Report on Project 2

Big picture thoughts and ideas

To implement a simple CPU module in project 2, we need to know how a CPU works.

First, what we input each time is a 32-bit MIPS instruction, regA and regB storing 32-bit value.

Then, parse an instruction into opcode, rs, rt, rd, shamt, func, imm(stands for immediate).

According to the opcode, we can decide which operation ALU should take except the R type instructions.

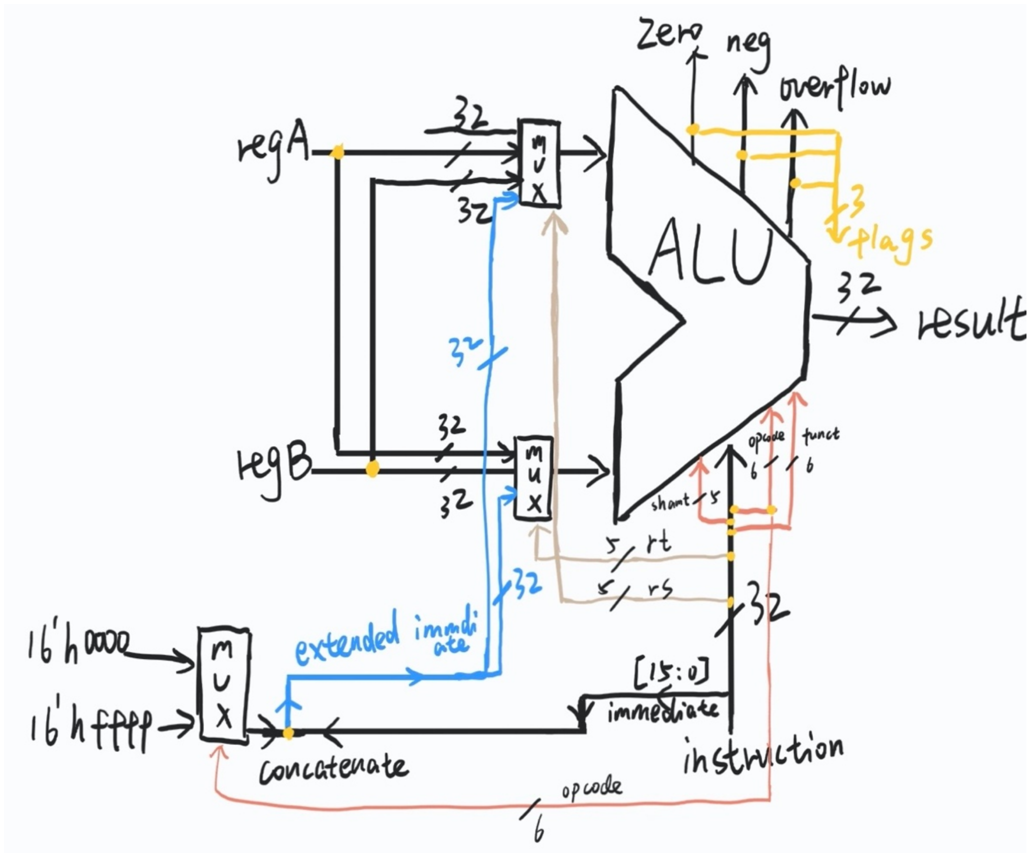
Furthermore, according to the func, we can decide which R-type instruction is and ALU will perform its corresponding operation, say, “add” and “xor”.

After knowing the operation, we fetch data that the operation needs which may store in regA, regB, shamt, func or imm.

Then we process the data, say, doing sign/zero-extension to imm(for I type instruction) or slicing the 32-bit value stored in regA or regB to get the shamt(for sllv, srlv and srav).

Finally, we can use these processed data, do corresponding operation and receive the result.

A data flow chart



The detailed running process for instruction

addu

The **addu** operation provides addition with rs and rt. The overflow flag will be ignored.

instruction	reg_A	reg_B
000000 00000 00001 00010 00001 100001	1000_0000_0000_0000 0000_0000_0000_0000	1111_1111_1111_1111 0000_0000_0000_0000
000000 00000 00001 00010 00010 100001	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00100 100001	0000_0000_0000_0000 0000_0000_0000_0001	0111_1111_1111_1111 1111_1111_1111_1110

Output:

reg_A	reg_B	result	zero	neg	overflow
80000000	ffff0000	7fff0000	0	0	0
00000001	fffffffe	ffffffff	0	1	0
00000001	7ffffffe	7fffffff	0	0	0

The operation is **addu** because the opcode is 000000 and the func is 100001. “rs” is 00000 which means we need to fetch regA. “rt” is 00001 which means we need to fetch regB. Then we add regA and regB and get the result. If the result is negative, the neg flag will be 1.

