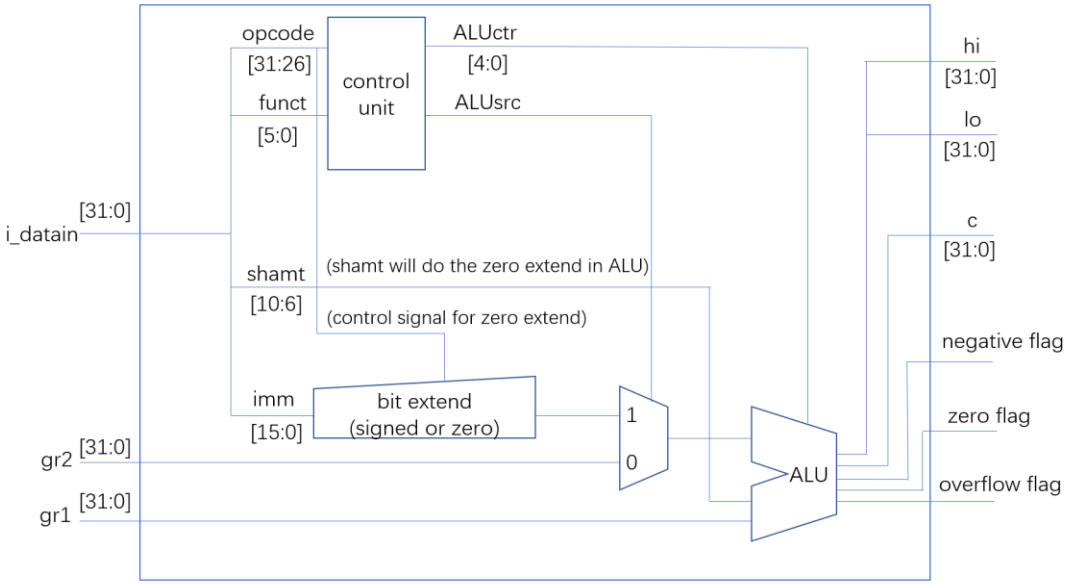
Some information about the project

The file ALU.v is the main program which implement the ALU. The file test_ALU.v is the test file used to test the ALU.v. And all the examples in this report are from the test file.

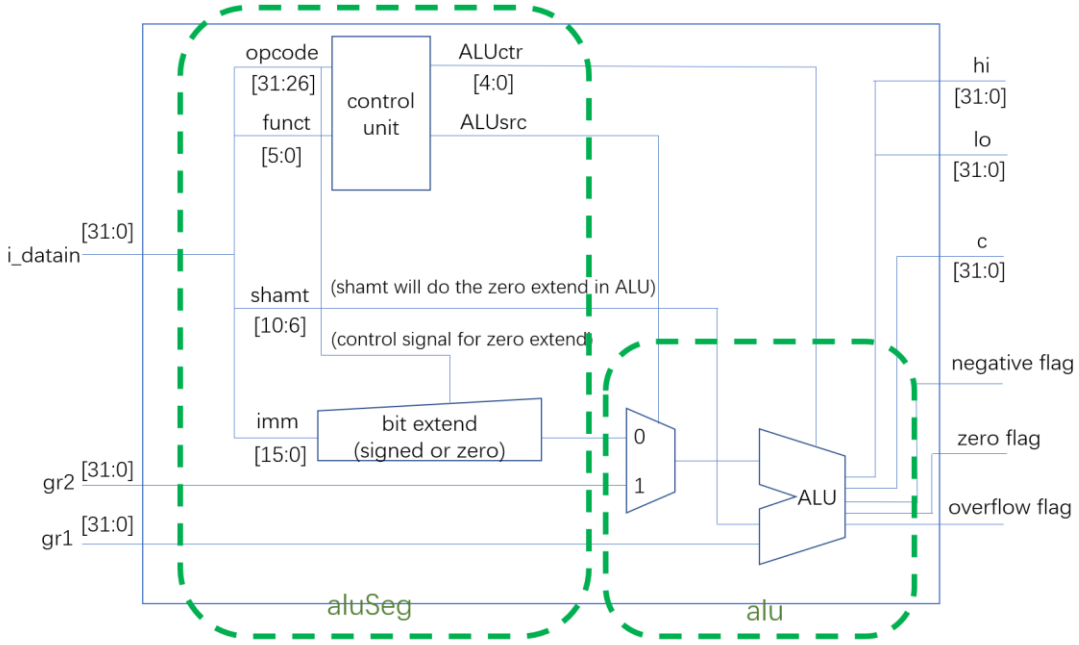
The main idea of the project

The project is to design a simple ALU using Verilog. The following are the details of the implementation.

Block diagram



The diagram above is the structure of my project. The main module of the project would take three inputs, which are i_datain, gr1 and gr2, which represent the instructions, the value of corresponding registers in the instructions.



The program is composed of two parts, which are indicated by the two green circles shown in the diagram above.

