CSC3050 Project 3 Report: Design of an ALU

杨宇凡 117010349

Summary:

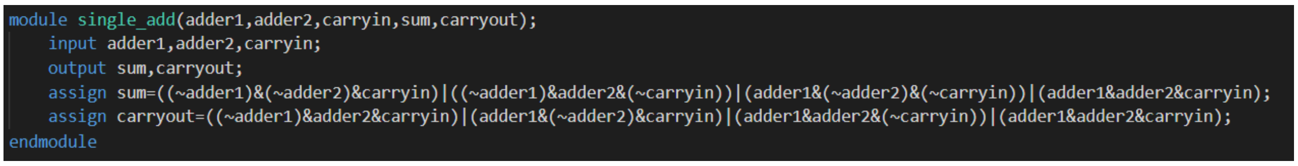
The method I do this project is a little bit complicated. I don't use any grammar skills (like for loop, if adjustment) but build all the electricity modules, and use these basic modules to make up the whole ALU. The reason why I do this is because hardware design requires us to think like hardware instead of software. If you just use these grammar skills, it's no difference of writing C languages program instead of a hardware. This project requires us to support about 20 MIPS functions, and I write each function as a Verilog module. Besides, I also write a lot of basic modules, like single adder, comparator, single divisor and so on. These basic modules help a lot for building these complicated modules.

Since the electricity map is very complicated, the running of testbench will take some time (about 30 seconds for each module's test, the whole test methods about 10 minutes).

Program Logic:

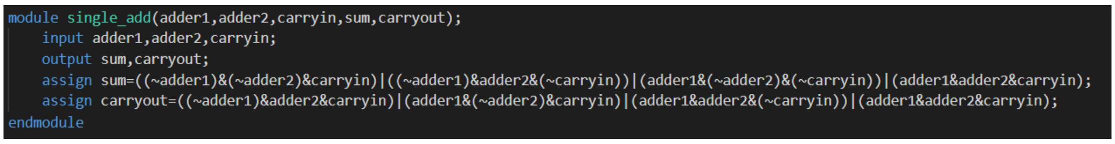
Write different MIPS's functions in different modules, and finally combine these modules together in the ALU. Therefore, these modules can get the input from the ALU, and transfer the output to the ALU. Then ALU needs to select the proper output according to the opcode. To do this, the multiplexers are designed as well. After the selection, ALU is responsible for transferring the data to the corresponding registers or other data structures.

Basic Module Instructions:



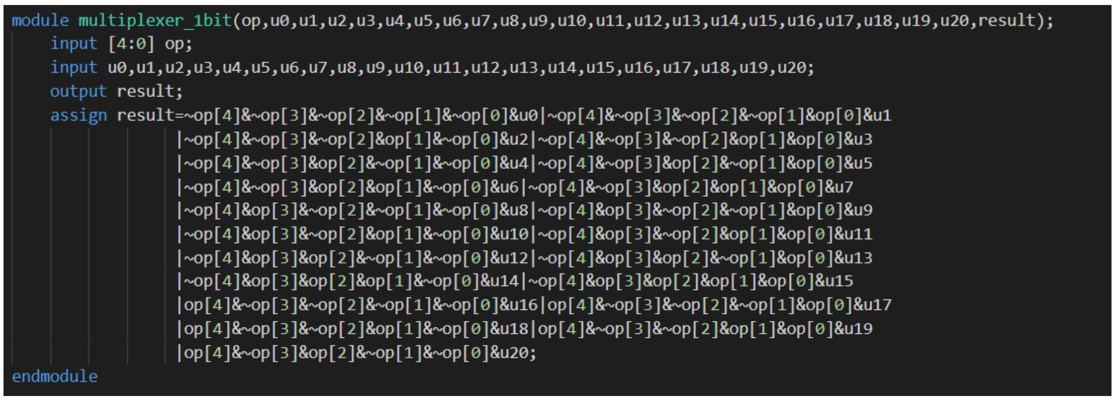
```
module single_add(adder1,adder2,carryin,sum,carryout);
    input adder1,adder2,carryin;
    output sum,carryout;
    assign sum=((~adder1)&(~adder2)&carryin)|((~adder1)&adder2&(~carryin))|(adder1&(~adder2)&(~carryin))|(adder1&adder2&carryin);
    assign carryout=((~adder1)&adder2&carryin)|(adder1&(~adder2)&carryin)|(adder1&adder2&(~carryin))|(adder1&adder2&carryin);
endmodule
```

This module is used for doing one bit add operation and output the carry out.



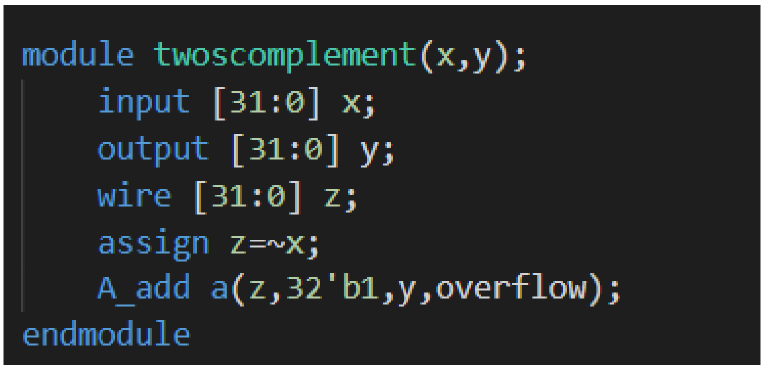
```
module single_add(adder1,adder2,carryin,sum,carryout);
    input adder1,adder2,carryin;
    output sum,carryout;
    assign sum=((~adder1)&(~adder2)&carryin)|((~adder1)&adder2&(~carryin))|(adder1&(~adder2)&(~carryin))|(adder1&adder2&carryin);
    assign carryout=((~adder1)&adder2&carryin)|(adder1&(~adder2)&carryin)|(adder1&adder2&(~carryin))|(adder1&adder2&carryin);
endmodule
```

