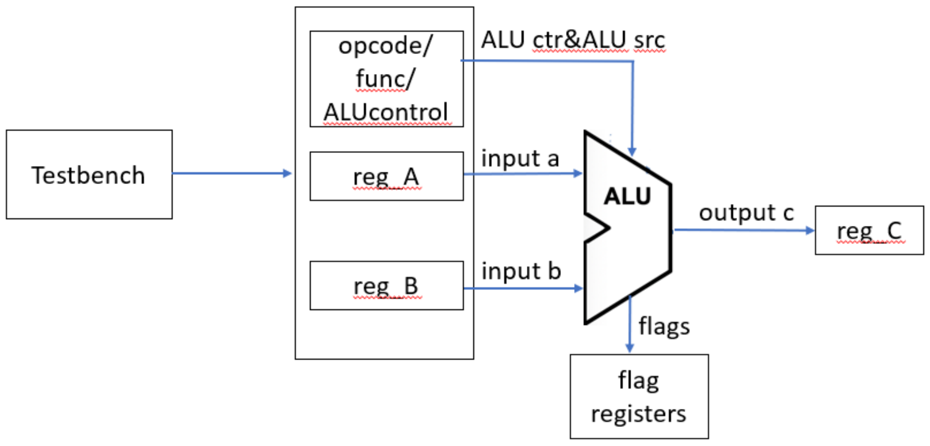
Project Report

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:

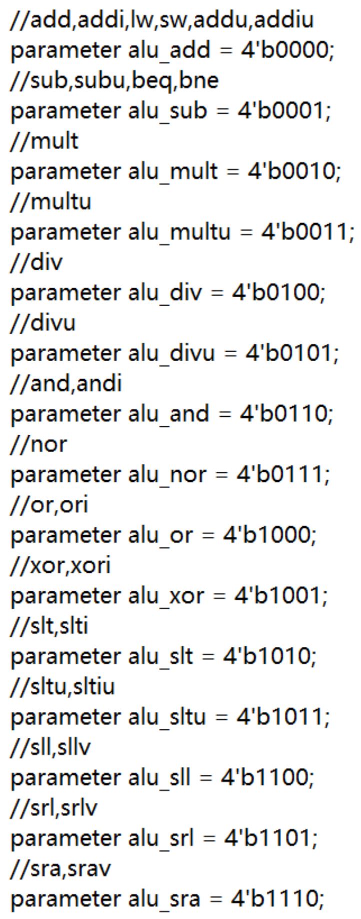


The registers I have used are as follows:

```
reg signed[31:0] reg_C;
reg [2:0] reg_zon = 3'b000;
reg signed[31:0]reg_A,reg_B;
reg unsigned[31:0]reg_Au, reg_Bu;
reg[63:0]reg_hilo;
reg[3:0] alu_ctr;
reg alu_src;
reg overflow_sign;
reg signed[31:0] reg_hi,reg_lo;
```

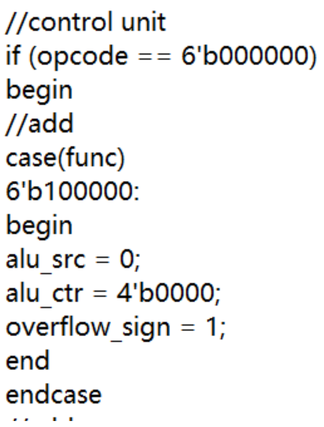
The reg_C is used to store the value of ouput C.
The reg_zon is used to store the flags. In this register, reg_zon[0] stores negative flag. reg_zon[1] stores overflow flag. reg_zon[2] stores zero flag.
The reg_A and reg_B are used to store the value of input a and b.
The unsigned reg_Au and reg_Bu are 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.



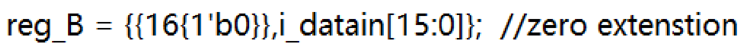
```
//add,addi,lw,sw,addu,addiu
parameter alu_add = 4'b0000;
//sub,subu,beq,bne
parameter alu_sub = 4'b0001;
//mult
parameter alu_mult = 4'b0010;
//multu
parameter alu_multu = 4'b0011;
//div
parameter alu_div = 4'b0100;
//divu
parameter alu_divu = 4'b0101;
//and,andi
parameter alu_and = 4'b0110;
//nor
parameter alu_nor = 4'b0111;
//or,ori
parameter alu_or = 4'b1000;
//xor,xori
parameter alu_xor = 4'b1001;
//slt,slti
parameter alu_slt = 4'b1010;
//sltu,sltiu
parameter alu_sltu = 4'b1011;
//sll,sllv
parameter alu_sll = 4'b1100;
//srl,srlv
parameter alu_srl = 4'b1101;
//sra,srav
parameter alu_sra = 4'b1110;
```

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:

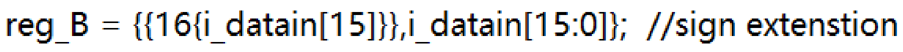


```
//control unit
if (opcode == 6'b000000)
begin
//add
case(func)
6'b100000:
begin
alu_src = 0;
alu_ctr = 4'b0000;
overflow_sign = 1;
end
endcase
```

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:

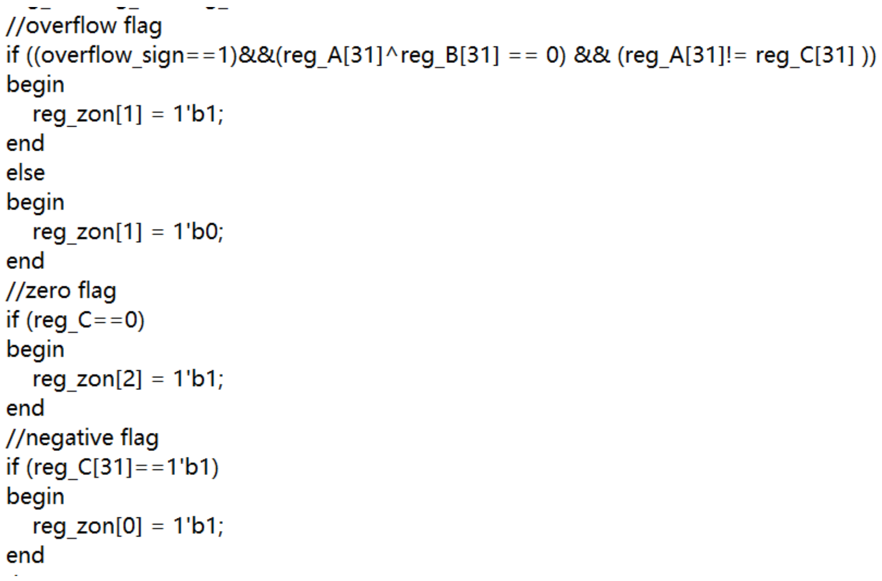


```
reg_B = {{16{1'b0}},i_datain[15:0]};  //zero extenstion
```



```
reg_B = {{16{i_datain[15]}},i_datain[15:0]};  //sign extenstion
```

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:

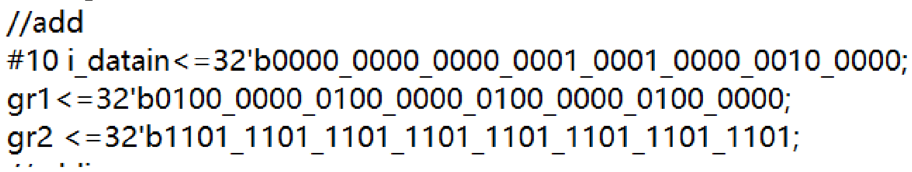


```
//overflow flag
if ((overflow_sign==1)&&(reg_A[31]^reg_B[31] == 0) && (reg_A[31]!= reg_C[31] ))
begin
   reg_zon[1] = 1'b1;
end
else
begin
   reg_zon[1] = 1'b0;
end
//zero flag
if (reg_C==0)
begin
   reg_zon[2] = 1'b1;
end
//negative flag
if (reg_C[31]==1'b1)
begin
   reg_zon[0] = 1'b1;
end
```

3 explanation of all required instructions:

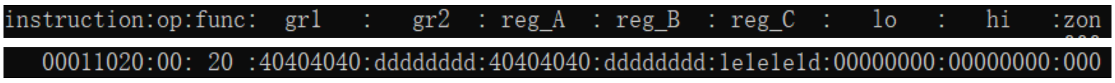
add:

The input:



```
//add
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0000;
gr1<=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

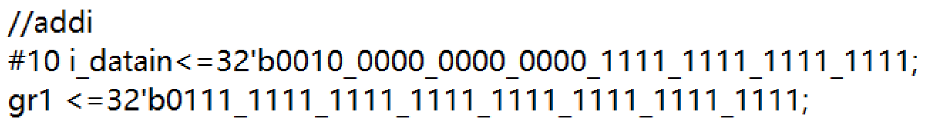


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   00011020:00: 20 :40404040:dddddddd:40404040:dddddddd:1e1e1e1d:00000000:00000000:000
```

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.

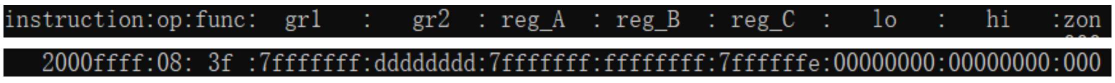
addi:

The input:



```
//addi
#10 i_datain<=32'b0010_0000_0000_0000_1111_1111_1111_1111;
gr1 <=32'b0111_1111_1111_1111_1111_1111_1111_1111;
```

The output:



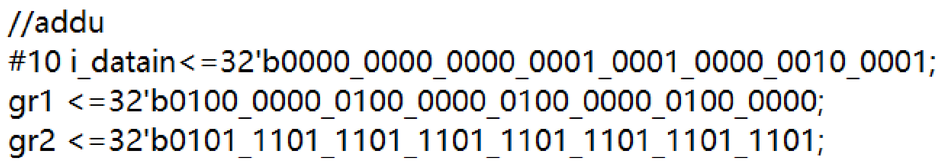
```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   2000ffff:08: 3f :7fffffff:dddddddd:7fffffff:ffffffff:7ffffffe:00000000:00000000:000
```

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.

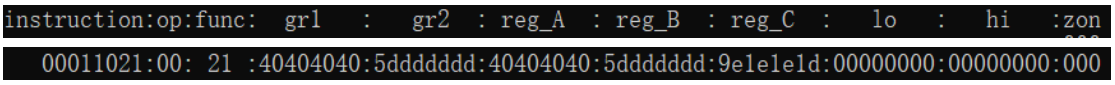
addu:

The input:



```
//addu
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0001;
gr1 <=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

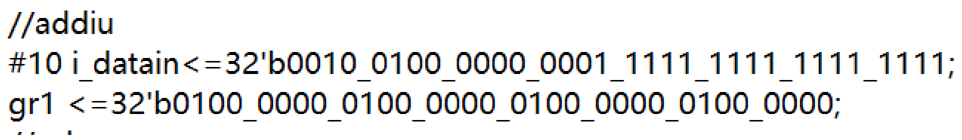


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   00011021:00: 21 :40404040:5ddddddd:40404040:5ddddddd:9e1e1e1d:00000000:00000000:000
```

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.

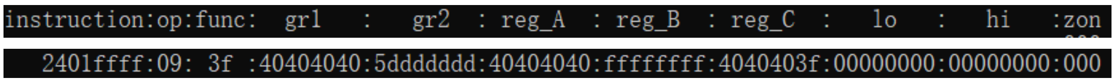
addiu:

The input:



```
//addiu
#10 i_datain<=32'b0010_0100_0000_0001_1111_1111_1111_1111;
gr1 <=32'b0100_0000_0100_0000_0100_0000_0100_0000;
```

The output:



```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   2401ffff:09: 3f :40404040:5ddddddd:40404040:ffffffff:4040403f:00000000:00000000:000
```

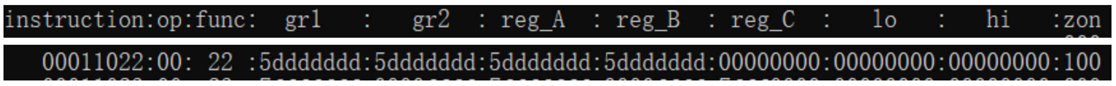
1077952576 + (-1) = 1077952575. Reg_C shows the right output. Therefore, the instruction works well. Reg_C is not equal to zero or be negative and addiu does not consider overflow. Therefore, the zero flag, overflow flag and negative flag are all zero.

sub:

The input:

```
//sub
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0010;
gr1 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
gr2 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
```

The output:



```
instruction:op:func:  gr1    :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
  00011022:00: 22 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
```

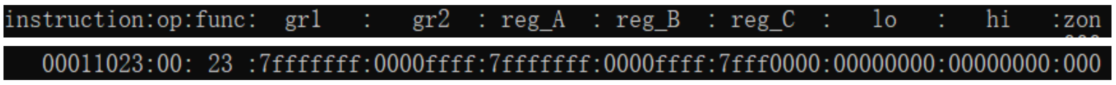
1574821341 - 1574821341 = 0. Reg_C shows the right output. Therefore, the instruction works well. Reg_C is not negative or overflow. Reg_C = 0, so the zero flag is 1.

subu:

The input:

```
//subu
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0011;
gr1 <=32'b0111_1111_1111_1111_1111_1111_1111_1111;
gr2 <=32'b0000_0000_0000_0000_1111_1111_1111_1111;
```

The output:



```
instruction:op:func:  gr1    :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
  00011023:00: 23 :7fffffff:0000ffff:7fffffff:0000ffff:7fff0000:00000000:00000000:000
```

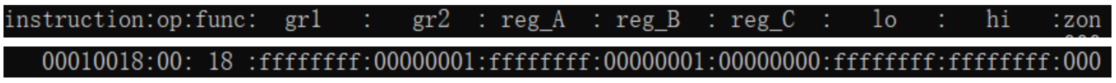
2147483647- 65535= 2147418112. Reg_C shows the right output. Therefore, the instruction works well. Reg_C is not equal to zero or overflow or be negative. Therefore, the zero flag, overflow flag and negative flag are all zero.