The first part is called aluSeg, which is indicated by the left circle. It is the main module of the program. And it has the interface shown in the following:

```
module aluSeg(i_datain, gr1, gr2, zero_wire, negative_wire, overflow_wire, c, hi_wire, lo_wire);

    //output result
    output signed[31:0] c;

    //flags
    output zero_wire, negative_wire, overflow_wire;

    //lo, hi
    output [31:0] lo_wire, hi_wire;

    input [31:0] i_datain, gr1, gr2;
```

In this part, eight registers are involved, which will store corresponding values. Notice that, reg_A store the value of gr1, and reg_B store the value of gr2. The registers are shown in the following picture:

```
reg [5:0] opcode, funct;
reg [4:0] shamt;
reg [4:0] aluctr;
reg alusrc;
reg [31:0] reg_A, reg_B;
reg [31:0] imm;
```

This part will generate the ALUctr and ALUsrc codes according to the opcodes

and funct codes. This process is done by using if and else loop to judges the opcodes and funct codes and giving out the ALUctr and ALUsrc codes accordingly. For the ALUsrc, the value of all the R-type instructions and bne, beq will be 0, while the

ALUctr	00000	00001	00010	00011	00100	00101	00110	00111
instruction	add,addi	addu,addiu	sub,beq,bne	subu	mult	multu	div	divu
ALUctr	01000	01001	01010	01011	01100	01101	01110	01111
instruction	and,andi	nor	or,ori	xor,xori	slt,slti	sltu.sltiu	lw	sw
ALUctr	10000	10001	10010	10011	10100	10101	111111	
instruction	sll	sllv	srl	srlv	sra	srav	invalid	

others will be 1. The related ALUctr codes are shown in the following table:

For the extension part, the 16-bit imm will be extended to 32 bits. The imm will first be signed extended to 32 bits by filling its signed bit to the former 16 bits, using the way shown in the following:

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

Since some instructions, like andi, ori, xori needs the imm to be zero extended. So after detecting these kind of instructions, the signed extended imm would change to zero extended, using the following way:

```
imm = {{16{1'b0}}, i_datain[15:0]};
```

After the first module finish the ALUctr and ALUsrc bit generation and imm extension, it will pass the corresponding value to the second module and call it.

The second module is alu part, which will implement the functions of calculations. It has the interfaces shown in the following:

```
module alu(aluctr, alusrc, gr1, gr2, imm, shamt, zero, negative, overflow, c, hi_wire, lo_wire);
```

```
//output result
output signed[31:0] c;

//flags
output zero, negative, overflow;

//lo, hi
output signed[31:0]lo_wire, hi_wire;

input signed[31:0] gr1,gr2;
input [4:0] shamt;
input [31:0] imm;
input [4:0] aluctr;
input alusrc;
```

This part uses 10 registers, which store the related values. Notice that the registers reg_A and reg_B will perform the corresponding signed calculations with gr1 and other values. And the unsigned_reg_A and unsigned_reg_B will be used to perform the unsigned calculations. The registers are shown below:

```
reg zero, negative, overflow;
reg signed[31:0] reg_C, reg_A, reg_B;
reg [31:0] unsigned_reg_B, unsigned_reg_A;
reg [31:0] lo, hi;
```

In this part, the ALUsrc will determine the input for reg_B or unsigned_reg_B. Also, the ALUctr would ask the alu to perform the corresponding instructions. Notice that although the input of shamt is only 5 bit, it will be later extend to 32 bits using zero extension in the alu.

Explanation of the required instructions:

Notice: for the display of results, the reg_A and reg_B infer to the registers in module 2, which means the value may not be equal to the reg_A and reg_B in module 1. Their values are called in the form in the red circles shown in the following:

```
$display("instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow");
$monitor("   %h:%h: %h :  %h  :   %h  :%h:%h:%h:%h:%h:%h:%h:%h:%h:   %h  :    %h    :    %h",
i_datain, testalu.opcode, testalu.funct, testalu.aluctr, testalu.alusrc, gr1 , gr2, c, testalu.alu0.reg_A, testalu.alu0.reg_B, testalu.c,
hi, lo, zero, negative, overflow);
```

add

```
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0100_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_1101_1101;
gr2<=32'b1111_1111_1111_1111_1111_1111_0010_0011;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0101_1101;
gr2<=32'b1111_1111_1111_1111_1111_1111_0010_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
   00620820:00: 20 :  00  :   0  :00000001:00000009:0000000a:00000001:00000009:0000000a:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620820:00: 20 :  00  :   0  :80000001:80000001:00000002:80000001:80000001:00000002:xxxxxxxx:xxxxxxxx:   0  :     0    :     1
   00620820:00: 20 :  00  :   0  :40000001:40000001:80000002:40000001:40000001:80000002:xxxxxxxx:xxxxxxxx:   0  :     1    :     1
   00620820:00: 20 :  00  :   0  :000000dd:ffffff23:00000000:000000dd:ffffff23:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   00620820:00: 20 :  00  :   0  :0000005d:ffffff23:ffffff80:0000005d:ffffff23:ffffff80:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A + reg_B.

This part is implemented by using a parallel adder, which is designed by connecting 32 full-adders together.

addi

```
#10 i_datain<=32'b001000_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001000_00011_00010_0000000000100010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001000_00011_00010_0000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001000_00011_00010_0000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0010_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   20628020:08: 20 :  00  :   1  :80000001:80000001:7fff8021:80000001:ffff8020:7fff8021:xxxxxxxx:xxxxxxxx:   0  :     0    :     1
   20620022:08: 22 :  00  :   1  :80000001:80000001:80000023:80000001:00000022:80000023:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   20620020:08: 20 :  00  :   1  :80000001:80000001:80000021:80000001:00000020:80000021:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   20620020:08: 20 :  00  :   1  :00000021:80000001:00000041:00000021:00000020:00000041:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1, reg_B = imm. Reg_C = reg_A + reg_B.

