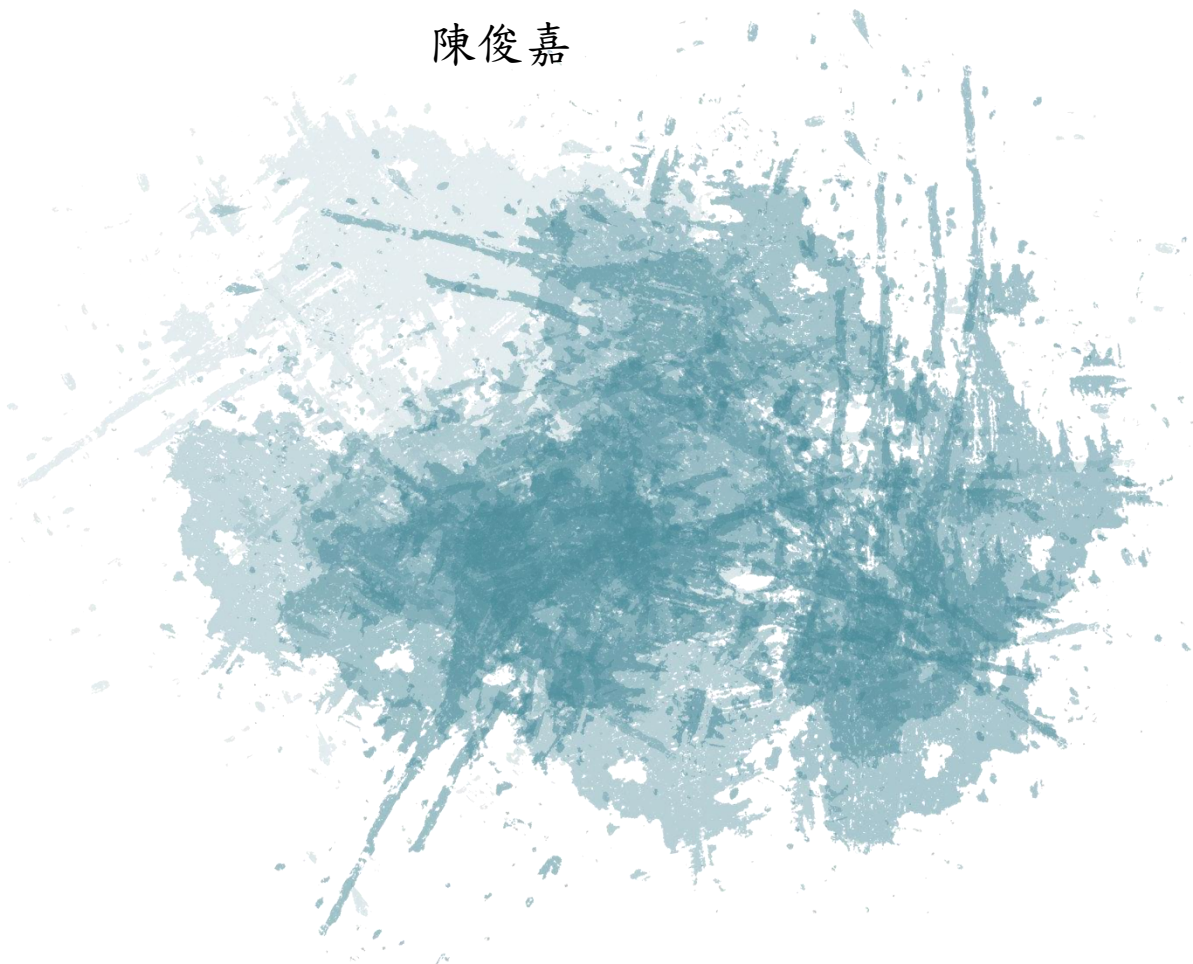
Computer Organization Project 3

四電機三甲
B10830009
陳俊嘉



Part III :Implement a 5-stage pipelined processor with forwarding and hazard detection.

- Adder Module

Adder.v > {} Adder

```v
module Adder (data1, data2, data_out) ;
    output [31:0]data_out;
    input [31:0]data1, data2;

    assign data_out = data1 + data2;

endmodule
```

Adder Module 在這次PA中一樣是做計算下一個program counter的位置，data1會放目前的address，data2會放4，以達成每次位置加四的功能。

- Instruction Memory Module

```v
`define INSTR_MEM_SIZE  128 // Bytes

/*
 * Declaration of Instruction Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module IM(
    // Outputs
    output wire [31:0]instr,
    // Inputs
    input [31:0]instr_addr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign instr = {InstrMem[instr_addr], InstrMem[instr_addr + 1], InstrMem[instr_addr + 2], InstrMem[instr_addr + 3]};
endmodule
```

Instruction Memory內儲存要執行的指令，以一個byte為單位，一次取四個byte (32bits)，為Big Endian，第一個指到的byte會放在MSB。

- IF/ID Module

IF_ID.v > {} IF_ID

```v
module IF_ID(
    //Output
    output [31:0]instr_out,
    //Inputs
    input [31:0]instr_in,
    input IF_ID_write,
    input clk
);

reg [31:0]instr_reg;


always @(negedge clk) begin
    if(IF_ID_write)begin
        instr_reg <= instr_in;
    end
end

assign instr_out = instr_reg;

endmodule
```

此Module主要作為儲存instruction fetch出來的指令的暫存器，為negative edge trigger，並透過IF_ID_write來處理要stall時的情況。

