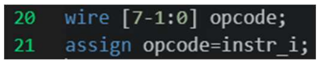
Report-Lab04　　　　Name:杜峯/林念慈　　　ID:109550096/109550130

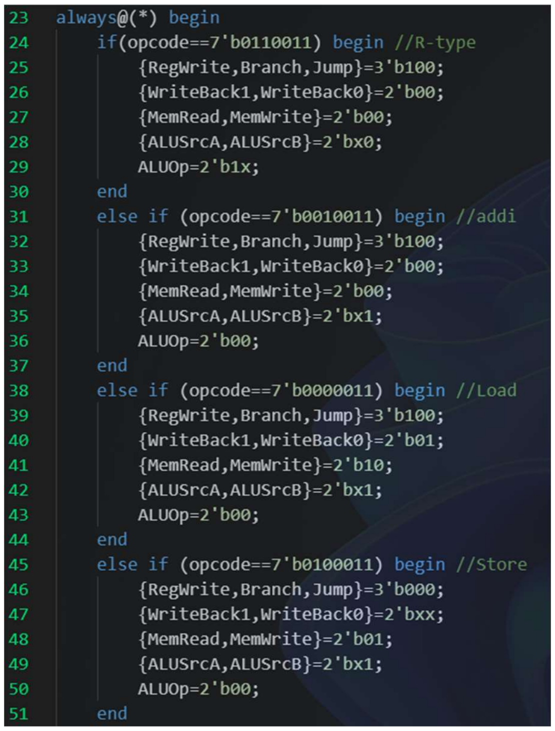
1. Detailed description of the implementation

a. Decoder.v

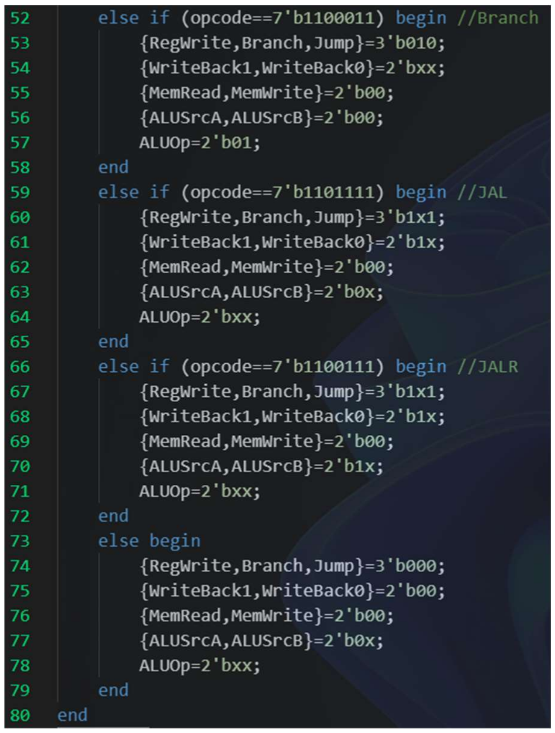


```
wire [7-1:0] opcode;
assign opcode=instr_i;
```

Assign the value of instr_i to a new parameter named "opcode".



```
always@(*) begin
    if(opcode==7'b0110011) begin //R-type
        {RegWrite,Branch,Jump}=3'b100;
        {WriteBack1,WriteBack0}=2'b00;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'bx0;
        ALUOp=2'b1x;
    end
    else if (opcode==7'b0010011) begin //addi
        {RegWrite,Branch,Jump}=3'b100;
        {WriteBack1,WriteBack0}=2'b00;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'bx1;
        ALUOp=2'b00;
    end
    else if (opcode==7'b0000011) begin //Load
        {RegWrite,Branch,Jump}=3'b100;
        {WriteBack1,WriteBack0}=2'b01;
        {MemRead,MemWrite}=2'b10;
        {ALUSrcA,ALUSrcB}=2'bx1;
        ALUOp=2'b00;
    end
    else if (opcode==7'b0100011) begin //Store
        {RegWrite,Branch,Jump}=3'b000;
        {WriteBack1,WriteBack0}=2'bxx;
        {MemRead,MemWrite}=2'b01;
        {ALUSrcA,ALUSrcB}=2'bx1;
        ALUOp=2'b00;
    end
```

