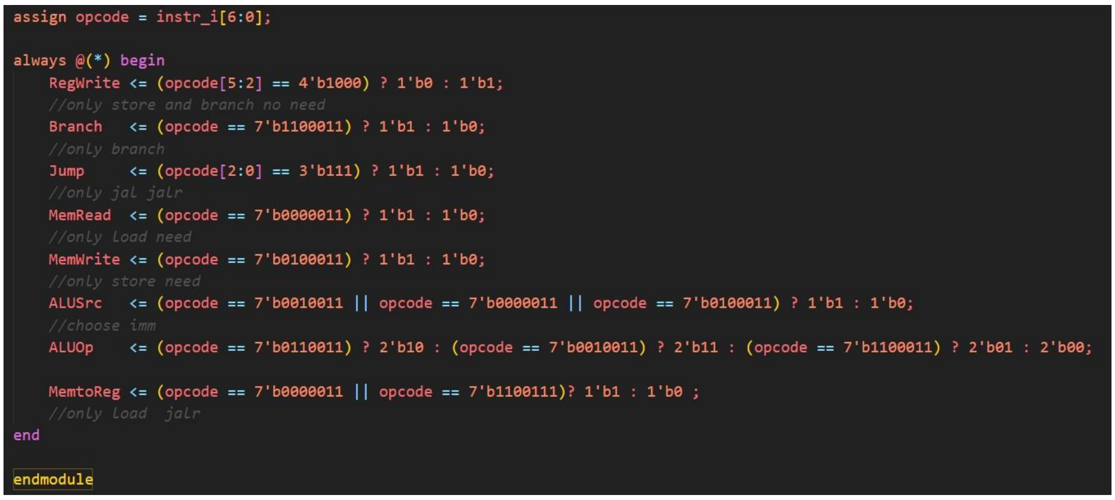
LAB 5 : 5-Stage Pipeline Processor

109550167 陳唯文

Implementation details

Decoder.v



```verilog
assign opcode = instr_i[6:0];

always @(*) begin
    RegWrite <= (opcode[5:2] == 4'b1000) ? 1'b0 : 1'b1;
    //only store and branch no need
    Branch   <= (opcode == 7'b1100011) ? 1'b1 : 1'b0;
    //only branch
    Jump     <= (opcode[2:0] == 3'b111) ? 1'b1 : 1'b0;
    //only jal jalr
    MemRead  <= (opcode == 7'b0000011) ? 1'b1 : 1'b0;
    //only load need
    MemWrite <= (opcode == 7'b0100011) ? 1'b1 : 1'b0;
    //only store need
    ALUSrc   <= (opcode == 7'b0010011 || opcode == 7'b0000011 || opcode == 7'b0100011) ? 1'b1 : 1'b0;
    //choose imm
    ALUOp    <= (opcode == 7'b0110011) ? 2'b10 : (opcode == 7'b0010011) ? 2'b11 : (opcode == 7'b1100011) ? 2'b01 : 2'b00;

    MemtoReg <= (opcode == 7'b0000011 || opcode == 7'b1100111)? 1'b1 : 1'b0 ;
    //only load  jalr
end

endmodule
```

The difference in decoder.v compared to lab4 is ALUSrc becomes one output and delete writeback. writback become memtoreg which is controlled by the last MUX_3to1 in pipeline_cpu.v, ALUSrc A control jalr and this part is now controlled by IF.flush in the first MUX_2to1 in pipeline_cpu.v.

Pipeline_CPU.v

按照 spec 提供的圖還有參考 chap4-part3 page5 的圖實作 IF.Flush, chap4-part2 page15 實作 MUX2to1 MUX_control.

Adder.v

```
module Adder(
    input  [32-1:0] src1_i,
    input  [32-1:0] src2_i,
    output reg [32-1:0] sum_o
);

/* Write your code HERE */
always @(*) begin
    sum_o <= src1_i + src2_i;
end


endmodule
```