- RF Module

```v
`define REG_MEM_SIZE    32  // Words

/*
 * Declaration of Register File for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module RF(
    // Outputs
    output [31:0]src_data,
    output [31:0]tar_data,
    // Inputs
    input [4:0]src_addr,
    input [4:0]tar_addr,
    input [4:0]dst_addr,
    input [31:0]dst_data,
    input clk,
    input reg_write
);

    /*
     * Declaration of inner register.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [31:0]R[0:`REG_MEM_SIZE - 1];

        //assign value by register point by address
        assign src_data = R[src_addr];
        assign tar_data = R[tar_addr];
    always@(posedge clk)begin
        //write result to register addressed by dst_data
        if(reg_write)begin
            R[dst_addr] <= dst_data;
        end
    end
endmodule
```

與之前的PA一樣，透過從instruction memory讀出來的source address 和target address去找到相對應的資料並輸出，並透過reg write 訊號控制寫入資料在destination address指到的位置。

- Control Module

Control.v > {} Control

```v
module Control(
    //outputs
    output reg [1:0]ALU_OP,
    output reg reg_write,
    output reg reg_dst,
    output reg ALU_src,
    output reg mem_write,
    output reg mem_read,
    output reg mem2reg,
    output reg branch,
    output reg jump,
    //inputs
    input [5:0]OP
);
    always@(*)begin
        case(OP)
    //if is R type -> set ALU_OP = 10
        6'd0:begin
            ALU_OP <= 2'b10;
            reg_write <= 1;
            reg_dst <= 1;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //ADD immediate
        6'b001100:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //SUB immediate
        6'b001101:begin
            ALU_OP <= 2'b01;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //store
        6'b010000:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            //reg_dst <= 0; Don't care
            ALU_src <= 1;
            mem_write <= 1;
            mem_read <= 0;
            //mem2reg <= 0; Don't care
            branch <= 0;
            jump <= 0;
        end
        //load
        6'b010001:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 1;
            mem2reg <= 1;
            branch <= 0;
            jump <= 0;
        end
    //beq
        6'b010011:begin
            ALU_OP <= 2'b01;
            reg_write <= 0;
            //reg_dst <= 0; Don't care
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            //mem2reg <= 1; Don't care
            branch <= 1;
            jump <= 0;
        end
    //jump
        6'b011100:begin
            ALU_OP <= 2'b01;
            reg_write <= 0;
            //reg_dst <= 0; Don't care
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            //mem2reg <= 1; Don't care
            branch <= 0;
            jump <= 1;
        end
        default:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            reg_dst <= 0;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
        jump <= 0;
        end
        endcase
    end

endmodule
```

Control Module 為處理大部分訊號的controller，會透過判斷instruction最前面的6個bits，也就是OP code，來決定要給其他相對應的component的訊號。

- Hazard Detection Module

```
module Hazard_Detection (
    //Outputs
    output reg PC_write,
    output reg stall,
    output reg IF_ID_write,
    //Inputs
    input [4:0]EX_tar_addr,
    input [4:0]ID_tar_addr,
    input [4:0]ID_src_addr,
    input EX_mem_read
);
    initial begin
        stall <= 0;
        IF_ID_write <= 1;
        PC_write <= 1;
    end
    always @(*) begin
        if(EX_mem_read && ((ID_src_addr == EX_tar_addr) || (EX_tar_addr == ID_tar_addr)))begin
            stall <= 1;
            IF_ID_write <= 0;
            PC_write <= 0;
        end
        else begin
            stall <= 0;
            IF_ID_write <= 1;
            PC_write <= 1;
        end
    end
endmodule
```

此module是用來判斷有沒有發生hazard的，並輸出PC write、stall和IF_ID write訊號來決定要不要stall，而要stall的條件為當前的instruction會用到上個instruction會改變到的register。通常會在執行load指令時發生。

- MUX Module (for stall)

```
module Stall_MUX(
    //Outputs
    output reg [4:0]src_addr_out,
    output reg [4:0]I_dst_addr_out,
    output reg [1:0]ALU_OP_out,
    output reg reg_write_out,
    output reg reg_dst_out,
    output reg ALU_src_out,
    output reg mem_write_out,
    output reg mem_read_out,
    output reg mem2reg_out,
    output reg branch_out,
    output reg jump_out,
    //Inputs
    input [4:0]src_addr_in,
    input [4:0]I_dst_addr_in,
    input [1:0]ALU_OP_in,
    input reg_write_in,
    input reg_dst_in,
    input ALU_src_in,
    input mem_write_in,
    input mem_read_in,
    input mem2reg_in,
    input branch_in,
    input jump_in,
    input stall
);
```

```
always@(*)begin
    if(stall)begin
        src_addr_out <= 0;
        I_dst_addr_out <= 0;
        ALU_OP_out <= 0;
        reg_write_out <= 0;
        reg_dst_out <= 0;
        ALU_src_out <= 0;
        mem_write_out <= 0;
        mem_read_out <= 0;
        mem2reg_out <= 0;
        branch_out <= 0;
        jump_out <= 0;
    end
    else begin
        src_addr_out <= src_addr_in;
        I_dst_addr_out <= I_dst_addr_in;
        ALU_OP_out <= ALU_OP_in;
        reg_write_out <= reg_write_in;
        reg_dst_out <= reg_dst_in;
        ALU_src_out <= ALU_src_in;
        mem_write_out <= mem_write_in;
        mem_read_out <= mem_read_in;
        mem2reg_out <= mem2reg_in;
        branch_out <= branch_in;
        jump_out <= jump_in;
    end
end


endmodule
```

這個mux是用來控制control訊號的，若發生hazard時，為了要stall，所以要將接下來的訊號都設為零。而因為在接下來的forwarding unit內會要判斷EX stage的src及tar address，所以在stall時，也會將src及tar address設為零。

