
    output [31:0] alu_in2_out,
    output [31:0] alu_result
    );
    //REGS and WIRES
    //alu_result

    //alu_in2_out

    //ASSIGNS
    wire [31:0]id_ex_reg1_mux_wire;
    wire [31:0]id_ex_reg2_mux_wire;
    wire [31:0]alu_in2_wire;
    wire [3:0]ALU_Control_wire;

    wire [31:0]zero_wire;
    assign zero_wire = 32'b00000000000000000000000000000000;
    mux4 #(.mux_width(32)) id_ex_reg1_mux
    (   .a(reg1),
        .b(mem_wb_write_back_result),
        .c(ex_mem_alu_result),
        .d(32'b00000000000000000000000000000000),//idk what to put here
        .sel(Forward_A),
        .y(id_ex_reg1_mux_wire));

    mux4 #(.mux_width(32)) id_ex_reg2_mux
    (   .a(reg2),
        .b(mem_wb_write_back_result),
        .c(ex_mem_alu_result),
        .d(32'b00000000000000000000000000000000),//idk what to put here
        .sel(Forward_B),
        .y(id_ex_reg2_mux_wire));


    mux2 #(.mux_width(32)) id_ex_imm_value_mux
    (   .a(id_ex_reg2_mux_wire),
        .b(id_ex_imm_value),
        .sel(id_ex_alu_src),
        .y(alu_in2_wire));

    ALUControl ALUControl_inst(
        .ALUOp(id_ex_alu_op),
        .Function(id_ex_instr[5:0]),
        .ALU_Control(ALU_Control_wire)
    );

    ALU ALU_inst (
        .a(id_ex_reg1_mux_wire),
        .b(alu_in2_wire),
        .alu_control(ALU_Control_wire),
        .zero(zero_wire),
        .alu_result(alu_result)
    );


endmodule
```

EX_Forwarding_unit.v

```
module EX_Forwarding_unit(
    input ex_mem_reg_write,//USED
    input [4:0] ex_mem_write_reg_addr, //EX/MEM.Register
    input [4:0] id_ex_instr_rs, //ID/EX.RegisterRs
    input [4:0] id_ex_instr_rt, //ID/EX.RegisterRt
    input mem_wb_reg_write,//USED
    input [4:0] mem_wb_write_reg_addr,//
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
    );

    always @(*)
    begin
            // Write your code here that calculates the values of Forward_A and Forward_B
              if (ex_mem_reg_write
                  & (ex_mem_write_reg_addr != 5'b00000)
                  & (ex_mem_write_reg_addr == id_ex_instr_rs))
                  Forward_A <= 2'b10;
              else if (mem_wb_reg_write
                  & (mem_wb_write_reg_addr != 5'b00000)
                  & ~(ex_mem_write_reg_addr && (ex_mem_write_reg_addr != 5'b00000)
                      & (ex_mem_write_reg_addr == id_ex_instr_rs))
                  & (mem_wb_write_reg_addr == id_ex_instr_rs))
                  Forward_A <= 2'b01;
              else
                  Forward_A = 2'b00;
```

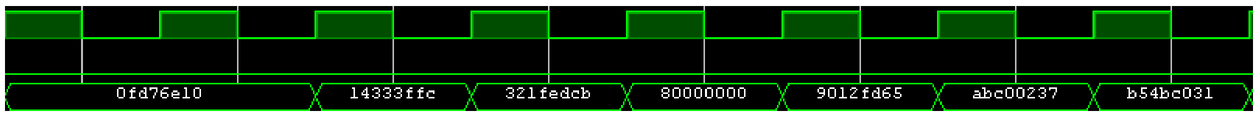
```
        if (ex_mem_reg_write
          & (ex_mem_write_reg_addr != 5'b00000)
          & (ex_mem_write_reg_addr == id_ex_instr_rt))
          Forward_B <= 2'b10;
        else if (mem_wb_reg_write
          & (mem_wb_write_reg_addr != 5'b00000)
          & ~(ex_mem_reg_write & (ex_mem_write_reg_addr != 5'b00000)
            & (ex_mem_write_reg_addr == id_ex_instr_rt))
          & (mem_wb_write_reg_addr == id_ex_instr_rt))
          Forward_B <= 2'b01;
        else
          Forward_B = 2'b00;



  end
endmodule
```

Simulation Results

Even given the testbench, it was impossibly difficult to begin to search for what segment of the pipeline would cause the output of the processor to fail.



This is a chunk of the output from simulation. As you can see it produces correct values in the memory and the cases from the test bench are able to pass. I will now detail the implementation of the stages of the pipeline as I completed them

IF

Instruction Fetch required the implementation of 2 muxes, a PC Gen, and a module to access the instruction memory. The first MUX took a selection input from the signal branch_taken. It outputs either an input branch_address or pc_plus4 into the next MUX. The second MUX takes a selection signal from jump, and sends either jump_address or the output of the first MUX to the PC. This PC is modified to only cycle when it receives a signal from en. When this occurs, it passes the result of the second MUX to the Instruction Memory module so that the next instruction can be output.

ID

Instruction Decode is a far more complicated module than Instruction Fetch. It requires input from Hazard Detection and Write Back in order to function. After splicing the 32 bit instruction it obtained from Instruction Fetch, it will output a number of things. In this section of processing, sign extension, destination register calculation, jump and branch address, the branch_taken flag, register file outputs, and parsed information from the original instruction are generated and output into the Execution step. This step essentially handles all of the writing and interpreting of the given instruction input. Instruction Decode relies heavily on the Hazard Detection module, as it uses it to decide whether or not an NOP is in the processing of the provided instruction.

EX

The execution step is very simple compared to what Instruction Decode does, but it requires the Forwarding Unit to function. This step is essentially just passing information through the ALU to complete operations. It requires 3 MUXes and access to both the ALU and the ALU_control module. It then outputs the alu results as well as what the second operand was.The first two muxes are simply selections between the values stored in the two registers passed earlier or immediate values.

MEM

Memory required actual writing of a module, and we were able to use the old architecture from Lab 2 to complete this section. I simply created an instance of this in the main processor file, and was then able to implement it into the cycle.

WB

Write Back was a simple bridging between the Execution output and the Regfile that is written into and modified in the Instruction Decode step. It is simply composed of a single MUX that takes a mem_to_reg signal and outputs either the alu_result that was obtained or read data from the called register. This output is then sent under the name write_back_data to the Instruction decode section and the cycle continues from there.