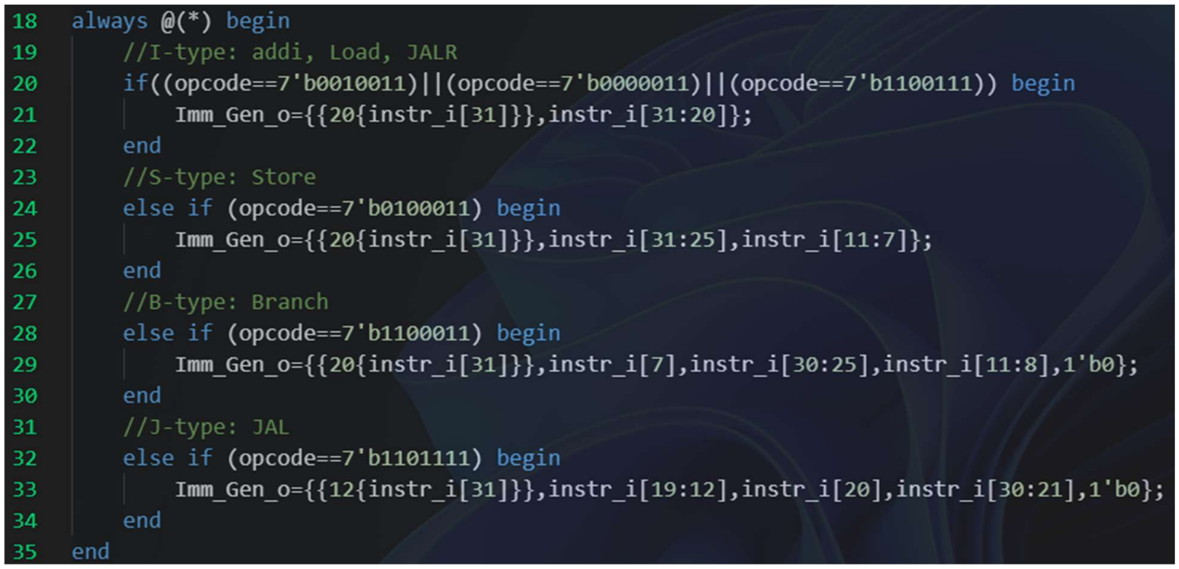


```
    else if (opcode==7'b1100011) begin //Branch
        {RegWrite,Branch,Jump}=3'b010;
        {WriteBack1,WriteBack0}=2'bxx;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'b00;
        ALUOp=2'b01;
    end
    else if (opcode==7'b1101111) begin //JAL
        {RegWrite,Branch,Jump}=3'b1x1;
        {WriteBack1,WriteBack0}=2'b1x;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'b0x;
        ALUOp=2'bxx;
    end
    else if (opcode==7'b1100111) begin //JALR
        {RegWrite,Branch,Jump}=3'b1x1;
        {WriteBack1,WriteBack0}=2'b1x;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'b1x;
        ALUOp=2'bxx;
    end
    else begin
        {RegWrite,Branch,Jump}=3'b000;
        {WriteBack1,WriteBack0}=2'b00;
        {MemRead,MemWrite}=2'b00;
        {ALUSrcA,ALUSrcB}=2'b0x;
        ALUOp=2'bxx;
    end
end
```

