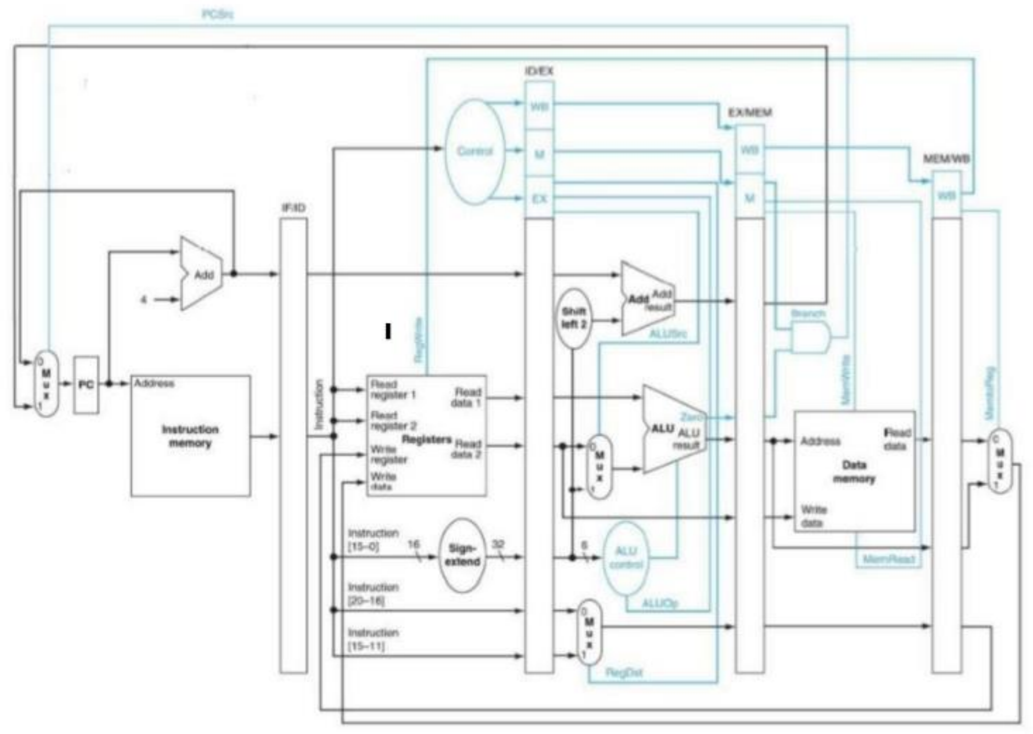
Computer Organization Lab4

Name:林秉

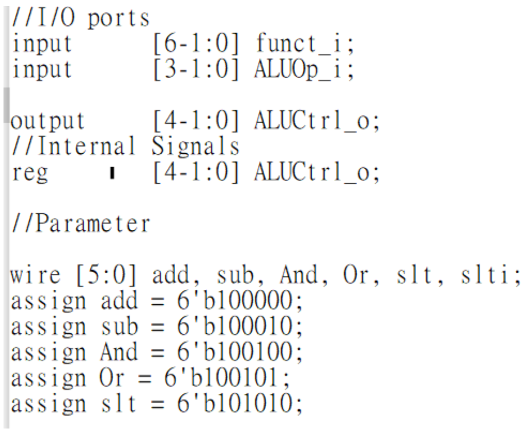
ID:109550112

Architecture diagrams:



Hardware module analysis:

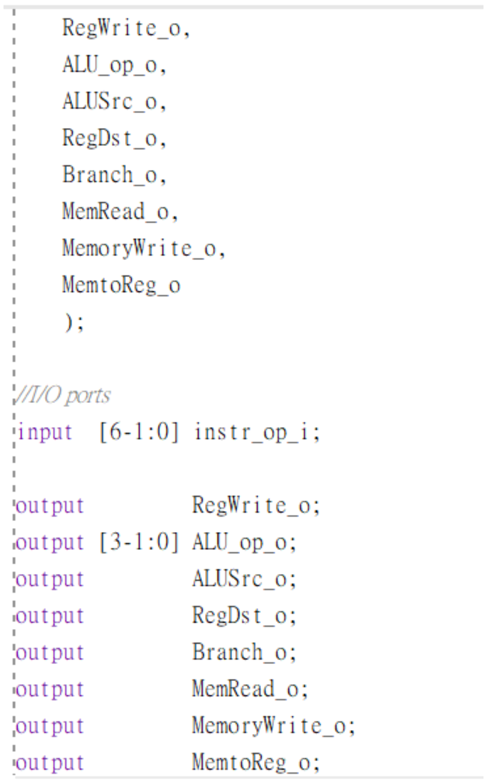
Comparing lab3, we don’t need to deal with jump, jal, etc. So we can omit some control bits, and can realize without 3to1 mux. Since most module is the same as lab3, I only introduce the changed part.
ALU_Ctrl:



```
//I/O ports
input      [6-1:0] funct_i;
input      [3-1:0] ALUOp_i;

output     [4-1:0] ALUCtrl_o;
//Internal Signals
reg        [4-1:0] ALUCtrl_o;

//Parameter

wire [5:0] add, sub, And, Or, slt, slti;
assign add = 6'b100000;
assign sub = 6'b100010;
assign And = 6'b100100;
assign Or = 6'b100101;
assign slt = 6'b101010;
```

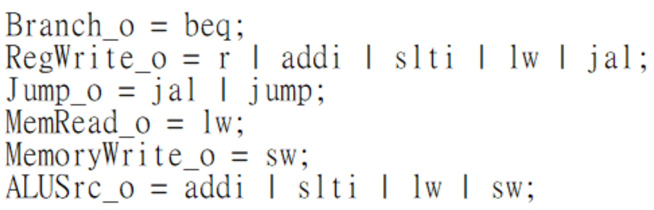
The module is the same except for the jump signal output.

Decoder:



```
    RegWrite_o,
    ALU_op_o,
    ALUSrc_o,
    RegDst_o,
    Branch_o,
    MemRead_o,
    MemoryWrite_o,
    MemtoReg_o
    );

//I/O ports
input  [6-1:0] instr_op_i;

output         RegWrite_o;
output [3-1:0] ALU_op_o;
output         ALUSrc_o;
output         RegDst_o;
output         Branch_o;
output         MemRead_o;
output         MemoryWrite_o;
output         MemtoReg_o;
```

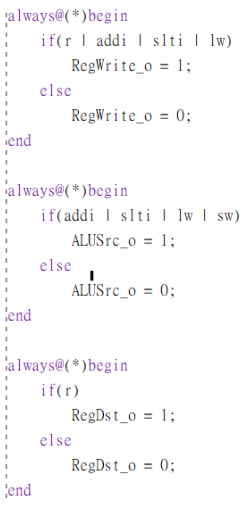
Again, the module deletes the jump , jal signal, so we san simplify the output signal, and save output signal bits like RegDst.



```
Branch_o = beq;
RegWrite_o = r | addi | slti | lw | jal;
Jump_o = jal | jump;
MemRead_o = lw;
MemoryWrite_o = sw;
ALUSrc_o = addi | slti | lw | sw;
```

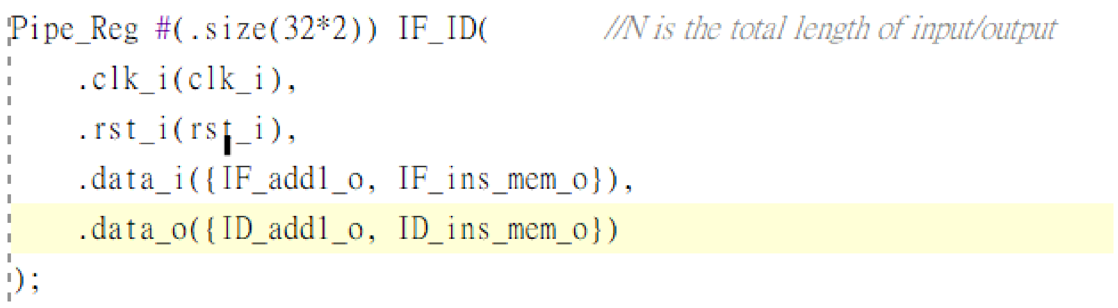
I used to write the output code as above, so I can avoid using a lot of unnecessary

always@(*), this works in Lab2 and 3, but fail when I testing, so I cover every signal with always@(*) this time.