This module is used for doing one bit comparing, which can be applied into making up more complicated comparator. The comparator is very use for the make up of some function modules. DIV is the representative.



```
module multiplexer_1bit(op,u0,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u12,u13,u14,u15,u16,u17,u18,u19,u20,result);
    input [4:0] op;
    input u0,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u12,u13,u14,u15,u16,u17,u18,u19,u20;
    output result;
    assign result=~op[4]&~op[3]&~op[2]&~op[1]&~op[0]&u0|~op[4]&~op[3]&~op[2]&~op[1]&op[0]&u1
                    |~op[4]&~op[3]&~op[2]&op[1]&~op[0]&u2|~op[4]&~op[3]&~op[2]&op[1]&op[0]&u3
                    |~op[4]&~op[3]&op[2]&~op[1]&~op[0]&u4|~op[4]&~op[3]&op[2]&~op[1]&op[0]&u5
                    |~op[4]&~op[3]&op[2]&op[1]&~op[0]&u6|~op[4]&~op[3]&op[2]&op[1]&op[0]&u7
                    |~op[4]&op[3]&~op[2]&~op[1]&~op[0]&u8|~op[4]&op[3]&~op[2]&~op[1]&op[0]&u9
                    |~op[4]&op[3]&~op[2]&op[1]&~op[0]&u10|~op[4]&op[3]&~op[2]&op[1]&op[0]&u11
                    |~op[4]&op[3]&op[2]&~op[1]&~op[0]&u12|~op[4]&op[3]&op[2]&~op[1]&op[0]&u13
                    |~op[4]&op[3]&op[2]&op[1]&~op[0]&u14|~op[4]&op[3]&op[2]&op[1]&op[0]&u15
                    |op[4]&~op[3]&~op[2]&~op[1]&~op[0]&u16|op[4]&~op[3]&~op[2]&~op[1]&op[0]&u17
                    |op[4]&~op[3]&~op[2]&op[1]&~op[0]&u18|op[4]&~op[3]&~op[2]&op[1]&op[0]&u19
                    |op[4]&~op[3]&op[2]&~op[1]&~op[0]&u20;
endmodule
```

This module is called multiplexer and it is responsible for select the proper data according to the opcode, which is very important for the ALU.

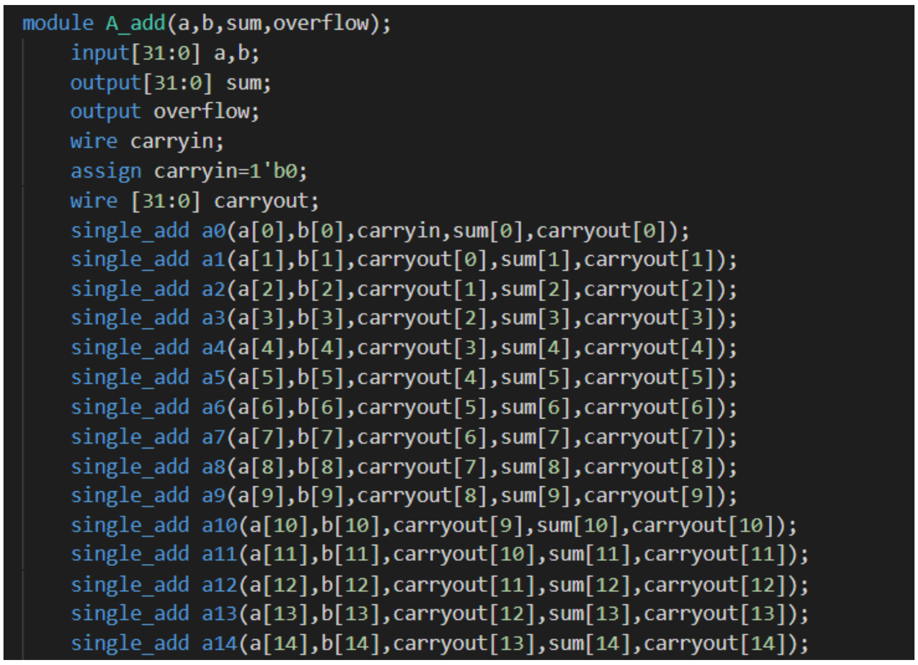


```
module twoscomplement(x,y);
    input [31:0] x;
    output [31:0] y;
    wire [31:0] z;
    assign z=~x;
    A_add a(z,32'b1,y,overflow);
endmodule
```

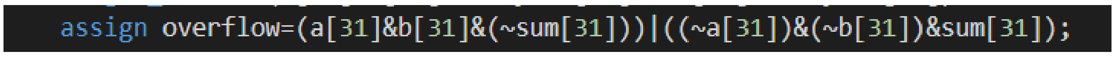
Two's complement module is for doing the two's complement, which is very useful in the subtraction operation.

Function module instructions:

In this part I will choose some representative modules to introduce.

