國 立 台 灣 科 技 大 學

電 機 工 程 系

# 計算機組織

(EE3005301)

# Project3

班級： 四電機三甲

學號： B10507004

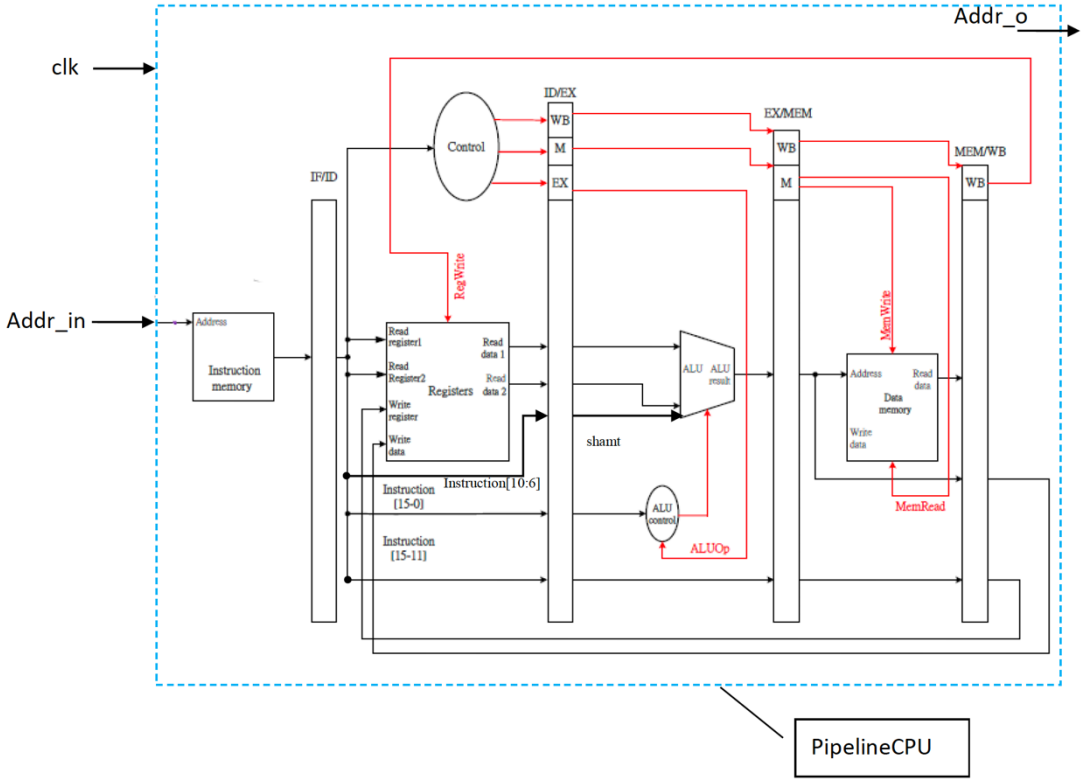
指導老師： 陳雅淑

姓名： 游照臨

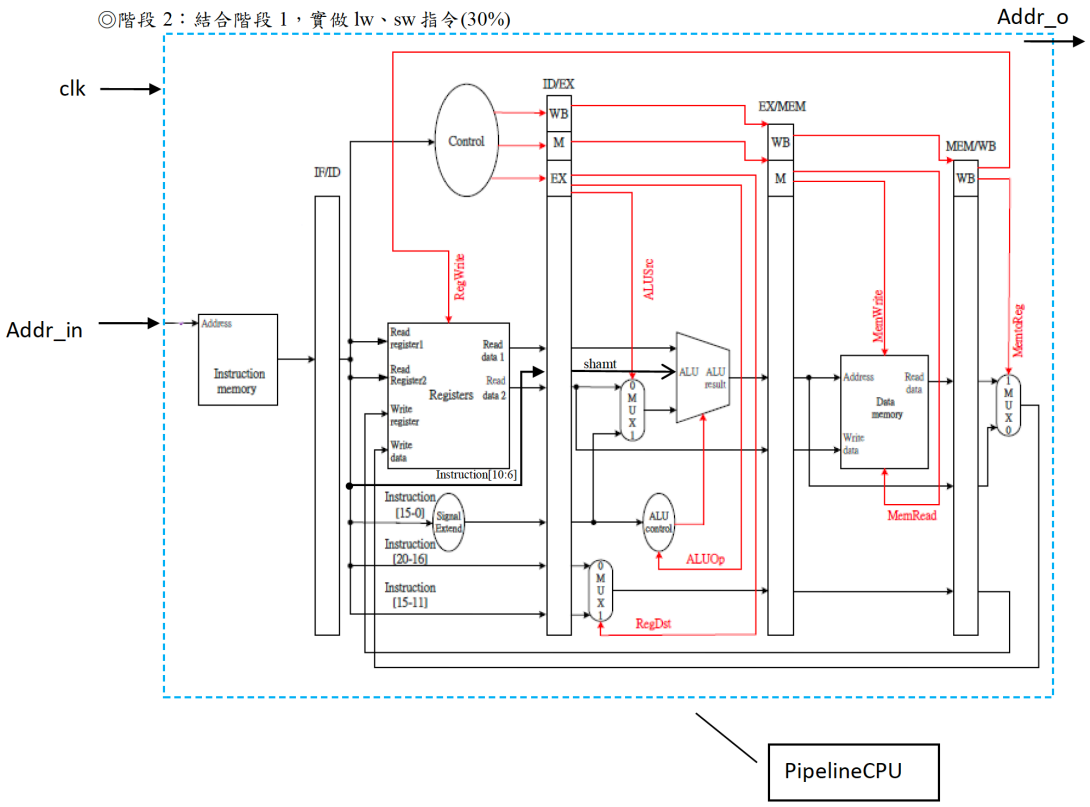
中華民國 108 年 6 月 10 日

# 壹、程式功能

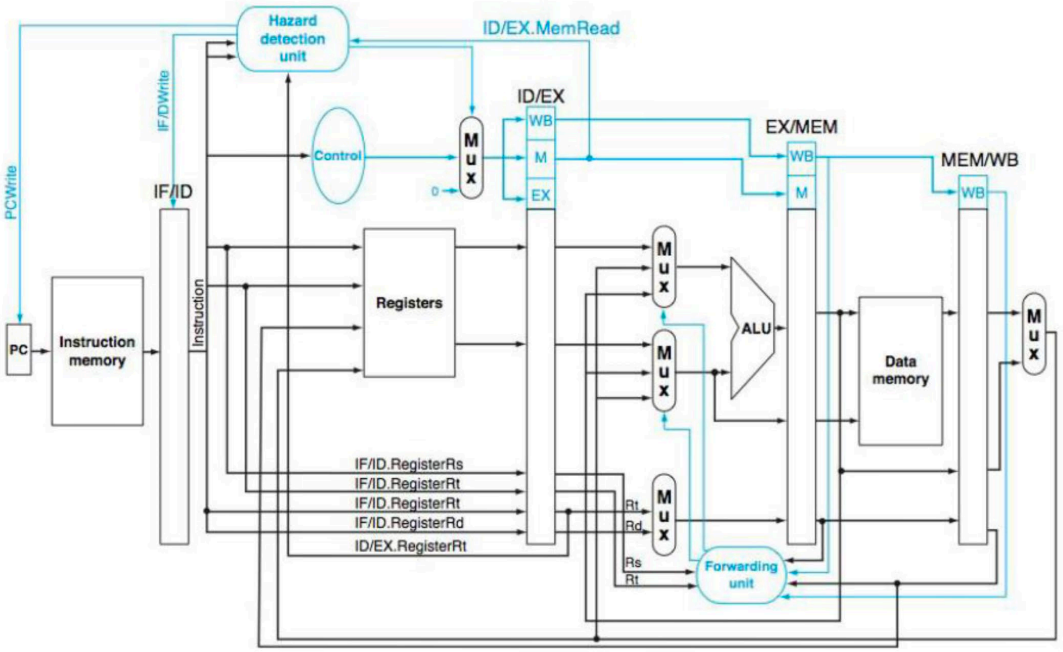
## 一、完成一個支援 ADD、SUB、SHR、SHL 指令的 MIPS CPU