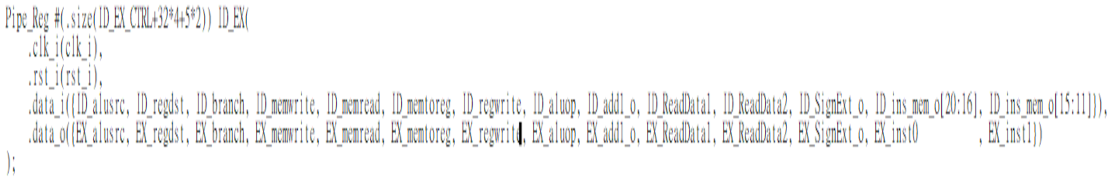

```
always@(*)begin
    if(r | addi | slti | lw)
        RegWrite_o = 1;
    else
        RegWrite_o = 0;
end

always@(*)begin
    if(addi | slti | lw | sw)
        ALUSrc_o = 1;
    else
        ALUSrc_o = 0;
end

always@(*)begin
    if(r)
        RegDst_o = 1;
    else
        RegDst_o = 0;
end
```

Pipe_reg:

I only introduce the stage register part since rest module is basically the same as lab3.

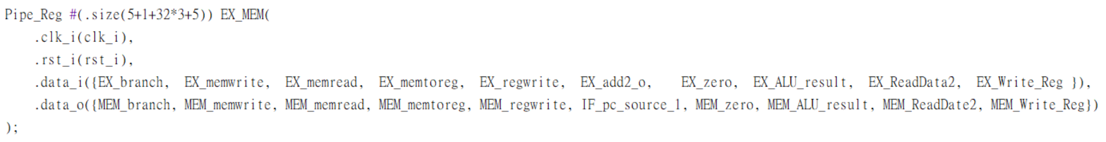

```
Pipe_Reg #(.size(32*2)) IF_ID(          //N is the total length of input/output
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({IF_add1_o, IF_ins_mem_o}),
    .data_o({ID_add1_o, ID_ins_mem_o})
);
```

IF/ID packs the next instruction sequence and the instruction. So, we need 32*2 bits.


```
Pipe_Reg #(.size(ID_EX_CTRL+32*4+5*2)) ID_EX(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({ID_alusrc, ID_regdst, ID_branch, ID_memwrite, ID_memread, ID_memtoreg, ID_regwrite, ID_aluop, ID_add1_o, ID_ReadData1, ID_ReadData2, ID_SignExt_o, ID_ins_mem_o[20:16], ID_ins_mem_o[15:11]}),
    .data_o({EX_alusrc, EX_regdst, EX_branch, EX_memwrite, EX_memread, EX_memtoreg, EX_regwrite, EX_aluop, EX_add1_o, EX_ReadData1, EX_ReadData2, EX_SignExt_o, EX_inst0          , EX_inst1})
);
```

ID/EX packs all the control signal, the instruction sequence of IF/ID, the data read from register, sign_extend address and the rs rt register number. So we need Control bytes and 4*32 bits for register data(2) the instruction sequence data(1) and sign extend address(1) and 10 bits for rs rt, 5 bits each.

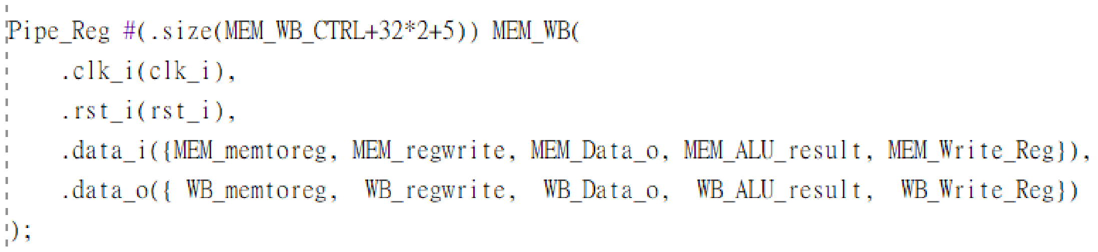
EX/MEM:


```
Pipe_Reg #(.size(5+1+32*3+5)) EX_MEM(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({EX_branch,  EX_memwrite,  EX_memread,  EX_memtoreg,  EX_regwrite,  EX_add2_o,    EX_zero,  EX_ALU_result,  EX_ReadData2,  EX_Write_Reg }),
    .data_o({MEM_branch, MEM_memwrite, MEM_memread, MEM_memtoreg, MEM_regwrite, IF_pc_source_1, MEM_zero, MEM_ALU_result, MEM_ReadDate2, MEM_Write_Reg})
);
```

