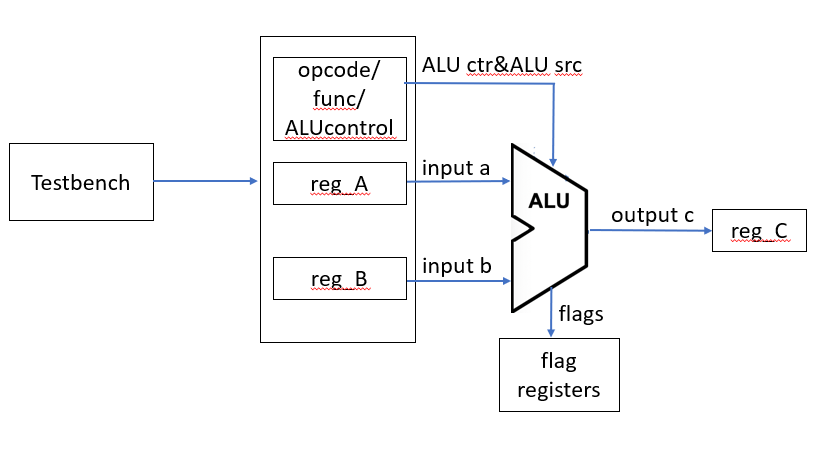
**Project Report**

**李华悦 118010138**

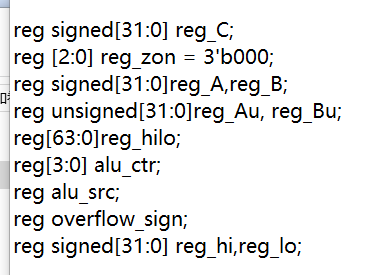
**1 understanding of this project:**

The purpose of this project is to write an ALU which will show the corresponding output for the instruction if you take a instruction and values input. The ALU will first send the instruction and values to the ALU file and then the control unit part will recognize the instruction and integrate all the information like opcode, func and so on to generate aluctr and alusrc. Then the ALU will recognize the aluctr and alusrc to allocate values to registers and then implement corresponding function of that instruction. Finally, the ALU will print the outputs and flags. In short, we will execute ALU through Verilog code.

**2 block diagram:**



The registers I have used are as follows:



The reg\_C is used to store the value of ouput C.

The reg\_zon is used to store the flags. In this register, reg\_zon[0] stores negative flag. reg\_zon[1] stores overflow flag. reg\_zon[2] stores zero flag.

The reg\_A and reg\_B are used to store the value of input a and b.

The unsigned reg\_Au and reg\_Bu are used to store the value of unsigned input a and b for some special instructions.

The reg\_hilo is used to store the result of mult instruction.

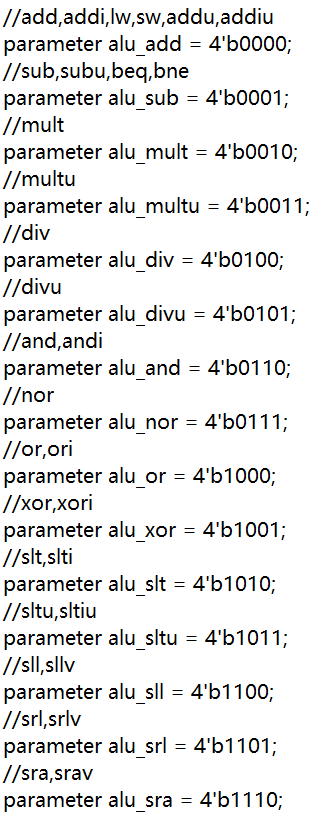
The alu\_ctr is used to store the alu control which is 4 bit to recognize the instruction and call the corresponding function.

The alu\_src is used to store the alu src which is used to judge if the register needs the value of general register or immediate value.

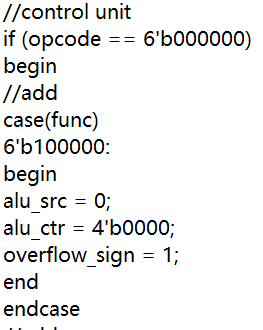
The overflow\_sign is used to judge whether the instruction needs to judge overflow or not.

The reg\_hi and reg\_lo are used to store the value of hi and lo for special instructions: mult, multu, div, divu.

I finish the control part by judging and integration.