The algorithm is same as add.

addu:

```
//addu
#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0100_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0000_0000_0000_0000_0000_0000_1101_1101;
gr2<=32'b1111_1111_1111_1111_1111_1111_0010_0011;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0101_1101;
gr2<=32'b1111_1111_1111_1111_1111_1111_0010_0011;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620821:00: 21 :  01  :   0  :00000001:00000009:0000000a:00000001:00000009:0000000a:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   00620821:00: 21 :  01  :   0  :80000001:80000001:00000002:80000001:80000001:00000002:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   00620821:00: 21 :  01  :   0  :40000001:40000001:80000002:40000001:40000001:80000002:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   00620821:00: 21 :  01  :   0  :000000dd:ffffff23:00000000:000000dd:ffffff23:00000000:xxxxxxxx:xxxxxxxx:   1  :    0    :    0
   00620821:00: 21 :  01  :   0  :0000005d:ffffff23:ffffff80:0000005d:ffffff23:ffffff80:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   00620821:00: 21 :  01  :   0  :00000000:00000000:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :    0    :    0
```

In this part, unsigned_reg_A = gr1, unsigned_reg_B = gr2. Reg_C = unsigned_reg_A + unsigned_reg_B.

The algorithm is same as the add.

addiu:

```
//addiu
#10 i_datain<=32'b001001_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001001_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

#10 i_datain<=32'b001001_00011_00010_0000000000000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   24628020:09: 20 :  01  :   1  :80000001:80000001:7fff8021:80000001:ffff8020:7fff8021:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   24628020:09: 20 :  01  :   1  :00000001:00000003:ffff8021:00000001:ffff8020:ffff8021:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
   24620000:09: 00 :  01  :   1  :00000000:00000000:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :    0    :    0
```

In this part, unsigned_reg_A = gr1, unsigned_reg_B = imm. Reg_C =

unsigned_reg_A + unsigned_reg_B.

The algorithm is same as the add.

sub:

```
//sub
#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b1111_0000_0000_0000_0000_0000_0101_1101;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620822:00: 22 :  02  :   0  :f000005d:00000001:f000005c:f000005d:00000001:f000005c:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620822:00: 22 :  02  :   0  :80000001:7000005d:0fffffa4:80000001:7000005d:0fffffa4:xxxxxxxx:xxxxxxxx:   0  :     0    :     1
   00620822:00: 22 :  02  :   0  :00000001:f000005d:0fffffa4:00000001:f000005d:0fffffa4:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620822:00: 22 :  02  :   0  :00000000:00000000:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

In this part, reg_A = gr1, reg_B = imm. Reg_C = reg_A - reg_B. The subtraction is equal to add the minuend with the 2's complement of the subtrahend.

subu:

```
//subu
#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b0111_0000_0000_0000_0000_0000_0101_1101;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b0000_0000_0000_0000_0000_1000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620823:00: 23 :  03  :   0  :7000005d:80000001:f000005c:7000005d:80000001:f000005c:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620823:00: 23 :  03  :   0  :80000001:7000005d:0fffffa4:80000001:7000005d:0fffffa4:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620823:00: 23 :  03  :   0  :00000001:f000005d:0fffffa4:00000001:f000005d:0fffffa4:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620823:00: 23 :  03  :   0  :00000801:0000005d:000007a4:00000801:0000005d:000007a4:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620823:00: 23 :  03  :   0  :00000000:00000000:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :    0   :    0
```

In this part, unsigned_reg_A = gr1, unsigned_reg_B = gr2. Reg_C = unsigned_reg_A - unsigned_reg_B. The subtraction is equal to add the minuend with the 2's complement of the subtrahend.

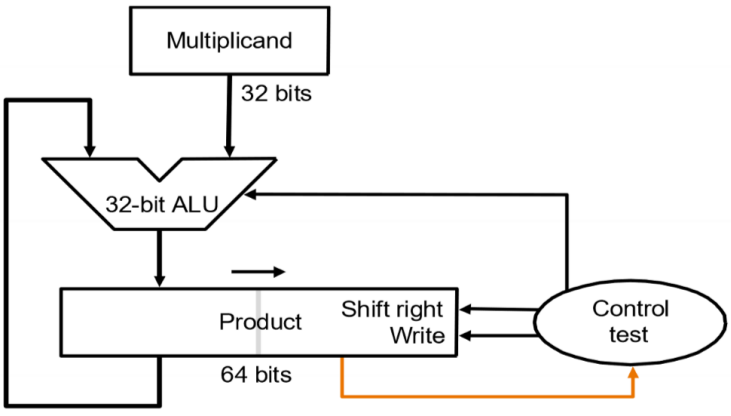
mult:

```
//mult
#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_0000_0000_0000_0000_0000_0101_1101;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620818:00: 18 :  04  :   0  :00000001:f000005d:00000000:00000001:f000005d:00000000:ffffffff:f000005d:   0  :    1   :    0
   00620818:00: 18 :  04  :   0  :00000003:0000005d:00000000:00000003:0000005d:00000000:00000000:00000117:   0  :    0   :    0
   00620818:00: 18 :  04  :   0  :00000001:00000000:00000000:00000001:00000000:00000000:00000000:00000000:   1  :    0   :    0
   00620818:00: 18 :  04  :   0  :80000001:f000005d:00000000:80000001:f000005d:00000000:07ffffd1:7000005d:   0  :    0   :    0
```

In the program, the multiplication is done by using the algorithm in Verilog. reg_A = gr1, reg_B = gr2. {hi, lo} = reg_A * reg_B.

The circuits of the multiplication are implemented like the diagram in the following:



Algorithm:

Set up: put the multiplier on the right 32 bits in the Product part, and set the left 32 bits of the Product to be 0.

1: check the rightmost bit of the Product.

1.1: if the right most bit is 1:

add the multiplicand with the left part, and put the result back to the left part of the product

1.2: if the right most bit is 0:

do not need to do the addition

2: shift the Product part left 1 bit

3: check whether it is the 32 repetition

3:1 if not:

jump to step 1

3:2 if so

finish the loop

multu:

```
//multu
#10 i_datain<=32'b000000_00011_00010_00001_00000_011001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_0000_0000_0000_0000_0000_0101_1101;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011001;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0010_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620819:00: 19 :  05  :   0  :00000001:f000005d:00000000:00000001:f000005d:00000000:00000000:f000005d:   0  :     0    :     0
   00620819:00: 19 :  05  :   0  :00000001:00000000:00000000:00000001:00000000:00000000:00000000:00000000:   1  :     0    :     0
   00620819:00: 19 :  05  :   0  :80000001:00000020:00000000:80000001:00000020:00000000:00000010:00000020:   0  :     0    :     0
```

In the program, the multiplication is done by using the algorithm in Verilog. reg_A_unsigned = gr1, reg_B_unsigned = gr2. {hi, lo} = reg_A_unsigned * reg_B_unsigned.

The algorithm of it is same as the mult.

div:

```
//div
#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b0000_0000_0000_0000_0000_0000_1100_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b1000_0000_0000_0110_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_1001_0000_1111_0000_0000_0011;
```

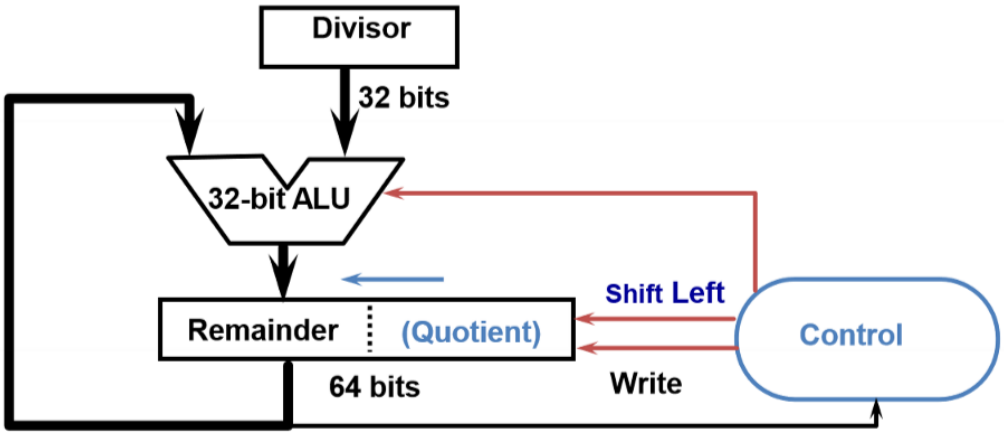
```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   0062081a:00: 1a :  06  :   0  :000000c1:00000032:00000000:000000c1:00000032:00000000:0000002b:00000003:   0  :     0    :     0
   0062081a:00: 1a :  06  :   0  :80000000:00000032:00000000:80000000:00000032:00000000:fffffffd0:fd70a3d8:   0  :     1    :     0
   0062081a:00: 1a :  06  :   0  :80060000:00000032:00000000:80060000:00000032:00000000:fffffffe0:fd70c290:   0  :     1    :     0
   0062081a:00: 1a :  06  :   0  :8c200000:00000000:00000000:8c200000:00000000:00000000:xxxxxxxx:xxxxxxxx:   0  :     x    :     0
   0062081a:00: 1a :  06  :   0  :80000000:ffffffff:00000000:80000000:ffffffff:00000000:00000000:80000000:   0  :     1    :     1
   0062081a:00: 1a :  06  :   0  :00000000:0090f003:00000000:00000000:0090f003:00000000:00000000:00000000:   1  :     0    :     0
```

In the program, the multiplication is done by using the algorithm in Verilog. reg_A = gr1, reg_B = gr2. lo = reg_A / reg_B. hi = reg_A % reg_B.

Notice that there is only one situation that the result is overflow, which is reg_A = 32'h8000_0000 and reg_B = -1.

Also, since reg_B = 0 is invalid, every time reg_B = 0, the results of lo and hi will be invalid that is 32’hxxxx_xxxx, and the negative part will be set to x.

The circuits of the division are implemented like the diagram in the following:



Algorithm:

Setup: Put the dividend on the right 32-bit part of the Reminder (Quotient) part.

1. Shift the Reminder part left by 1 bit.
2. Subtract the division from the left 32-bit part of the Reminder, and put the result back to the left half part of the Reminder.
3. Check the sign of the reminder:
   3.1 if the sign is negative:
   Restore the original value of the Reminder by adding the Divisor back to it. After that, shift the Reminder left by 1 bit, and set the new least bit to be 0.
   3.2 if the sign is positive:
   Shift the Reminder left by 1 bit and set the new least bit to be 1.
4. Check whether it is the 32 repetition:
   4.1 if not:
   Jump to the second step
   4.2 if so
   Finish the algorithm

divu:

```
//divu
#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b0000_0000_0000_0000_0000_0000_1100_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   0062081b:00: 1b :  07  :   0  :000000c1:00000032:00000000:000000c1:00000032:00000000:0000002b:00000003:   0  :     0    :     0
   0062081b:00: 1b :  07  :   0  :80000000:00000032:00000000:80000000:00000032:00000000:00000030:028f5c28:   0  :     0    :     0
   0062081b:00: 1b :  07  :   0  :8c200000:00000032:00000000:8c200000:00000032:00000000:0000002a:02cd70a3:   0  :     0    :     0
   0062081b:00: 1b :  07  :   0  :80000000:ffffffff:00000000:80000000:ffffffff:00000000:80000000:00000000:   0  :     0    :     0
   0062081b:00: 1b :  07  :   0  :00000000:ffffffff:00000000:00000000:ffffffff:00000000:00000000:00000000:   1  :     0    :     0
   0062081b:00: 1b :  07  :   0  :40000000:00000000:00000000:40000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In the program, the multiplication is done by using the algorithm in Verilog. reg_A_unsigned = gr1, reg_B_unsigned = gr2. lo = reg_A_unsigned / reg_B. hi = reg_A_unsigned % reg_B_unsigned.