## 二、結合第一階段的架構，新增 LW、SW 功能




## 三、結合前兩階段的架構，新增 Forwarding 與 Hazard Detection(Stall)



## 貳、程式原始碼(順序由模組英文字母排序)

### 一、Adder

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

assign data_o = data1 + data2;

endmodule
```

Adder 模組為一個 32bit 輸入，32bit 輸出的雙輸入、單輸出加法器，其實現方式為使用 Verilog 中的 Data Flow Model。

### 二、ALU

```
module ALU(input [4:0] shamt,
           input [31:0] src1,
           input [31:0] src2,
           input [5:0] operation,
           output reg [31:0] result,
           output zero);

/*
 add 27
 sub 28
 and 29
 or 30
 srl 31
 sll 32
 */

assign zero = result==0?1:0;

always@(*)begin
    case(operation)
    6'd27:
        result <= src1 + src2;
```

```
        6'd28:
            result <= src1 - src2;
        6'd29:
            result <= src1 & src2;
        6'd30:
            result <= src1 | src2;
        6'd31:
            result <=    src2 >> shamt;
        6'd32:
            result <= src2 << shamt;
        default:
            result <= 0;
        endcase
end
endmodule
```

透過 ALU 的 OP Code 定義，定義出 27 為加法；28 為減法；29 為邏輯與運算；30 為邏輯或運算；31 為位元右移；32 為位元左移，並且當計算結果為 0 時，zero 腳位需要輸出高電位。

三、ALUctrl

```
module ALUctrl(
    input [5:0] funct,
    input [2:0] ALUOp,
    output reg[5:0] operation
);
//funct or ALUOp
always@(*)begin
    if(ALUOp == 3'b010)begin
        case(funct)
        6'd21:
            operation <= 27;
        6'd22:
            operation <= 28;
        6'd23:
            operation <= 29;
        6'd24:
            operation <= 30;
        6'd25:
```

```
                operation <= 31;
            6'd26:
                operation <= 32;
            endcase
        end
        else if (ALUOp == 3'b000)begin //lw sw addi
            operation <= 27; //add
        end
        else if (ALUOp == 3'b001)begin //sbui
            operation <= 28; //sub
        end
        else if (ALUOp == 3'b101)begin
            operation <= 28; //sub
        end
    end

    endmodule
```

此程式碼的主要功能為，將 funct 與 ALUOp 兩者的輸入，轉換為 ALU 的 Operation，在階段 1 時 ALUOp 一定會等於 3’b010，因此只須考慮 funct 為 21 至 26 的狀況，依照題本第 3 至 4 頁的對照，將指令對應到指定的 ALU opcode。

四、Control

```
module Control(
input [5:0]Op,
output reg RegDst,
output reg MemRead,
output reg MemtoReg,
output reg [2:0]ALUOp,
output reg MemWrite,
output reg ALUSrc,
output reg RegWrite,

output reg Jump,
output reg Branch
);



always@(Op)begin
```

```
case(Op)
    6'd54: begin// Rtyp
        ALUOp <= 3'b010;
        RegDst <= 1'b1;
        ALUSrc <= 1'b0;
        MemtoReg <= 1'b0;
        RegWrite <= 1'b1;
        MemRead <= 1'b0;
        MemWrite <= 1'b0;
        Branch <= 1'b0;
        Jump <= 1'b0;
    end

    6'd39: begin// sw
        ALUOp <= 3'b000;
        // RegDst <= 1'bz;
        ALUSrc <= 1'b1;
        // MemtoReg <= 1'bz;
        RegWrite <= 1'b0;
        MemRead <= 1'b0;
        MemWrite <= 1'b1;
        Branch <= 1'b0;
        Jump <= 1'b0;
    end

    6'd40: begin// lw
        ALUOp <= 3'b000;
        RegDst <= 1'b0;
        ALUSrc <= 1'b1;
        MemtoReg <= 1'b1;
        RegWrite <= 1'b1;
        MemRead <= 1'b1;
        MemWrite <= 1'b0;
        Branch <= 1'b0;
        Jump <= 1'b0;
    end

    6'd41: begin//addi
```

```
            ALUOp <= 3'b000;
            RegDst <= 1'b0;
            ALUSrc <= 1'b1;
            MemtoReg <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            Branch <= 1'b0;
            Jump <= 1'b0;
        end
        6'd42: begin //subi
            ALUOp <= 3'b001;
            RegDst <= 1'b0;
            ALUSrc <= 1'b1;
            MemtoReg <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            Branch <= 1'b0;
            Jump <= 1'b0;
        end

        6'd31:begin //beq
            ALUOp <= 3'b101;
            RegDst <= 1'bz;
            ALUSrc <= 1'b0;
            MemtoReg <= 1'bz;
            RegWrite <= 1'b0;
            MemRead <= 1'b0;
            MemWrite <= 1'b0;
            Branch <= 1'b1;
            Jump <= 1'b0;
        end

        6'd32:begin //j
            ALUOp <= 3'b101;
            RegDst <= 1'bz;
            ALUSrc <= 1'b0;
```

```
                MemtoReg <= 1'bz;
                RegWrite <= 1'b0;
                MemRead <= 1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b1;
            end


            default: begin
                ALUOp <= 3'bzzz;
                RegDst <= 1'bz;
                ALUSrc <= 1'bz;
                MemtoReg <= 1'bz;
                RegWrite <= 1'bz;
                MemRead <= 1'bz;
                MemWrite <= 1'bz;
                Jump <= 1'bz;
            end
        endcase


end

endmodule
```

Control 模組透過讀取 instruction 的 operation，可以將輸出結果用以控制 Reg、Mem、ALUctrl 及其他多工器的選擇腳位。

五、DM

```
module DM(input clk,
            input [31:0] addr,
            input [31:0] data,
            input MemRead,
            input MemWrite,
            output reg [31:0] DM_data);


    integer i;
    reg [7:0]Mem[127:0];
```

```
    initial begin
        for(i = 0;i<128;i = i+1)begin
            Mem[i] = 8'd0;
        end
        DM_data = 32'b0;
    end

//clk &
//clk or MemWrite
    always@(clk or MemWrite)begin
        if(MemWrite)begin
            Mem[addr] <= data[31:24];
            Mem[addr + 1] <= data[23:16];
            Mem[addr + 2] <= data[15:8];
            Mem[addr + 3] <= data[7:0];
        end
    end

    always@(clk)begin
        if(MemRead)begin
            DM_data <= {Mem[addr],Mem[addr+1],Mem[addr+2],Mem[addr+3]};
        end
    end

endmodule
```

Data Memory 中總共有 128 格，預設所有內容都是 0，可以透過 MemRead 與 MemWrite 等接腳控制輸入與輸出。

六、EX_MEM

```
module EX_MEM(
    input clk,
    input [1:0]WB_in,
    input [1:0]M_in,
    input [31:0]ALUresult_in,
    input [31:0]write_mem_data_in,
    input [4:0]write_register_in,
    input [4:0]Rd_in,
```