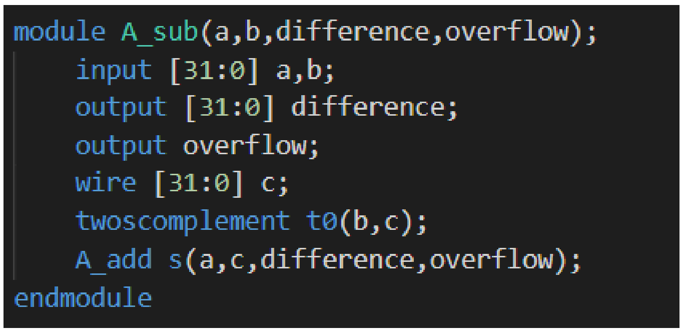


```
module A_add(a,b,sum,overflow);
    input[31:0] a,b;
    output[31:0] sum;
    output overflow;
    wire carryin;
    assign carryin=1'b0;
    wire [31:0] carryout;
    single_add a0(a[0],b[0],carryin,sum[0],carryout[0]);
    single_add a1(a[1],b[1],carryout[0],sum[1],carryout[1]);
    single_add a2(a[2],b[2],carryout[1],sum[2],carryout[2]);
    single_add a3(a[3],b[3],carryout[2],sum[3],carryout[3]);
    single_add a4(a[4],b[4],carryout[3],sum[4],carryout[4]);
    single_add a5(a[5],b[5],carryout[4],sum[5],carryout[5]);
    single_add a6(a[6],b[6],carryout[5],sum[6],carryout[6]);
    single_add a7(a[7],b[7],carryout[6],sum[7],carryout[7]);
    single_add a8(a[8],b[8],carryout[7],sum[8],carryout[8]);
    single_add a9(a[9],b[9],carryout[8],sum[9],carryout[9]);
    single_add a10(a[10],b[10],carryout[9],sum[10],carryout[10]);
    single_add a11(a[11],b[11],carryout[10],sum[11],carryout[11]);
    single_add a12(a[12],b[12],carryout[11],sum[12],carryout[12]);
    single_add a13(a[13],b[13],carryout[12],sum[13],carryout[13]);
    single_add a14(a[14],b[14],carryout[13],sum[14],carryout[14]);
```



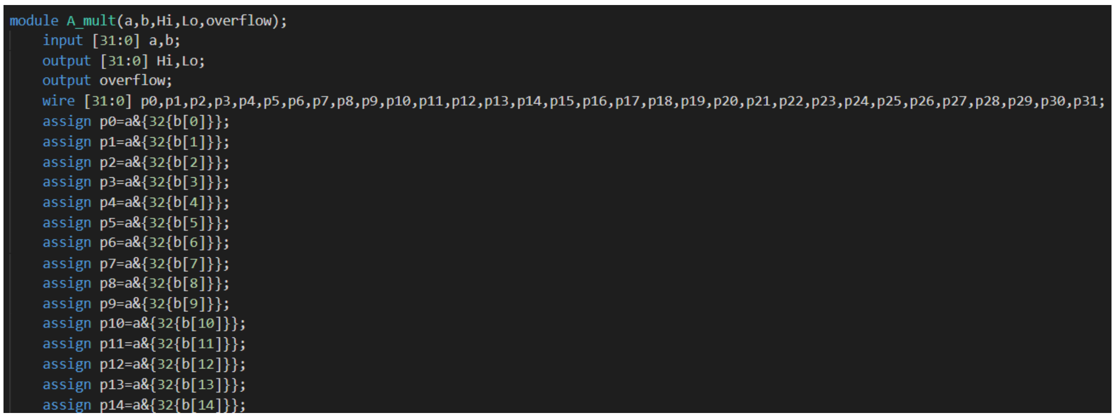
```
    assign overflow=(a[31]&b[31]&(~sum[31]))|((~a[31])&(~b[31])&sum[31]);
```

Add module is made up of the single adders one by one. The overflow check method is shown above.

