3 计算机算术算法和实现

英语

1. *contraction*收缩
2. *ripple*波纹

*ripple adder*行波进位加法器

3. *propagator*传播者
4. *divisor*除数
5. *denominator*分母
6. *converge*收敛
7. *parentheses*括号

1. *complement*补充、补足物
2. *cascade*瀑布、逐级
3. *carry*进位
4. *dividend*被除数
5. *quotient* 商数
6. *be rounded to*四舍五入为
7. *approximation*近似值
8. *radicand*被开方数

3.1二进制整数

3.1.1 二进制和十六进制形式

只给定一个二进制数是不知道其含义的

比如给定：00110011110111100000000100000000

可以是一个整数：870187264

可以是一个浮点数：0.000000103378624771721661090850830078125

可以是一条MIPS指令： *addi $30, $30, 256*

可以是一个地址

可以是一个媒体数据等

> *The correct answer is “don’t know.” The exact meaning of the code depends on where it will be used. If it is treated as an integer, its value is 870,187,264. If it is a floating-point number, its value is 0.000000103378624771721661090850830078125. If it is an instruction and executed by a MIPS CPU (microprocessor without interlocked pipeline stages central processing unit), then it is addi $30, $30, 256, an immediate addition instruction. It may be an IP address, data of image or music, or something else.*

十六进制是为方便记忆产生的，每4位二进制看作一位十六进制

Table 3.1 Relationship between hexadecimal numbers and binary numbers

Binary number	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal number	0	1	2	3	4	5	6	7
Binary number	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal number	8	9	a	b	c	d	e	f

3.1.2 无符号二进制整数

Suppose that we use an n-bit binary number $b_{n-1}b_{n-2} \cdots b_1b_0$, where b_i $(i = 0, 1, \cdots, n-2, n-1)$ is a 0 or a 1, to denote an unsigned number: its value is

$$b_{n-1}b_{n-2} \cdots b_1\ b_0 = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

3.1.3有符号二进制整数

The 2's complement representation is widely used to denote signed integers. The value of an n-bit 2's complement binary number is

$$b_{n-1}b_{n-2} \cdots b_1\ b_0 = \boxed{-b_{n-1} \times 2^{n-1}} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

1. 已知x的二进制求-x的二进制方法：将x按位取反再加1

$$-x = barx + 1$$

除了常见的无符号数、有符号数原码、有符号数补码外还有有符号数偏置码——即无符号数-偏差值$(2^{n-1} - 1)$

Table 3.2 The values of 4-bit binary numbers at different representations

Binary number	Unsigned	2's complement	Sign-absolute	Biased
0000	0	0	+0	−7
0001	1	+1	+1	−6
0010	2	+2	+2	−5
0011	3	+3	+3	−4
0100	4	+4	+4	−3
0101	5	+5	+5	−2
0110	6	+6	+6	−1
0111	7	+7	+7	0
1000	8	−8	−0	+1
1001	9	−7	−1	+2
1010	10	−6	−2	+3
1011	11	−5	−3	+4
1100	12	−4	−4	+5
1101	13	−3	−5	+6
1110	14	−2	−6	+7
1111	15	−1	−7	+8

3.2二进制加法和减法

3.2.1 行波进位加法器和减法器设计

半加器——不加低位的进位

The circuit that adds two 1-bit numbers is called a half adder. The half adder performs 0 + 0 = 00, 0 + 1 = 01, 1 + 0 = 01, and 1 + 1 = 10. The left bit of the result is called a carry out and the right bit is called a sum.

全加器——加低位的进位

全加器不仅仅加两个1bit的数字，而且也加低位的进位。1bit全加器有**3个输入a、b、ci，两个输出s、co**

> *A full adder adds not only the two 1-bit numbers but also a carry in which is the carry out of the next bit to the right.The three inputs of a full adder are a, b,andci (carry in), and the outputs are co (carry out) and s (sum).*

全加器的逻辑表达式是：

$$s = barabarbc_i + baraboverlinec\_i + abarboverlinec\_i + abc_i = a \oplus b \oplus c_i$$