Because the operations of some instructions in the ALU are similar, so I put them together to avoid using op code and func of every instruction to judge and call the function in ALU. In the control part, I integrate the func and opcode of every instruction and give every instruction an alu ctr and an alu src. Here is an example:



Then the work of the control unit is over. It integrates the func and opcode of every instruction and gives instructions of “same type” same alu\_ctr and alu\_src.

If the value needs extension, I will extend it and then put it into the register.

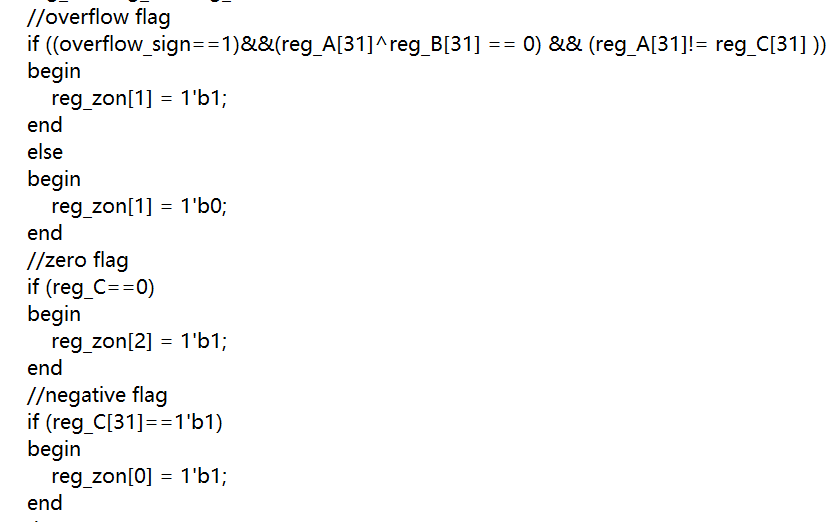
Here are two examples of zero extension and sign extension:





The flag part will be finished by using reg\_zon. The code will judge and give the information to the reg\_zon if the result meet the requirement. Finally the reg\_zon will be printed to the user.

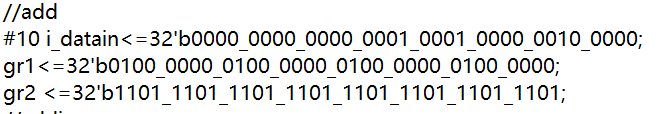
Here is an example to show the implement of zero flag, overflow flag and negative flag:



**3 explanation of all required instructions:**

**add:**

The input:



The output:

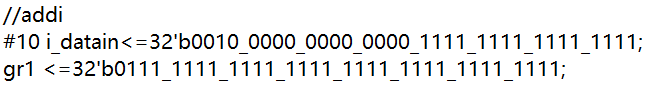




1077952576 - 572662307 = 505290269. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**addi:**

The input:



The output:



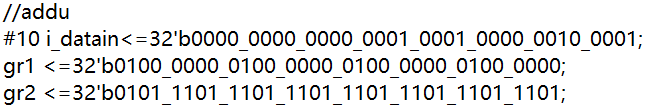


Because when there is only gr1 or gr2 input, the printed part will use the gr1 and gr2 of the last one, you may see wrong gr1 or gr2 in the output part. However, it does not affect the result. We will just focus on the reg\_A and reg\_B.

2147483647 - 1 = 2147483646. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**addu:**

The input:



The output:

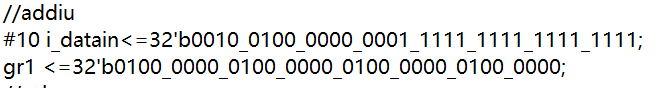




1077952576 + 1574821341= 2652773917. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**addiu:**

The input:



The output:

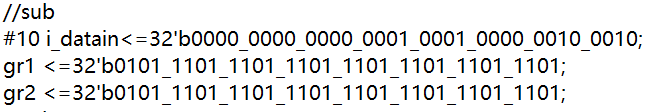




1077952576 + (-1) = 1077952575. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or be negative and addiu does not consider overflow. Therefore, the zero flag, overflow flag and negative flag are all zero.

**sub:**

The input:



The output:

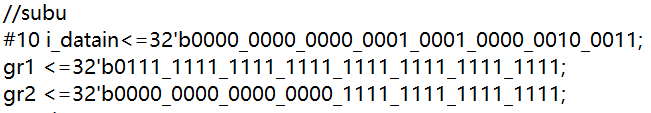




1574821341 - 1574821341 = 0. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not negative or overflow. Reg\_C = 0, so the zero flag is 1.