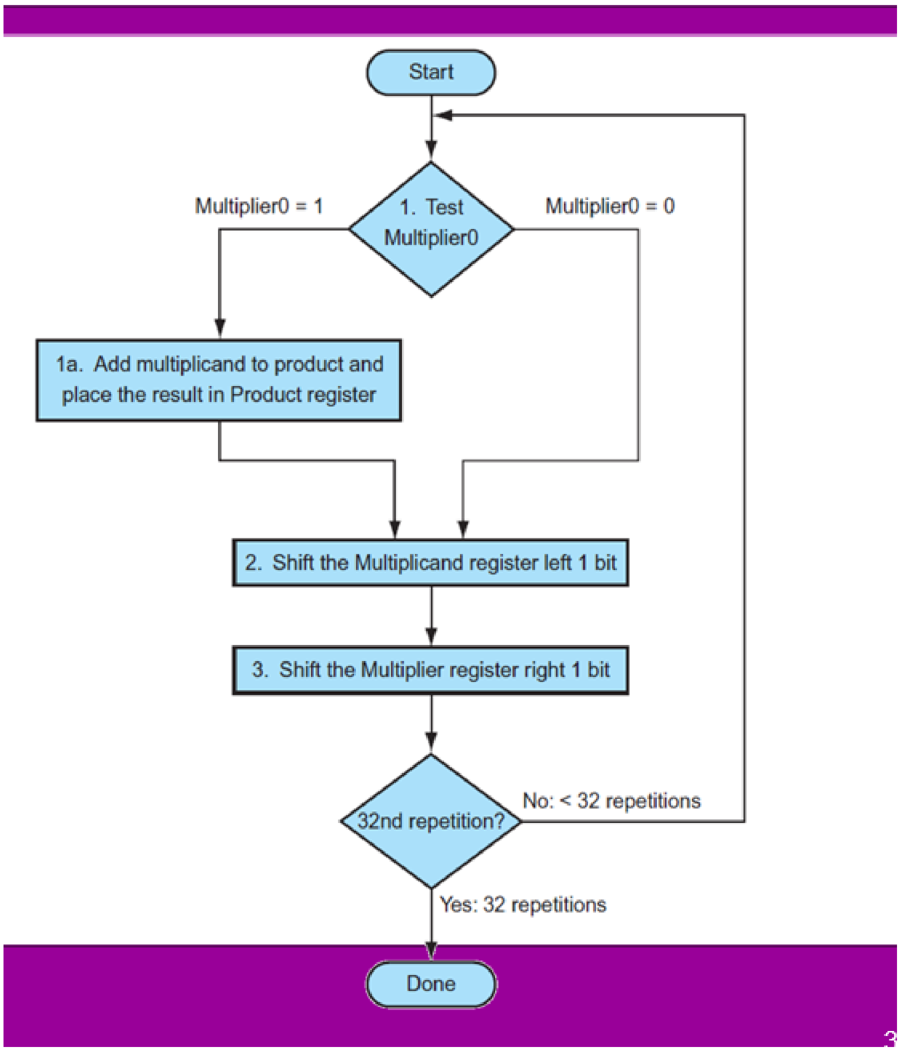


```
module A_sub(a,b,difference,overflow);
    input [31:0] a,b;
    output [31:0] difference;
    output overflow;
    wire [31:0] c;
    twoscomplement t0(b,c);
    A_add s(a,c,difference,overflow);
endmodule
```

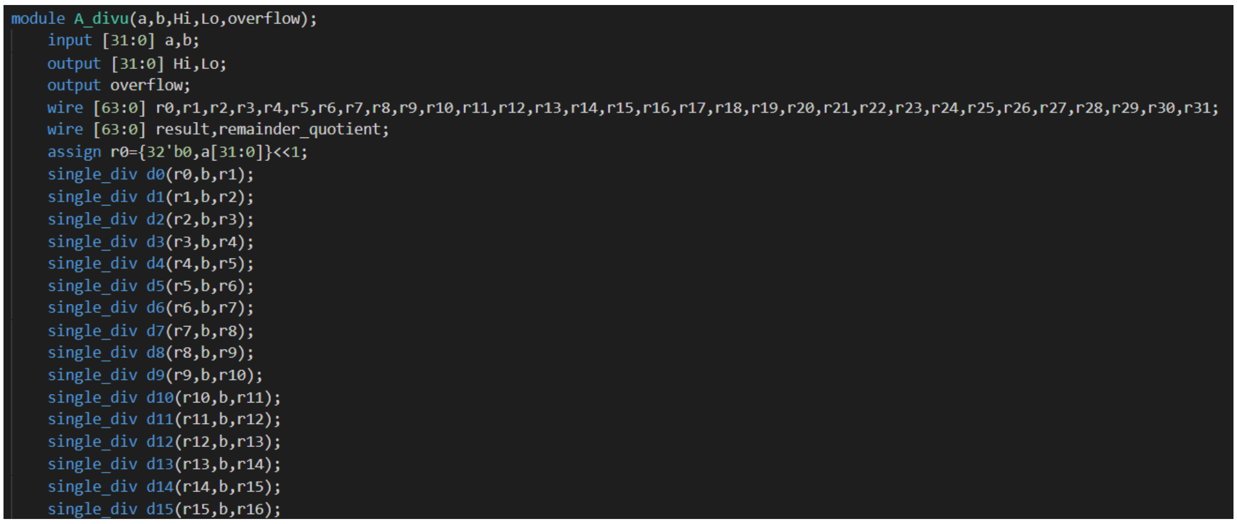
The sub operation consist of two’s complement module and the add module.



```
module A_mult(a,b,Hi,Lo,overflow);
    input [31:0] a,b;
    output [31:0] Hi,Lo;
    output overflow;
    wire [31:0] p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p23,p24,p25,p26,p27,p28,p29,p30,p31;
    assign p0=a&{32{b[0]}};
    assign p1=a&{32{b[1]}};
    assign p2=a&{32{b[2]}};
    assign p3=a&{32{b[3]}};
    assign p4=a&{32{b[4]}};
    assign p5=a&{32{b[5]}};
    assign p6=a&{32{b[6]}};
    assign p7=a&{32{b[7]}};
    assign p8=a&{32{b[8]}};
    assign p9=a&{32{b[9]}};
    assign p10=a&{32{b[10]}};
    assign p11=a&{32{b[11]}};
    assign p12=a&{32{b[12]}};
    assign p13=a&{32{b[13]}};
    assign p14=a&{32{b[14]}};
```


Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

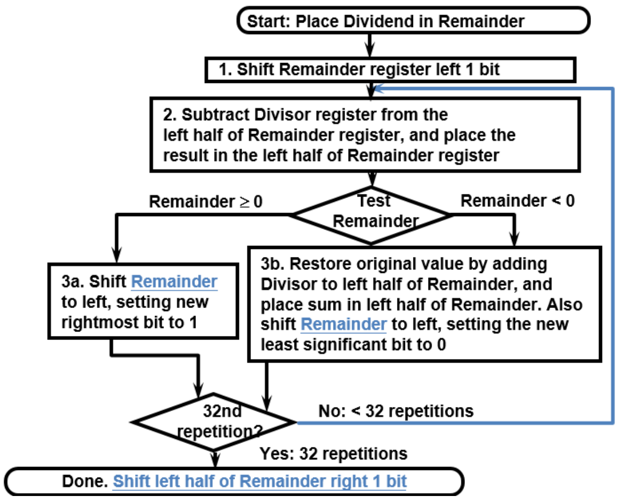
This is the multiple unsigned operation module. To build it, I use the method showed above. It is kind of complicated.



```
module A_divu(a,b,Hi,Lo,overflow);
    input [31:0] a,b;
    output [31:0] Hi,Lo;
    output overflow;
    wire [63:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16,r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31;
    wire [63:0] result,remainder_quotient;
    assign r0={32'b0,a[31:0]}<<1;
    single_div d0(r0,b,r1);
    single_div d1(r1,b,r2);
    single_div d2(r2,b,r3);
    single_div d3(r3,b,r4);
    single_div d4(r4,b,r5);
    single_div d5(r5,b,r6);
    single_div d6(r6,b,r7);
    single_div d7(r7,b,r8);
    single_div d8(r8,b,r9);
    single_div d9(r9,b,r10);
    single_div d10(r10,b,r11);
    single_div d11(r11,b,r12);
    single_div d12(r12,b,r13);
    single_div d13(r13,b,r14);
    single_div d14(r14,b,r15);
    single_div d15(r15,b,r16);
```

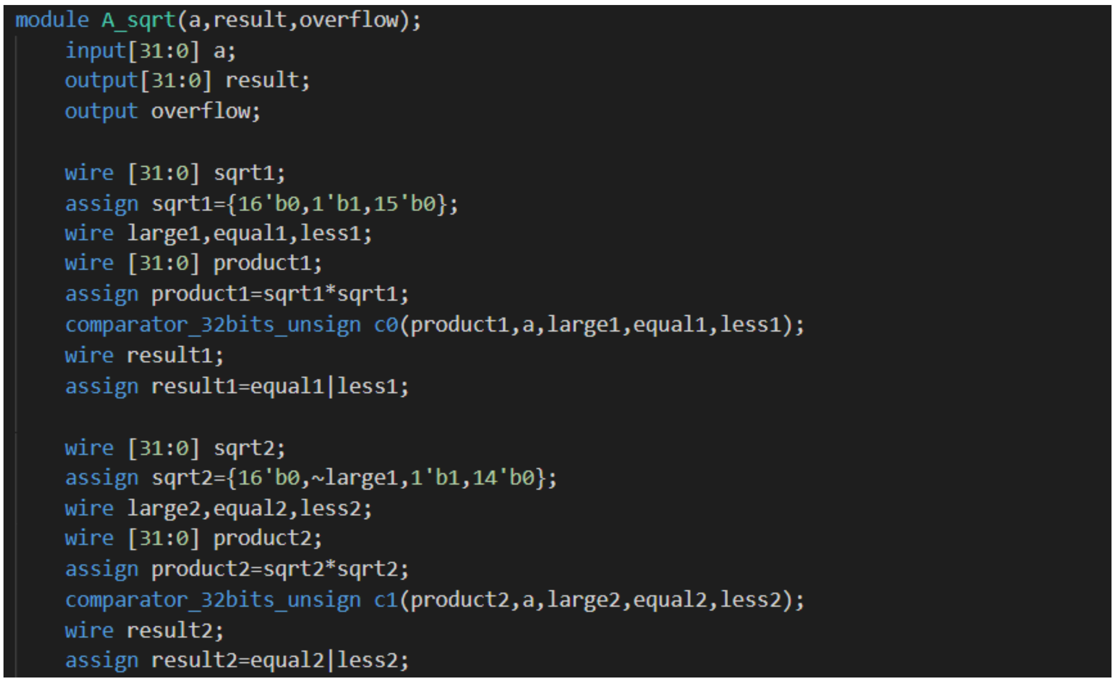
```
0111 / 0010

Step   Remainder   Div.
0      0000 0111   0010
1.1    0000 1110
1.2    1110 1110
1.3b   0001 1100
2.2    1111 1100
2.3b   0011 1000
3.2    0001 1000
3.3a   0011 0001
4.2    0001 0001
4.3a   0010 0011
       0001 0011
```


Start: Place Dividend in Remainder
1. Shift Remainder register left 1 bit
2. Subtract Divisor register from the left half of Remainder register, and place the result in the left half of Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
3a. Shift Remainder to left, setting new rightmost bit to 1
3b. Restore original value by adding Divisor to left half of Remainder, and place sum in left half of Remainder. Also shift Remainder to left, setting the new least significant bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

This is the division unsigned module. The method is shown above which uses shifting, comparison and loop operations. Therefore it is kind of complicated to realize with the electricity map. Before the whole module, the single division module is built to simply conducted the loop.

