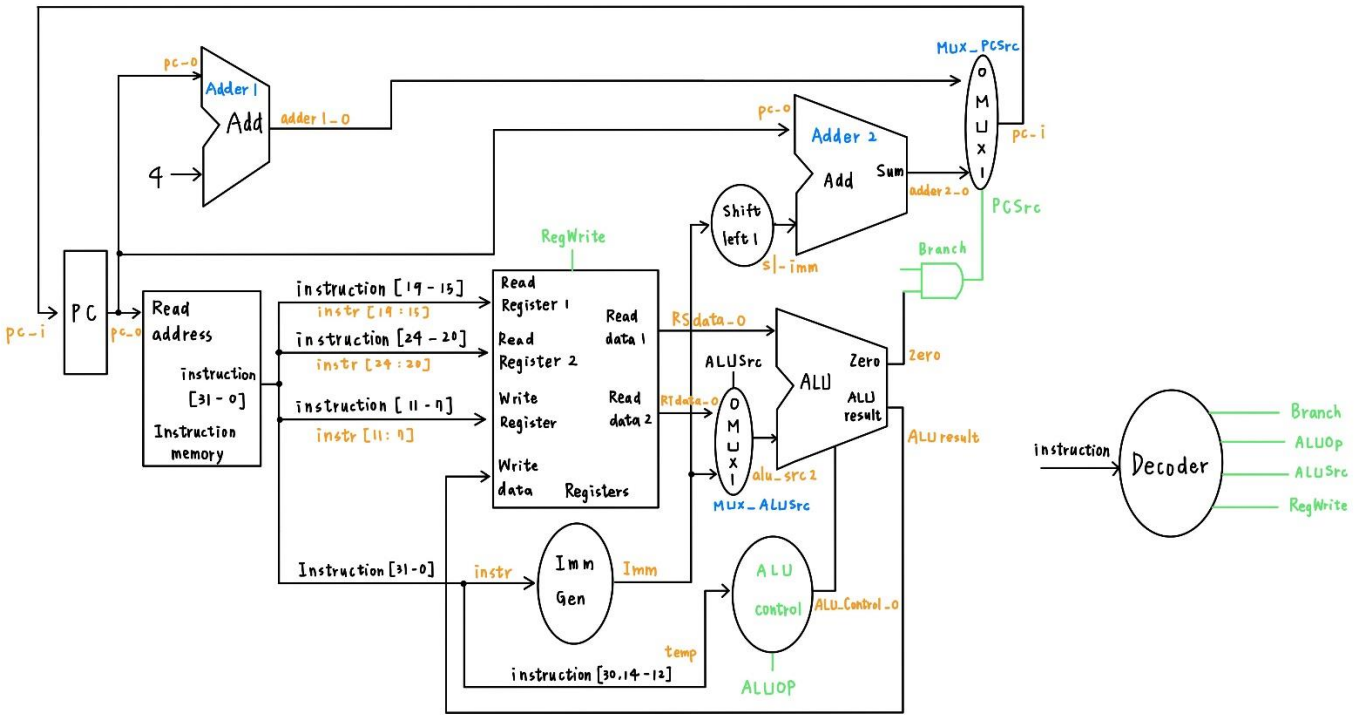
Computer Organization

Architecture diagram:



Detailed description of the implementation:

- Decoder

  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 就可以輸出 control 訊號。

  - ALUSrc: I-type 為 1，其他為 0。
  - RegWrite: B-type 為 0，R-type 為 1，I-type 為 1，其他為 0。
  - Branch: B-type 為 1，其他為 0。

- ALUOp: B-type 為 01，R-type 為 10，I-type 為 11，S-type 為 00。

```
assign ALUSrc = (opcode == 7'b0010011)?1:0; //I-type = 1
assign RegWrite = (opcode == 7'b1100011)?0: //B-type = 0
                  (opcode == 7'b0110011)?1: //R-type = 1
                  (opcode == 7'b0010011)?1:0; //I-type = 1
assign Branch = (opcode == 7'b1100011)?1:0; //B-type = 1
assign ALUOp = (opcode == 7'b1100011)?2'b01:( //B-type
               opcode == 7'b0110011)?2'b10:( //R-type
               opcode == 7'b0010011)?2'b11:2'b00; //I-type S-type
```

- 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'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]};
endcase
```

- Shift_Left_1

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

- Adder

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

- 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;
```