- Sign Extension Module

若執行的指令為I type的，就需要透過此module將instruction的後16bits，也就是immediate，轉成32 bits，才能給接下來的ALU進行運算。

```
module Sign_Extend(
    //output
    output reg [31:0]extend_out,
    //input
    input [15:0]immediate
);

always @(*) begin
    if(immediate[15] == 0)begin
        extend_out <= {16'd0, immediate};
    end
    else if(immediate[15] == 1)begin
        extend_out <= {16'hFFFF, immediate};
    end
end


endmodule
```

- ID/EX Module

```
module ID_EX(
    //Outputs
    output wb_out,
    output ALU_src_out,
    output mem_read_out,
    output mem_write_out,
    output reg_dst_out,
    output mem2reg_out,
    output branch_out,
    output jump_out,
    output [1:0]ALU_OP_out,
    output [31:0]src_data_out,
    output [31:0]R_tar_data_out,
    output [31:0]I_tar_data_out,
    output [4:0]shamt_out,
    output [4:0]R_dst_addr_out,
    output [4:0]I_dst_addr_out,
    output [4:0]src_addr_out,
    output [5:0]funct_ctrl_out,
    //Inputs
    input wb_in,
    input ALU_src_in,
    input mem_read_in,
    input mem_write_in,
    input reg_dst_in,
    input mem2reg_in,
    input branch_in,
    input jump_in,
    input [1:0]ALU_OP_in,
    input [31:0]src_data_in,
    input [31:0]R_tar_data_in,
    input [31:0]I_tar_data_in,
    input [4:0]shamt_in,
    input [4:0]R_dst_addr_in,
    input [4:0]I_dst_addr_in,
    input [4:0]src_addr_in,
    input [5:0]funct_ctrl_in,
    input clk
);
```

```
reg wb, ALU_src, mem_read, mem_write, reg_dst, mem2reg;
reg branch, jump;
reg [1:0]ALU_OP;
reg [31:0]src_data, R_tar_data, I_tar_data;
reg [4:0]shamt, R_dst_addr, I_dst_addr, src_addr;
reg [5:0]funct_ctrl;

always @(negedge clk) begin
    wb <= wb_in;
    ALU_src <= ALU_src_in;
    mem_read <= mem_read_in;
    mem_write <= mem_write_in;
    mem2reg <= mem2reg_in;
    reg_dst <= reg_dst_in;
    branch <= branch_in;
    jump <= jump_in;
    ALU_OP <= ALU_OP_in;
    src_data <= src_data_in;
    R_tar_data <= R_tar_data_in;
    I_tar_data <= I_tar_data_in;
    src_addr <= src_addr_in;
    shamt <= shamt_in;
    R_dst_addr <= R_dst_addr_in;
    I_dst_addr <= I_dst_addr_in;
    funct_ctrl <= funct_ctrl_in;
end

assign wb_out = wb;
assign ALU_src_out = ALU_src;
assign mem_read_out = mem_read;
assign mem_write_out = mem_write;
assign reg_dst_out = reg_dst;
assign mem2reg_out = mem2reg;
assign branch_out = branch;
assign jump_out = jump;
assign ALU_OP_out = ALU_OP;
assign src_data_out = src_data;
assign R_tar_data_out = R_tar_data;
assign I_tar_data_out = I_tar_data;
assign shamt_out = shamt;
assign R_dst_addr_out = R_dst_addr;
assign I_dst_addr_out = I_dst_addr;
assign src_addr_out = src_addr;
assign funct_ctrl_out = funct_ctrl;

endmodule
```

此module為一暫存器，在negative edge時存取在ID階段的輸出訊號，並輸出至EX stage 的component。

- Forwarding Unit Module

```
module Forward(
    //Outputs
    output reg [1:0]forward_src,
    output reg [1:0]forward_tar,
    //Inputs
    input [4:0]MEM_dst_addr,
    input [4:0]dst_addr,
    input [4:0]EX_src_addr,
    input [4:0]I_dst_addr,
    input MEM_reg_write,
    input reg_write
);
always@(*)begin
    /*
    *   forward_src signal
    */
    //mem hazard -> forward data from the prior alu result
    if(MEM_reg_write && (MEM_dst_addr != 0) && (MEM_dst_addr == EX_src_addr))begin
        forward_src <= 2'b10;
    end
    //EX hazard -> forward data from data memory or earlier alu result
    else if(reg_write && (dst_addr != 0) && (MEM_dst_addr != EX_src_addr) && (dst_addr ==EX_src_addr))begin
        forward_src = 2'b01;
    end
    else forward_src = 0;
    /*
    *   forward_tar signal
    */
    //mem hazard
    if(MEM_reg_write && (MEM_dst_addr != 0) && (MEM_dst_addr == I_dst_addr))begin
        forward_tar <= 2'b10;
    end
    //EX hazard
    else if(reg_write && (dst_addr != 0) && (MEM_dst_addr != I_dst_addr) && (dst_addr ==I_dst_addr))begin
        forward_tar = 2'b01;
    end
    else forward_tar = 0;
end
endmodule
```

判斷是否有需要用到forwarding，並輸出forward_src及forward_tar，給在ALU前的兩個MUX來選擇輸入ALU的資料。

- MUX Module

```
module MUX5(
    //Outputs
    output reg [4:0]data_out,
    //Inputs
    input [4:0]data0,
    input [4:0]data1,
    input select
);
always @(*) begin
    if(select)begin
        data_out <= data1;
    end else begin
        data_out <= data0;
    end
end

endmodule
```

```
module MUX32(
    //Outputs
    output reg [31:0]data_out,
    //Inputs
    input [31:0]data0,
    input [31:0]data1,
    input select
);
always @(*) begin
    if(select)begin
        data_out <= data1;
    end else begin
        data_out <= data0;
    end
end

endmodule
```

```
module MUX32_3(
    //Outputs
    output reg [31:0]data_out,
    //Inputs
    input [31:0]data0,
    input [31:0]data1,
    input [31:0]data2,
    input [1:0]select
);
always @(*) begin
    case(select)
        0:  data_out <= data0;
        1:  data_out <= data1;
        2:  data_out <= data2;
        default:
            data_out <= 0;
    endcase
end

endmodule
```