**subu:**

The input:



The output:

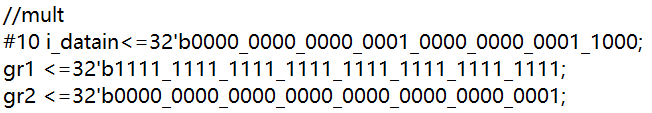




2147483647- 65535= 2147418112. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**mult:**

The input:



The output:

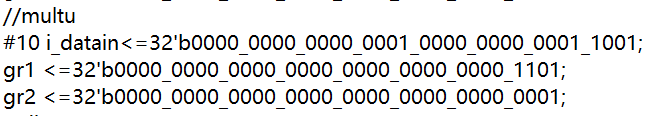




-1\*1=1. There is no output C in mult. There are hi and lo output in mult. Because the output of the multiple is 64bit. I first use a 64bit register hilo to store the result of the multiple of a and b. Then I use two 32 bit registers hi and lo to get the first 32 bit of hilo and last 32 bit of hilo. Finally I get the result of multiple: hi and lo. hi and lo show the right results, so the instruction works well.

**multu:**

The input:



The output:

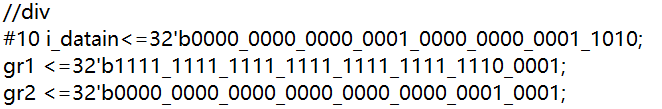




13\*1=1. There is no output C in multu. There are hi and lo output in multu. Because the ouput of the multiple is 64bit. I first use a 64bit register hilo to store the result of the multiple of a and b. Then I use two 32 bit registers hi and lo to get the first 32 bit of hilo and last 32 bit of hilo. Finally I get the result of multiple: hi and lo. hi and lo show the right results, so the instruction works well.

**div:**

The input:



The output:

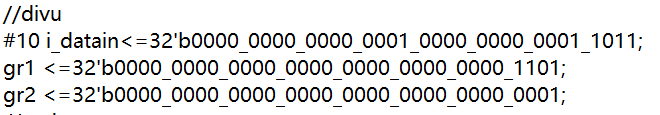




-31/17=-1, -31%17=-14. There is no output C in div. There are hi and lo output in div. I use the hi to store the quotient and lo to store the remainder. The hi and lo show right result, so the instruction works well.

**divu:**

The input:



The output:

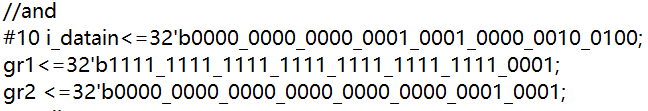




13/1=13, 13%1=0. There is no output C in divu. There are hi and lo output in divu. I use the hi to store the quotient and lo to store the remainder. The hi and lo show right result, so the instruction works well.

**and:**

The input:



The output:

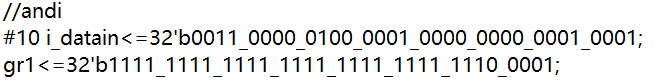


****

The and of a and b is to use and gate for every bit of a and b. The reg\_C shows right result. Therefore, the instruction works well.

**andi:**

The input:



The output:

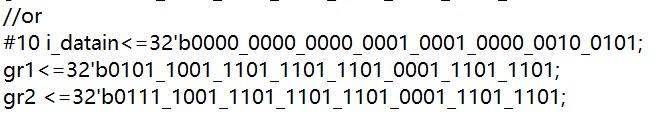


****

The andi of a and b is to use and gate for every bit of a and b. The b uses zero-extension for imm. The reg\_C shows right result. Therefore, the instruction works well.

**or:**

The input:



The output:

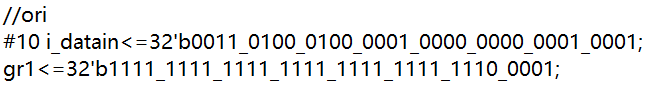


****

The or of a and b is to use or gate for every bit of a and b. The reg\_C shows right result. Therefore, the instruction works well.

**ori:**

The input:



The output:

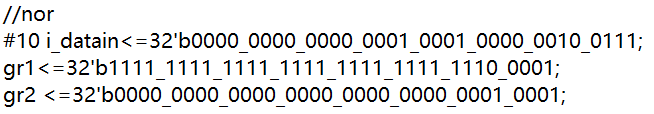


****

The ori of a and b is to use or gate for every bit of a and b. The b uses zero-extension for imm. The reg\_C shows right result. Therefore, the instruction works well.