Besides, the unsigned operation of add and sub ignore the overflow operation. The signed operation of division and multiply uses the sign checking.



```
module A_sqrt(a,result,overflow);
    input[31:0] a;
    output[31:0] result;
    output overflow;

    wire [31:0] sqrt1;
    assign sqrt1={16'b0,1'b1,15'b0};
    wire large1,equal1,less1;
    wire [31:0] product1;
    assign product1=sqrt1*sqrt1;
    comparator_32bits_unsign c0(product1,a,large1,equal1,less1);
    wire result1;
    assign result1=equal1|less1;

    wire [31:0] sqrt2;
    assign sqrt2={16'b0,~large1,1'b1,14'b0};
    wire large2,equal2,less2;
    wire [31:0] product2;
    assign product2=sqrt2*sqrt2;
    comparator_32bits_unsign c1(product2,a,large2,equal2,less2);
    wire result2;
    assign result2=equal2|less2;
```

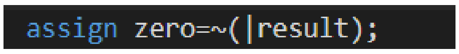
The design of sqrt is very complicated. The basic logic is to assume a correct answer and then checking. This method is easy to realized by software programming, but very difficult for hardware design. However, this time I still use this logic.

Three flags:

✧ Overflow

Overflow flag is check by each module. The method for each module is different. Many functions ignore the overflow, like most unsigned functions.

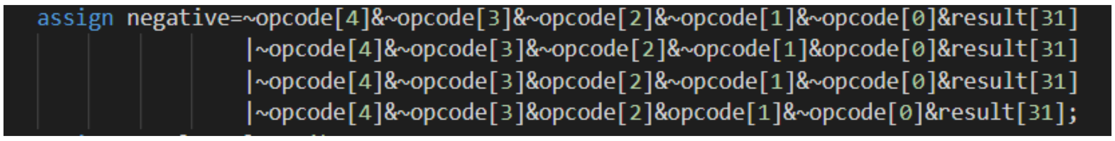
✧ Zero:



```
assign zero=~(|result);
```

Zero flag is checked in ALU by the above logic, doing the or operation inside the result.

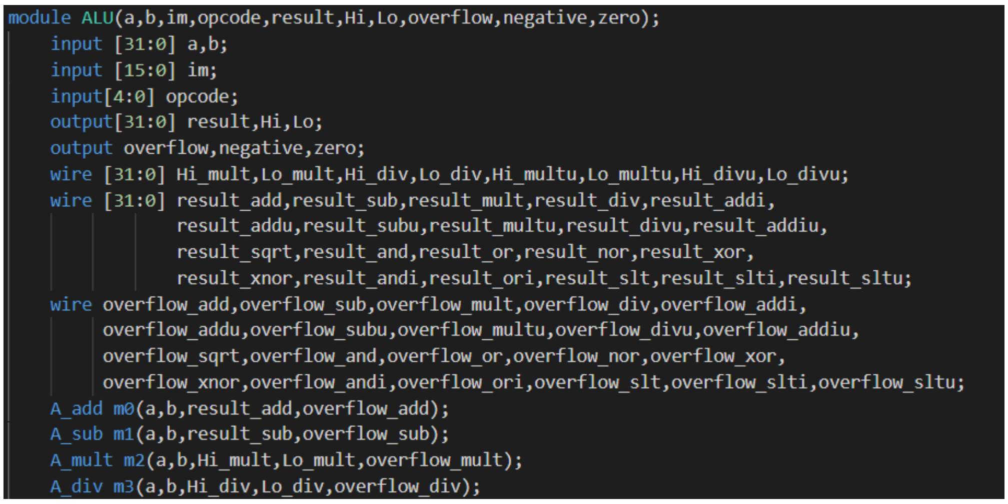
✧ Negative:



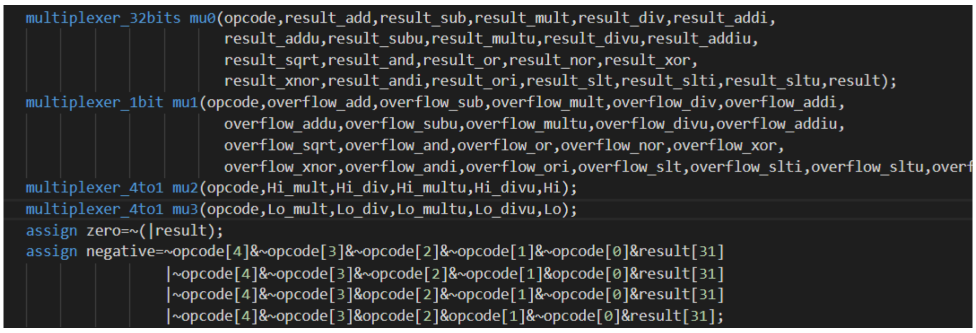
```
assign negative=~opcode[4]&~opcode[3]&~opcode[2]&~opcode[1]&~opcode[0]&result[31]
                |~opcode[4]&~opcode[3]&~opcode[2]&~opcode[1]&opcode[0]&result[31]
                |~opcode[4]&~opcode[3]&opcode[2]&~opcode[1]&~opcode[0]&result[31]
                |~opcode[4]&~opcode[3]&opcode[2]&opcode[1]&~opcode[0]&result[31];
```

Negative flag is done by the above logic, which will be only activated in some specific functions (opcodes) like signed adder, unsigned sub.

