Project 2 Report

[Big picture thoughts and ideas]

What should be done is to implement an ALU by Verilog. Therefore, the first idea is to make clear what ALU is and the functions in ALU in reality.

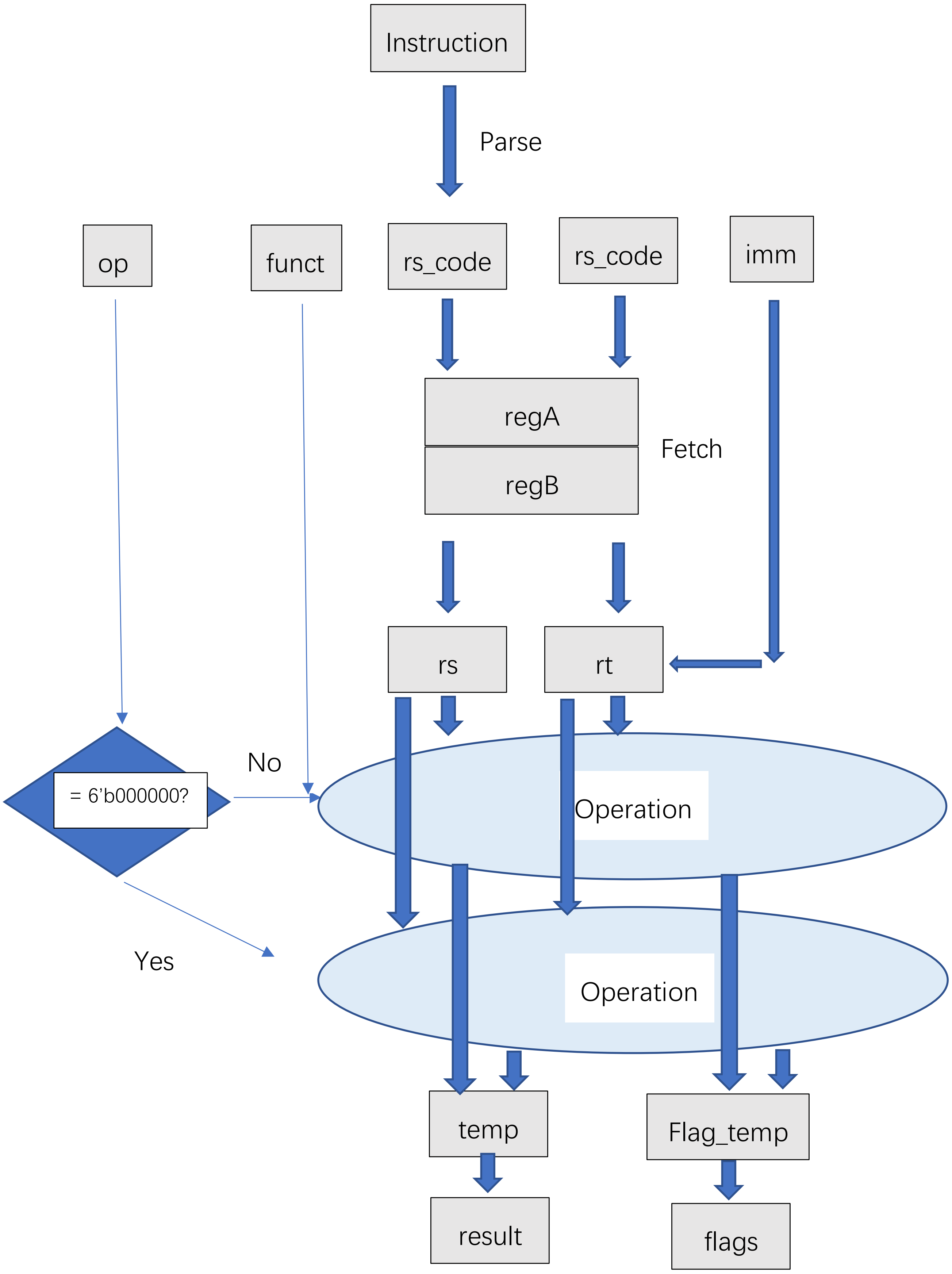
ALU is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. The inputs of the ALU are the operands, which are the data in the registers to be operated on, and the code indicating which operation should be performed. The outputs are the result of the performed operation and the flags which indicate the status.