最左的為5-bit MUX，用來選擇destination address。中間的是32-bit MUX 用來選擇輸入ALU target值及選擇要寫回Register File的值。右邊的一樣是32-bit MUX，但是有三個in/output，用來判斷forward的data。

- ALU Control Module

```
module ALU_Control (
    output reg [5:0]funct,
    //inputs
    input [5:0]funct_ctrl,
    input [1:0]ALU_OP
);
    always@(ALU_OP or funct_ctrl)begin
    //R type
        if(ALU_OP == 2'b10)begin
            case(funct_ctrl)
                6'b001011:  funct <= 6'b001001; //ADDU
                6'b001101:  funct <= 6'b001010; //SUBU
                6'b100101:  funct <= 6'b010010; //OR
                6'b000010:  funct <= 6'b100010; //SRL
            endcase
        end
        //I type that use ALU?for add only
        else if(ALU_OP == 2'b00)begin
            funct <= 6'b001001;
        end
    //I type that use ALU for subtract only
        else if(ALU_OP == 2'b01)begin
            funct <= 6'b001010;
        end

        else begin
            funct <= 0;
        end
    end
endmodule
```

與之前的PA相同，主要透過instruction給的function control 訊號及controller判斷過後輸出的ALU_OP來判斷ALU需要做甚麼運算，並輸出function給ALU做對應的動作。

- ALU Module

```
module ALU (
    //outputs
    output reg [31:0]data_out,
    output reg zero,
    //inputs
    input [31:0]src_data,
    input [31:0]tar_data,
    input [4:0]shamt,
    input [5:0]funct
);
    always@(*)begin
        case(funct)
            //ADDU
            6'b001001:  data_out <= src_data + tar_data;
            //SUBU
            6'b001010:  data_out <= src_data - tar_data;
            //OR
            6'b010010:  data_out <= src_data | tar_data;
            //SRL
            6'b100010:  data_out <= src_data >> shamt;
            default:    data_out <= 32'd0;
        endcase
        zero <= (data_out == 0)? 1'b1 : 1'b0;
    end

endmodule
```

此module為CPU中主要做運算的component，透過ALU controller給的function來執行動作。

- EX/MEM Module

```
module EX_MEM(
    //outputs
    output wb_out,
    output mem_read_out,
    output mem_write_out,
    output mem2reg_out,
    output branch_out,
    output jump_out,
    output [31:0]mem_write_data_out,
    output [31:0]ALU_result_out,
    output [4:0]dst_addr_out,
    //Inputs
    input wb_in,
    input mem_read_in,
    input mem_write_in,
    input mem2reg_in,
    input branch_in,
    input jump_in,
    input [31:0]mem_write_data_in,
    input [31:0]ALU_result_in,
    input [4:0]dst_addr_in,
    input clk
);
reg wb, mem_read, mem_write, mem2reg, branch, jump;
reg [31:0]ALU_result, mem_write_data;
reg [4:0]dst_addr;

always @(negedge clk) begin
    wb <= wb_in;
    mem_read <= mem_read_in;
    mem_write <= mem_write_in;
    mem2reg <= mem2reg_in;
    branch <= branch_in;
    jump <= jump_in;
    mem_write_data <= mem_write_data_in;
    ALU_result <= ALU_result_in;
    dst_addr <= dst_addr_in;

end

assign wb_out = wb;
assign mem_read_out = mem_read;
assign mem_write_out = mem_write;
assign mem2reg_out = mem2reg;
assign branch_out = branch;
assign jump_out = jump;
assign mem_write_data_out = mem_write_data;
assign ALU_result_out = ALU_result;
assign dst_addr_out = dst_addr;



endmodule
```

與前面的IF/ID和ID/EX一樣，為儲存EX stage輸出至下一階段的暫存器，為negative edge trigger。

- DM Module

```
`define DATA_MEM_SIZE   128 // Bytes

/*
 * Declaration of Data Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module DM(
    // Outputs
    output reg [31:0]read_data,
    // Inputs
    input [31:0]write_data,
    input [31:0]addr,
    input mem_read,
    input mem_write,
    input clk
);

    /*
     * Declaration of data memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];

    always@(posedge clk)begin
        if(mem_read)begin
            read_data <= {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]};
        end

        else if(mem_write)begin
            {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]} <= write_data;
        end
    end
endmodule
```

Data memory 為除了Instruction memory及Register File以外的memory，可以透過load和store指令來存取其中的資料。

- MEM/WB Module

為五個stage中最後要寫回資料到Register File的暫存器。

```
module MEM_WB(
    //outputs
    output wb_out,
    output mem2reg_out,
    output [31:0]mem_read_data_out,
    output [31:0]ALU_result_out,
    output [4:0]dst_addr_out,
    //Inputs
    input wb_in,
    input mem2reg_in,
    input [31:0]mem_read_data_in,
    input [31:0]ALU_result_in,
    input [4:0]dst_addr_in,
    input clk
);
reg wb, mem2reg;
reg [31:0]ALU_result, mem_read_data;
reg [4:0]dst_addr;
always @(negedge clk) begin
    wb <= wb_in;
    mem2reg <= mem2reg_in;
    mem_read_data <= mem_read_data_in;
    ALU_result <= ALU_result_in;
    dst_addr <= dst_addr_in;
end

assign wb_out = wb;
assign mem2reg_out = mem2reg;
assign mem_read_data_out = mem_read_data;
assign ALU_result_out = ALU_result;
assign dst_addr_out = dst_addr;

endmodule
```

