1. Usage

Since my ALU is based on Linux, I wrote a makefile to help run more conveniently. Before running the program, you may need to make sure there exists **iverilog** and **gtkwave** in your Linux system(former is used to compile and simulate Verilog file, latter is used to display waveform).

If you do not have two things, please use the instructions below to install them:

```
$ sudo apt-get update
$ sudo apt-get install iverilog
$ sudo apt-get install gtkwave
```

After installing necessary dependencies, please use the instructions below to run proj.

```
$ cd ./ALU
$ make
```

2. Design Ideas

In the principle of computer operation, no matter what kind of high-level language, the final delivery to the CPU runtime will be converted into machine instructions. After the MIPS instructions are sent to the CPU, the controller will send specific control signals according to the operation code and functional code of the instructions (such as multiplexer selection signal, register selection signal, etc.). Each component of the data path will do data processing and transmission correctly according to these control signals. The ALU of this experiment is a very important component of the data path.

ALU is an arithmetic logic unit of the whole CPU, which is designed to compute two values (possibly from general registers, PC registers, instant numbers, etc.) according to the corresponding instructions. Because of its frequent calls, it can greatly reduce the computation time by using a combinational logic circuit.

Because this experiment implements ALU independently of the controller, the control signal is not taken into account in my design process. Conventional ALU inputs are A port, B port and control signal. Here we modify them to regA, regB and instruction, hoping to achieve real results when running a single command

2.1 Instruction Struction

According to the instruction structure of the MIPS instruction set, we can divide all MIPS instructions into three types : R, I, J as following:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	R_s	R_t	R_d	shamt	funct

Figure1 : R type Instruction

6 bits	5 bits	5 bits	16 bits
OP	R_s	R_t	

Figure2 : I type Instruction

6 bits	26 bits
OP	立即数

Figure3 : J type Instruction

Since only accomplishing ALU in this experiment, we do not consider J type instruction here. As you can see, the structure is very regular. We can identify each instruction by opcode and function code. This facilitates our programming by using switch–case statements based on opcode and functional code to define different computing processes for each instruction. Below is implemention.

```
module alu(instruction, regA, regB, result, flag);
        input  [31:0] instruction, regA, regB;
        output[31:0] result;
        output[2:0] flag;
        always @ (*)
          begin
          opcode = instruction[31:26];
          func = instruction[5:0];
          flag = 3'b0;
          case(opcode)
                begin
                      case(func)
```

Using this way, I implement following instructions:

R type:

add, sub, addu. subu, and, nor, or, xor, slt, sltu, sll, sllv, srl, srlv, sra, srav.

I type:

addi, addiu, andi, beq, bne, ori, xori, slti, sltiu, lw, sw

Notably, in the flag array output by our ALU module, the first is the overflow sign, the second is the negative sign, and the third is the zero sign.

In this project, we only consider **add, addi, sub** for the overflow flag. We judge the overflow flag by comparing two calculations with the symbol bits of the final result(actually, only two situations can be judged to overflow: 001 and 110).

We consider **beq** and **bne** for the zero flag. We determine the zero sign by comparing the subtraction of the two calculation numbers. If the subtraction result is zero, the zero sign is set as 1

We consider **slt, slti, sltu, sltiu** for negative flag. For **slt** and **slti**, we take the way of subtracting two operands with the highest result. For **sltu** and **sltiu,** we take the following results as markers:

$$rs < rt$$

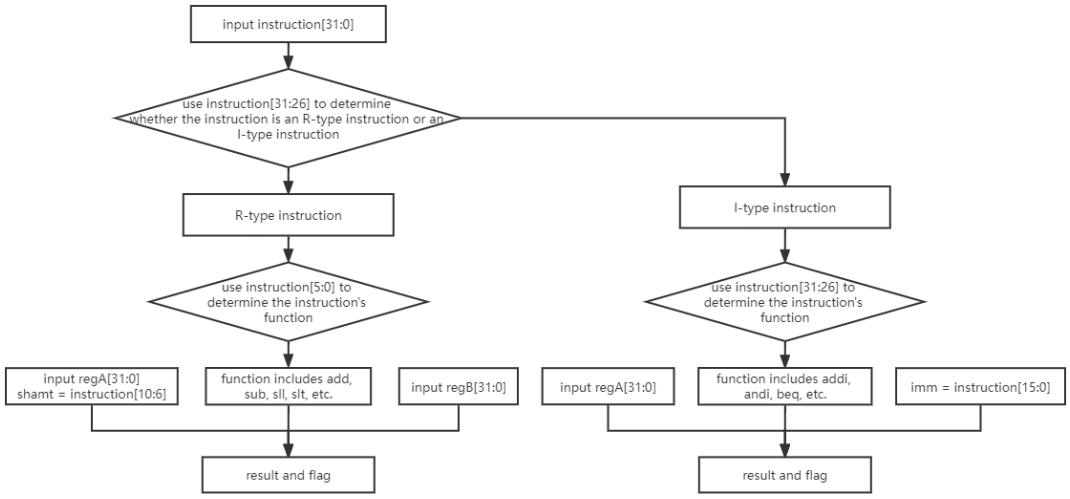

Figure4 : data flow chart

2.2 Other Details(Difference between − and <)

As you can see from the judgement of flags, we use two different methods to judge negative Flags. This is because the operands are distinguished between signed and unsigned. In instructions similar to **sltu**, the computer must treat the operands as unsigned numbers for calculation, while in instructions similar to **slt**, the computer must treat them as signed numbers for calculation.

The difference between − and < is that when we do subtraction, the computer will treat it as a complement, which means that two operands are symbolic, and when we use <, the computer will treat it as an unsigned number. This difference can be of great use when we do symbolic number operation and symbolic number operation.

3. Test Bench

With the help of the gtkwave tool, it is easy to display the test results in a waveform, and it is easy to judge the right and wrong. Meanwhile, We use two different methods to show out simulation results: one is waveform, the other is console output.

Here, I manually wrote the test.v file, and tested all the instructions for this experiment ´s ALU, including different symbol bits such as overflow and non−overflow, and finally got the correct results.

```
`timescale 1ns/1ps

module test();
reg  [31:0] instruction;
reg  [31:0] regA;
reg  [31:0] regB;
wire [31:0] result;
wire [2:0] flag;

 initial begin
   $dumpfile("test.vcd");
   $dumpvars;
 end

 initial begin
   $display("\n              initialize");
   $monitor("instruction = 32'h%h, regA = 32'h%h, regB = 32'h%h, result = 32'h%h, flag = 3'b%b",instruction,regA,regB,result,flag);
       instruction = 32'b0;
       regA = 32'b0;
       regB = 32'b0;
       #500
   $display("\n              for R Type");
   $display("\n              format is");
   $display("\n   instruction:regA:regB:result:flag");
   $display("\n              add overflow test");
       instruction = 32'h00000020;   // testing for add overflow
       regA = 32'h7FFFFFFF;
       regB = 32'h00000006;
       #100
```

Figure 5: test.v

Figure 6: test.vcd(Waveform)

```
iverilog -o run.out test.v alu.v
vvp -n run.out
VCD info: dumpfile test.vcd opened for output.

                                initialize
instruction = 32'h00000000, regA = 32'h00000000, regB = 32'h00000000, result = 32'h00000000, flag = 3'b000

                                for R Type

                                format is

                     instruction:regA:regB:result:flag

                              add overflow test
instruction = 32'h00000020, regA = 32'h7fffffff, regB = 32'h00000006, result = 32'h80000005, flag = 3'b001

                              add normal test
instruction = 32'h00000020, regA = 32'h00000001, regB = 32'h00000002, result = 32'h00000003, flag = 3'b000

                              sub overflow test
instruction = 32'h00000022, regA = 32'h7fffffff, regB = 32'hffffffff, result = 32'h80000000, flag = 3'b001

                              sub normal test
instruction = 32'h00000022, regA = 32'h00000004, regB = 32'h00000003, result = 32'h00000001, flag = 3'b000

                                addu test
instruction = 32'h00000021, regA = 32'h7fffffff, regB = 32'h00000006, result = 32'h80000005, flag = 3'b000

                                subu test
instruction = 32'h00000023, regA = 32'h7fffffff, regB = 32'hffffffff, result = 32'h80000000, flag = 3'b000

                                and test
instruction = 32'h00000024, regA = 32'hffffffff, regB = 32'h00000001, result = 32'h00000001, flag = 3'b000

                                 or test
instruction = 32'h00000025, regA = 32'h00000000, regB = 32'h00000001, result = 32'h00000001, flag = 3'b000

                                xor test
instruction = 32'h00000026, regA = 32'hffffffff, regB = 32'h0000000f, result = 32'hfffffff0, flag = 3'b000
```

Figure 7: test.vcd(Console Output)