```
    output reg [1:0]WB_out,
    output reg [1:0]M_out,
    output reg [31:0]ALUresult_out,
    output reg [31:0]write_mem_data_out,
    output reg [4:0]write_register_out,
    output reg [4:0]Rd_out
);
initial begin
    WB_out = 2'b0;
    M_out = 2'b0;
    ALUresult_out = 32'b0;
    write_mem_data_out = 32'b0;
    write_register_out = 5'b0;
    Rd_out = 5'b0;
end

always @(posedge clk)begin
    WB_out <= WB_in;
    M_out <= M_in;
    ALUresult_out <= ALUresult_in;
    write_mem_data_out <= write_mem_data_in;
    write_register_out <= write_register_in;
    Rd_out <= Rd_in;
end

endmodule
```

EX_MEM 是 pipeline 中間的暫存記憶體模組，主要負責將 ALU 計算後的資料傳輸給 Memory 與下一級模組。

七、Forwarding

```
module Forwarding(
    input clk,
    input [4:0]ID_EX_Rs,
    input [4:0]ID_EX_Rt,
    input [4:0]EX_MEM_Rd,
    input [4:0]MEM_WB_Rd,
    input EX_MEM_RegWrite,
```

```
    input MEM_WB_RegWrite,

    output reg [1:0]ForwardA,
    output reg [1:0]ForwardB
);
initial begin
    ForwardA <= 2'b00;
    ForwardB <= 2'b00;
end
// posedge
always @(*)begin
    if(EX_MEM_RegWrite == 1 &&
        EX_MEM_Rd != 0 &&
        EX_MEM_Rd == ID_EX_Rs)begin

        ForwardA <= 2'b10;
    end

    else if(MEM_WB_RegWrite == 1 &&
        MEM_WB_Rd != 0 &&
        MEM_WB_Rd == ID_EX_Rs)begin

        ForwardA <= 2'b01;
    end

    else begin
        ForwardA <= 2'b00;
    end



    if(EX_MEM_RegWrite == 1 &&
        EX_MEM_Rd != 0 &&
        EX_MEM_Rd == ID_EX_Rt)begin

        ForwardB <= 2'b10;

    end
```

```
    else if(MEM_WB_RegWrite == 1 &&
        MEM_WB_Rd != 0 &&
        MEM_WB_Rd == ID_EX_Rt)begin

        ForwardB <= 2'b01;

    end

    else begin
        ForwardB <= 2'b00;
    end
end



endmodule
```

Forwarding 模組的用途是判斷到非 lw 的 Data Hazard 時的處理方式，其判斷條件如上方程式所示，主要用以控制兩個 Forwarding 專用的 MUX 以達到提早將輸出拉回輸入使用的功能。

八、Hazard_detection

```
module Hazard_detection(
    input clk,
    input ID_EX_MemRead,
    input ID_EX_RegisterRt,
    input IF_ID_RegisterRs,
    input IF_ID_RegisterRt,

    output reg stall_mux,
    output reg IF_ID_Write,
    output reg PC_Write
);


initial begin
    stall_mux <= 0;
    IF_ID_Write <= 1;
    PC_Write <= 1;
end
```

```
//posedge clk
always @(*)begin
    if(ID_EX_MemRead &&(
        (ID_EX_RegisterRt == IF_ID_RegisterRs) || (ID_EX_RegisterRt ==
IF_ID_RegisterRt)
        ))begin
        stall_mux = 1;
        IF_ID_Write = 0;
        PC_Write = 0;
    end
    else begin
        stall_mux = 0;
        IF_ID_Write = 1;
        PC_Write = 1;
    end
end



endmodule
```

Hazard Detection 主要作用於 lw 狀態下的 Data Hazard，由於這種情形無法透過 Forwarding 的方式將資料提早拉回，因此需要 stall 一個 clk 的時間，因此本模組可以控制 PC 是否要寫入下一個 instruction 的控制腳。

九、ID_EX

```
module ID_EX(
    input clk,
    input [1:0]WB_in,
    input [1:0]M_in,
    input [4:0]EX_in,
    input [31:0] RTdata_in,
    input [31:0] srcl_in,
    input [4:0] shamt_in,
    input [31:0] se_in,
    input [4:0] R_add_in,
    input [4:0] I_add_in,
    input [4:0] Rs_in,
    input [4:0] Rt_in,
```

```
    input [4:0] Rd_in,

    output reg [1:0]    WB_out,
    output reg [1:0]    M_out,
    output reg [4:0]    EX_out,
    output reg [31:0] RTdata_out,
    output reg [31:0] srcl_out,
    output reg [4:0]    shamt_out,
    output reg [31:0] se_out,
    output reg [4:0]    R_add_out,
    output reg [4:0]    I_add_out,
    output reg [4:0]    Rs_out,
    output reg [4:0]    Rt_out,
    output reg [4:0]    Rd_out
);

initial begin
        WB_out = 2'b0;
        M_out = 2'b0;
        EX_out = 5'b0;
        RTdata_out = 32'b0;
        srcl_out = 32'b0;
        shamt_out = 5'b0;
        se_out = 32'b0;
        R_add_out = 5'b0;
        I_add_out = 5'b0;
        Rs_out = 5'b0;
        Rt_out = 5'b0;
        Rd_out = 5'b0;
end

always @(posedge clk)begin
        WB_out <= WB_in;
        M_out <= M_in;
        EX_out <= EX_in;
        RTdata_out <= RTdata_in;
        srcl_out <= srcl_in;
        shamt_out <= shamt_in;
```

```
            se_out <= se_in;
            R_add_out <= R_add_in;
            I_add_out <= I_add_in;
            Rs_out <= Rs_in;
            Rt_out <= Rt_in;
            Rd_out <= Rd_in;
end

endmodule
```

ID_EX 為 pipeline 中的第二個暫存器，主要負責將 Control 與 Register 讀取到的值給 ALU 及其相關模組使用。

十、IF_ID

```
module IF_ID(
      input clk,
      input enable,
      input [31:0]instruction_in,
      output reg [31:0]instruction_out
);

initial begin
      instruction_out = 32'b0;
end



always @(posedge clk)begin
      instruction_out = enable?instruction_in:instruction_out;
end


endmodule
```

IF_ID 模組為 pipeline 中的第一個暫存器，主要負責把 instruction memory 中的 instruction 提供給 Control、Register 使用。

十一、IM

```
module IM(input [31:0] Addr_in,
                output reg [31:0] instr);

```

```
    reg [31:0]Instr[31:0];

    initial begin
        Instr[0] = {6'd54,5'd9,5'd10,5'd8,5'd0,6'd21};
        Instr[1] = {6'd54,5'd9,5'd10,5'd9,5'd0,6'd22};
        Instr[2] = {6'd54,5'd0,5'd12,5'd11,5'd2,6'd25};
        Instr[3] = {6'd54,5'd0,5'd15,5'd13,5'd5,6'd26};

        Instr[4] = {6'd39,5'd11,5'd8,16'd2};
        Instr[5] = {6'd40,5'd10,5'd19,16'd2};
        Instr[6] = {6'd39,5'd10,5'd20,16'd4};
        Instr[7] = {6'd39,5'd10,5'd8,16'd2};
        Instr[8] = {6'd40,5'd11,5'd20,16'd3};


        Instr[9] = {6'd54,5'd13,5'd12,5'd14,5'd0,6'd21};
        Instr[10] = {6'd54,5'd14,5'd13,5'd15,5'd0,6'd22};


