109550025 謝翔丞 109550069 林宇柔

Report LAB4

Implementation

Decoder.v

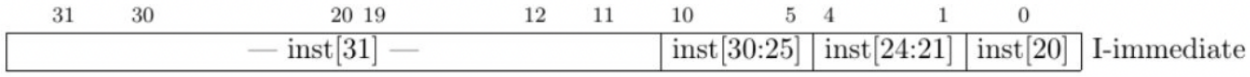
```
wire [7-1:0]opcode;
wire [3-1:0]funct3;

assign opcode = instr_i[6:0];
assign funct3 = instr_i[14:12];



assign RegWrite = (opcode[5:3] == 3'b100)? 0 : 1 ;
// except for sw beq
assign Branch = (opcode[6:2] == 5'b11000)? 1 : 0 ;
// branch
assign Jump = instr_i[2];
// jal or jalr
assign WriteBack0 = (opcode[6:4] == 3'b011 || opcode[6:4] == 3'b001)? 0 : 1;
// Rtype or addi use result  directly
assign WriteBack1 = instr_i[2];
// only jal or jalr need
assign MemRead = (opcode[6:4] == 3'b000)? 1 : 0 ;
// only lw
assign MemWrite = (opcode[6:4] == 3'b010)? 1 : 0 ;
// only sw
assign ALUSrcA = ( opcode[3:2] == 2'b01 )? 1 : 0 ;
// only jalr need choose 1 (use src1 + imme)
assign ALUSrcB = (opcode[6:4] == 3'b000 || opcode[6:4] == 3'b001 || opcode[6:4] == 3'b010 )? 1 : 0 ;
// only addi or lw or sw take imme
assign ALUOp[1:0] = (opcode[6:4] == 3'b110)? 2'b01 : (opcode[6:4] == 3'b011)? 2'b10 : (opcode[6:4] == 3'b001)? 2'b11 : 2'b00 ;
// r-type => 10 ; branch => 01 ; lw/sw => 00 ; addi => 11
```

To begin with, we check whether each operation should be assigned as 0 or 1
Then, we use this classification to see the correlation in between, and write the conditional expressions.
For RegWrite, if the operation is sw or beq, it will be set to 1.
For Branch, check if the operation is branch or not.
For jump and WriteBack[1], check the third bit of the instruction to see if it is jal or jalr.
For WriteBack[0], set to 0 if the operation is R-type or addi.
For MemRead, set to 1 if the operation is lw.
For MemWrite, set to 1 if the operation is sw.
For ALUSrcA, set to 1 if the operation is jalr since it’s the only instruction which needs to check src1+immediate.
For ALUSrcB, set to 1 if the operation is addi or lw or sw.
For ALUOp[1:0], set to 2’b10 if the operation is R-type, 2’b01 if branch, 2’b00 if lw or sw, and 2’b11 if addi.

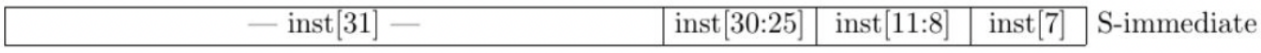
Imm_Gen.v

```
always @(*) begin

    case(opcode)
        //addi , jalr , lw
        7'b0010011, 7'b0000011 , 7'b1100111:
            Imm_Gen_o <= { {21{instr_i[31]}}, instr_i[30:20] };
        //R-type
        // 7'b0110011:
            // Imm_Gen_o <= 0;
        //sw
        7'b0100011:
            Imm_Gen_o <= {  {21{instr_i[31]}} , instr_i[30:25] , instr_i[11:7] };
        //branch
        7'b1100011:
            Imm_Gen_o <= { {20{instr_i[31]}}, instr_i[7] ,instr_i[30:25] , instr_i[11:8] ,1'b0 };
        //jal
        7'b1101111:
            Imm_Gen_o <= {  { 12{instr_i[31]}}, instr_i[19:12], instr_i[20] , instr_i[30:21] , 1'b0 };

        default:
            Imm_Gen_o <= 0;

    endcase

end
```

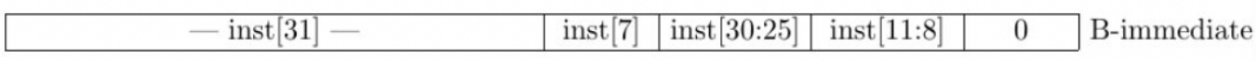
We implement this according to the sign-extension instructions given in the slides. First of all, we classify each operation type with their opcode, then assign them the correct sign extension.
For addi, jalr, lw, the MSB should be extended by 21 bits, then concatenate with the following 11 bits, representing the I-immediate.