Depending on the various value of opcode, the corresponded operation and control signal would be executed. It's easy to understand by the following sheet:

signal\opcode	R-type	addi	Load	Store	Branch	JAL	JALR	Default
RegWrite	1	1	1	0	0	1	1	0
Branch	0	0	0	0	1	x	x	0
Jump	0	0	0	0	0	1	1	0
WriteBack1	0	0	0	x	x	1	1	0
WriteBack0	0	0	1	x	x	x	x	0
MemRead	0	0	1	0	0	0	0	0
MemWrite	0	0	0	1	0	0	0	0
ALUSrcA	x	x	x	x	0	0	1	0
ALUSrcB	0	1	1	1	0	x	x	x
ALUOp	xx	00	00	00	01	xx	xx	xx

b. Imm_Gen.v



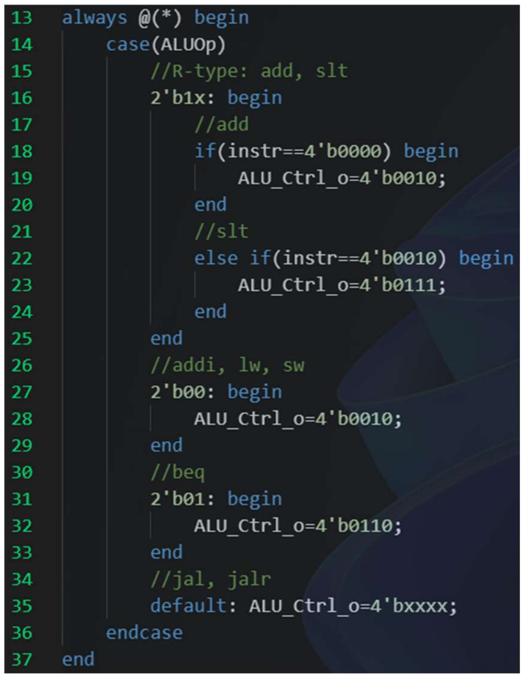
```
always @(*) begin
    //I-type: addi, Load, JALR
    if((opcode==7'b0010011)||(opcode==7'b0000011)||(opcode==7'b1100111)) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[31:20]};
    end
    //S-type: Store
    else if (opcode==7'b0100011) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[31:25],instr_i[11:7]};
    end
    //B-type: Branch
    else if (opcode==7'b1100011) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[7],instr_i[30:25],instr_i[11:8],1'b0};
    end
    //J-type: JAL
    else if (opcode==7'b1101111) begin
        Imm_Gen_o={{12{instr_i[31]}},instr_i[19:12],instr_i[20],instr_i[30:21],1'b0};
    end
end
```

At first, we deal with I-type, addi, Load and JALR. We generated the value of Imm_Gen_o by sign extension of the thirty-first bit of instr_i, and the rest is the thirty-first to the 20th bit of instr_i.

Second, we deal with S-type, Store. Similarly, we generated the value of Imm_Gen_o by sign extension of the thirty-first bit of instr_i, and the other two sections are the thirty-first bit to the twenty fifth bit of instr_i and the eleventh bit to seventh bit of instr_i.

For the final two types, Branch and JAL, we do the similar things as the above. However, we need to plus ‘0’ at the last bit of Imm_Gen_o, since we have to double the value. i.e. 010(2) -> 0100(4)

c. ALU_Ctrl.v



```
always @(*) begin
    case(ALUOp)
        //R-type: add, slt
        2'b1x: begin
            //add
            if(instr==4'b0000) begin
                ALU_Ctrl_o=4'b0010;
            end
            //slt
            else if(instr==4'b0010) begin
                ALU_Ctrl_o=4'b0111;
            end
        end
        //addi, lw, sw
        2'b00: begin
            ALU_Ctrl_o=4'b0010;
        end
        //beq
        2'b01: begin
            ALU_Ctrl_o=4'b0110;
        end
        //jal, jalr
        default: ALU_Ctrl_o=4'bxxxx;
    endcase
end
```