Also, since reg_B_unsigned = 0 is invalid, every time reg_B_unsigned = 0, the results of lo and hi will be invalid that is 32’hxxxx_xxxx.

The algorithm is same as the div.

and:

```
//and
#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b1000_0000_1001_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_1001_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620824:00: 24 :  08  :   0  :80000000:ffffffff:80000000:80000000:ffffffff:80000000:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620824:00: 24 :  08  :   0  :80900000:00900032:00900000:80900000:00900032:00900000:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620824:00: 24 :  08  :   0  :8c200000:00000032:00000000:8c200000:00000032:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   00620824:00: 24 :  08  :   0  :80000000:ffffffff:80000000:80000000:ffffffff:80000000:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A & reg_B.

This part is implemented by doing the and operations of every bit of the two input numbers.

andi:

```
//andi
#10 i_datain<=32'b001100_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001100_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001100_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   30628020:0c: 20 :  08  :   1  :80000001:80000001:00000000:80000001:00008020:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   30628020:0c: 20 :  08  :   1  :80000020:80000001:00000020:80000020:00008020:00000020:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   30628020:0c: 20 :  08  :   1  :00000001:00000003:00000000:00000001:00008020:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = imm. reg_C = reg_A & reg_B.

The algorithm is same as and.

nor:

```
//nor
#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620827:00: 27 :  09  :   0  :80000000:ffffffff:00000000:80000000:ffffffff:00000000:xxxxxxxx:xxxxxxxx:   1  :    0   :    0
   00620827:00: 27 :  09  :   0  :80000000:00000032:7fffffcd:80000000:00000032:7fffffcd:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620827:00: 27 :  09  :   0  :8c200000:00000032:73dfffcd:8c200000:00000032:73dfffcd:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   00620827:00: 27 :  09  :   0  :80000000:ffffffff:00000000:80000000:ffffffff:00000000:xxxxxxxx:xxxxxxxx:   1  :    0   :    0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = ~(reg_A | reg_B).

This part is implemented by doing the nor operations of every bit of the two input numbers.

or:

```
//or
#10 i_datain<=32'b000000_00011_00010_00001_11000_100101;
gr1<=32'b1111_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_100101;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_11000_100101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e25:00: 25 :  0a  :   0  :fc200000:00000032:fc200032:fc200000:00000032:fc200032:xxxxxxxx:xxxxxxxx:   0  :    1   :    0
   00620e25:00: 25 :  0a  :   0  :40000000:ffffffff:ffffffff:40000000:ffffffff:ffffffff:xxxxxxxx:xxxxxxxx:   0  :    1   :    0
   00620e25:00: 25 :  0a  :   0  :00000000:00000000:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :    0   :    0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A | reg_B.

This part is implemented by doing the or operations of every bit of the two input numbers.

ori:

```
//ori
#10 i_datain<=32'b001101_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001101_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001101_00011_00010_0000000000000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   34628020:0d: 20 :  0a  :   1  :80000001:80000001:80008021:80000001:00008020:80008021:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   34628020:0d: 20 :  0a  :   1  :80000020:80000001:80008020:80000020:00008020:80008020:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   34620000:0d: 00 :  0a  :   1  :00000000:00000003:00000000:00000000:00000000:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = imm. reg_C = reg_A | reg_B.

The algorithm is same as or.

xor:

```
//xor
#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620826:00: 26 :  0b  :   0  :80000000:ffffffff:7fffffff:80000000:ffffffff:7fffffff:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620826:00: 26 :  0b  :   0  :80000000:00000032:80000032:80000000:00000032:80000032:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620826:00: 26 :  0b  :   0  :8c200000:00000032:8c200032:8c200000:00000032:8c200032:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620826:00: 26 :  0b  :   0  :80000000:ffffffff:7fffffff:80000000:ffffffff:7fffffff:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A ^ reg_B.

This part is implemented by doing the xor operations of every bit of the two input numbers.

xori:

```
//xori
#10 i_datain<=32'b001110_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001110_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001110_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   38628020:0e: 20 :  0b  :   1  :80000001:80000001:80008021:80000001:00008020:80008021:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   38628020:0e: 20 :  0b  :   1  :80000020:80000001:80008000:80000020:00008020:80008000:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   38628020:0e: 20 :  0b  :   1  :00000001:00000003:00008021:00000001:00008020:00008021:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = imm. reg_C = reg_A ^ reg_B.

The algorithm is same as the xor.

beq/bne:

```
//beq/bne
#10 i_datain<=32'b000100_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000100_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b000100_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   10628020:04: 20 :  02  :   0  :80000001:80000001:00000000:80000001:80000001:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   10628020:04: 20 :  02  :   0  :80000020:80000001:0000001f:80000020:80000001:0000001f:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   10628020:04: 20 :  02  :   0  :00000001:00000003:fffffffe:00000001:00000003:fffffffe:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
```

Since the program only implement the ALU part, which means it does not need to branch the pc, the calculation of it is same as sub.

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A - reg_B.

slt:

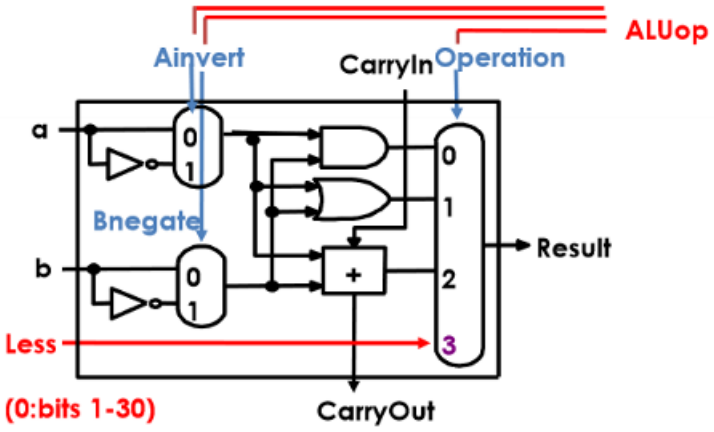
```
//slt
#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b0100_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   0062082a:00: 2a :  0c  :   0  :80000000:ffffffff:00000001:80000000:ffffffff:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   0062082a:00: 2a :  0c  :   0  :80000000:00000032:00000001:80000000:00000032:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   0062082a:00: 2a :  0c  :   0  :4c200000:00000032:00000000:4c200000:00000032:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   0062082a:00: 2a :  0c  :   0  :80000000:ffffffff:00000001:80000000:ffffffff:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