ALU:



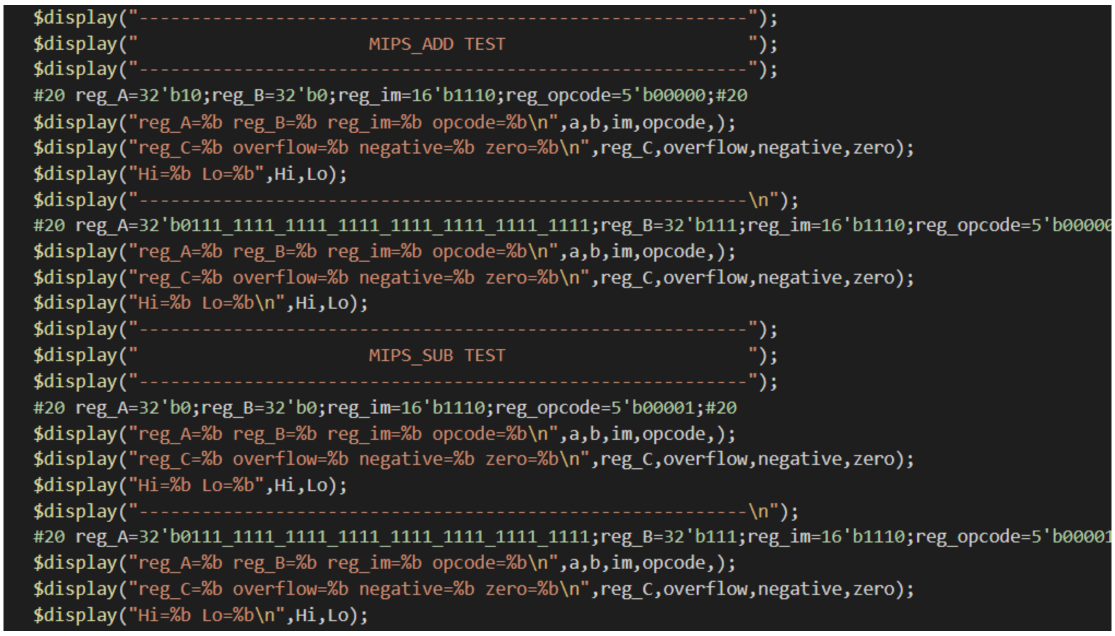
```
module ALU(a,b,im,opcode,result,Hi,Lo,overflow,negative,zero);
    input [31:0] a,b;
    input [15:0] im;
    input[4:0] opcode;
    output[31:0] result,Hi,Lo;
    output overflow,negative,zero;
    wire [31:0] Hi_mult,Lo_mult,Hi_div,Lo_div,Hi_multu,Lo_multu,Hi_divu,Lo_divu;
    wire [31:0] result_add,result_sub,result_mult,result_div,result_addi,
                result_addu,result_subu,result_multu,result_divu,result_addiu,
                result_sqrt,result_and,result_or,result_nor,result_xor,
                result_xnor,result_andi,result_ori,result_slt,result_slti,result_sltu;
    wire overflow_add,overflow_sub,overflow_mult,overflow_div,overflow_addi,
         overflow_addu,overflow_subu,overflow_multu,overflow_divu,overflow_addiu,
         overflow_sqrt,overflow_and,overflow_or,overflow_nor,overflow_xor,
         overflow_xnor,overflow_andi,overflow_ori,overflow_slt,overflow_slti,overflow_sltu;
    A_add m0(a,b,result_add,overflow_add);
    A_sub m1(a,b,result_sub,overflow_sub);
    A_mult m2(a,b,Hi_mult,Lo_mult,overflow_mult);
    A_div m3(a,b,Hi_div,Lo_div,overflow_div);
```



```
    multiplexer_32bits mu0(opcode,result_add,result_sub,result_mult,result_div,result_addi,
                           result_addu,result_subu,result_multu,result_divu,result_addiu,
                           result_sqrt,result_and,result_or,result_nor,result_xor,
                           result_xnor,result_andi,result_ori,result_slt,result_slti,result_sltu,result);
    multiplexer_1bit mu1(opcode,overflow_add,overflow_sub,overflow_mult,overflow_div,overflow_addi,
                           overflow_addu,overflow_subu,overflow_multu,overflow_divu,overflow_addiu,
                           overflow_sqrt,overflow_and,overflow_or,overflow_nor,overflow_xor,
                           overflow_xnor,overflow_andi,overflow_ori,overflow_slt,overflow_slti,overflow_sltu,overf
    multiplexer_4to1 mu2(opcode,Hi_mult,Hi_div,Hi_multu,Hi_divu,Hi);
    multiplexer_4to1 mu3(opcode,Lo_mult,Lo_div,Lo_multu,Lo_divu,Lo);
    assign zero=~(|result);
    assign negative=~opcode[4]&~opcode[3]&~opcode[2]&~opcode[1]&~opcode[0]&result[31]
                    |~opcode[4]&~opcode[3]&~opcode[2]&~opcode[1]&opcode[0]&result[31]
                    |~opcode[4]&~opcode[3]&opcode[2]&~opcode[1]&~opcode[0]&result[31]
                    |~opcode[4]&~opcode[3]&opcode[2]&opcode[1]&~opcode[0]&result[31];
```

