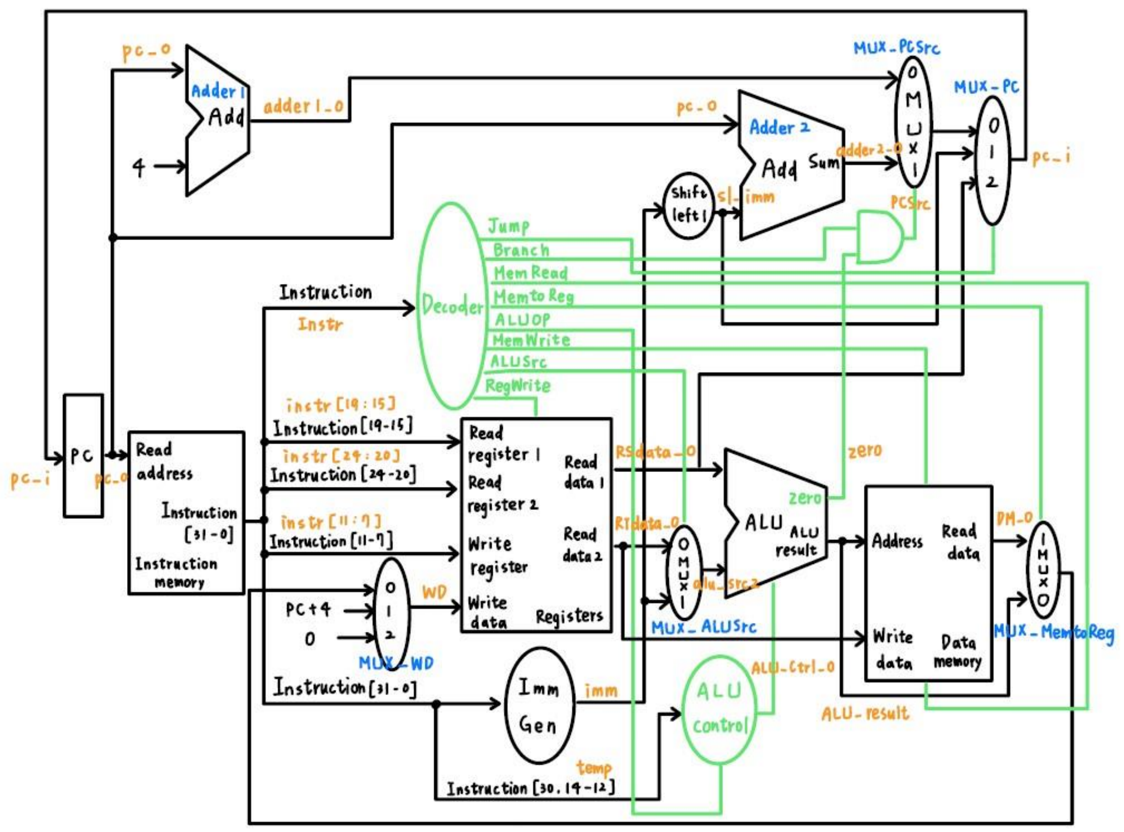
Computer Organization

Architecture diagram:



Detailed description of the implementation:

- Imm_Gen

  讀進整個 instruction 之後拆解出 opcode 的部分，利用 opcode 判斷指令的 type，因為目前只需要判斷指令的 type 就可以抓出需要的 bit 組合出 immediate，要記得做 sign extension。

```
case(opcode)
    7'b0010011: //I type
        Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:20]};
    7'b0000011: //lw
        Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:20]};
    7'b0100011: //S type
        Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:25], instr_i[11:7]};
    7'b1100011: //B type
        Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31], instr_i[7], instr_i[30:25], instr_i[11:8]};
    7'b1101111: //J type
        Imm_Gen_o <= {{12{instr_i[31]}}, instr_i[31], instr_i[19:12], instr_i[20], instr_i[30:21]};
endcase
```

- Shift_Left_1

```
assign data_o = data_i << 1;
```

- MUX_2to1

根據 select_i 決定要傳出去的是 data0_i 還是 data1_i。

```
if(select_i)
    data_o <= data1_i;
else
    data_o <= data0_i;
```