- Final CPU Module

```
module FinalCPU(
    // Outputs
    output  wire            PCWrite,
    output  wire    [31:0]  Addr_Out,

    // Inputs
    input   wire    [31:0]  Addr_In,
    input   wire            clk
);
    wire [31:0]IM_OUT, INSTR;
    wire [31:0]ID_SRC_DATA, SRC_DATA, FORWARD_SRC_DATA;
    wire [31:0]ID_R_TAR_DATA, R_TAR_DATA, FORWARD_TAR_DATA;
    wire [31:0]ID_I_TAR_DATA, I_TAR_DATA, TAR_DATA;
    wire [31:0]ALU_OUT, MEM_ALU_OUT, WB_ALU_OUT;
    wire [31:0]MEM_READ_DATA, WB_MEM_READ_DATA;
    wire [31:0]MEM_WRITE_DATA;
    wire [31:0]DST_DATA;
    wire [5:0]FUNCT_CTRL, FUNCT;
    wire [4:0]R_DST_ADDR, I_DST_ADDR, EX_DST_ADDR, MEM_DST_ADDR, DST_ADDR;
    wire [4:0]EX_SRC_ADDR;
    wire [4:0]ID_I_DST_ADDR, ID_SRC_ADDR;
    wire [4:0]SHAMT;
    wire [1:0]CTRL_ALU_OP, ID_ALU_OP, ALU_OP;
    wire [1:0]FORWARD_TAR, FORWARD_SRC;
    wire CTRL_REG_WRITE, ID_REG_WRITE, EX_REG_WRITE, MEM_REG_WRITE, REG_WRITE;
    wire CTRL_REG_DST, ID_REG_DST, REG_DST;
    wire CTRL_ALU_SRC, ID_ALU_SRC, ALU_SRC;
    wire CTRL_MEM_WRITE, ID_MEM_WRITE, EX_MEM_WRITE, MEM_WRITE;
    wire CTRL_MEM_READ, ID_MEM_READ, EX_MEM_READ, MEM_READ;
    wire CTRL_MEM2REG, ID_MEM2REG, EX_MEM2REG, MEM_MEM2REG, MEM2REG;
    wire CTRL_BRANCH, ID_BRANCH, EX_BRANCH, BRANCH;
    wire CTRL_JUMP, ID_JUMP, EX_JUMP, JUMP;
    wire ZERO_FLAG;
    wire STALL, IF_ID_WRITE;

    Adder Address_Adder(
        //Outputs
        .data_out(Addr_Out),
        //Inputs
        .data1(Addr_In),
        .data2(32'd4)
    );
    /*
     * Declaration of Instruction Memory.
     * CAUTION: DONT MODIFY THE NAME.
     */
    IM Instr_Memory(
        // Outputs
        .instr(IM_OUT),
        // Inputs
        .instr_addr(Addr_In)
    );

    IF_ID IF_ID_Register(
        //Outputs
        .instr_out(INSTR),
        //Inputs
        .instr_in(IM_OUT),
        .IF_ID_write(IF_ID_WRITE),
        .clk(clk)
    );

    Hazard_Detection Hazard_Detection_Unit(
        //Outputs
        .PC_write(PCWrite),
        .stall(STALL),
        .IF_ID_write(IF_ID_WRITE),
        //Inputs
        .EX_tar_addr(I_DST_ADDR),
        .ID_tar_addr(INSTR[20:16]),
        .ID_src_addr(INSTR[25:21]),
        .EX_mem_read(EX_MEM_READ)
    );
```

```
    Control Controller(
        //outputs
        .ALU_OP(CTRL_ALU_OP),
        .reg_write(CTRL_REG_WRITE),
        .reg_dst(CTRL_REG_DST),
        .ALU_src(CTRL_ALU_SRC),
        .mem_write(CTRL_MEM_WRITE),
        .mem_read(CTRL_MEM_READ),
        .mem2reg(CTRL_MEM2REG),
        .branch(CTRL_BRANCH),
        .jump(CTRL_JUMP),
        //Inputs
        .OP(INSTR[31:26])
    );

    Stall_MUX Stallation_MUX(
        //Outputs
        .src_addr_out(ID_SRC_ADDR),
        .I_dst_addr_out(ID_I_DST_ADDR),
        .ALU_OP_out(ID_ALU_OP),
        .reg_write_out(ID_REG_WRITE),
        .reg_dst_out(ID_REG_DST),
        .ALU_src_out(ID_ALU_SRC),
        .mem_write_out(ID_MEM_WRITE),
        .mem_read_out(ID_MEM_READ),
        .mem2reg_out(ID_MEM2REG),
        .branch_out(ID_BRANCH),
        .jump_out(ID_JUMP),
        //Inputs
        .src_addr_in(INSTR[25:21]),
        .I_dst_addr_in(INSTR[20:16]),
        .ALU_OP_in(CTRL_ALU_OP),
        .reg_write_in(CTRL_REG_WRITE),
        .reg_dst_in(CTRL_REG_DST),
        .ALU_src_in(CTRL_ALU_SRC),
        .mem_write_in(CTRL_MEM_WRITE),
        .mem_read_in(CTRL_MEM_READ),
        .mem2reg_in(CTRL_MEM2REG),
        .branch_in(CTRL_BRANCH),
        .jump_in(CTRL_JUMP),
        //select signal
        .stall(STALL)
    );
    /*
     * Declaration of Register File.
     * CAUTION: DONT MODIFY THE NAME.
     */
    RF Register_File(
        // Outputs
        .src_data(ID_SRC_DATA),
        .tar_data(ID_R_TAR_DATA),
        // Inputs
        .src_addr(INSTR[25:21]),
        .tar_addr(INSTR[20:16]),
        .dst_addr(DST_ADDR),
        .dst_data(DST_DATA),
        .reg_write(REG_WRITE),
        .clk(clk)
    );

    Sign_Extend Sign_Extension(
        //Output
        .extend_out(ID_I_TAR_DATA),
        //Input
        .immediate(INSTR[15:0])
    );
```

```
    ID_EX ID_EX_Register(
        //Outputs
        .wb_out(EX_REG_WRITE),
        .ALU_src_out(ALU_SRC),
        .mem_read_out(EX_MEM_READ),