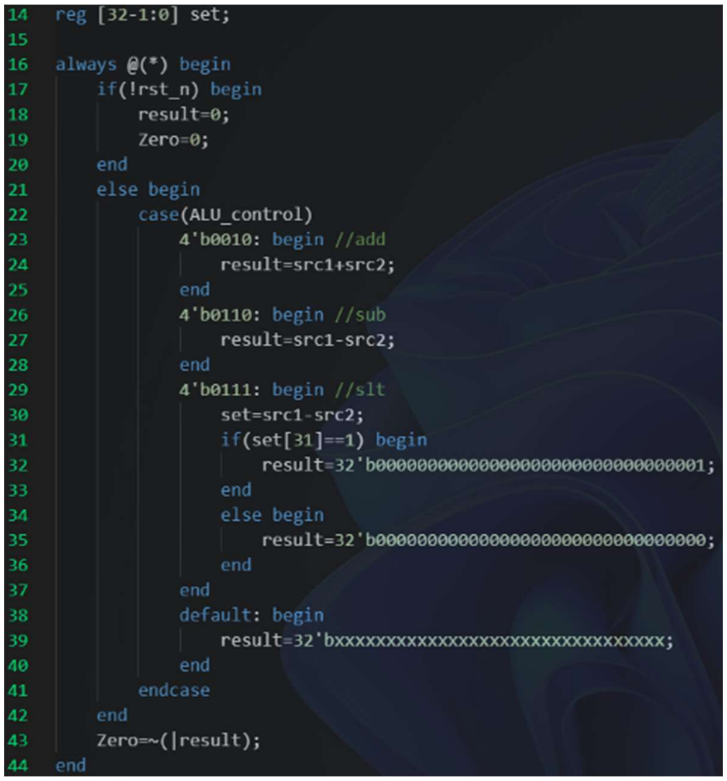
At first, we process R-type, “add” and “slt”, with “ALUOp” equal to “1x”. If “instr” equals to “0000”, the output of ALU control is “0010”, which is the operation of “add”. If “instr” equals to “0010”, the output of ALU control is “0111”, which is the operation of “slt”.

Next, we process “addi”, “lw” and “sw”, with “ALUOp” equal to “00”. In this section, the output of ALU control is “0010”, which would also be executed as the operation of addition in ALU.

For “ALUOp” equal to “01”, we deal with “beq”. The output of ALU control would be assigned to be “0110”, which would execute the subtraction in ALU.

By default, we directly assign the output of ALU control to be “xxxx”, which deal with “jal” and “jalr”.

d. alu.v



```
reg [32-1:0] set;

always @(*) begin
    if(!rst_n) begin
        result=0;
        Zero=0;
    end
    else begin
        case(ALU_control)
            4'b0010: begin //add
                result=src1+src2;
            end
            4'b0110: begin //sub
                result=src1-src2;
            end
            4'b0111: begin //slt
                set=src1-src2;
                if(set[31]==1) begin
                    result=32'b00000000000000000000000000000001;
                end
                else begin
                    result=32'b00000000000000000000000000000000;
                end
            end
            default: begin
                result=32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
            end
        endcase
    end
    Zero=~(|result);
end
```

At first, we initialize a parameter “set”. If “rst_n” equals to zero, we assign the value of “result” and “zero” to be zero. If not, we do different operations depending on “ALU_control”.

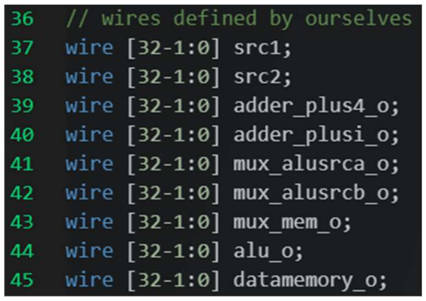
If “ALU_control” is “0010”, we do the operation of addition. Hence, the “result” is “src1” plus “src2”.

If “ALU_control” is “0110”, we do the operation of subtraction. Hence, the “result” is “src1” subtracting “src2”.