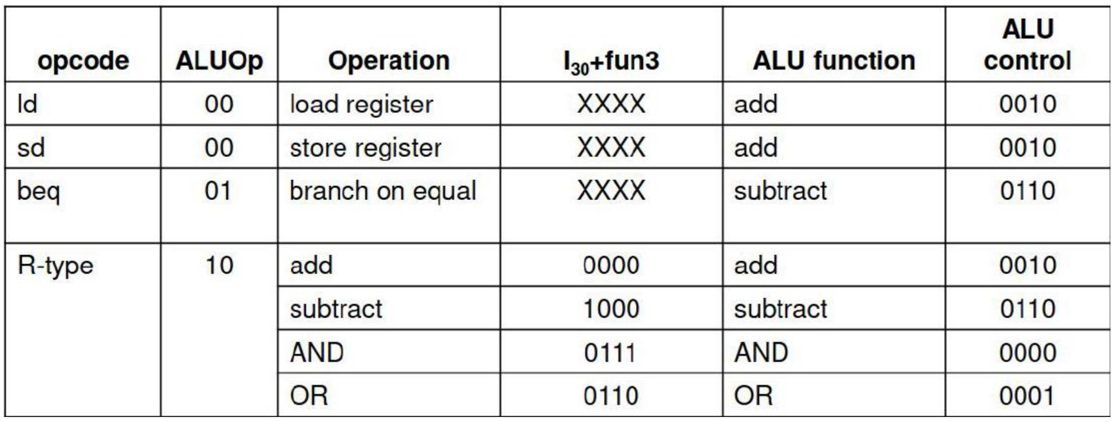
使用 operator 完成加法。

ALU_Ctrl.v

```
parameter aluadd = 4'b0010;
parameter alusub = 4'b0110;
parameter aluand = 4'b0000;
parameter aluor  = 4'b0001;
parameter aluslt = 4'b0111;
parameter aluxor = 4'b1000;
parameter alusll = 4'b1010;
```

將變數名稱和值設定好，方便接下來的 code。

```
always @(*) begin
        case (func3)
                3'b000  : alufunct3 <= aluadd; //addi
                3'b001  : alufunct3 <= alusll; //slli
                3'b010  : alufunct3 <= aluslt; //slti
        endcase
end

always @(*) begin
        case (instr)
                4'b0000 : alufunc <= aluadd;
                4'b1000 : alufunc <= alusub;
                4'b0111 : alufunc <= aluand;
                4'b0110 : alufunc <= aluor;
                4'b0010 : alufunc <= aluslt;
                4'b0100 : alufunc <= aluxor;
        endcase
end

always @(*) begin
        case (ALUOp)
                2'b00: ALU_Ctrl_o <= aluadd;
                2'b01: ALU_Ctrl_o <= alusub;
                2'b10: ALU_Ctrl_o <= alufunc;
                2'b11: ALU_Ctrl_o <= alufunct3;
                default: ALU_Ctrl_o <= 4'b1111;
        endcase
end
```



opcode	ALUOp	Operation	I_{30}+fun3	ALU function	ALU control
ld	00	load register	XXXX	add	0010
sd	00	store register	XXXX	add	0010
beq	01	branch on equal	XXXX	subtract	0110
R-type	10	add	0000	add	0010
		subtract	1000	subtract	0110
		AND	0111	AND	0000
		OR	0110	OR	0001

由於只有 R-type 指令除了 ALUOp 外，還需要 ALU control 的判斷，所以這裡拆成兩部分寫。第一個 always 的 alufunc 即為在 R-type 指令下，對於不同 ALU control 的判斷和相對應的動作，第二個 always 寫的則是 ALUOp 對應到不同 type 的指令需要輸出的動作。

Func3 的部分則是用來判斷 I-type 指令。

ALU.v

```
assign zero = (result == 32'b0);

always @(*) begin
    if(~rst_n) begin
        result <= 32'b0;
    end
    else begin
        case (ALU_control)
            4'b0010: result <= src1 + src2;
            4'b0110: result <= src1 - src2;
            4'b0000: result <= src1 & src2;
            4'b0001: result <= src1 | src2;
            4'b0111: result <= {31'b0, (src1 < src2)};
            4'b1000: result <= src1 ^ src2;   //xor
            4'b1010: result <= src1 << src2; //sll
            default: result <= result;
        endcase
    end
end
```

使用 operator 完成 and, or, add, sub, slt, xor, sll 的指令

Hazard_detection.v

```
always @(*) begin
    if(IDEXE_memRead && (IDEXE_regRd == IFID_regRs || IDEXE_regRd == IFID_regRt)) begin
        PC_write              <= 1'b0;
        IFID_write            <= 1'b0;
        control_output_select <= 1'b1;
    end
    else begin
        PC_write              <= 1'b1;
        IFID_write            <= 1'b1;
        control_output_select <= 1'b0;
    end
end
endmodule
```

這裡負責解決 load use 時會發生的 data hazard。用 memRead 判斷是否為 load 指令，當指令為 Load 且當前在 ID 階段的 rs 或 rt 等於 load 指令的 rd 時，將 PC_write, IFID_write 設為 0, control_output_select 設為 1，這樣在下一個 cycle 時即可產生 nop，避免 data hazard 的情況。