**nor:**

The input:



The output:

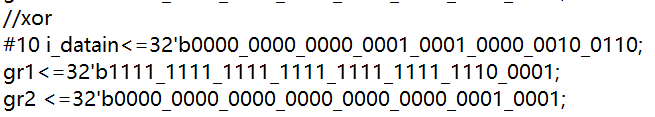


****

The nor of a and b is to use nor gate for every bit of a and b. The reg\_C shows right result. Therefore, the instruction works well.

**xor:**

The input:

****

The output:

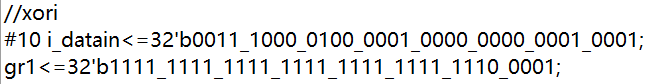


****

The xor of a and b is to use xor gate for every bit of a and b. The reg\_C shows right result. Therefore, the instruction works well.

**xori:**

The input:



The output:

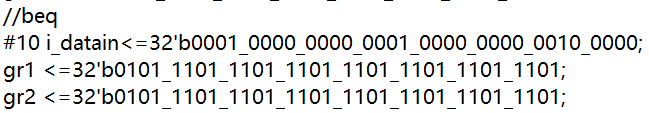


****

The xori of a and b is to use xor gate for every bit of a and b. The b uses zero-extension for imm. The reg\_C shows right result. Therefore, the instruction works well.

**beq:**

The input:



The output:



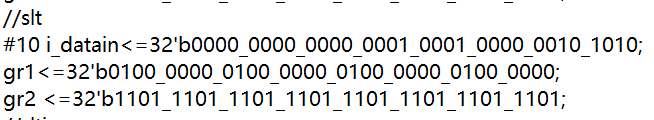


The beq is similar to sub in ALU. Therefore, I put it into the alu control of sub.

2044580317 - 2044580317 = 0. The register C shows the right result, so the instruction works well. Because the result is 0, the zero flag is 1.

**slt:**

The input:



The output:

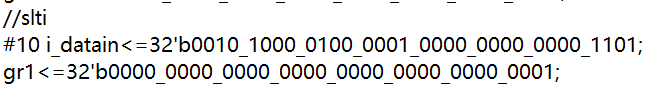




gr1>0,gr2<0, gr1-gr2>0, so the output is 0. Reg\_C shows the right output. Therefore, the instruction works well. Because the result is 0, the zero flag is 1.

**slti:**

The input:



The output:

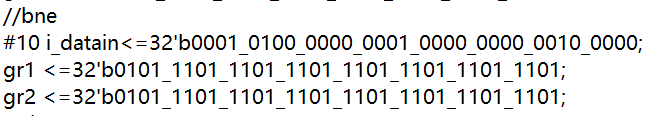




1 – 13 < 0, so the output is 1. Reg\_C shows the right output. Therefore, the instruction works well.

**bne:**

The input:



The output:



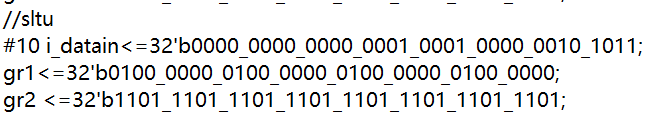


The bne is similar to sub in ALU. Therefore, I put it into the alu control of sub.

2044580317 - 2044580317 = 0. The register C shows the right result, so the instruction works well. Because the result is 0, the zero flag is 1.

**sltu:**

The input:



The output:

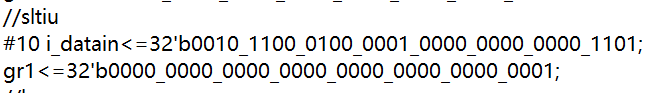




1077952576 – 3722304989 < 0, so the result is 1. Reg\_C shows the right output. Therefore, the instruction works well.

**sltiu:**

The input:



The output:

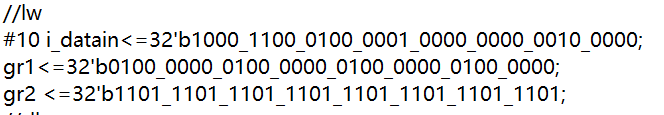




1 – 13 < 0, so the output is 1. Reg\_C shows the right output. Therefore, the instruction works well.