If “ALU_control” is “0111”, we do the operation of “slt”. Hence, we assign the value of “set” to be the value that “src1” subtracts “src2”. Then, if the thirty first bit of “set” equals to 1, the “result” is “00000000000000000000000000000001”. Otherwise, the “result” is “00000000000000000000000000000000”.

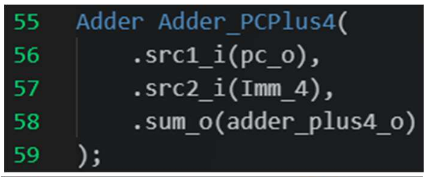
By default, “result” is “xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx”.

e. Simple_Single_CPU.v



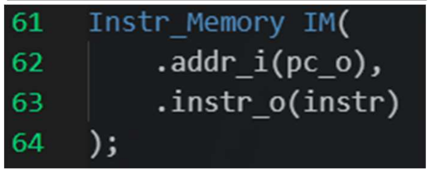
```
// wires defined by ourselves
wire [32-1:0] src1;
wire [32-1:0] src2;
wire [32-1:0] adder_plus4_o;
wire [32-1:0] adder_plusi_o;
wire [32-1:0] mux_alusrca_o;
wire [32-1:0] mux_alusrcb_o;
wire [32-1:0] mux_mem_o;
wire [32-1:0] alu_o;
wire [32-1:0] datamemory_o;
```

At first, we initialize some parameters that would be used later.



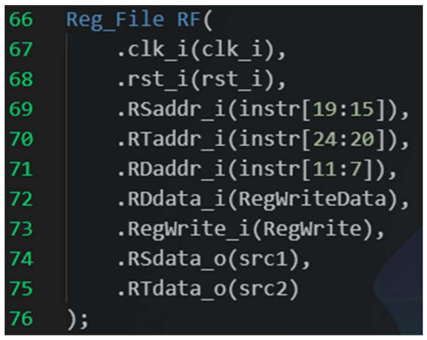
```
Adder Adder_PCPlus4(
    .src1_i(pc_o),
    .src2_i(Imm_4),
    .sum_o(adder_plus4_o)
);
```

In this section, we do a general case, PC=PC+4. The parameter “adder_plus4_o” is the output of the adder.



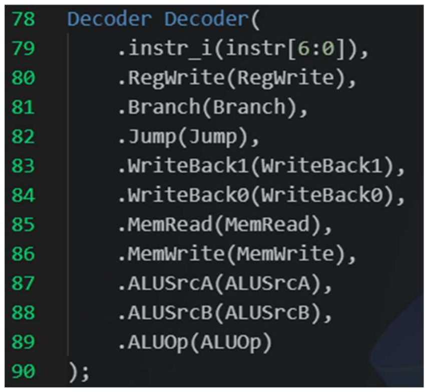
```
Instr_Memory IM(
    .addr_i(pc_o),
    .instr_o(instr)
);
```

In this section, the input is the output of the program counter, and the output is the instruction.



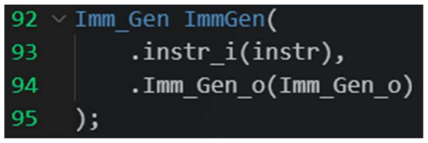
```
Reg_File RF(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .RSaddr_i(instr[19:15]),
    .RTaddr_i(instr[24:20]),
    .RDaddr_i(instr[11:7]),
    .RDdata_i(RegWriteData),
    .RegWrite_i(RegWrite),
    .RSdata_o(src1),
    .RTdata_o(src2)
);
```

In this section, the RSaddr_i, RTaddr_i, and RDaddr_i are set by the bits from instruction based on the format in risc-v. The RDdata_i is assigned as the result of ALU. With others parameters, I just put the corresponding wire to them.