Forwarding.v

```verilog
always @(*) begin
    if(EXEMEM_RegWrite && EXEMEM_RD != 5'b0 && (EXEMEM_RD == IDEXE_RS1))begin
        ForwardA <= 2'b10;
    end
    else if(MEMWB_RegWrite && MEMWB_RD != 5'b0 && (EXEMEM_RD != IDEXE_RS1) && (MEMWB_RD == IDEXE_RS1))begin
        ForwardA <= 2'b01;
    end
    else begin
        ForwardA <= 2'b00;
    end

    if(EXEMEM_RegWrite && EXEMEM_RD != 5'b0 && (EXEMEM_RD == IDEXE_RS2))begin
        ForwardB <= 2'b10;
    end
    else if(MEMWB_RegWrite && MEMWB_RD != 5'b0 && (EXEMEM_RD != IDEXE_RS2) && (MEMWB_RD == IDEXE_RS2))begin
        ForwardB <= 2'b01;
    end
    else begin
        ForwardB <= 2'b00;
    end

end
endmodule
```

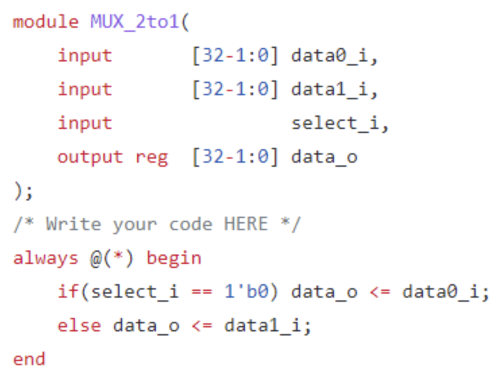
用來判斷 ALU 用到的 rs 跟 rt 跟前面兩個 instruction 的 rd 有沒有 dependency 且會 register write。如果 rs 或 rt 會用到前一個指令的 rd，輸出 2’b10，如果是用到再前一個指令的 rd 則輸出 2’b01，如果兩個指令皆有使用到，優先度是 2’b10 大於 2’b01。

Imm_Gen.v

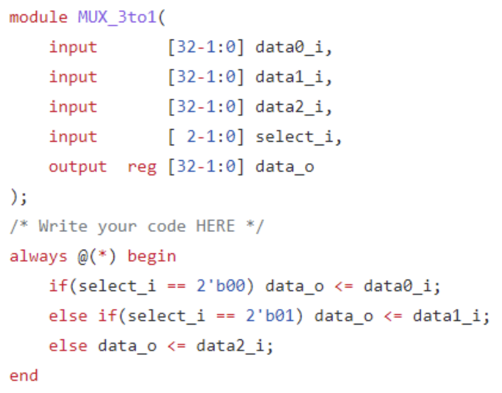
```verilog
wire    [7-1:0] opcode;
wire    [2:0]   func3;
wire    [3-1:0] Instr_field;

assign opcode = instr_i[6:0];
assign func3  = instr_i[14:12];

/* Write your code HERE */
always @(*) begin
    case (opcode)
        7'b0110011 : Imm_Gen_o <= 0; //R-type
        7'b0010011 : Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:20]}; //addi
        7'b0000011 : Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:20]}; //lw
        7'b1100111 : Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:20]}; //jalr
        7'b0100011 : Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:25], instr_i[11:7]}; //sw
        7'b1100011 : Imm_Gen_o <= {1'b0, {20{instr_i[31]}}, instr_i[7], instr_i[30:25], instr_i[11:8]}; //beq
        7'b1101111 : Imm_Gen_o <= {1'b0, {12{instr_i[31]}}, instr_i[19:12], instr_i[20] , instr_i[30:21]}; //jal
        default    : Imm_Gen_o <= 0;
    endcase
end
```