mult:

The input:

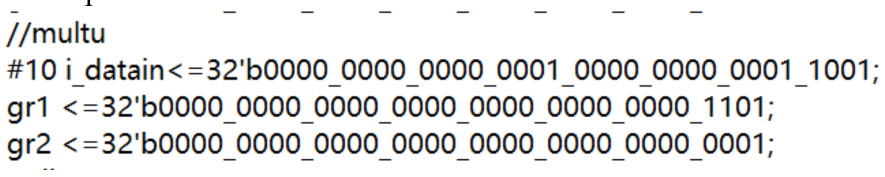
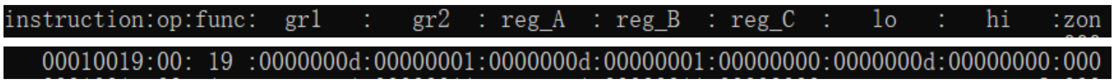
```
//mult
#10 i_datain<=32'b0000_0000_0000_0001_0000_0000_0001_1000;
gr1 <=32'b1111_1111_1111_1111_1111_1111_1111_1111;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

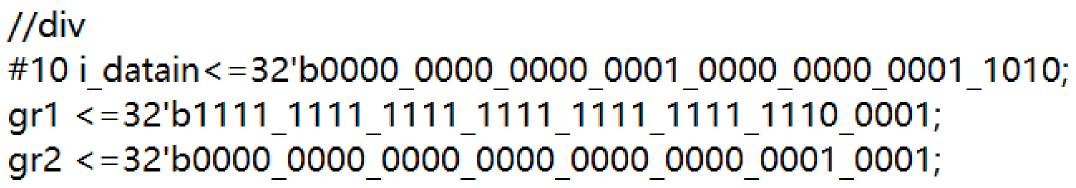
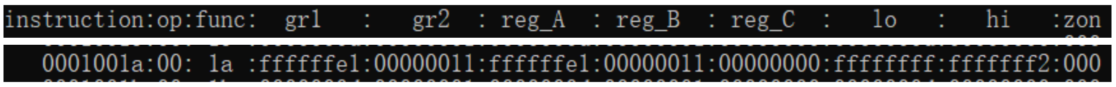
The output:

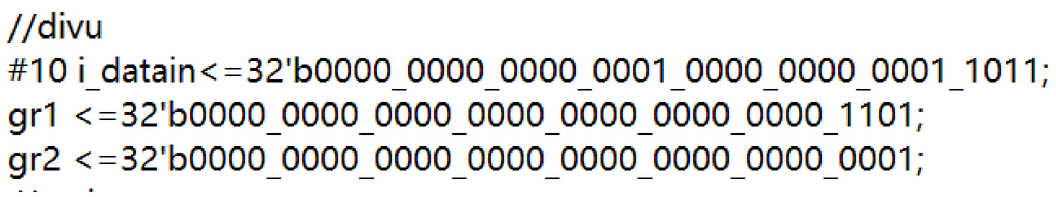
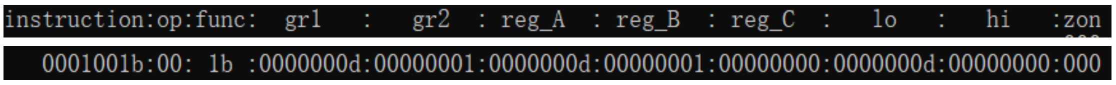


```
instruction:op:func:  gr1    :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
  00010018:00: 18 :ffffffff:00000001:ffffffff:00000001:00000000:ffffffff:ffffffff:000
```

-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:

```
//multu
#10 i_datain<=32'b0000_0000_0000_0001_0000_0000_0001_1001;
gr1 <=32'b0000_0000_0000_0000_0000_0000_0000_1101;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```
The output:

```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  00010019:00: 19 :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
```
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:

```
//div
#10 i_datain<=32'b0000_0000_0000_0001_0000_0000_0001_1010;
gr1 <=32'b1111_1111_1111_1111_1111_1111_1110_0001;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0001_0001;
```
The output:

```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  0001001a:00: 1a :ffffffe1:00000011:ffffffe1:00000011:00000000:ffffffff:fffffff2:000
```
-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.

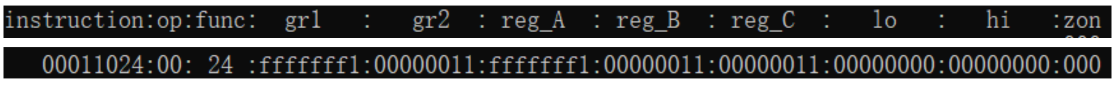
divu:
The input:

```
//divu
#10 i_datain<=32'b0000_0000_0000_0001_0000_0000_0001_1011;
gr1 <=32'b0000_0000_0000_0000_0000_0000_0000_1101;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```
The output:

```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  0001001b:00: 1b :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
```
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:

```
//and
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0100;
gr1<=32'b1111_1111_1111_1111_1111_1111_1111_0001;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0001_0001;
```

The output:



```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :    lo    :    hi    :zon
   00011024:00: 24 :fffffff1:00000011:fffffff1:00000011:00000011:00000000:00000000:000
```

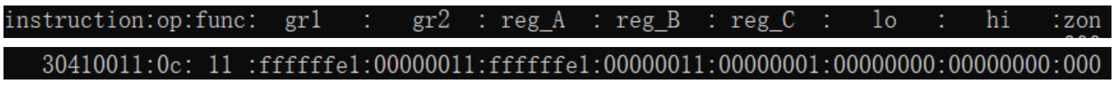
The and of a and b is to use and gate for every bit of a and b. The reg_C shows right result. Therefore, the instruction works well.

andi:

The input:

```
//andi
#10 i_datain<=32'b0011_0000_0100_0001_0000_0000_0001_0001;
gr1<=32'b1111_1111_1111_1111_1111_1111_1110_0001;
```

The output:



```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :    lo    :    hi    :zon
   30410011:0c: 11 :ffffffe1:00000011:ffffffe1:00000011:00000001:00000000:00000000:000
```

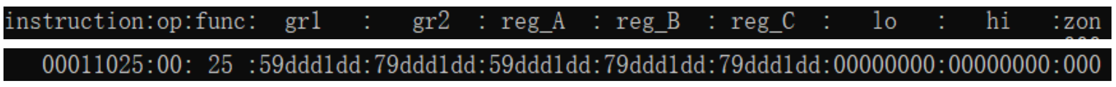
The andi of a and b is to use and gate for every bit of a and b. The b uses zero-extension for imm. The reg_C shows right result. Therefore, the instruction works well.

or:

The input:

```
//or
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0101;
gr1<=32'b0101_1001_1101_1101_1101_0001_1101_1101;
gr2 <=32'b0111_1001_1101_1101_1101_0001_1101_1101;
```

The output:



```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :    lo    :    hi    :zon
   00011025:00: 25 :59ddd1dd:79ddd1dd:59ddd1dd:79ddd1dd:79ddd1dd:00000000:00000000:000
```

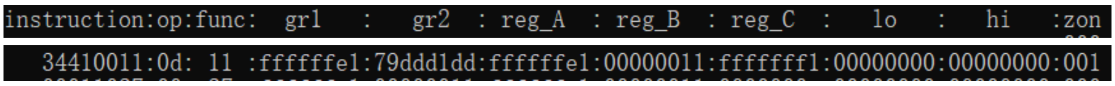
The or of a and b is to use or gate for every bit of a and b. The reg_C shows right result. Therefore, the instruction works well.

ori:

The input:

```
//ori
#10 i_datain<=32'b0011_0100_0100_0001_0000_0000_0001_0001;
gr1<=32'b1111_1111_1111_1111_1111_1111_1110_0001;
```

The output:

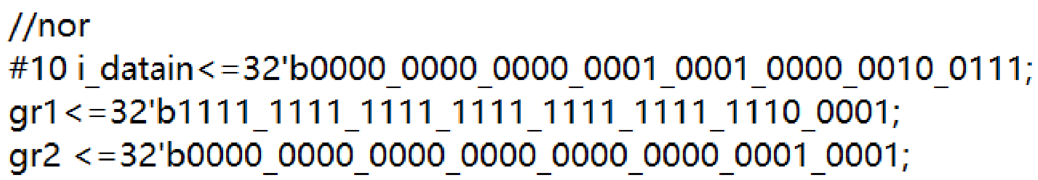


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   34410011:0d: 11 :ffffffe1:79ddd1dd:ffffffe1:00000011:fffffff1:00000000:00000000:001
```

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.

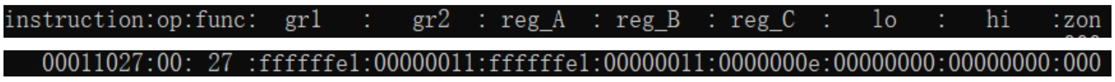
nor:

The input:



```
//nor
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0111;
gr1<=32'b1111_1111_1111_1111_1111_1111_1110_0001;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0001_0001;
```

The output:

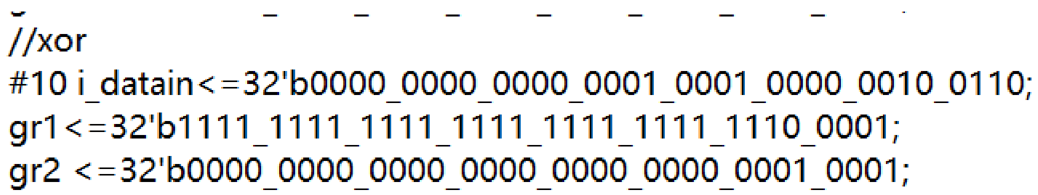


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   00011027:00: 27 :ffffffe1:00000011:ffffffe1:00000011:0000000e:00000000:00000000:000
```

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.

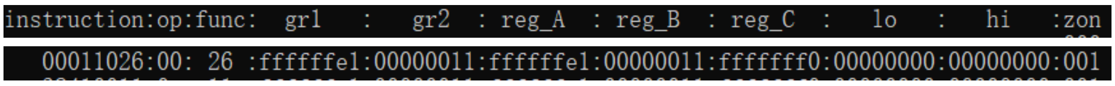
xor:

The input:



```
//xor
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_0110;
gr1<=32'b1111_1111_1111_1111_1111_1111_1110_0001;
gr2 <=32'b0000_0000_0000_0000_0000_0000_0001_0001;
```

The output:

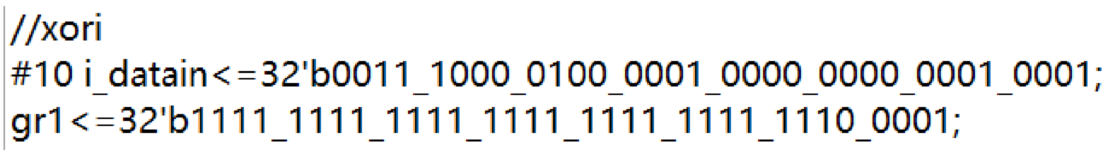


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   00011026:00: 26 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
```

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.

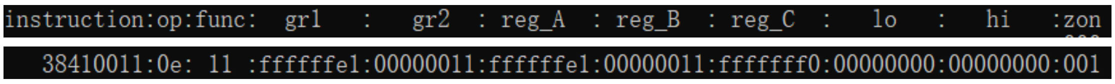
xori:

The input:



```
//xori
#10 i_datain<=32'b0011_1000_0100_0001_0000_0000_0001_0001;
gr1<=32'b1111_1111_1111_1111_1111_1111_1110_0001;
```

The output:

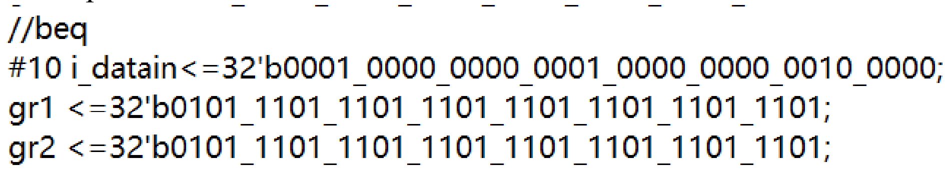


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
   38410011:0e: 11 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
```

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.

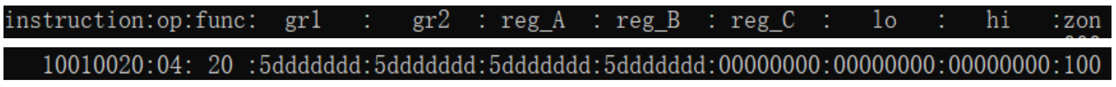
beq:

The input:



```
//beq
#10 i_datain<=32'b0001_0000_0000_0001_0000_0000_0010_0000;
gr1 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
gr2 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

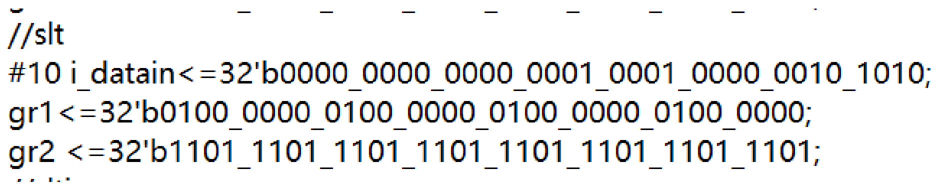


```
instruction:op:func:  gr1    :   gr2  : reg_A   : reg_B   : reg_C   :   lo    :   hi   :zon
   10010020:04: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
```

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.