- MUX_3to1

根據 select_i 決定要傳出去的是 data0_i、data1_i 還是 data2_i。

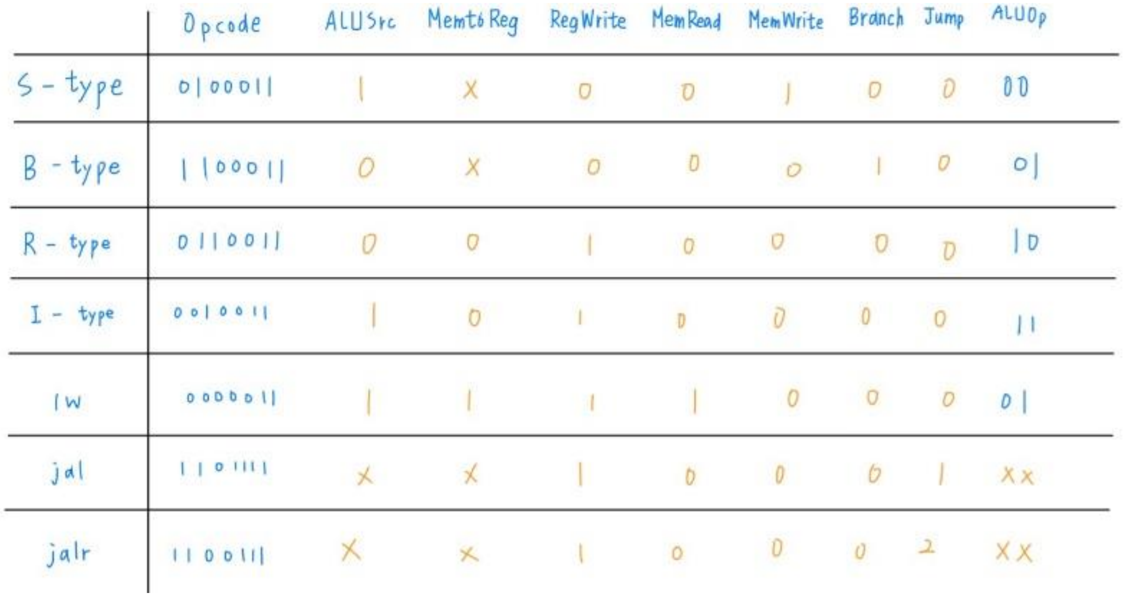
```
if(select_i == 2'b00)
    data_o <= data0_i;
else if(select_i == 2'b01)
    data_o <= data1_i;
else
    data_o <= data2_i;
```