$$co = ab + ac_i + bc_i = ab + (a + b)c_i$$

根据表达式实现电路是：

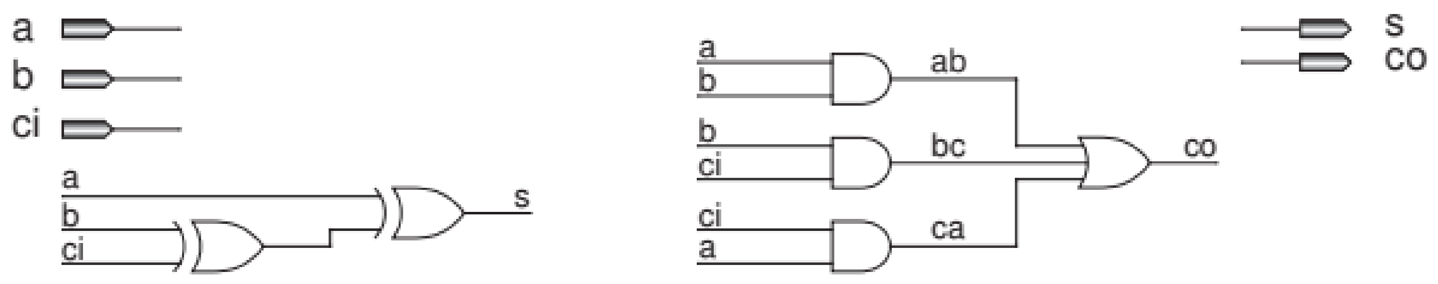


Figure 3.5 Schematic diagram of full adder (using XOR gates)

逻辑门实现

```verilog
module fullAdder_designBylogic(
    input a,b,ci,
    output co,s
);
    wire temp;
    xor xor1(temp,b,ci);
    xor xor2(s,a,temp);//s=a^b^ci

    wire a_b1,a_b2,aob;
    and and1(a_b1,a,b);//ab
    or or1(aob,a,b);//a+b
    and and2(a_b2,aob,ci);//(a+b)ci
    or or2(co,a_b1,a_b2);
endmodule
```

Verilog

数据流风格实现

```verilog
module fullAdder_designBydata(
    input a,b,ci,
    output co,s
);
    assign s=a^b^ci;
    assign co=a&b|(a|b)&ci;
endmodule
```
Verilog

行为风格实现

```
module fullAdder_designBybehav(
    input a,b,ci,
    output co,s
);
    assign {co,s}=a+b+ci;
endmodule
```
Verilog

```
`timescale 1ns / 1ps

module fullAdder_test();

    reg a,b,ci;
    wire co1,s1,co2,s2,co3,s3;

    reg [2:0]temp;
    initial begin
        temp=3'b0;
        repeat (8) begin
            {a,b,ci}=temp;
            #5;
            temp=temp+1;
        end
        $finish;
    end
    fullAdder_designBylogic  fullAdder_designBylogic_inst (
```

```verilog
        .a(a),
        .b(b),
        .ci(ci),
        .co(co1),
        .s(s1)
    );
    fullAdder_designBydata  fullAdder_designBydata_inst (
        .a(a),
        .b(b),
        .ci(ci),
        .co(co2),
        .s(s2)
    );
    fullAdder_designBybehav  fullAdder_designBybehav_inst (
        .a(a),
        .b(b),
        .ci(ci),
        .co(co3),
        .s(s3)
    );
endmodule
```

Verilog

行波进位加法器

行波进位加法器是用一位全加器逐级连接构成的，**除第一个和最后一个全加器外，每个全加器的co接前面一个全加器的ci，第一个的co输出，最后一个的ci接cin**

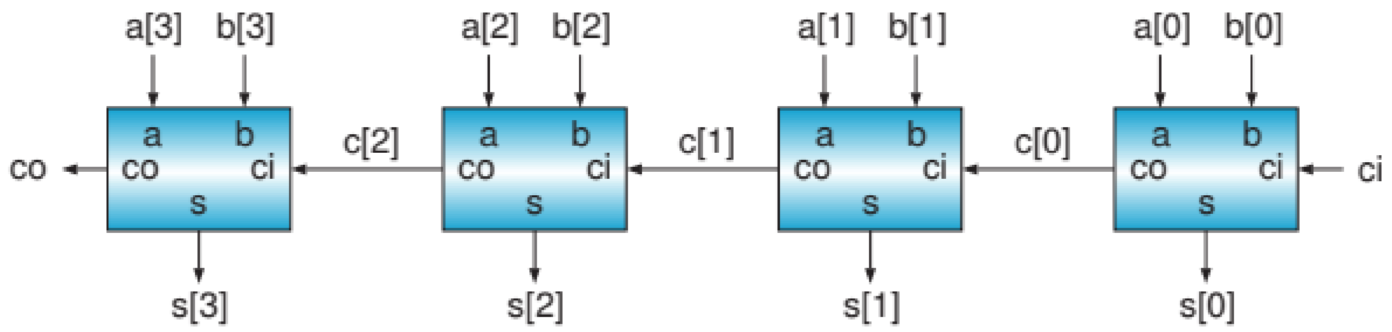


Figure 3.7 Schematic diagram of 4-bit ripple adder

```verilog
`timescale 1ns / 1ps

module ripper_adder4_design(
    input [3:0]a,b,
    input ci,sign,

    output [3:0]s,
    output co,
    //溢出判断
    output overflow
);
    wire [3:0]c;
    fullAdder_designBybehav a1(a[0],b[0],ci,c[0],s[0]);
    fullAdder_designBybehav a2(a[1],b[1],c[0],c[1],s[1]);
    fullAdder_designBybehav a3(a[2],b[2],c[1],c[2],s[2]);
    fullAdder_designBybehav a4(a[3],b[3],c[2],c[3],s[3]);
    assign co=c[3];
    //有符号溢出: 符号位进位和数值位进位异或
    //无符号溢出: 有进位输出则溢出
    assign overflow=sign?(c[3]^c[2]):c[3];
endmodule
```