        Instr[11] = {6'd40,5'd11,5'd19,16'd2};
        Instr[12] = {6'd39,5'd9,5'd19,16'd2};
        Instr[13] = {6'd40,5'd9,5'd11,16'd2};
        Instr[14] = {6'd54,5'd11,5'd10,5'd11,5'd0,6'd21};

        instr = Instr[0];
    end

    always@(Addr_in)begin
        instr <= Instr[Addr_in>>2];
    end

endmodule
```

Instruction Memory 負責存放準備被執行的指令，以本次專案為例總共有 15 個指令等待被執行，將組合語言翻譯為機器語言後，把指定的 Instruction memory address 除以四(右移兩格)後，把指令給輸出。

## 十二、MEM_WB

```
module MEM_WB(
```

```
    input clk,
    input [1:0]WB_in,
    input [31:0]Read_data_in,
    input [31:0]ALUresult_in,
    input [4:0]write_register_in,
    input [4:0]Rd_in,

    output reg [1:0]WB_out,
    output reg [31:0]Read_data_out,
    output reg [31:0]ALUresult_out,
    output reg [4:0]write_register_out,
    output reg [4:0]Rd_out
);


initial begin
    WB_out = 2'b0;
    Read_data_out = 32'b0;
    ALUresult_out = 32'b0;
    write_register_out = 5'b0;
    Rd_out = 5'b0;
end

always @(posedge clk)begin
    WB_out = WB_in;
    Read_data_out = Read_data_in;
    ALUresult_out = ALUresult_in;
    write_register_out = write_register_in;
    Rd_out = Rd_in;
end


endmodule
```

MEM_WB 為 pipeline 中最後一個 register，負責把 ALU 的計算結果或 Memory 的讀取資料給送回 Register 供寫入使用。

## 十三、MUX5b

```
module MUX5b(
    input [4:0] data1,
```

```
    input [4:0] data2,
    input select,
    output [4:0] data_o
);

assign data_o = select?data2:data1;

endmodule
```

MUX5b 是一個 5bit，雙輸入、單輸出的多工器。

十四、MUX32_3to1

```
module MUX32_3to1(
    input [31:0] data1,
    input [31:0] data2,
    input [31:0] data3,
    input [1:0]select,

    output reg [31:0] data_o

);
always @(*)begin
    data_o = select[1]? data3 : (select[0]?data2:data1);
end
endmodule
```

MUX32_3to1 是一個 32bit，3 輸入、1 輸出的多工器。

十五、MUX32b

```
module MUX32b(
    input [31:0] data1,
    input [31:0] data2,
    input select,
    output [31:0] data_o
);

assign data_o = select?data2:data1;
// assign data_o = data1;

endmodule
```

MUX32b 是一個 32bit，雙輸入、單輸出的多工器。

十六、PC

```
module PC(
    input clk,
    input [31:0]address_in,
    input enable,

    output reg [31:0]address_o
);
initial begin
    address_o = 32'b0;
end

always@(posedge clk)begin
    if(enable)begin
    address_o <= address_in;
    end
end
// assign address_o = enable ? address_in:address_o;
endmodule
```

PC 為 Program Counter，透過 Hazard Detection 模組可以控制 PC 是否要讀入下一個指令，其透過正緣的 CLK 觸發。

十七、PipelineCPU

```
module PipelineCPU(
    input [31:0]Addr_in,
    input clk,
    output [31:0]Addr_o
);

wire [31:0]instr_Addr;
wire PCWrite;
PC pc(
    .clk(clk),
    .address_in(Addr_in),
    .enable(PCWrite),
    .address_o(instr_Addr)
);
```

```
wire [31:0]instr;
IM im(
	.Addr_in(instr_Addr),
	.instr(instr)
);


Adder adder(
	.data1(instr_Addr),
	.data2(32'd4),
	.data_o(Addr_o)
);

//////////////////////////////////////////
wire IF_IDWrite;
wire [31:0]instr_o;
IF_ID if_id(
	.clk(clk),
	.enable(IF_IDWrite),
	.instruction_in(instr),
	.instruction_out(instr_o)
);
wire [4:0]RT_addr;
wire ID_EX_MemRead;
wire stall;
wire ID_EX_RegisterRt;
wire IF_ID_RegisterRs;
wire IF_ID_RegisterRt;
assign ID_EX_RegisterRt = RT_addr;
assign IF_ID_RegisterRs = instr[25:21];
assign IF_ID_RegisterRt = instr[20:16];

Hazard_detection hz_det(
	.clk(clk),
	.ID_EX_MemRead(ID_EX_MemRead),
	.ID_EX_RegisterRt(ID_EX_RegisterRt),
	.IF_ID_RegisterRs(IF_ID_RegisterRs),
	.IF_ID_RegisterRt(IF_ID_RegisterRt),
	.stall_mux(stall),
```

```
    .IF_ID_Write(IF_IDWrite),
    .PC_Write(PCWrite)
);


wire RegDst;
wire MemRead;
wire MemtoReg;
wire [2:0]ALUOp;
wire MemWrite;
wire ALUSrc;
wire RegWrite;

Control control(
    .Op(instr[31:26]),
    .RegDst(RegDst),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .ALUOp(ALUOp),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .RegWrite(RegWrite)
);

wire RegDst_o;
wire MemRead_o;
wire MemtoReg_o;
wire [2:0]ALUOp_o;
wire MemWrite_o;
wire ALUSrc_o;
wire RegWrite_o;

wire [22:0]tmp;
MUX32b stall_mux(
    .data1({23'b0,RegDst,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite}),
    .data2(32'b0),
    .select(stall),
    .data_o({tmp,RegDst_o,MemRead_o,MemtoReg_o,ALUOp_o,MemWrite_o,ALUSrc_o,R
egWrite_o})
```

```
);

wire [1:0]WB_ooo;
wire [4:0]WB_addr_oo;
wire [31:0]WB_data;
wire [31:0]read1;
wire [31:0]read2;

RF rf(
	.clk(clk),
	.RegWrite(WB_ooo[1]),
	.RSaddr(instr[25:21]),
	.RTaddr(instr[20:16]),
	.RDaddr(WB_addr_oo),
	.RDdata(WB_data),

	.RTdata(read2),
	.srcl(read1)
);

wire [31:0]SE_o;
SE se(
	.data_i(instr[15:0]),
	.data_o(SE_o)
);

////////////////////////////////////////////

wire [1:0]WB_o;
wire [1:0]MEM_o;
wire [4:0]EX_o;
wire [31:0]SE_oo;
wire [4:0]RS_addr;
wire [4:0]RD_addr;
wire [31:0]read1_o;
wire [31:0]read2_o;
wire [4:0]shamt_o;
```

```
ID_EX id_ex(
      .clk(clk),
      .WB_in({RegWrite_o,MemtoReg_o}),
      .M_in({MemWrite_o,MemRead_o}),
      .EX_in({RegDst_o,ALUSrc_o,ALUOp_o}),
            //     [4]RegDst [3]ALUSrc     [2:0] ALUOp
      .RTdata_in(read2),
      .srcl_in(read1),
      .shamt_in(instr[10:6]),
      .se_in(SE_o),
      // .R_add_in(),
      // .I_add_in(),
      .Rs_in(instr[25:21]),
      .Rt_in(instr[20:16]),
      .Rd_in(instr[15:11]),

      .WB_out(WB_o),
      .M_out(MEM_o),
      .EX_out(EX_o),
      .RTdata_out(read2_o),
      .srcl_out(read1_o),
      .shamt_out(shamt_o),
      .se_out(SE_oo),
      // .R_add_out()
      // .I_add_out()
      .Rs_out(RS_addr),
      .Rt_out(RT_addr),
      .Rd_out(RD_addr)
);


assign ID_EX_MemRead = MEM_o[0];

wire [31:0]ALU_o;
wire [1:0]ALU_src1;
wire [31:0]ALU_in1;