- Adder

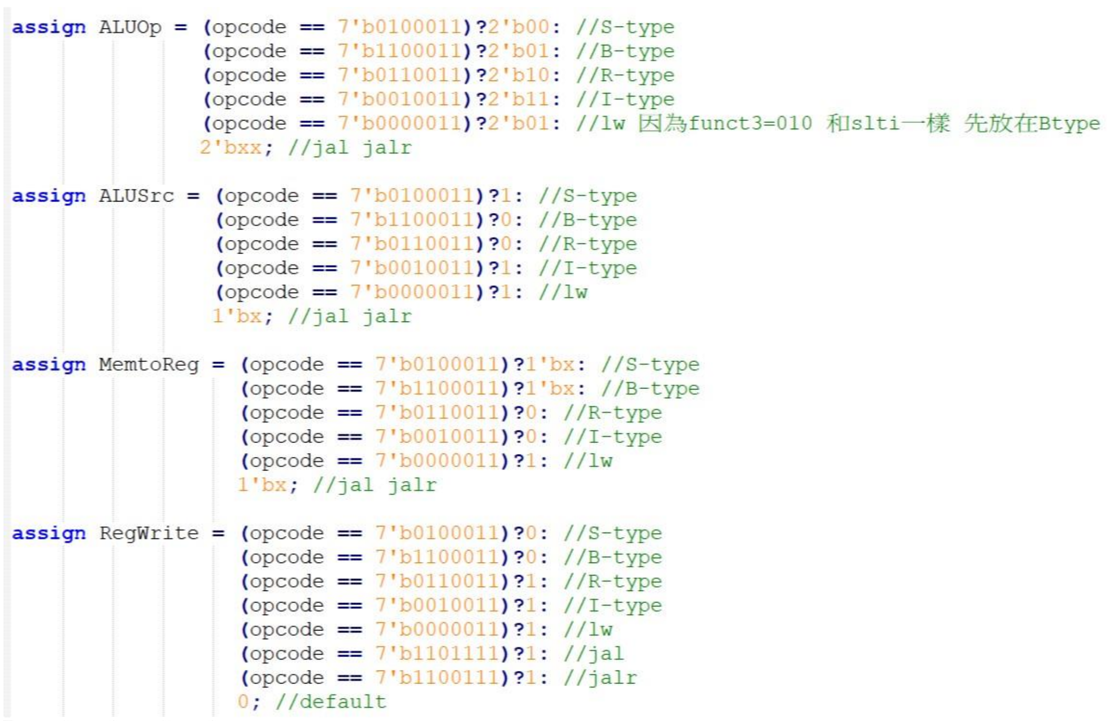
```
assign sum_o = src1_i + src2_i;
```

- Decoder

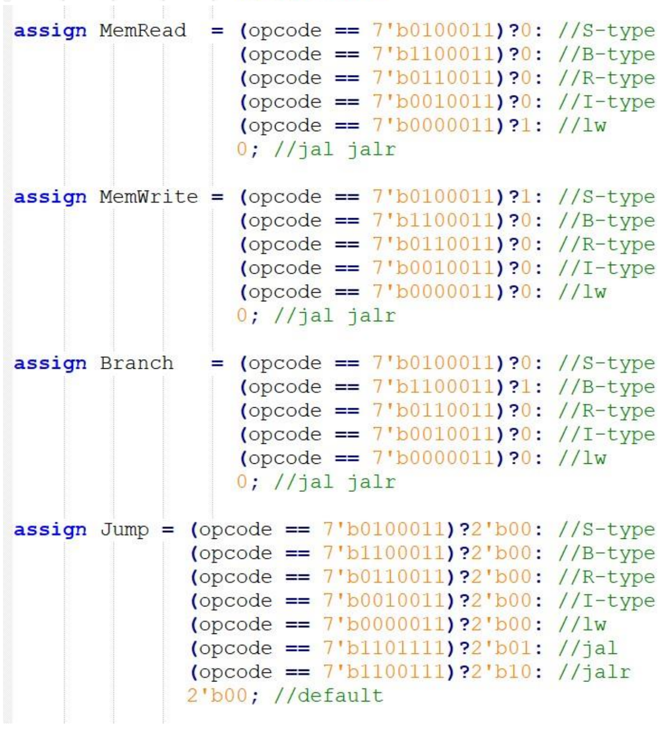
讀進整個 instruction 之後拆解出 opcode 的部分，利用 opcode 判斷指令的 ALUOp、ALUSrc、MemtoReg、RegWrite、MemRead、MemWrite、Branch、Jump。



	Opcode	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALUOp
S-type	0100011	1	X	0	0	1	0	0	00
B-type	1100011	0	X	0	0	0	1	0	01
R-type	0110011	0	0	1	0	0	0	0	10
I-type	0010011	1	0	1	0	0	0	0	11
lw	0000011	1	1	1	1	0	0	0	01
jal	1101111	X	X	1	0	0	0	1	XX
jalr	1100111	X	X	1	0	0	0	2	XX



