

## 1. Description of the implementation

### a. alu.v

```

113 //Lab3-code start
114 4'b1000:begin //xor
115     result=src1^src2;
116     overflow=0;
117     cout=0;
118 end
119 4'b1000:begin //sll
120     result=src1<<src2;
121     overflow=0;
122     cout=0;
123 end
124 4'b1001:begin //sra
125     result=src1>>>src2;
126     overflow=0;
127     cout=0;
128 end
129 default:begin //maybe exist other condition
130     result=32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
131     overflow=0;
132     cout=0;
133 end
134 //Lab3-code end

```

For implementing “alu.v”, I imported “alu\_1bit.v” from Lab02. There are five operations, and\or\add\sub\slt, in the Lab02. With the additional advanced operation, xor\sll\sra, I used operator to achieve the calculation.

**xor**: I implemented the operator “^” on the src1 and src2 and set both overflow and cout as ‘0’.

**sll**: I implemented the operator “<<” on the src1 and src2 and also set both overflow and cout as ‘0’.

**sra**: I implemented the operator “>>>” on the src1 and src2 and also set both overflow and cout as ‘0’.

In the end of this part, there was a **default** case. This case is for other conditions that we hadn’t mentioned. For example, if there exist a command “xand”, it would go to default case. But in Lab03, there doesn’t exist the exceptional case.

### b. ALU\_Ctrl.v

```

11 //Lab3-code start
12 always@(*)begin
13     case(ALUOp)
14     2'b10: //R-type
15         case(instr)
16             4'b0000:begin //add
17                 ALU_Ctrl_o=4'b0010;
18             end
19             4'b1000:begin //sub
20                 ALU_Ctrl_o=4'b0110;
21             end
22             4'b0111:begin //and
23                 ALU_Ctrl_o=4'b0000;
24             end
25             4'b0110:begin //or
26                 ALU_Ctrl_o=4'b0001;
27             end
28             4'b0100:begin //xor
29                 ALU_Ctrl_o=4'b1100;
30             end
31             4'b0010:begin //slt
32                 ALU_Ctrl_o=4'b0111;
33             end
34             4'b0001:begin //sll
35                 ALU_Ctrl_o=4'b1000;
36             end
37             4'b1101:begin //sra
38                 ALU_Ctrl_o=4'b1001;
39             end
40             default:begin
41                 ALU_Ctrl_o=4'bxxxx;
42             end
43         endcase
44     default:begin //other cases
45         ALU_Ctrl_o=4'bxxxx;
46     end
47 endcase
48 end
49 //Lab-3 code end

```

In this file, we need to detect whether this instruction is an R-type command or not. I used “case” to determine “ALUOp”. If “ALUOp” is equal to “10”, which means this instruction is an R-type instruction, it will go on to the second level of detecting which kind of instruction this command is. Otherwise, it will go to the default case and return “xxxx”.

In the instruction case, we will have a match table:

IO\Instr.	add	sub	and	or	xor	slt	sll	sra
Input	0000	1000	0111	0110	0100	0010	0001	1101
Output	0010	0110	0000	0001	1100	0111	1000	1001

For the ALU control of “xor\sll\sra”, I defined them as “1100\1000\1001” respectively. The others commands are based on the slide of Lab03.

There is also a default case for unexpected case. It will return “xxxx”, when the case of “instr” is not in the match table above.

### c. Adder.v

```

11 //Lab3-code start
12 assign sum_o=src1_i+src2_i;
13 //Lab3-code end

```

For the “Adder”, we just assign the sum of “src1\_i” and “src2\_i” to “sum\_o”, with using the operator “+”.

#### d. Decoder.v

```
19 //Lab3-code start
20 assign opcode=instr_i[6:0];
21 assign funct3=instr_i[14:12];
22
23 assign ALUSrc=1'b0;
24 assign RegWrite=1'b1;
25 assign Branch=1'b0;
26 assign ALUOp=2'b10;
27 //Lab3-code end
```

In the decoder, I assigned “instr\_i” to “opcode” from the sixth bit to the zeroth bit and assigned “instr\_i” to “funct3” from the fourteenth bit to the twelfth bit. It’s based on the instruction format of risc-v.

Since Lab03 only include the R-type instruction. I directly assigned “ALUSrc\RegWrite\Branch\ALUOp” as “0\1\0\10” respectively.

For “ALUSrc”, we always want the RS2 to go to the ALU in R-type, so the value of “ALUSrc” is “1”.

For “RegWrite”, we want the output data can be written to the RD after the calculation in ALU. So, I assigned this wire as “1”.

For “Branch”, it is assigned as “0”. Because we do not need to jump instructions, the “Branch” should be set to “0” to avoid something wrong with the program counter.

For “ALUOp”, it is based on the slide in the class to assigned as “10”.