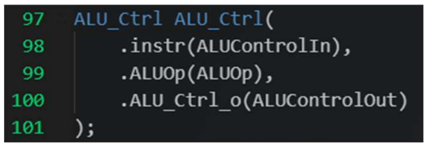
```
Decoder Decoder(
    .instr_i(instr[6:0]),
    .RegWrite(RegWrite),
    .Branch(Branch),
    .Jump(Jump),
    .WriteBack1(WriteBack1),
    .WriteBack0(WriteBack0),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),
    .ALUOp(ALUOp)
);
```

In this section, the main task is to decode the instruction to the signal control path.



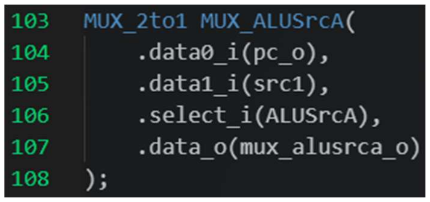
```
Imm_Gen ImmGen(
    .instr_i(instr),
    .Imm_Gen_o(Imm_Gen_o)
);
```

In this section, it is used to translate the instruction into the immediate value we want.



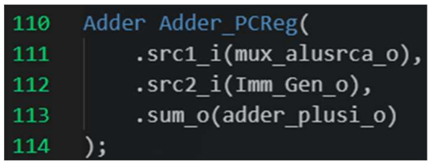
```
ALU_Ctrl ALU_Ctrl(
    .instr(ALUControlIn),
    .ALUOp(ALUOp),
    .ALU_Ctrl_o(ALUControlOut)
);
```

In this section, I just put the corresponding wires to ALU control registers.



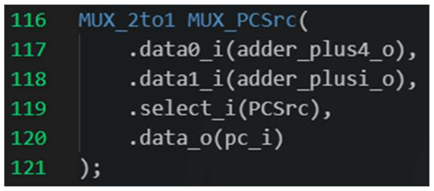
```
MUX_2to1 MUX_ALUSrcA(
    .data0_i(pc_o),
    .data1_i(src1),
    .select_i(ALUSrcA),
    .data_o(mux_alusrca_o)
);
```

In this section, the inputs are the output of the pc and “src1”, with a selector “ALUSrcA”. The output is “mux_alusrca_o”.



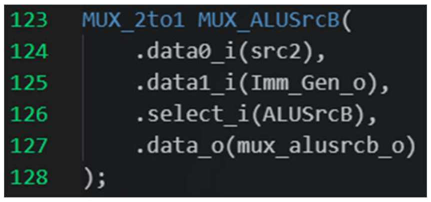
```
Adder Adder_PCReg(
    .src1_i(mux_alusrca_o),
    .src2_i(Imm_Gen_o),
    .sum_o(adder_plusi_o)
);
```

In this section, the inputs are “mux_alusrca_o” and the immediate value of “Imm_Gen_o”. The output is “adder_plusi_o”.



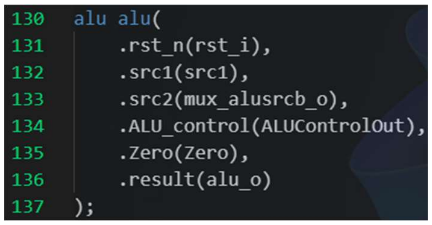
```
MUX_2to1 MUX_PCSrc(
    .data0_i(adder_plus4_o),
    .data1_i(adder_plusi_o),
    .select_i(PCSrc),
    .data_o(pc_i)
);
```

In this section, the inputs are two outputs from the general case “adder_plus4_o” and the branch case “adder_plusi_o” with the selector “PCSrc”. Finally, the output would go back to the program counter.



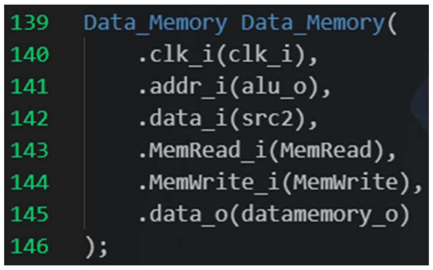
```
MUX_2to1 MUX_ALUSrcB(
    .data0_i(src2),
    .data1_i(Imm_Gen_o),
    .select_i(ALUSrcB),
    .data_o(mux_alusrcb_o)
);
```

In this section, the inputs are "src2" and immediate value. We would use the selector "ALUSrcB" to determine which data is what we want. The output is "mux_alusrcb_o".