        .mem_write_out(EX_MEM_WRITE),
        .reg_dst_out(REG_DST),
        .mem2reg_out(EX_MEM2REG),
        .branch_out(EX_BRANCH),
        .jump_out(EX_JUMP),
        .ALU_OP_out(ALU_OP),
        .src_data_out(SRC_DATA),
        .R_tar_data_out(R_TAR_DATA),
        .I_tar_data_out(I_TAR_DATA),
        .shamt_out(SHAMT),
        .R_dst_addr_out(R_DST_ADDR),
        .I_dst_addr_out(I_DST_ADDR),
        .src_addr_out(EX_SRC_ADDR),
        .funct_ctrl_out(FUNCT_CTRL),
        //Inputs
        .wb_in(ID_REG_WRITE),
        .ALU_src_in(ID_ALU_SRC),
        .mem_read_in(ID_MEM_READ),
        .mem_write_in(ID_MEM_WRITE),
        .reg_dst_in(ID_REG_DST),
        .mem2reg_in(ID_MEM2REG),
        .branch_in(ID_BRANCH),
        .jump_in(ID_JUMP),
        .ALU_OP_in(ID_ALU_OP),
        .src_data_in(ID_SRC_DATA),
        .R_tar_data_in(ID_R_TAR_DATA),
        .I_tar_data_in(ID_I_TAR_DATA),
        .shamt_in(INSTR[10:6]),
        .R_dst_addr_in(INSTR[15:11]),
        .I_dst_addr_in(ID_I_DST_ADDR),
        .src_addr_in(ID_SRC_ADDR),
        .funct_ctrl_in(INSTR[5:0]),
        .clk(clk)
    );

    Forward Forwarding_Unit(
        //Outputs
        .forward_src(FORWARD_SRC),
        .forward_tar(FORWARD_TAR),
        //Inputs
        .MEM_dst_addr(MEM_DST_ADDR),
        .dst_addr(DST_ADDR),
        .EX_src_addr(EX_SRC_ADDR),
        .I_dst_addr(I_DST_ADDR),
        .MEM_reg_write(MEM_REG_WRITE),
        .reg_write(REG_WRITE)
    );

    MUX32_3 Forward_SRC_MUX(
        //output
        .data_out(FORWARD_SRC_DATA),
        //inputs
        .data0(SRC_DATA),
        .data1(DST_DATA),
        .data2(MEM_ALU_OUT),
        .select(FORWARD_SRC)
    );

    MUX32_3 Forward_TAR_MUX(
        //output
        .data_out(FORWARD_TAR_DATA),
        //inputs
        .data0(R_TAR_DATA),
        .data1(DST_DATA),
        .data2(MEM_ALU_OUT),
        .select(FORWARD_TAR)
    );
```

```
    ALU_Control ALU_Controller(
        //output
        .funct(FUNCT),
        //inputs
        .funct_ctrl(FUNCT_CTRL),
        .ALU_OP(ALU_OP)
    );

    MUX32 ALU_SRC_MUX(
        //output
        .data_out(TAR_DATA),
        //inputs
        .data0(FORWARD_TAR_DATA),
        .data1(I_TAR_DATA),
        .select(ALU_SRC)
    );

    ALU ALU_Unit(
        //output
        .data_out(ALU_OUT),
        .zero(ZERO_FLAG),
        //inputs
        .src_data(FORWARD_SRC_DATA),
        .tar_data(TAR_DATA),
        .shamt(SHAMT),
        .funct(FUNCT)
    );

    MUX5 Reg_Dst_MUX(
        .data_out(EX_DST_ADDR),
        //inputs
        .data0(I_DST_ADDR),
        .data1(R_DST_ADDR),
        .select(REG_DST)
    );

    EX_MEM EX_MEM_Register(
        //outputs
        .wb_out(MEM_REG_WRITE),
        .mem_read_out(MEM_READ),
        .mem_write_out(MEM_WRITE),
        .mem2reg_out(MEM_MEM2REG),
        .branch_out(BRANCH),
        .jump_out(JUMP),
        .mem_write_data_out(MEM_WRITE_DATA),
        .ALU_result_out(MEM_ALU_OUT),
        .dst_addr_out(MEM_DST_ADDR),
        //Inputs
        .wb_in(EX_REG_WRITE),
        .mem_read_in(EX_MEM_READ),
        .mem_write_in(EX_MEM_WRITE),
        .mem2reg_in(EX_MEM2REG),
        .branch_in(EX_BRANCH),
        .jump_in(EX_JUMP),
        .mem_write_data_in(FORWARD_TAR_DATA),
        .ALU_result_in(ALU_OUT),
        .dst_addr_in(EX_DST_ADDR),
        .clk(clk)
    );
```

```
    DM Data_Memory(
        //Ouput
        .read_data(MEM_READ_DATA),
        //Inputs
        .write_data(MEM_WRITE_DATA),
        .addr(MEM_ALU_OUT),
        .mem_read(MEM_READ),
        .mem_write(MEM_WRITE),
        .clk(clk)
    );

    MEM_WB MEM_WB_Register(
        //Outputs
        .wb_out(REG_WRITE),
        .mem2reg_out(MEM2REG),
        .mem_read_data_out(WB_MEM_READ_DATA),
        .ALU_result_out(WB_ALU_OUT),
        .dst_addr_out(DST_ADDR),
        //Inputs
        .wb_in(MEM_REG_WRITE),
        .mem2reg_in(MEM_MEM2REG),
        .mem_read_data_in(MEM_READ_DATA),
        .ALU_result_in(MEM_ALU_OUT),
        .dst_addr_in(MEM_DST_ADDR),
        .clk(clk)
    );

    MUX32 Dst_Data_MUX(
        //output
        .data_out(DST_DATA),
        //inputs
        .data0(WB_ALU_OUT),
        .data1(WB_MEM_READ_DATA),
        .select(MEM2REG)
    );

endmodule
```

