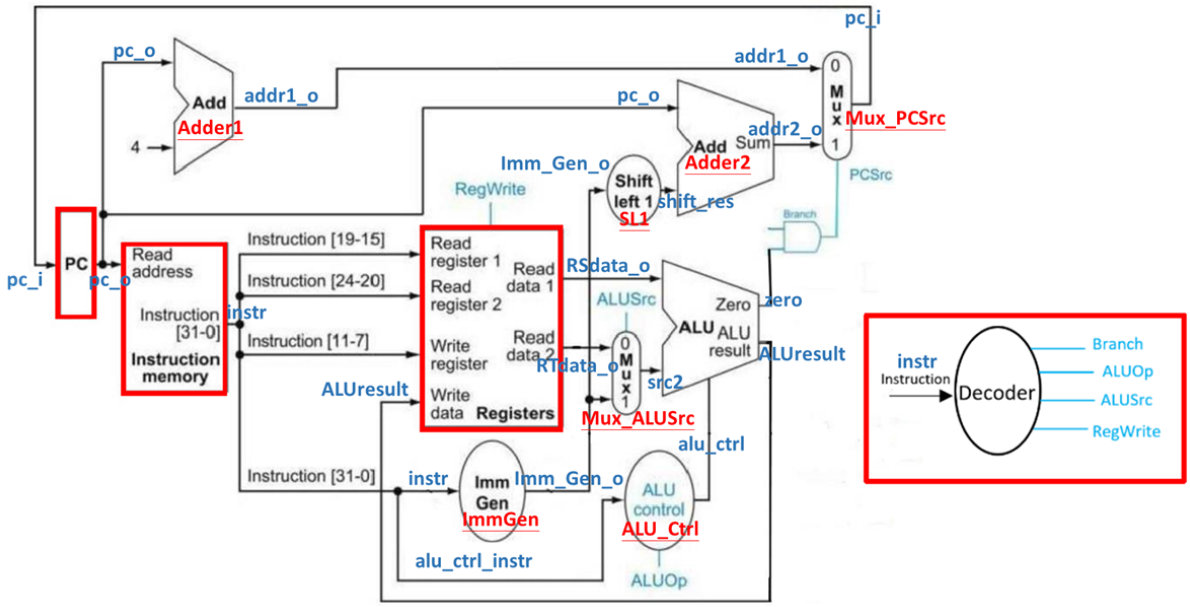
Lab 3: Single-Cycle CPU (Simple Version)

姓名：葉致廷/林廷澔
學號：0846013/0846011

Architecture diagram

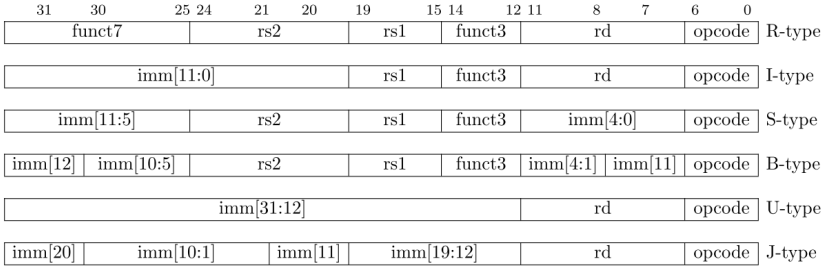


Detailed description of the implementation

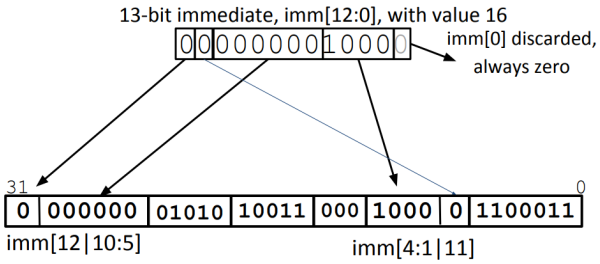
- ***Imm_Gen Module***
  - **Description**