add

The **add** operation provides addition with rs and rt. We only care about the overflow flag so zero flag and negative flag will be ignored.

instruction	reg_A	reg_B
000000 00000 00001 00000 00000 100000	0000_0000_0000_0000 0000_0000_0000_0101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00000 00010 100000	0111_1111_1111_1111 1111_1111_1111_1111	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00000 00010 100000	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1110

Output:

reg_A	reg_B	result	zero	neg	overflow
00000005	00000001	00000006	0	0	0
7fffffff	00000001	80000000	0	0	1
00000001	fffffffe	ffffffff	0	0	0

The operation is **add** because the opcode is 000000 and the func is 100000. “rs” is 00000 which means we need to fetch regA. “rt” is 00001 which means we need to fetch regB. Then we add regA and regB and get the result. If the result overflows, the overflow flag will be 1.

Note that the running processes of **sub, subu, and, or, xor, nor** are quite similar to **add** and **addu**. Determine the operation by opcode and func. Fetch the data and do the corresponding operation and get the result. Thus, they will not be illustrated here.

How to check if the result of addition and subtraction overflow?

If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

Thus, we can check if result[31] equals to addend[31]. If they are equal, no overflow. If not, overflow happens.

If 2 Two's Complement numbers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend.

Thus, we can check if result[31] equals to subtrahend[31]. If they are equal, overflow happens. If not, no overflow.

slt

The **slt** operation compares rs and rt. If rs < rt, set the negative flag to 1.

instruction	reg_A	reg_B
000000 00000 00001 00000 00000 101010	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00000 00010 101010	1011_1111_1111_1111 1111_1111_1111_1111	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00000 00010	0000_0000_0000_0000	0000_0000_0000_0000

101010	0000_0000_0000_0111	0000_0000_0000_0111

Output:

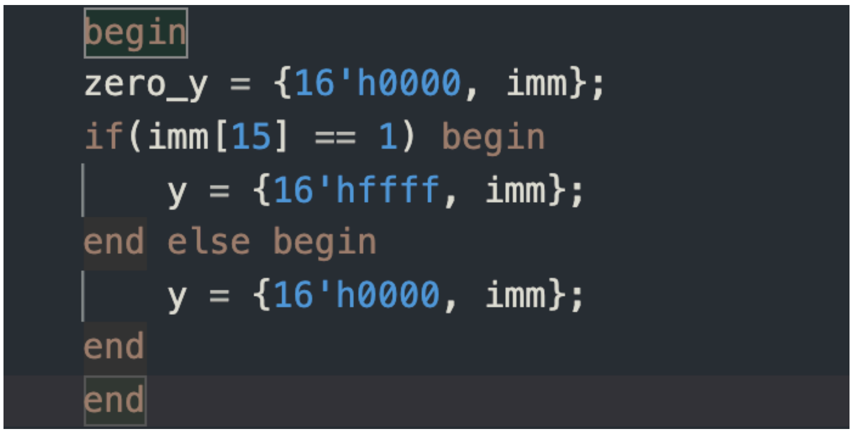
reg_A	reg_B	result	zero	neg	overflow
00000001	00000002	00000000	0	1	0
bfffffff	00000001	00000000	0	1	0
00000007	00000007	00000000	0	0	0

The operation is **slt** because the opcode is 000000 and the func is 101010. “rs” is 00000 which means we need to fetch regA. “rt” is 00001 which means we need to fetch regB. Then we compare regA and regB and set the flag. If regA < regB, neg flag is 1. Otherwise, the flag is 0.

To compare two numbers, first we need to compare their 32nd bit to see their sign. Then we compare their value. Since **sltu, beq** and **bne** are similar to **slt**, their tables will not be shown here.

For I type instruction, the only difference from R type instruction is to change an operand which should be a 32-bit value stored in register in R type instruction to an extended immediate in I type instruction. The rest is the same. Thus, the detailed running process of I type will not be repeated again and we focus on the following problem.

How to do zero extend and sign extend for I type instruction?