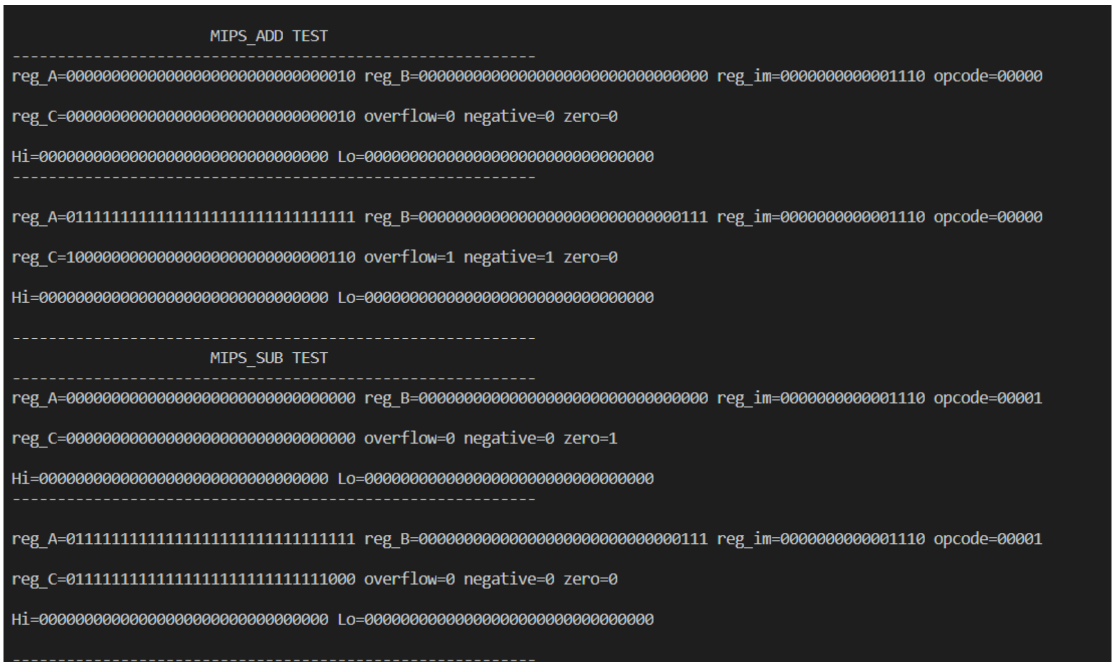
ALU is shown above. It uses the multiplexer to select the right output and transfer it to the corresponding register or other data structures.

Test



```
  $display("----------------------------------------------------------");
  $display("                      MIPS_ADD TEST                       ");
  $display("----------------------------------------------------------");
  #20 reg_A=32'b10;reg_B=32'b0;reg_im=16'b1110;reg_opcode=5'b00000;#20
  $display("reg_A=%b reg_B=%b reg_im=%b opcode=%b\n",a,b,im,opcode,);
  $display("reg_C=%b overflow=%b negative=%b zero=%b\n",reg_C,overflow,negative,zero);
  $display("Hi=%b Lo=%b",Hi,Lo);
  $display("----------------------------------------------------------\n");
  #20 reg_A=32'b0111_1111_1111_1111_1111_1111_1111_1111;reg_B=32'b111;reg_im=16'b1110;reg_opcode=5'b0000
  $display("reg_A=%b reg_B=%b reg_im=%b opcode=%b\n",a,b,im,opcode,);
  $display("reg_C=%b overflow=%b negative=%b zero=%b\n",reg_C,overflow,negative,zero);
  $display("Hi=%b Lo=%b\n",Hi,Lo);
  $display("----------------------------------------------------------");
  $display("                      MIPS_SUB TEST                       ");
  $display("----------------------------------------------------------");
  #20 reg_A=32'b0;reg_B=32'b0;reg_im=16'b1110;reg_opcode=5'b00001;#20
  $display("reg_A=%b reg_B=%b reg_im=%b opcode=%b\n",a,b,im,opcode,);
  $display("reg_C=%b overflow=%b negative=%b zero=%b\n",reg_C,overflow,negative,zero);
  $display("Hi=%b Lo=%b",Hi,Lo);
  $display("----------------------------------------------------------\n");
  #20 reg_A=32'b0111_1111_1111_1111_1111_1111_1111_1111;reg_B=32'b111;reg_im=16'b1110;reg_opcode=5'b00001
  $display("reg_A=%b reg_B=%b reg_im=%b opcode=%b\n",a,b,im,opcode,);
  $display("reg_C=%b overflow=%b negative=%b zero=%b\n",reg_C,overflow,negative,zero);
  $display("Hi=%b Lo=%b\n",Hi,Lo);
```

The test module is shown above. Each instruction consists of two instances. All the related register, opcode, three flags, Hi and Lo will be shown on the test result. Following are the example output:



```
                      MIPS_ADD TEST
----------------------------------------------------------
reg_A=00000000000000000000000000000010 reg_B=00000000000000000000000000000000 reg_im=0000000000001110 opcode=00000

reg_C=00000000000000000000000000000010 overflow=0 negative=0 zero=0

Hi=00000000000000000000000000000000 Lo=00000000000000000000000000000000
----------------------------------------------------------

reg_A=01111111111111111111111111111111 reg_B=00000000000000000000000000000111 reg_im=0000000000001110 opcode=00000

reg_C=10000000000000000000000000000110 overflow=1 negative=1 zero=0

Hi=00000000000000000000000000000000 Lo=00000000000000000000000000000000

----------------------------------------------------------
                      MIPS_SUB TEST
----------------------------------------------------------
reg_A=00000000000000000000000000000000 reg_B=00000000000000000000000000000000 reg_im=0000000000001110 opcode=00001

reg_C=00000000000000000000000000000000 overflow=0 negative=0 zero=1

Hi=00000000000000000000000000000000 Lo=00000000000000000000000000000000
----------------------------------------------------------

reg_A=01111111111111111111111111111111 reg_B=00000000000000000000000000000111 reg_im=0000000000001110 opcode=00001

reg_C=01111111111111111111111111111000 overflow=0 negative=0 zero=0

Hi=00000000000000000000000000000000 Lo=00000000000000000000000000000000

----------------------------------------------------------
```