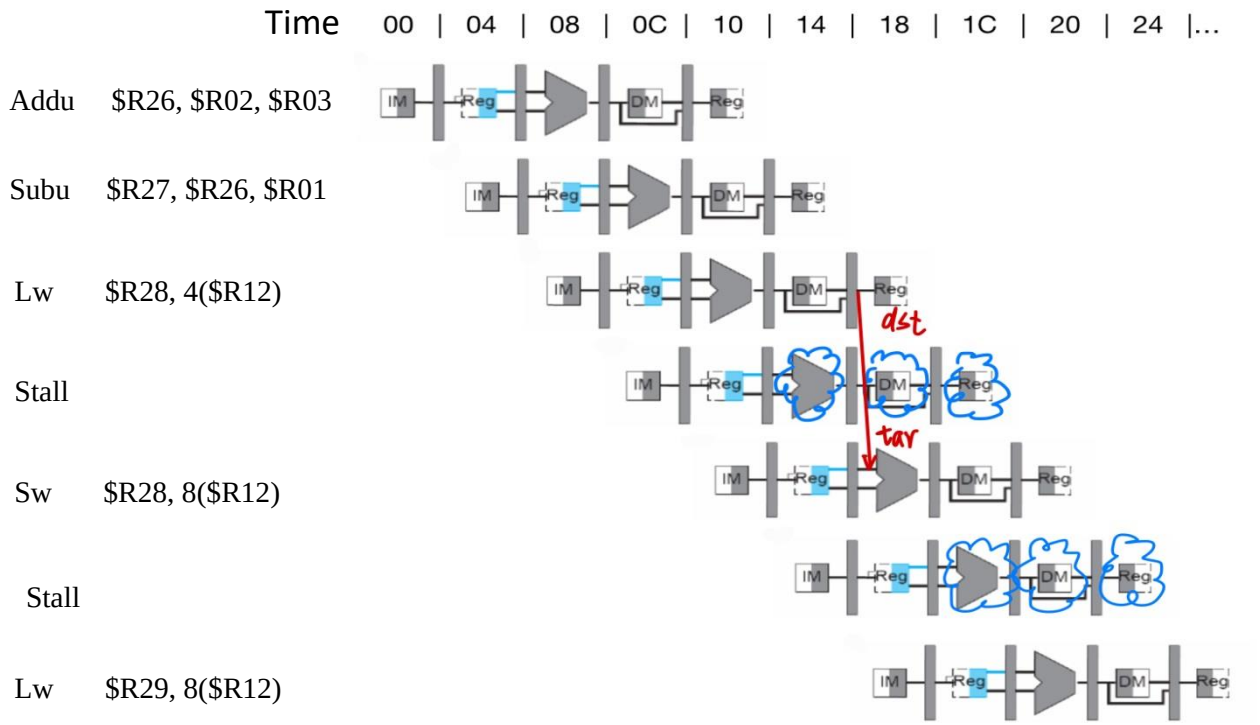
- Final CPU test bench

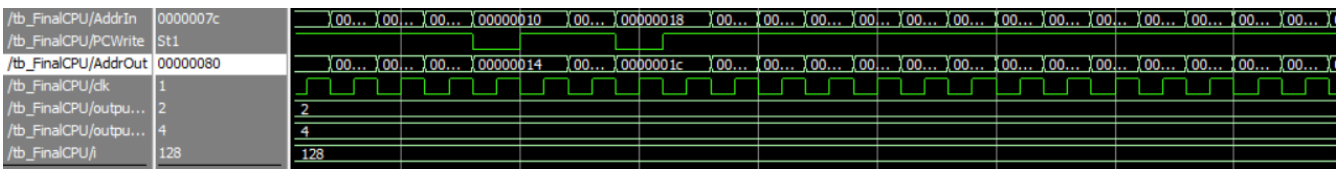
```
// Setting timescale
`timescale 10 ns / 1 ns

// Declarations
`define DELAY           1   // # * timescale
`define INSTR_SIZE      8   // bit width
`define INSTR_MAX       128 // bytes
`define INSTR_FILE      "testbench/IM.dat"
`define REG_SIZE        32  // bit width
`define REG_MAX         32  // words
`define REG_FILE        "testbench/RF.dat"
`define DATA_SIZE       8   // bit width
`define DATA_MAX        128 // bytes
`define DATA_FILE       "testbench/DM.dat"
`define OUTPUT_REG      "testbench/RF.out"
`define OUTPUT_DATA     "testbench/DM.out"

// Declaration
`define LOW     1'b0
`define HIGH    1'b1

module tb_FinalCPU;

    // Inputs
    reg [31:0] AddrIn;

    // Outputs
    wire        PCWrite;
    wire [31:0] AddrOut;

    // Clock
    reg clk;

    // Testbench variables
    reg [`INSTR_SIZE-1  :0] instrMem    [0:`INSTR_MAX-1];
    reg [`REG_SIZE-1    :0] regMem      [0:`REG_MAX-1];
    reg [`DATA_SIZE-1   :0] dataMem     [0:`DATA_MAX-1];
    integer output_reg;
    integer output_data;
    integer i;

    // Instantiate the Unit Under Test (UUT)
    FinalCPU UUT(
        // Outputs
        .PCWrite(PCWrite),
        .Addr_Out(AddrOut),
        // Inputs
        .Addr_In(AddrIn),
        .clk(clk)
    );
```

```
    initial
    begin : Preprocess
        // Initialize inputs
        AddrIn  = 32'd0;
        clk = `LOW;

        // Initialize testbench files
        $readmemh(`INSTR_FILE,  instrMem);
        $readmemh(`REG_FILE,    regMem);
        $readmemh(`DATA_FILE,   dataMem);
        output_reg  = $fopen(`OUTPUT_REG);
        output_data = $fopen(`OUTPUT_DATA);

        // Initialize intruction memory
        for (i = 0; i < `INSTR_MAX; i = i + 1)
        begin
            UUT.Instr_Memory.InstrMem[i] = instrMem[i];
        end

        // Initialize register file
        for (i = 0; i < `REG_MAX; i = i + 1)
        begin
            UUT.Register_File.R[i] = regMem[i];
        end

        // Initialize data memory
        for (i = 0; i < `DATA_MAX; i = i + 1)
        begin
            UUT.Data_Memory.DataMem[i] = dataMem[i];
        end

        #`DELAY;    // Wait for global reset to finish
    end

    always
    begin : ClockGenerator
        #`DELAY;
        clk <= ~clk;
    end

    always
    begin : StimuliProcess
        // Start testing
        while (AddrIn < `INSTR_MAX - 4)
        begin
            @(negedge clk);
            AddrIn <= (PCWrite) ? AddrOut : AddrIn;
            @(posedge clk);
        end

        // Read out all register value
        for (i = 0; i < `REG_MAX; i = i + 1)
        begin