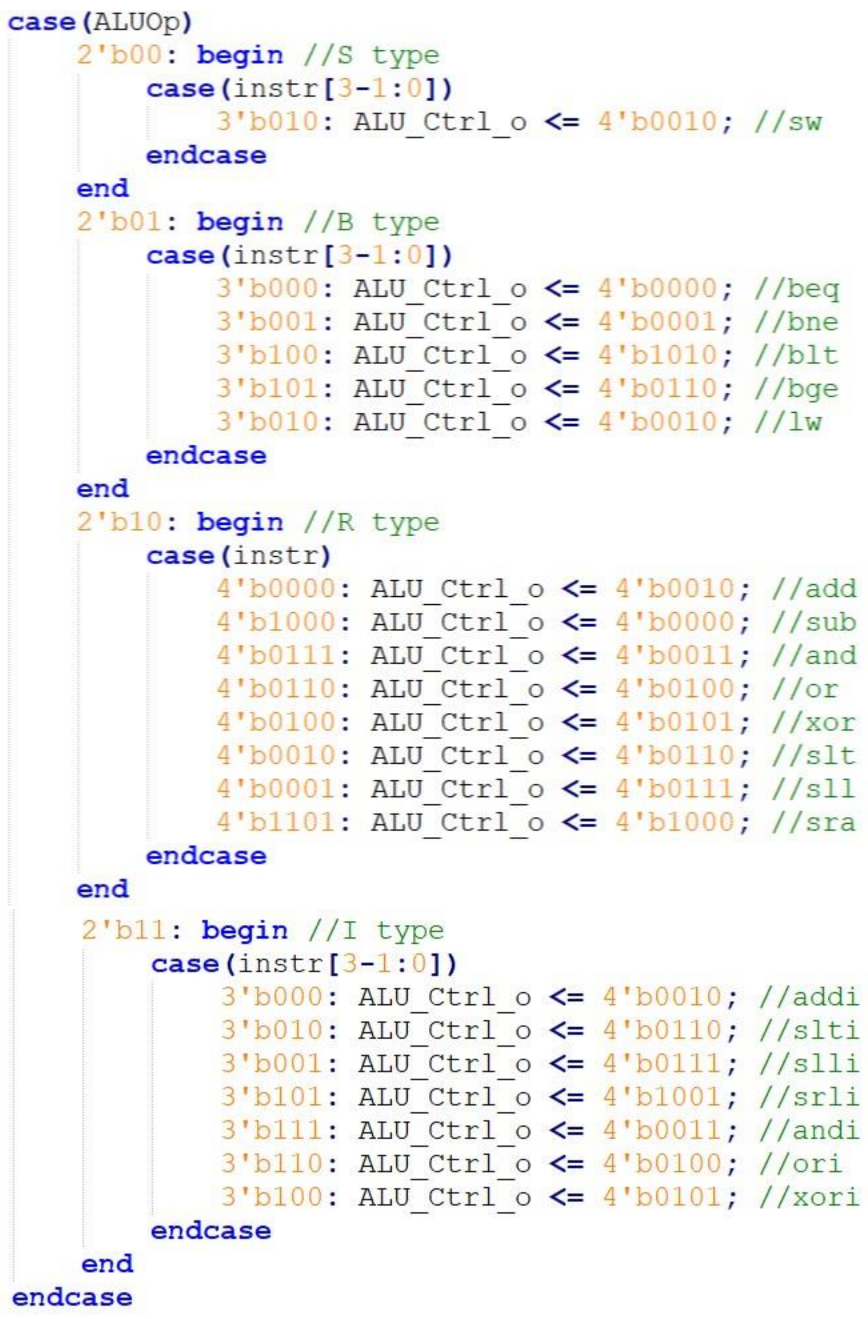
```
assign ALUOp = (opcode == 7'b0100011)?2'b00: //S-type
               (opcode == 7'b1100011)?2'b01: //B-type
               (opcode == 7'b0110011)?2'b10: //R-type
               (opcode == 7'b0010011)?2'b11: //I-type
               (opcode == 7'b0000011)?2'b01: //lw 因為funct3=010 和slti一樣 先放在Btype
               2'bxx; //jal jalr

assign ALUSrc = (opcode == 7'b0100011)?1: //S-type
                (opcode == 7'b1100011)?0: //B-type
                (opcode == 7'b0110011)?0: //R-type
                (opcode == 7'b0010011)?1: //I-type
                (opcode == 7'b0000011)?1: //lw
                1'bx; //jal jalr

assign MemtoReg = (opcode == 7'b0100011)?1'bx: //S-type
                  (opcode == 7'b1100011)?1'bx: //B-type
                  (opcode == 7'b0110011)?0: //R-type
                  (opcode == 7'b0010011)?0: //I-type
                  (opcode == 7'b0000011)?1: //lw
                  1'bx; //jal jalr

assign RegWrite = (opcode == 7'b0100011)?0: //S-type
                  (opcode == 7'b1100011)?0: //B-type
                  (opcode == 7'b0110011)?1: //R-type
                  (opcode == 7'b0010011)?1: //I-type
                  (opcode == 7'b0000011)?1: //lw
                  (opcode == 7'b1101111)?1: //jal
                  (opcode == 7'b1100111)?1: //jalr
                  0; //default
```



```
assign MemRead  = (opcode == 7'b0100011)?0: //S-type
                  (opcode == 7'b1100011)?0: //B-type
                  (opcode == 7'b0110011)?0: //R-type
                  (opcode == 7'b0010011)?0: //I-type
                  (opcode == 7'b0000011)?1: //lw
                  0; //jal jalr

assign MemWrite = (opcode == 7'b0100011)?1: //S-type
                  (opcode == 7'b1100011)?0: //B-type
                  (opcode == 7'b0110011)?0: //R-type
                  (opcode == 7'b0010011)?0: //I-type
                  (opcode == 7'b0000011)?0: //lw
                  0; //jal jalr

assign Branch   = (opcode == 7'b0100011)?0: //S-type
                  (opcode == 7'b1100011)?1: //B-type
                  (opcode == 7'b0110011)?0: //R-type
                  (opcode == 7'b0010011)?0: //I-type
                  (opcode == 7'b0000011)?0: //lw
                  0; //jal jalr

assign Jump = (opcode == 7'b0100011)?2'b00: //S-type
              (opcode == 7'b1100011)?2'b00: //B-type
              (opcode == 7'b0110011)?2'b00: //R-type
              (opcode == 7'b0010011)?2'b00: //I-type
              (opcode == 7'b0000011)?2'b00: //lw
              (opcode == 7'b1101111)?2'b01: //jal
              (opcode == 7'b1100111)?2'b10: //jalr
              2'b00; //default
```