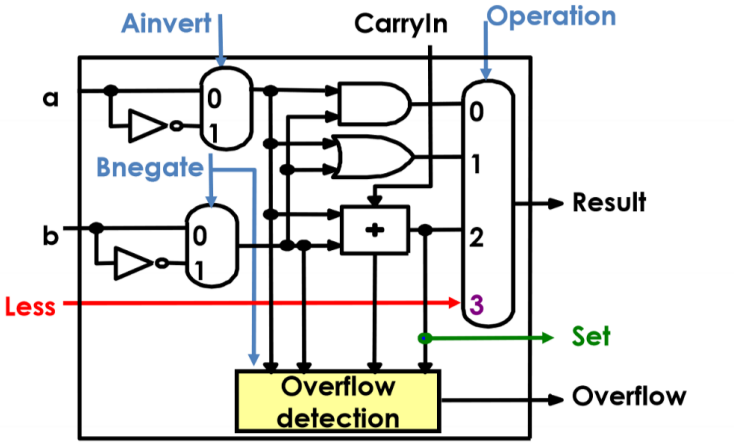
In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A < reg_B.

This is implemented by subtraction and return the result of the MSB.

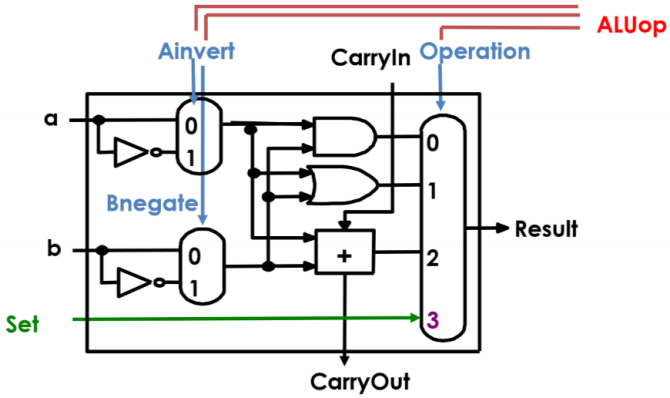
For all the ALU unit, the output operation will be selected as the “less”, which is the 3 shown in the following diagram. For the adder part of the ALU, it will be set to perform subtraction. For the bit of 1 to 30, the “less” will be set as 0, their structure is shown below.



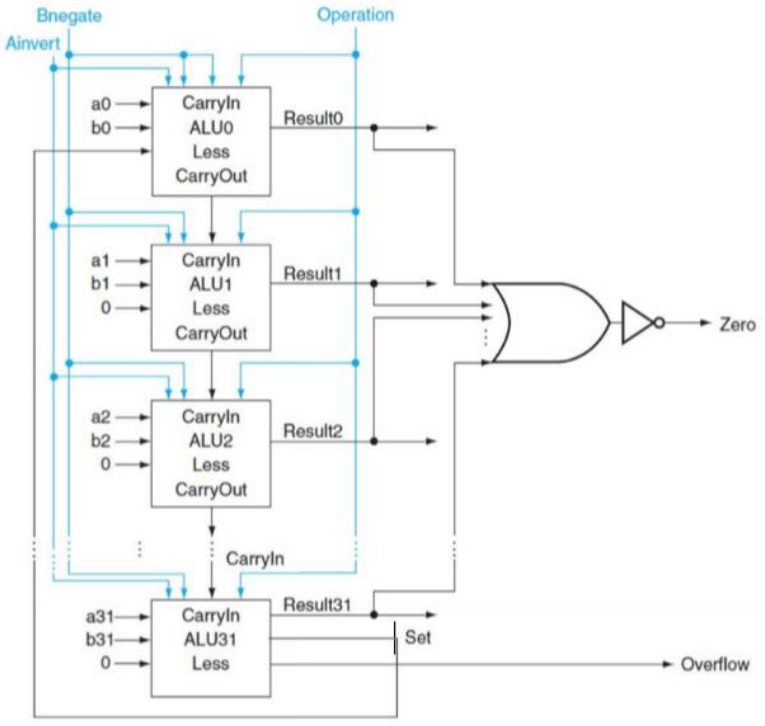
For the bit 31, the “less” is also set to be 0. And the subtraction result will be set out to bit 1. The structure is shown below:



For the bit 0, the “less” is set to be the result from the bit 31.