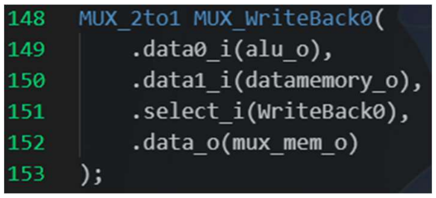
```
alu alu(
    .rst_n(rst_i),
    .src1(src1),
    .src2(mux_alusrcb_o),
    .ALU_control(ALUControlOut),
    .Zero(Zero),
    .result(alu_o)
);
```

In ALU, the src1 is "src1", but the src2 is the output of "MUX_ALUSrcB".



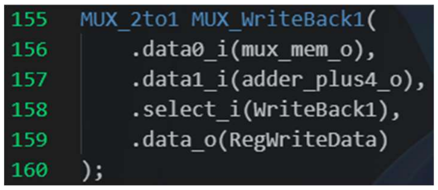
```
Data_Memory Data_Memory(
    .clk_i(clk_i),
    .addr_i(alu_o),
    .data_i(src2),
    .MemRead_i(MemRead),
    .MemWrite_i(MemWrite),
    .data_o(datamemory_o)
);
```

In this section, we just put the corresponding wires to the data memory.



```
MUX_2to1 MUX_WriteBack0(
    .data0_i(alu_o),
    .data1_i(datamemory_o),
    .select_i(WriteBack0),
    .data_o(mux_mem_o)
);
```

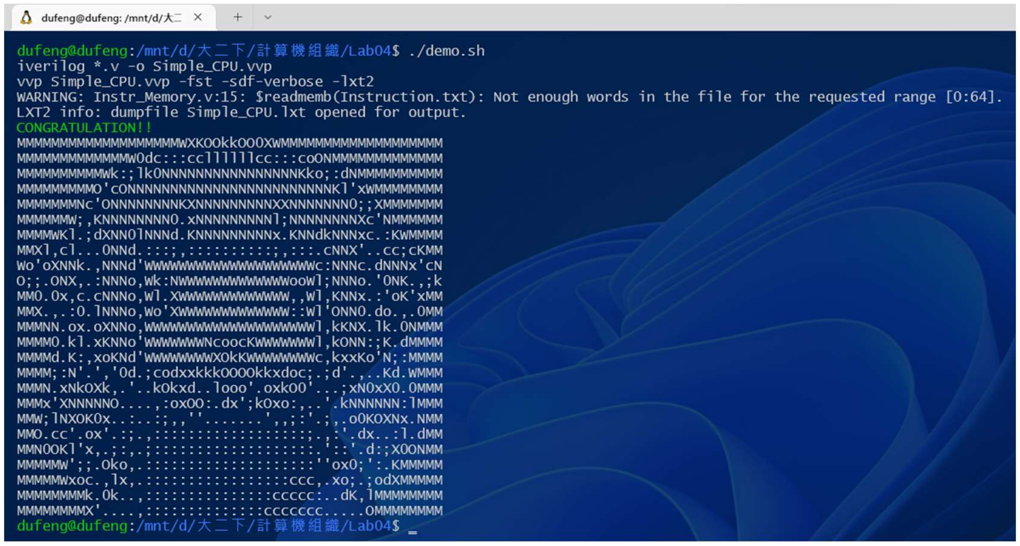
In this section, there are two inputs, the output of the alu and output of the data memory. It is controlled by "WriteBack0". The output is "mux_mem_o".



```
MUX_2to1 MUX_WriteBack1(
    .data0_i(mux_mem_o),
    .data1_i(adder_plus4_o),
    .select_i(WriteBack1),
    .data_o(RegWriteData)
);
```

In the final section, there are two inputs, the output of the "MUX_WriteBack0" and the result of "adder_plus4_o". It is controlled by "WriteBack1". The output is "RegWriteData".