- ALU_Ctrl

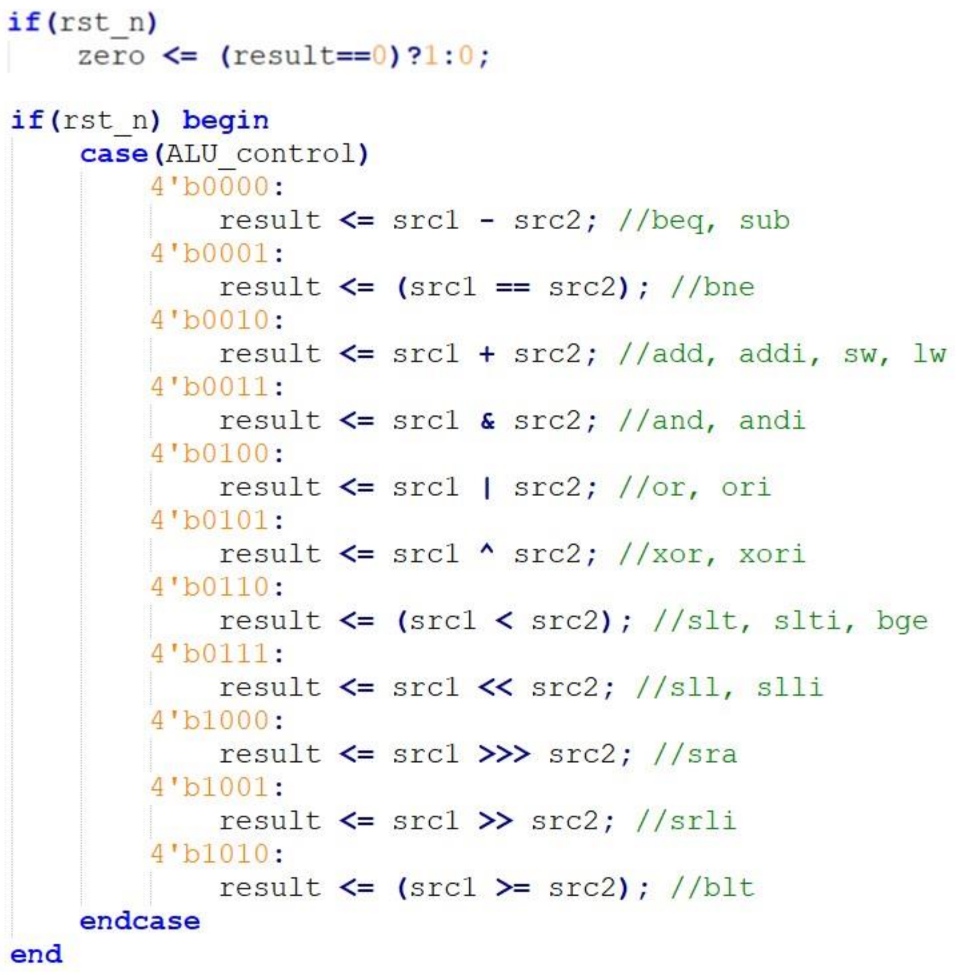
根據傳進來的 4 個 bit 的 instr 和 ALUOp 輸出 ALU_Ctrl_o 控制 ALU，先從 ALUOp 判斷指令的 type 再用 instr 判斷實際上的指令是什麼。



```
case(ALUOp)
    2'b00: begin //S type
        case(instr[3-1:0])
            3'b010: ALU_Ctrl_o <= 4'b0010; //sw
        endcase
    end
    2'b01: begin //B type
        case(instr[3-1:0])
            3'b000: ALU_Ctrl_o <= 4'b0000; //beq
            3'b001: ALU_Ctrl_o <= 4'b0001; //bne
            3'b100: ALU_Ctrl_o <= 4'b1010; //blt
            3'b101: ALU_Ctrl_o <= 4'b0110; //bge
            3'b010: ALU_Ctrl_o <= 4'b0010; //lw
        endcase
    end
    2'b10: begin //R type
        case(instr)
            4'b0000: ALU_Ctrl_o <= 4'b0010; //add
            4'b1000: ALU_Ctrl_o <= 4'b0000; //sub
            4'b0111: ALU_Ctrl_o <= 4'b0011; //and
            4'b0110: ALU_Ctrl_o <= 4'b0100; //or
            4'b0100: ALU_Ctrl_o <= 4'b0101; //xor
            4'b0010: ALU_Ctrl_o <= 4'b0110; //slt
            4'b0001: ALU_Ctrl_o <= 4'b0111; //sll
            4'b1101: ALU_Ctrl_o <= 4'b1000; //sra
        endcase
    end
    2'b11: begin //I type
        case(instr[3-1:0])
            3'b000: ALU_Ctrl_o <= 4'b0010; //addi