The final result will be generated by using an or gate to combine all the 32 results, which is shown in the following:



slti:

```
//slti
#10 i_datain<=32'b001010_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001010_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001010_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   28628020:0a: 20 :  0c  :   1  :80000001:80000001:00000001:80000001:ffff8020:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   28628020:0a: 20 :  0c  :   1  :80000020:80000001:00000001:80000020:ffff8020:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   28628020:0a: 20 :  0c  :   1  :00000001:00000003:00000000:00000001:ffff8020:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = imm. reg_C = reg_A < reg_B.

The algorithm is same as the slt.

sltu:

```
//sltu
#10 i_datain<=32'b000000_00011_00010_00001_00000_101011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101011;
gr1<=32'b1000_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_00000_101011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   0062082b:00: 2b :  0d  :   0  :80000000:ffffffff:00000001:80000000:ffffffff:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   0062082b:00: 2b :  0d  :   0  :80000000:00000032:00000000:80000000:00000032:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   0062082b:00: 2b :  0d  :   0  :8c200000:00000032:00000000:8c200000:00000032:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   0062082b:00: 2b :  0d  :   0  :80000000:ffffffff:00000001:80000000:ffffffff:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A_unsigned = gr1 and reg_B_unsigned = gr2. reg_C = reg_A_unsigned < reg_B_unigned.

The algorithm is same as the slt.

stliu:

```
//stliu
#10 i_datain<=32'b001011_00011_00010_0000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001011_00011_00010_0000000000010000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001011_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b001011_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   2c620020:0b: 20 :  0d  :   1  :00000020:00000001:00000000:00000020:00000020:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   2c620010:0b: 10 :  0d  :   1  :00000020:00000001:00000000:00000020:00000010:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   2c628020:0b: 20 :  0d  :   1  :80000001:80000001:00000001:80000001:ffff8020:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   2c628020:0b: 20 :  0d  :   1  :00000001:00000003:00000001:00000001:ffff8020:00000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A_unsigned = gr1 and reg_B_unsigned = imm. reg_C = reg_A_unsigned < reg_B_unigned.

The algorithm is same as the slt.

sw:

```
//sw
#10 i_datain<=32'b101011_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b101011_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b101011_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   ac628020:2b: 20 :  0f  :   1  :80000001:80000001:7fff8021:80000001:ffff8020:7fff8021:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   ac628020:2b: 20 :  0f  :   1  :80000020:80000001:7fff8040:80000020:ffff8020:7fff8040:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   ac628020:2b: 20 :  0f  :   1  :00000001:00000003:ffff8021:00000001:ffff8020:ffff8021:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

Since the program only implement the ALU part, which means it does not need to perform the word loading, the calculation of it is same as addition. But the program wants to distinguish it from addition, so sw was separated from addition.

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A + reg_B.

The algorithm is same as the slt.

lw:

```
//lw
#10 i_datain<=32'b100011_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b100011_00011_00010_1000000000100000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0001;

#10 i_datain<=32'b100011_00011_00010_1000000000100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   8c628020:23: 20 :  0e  :   1  :80000001:80000001:7fff8021:80000001:ffff8020:7fff8021:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   8c628020:23: 20 :  0e  :   1  :80000020:80000001:7fff8040:80000020:ffff8020:7fff8040:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
   8c628020:23: 20 :  0e  :   1  :00000001:00000003:ffff8021:00000001:ffff8020:ffff8021:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
```

Since the program only implement the ALU part, which means it does not need to perform the word storing, the calculation of it is same as addition. But the program wants to distinguish it from addition, so lw was separated from addition.

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A + reg_B.

sll:

```
//sll
#10 i_datain<=32'b000000_00011_00010_00001_11000_000000;
gr1<=32'b1111_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0100;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e00:00: 00 :  10  :   0  :fc200000:00000032:00000000:fc200000:00000018:00000000:xxxxxxxx:xxxxxxxx:   1  :    0   :    0
   00620e00:00: 00 :  10  :   0  :00000004:ffffffff:04000000:00000004:00000018:04000000:xxxxxxxx:xxxxxxxx:   0  :    0   :    0
```

In this part, reg_A = gr1 and reg_B = shamt. reg_C = reg_A << reg_B.

This instruction would shift the value left and set the new bit as 0.

sllv:

```
//sllv
#10 i_datain<=32'b000000_00011_00010_00001_11000_000100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0110;
gr2<=32'b0000_0000_0000_0000_0000_0000_0001_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e04:00: 04 :  11  :   0  :00000020:00000032:00000000:00000020:00000032:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   00620e04:00: 04 :  11  :   0  :00000006:00000010:00060000:00000006:00000010:00060000:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A<< reg_B.

This instruction would shift the value left and set the new bit as 0.

srl:

```
//srl
#10 i_datain<=32'b000000_00011_00010_00001_11000_000010;
gr1<=32'b1111_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000010;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e02:00: 02 :  12  :   0  :fc200000:00000032:000000fc:fc200000:00000018:000000fc:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620e02:00: 02 :  12  :   0  :40000000:ffffffff:00000040:40000000:00000018:00000040:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = shamt. reg_C = reg_A >> reg_B.

This instruction would shift the value right and set the new bit as 0.

srlv:

```
//srlv
#10 i_datain<=32'b000000_00011_00010_00001_11000_000110;
gr1<=32'b0000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000110;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0110;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e06:00: 06 :  13  :   0  :00000020:00000002:00000008:00000020:00000002:00000008:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620e06:00: 06 :  13  :   0  :80000006:00000008:00800000:80000006:00000008:00800000:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B = gr2. reg_C = reg_A >> reg_B.

This instruction would shift the value right and set the new bit as 0.

sra:

```
//sra
#10 i_datain<=32'b000000_00011_00010_00001_11000_000011;
gr1<=32'b1111_1100_0010_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0011_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000011;
gr1<=32'b0100_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e03:00: 03 :  14  :   0  :fc200000:00000032:fffffffc:fc200000:00000018:fffffffc:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620e03:00: 03 :  14  :   0  :40000000:ffffffff:00000040:40000000:00000018:00000040:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

In this part, reg_A = gr1 and reg_B =shamt. reg_C = reg_A >> >reg_B.