2. Implementation results



```
dufeng@dufeng:/mnt/d/大二下/計算機組織/Lab04$ ./demo.sh
iverilog *.v -o Simple_CPU.vvp
vvp Simple_CPU.vvp -fst -sdf-verbose -lxt2
WARNING: Instr_Memory.v:15: $readmemb(Instruction.txt): Not enough words in the file for the requested range [0:64].
LXT2 info: dumpfile Simple_CPU.lxt opened for output.
CONGRATULATION!!
MMMMMMMMMMMMMMMMMMMMWXKOOkkOOOXWMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMW0dc:::ccllllllcc:::coONMMMMMMMMMMMMM
MMMMMMMMMMWk:;lkONNNNNNNNNNNNNNNNNNNKko;:dNMMMMMMMMM
MMMMMMMMMO'cONNNNNNNNNNNNNNNNNNNNNNNNNKl'xWMMMMMMMM
MMMMMMMNc'ONNNNNNNNKXNNNNNNNNNXXNNNNNNN0;;XMMMMMMM
MMMMMMW;,KNNNNNNNN0.xNNNNNNNNNl;NNNNNNNNXc'NMMMMMM
MMMMWKl.;dXNN0lNNNd.KNNNNNNNNNx.KNNdkNNNxc.:KWMMMM
MMXl,cl...0NNd.:::;,::::::::::;,:::.cNNX'..cc;cKMM
Wo'oXNNk.,NNNd'WWWWWWWWWWWWWWWWWWWWWc:NNNc.dNNNx'cN
O;;.ONX,.:NNNo,Wk:NWWWWWWWWWWWWWWWooWl;NNNo.'ONK.,;k
MM0.0x,c.cNNNo,Wl.XWWWWWWWWWWWWWWW,,Wl,KNNx.:'oK'xMM
MMX.,.:0.lNNNo,Wo'XWWWWWWWWWWWWWWW::Wl'ONN0.do.,.0MM
MMMNN.ox.oXNNo,WWWWWWWWWWWWWWWWWWWWWWl,kKNX.lk.0NMMM
MMMM0.kl.xKNNo'WWWWWWWNcoocKWWWWWWWl,kONN:;K.dMMMM
MMMMd.K:,xoKNd'WWWWWWWWXOkKWWWWWWWWWc,kxxKo'N;:MMMM
MMMM;:N'.','0d.;codxxkkk0000kkxdoc;.;d'.,..Kd.WMMM
MMMN.xNkOXk,.'..kOkxd..looo'.oxkO0'...;xN0xX0.0MMM
MMMx'XNNNNN0....,:ox00:.dx';kOxo:,..'.kNNNNNN:lMMM
MMW;lNXOK0x..:..:;,,''.......',,;:'.;,.o0KOXNx.NMM
MMO.cc'.ox'.:;.,:::::::::::::::::::;.,:'.dx..:l.dMM
MMNOOKl'x,.;:,.;:::::::::::::::::::::.'::'.d:;XOONMM
MMMMMW';;.Oko,.:::::::::::::::::::::''ox0;':.KMMMMM
MMMMMWxoc.,lx,.::::::::::::::::::ccc,.xo;.;odXMMMMM
MMMMMMMMk.0k..,:::::::::::::::::ccccc:..dK,lMMMMMMMM
MMMMMMMMX'....,:::::::::::::::cccccccc.....OMMMMMMMM
dufeng@dufeng:/mnt/d/大二下/計算機組織/Lab04$ _
```

3. Problems encountered and solutions

In Simple_Single_CPU.v, we got WRONG after executing demo.sh, for we didn’t know how to finish some empty brackets. Hence, we tried lots of times to complete the needed parameters by study the gram in Slide.pdf.