MUX32_3to1 mux32_3to1A(
      .data1(read1_o),
```

```
        .data2(WB_data),
        .data3(ALU_o),
        .select(ALU_src1),
        .data_o(ALU_in1)
);

wire [1:0]ALU_src2;
wire [31:0]To_MEM;
MUX32_3to1 mux32_3to1B(
        .data1(read2_o),
        .data2(WB_data),//換
        .data3(ALU_o), //  換
        .select(ALU_src2),
        .data_o(To_MEM)
);

wire [31:0]ALU_in2;
MUX32b mux32_2to1(
        .data1(To_MEM),
        .data2(SE_oo),
        .select(EX_o[3]),
        .data_o(ALU_in2)
);

wire [4:0]WB_addr;
MUX5b mux5_2to1(
        .data1(RT_addr),
        .data2(RD_addr),
        .select(EX_o[4]),
        .data_o(WB_addr)
);

wire [5:0]ALU_Op;
ALUctrl aluctrl(
        .funct(SE_oo[5:0]),
        .ALUOp(EX_o[2:0]),
        .operation(ALU_Op)
);
```

```
wire [31:0]ALU_out;
ALU alu(
        .shamt(shamt_o),
        .src1(ALU_in1),
        .src2(ALU_in2),
        .operation(ALU_Op),
        .result(ALU_out)
);

wire [4:0]WB_addr_o;
wire [1:0]WB_oo;
Forwarding fw(
        .clk(clk),
        .ID_EX_Rs(RS_addr),
        .ID_EX_Rt(RT_addr),
        .EX_MEM_Rd(WB_addr_o),
        .MEM_WB_Rd(WB_addr_oo),
        .EX_MEM_RegWrite(WB_oo[1]),
        .MEM_WB_RegWrite(WB_ooo[1]),

        .ForwardA(ALU_src1),
        .ForwardB(ALU_src2)
);

////////////////////////////////////////////

wire [1:0]MEM_oo;
// wire [31:0]ALU_o;
wire [31:0]in_MEM;
EX_MEM ex_mem(
        .clk(clk),
        .WB_in(WB_o),
        .M_in(MEM_o),
        .ALUresult_in(ALU_out),
        .write_mem_data_in(To_MEM),
        .write_register_in(WB_addr),
        // .Rd_in()
```

```
    .WB_out(WB_oo),
    .M_out(MEM_oo),
    .ALUresult_out(ALU_o),
    .write_mem_data_out(in_MEM),
    .write_register_out(WB_addr_o)
);

wire [31:0]MEM_data;
DM dm(
    .clk(clk),
    .addr(ALU_o),
    .data(in_MEM),
    .MemRead(MEM_oo[0]),
    .MemWrite(MEM_oo[1]),
    .DM_data(MEM_data)
);

////////////////////////////////////
wire [31:0]MEM_data_o;
wire [31:0]ALU_oo;

MEM_WB mem_wb(
    .clk(clk),
    .WB_in(WB_oo),
    .Read_data_in(MEM_data),
    .ALUresult_in(ALU_o),
    .write_register_in(WB_addr_o),
    // .RD_in()

    .WB_out(WB_ooo),
    .Read_data_out(MEM_data_o),
    .ALUresult_out(ALU_oo),
    .write_register_out(WB_addr_oo)
);

MUX32b mux32b_memtoreg(
    .data1(ALU_oo),
```

```
    .data2(MEM_data_o),
    .select(WB_ooo[0]),
    .data_o(WB_data)
);


endmodule
```

PipelineCPU 實作題目中的 Part3 所有功能，即 R-type 指令、lw、sw、Hazard 的 stall 與 Forwarding 等所有功能，將上述的模組全部組起來即可完成。

十八、RF

```
module RF(input clk,
            input RegWrite,
            input [4:0] RSaddr,
            input [4:0] RTaddr,
            input [4:0] RDaddr,
            input [31:0] RDdata,
            output reg [31:0] RTdata,
            output reg [31:0] srcl);



    reg [31:0]REGISTER[31:0];

    initial begin
        REGISTER[0]  = 32'd0;
        REGISTER[1]  = 32'd11;
        REGISTER[2]  = 32'd370;
        REGISTER[3]  = 32'd183;
        REGISTER[4]  = 32'd91;
        REGISTER[5]  = 32'd234;
        REGISTER[6]  = 32'd53;
        REGISTER[7]  = 32'd127;
        REGISTER[8]  = 32'd317;
        REGISTER[9]  = 32'd179;
        REGISTER[10] = 32'd101;
        REGISTER[11] = 32'd161;
        REGISTER[12] = 32'd152;
        REGISTER[13] = 32'd39;
        REGISTER[14] = 32'd39;
```

```
            REGISTER[15] = 32'd44;
            REGISTER[16] = 32'd29;
            REGISTER[17] = 32'd334;
            REGISTER[18] = 32'd245;
            REGISTER[19] = 32'd19;
            REGISTER[20] = 32'd2;
            REGISTER[21] = 32'd13;
            REGISTER[22] = 32'd262;
            REGISTER[23] = 32'd185;
            REGISTER[24] = 32'd180;
            REGISTER[25] = 32'd180;
            REGISTER[26] = 32'd198;
            REGISTER[27] = 32'd178;
            REGISTER[28] = 32'd235;
            REGISTER[29] = 32'd22;
            REGISTER[30] = 32'd1000;
            REGISTER[31] = 32'd75;
        end
        //
        always@(posedge clk or RSaddr or RTaddr or RDaddr )begin
            RTdata <= REGISTER[RTaddr];
            srcl     <= REGISTER[RSaddr];
        end
        //negedge
        always@(clk)begin
            if (RegWrite)begin
                REGISTER[RDaddr] <= RDdata;
            end
        end
endmodule
```

RF 為 Register File，預設的內部儲存值與 Project2 相同。

## 十九、SE

```
module SE(
    input [15:0] data_i,
    output [31:0] data_o
);
assign data_o = data_i[15]?{16'hffff,data_i}:{16'h0000,data_i};
```

```
endmodule
```

SE 可以將 16bit 的有號數數值轉換成 32bit 的有號數數值。

## 參、模擬結果

### 一、Testbench

```
`timescale 1ns / 1ps

module PipelineCPU_tb;

    // Inputs
    reg [31:0] Addr_in;
    reg clk;

    // Outputs
    wire [31:0] Addr_o;

    // Instantiate the Unit Under Test (UUT)
    PipelineCPU uut (
        .Addr_in(Addr_in),
        .clk(clk),
        .Addr_o(Addr_o)
    );

    initial begin
        // Initialize Inputs
        Addr_in = 0;
        clk = 0;

        // Wait 100 ns for global reset to finish
        #700
        $finish;