    ImmGen會根據不同type的opcode來決定如何組出Immediate value。根據查表opcode在Instruction的0-6bit，因此會先取出opcode：`wire [7-1:0] opcode = instr_i[6:0];`。之後依據Fig1來組成type的Immediate value。舉例來說：SB Type，opcode為`7'b1100011`，組成為：`Imm_Gen_out <= {{20{instr_i[31]}},instr_i[31],instr_i[7],instr_i[30:25],instr_i[11:8]};`。參考Fig2，Immediate value依LSB⇒MSB分別填入Instruction的8-11bit⇒Instruction的25-30bit⇒Instruction的7bit⇒Instruction的31bit⇒最後20位為Instruction的31bit extension。

  - **Different type instruction construction_Fig1**



- **SB type immediate value construction_Fig2**



- **Source Code**

```
module Imm_Gen(input [31:0] instr_i,
               output [31:0] Imm_Gen_o);

    /* Write your code HERE */
    reg [31:0] Imm_Gen_out;
    wire [7-1:0] opcode = instr_i[6:0];

   always@ *
    begin
    case(opcode)
        // R_type
        7'b0110011: Imm_Gen_out <= instr_i;
        // Load
        7'b0000011:
        begin
            Imm_Gen_out <= {{20{instr_i[31]}},instr_i[31:20]};
        end
        // STORE
        7'b0100011:
        begin
            Imm_Gen_out <= {{20{instr_i[31]}},instr_i[31:25],instr_i[11:7]};
        end
        // I_type Imm
        7'b0010011:
        begin
            Imm_Gen_out <= {{20{instr_i[31]}},instr_i[31:20]};
        end
        // SB
        7'b1100011:
        begin
            Imm_Gen_out <= {{20{instr_i[31]}},instr_i[31],instr_i[7],instr_i[30:25],instr_i[11:8]};
        end

    endcase
    end

    assign Imm_Gen_o = Imm_Gen_out;

endmodule
```

- ***Shift_Left_1 Module***
  - **Description**

    此Module功能單純，就是把input(也就是Immediate value)左移一位即可，操作上取32bit input的[30:0]，然後LSB補0即可：`assign data_o = {data_i[30:0], 1'b0};`

  - **Source Code**

```
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};

endmodule
```

- ***ALU_Ctrl Module***
  - **Description**

    此Module主要是透過輸入的ALUOp，與Instruction3/4個bit，來輸出我們自定義的ALU Control Input。相關定義如下：

    💡 ALUOp Type
    00：Load/Store/Immediate
    01：SB Type
    10：R Type

    💡 Different Action depend on instruction, customized define the ALU Control Input

    ALUOp Type：0 ⇒ depend on first 3 bit of instruction
    000 ⇒ ADDI ⇒ output：2
    010 ⇒ SLTI ⇒ output：7
    100 ⇒ XORI ⇒ output：13
    110 ⇒ ORI ⇒ output：1
    111 ⇒ ANDI ⇒ output：0
    001 ⇒ SLLI ⇒ output：3
    101 ⇒ SRLI ⇒ output：5

    ALUOp Type：1 ⇒ depend on first 3 bit of instruction
    000 ⇒ BEQ ⇒ output：10
    001 ⇒ BNE ⇒ output：11

    ALUOp Type：2 ⇒ depend on first 4 bit of instruction
    0111 ⇒ AND ⇒ output：0
    0110 ⇒ OR ⇒ output：1
    0000 ⇒ ADD ⇒ output：2
    0001 ⇒ SLL ⇒ output：3
    0101 ⇒ SRL ⇒ output：4
    0010 ⇒ SLT ⇒ output：7
    0100 ⇒ XOR ⇒ output：13
    1000 ⇒ SUB⇒ output：6
    1101 ⇒ SRA ⇒ output：5