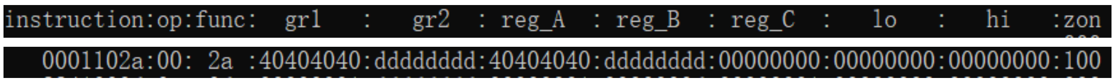
slt:

The input:



```
//slt
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_1010;
gr1<=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

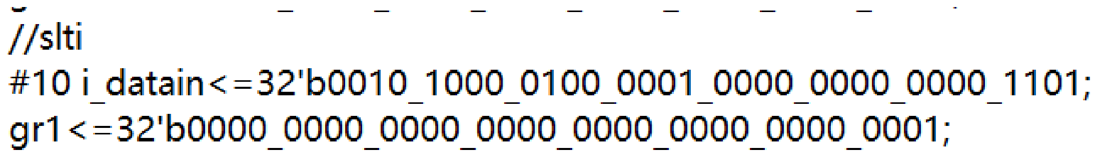


```
instruction:op:func:  gr1    :   gr2  : reg_A   : reg_B   : reg_C   :   lo    :   hi   :zon
   0001102a:00: 2a :40404040:dddddddd:40404040:dddddddd:00000000:00000000:00000000:100
```

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.

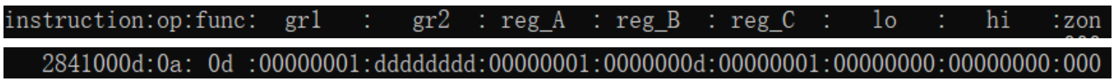
slti:

The input:



```
//slti
#10 i_datain<=32'b0010_1000_0100_0001_0000_0000_0000_1101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

The output:

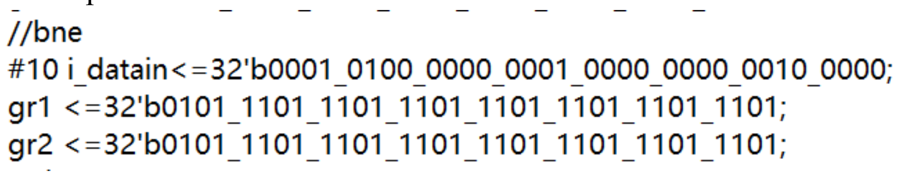


```
instruction:op:func:  gr1    :   gr2  : reg_A   : reg_B   : reg_C   :   lo    :   hi   :zon
   2841000d:0a: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
```

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

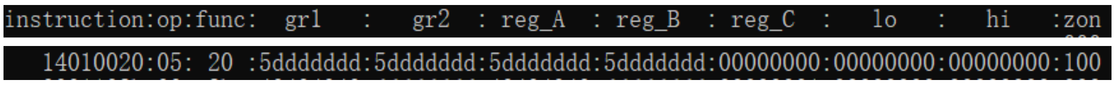
bne:

The input:



```
//bne
#10 i_datain<=32'b0001_0100_0000_0001_0000_0000_0010_0000;
gr1 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
gr2 <=32'b0101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

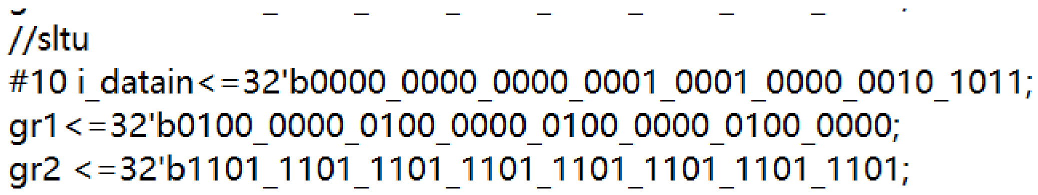


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  14010020:05: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
```

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.

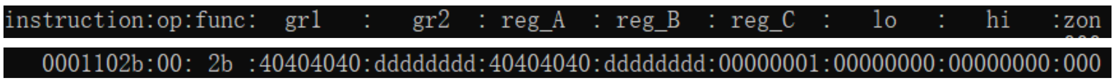
sltu:

The input:



```
//sltu
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0010_1011;
gr1<=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

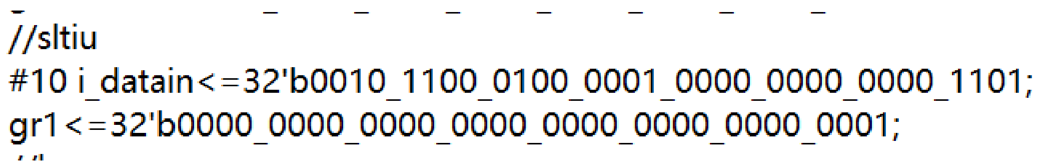


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  0001102b:00: 2b :40404040:dddddddd:40404040:dddddddd:00000001:00000000:00000000:000
```

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

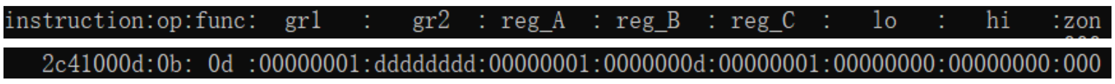
sltiu:

The input:



```
//sltiu
#10 i_datain<=32'b0010_1100_0100_0001_0000_0000_0000_1101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

The output:

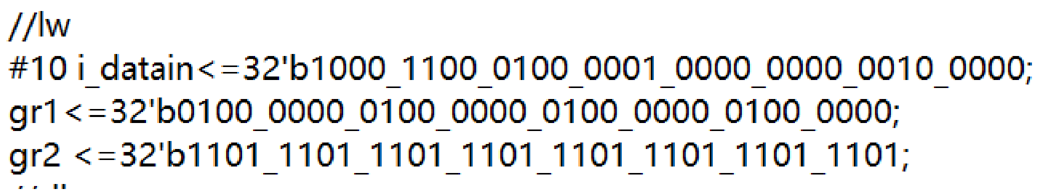


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  2c41000d:0b: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
```

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

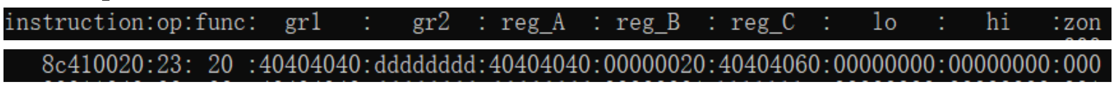
lw:

The input:



```
//lw
#10 i_datain<=32'b1000_1100_0100_0001_0000_0000_0010_0000;
gr1<=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

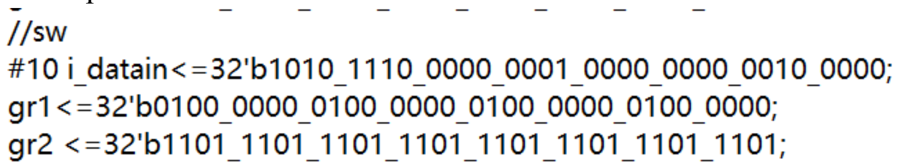


```
instruction:op:func:  gr1    :   gr2   : reg_A   : reg_B   : reg_C   :   lo    :   hi    :zon
  8c410020:23: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
```

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.

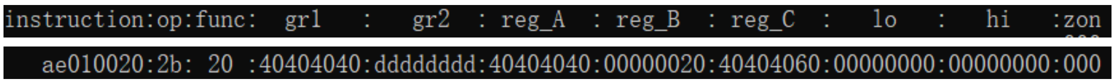
sw:

The input:



```
//sw
#10 i_datain<=32'b1010_1110_0000_0001_0000_0000_0010_0000;
gr1<=32'b0100_0000_0100_0000_0100_0000_0100_0000;
gr2 <=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

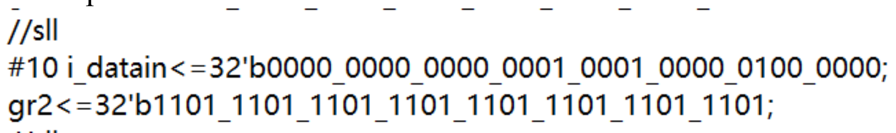


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   ae010020:2b: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
```

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.

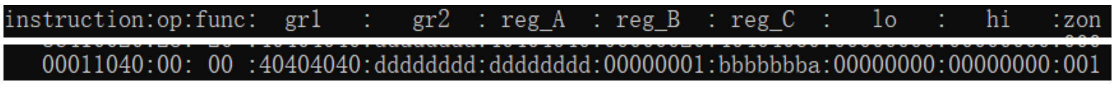
sll:

The input:



```
//sll
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0100_0000;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

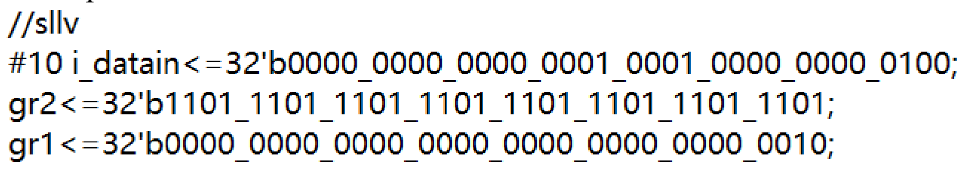


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011040:00: 00 :40404040:dddddddd:dddddddd:00000001:bbbbbbba:00000000:00000000:001
```

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

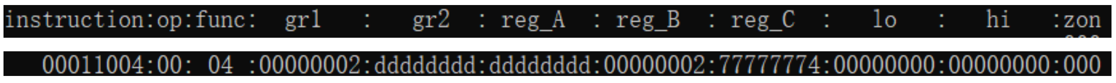
sllv:

The input:



```
//sllv
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0000_0100;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
```

The output:

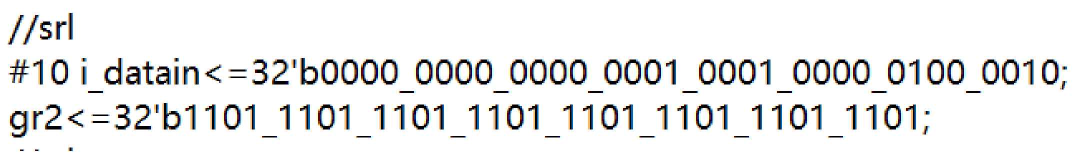


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011004:00: 04 :00000002:dddddddd:dddddddd:00000002:77777774:00000000:00000000:000
```

Shift left 2 bit for the gr2. The result is 11101110111011101110111011110100. Reg_C shows the right result. Therefore, the instruction works well.

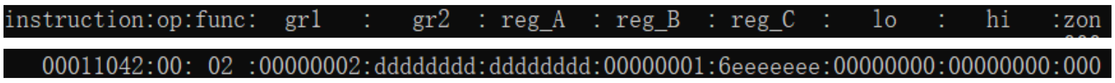
srl:

The input:



```
//srl
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0100_0010;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

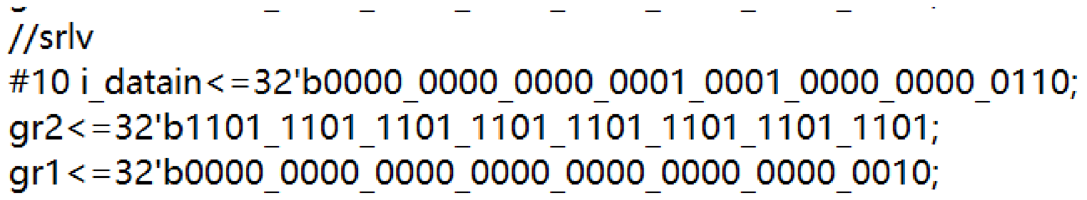


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011042:00: 02 :00000002:dddddddd:dddddddd:00000001:6eeeeeee:00000000:00000000:000
```

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

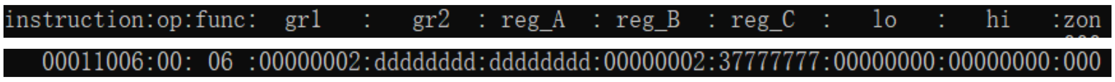
srlv:

The input:



```
//srlv
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0000_0110;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
```

The output:

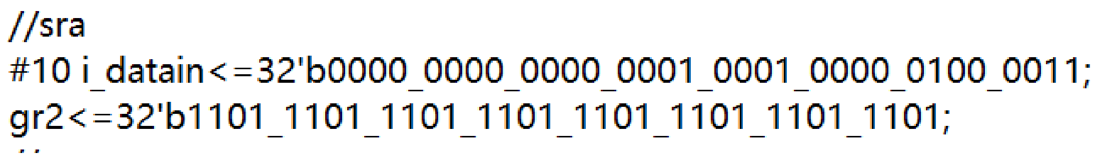


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011006:00: 06 :00000002:dddddddd:dddddddd:00000002:37777777:00000000:00000000:000
```

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

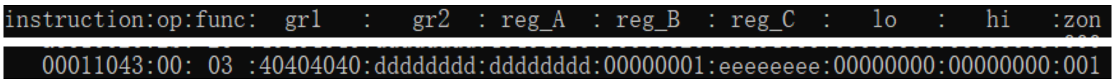
sra:

The input:



```
//sra
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0100_0011;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
```

The output:

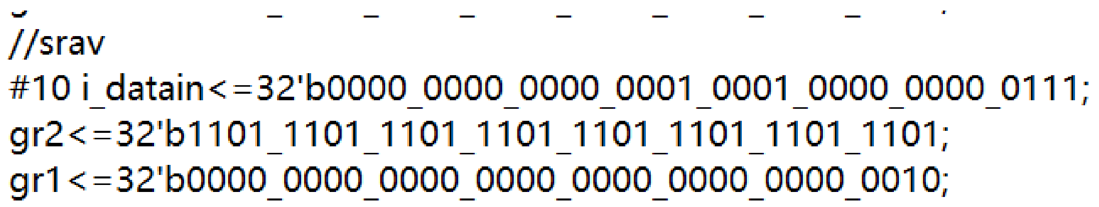


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011043:00: 03 :40404040:dddddddd:dddddddd:00000001:eeeeeeee:00000000:00000000:001
```