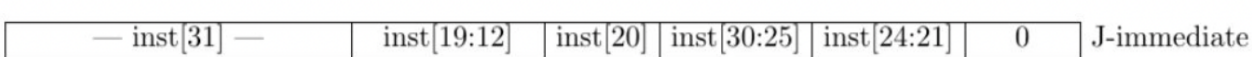
For sw, MSB should also extend by 21 bits, concatenated with the following 6 bits and the 12th to 8th bit, representing the S-immediate.



For branch, the MSB should extend by 20 bits, and the concatenation part is the same as sw but the 8th bit was moved forward and the last bit is assigned with 0, representing the B-immediate.



For jal, the MSB should be extended by 12 bits, then concatenate with the 20th to 13th bit, then the 21bit, 31th to 22th bit, the last bit would be assigned to 0, representing the J-immediate.



ALU_Ctrl.v

```
reg [3:0] ALU_Ctrl_o_reg;
assign ALU_Ctrl_o = ALU_Ctrl_o_reg;
always @(*) begin
    case(ALUOp)
        2'b00://lw sw => add
            ALU_Ctrl_o_reg <= 4'b0010;
        2'b01://branch => sub
            ALU_Ctrl_o_reg <= 4'b0110;
        2'b10://R-type
            begin

                case(instr)
                    4'b0000://add
                        ALU_Ctrl_o_reg <= 4'b0010;
                    4'b1000://sub
                        ALU_Ctrl_o_reg <= 4'b0110;
                    4'b0111://and
                        ALU_Ctrl_o_reg <= 4'b0000;
                    4'b0110://or
                        ALU_Ctrl_o_reg <= 4'b0001;
                    4'b0010://slt
                        ALU_Ctrl_o_reg <= 4'b0111;
                    default:
                        ALU_Ctrl_o_reg <= 4'b0000;

                endcase

            end
        2'b11:
            ALU_Ctrl_o_reg <= 4'b0010;

        default:
            ALU_Ctrl_o_reg <= 4'b1111;
    endcase
end
```

We determine the ALU control with the ALUOp, if it is a R-type instruction, we further determine them by which instruction it is.

alu.v

```
reg signed[32-1:0] a,b;

assign Zero = ~(|result);

always @(*) begin

    a <= src1;
    b <= src2;
    if(~rst_n)begin
        result <= 0;
    end
    else begin
        case(ALU_control)
            4'b0010: // add
                result <= a + b;
            4'b0110://sub
                result <= a - b;
            4'b0000://and
                result <= a & b;
            4'b0001://or
                result <= a | b;
            4'b0111://slt
                begin
                    result[0] <= (a < b);
                    result[31:1] <= 0;
                end
            default:
                result <= result;

        endcase
    end


end
```

For ALU, set the result to 0 if the rst_n is 0, otherwise judge by the ALU_control to see which operation should be executed and saved as the result.

Simple_Single_CPU.v

```
ProgramCounter PC(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .pc_i(pc_i),
    .pc_o(pc_o)
);

Adder Adder_PCPlus4(
    .src1_i(pc_o),
    .src2_i(Imm_4),
    .sum_o(PcPlus4)
);

Instr_Memory IM(
    .addr_i(pc_o),
    .instr_o(instr)
);

Reg_File RF(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .RSaddr_i(instr[19:15]),
    .RTaddr_i(instr[24:20]),
    .RDaddr_i(instr[11:7]),
    .RDdata_i(RegWriteData),
    .RegWrite_i(RegWrite),
    .RSdata_o(RSdata_o),
    .RTdata_o(RTdata_o)
);
```

```
Decoder Decoder(
    .instr_i(instr[6:0]),
    .RegWrite(RegWrite),
    .Branch(Branch),
    .Jump(Jump),
    .WriteBack1(WriteBack1),
    .WriteBack0(WriteBack0),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),
    .ALUOp(ALUOp)
);

Imm_Gen ImmGen(
    .instr_i(instr),
    .Imm_Gen_o(Imm_Gen_o)
);

ALU_Ctrl ALU_Ctrl(
    .instr(ALUControlIn),
    .ALUOp(ALUOp),
    .ALU_Ctrl_o(ALUControlOut)
);

MUX_2to1 MUX_ALUSrcA(
    .data0_i(pc_o),
    .data1_i(RSdata_o),
    .select_i(ALUSrcA),
    .data_o(MUX_ALUSrcA_o)
);
```

```
Adder Adder_PCReg(
    .src1_i(MUX_ALUSrcA_o),
    .src2_i(Imm_Gen_o),
    .sum_o(Adder_PCReg_o)
);

MUX_2to1 MUX_PCSrc(
    .data0_i(PcPlus4),
    .data1_i(Adder_PCReg_o),
    .select_i(PCSrc),
    .data_o(pc_i)
);

MUX_2to1 MUX_ALUSrcB(
    .data0_i(RTdata_o),
    .data1_i(Imm_Gen_o),
    .select_i(ALUSrcB),
    .data_o(MUX_ALUSrcB_o)
);

alu alu(
    .rst_n(rst_i),
    .src1(RSdata_o),
    .src2(MUX_ALUSrcB_o),
    .ALU_control(ALUControlOut),
    .Zero(Zero),
    .result(ALU_Result)
);
```

```
Data_Memory Data_Memory(
    .clk_i(clk_i),
    .addr_i(ALU_Result),
    .data_i(RTdata_o),
    .MemRead_i(MemRead),
    .MemWrite_i(MemWrite),
    .data_o(Data_Memory_o)
);

MUX_2to1 MUX_WriteBack0(
    .data0_i(ALU_Result),
    .data1_i(Data_Memory_o),
    .select_i(WriteBack0),
    .data_o(WriteBack0_o)
);

MUX_2to1 MUX_WriteBack1(
    .data0_i(WriteBack0_o),
    .data1_i(PcPlus4),
    .select_i(WriteBack1),
    .data_o(RegWriteData)
);
```