This register packs branch for beq, and memory control bits for memory module,

and regwrite for mux total 5 bits, and 32 bits each for beq jump address, alu_result for arithmetic calculation or calculating address, and data2 for Registers. Finally we have a 5 bits Write_reg for arithmetic calculation instruction.

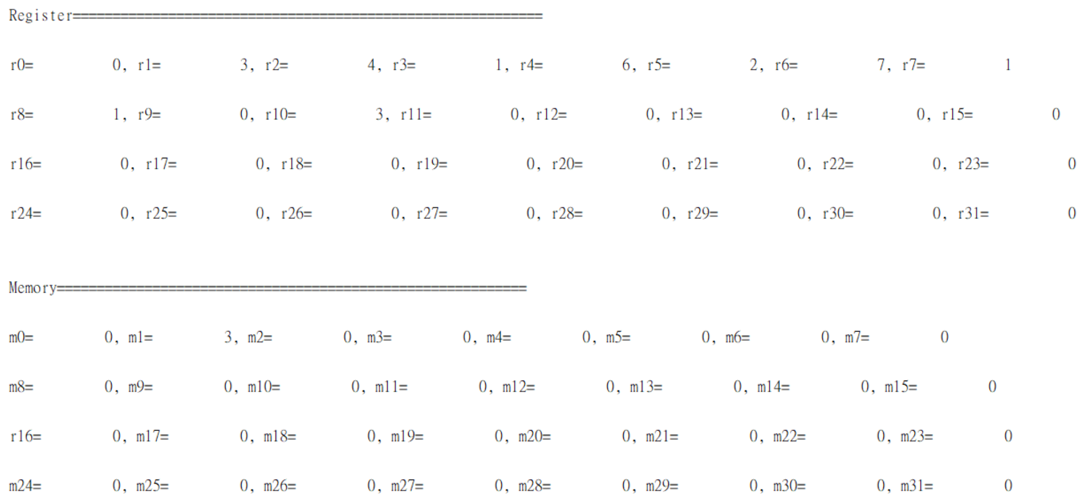
MEM/WB:



```
Pipe_Reg #(.size(MEM_WB_CTRL+32*2+5)) MEM_WB(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({MEM_memtoreg, MEM_regwrite, MEM_Data_o, MEM_ALU_result, MEM_Write_Reg}),
    .data_o({ WB_memtoreg,  WB_regwrite,  WB_Data_o,  WB_ALU_result,  WB_Write_Reg})
);
```

We store 2 control bits for memtoreg and regwrite . And store 32 bits each for ALU_output and memory data output, finally store 5 bits for Write_reg in EX/MEM section.

Finished part:

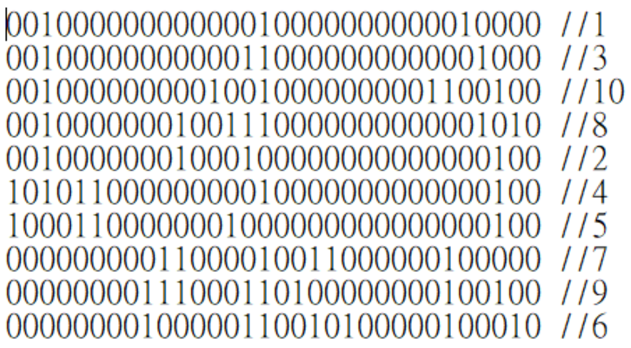


```
Register===========================================================

r0=        0, r1=        3, r2=        4, r3=        1, r4=        6, r5=        2, r6=        7, r7=        1

r8=        1, r9=        0, 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

Memory===========================================================

m0=        0, m1=        3, m2=        0, m3=        0, m4=        0, m5=        0, m6=        0, m7=        0

m8=        0, m9=        0, m10=       0, m11=       0, m12=       0, m13=       0, m14=       0, m15=       0

r16=       0, m17=       0, m18=       0, m19=       0, m20=       0, m21=       0, m22=       0, m23=       0

m24=       0, m25=       0, m26=       0, m27=       0, m28=       0, m29=       0, m30=       0, m31=       0
```

