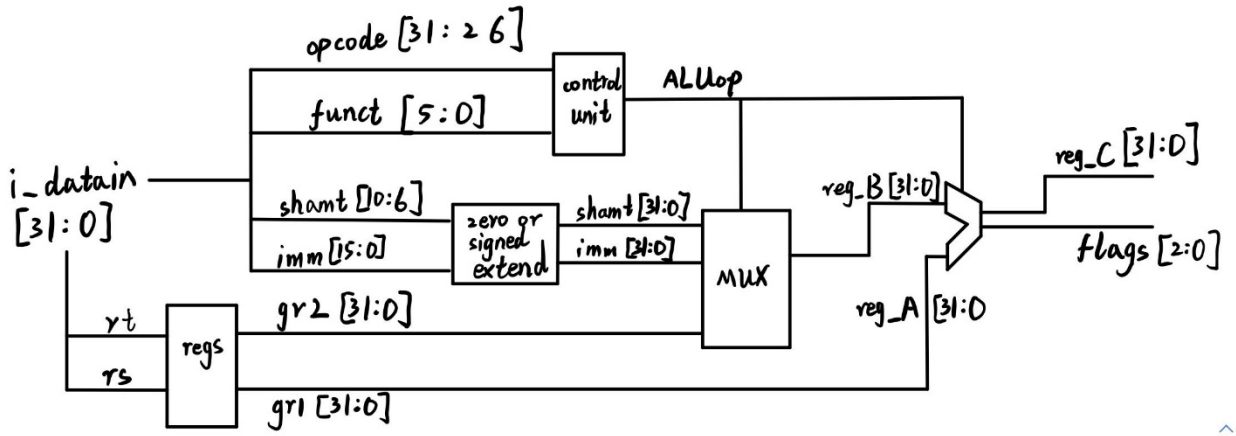
In this project, the ALU is implemented in “ALU.v” file, and the “test_ALU.v” is used to test the functions of this ALU. Users can use “make” to compile and run this project.

Big Picture Thoughts and Ideas

The whole project is divided into three stages. First, according to the input instruction, find the data that we need to input to ALU, defined them as “unsigned_reg_A or reg_A” and “unsigned_reg_B or reg_B”. Then, according to the input instruction, find the opcode and let ALU do the corresponding operation with “unsigned_reg_A or reg_A” and “unsigned_reg_B or reg_B”. Last, output the operating result and flags (zero, negative, and overflow), defined them as “reg_C (c)” and “flags”.

Data Flow Chart



The High-Level Implementation Ideas

For the first stage, we first divide flags into “reg zero, negative, overflow” because the “wire” type can’t be used in “always” block in Verilog. Then we define “unsigned_reg_A or reg_A = gr1” because that there is always an input of ALU will be the value stored in rs address. Then we split the instruction to get opcode, funct, gr2, imm and shamt. According to the opcode and funct, we can know what the instruction is, and so we can choose “unsigned_reg_B” from gr2, imm and shamt. Last, according the function of the instruction, we can determine whether the ALU operation requires signed or unsigned numbers, so we choose one of reg_A and unsigned_reg_A as one of the input of the ALU, choose one of reg_B and unsigned_reg_B as the another input of the ALU.

For the second stage, since we have gotten the opcode and funct in stage 1, we can determine what the instruction is directly. Then, we should simulate this instruction's function. So, in this part, we need to implement the required instructions' operation in the ALU.

For the third stage, we just use "assign c = reg_C[31:0]" and "assign flags = {zero, negative, overflow}" to get the output.

The Implementation Details

First, we can split the instruction to get opcode, funct, i.e., "opcode = i_datain[31:26]; func = i_datain[5:0];". Then, we use funct and opcode to determine the instruction. According to the instruction's function, we get sign extend imm by "imm = {{16{i_datain[15]}} ,i_datain[15:0]};", zero extend imm by "imm = {{16{1'b0}}, i_datain[15:0]};", zero extend shamt by "shamt = {{27{1'b0}, i_datain[10:6]}" and sign extend shamt by by "shamt = {{27{ i_datain[10]}, i_datain[10:6]}". Then, according to the instruction, choose one of the above four value or gr2 as "reg_B" or "unsigned_reg_B". Then, choose one of "reg_B" and "unsigned_reg_B" as one of the input of the ALU. Another input of the ALU is gr1. We let "reg_A = gr1 and "unsigned_reg_A = gr1, then choose one of "reg_A" and "unsigned_reg_A" as another input according to the instruction's function. One thing we need to prepare is that the flags need to be divided into zero, negative and overflow 3 regs so that we can operate flags in the always block. Because we implement every instruction in the always block.

Second, we implement every instruction's function. Since the instructions are too many, I just explain some special trick here, the details can be seen in "ALU.v" file. For overflow detection, we can sign extend reg_A and reg_B, then let a reg "ovf" be the result[32], if result is just [31,0], then "ovf" equal to 0, e.g., "{ovf, reg_C} = {reg_A[31], reg_A} - {reg_B[31], reg_B}". Then we can detect overflow by comparing reg_C[31] and ovf, if ovf and reg_C[31] are same, it doesn't overflow, else, it overflow. For beq/bne instructions, since they use sub to judge in the ALU, so the output of them is "reg_C = reg_A – reg_B", and if reg_C = 0, then zero flag will be 1, else, the zero flag will be 0. For srl/srlv/sra/srav, I use ">>> for srl\srlv and ">>" for sra\srav, because that ">>" is arithmetic right shift operator and ">>>" is logical right shift operator. For negative flag

of slt/sltu/stli/sltiu, in the ALU, it judges by subtract 2 inputs, and if the difference is negative, the negative flag will be set to 1, else it will be set to 0, but the output of the ALU will be “reg_A – reg_B”. For any other instructions’ implement details, please see the “ALU.v” file.

Third, since in the always block, we divided flags into zero, negative and overflow 3 parts, so after always block, we should combine them together to get flags again. We can do this directly using Verilog's assignment and concatenation syntaxes, i.e., “assign flags = {zero, negative, overflow}”. And then we also use it to get the output of this module, i.e., “assign c = reg_C[31:0]”.