#### e. Simple\_Single\_CPU.v

```
22 ProgramCounter PC(
23     .clk_i(clk_i),
24     .rst_i(rst_i),
25     .pc_i(pc_i),
26     .pc_o(pc_o)
27 );
28
29 Instr_Memory IM(
30     .addr_i(pc_o),
31     .instr_o(instr)
32 );
```

```
34 Reg_File RF(
35     .clk_i(clk_i),
36     .rst_i(rst_i),
37     .RSaddr_i(instr[19:15]),
38     .RTaddr_i(instr[24:20]),
39     .RDaddr_i(instr[11:7]),
40     .RDdata_i(ALUresult),
41     .RegWrite_i(RegWrite),
42     .RSdata_o(RSdata_o),
43     .RTdata_o(RTdata_o)
44 );
```

```
46 Decoder Decoder(
47     .instr_i(instr),
48     .ALUSrc(ALUSrc),
49     .RegWrite(RegWrite),
50     .Branch(branch),
51     .ALUOp(ALUOp)
52 );
53
54 Adder PC_plus_4_Adder(
55     .src1_i(pc_o),
56     .src2_i(imm_4),
57     .sum_o(pc_i)
58 );
```

```
60 ALU_Ctrl ALU_Ctrl(
61     .instr({instr[30],instr[14:12]}),
62     .ALUOp(ALUOp),
63     .ALU_Ctrl_o(ALU_control)
64 );
```

```
66 alu alu(
67     .rst_n(rst_i),
68     .src1(RSdata_o),
69     .src2(RTdata_o),
70     .ALU_control(ALU_control),
71     .result(ALUresult),
72     .zero(zero),
73     .cout(cout),
74     .overflow(overflow)
75 );
```

In “ProgramCounter PC”, I just put the corresponding wire to the correct place.

In “Instr\_Memory IM”, its address input is come from the output of program counter and it has a “instruction” output.

In “Reg\_File RF”, 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.

In “Decoder Decoder”, I also put the corresponding wire to the right place.

In “Adder PC\_plus\_4\_Adder”, the source1 is the output of the program counter, and source2 is an immediate value “4”. The output, sum of two sources, is going to be the input for the program

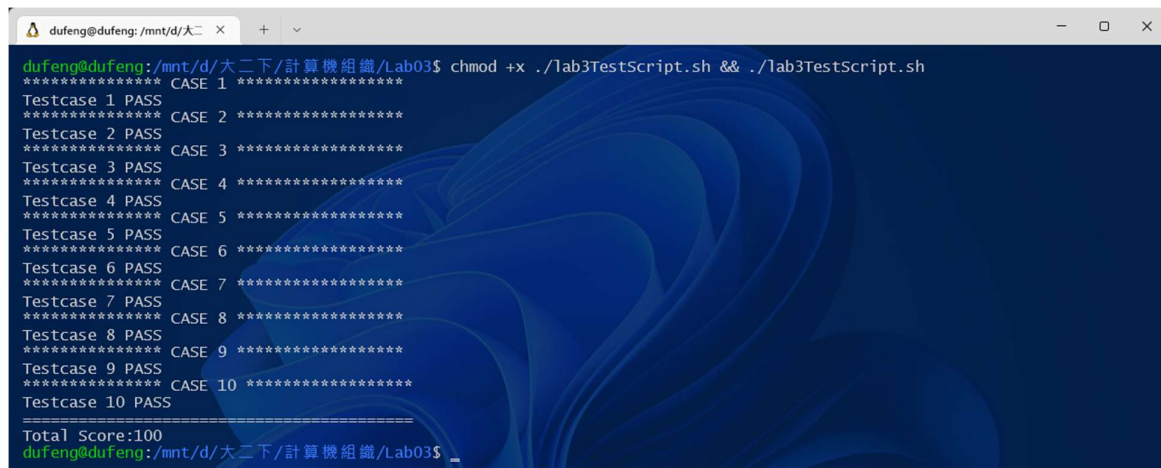
counter.

In “ALU\_Ctrl ALU\_Ctrl”, the instruction input contains the thirtieth bit and the bits from the fourteenth to twelfth in the instruction input. The last two wire just put the corresponding wire to them.

In “alu alu”, the input sources are the output of the “Reg\_File”, which named “RSdata\_o” and “RTdata\_o”. The result of ALU is called “ALUresult”. With the others just put the corresponding wire to them, too.

## 2. Implementation results

The results: all pass



```
dufeng@dufeng: /mnt/d/大二下/计算机组织/Lab03$ chmod +x ./lab3TestScript.sh && ./lab3TestScript.sh
***** CASE 1 *****
Testcase 1 PASS
***** CASE 2 *****
Testcase 2 PASS
***** CASE 3 *****
Testcase 3 PASS
***** CASE 4 *****
Testcase 4 PASS
***** CASE 5 *****
Testcase 5 PASS
***** CASE 6 *****
Testcase 6 PASS
***** CASE 7 *****
Testcase 7 PASS
***** CASE 8 *****
Testcase 8 PASS
***** CASE 9 *****
Testcase 9 PASS
***** CASE 10 *****
Testcase 10 PASS
=====
Total Score:100
dufeng@dufeng: /mnt/d/大二下/计算机组织/Lab03$
```

## 3. Problems encountered and solution

I had met a big problem in the implementation of “Decoder.v”. I know that the four wires of the output are from opcode. But I didn’t understand how to decode from opcode. I search for many information. And the result I find is it is just like a relation in the “map”. Such as the opcode in the Lab03, 0110011, is an R-type instruction, and it has corresponding bits set with the output. After surfing on the HackMD and seeing the TA’s responses to some questions, I know that there are all R-type instructions in Lab03. So, I just assigned “ALUSrc” as “0”, assigned “RegWrite” as “1”, assigned “Branch” as “0”, and “ALUOp” as “10”. But if next lab requires more types of function, this method cannot implement successfully. Maybe we need to use “case” to detect which types the instruction is.