Problems you met and solutions:

As I mentioned in hardware module analysis, I find it weird to have the same coding style but with different results, I can’t figure the solution for the simplified version, so I use always for every control signal output. And I find it difficult to apply values in the pipe registers, so when I applying those values, I double care those values in case I make a mistake.

Bonus (optional):

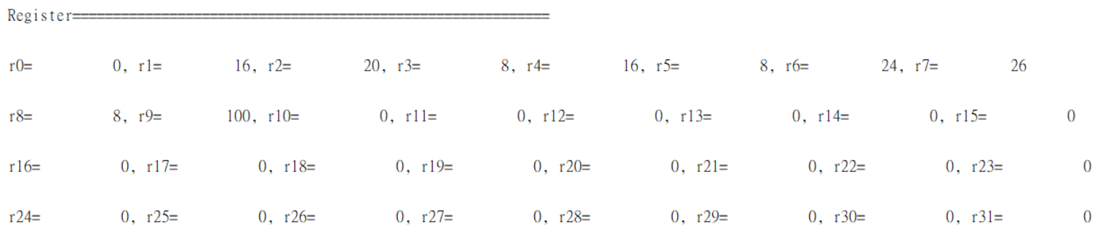


```
00100000000000010000000000010000  //1
00100000000000110000000000001000  //3
00100000000010010000000001100100  //10
00100000001001110000000000001010  //8
00100000001000100000000000000100  //2
10101100000000010000000000000100  //4
10001100000001000000000000000100  //5
00000000011000010011000000100000  //7
00000000111000110100000000100100  //9
00000000100000110010100000100010  //6
```

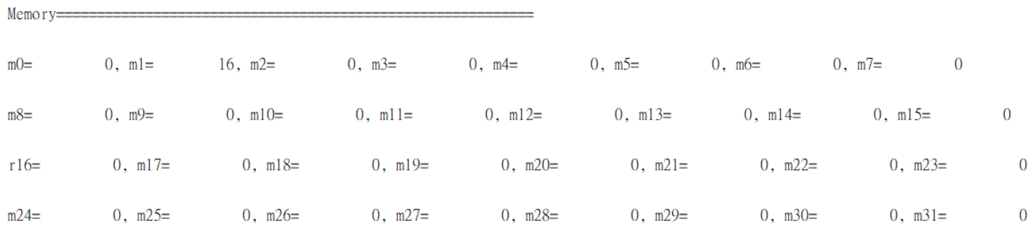
Sequence works as follows, the comment at end stands for the sequence order of the mips operation in pdf.

Since this cpu can't do forwarding, for the data hazard area, we need to add 2 instructions not using the former data which works for separating the hazard instructions .

Execution result:



```
Register=====================================================

r0=          0, r1=         16, r2=         20, r3=          8, r4=         16, r5=          8, r6=         24, r7=         26

r8=          8, r9=        100, 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
```



```
Memory=====================================================

m0=          0, m1=         16, m2=          0, m3=          0, m4=          0, m5=          0, m6=          0, m7=          0

m8=          0, m9=          0, m10=         0, m11=         0, m12=         0, m13=         0, m14=         0, m15=          0

r16=          0, m17=         0, m18=         0, m19=         0, m20=         0, m21=         0, m22=         0, m23=          0

m24=          0, m25=         0, m26=         0, m27=         0, m28=         0, m29=         0, m30=         0, m31=          0
```

Summary:

This is the most difficult lab, it requires lots of concepts and patience since the variables are a lot more than ever, so naming every variable is critical, after a lot of complicated variable checking, I managed to complete the lab. And I find the bonus is pretty inspiring, by solving the question, I get used to the data hazard more, so I think I understand the structure better now.