            regMem[i] = UUT.Register_File.R[i];
            $fwrite(output_reg, "%x\n", regMem[i]);
        end

        // Read out all memory value
        for (i = 0; i < `DATA_MAX; i = i + 1)
        begin
            dataMem[i] = UUT.Data_Memory.DataMem[i];
            $fwrite(output_data, "%x\n", dataMem[i]);
        end

        // Close output files for safety
        $fclose(output_reg);
        $fclose(output_data);

        // Stop the simulation
        $stop();
    end

endmodule
```

- Final CPU Simulation

```
|  InstrAddr  |   Instruction                                |   Binary Code                                                  |   Hexadecimal   |
|             |                                              |  OPCode | Rs   | Rt   | immediate                         |                 |
     0x0000_0000            addu    $R26, $R02, $R03       ; 0b000000_00010_00011_11010_00000_001011 => 0x00_43_D0_0B
     0x0000_0004            subu    $R27, $R26, $R01       ; 0b000000_11010_00001_11011_00000_001101 => 0x03_41_D8_0D
     0x0000_0008            lw      $R28, 4($R12)          ; 0b010001_01100_11100_0000000000000100   => 0x45_9C_00_04
     0x0000_000C            sw      $R28, 8($R12)          ; 0b010000_01100_11100_0000000000001000   => 0x41_9C_00_08
     0x0000_0010            lw      $R29, 8($R12)          ; 0b010001_01100_11101_0000000000001000   => 0x45_9D_00_08
     0x0000_0014            addu    $R30, $R29, $R10       ; 0b000000_11101_01010_11110_00000_001011 => 0x03_AA_F0_0B
```





在Adder_in = 10的那個cycle，因為store 要用到load到register的值，所以hazard detection unit會讓PC write設為零，讓address不要繼續往下跑，並讓stall訊號為一，把後面的訊號都設為零以達到stall的效果。由上圖可知PC write 訊號正常升降。

Conclusion and insights

這次的PA在前面的兩個Part其實我個人覺得蠻簡單的，因為大部分都是上次PA做過的，只要在每個stage中間再加上暫存器就好了，也沒有Hazard和forwarding的問題，所以前面其實沒有花太多的時間，經過上次PA的教訓後，在Top Module的接線也透過更易懂的命名還有更加的細心，讓我這次在前兩個Part中免於debug的凌虐。

因為在上課的時候，在講發生hazard及forwarding的條件那邊稍微沒有很專注在聽，所以這次做到part3的時候還花了一點時間去複習那部分才開始做hazard detection 和 forwarding unit這兩個module，但還是遇到了不少問題，更是花了我好幾天才解決問題。

上課的時候聽都覺得沒甚麼問題，但真的做了這次的PA才知道自己不太熟悉的地方，像是在address是0C的那個store要做stall，但我一開始就以為要在0C的那個cycle做stall，但其實因為是在ID的那個階段才能知道是否會有hazard，所以是在下一個cycle，也就是address是10的時候才會將PC write關掉。

在確定自己hazard detection的PC write那些訊號的部分沒有問題後，卻發現還是沒有正常的store值回去，所以我就改了Register file位置是28($R28)的值，發現是forwarding的那兩個MUX還是用原本的訊號，所以才確定是forwarding unit的問題，但在詳細的複習那部分的條件後發現forwarding unit module的程式條件並沒有打錯，所以就把全部的圖畫出來(上頁中間的圖)才發現問題，因為雖然controller後面的MUX在stall的時候把全部的訊號都關掉了，但instruction給的source 和 target address還是會正常傳到EX的stage，並造成forwarding unit的誤判，所以雖然在PA3給的圖中應該沒有要將那兩個address消除，但我還是把那兩個address加進MUX裡面，這樣在stall的時候address就不會被傳進去了，最後問題就解決了。

這次的PA也是讓我學到了不少，雖然也是花了不少的時間，但是也順便複習了之前上課學的東西，也加深了不足的觀念。