  - **Source Code**

```
module ALU_Ctrl(input [4-1:0] instr,
                input [2-1:0] ALUOp,
                output  [4-1:0] ALU_Ctrl_o);
```

```
    /* Write your code HERE */
    reg [4-1:0] ALU_Ctrl_out;

    always@*
    begin
    case(ALUOp)
        /* Load, Store, Imm*/
        2'b00:
        begin
            case(instr[3-1:0])
                //ADDI do add
                3'b000: ALU_Ctrl_out <= 4'b0010;
                //SLTI do slt
                3'b010: ALU_Ctrl_out <= 4'b0111;
                //XORI do xor:13
                3'b100: ALU_Ctrl_out <= 4'b1101;
                //ORI do or
                3'b110: ALU_Ctrl_out <= 4'b0001;
                //ANDI do and
                3'b111: ALU_Ctrl_out <= 4'b0000;
                //SLLI do sll:3
                3'b001: ALU_Ctrl_out <= 4'b0011;
                //SRAI do sra:5
                3'b101: ALU_Ctrl_out <= 4'b0101;
            endcase
        end
        /* SB */
        2'b01:
        begin
            case(instr[3-1:0])
                //BEQ do beq:10
                3'b000: ALU_Ctrl_out <= 4'b1010;
                //BNE do bne:11
                3'b001: ALU_Ctrl_out <= 4'b1011;
            endcase
        end
        /* R */
        2'b10:
        begin
            case(instr)
                //AND do and: 0
                4'b0111: ALU_Ctrl_out <= 4'b0000;
                //OR do  or: 1
                4'b0110: ALU_Ctrl_out <= 4'b0001;
                //ADD do add: 2
                4'b0000: ALU_Ctrl_out <= 4'b0010;
                //SLL do sll: 3
                4'b0001: ALU_Ctrl_out <= 4'b0011;
                //SRL do srl: 4
                4'b0101: ALU_Ctrl_out <= 4'b0100;
                //SLT do slt: 7
                4'b0010: ALU_Ctrl_out <= 4'b0111;
                //XOR do xor :13
                4'b0100: ALU_Ctrl_out <= 4'b1101;
                //SUB do sub: 6
                4'b1000: ALU_Ctrl_out <= 4'b0110;
                //SRA do sra: 5
                4'b1101: ALU_Ctrl_out <= 4'b0101;
                //NOR: 12

            endcase
        end
    endcase
    end

    assign ALU_Ctrl_o = ALU_Ctrl_out;
endmodule
```

- ***alu Module***
  - **Description**

根據ALU_Ctrl Module所自定義的ALU Control Input，alu Module就要針對這些Control Input做不同的操作。舉例來說，當`ALU_control = 12`，操作動作為`NOR`，輸出就為：`result <= ~(src1 | src2);`。

- **Source Code**

```
module alu(input rst_n,
           input   [32-1:0] src1,
           input   [32-1:0] src2,
           input   [4-1:0]  ALU_control,
           output reg [32-1:0]  result,
           output reg zero,
           output reg cout,
           output reg overflow);

    /* Write your code HERE */
    wire   [32-1:0] result_wire;
    wire            carryout;
    wire            overflow_wire;
    wire            zeroWire;
    assign zeroWire = (result == 0) ? 1'b1 :1'b0 ;


    always @*
    begin
    if (rst_n) begin
        cout     <= 1'b0; //Don't care
        overflow <= 1'b0; //Don't care

        case(ALU_control)
            /* AND :0*/
            4'b0000:
            begin
                result = src1 & src2;
                zero = zeroWire;
            end
            /* OR :1*/
            4'b0001:
            begin
                result = src1 | src2;
                zero = zeroWire;
            end
            /* ADD :2*/
            4'b0010:
            begin
                result = src1 + src2;
                zero = zeroWire;
            end
            /* SUB :6*/
            4'b0110:
            begin
                result = src1 - src2;
                zero = zeroWire;
            end
            /* SLT :7*/
            4'b0111:
            begin
                result = (src1 < src2)?1:0;
                zero = zeroWire;
            end
            /* NOR :12*/
            4'b1100:
            begin
                result = ~(src1 | src2);
                zero = zeroWire;
            end
            /* XOR :13*/
            4'b1101:
            begin
                result = src1 ^ src2;
                zero = zeroWire;
```

```
            end
            /* SLL :3*/
            4'b0011:
            begin
                result = src1 << src2;
                zero = zeroWire;
            end
            /* SRL :4*/
            4'b0100:
            begin
                result = src1 >> src2;
                zero = zeroWire;
            end
            /* SRA :5*/
            4'b0101:
            begin
                result = src1 >>> src2;
                zero = zeroWire;
            end
            /* BEQ :10*/
            4'b1010:
            begin
                result = src1 - src2;
                zero = zeroWire;
            end
            /* BNE :11*/
            4'b1011:
            begin
                result = (src1 ^ src2) != 32'b0 ? 32'd0: 32'd1;
                zero = zeroWire;
            end

            default: result = 32'd0;

        endcase
    end
    end
endmodule
```