To implement the inputs and outputs of ALU, I do follow things. First, for the operands, I use **rs** and **rt** to store the integer binary numbers. The type of **rs** and **rt** is register. The data is initially stored in the registers (use **regA** and **regB** to represent). Then, the data in registers will be fetched according to the instruction and store into **rs** and **rt**. **rs** and **rt** will be operated on as the role of operands. For the other input: opcode, I use **op** and **funct** of the instruction to represent. Once parse the instruction outside the ALU, I will get **op** and **funct**. **Op** and **funct** will play the role of opcode of ALU to recognize what operation will be performed in ALU. For the outputs, variables **result** and **flags** will be the output ports that plays the role of ALU's outputs. For the flags, I use a 3 bits binary number to representing the status. The first bit of flags (flags [0]) is the zero flag. The second bit of flags (flags [1]) is the negative flag. The third bit of flags (flags [3]) is the overflow bit.

The functions of ALU in reality, can be divided into three parts, (1) Arithmetic operations, (2) Bitwise logical operations, and (3) Bit shift operations.

For the (1) Arithmetic operations, there are **Add**, **Addi**, **Addu**, **Addiu**, **Sub** and **Subu**, **Beq**, **Bne**, **Lw**, **Sw**. I directly use +, **-** operations in Verilog to implement the full adder in ALU and use ==, < operations in Verilog to implement the comparator. Also, I will update and output the status as the real ALU does. For the (2) Bitwise logical operations, there are **And**, **Andi**, **Nor**, **Or**, **Ori**, **Xor**, **Xori**. I directly use the |, **&**, ^, operations in Verilog to implement the bitwise logical operations in real ALU. For the (3) Bit shift operations, there are **Sll**, **Sllv**, **Srl**, **Srlv**, **Sra**, **Srav**. I use << and >> to implement the logical shift left and logical shift right. Also, I use <<< and >>> to implement the arithmetic shift left and arithmetic shift right.

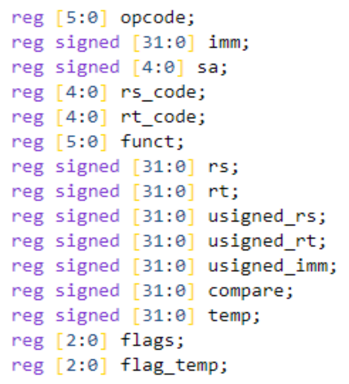
[Data flow Chart]



[High-Level implementation]

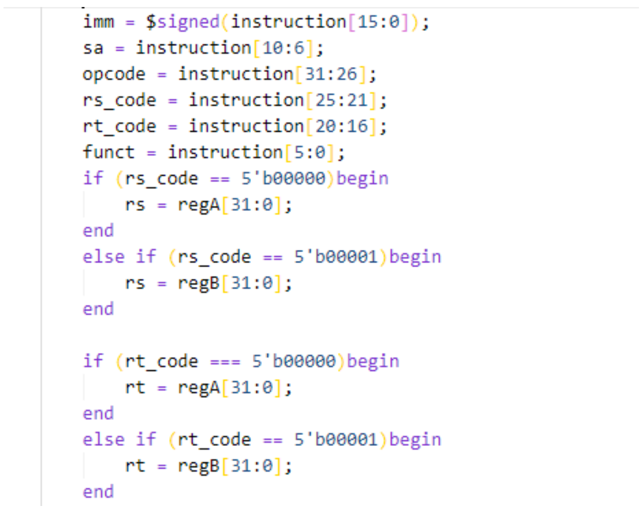
I divided the implementation into 3 parts. The (1) declaration part (2) fetching part (3) operation part.