            3'b010: ALU_Ctrl_o <= 4'b0110; //slti
            3'b001: ALU_Ctrl_o <= 4'b0111; //slli
            3'b101: ALU_Ctrl_o <= 4'b1001; //srli
            3'b111: ALU_Ctrl_o <= 4'b0011; //andi
            3'b110: ALU_Ctrl_o <= 4'b0100; //ori
            3'b100: ALU_Ctrl_o <= 4'b0101; //xori
        endcase
    end
endcase
```

- alu

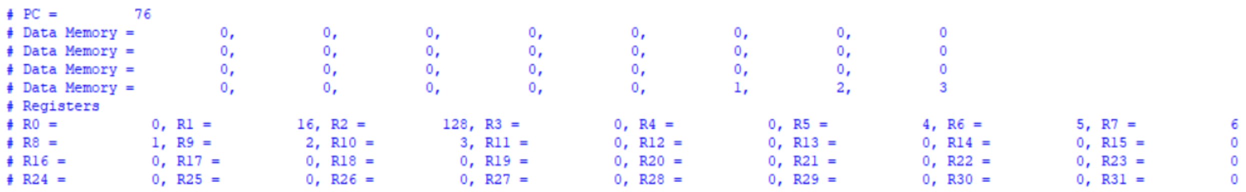
根據 ALU_control 決定 result 以及 zero 。



```
if(rst_n)
    zero <= (result==0)?1:0;

if(rst_n) begin
    case(ALU_control)
        4'b0000:
            result <= src1 - src2; //beq, sub
        4'b0001:
            result <= (src1 == src2); //bne
        4'b0010:
            result <= src1 + src2; //add, addi, sw, lw
        4'b0011:
            result <= src1 & src2; //and, andi
        4'b0100:
            result <= src1 | src2; //or, ori
        4'b0101:
            result <= src1 ^ src2; //xor, xori
        4'b0110:
            result <= (src1 < src2); //slt, slti, bge
        4'b0111:
            result <= src1 << src2; //sll, slli
        4'b1000:
            result <= src1 >>> src2; //sra
        4'b1001:
            result <= src1 >> src2; //srli
        4'b1010:
            result <= (src1 >= src2); //blt
    endcase
end
```