- ***Adder Module***
  - **Description**

    此Module會用在Adder1(Program Counter+4)和Adder2(Program Counter+shift immediate value)中。此模組功能單純，就是把兩輸入相加在輸出而已。

  - **Source Code**

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

    /* Write your code HERE */
    assign  sum_o = src1_i + src2_i;

endmodule
```

- ***MUX_2to1 Module***
  - **Desctription**

    此Module會用在Mux_ALUSrc(由ALUSrc輸入來決定要輸出Read data2或是Immediate value)和Mux_PCSrc(由PCSrc輸入來決定要輸出PC+4還是PC+OFFSET)中。此模組功能單純，根據select_i輸入來決定要輸出data0_i或是data1_i。

  - **Source Code**

```
module MUX_2to1(input [32-1:0] data0_i,
                input [32-1:0] data1_i,
                input   select_i,
                output [32-1:0] data_o);

    reg [32-1:0] data_out;
    /* Write your code HERE */
    always@(*)begin
        if (!select_i) begin//sel = 0 output_Z = input_A
            data_out = data0_i;
        end
        else begin//sel = 1 output_Z = input_B
            data_out = data1_i;
        end
    end

    assign data_o = data_out;

endmodule
```

- ***Simple_Single_CPU Module***
  - **Description**

    Top Module，根據Architecture將各模組的線聯起來即可。

Implementation results

- test_data1_result

```
VSIM 10> run -all
# r0 =   0, r1 =  21, r2 =   9, r3 =   1,
# r4 =  20, r5 =   1, r6 =   0, r7 =   0,
# r8 =   0, r9 =   0, r10 =  0, r11 =  0
# ** Note: $stop    : C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v line 32

VSIM 11>
```

- test_data2_result

```
VSIM 8> run -all
# r0 =   0, r1 =   0, r2 =   0, r3 =   0,
# r4 =   0, r5 =   0, r6 =   2, r7 =   5,
# r8 =   7, r9 =   9, r10 =  0, r11 =  0
# ** Note: $stop    : C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v line 32

VSIM 9>
```

- test_data3_result

```
VSIM 6> run -all
# r0 =   0, r1 =   0, r2 =   0, r3 =   0,
# r4 =   0, r5 =   0, r6 =   0, r7 =   0,
# r8 =   0, r9 =   0, r10 =  2, r11 =  2
# ** Note: $stop    : C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Users/justinyeh1995/Desktop/Computer Organization Lab3/testbench.v line 32

VSIM 7>
```

Problems encountered and solutions

▼ 在alu.v中，原本將 zero 指定為 `zero <= result == 0?:1:0`，但在思考過後發現，zero 跟 result 同時執行，zero 很有可能因此永遠是32'b0，只是這次測資剛好不會影響到計算結果而已。 解法為目前的寫法，用 assign 的方式，讓 result 算完能夠改變 zeroWire 進而改變 zero 使得最終的結果與我們預期的一致，或直接改成src1 與 src2 判斷比較，如此一來可讓 zero 的結果不用依賴result，去除這個問題。

Comment

▼ Decoder matching 經過耗時的比對與修正才讓結果正確。而ALU_Control 的做法了解到老師上課說的從opcode到ALUOp到ALU Control的過程究竟是甚麼意思。是兩件這次學到最多的事情。原本讀這段時一直不確定opcode是怎麼產生ALUOp的，也不知道instruction[30|14:12]的用意，這次一個一個比對才找到其中的關連。寫完作業有更深一層的理解上課內容。