Give all correct values.

Result

```
~/110second/Computer_Organization/Lab04 on  master
❯ ./demo.sh
iverilog *.v -o Simple_CPU.vvp
vvp Simple_CPU.vvp -fst -sdf-verbose -lxt2
WARNING: Instr_Memory.v:15: $readmemb(Instruction.txt): Not enough words in the file for the requested range [0:64].
LXT2 info: dumpfile Simple_CPU.lxt opened for output.
CONGRATULATION!!
MMMMMMMMMMMMMMMMMMWXKOOkkOO0XWMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMW0dc:::cclllllcc:::coONMMMMMMMMMMMMM
MMMMMMMMMMWk:;lk0NNNNNNNNNNNNNNNKko;:dNMMMMMMMMM
MMMMMMMMMO'cONNNNNNNNNNNNNNNNNNNNNNKl'xWMMMMMMMM
MMMMMMMNc'ONNNNNNNNKXNNNNNNNNNXNNNNNN0;;XMMMMMM
MMMMMMW;,KNNNNNNN0.xNNNNNNNNl;NNNNNNNXc'NMMMMM
MMMMWKl.;dXNN0lNNNd.KNNNNNNNNx.KNNdkNNNxc.:KWMMM
MMXl,cl...0NNd.:::;,:::::::::;,:::.cNNX'..cc;cKMM
Wo'oXNNk.,NNNd'WWWWWWWWWWWWWWWWWWWc:NNNc.dNNNx'cN
O;;.ONX,.:NNNo,Wk:NWWWWWWWWWWWWooWl;NNNo.'0NK.,;k
MM0.0x,c.cNNNo,Wl.XWWWWWWWWWWWW,,Wl,KNNx.:'oK'xMM
MMX.,.:0.lNNNo,Wo'XWWWWWWWWWWW::Wl'ONN0.do.,.0MM
MMMNN.ox.oXNNo,WWWWWWWWWWWWWWWWWWl,kKNX.lk.0NMMM
MMMM0.kl.xKNNo'WWWWWWWNcoocKWWWWWWl,kONN:;K.dMMMM
MMMMd.K:,xoKNd'WWWWWWWXOkKWWWWWWWc,kxxKo'N;:MMMM
MMMM;:N'.','0d.;codxxkkkOOOOkkxdoc;.;d'.,..Kd.WMMM
MMMN.xNkOXk,.'..kOkxd..looo'.oxkO0'...;xN0xX0.0MMM
MMMx'XNNNNNO....,:oxOO:.dx';kOxo:,..'.kNNNNNN:lMMM
MMW;lNXOK0x..:..:;,,''.......',,;:'.;,.o0KOXNx.NMM
MMO.cc'.ox'.:;,,::::::::::::::::::;,,:'.dx..:l.dMM
MMN0OKl'x,.,:;,.;:::::::::::::::::::.'::'.d:;X0ONMM
MMMMMW';;.Oko,.:::::::::::::::::::''ox0;':.KMMMMM
MMMMMWxoc.,lx,.:::::::::::::::::ccc,.xo;.;odXMMMMM
MMMMMMMMk.0k..,:::::::::::::::ccccc:..dK,lMMMMMMMM
MMMMMMMMX'....,:::::::::::::::ccccccc.....OMMMMMMMM
```

Problems Encountered

This lab is truly interesting to us because we have to implement a whole simple single CPU, also, functions other than R-type are newly-increased, so we had to spend a lot of time to implement our decoder, deciding which should be assigned to 1 or 0.

Other than that, this lab is hard to debug, but fortunately there was few problem in the first version so we handled that pretty fast.

Lastly, we found that some ways of implementing the decoder is to operate them with different bits, for instance, “assign line = instr[4]^instr[5]”, but our way of implementing it was to classify the instructions, find their common ground and decide by using the conditional operations for example, assign line = (opcode[6:4] == 3’b010) ? 1’b1 : 1’b0; Still, we would like to further understand if there exists some difference of efficacy when we run these two ways of implementation.