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

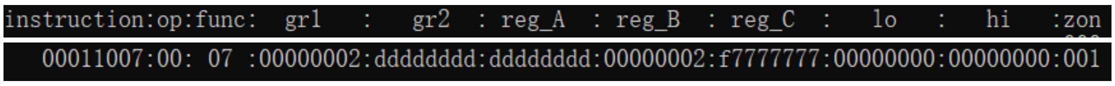
srav:

The input:



```
//srav
#10 i_datain<=32'b0000_0000_0000_0001_0001_0000_0000_0111;
gr2<=32'b1101_1101_1101_1101_1101_1101_1101_1101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
```

The output:

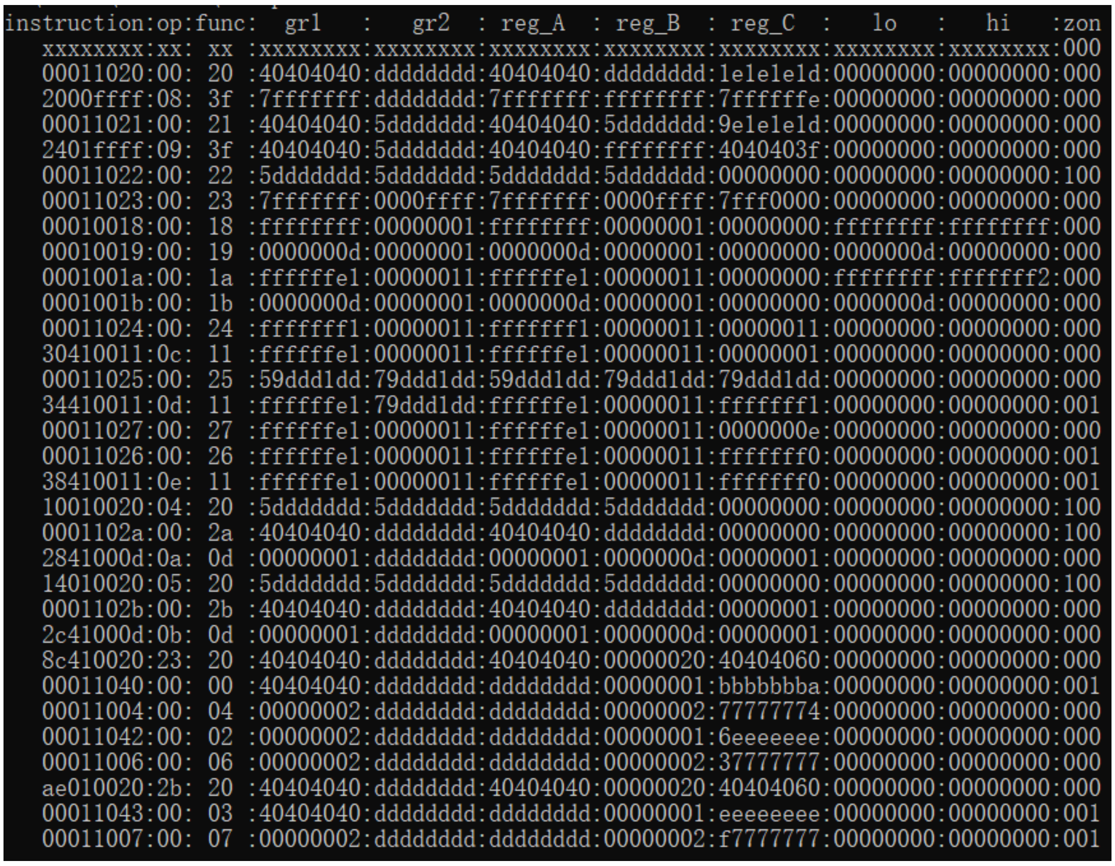


```
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   00011007:00: 07 :00000002:dddddddd:dddddddd:00000002:f7777777:00000000:00000000:001
```

Arithmetic shift right 2 bit for the gr2. The result is 1111011101110111011101110111. Reg_C shows the right result. Therefore, the instruction works well.

4.An extra explanation about flags:

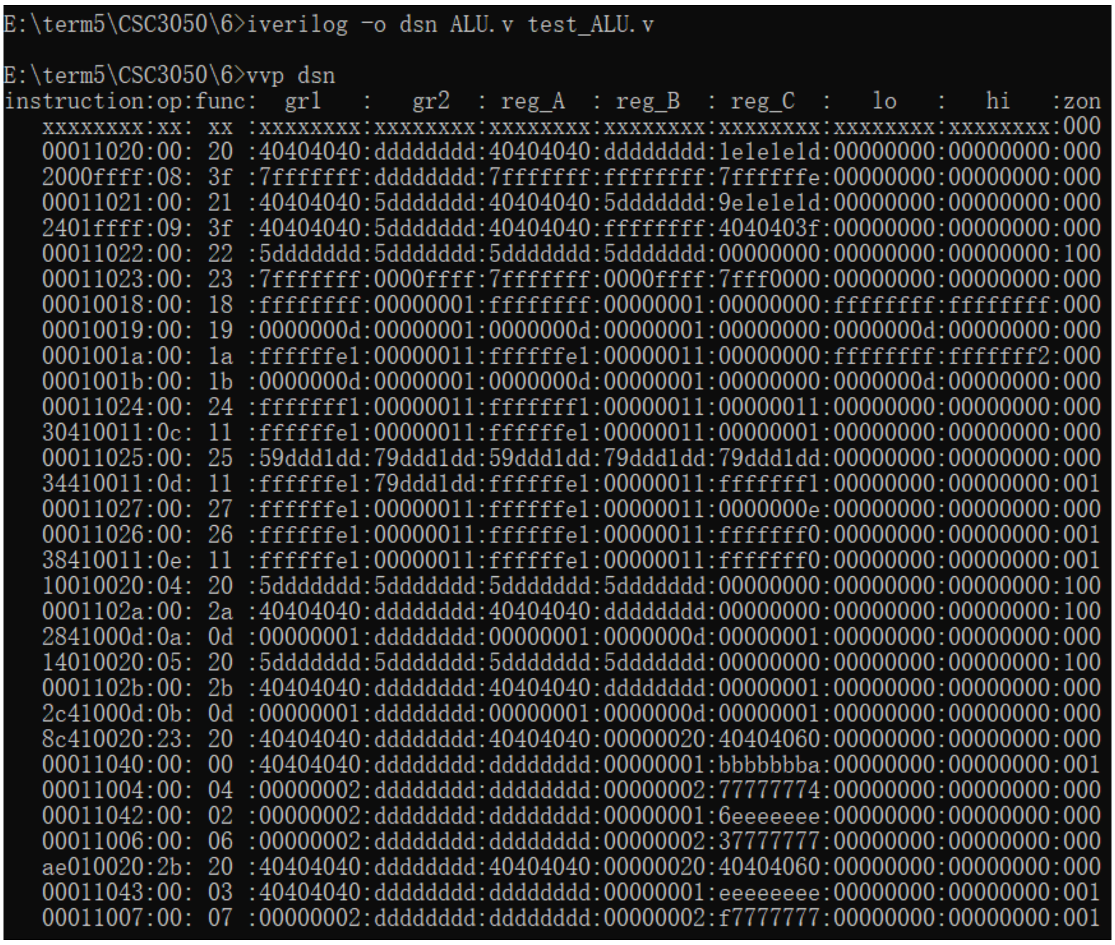
The reg_zon is used to store the flags. In this register, reg_zon[0] stores negative flag. reg_zon[1] stores overflow flag. reg_zon[2] stores zero flag. For zero flag, if the output is zero, then the zero flag will be 1. If the output is not zero, the zero flag will be 0. For the overflow flag, for signed 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.



```
instruction:op:func:  gr1   :   gr2   : reg_A   : reg_B   : reg_C   :    lo   :    hi   :zon
   xxxxxxxx:xx: xx :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:000
   00011020:00: 20 :40404040:dddddddd:40404040:dddddddd:1e1e1e1d:00000000:00000000:000
   2000ffff:08: 3f :7fffffff:dddddddd:7fffffff:ffffffff:7ffffffe:00000000:00000000:000
   00011021:00: 21 :40404040:5ddddddd:40404040:5ddddddd:9e1e1e1d:00000000:00000000:000
   2401ffff:09: 3f :40404040:5ddddddd:40404040:ffffffff:4040403f:00000000:00000000:000
   00011022:00: 22 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   00011023:00: 23 :7fffffff:0000ffff:7fffffff:0000ffff:7fff0000:00000000:00000000:000
   00010018:00: 18 :ffffffff:00000001:ffffffff:00000001:00000000:ffffffff:ffffffff:000
   00010019:00: 19 :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
   0001001a:00: 1a :ffffffe1:00000011:ffffffe1:00000011:00000000:ffffffff:fffffff2:000
   0001001b:00: 1b :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
   00011024:00: 24 :fffffff1:00000011:fffffff1:00000011:00000011:00000000:00000000:000
   30410011:0c: 11 :ffffffe1:00000011:ffffffe1:00000011:00000001:00000000:00000000:000
   00011025:00: 25 :59ddd1dd:79ddd1dd:59ddd1dd:79ddd1dd:79ddd1dd:00000000:00000000:000
   34410011:0d: 11 :ffffffe1:79ddd1dd:ffffffe1:00000011:fffffff1:00000000:00000000:001
   00011027:00: 27 :ffffffe1:00000011:ffffffe1:00000011:0000000e:00000000:00000000:000
   00011026:00: 26 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
   38410011:0e: 11 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
   10010020:04: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   0001102a:00: 2a :40404040:dddddddd:40404040:dddddddd:00000000:00000000:00000000:100
   2841000d:0a: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
   14010020:05: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   0001102b:00: 2b :40404040:dddddddd:40404040:dddddddd:00000001:00000000:00000000:000
   2c41000d:0b: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
   8c410020:23: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
   00011040:00: 00 :40404040:dddddddd:dddddddd:00000001:bbbbbbba:00000000:00000000:001
   00011004:00: 04 :00000002:dddddddd:dddddddd:00000002:77777774:00000000:00000000:000
   00011042:00: 02 :00000002:dddddddd:dddddddd:00000001:6eeeeeee:00000000:00000000:000
   00011006:00: 06 :00000002:dddddddd:dddddddd:00000002:37777777:00000000:00000000:000
   ae010020:2b: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
   00011043:00: 03 :40404040:dddddddd:dddddddd:00000001:eeeeeeee:00000000:00000000:001
   00011007:00: 07 :00000002:dddddddd:dddddddd:00000002:f7777777:00000000:00000000:001
```

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):



```
E:\term5\CSC3050\6>iverilog -o dsn ALU.v test_ALU.v

E:\term5\CSC3050\6>vvp dsn
instruction:op:func:  gr1   :   gr2  : reg_A  : reg_B  : reg_C  :   lo   :   hi   :zon
   xxxxxxxx:xx: xx :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:000
   00011020:00: 20 :40404040:dddddddd:40404040:dddddddd:1e1e1e1d:00000000:00000000:000
   2000ffff:08: 3f :7fffffff:dddddddd:7fffffff:ffffffff:7ffffffe:00000000:00000000:000
   00011021:00: 21 :40404040:5ddddddd:40404040:5ddddddd:9e1e1e1d:00000000:00000000:000
   2401ffff:09: 3f :40404040:5ddddddd:40404040:ffffffff:4040403f:00000000:00000000:000
   00011022:00: 22 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   00011023:00: 23 :7fffffff:0000ffff:7fffffff:0000ffff:7fff0000:00000000:00000000:000
   00010018:00: 18 :ffffffff:00000001:ffffffff:00000001:00000000:ffffffff:ffffffff:000
   00010019:00: 19 :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
   0001001a:00: 1a :ffffffe1:00000011:ffffffe1:00000011:00000000:ffffffff:fffffff2:000
   0001001b:00: 1b :0000000d:00000001:0000000d:00000001:00000000:0000000d:00000000:000
   00011024:00: 24 :fffffff1:00000011:fffffff1:00000011:00000011:00000000:00000000:000
   30410011:0c: 11 :ffffffe1:00000011:ffffffe1:00000011:00000001:00000000:00000000:000
   00011025:00: 25 :59ddd1dd:79ddd1dd:59ddd1dd:79ddd1dd:79ddd1dd:00000000:00000000:000
   34410011:0d: 11 :ffffffe1:79ddd1dd:ffffffe1:00000011:fffffff1:00000000:00000000:001
   00011027:00: 27 :ffffffe1:00000011:ffffffe1:00000011:0000000e:00000000:00000000:000
   00011026:00: 26 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
   38410011:0e: 11 :ffffffe1:00000011:ffffffe1:00000011:fffffff0:00000000:00000000:001
   10010020:04: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   0001102a:00: 2a :40404040:dddddddd:40404040:dddddddd:00000000:00000000:00000000:100
   2841000d:0a: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
   14010020:05: 20 :5ddddddd:5ddddddd:5ddddddd:5ddddddd:00000000:00000000:00000000:100
   0001102b:00: 2b :40404040:dddddddd:40404040:dddddddd:00000001:00000000:00000000:000
   2c41000d:0b: 0d :00000001:dddddddd:00000001:0000000d:00000001:00000000:00000000:000
   8c410020:23: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
   00011040:00: 00 :40404040:dddddddd:dddddddd:00000001:bbbbbbba:00000000:00000000:001
   00011004:00: 04 :00000002:dddddddd:dddddddd:00000002:77777774:00000000:00000000:000
   00011042:00: 02 :00000002:dddddddd:dddddddd:00000001:6eeeeeee:00000000:00000000:000
   00011006:00: 06 :00000002:dddddddd:dddddddd:00000002:37777777:00000000:00000000:000
   ae010020:2b: 20 :40404040:dddddddd:40404040:00000020:40404060:00000000:00000000:000
   00011043:00: 03 :40404040:dddddddd:dddddddd:00000001:eeeeeeee:00000000:00000000:001
   00011007:00: 07 :00000002:dddddddd:dddddddd:00000002:f7777777:00000000:00000000:001
```

6 Summary:

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