        // Add stimulus here
    end

    always begin
```

```
    #10 clk <= ~clk;
    #10 clk <= ~clk;
        Addr_in <= Addr_o;
  end

endmodule
```

我將助教提供的 Testbench，總時間縮短了一點，因為後半段程式碼執行完畢後的分析較無意義。

二、模擬結果 階段 1

(一) add $t0, $t1, $t2

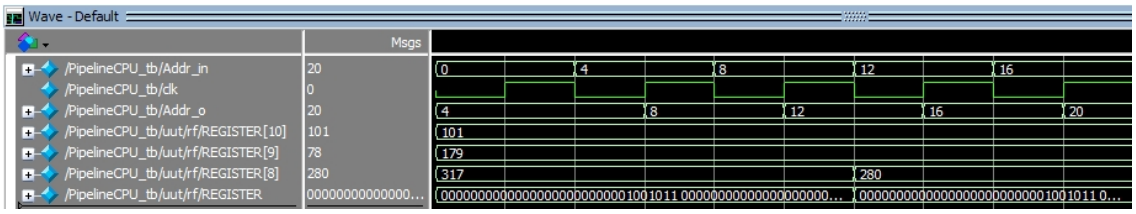
1.將$t1 的值與$t2 相加，放到$t0 中

2.$t1 的位子是 9；$t2 的位子是 10；$t0 的位子是 8

3.$t1 的值是 179；$t2 的值是 101；$t0 的值經過運算後是 280

4.本指令不會發生 Data Hazard

5.模擬結果圖




(二) sub $t1, $t1, $t2

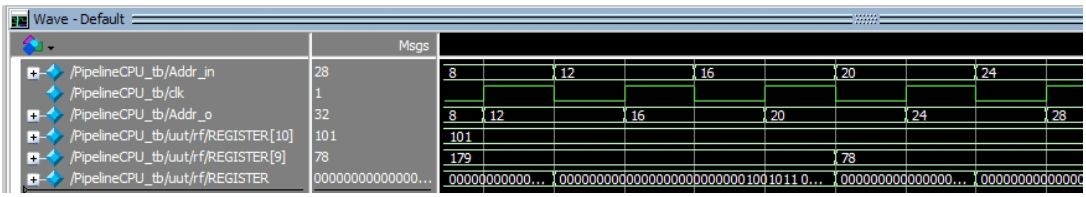
1.將$t1 的值與$t2 相減，放到$t1 中

2.$t1 的位子 9 是；$t2 的位子是 10；$t1 的位子是 9

3.$t1 的值是 179；$t2 的值是 101；$t1 的值經過運算後是 78

4.本指令不會發生 Data Hazard

5.模擬結果圖




(三) shr $t3, $t4, 2

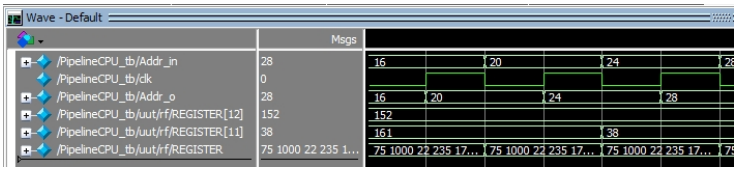
1.將$t4 的值右移 2 格，放到$t3 中

2.$t4 的位子是 12；$t3 的位子是 11

3.$t4 的值是 152； $t3 的值經過運算後是 38

4.本指令不會發生 Data Hazard

5.模擬結果圖




(四) shl $t5, $t7, 5

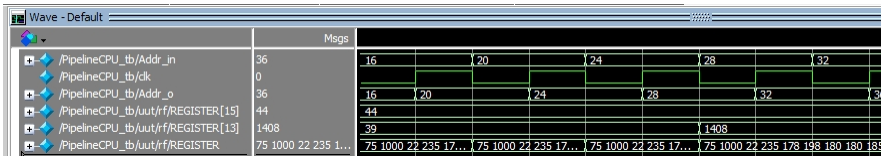
1.將$t7 的值左移 5 格，放到$t5 中

2.$t7 的位子是 15；$t5 的位子是 13

3.$t7 的值是 44； $t5 的值經過運算後是 1408

4.本指令不會發生 Data Hazard

5.模擬結果圖




三、模擬結果 階段 2

(一) sw $t0, 2($t3)

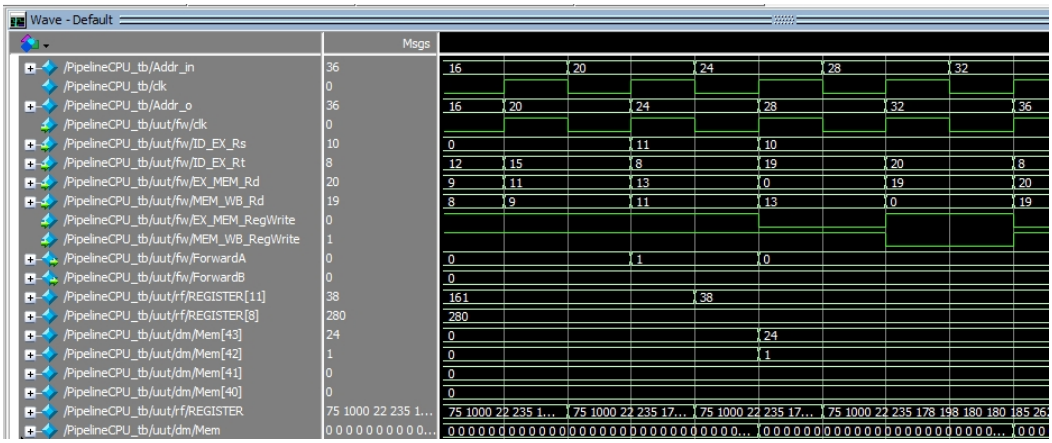
1.將$t0 的值放到記憶體($t3 內的值+2)的位址

2.$t0 的位址為 8；記憶體($t3 內的值+2)的位址為 40

3.$t0 的值為 280；記憶體該位置經過運算後的值為 280

4.本指令會發生 Data Hazard，因為前兩個指令 shr 的輸出放在$t3，而本指令需要讀取$t3 的值，因此需要透過 Forwarding 來解

5.模擬結果圖




(二) lw $s3, 2($t2)

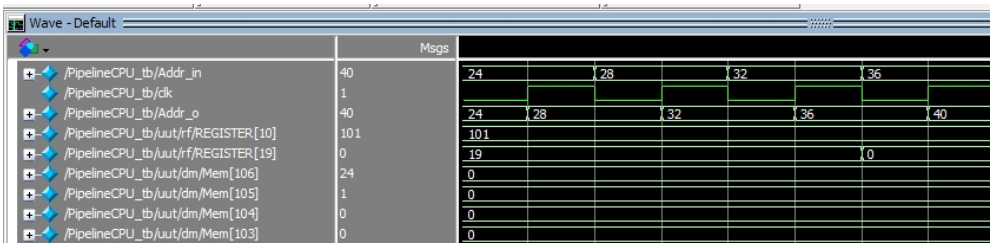