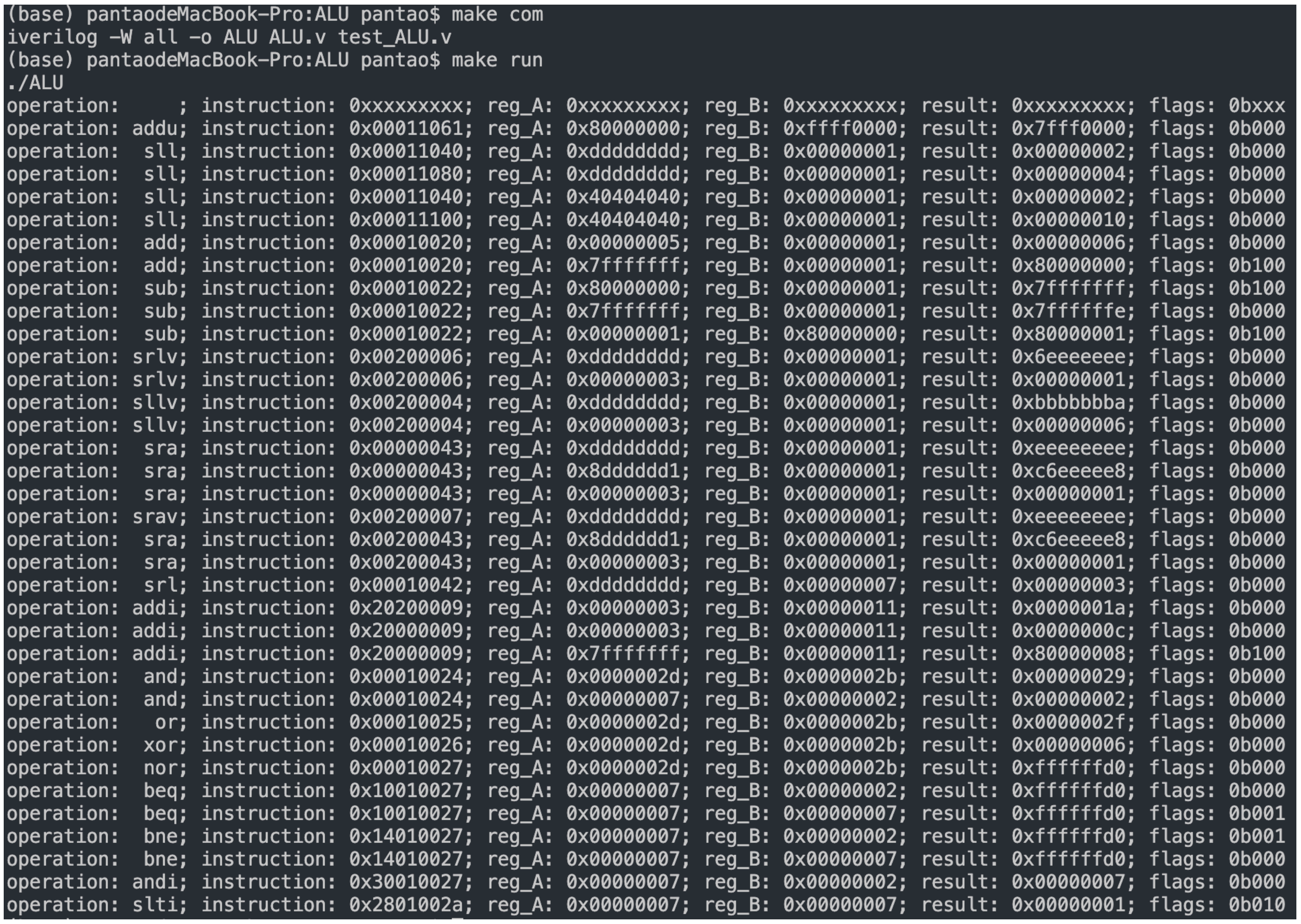
```
begin
zero_y = {16'h0000, imm};
if(imm[15] == 1) begin
    y = {16'hffff, imm};
end else begin
    y = {16'h0000, imm};
end
end
```

How to run the program?

First, enter the folder where ALU.v and test_ALU.v exist and enter “**make com**”(to compile the program).

Then, enter “**make run**” and it will show the test results on the terminal.