- 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'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
        endcase
    end
    2'b10: begin //R type
        case(instr)
            4'b0000: ALU_Ctrl_o <= 4'b0010; //add
            4'b1000: ALU_Ctrl_o <= 4'b0011; //sub
            4'b0111: ALU_Ctrl_o <= 4'b0100; //and
            4'b0110: ALU_Ctrl_o <= 4'b0101; //or
            4'b0100: ALU_Ctrl_o <= 4'b0110; //xor
            4'b0010: ALU_Ctrl_o <= 4'b0111; //slt
            4'b0001: ALU_Ctrl_o <= 4'b1000; //sll
            4'b1101: ALU_Ctrl_o <= 4'b1001; //sra
        endcase
    end
    2'b11: begin //I type
        case(instr[3-1:0])
            3'b000: ALU_Ctrl_o <= 4'b1010; //addi
            3'b010: ALU_Ctrl_o <= 4'b1011; //slti
            3'b001: ALU_Ctrl_o <= 4'b1100; //slli
            3'b101: ALU_Ctrl_o <= 4'b1101; //srli
            3'b111: ALU_Ctrl_o <= 4'b1110; //andi
            3'b110: ALU_Ctrl_o <= 4'b1111; //ori
        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
        4'b0001:
            result <= (src1 == src2); //bne
        4'b0010:
            result <= src1 + src2; //add
        4'b0011:
            result <= src1 - src2; //sub
        4'b0100:
            result <= src1 & src2; //and
        4'b0101:
            result <= src1 | src2; //or
        4'b0110:
            result <= src1 ^ src2; //xor
        4'b0111:
            result <= (src1 < src2); //slt
        4'b1000:
            result <= src1 << src2; //sll
        4'b1001:
            result <= src1 >>> src2; //sra
        4'b1010:
            result <= src1 + src2; //addi
        4'b1011:
            result <= (src1 < src2); //slti
        4'b1100:
            result <= src1 << src2; //slli
        4'b1101:
            result <= src1 >> src2; //srli
        4'b1110:
            result <= src1 & src2; //andi
        4'b1111:
            result <= src1 | src2; //ori
    endcase
end
```

Implementation results:

Test Data 1

```
VSIM 8> 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:/CO_lab/lab3/lab3_ModelSim/testbench.v(35)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/CO_lab/lab3/lab3_ModelSim/testbench.v line 35
```

Test Data 2

```
VSIM 5> 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:/CO_lab/lab3/lab3_ModelSim/testbench.v(35)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/CO_lab/lab3/lab3_ModelSim/testbench.v line 35
```

Test Data 3

```
VSIM 2> 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:/CO_lab/lab3/lab3_ModelSim/testbench.v(35)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/CO_lab/lab3/lab3_ModelSim/testbench.v line 35
```

Problems encountered and solutions:

Q: 為什麼 I-type 的 slli 和 srli 有 funct7，這樣不是會跟 imm 重疊嗎?

A: 因為 reg 大小只有 32bit，所以 imm 只需要用到 5 個 bit 就能表達所有可能了，剩下的 7 個 bit 直接用 0 寫死。

Q: 執行完 testdata3 後 r0 會變成 1。

A: 因為一開始的時候我們只判斷 branch 的時候把值給 0, 其餘的都給 1，導致 branch 的 default 是 1，最後的時候會改到 r0 的值。後來我們將所有的情況都列出來且將 default 給 0 後 r0 就不會被改掉了。

Q:執行完 testdata3 後 r11 = 8，bne 沒有跳回上一個指令。

A: 因為沒有將 result 放進 sensitivity list 中，導致 result 改變時 always block 沒有被執行，zero 的值沒有被改到，因此不會被判斷為 branch 指令。後來將 result 放進 sensitivity list 中就可以了。

Q: illegal reference to net

A: 會發生在使用 always block 的情況，解決方法是把 output wire 的 port 改成 output reg。

Comment:

跟個人作業相比，團體作業寫起來比較開心而且可以一起討論，討論的時候就會有新的想法出現，隊友也可以解決自己的盲點，真正在打 code 的時候，因為這次有許多 opcode, ALU_control 等等的數字需要對照，所以兩個人一起合作效率比較高。整體而言，我比較喜歡團體作業，在完成作業的同時還可以增進友誼。