**lw:**

The input:

****

The output:

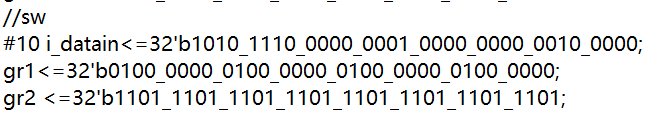


****

1077952576 + 32 = 1077952608. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**sw:**

The input:



The output:

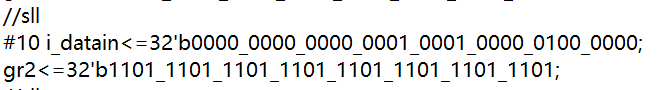


****

1077952576 + 32 = 1077952608. Reg\_C shows the right output. Therefore, the instruction works well. Reg\_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

**sll:**

The input:



The output:

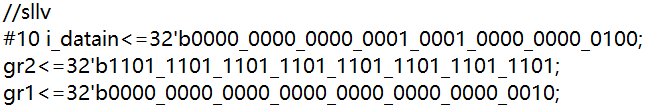




Shift left 1 bit for the gr2. The result is 10111011101110111011101110111010. Reg\_C shows the right result. Therefore, the instruction works well.

**sllv:**

The input:



The output:

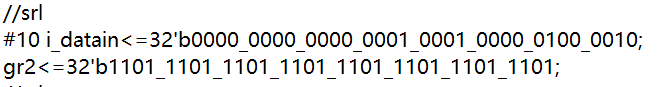




Shift left 2 bit for the gr2. The result is 1110111011101110111011101110100. Reg\_C shows the right result. Therefore, the instruction works well.

**srl:**

The input:



The output:

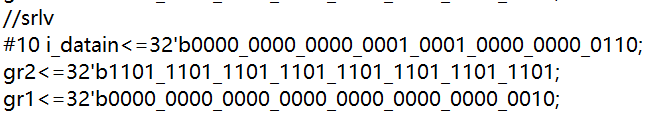




Shift right 1 bit for the gr2. The result is 1101110111011101110111011101110. Reg\_C shows the right result. Therefore, the instruction works well.

**srlv:**

The input:



The output:

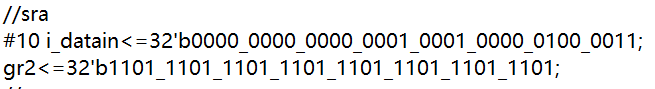




Shift right 2 bit for the gr2. The result is 110111011101110111011101110111. Reg\_C shows the right result. Therefore, the instruction works well.

**sra:**

The input:



The output:

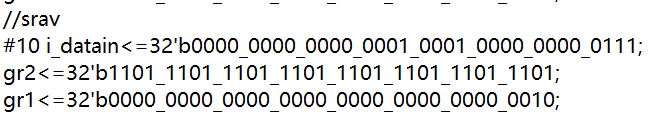




Arithmetic shift right 1 bit for the gr2. The result is 1110111011101110111011101110. Reg\_C shows the right result. Therefore, the instruction works well.

**srav:**

The input:



The output:





Arithmetic shift right 2 bit for the gr2. The result is 11110111011101110111011101110111. Reg\_C shows the right result. Therefore, the instruction works well.

**4.An extra explanation about flags:**

The reg\_zon is used to store the flags. In this register, reg\_zon[0] stores negative flag. reg\_zon[1] stores overflow flag. reg\_zon[2] stores zero flag. For zero flag, if the output is zero, then the zero flag will be 1. If the output is not zero, the zero flag will be 0. For the overflow flag, for signed value, if the output is overflow, the overflow flag will be 1. If the output does not overflow, the overflow flag will be 0. For the negative flag, for signed value, if the 32bit of the output is 1, the negative flag will be 1. If the 32bit of the output is not 1, the negative flag will be 0.



From my testbench, you can see that if the output C is 0, the zero flag will be 1. If the output C is not 0, the zero flag will be 0. For signed value, if the output C is negative, which means the 32bit of C is 1, the negative flag will be 1. If the output C is positive which means the 32bit of C is 0, the negative flag will be 0. If the output C is overflow, the overflow flag will be 1. If the output C does not overflow, the overflow flag will be 0. Generally, if these three flags do not be used in some instructions, it will be set to 0.

**5 How to run my code (in Windows):**

****

**6 Summary:**

I really try my best to finish this project. I have learnt a lot about Verilog and ALU from this project.