Here is an example:



```
(base) pantaodeMacBook-Pro:ALU pantao$ make com
iverilog -W all -o ALU ALU.v test_ALU.v
(base) pantaodeMacBook-Pro:ALU pantao$ make run
./ALU
operation:     ; instruction: 0xxxxxxxxx; reg_A: 0xxxxxxxxx; reg_B: 0xxxxxxxxx; result: 0xxxxxxxxx; flags: 0bxxx
operation: addu; instruction: 0x00011061; reg_A: 0x80000000; reg_B: 0xffff0000; result: 0x7fff0000; flags: 0b000
operation:  sll; instruction: 0x00011040; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0x00000002; flags: 0b000
operation:  sll; instruction: 0x00011080; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0x00000004; flags: 0b000
operation:  sll; instruction: 0x00011040; reg_A: 0x40404040; reg_B: 0x00000001; result: 0x00000002; flags: 0b000
operation:  sll; instruction: 0x00011100; reg_A: 0x40404040; reg_B: 0x00000001; result: 0x00000010; flags: 0b000
operation:  add; instruction: 0x00010020; reg_A: 0x00000005; reg_B: 0x00000001; result: 0x00000006; flags: 0b000
operation:  add; instruction: 0x00010020; reg_A: 0x7fffffff; reg_B: 0x00000001; result: 0x80000000; flags: 0b100
operation:  sub; instruction: 0x00010022; reg_A: 0x80000000; reg_B: 0x00000001; result: 0x7fffffff; flags: 0b100
operation:  sub; instruction: 0x00010022; reg_A: 0x7fffffff; reg_B: 0x00000001; result: 0x7ffffffe; flags: 0b000
operation:  sub; instruction: 0x00010022; reg_A: 0x00000001; reg_B: 0x80000000; result: 0x80000001; flags: 0b100
operation: srlv; instruction: 0x00200006; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0x6eeeeeee; flags: 0b000
operation: srlv; instruction: 0x00200006; reg_A: 0x00000003; reg_B: 0x00000001; result: 0x00000001; flags: 0b000
operation: sllv; instruction: 0x00200004; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0xbbbbbbba; flags: 0b000
operation: sllv; instruction: 0x00200004; reg_A: 0x00000003; reg_B: 0x00000001; result: 0x00000006; flags: 0b000
operation:  sra; instruction: 0x00000043; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0xeeeeeeee; flags: 0b000
operation:  sra; instruction: 0x00000043; reg_A: 0x8dddddd1; reg_B: 0x00000001; result: 0xc6eeeee8; flags: 0b000
operation:  sra; instruction: 0x00000043; reg_A: 0x00000003; reg_B: 0x00000001; result: 0x00000001; flags: 0b000
operation: srav; instruction: 0x00200007; reg_A: 0xdddddddd; reg_B: 0x00000001; result: 0xeeeeeeee; flags: 0b000
operation:  sra; instruction: 0x00200043; reg_A: 0x8dddddd1; reg_B: 0x00000001; result: 0xc6eeeee8; flags: 0b000
operation:  sra; instruction: 0x00200043; reg_A: 0x00000003; reg_B: 0x00000001; result: 0x00000001; flags: 0b000
operation:  srl; instruction: 0x00010042; reg_A: 0xdddddddd; reg_B: 0x00000007; result: 0x00000003; flags: 0b000
operation: addi; instruction: 0x20200009; reg_A: 0x00000003; reg_B: 0x00000011; result: 0x0000001a; flags: 0b000
operation: addi; instruction: 0x20000009; reg_A: 0x00000003; reg_B: 0x00000011; result: 0x0000000c; flags: 0b000
operation: addi; instruction: 0x20000009; reg_A: 0x7fffffff; reg_B: 0x00000011; result: 0x80000008; flags: 0b100
operation:  and; instruction: 0x00010024; reg_A: 0x0000002d; reg_B: 0x0000002b; result: 0x00000029; flags: 0b000
operation:  and; instruction: 0x00010024; reg_A: 0x00000007; reg_B: 0x00000002; result: 0x00000002; flags: 0b000
operation:   or; instruction: 0x00010025; reg_A: 0x0000002d; reg_B: 0x0000002b; result: 0x0000002f; flags: 0b000
operation:  xor; instruction: 0x00010026; reg_A: 0x0000002d; reg_B: 0x0000002b; result: 0x00000006; flags: 0b000
operation:  nor; instruction: 0x00010027; reg_A: 0x0000002d; reg_B: 0x0000002b; result: 0xffffffd0; flags: 0b000
operation:  beq; instruction: 0x10010027; reg_A: 0x00000007; reg_B: 0x00000002; result: 0xffffffd0; flags: 0b000
operation:  beq; instruction: 0x10010027; reg_A: 0x00000007; reg_B: 0x00000007; result: 0xffffffd0; flags: 0b001
operation:  bne; instruction: 0x14010027; reg_A: 0x00000007; reg_B: 0x00000002; result: 0xffffffd0; flags: 0b001
operation:  bne; instruction: 0x14010027; reg_A: 0x00000007; reg_B: 0x00000007; result: 0xffffffd0; flags: 0b000
operation: andi; instruction: 0x30010027; reg_A: 0x00000007; reg_B: 0x00000002; result: 0x00000007; flags: 0b000
operation: slti; instruction: 0x2801002a; reg_A: 0x00000007; reg_B: 0x00000007; result: 0x00000001; flags: 0b010
```

The first line can be ignored. The output of flags shown on the terminal is in binary and from right to left, the first bit is zero flag, the second bit is negative flag and the third bit is overflow flag.