1.將記憶體 ($t2 內的值+2)位址的值取出，放置到$s3

2.記憶體 ($t2 內的值+2)的位址為 103；$s3 的位址為 19

3.記憶體該位置的值為 0；$s3 經過運算後的值為 0

4.本指令不會發生 Data Hazard

5.模擬結果圖




(三) sw $s4, 4($t2)

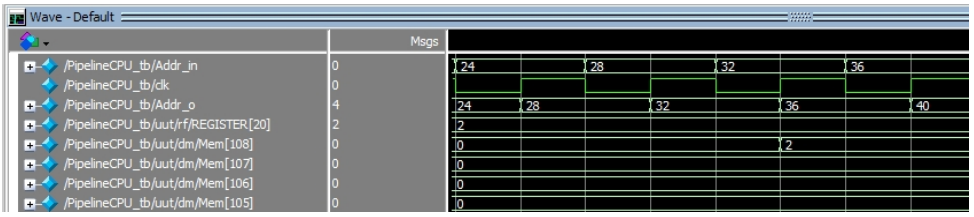
1.將$s4 的值放到記憶體($t2 內的值+4)的位址

2.$s4 的位址為 20；記憶體($t2 內的值+4)的位址為 105

3.$s4 的值為 2；記憶體該位置經過運算後的值為 2

4.本指令不會發生 Data Hazard

5.模擬結果圖




(四) sw $t0, 2($t2)

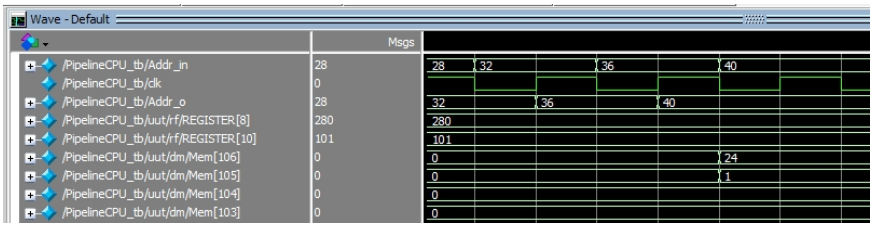
1.將$t0 的值放到記憶體($t2 內的值+2)的位址

2.$t0 的位址為 8；記憶體($t2 內的值+2)的位址為 103

3.$t0 的值為 280；記憶體該位置經過運算後的值為 280

4.本指令不會發生 Data Hazard

5.模擬結果圖




(五) lw $s4, 3($t3)

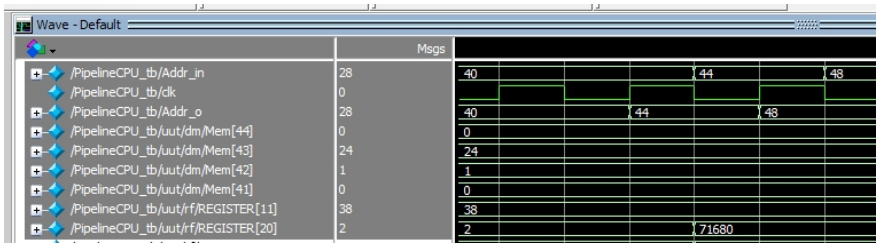
1.將記憶體 ($t3 內的值+3)位址的值取出，放置到$s4

2.記憶體 ($t3 內的值+3)的位址為 41；$s4 的位址為 20

3.記憶體該位置的值為 71680；$s4 經過運算後的值為 71680

4.本指令不會發生 Data Hazard

5.模擬結果圖




四、模擬結果 階段 3

(一) add $t6, $t5, $t4

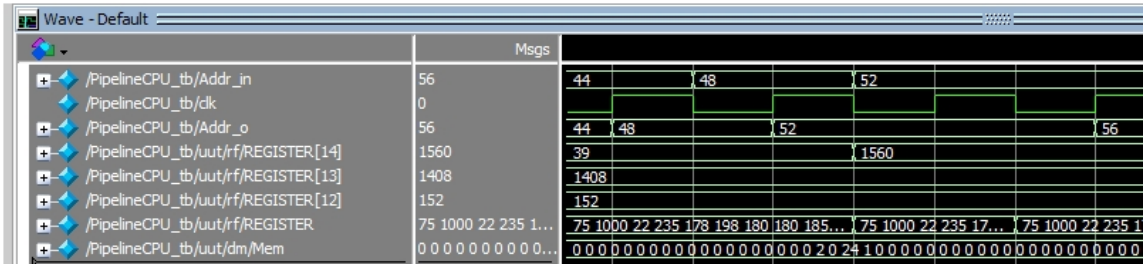
1.將$t5 的值與$t4 相加，放到$t6 中

2.$t5 的位子是 13；$t4 的位子是 12；$t6 的位子是 14

3.$t5 的值是 1408；$t4 的值是 152；$t6 的值經過運算後是 1560

4.本指令不會發生 Data Hazard

5.模擬結果圖




(二) sub $t7, $t6, $t5

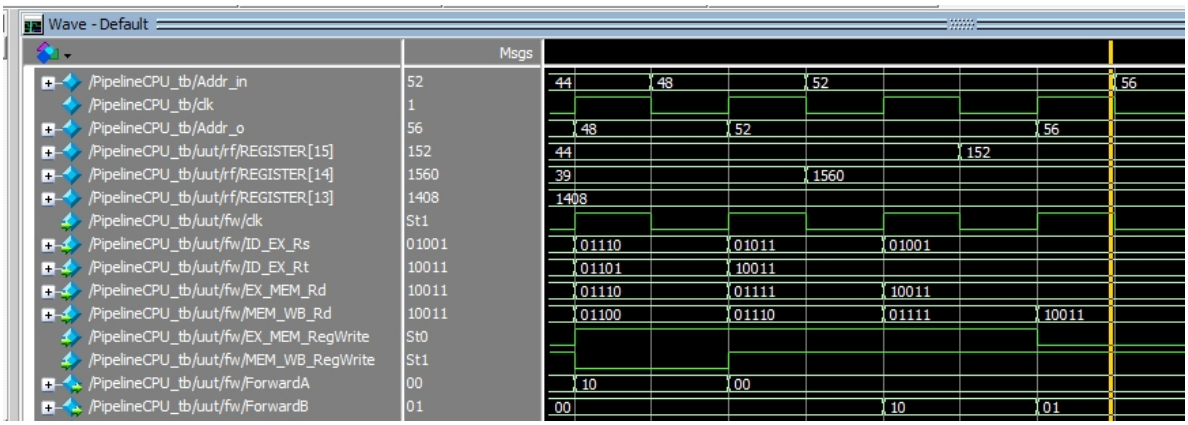
1.將$t6 的值與$t5 相加，放到$t7 中

2.$t6 的位子是 14；$t5 的位子是 13；$t7 的位子是 15

3.$t6 的值是 1560；$t5 的值是 1408；$t7 的值經過運算後是 152

4.本指令會發生 Data Hazard，因為前一個指令的 add 結果存在 t6，所以需要透過 Forwarding 的方式處理

5.模擬結果圖



(三) lw $s3, 2($t3)

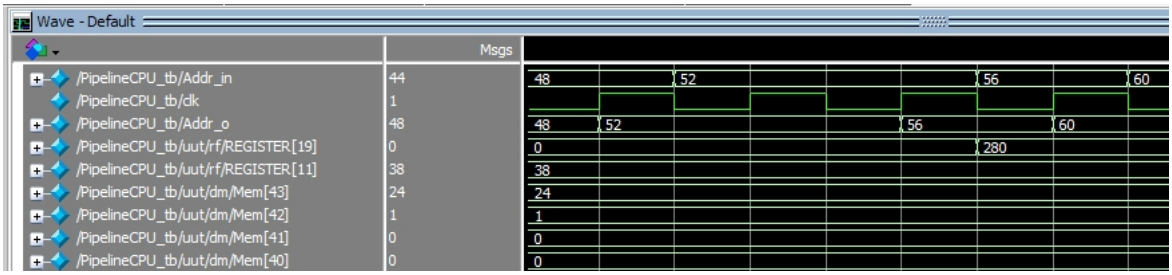
1.將記憶體 ($t3 內的值+2)位址的值取出，放置到$s3

2.記憶體 ($t3 內的值+2)的位址為 40；$s3 的位址為 19