依照 immediate 在不同 type 的指令中的位置合併輸出結果，並用最高位的 bit 作為 signed bit 補位。

MUX2to1, Mux3to1.v



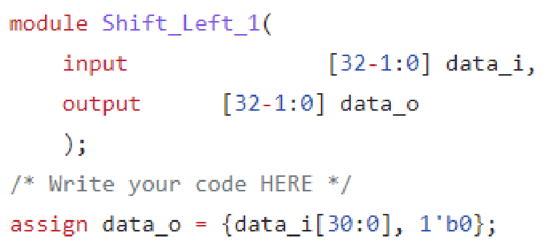
```
module MUX_2to1(
    input       [32-1:0] data0_i,
    input       [32-1:0] data1_i,
    input                select_i,
    output reg  [32-1:0] data_o
);
/* Write your code HERE */
always @(*) begin
    if(select_i == 1'b0) data_o <= data0_i;
    else data_o <= data1_i;
end
```



```
module MUX_3to1(
    input       [32-1:0] data0_i,
    input       [32-1:0] data1_i,
    input       [32-1:0] data2_i,
    input       [ 2-1:0] select_i,
    output  reg [32-1:0] data_o
);
/* Write your code HERE */
always @(*) begin
    if(select_i == 2'b00) data_o <= data0_i;
    else if(select_i == 2'b01) data_o <= data1_i;
    else data_o <= data2_i;
end
```

基本的多工器，依 select 的值決定 data 的輸出。在 MUX_3to1 中，2’b10 和 2’11 都會輸出 data2_i。

Shift_Left_1.v



```
module Shift_Left_1(
    input               [32-1:0] data_i,
    output      [32-1:0] data_o
    );
/* Write your code HERE */
assign data_o = {data_i[30:0], 1'b0};
```

輸出原本的 0-30 位並在最低位補上 0 相當於向左移一位。

IFID_register.v

```verilog
always @(posedge clk_i) begin
    if(~rst_i)begin
        address_o <= 32'b0;
        instr_o   <= 32'b0;
        pc_add4_o <= 32'b0;
    end
    else if(flush)begin
        address_o <= 32'b0;
        instr_o   <= 32'b00000000000000000000000000010011; // addi x0 x0 0
        pc_add4_o <= 32'b0;
    end
    else if(IFID_write)begin
        address_o <= address_i;
        instr_o   <= instr_i;
        pc_add4_o <= pc_add4_i;
    end
    else begin
        address_o <= address_o;
        instr_o   <= instr_o;
        pc_add4_o <= pc_add4_o;
    end
end
endmodule
```

初始值皆設為 0。
當 flush=1，即為 branch 或是 jump 發生，此時輸出指令為 addi x0, x0, 0，代表 nop。
當 IFID_write=0，代表出現 load use，於是將指令內容儲存。

IDEXE_register.v

```
always @(posedge clk_i) begin
    if(~rst_i)begin
        instr_o         <= 0;
        WB_o            <= 0;
        Mem_o           <= 0;
        Exe_o           <= 0;
        data1_o         <= 0;
        data2_o         <= 0;
        immgen_o        <= 0;
        alu_ctrl_input  <= 0;
        WBreg_o         <= 0;
        pc_add4_o       <= 0;
    end
    else begin
        instr_o         <= instr_i;
        WB_o            <= WB_i;
        Mem_o           <= Mem_i;
        Exe_o           <= Exe_i;
        data1_o         <= data1_i;
        data2_o         <= data2_i;
        immgen_o        <= immgen_i;
        alu_ctrl_input  <= alu_ctrl_instr;
        WBreg_o         <= WBreg_i;
        pc_add4_o       <= pc_add4_i;
    end
end
endmodule
```

初始值為 0，其他輸出=輸入。

EXEMEM_register.v

```
always @(posedge clk_i) begin
    if(~rst_i)begin
        instr_o   <= 0;
        WB_o      <= 0;
        Mem_o     <= 0;
        zero_o    <= 0;
        alu_ans_o <= 0;
        rtdata_o  <= 0;
        WBreg_o   <= 0;
        pc_add4_o <= 0;
    end
    else begin
        instr_o   <= instr_i;
        WB_o      <= WB_i;
        Mem_o     <= Mem_i;
        zero_o    <= zero_i;
        alu_ans_o <= alu_ans_i;
        rtdata_o  <= rtdata_i;
        WBreg_o   <= WBreg_i;
        pc_add4_o <= pc_add4_i;
    end
end
endmodule
```