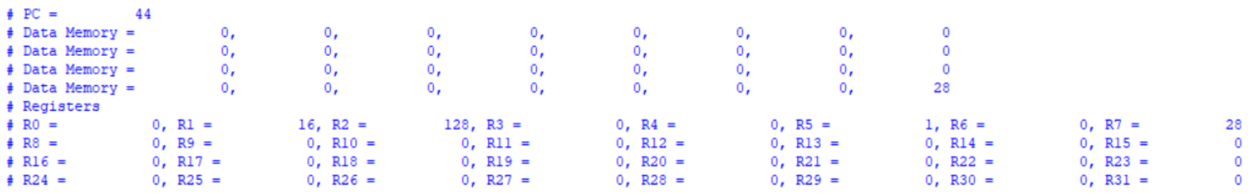
Implementation results:

Test Data 1



```
# PC =         76
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          1,          2,          3
# Registers
# R0 =          0, R1 =         16, R2 =        128, R3 =          0, R4 =          0, R5 =          4, R6 =          5, R7 =          6
# R8 =          1, R9 =          2, R10 =          3, R11 =          0, R12 =          0, R13 =          0, R14 =          0, R15 =          0
# R16 =          0, R17 =          0, R18 =          0, R19 =          0, R20 =          0, R21 =          0, R22 =          0, R23 =          0
# R24 =          0, R25 =          0, R26 =          0, R27 =          0, R28 =          0, R29 =          0, R30 =          0, R31 =          0
```

Test Data 2



```
# PC =         44
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,         28
# Registers
# R0 =          0, R1 =         16, R2 =        128, R3 =          0, R4 =          0, R5 =          1, R6 =          0, R7 =         28
# R8 =          0, R9 =          0, R10 =          0, R11 =          0, R12 =          0, R13 =          0, R14 =          0, R15 =          0
# R16 =          0, R17 =          0, R18 =          0, R19 =          0, R20 =          0, R21 =          0, R22 =          0, R23 =          0
# R24 =          0, R25 =          0, R26 =          0, R27 =          0, R28 =          0, R29 =          0, R30 =          0, R31 =          0
```

Problems encountered and solutions:

Q: lw 原本是 I-type，但是 funct3 = 010 和 slti 相同，如果 ALUOp 都設成 I-type 的 11 的話，在判斷 ALU_Ctrl_o 的時候會分不出來。

A: 觀察到 B-type 指令的 funct3 沒有 010 的，所以先把 lw 的 ALUOp 跟 B-type 一樣設成 01，之後在 ALU_Ctrl 再利用 funct3 判斷出指令。

Q: 上次作業總共有 16 個指令，所以之前我們是每個指令都給不同的 ALU_Ctrl_o，但是這次又新增了一些指令。

A: 整理出所有 ALU 功能之後重新編號，再判斷各個指令需要 ALU 的哪個功能。

Q: Debug

A: 這次新學會了看波形圖，找到了一些 ALU_Ctrl_o 沒有設定好的問題，以及 decoder 寫錯的問題。

Comment:

這次作業寫了差不多 4 小時而已，比起上次快了不少，我們覺得是因為上次 decoder 是自己寫的，所以這次要新加入指令也很順利，只需要思考如何讓指令之間不衝突就好。而且經過上次作業的訓練之後，我們也更懂得如何分配工作和排程，也沒有發生很多 verilog 語法錯誤。只是一開始發現 ALUresult 全部都是 x 時 debug 了很久，後來才發現是 Simple_Single_CPU 傳進去 alu 的 rst_i 寫成 rst_n，發現的時候整個快氣死。不過這次作業整體算是愉快的經驗，希望下一個 lab 也能如此順利。