3.記憶體該位置的值為 280；$s3 經過運算後的值為 280

4.本指令不會發生 Data Hazard

5.模擬結果圖




(四) sw $s3, 2($t1)

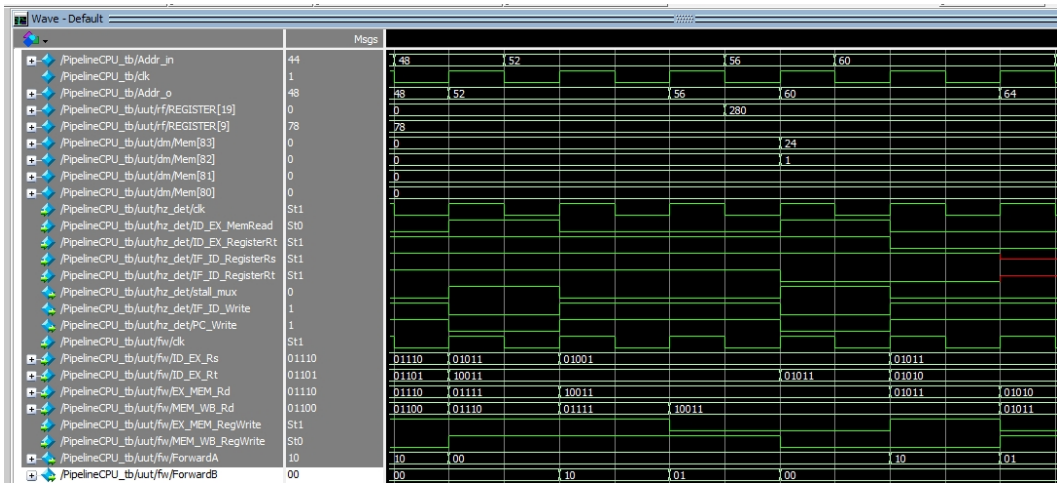
1.將$s3 的值放到記憶體($t1 內的值+2)的位址

2.$s3 的位址為；記憶體($t1 內的值+2)的位址為 80

3.$s3 的值為 280；記憶體該位置經過運算後的值為 280

4.本指令會發生 Data Hazard，因為前一個指令 lw 需要將值存到$s3 內，需要透過 Stall 來解。

5.模擬結果圖




(五) lw $t3, 2($t1)

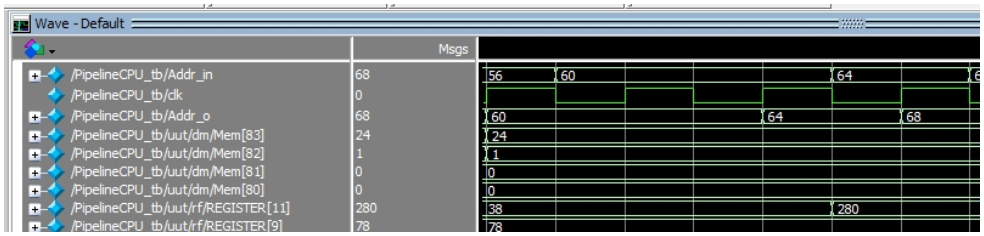
1.將記憶體 ($t1 內的值+2)位址的值取出，放置到$t3

2.記憶體 ($t1 內的值+2)的位址為 80；$t3 的位址為 11

3.記憶體該位置的值為 280；$t3 經過運算後的值為 280

4.本指令不會發生 Data Hazard

5.模擬結果圖




(六) add $t3, $t3, $t2

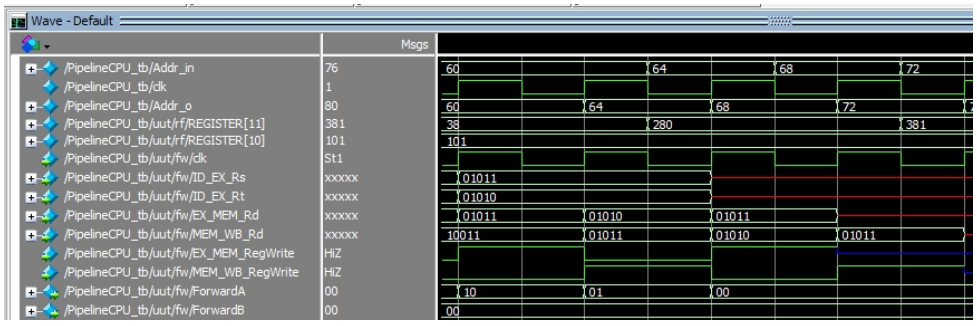
1.將$t3 的值與$t2 相加，放到$t3 中

2.$t3 的位子是 11；$t2 的位子是 10；$t3 的位子是 11

3.$t3 的值是 280；$t2 的值是 101；$t3 的值經過運算後是 381

4.本指令會發生 Data Hazard，因為前一個指令的 lw 結果存在$t3，所以需要透過 Forwarding 的方式處理

5.模擬結果圖



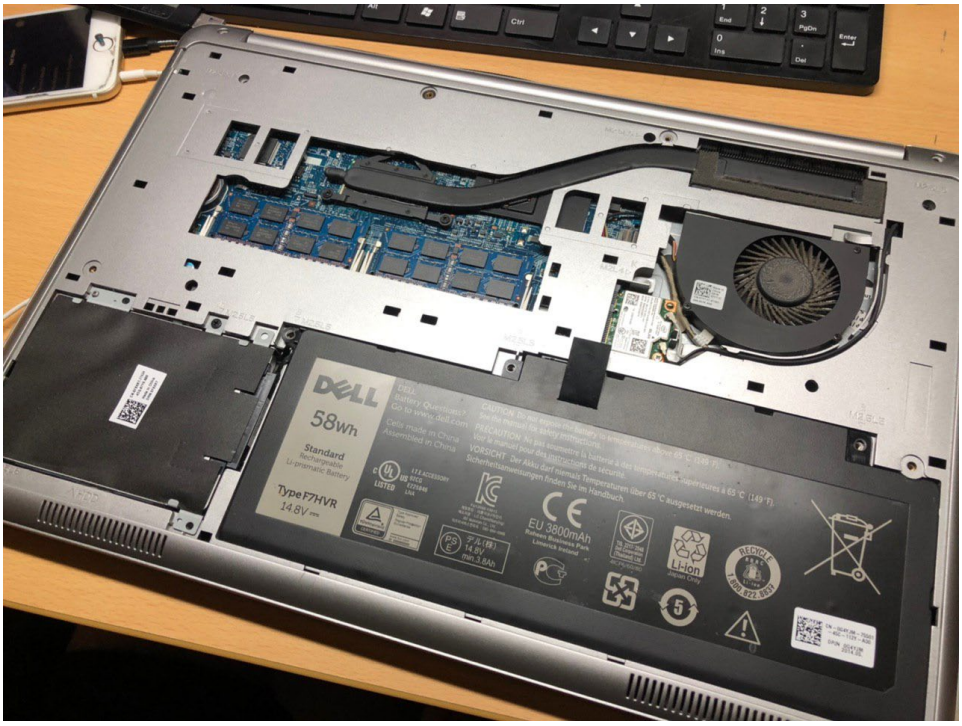

## 肆、心得

讀了題目發現這一次的報告好像沒有心得的分數耶，可是我連標題都已經打了，那就留下來吧！所以既然不算分的話那我也可以亂寫心得對吧？嘻嘻嘻！

好吧，還是正經一點寫一點正常的心得，這一次的報告真的比上一次的難度高了不少，雖然單論程式碼的行數其實並不多，但是由於 pipeline 同時執行多個指令，導致 Debug 的難度相當的高，讓我學習到了這種複雜狀態的 Debug 方法。我花最多時間除錯的地方是 Hazard 判斷，原先我透過 posedge 的 clk 來做觸發，研究了許久後發現，這邊的判斷並不需要與時脈相關，這次 Project 真的讓我學習到了非常多。

而做這次專案的過程中，也遇到了一個插曲，當我這份報告寫到一半的時候，我的筆電不知名的螢幕突然黑掉，無論充電燈、硬碟燈，任何燈號都沒有反應，但是 USB 卻是有電的狀況，在即將交報告的前一天晚上發

生這種事情，我急得快要哭出來了，後來嘗試用手邊的工具把電腦拆開。




發現電池鬆掉了 QQ，還好後來終於把它修好，才完成了這份艱難的 Project。謝謝天；謝謝地；謝謝蜂蜜檸檬；謝謝雅淑教授；謝謝助教辛苦的改作業，最後，謝謝我這台辛苦的電腦。