(1) For the declaration part, I declare some variable using to store the data. Some data fetching from registers will be stored in it and input into the ALU, and be operated on.



```
reg [5:0] opcode;
reg signed [31:0] imm;
reg signed [4:0] sa;
reg [4:0] rs_code;
reg [4:0] rt_code;
reg [5:0] funct;
reg signed [31:0] rs;
reg signed [31:0] rt;
reg signed [31:0] usigned_rs;
reg signed [31:0] usigned_rt;
reg signed [31:0] usigned_imm;
reg signed [31:0] compare;
reg signed [31:0] temp;
reg [2:0] flags;
reg [2:0] flag_temp;
```

(2) For the fetching part, I first parse the code of instruction to know which registers should be put as the inputs.



```
imm = $signed(instruction[15:0]);
sa = instruction[10:6];
opcode = instruction[31:26];
rs_code = instruction[25:21];
rt_code = instruction[20:16];
funct = instruction[5:0];
if (rs_code == 5'b00000)begin
    rs = regA[31:0];
end
else if (rs_code == 5'b00001)begin
    rs = regB[31:0];
end

if (rt_code === 5'b00000)begin
    rt = regA[31:0];
end
else if (rt_code == 5'b00001)begin
    rt = regB[31:0];
end
```

(3) For the operation part, **rs**, **rt**, **op** and **funct** will be sent as input. **rs**, **rt** will be operated on, and the result will be stored into the **temp** (an intermediate variable for result) and **flag_temp** (an intermediate variable

for flags). A part of the operation: (implement of Add operation)

```
case(opcode)
// Add
    Rtype: begin
        case(funct)
            Add: begin
                temp = rs + rt;
                if (rs[31] == 1'b1 && rt[31] == 1'b1 && temp[31] == 1'b0)begin
                    flag_temp = 3'b100;
                end
                if (rs[31] == 1'b0 && rt[31] == 1'b0 && temp[31] == 1'b1)begin
                    flag_temp = 3'b100;
                end
            end
```

[Implementation details]

(1) Use parameters to represent **op** and **funct**:

```
parameter Rtype = 6'b000000, Add = 6'b100000, Addi = 6'b001000, Addu = 6'b100001, Addiu = 6'b001001,
Sub = 6'b100010, Subu = 6'b100011, And = 6'b100100, Andi = 6'b001100, Nor = 6'b100111, Or = 6'b100101,
Ori = 6'b001101, Xor = 6'b100110, Xori = 6'b001110, Beq = 6'b000100, Bne = 6'b000101, Slt = 6'b101010,
Slti = 6'b001010, Sltiu = 6'b001011, Sltu = 6'b101011, Lw = 6'b100011, Sw = 6'b101011, Sll = 6'b000000,
Sllv = 6'b000100, Srl = 6'b000010, Srlv = 6'b000110, Sra = 6'b000011, Srav = 6'b000111;
```

(2) The zero extension and signed extension:

```
imm = $signed(instruction[15:0]);
```

```
imm = {16'b0, imm[15:0]};
```

(3) Check the overflow of Add, Sub operation:

```
// Add
    Rtype: begin
        case(funct)
            Add: begin
                temp = rs + rt;
                if (rs[31] == 1'b1 && rt[31] == 1'b1 && temp[31] == 1'b0)begin
                    flag_temp = 3'b100;
                end
                if (rs[31] == 1'b0 && rt[31] == 1'b0 && temp[31] == 1'b1)begin
                    flag_temp = 3'b100;
                end
            end
```

```
Sub: begin
    temp = rs - rt;
    if (rs[31] == 1'b1 && rt[31] == 1'b0 && temp[31] == 1'b0)begin
        flag_temp = 3'b100;
    end
    if (rs[31] == 1'b0 && rt[31] == 1'b1 && temp[31] == 1'b1)begin
        flag_temp = 3'b100;
    end
end
```