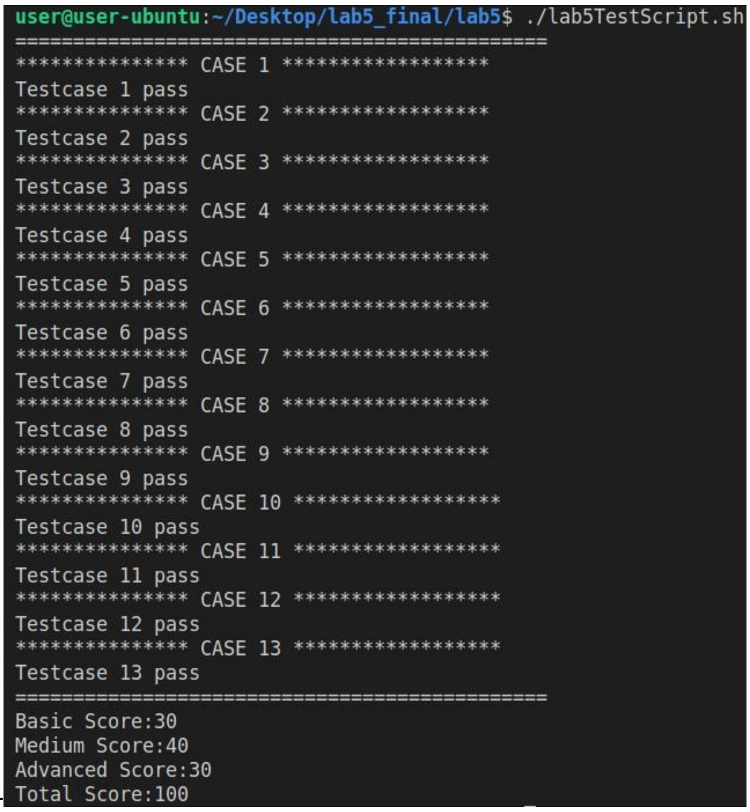
初始值為 0，其他輸出=輸入。

MEMWB_register.v

```
always @(posedge clk_i) begin
    if(~rst_i)begin
        WB_o      <= 0;
        DM_o      <= 0;
        alu_ans_o <= 0;
        WBreg_o   <= 0;
        pc_add4_o <= 0;
    end
    else begin
        WB_o      <= WB_i;
        DM_o      <= DM_i;
        alu_ans_o <= alu_ans_i;
        WBreg_o   <= WBreg_i;
        pc_add4_o <= pc_add4_i;
    end
end
endmodule
```

初始值為 0，其他輸出=輸入。

Implementation Result



```
user@user-ubuntu:~/Desktop/lab5_final/lab5$ ./lab5TestScript.sh
==============================================
*************** CASE 1 ******************
Testcase 1 pass
*************** CASE 2 ******************
Testcase 2 pass
*************** CASE 3 ******************
Testcase 3 pass
*************** CASE 4 ******************
Testcase 4 pass
*************** CASE 5 ******************
Testcase 5 pass
*************** CASE 6 ******************
Testcase 6 pass
*************** CASE 7 ******************
Testcase 7 pass
*************** CASE 8 ******************
Testcase 8 pass
*************** CASE 9 ******************
Testcase 9 pass
*************** CASE 10 ******************
Testcase 10 pass
*************** CASE 11 ******************
Testcase 11 pass
*************** CASE 12 ******************
Testcase 12 pass
*************** CASE 13 ******************
Testcase 13 pass
==============================================
Basic Score:30
Medium Score:40
Advanced Score:30
Total Score:100
```

Problem

In this lab, we implement a 5-stage pipeline cpu and we do encounter some problems while coding.
First, when implementing the forwarding unit, I first look up chap 4-part2 slides and try to bring the same idea into our cpu, using a lot of if to check which control message we need to send. But There is a bug in our code, which is that if we put `(EXEMEM_RD != IDEXE_RS1) && (MEMWB_RD == IDEXE_RS1)` inside another if expression, I will miss the situation which is supposed to let ForwardA/B to be 2'b00. So I put it with MEMWB_RegWrite and MEMWB_RD together and fixed the bug successfully.
Second, In alu ctrl.v,when I first implement this unit , I forgot difference between my function code and instr code, so I always thought there was some okward problem in alu.v. After I checked the alu ctrl.v, the problem was spotted clearly and fixed.
Third, In pipeline.v, the most difficult part is filling in the mux 2 to 1 after MUX_CONTROL, I have to figure out each of the control signals linked to which unit. If I put them in a bad sequence, I will have a hard time filling the later blank.

Experience

109550167 陳唯文
這次的 lab 應該是是從之前到現在需要花最多時間的一次，最麻煩的部分我認為是 pipeline cpu 的部分，因為這邊不僅需要對整個 pipeline processor 有一定程度的了解，還需要花很多時間在完成 function 上，而且大部分的 function 都是環環相扣的，在整理輸入和輸出上就需要許多時間，甚至有些 function 在一開始並不清楚它的作用，但完成之後確實能夠更了解整個 pipeline processor 的結構。

109550164 徐聖哲
這次的 lab 讓我們能實際了解 load use 還有 hazard detection 是怎麼執行的，看到透過精巧設計的 detection，可以有效解決 load use 還有 branch hazard.