This instruction would shift the value right and set the new bit as 1.

srav:

```
//srav
#10 i_datain<=32'b000000_00011_00010_00001_11000_000111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0010_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0001_0010;

#10 i_datain<=32'b000000_00011_00010_00001_11000_000111;
gr1<=32'b0000_0000_0000_0000_0001_0000_0000_0110;
gr2<=32'b0000_0000_0000_0000_0000_0000_0001_0000;
```

```
instruction:op:func:ALUctr:ALUsrc:  gr1   :  gr2   :   c    : reg_A  : reg_B  : reg_C  :   hi   :   lo   : zero : negative : overflow
   00620e07:00: 07 :  15  :   0  :80000020:00000012:ffffe000:80000020:00000012:ffffe000:xxxxxxxx:xxxxxxxx:   0  :     1    :     0
   00620e07:00: 07 :  15  :   0  :00001006:00000010:00000000:00001006:00000010:00000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

In this part, reg_A = gr1 and reg_B =gr2. reg_C = reg_A >> >reg_B.

This instruction would shift the value right and set the new bit as 1.

An extra explanation about flags:

zero flag:

The zero flags are mainly determined by using a simple a ternary operator of Verilog, which is:

```
zero = reg_C ? 0 : 1;
```

For multiplication, it is used in this way to check whether the result is 0:

```
zero = {hi, lo} ? 0 :1;
```

For division, the zero flag is determined by the dividend, that is:

```
zero = reg_A ? 0 : 1;
```

negative flag:

The negative flags are determined by the value of the most significant bit, which is like (for the case of 32-bit signed calculation):

```
negative = reg_C[MSB];
```

For signed multiplication, it is determined by the most significant bit of hi, that is:

```
negative = hi[MSB];
```

For signed division, it is determined by the signed bit of the quotient:

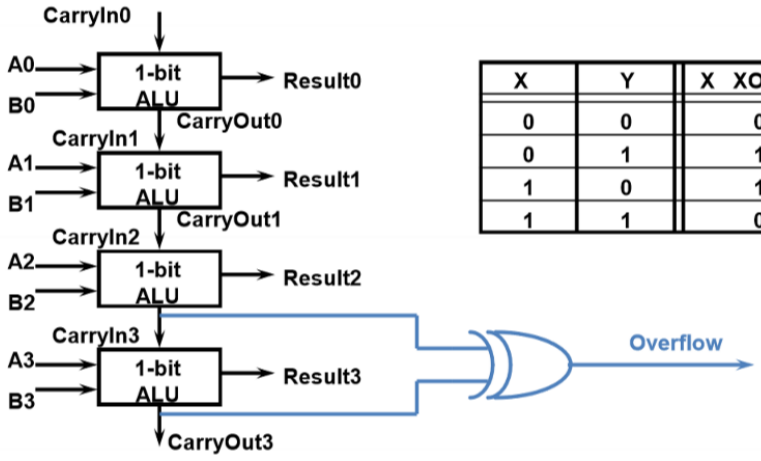
```
negative = lo[MSB];
```

For the unsigned calculation, the negative bit is set to be 0.

Overflow flag:

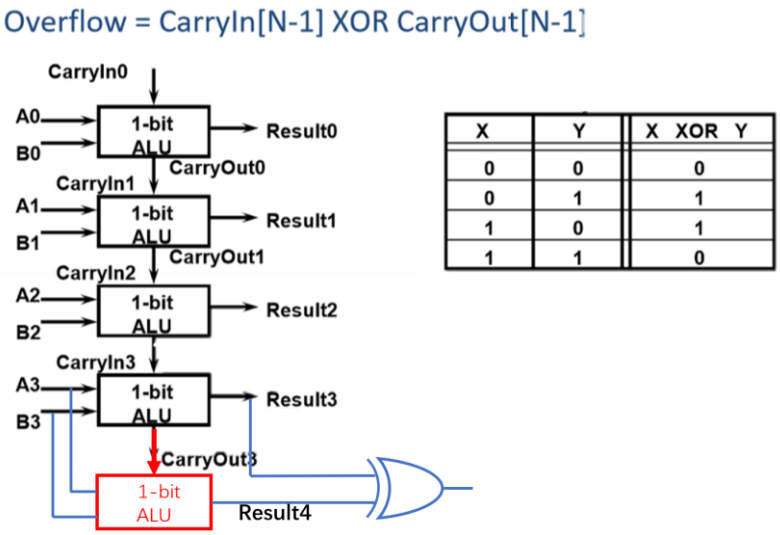
The overflow occurs when the result is set with the value of the result, which means the carry into most significant bit is not equal to the carry out of the most significant bit during the calculation. One way to detect the overflow signal is shown in the following:

- Overflow = CarryIn[N-1] XOR CarryOut[N-1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Based on this circuits, we can change a little bit to detect the carry out and carry in of the MSB.



Since CarryIn2 and CarryOut3 are added with the same number, which is A3 and B3, if CarryIn2 and CarryOut3 are different, then the Result3 and Result4 will be different. By checking the value of Result3 and Result4, we can get the situation of CarryIn3 and CarryOut3.

Based on this new idea, the following way is used in the program to detect the overflow:

```
//used to dectect overflow
reg extra;

{extra, reg_C} = {reg_A[MSB], reg_A} + {reg_B[MSB], reg_B};
overflow = extra ^ reg_C[MSB];
```

For division, since there is only one situation that the result is overflow, which is reg_A = 32'h8000_0000 and reg_B = -1, the overflow will be detected using this way:

```
if(reg_A==32'h8000_0000 && reg_B==-1)
    overflow = 1;
else
    overflow = 0;
```