Verilog

```verilog
`timescale 1ns / 1ps

module ripper_adder4_test();
    reg [3:0]a,b;
    reg ci,sign;

    wire [3:0]s;
    wire co,overflow;

    initial begin
        a=4'b0010;b=4'b0011;ci=0;sign=0;
        #5; a=4'b0101;b=4'b1011;ci=1;sign=1;
        #5; a=4'b0110;b=4'b0110;ci=0;sign=1;
    end

    ripper_adder4_design  ripper_adder4_design_inst (
        .a(a),
        .b(b),
        .ci(ci),
        .sign(sign),
        .s(s),
        .co(co),
        .overflow(overflow)
    );
endmodule
```
Verilog

行波进位加法器既可以实现无符号数的加法也可以实现有符号数补码的加法

注意溢出的判断：有符号数的溢出，判断符号位进位和数值位进位的异或值；无符号数加法的溢出，有进位输出则溢出

有符号数只存在溢出概念，而无符号数在执行减法时对应的是借位，当不够减时就会借位

减法器实现

根据之前提到的已知x的二进制求-x的二进制方法：将x按位取反再加1，那么$a - b = a + (-b) = a + barb + 1$

因此$a - b - c_i = a + (-b) + (-c_i) = a + barb + barc_i + 1 + 1 = a + barb + barc_i$，其中的“1+1”对于位的布尔运算来说为0

因此**可以对加法器添加一个sub控制输入，当sub为1时，执行**$= a + barb + overlinec\_i$**否则执行**$a + b + c_i$

从整体的角度来看a-b-ci,在级联过程中，后一个的进位输出到前一个的进位输入并不需要与sub异或

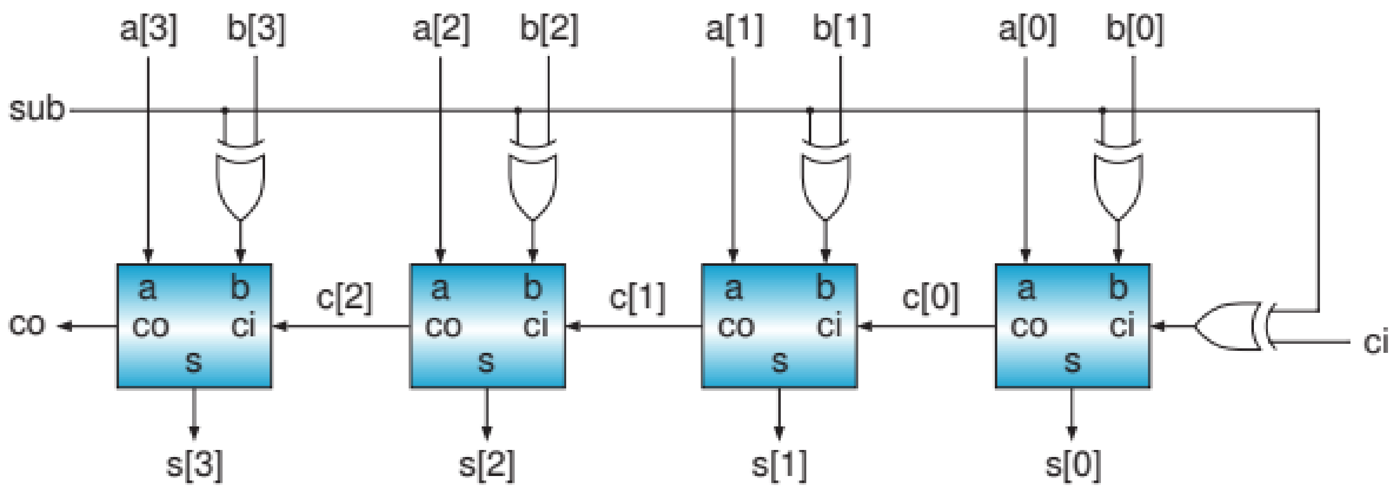


Figure 3.8 Schematic diagram of a 4-bit adder/subtracter

```
`timescale 1ns / 1ps

module addsub4_design(
    input [3:0]a,b,
    input ci,sign,sub,

    output [3:0]s,
    output co,
    //溢出判断
    output overflow,
    output cf_take
);
    wire [3:0]c;

    //从整体的角度来看a-b-ci,在级联过程中，后一个的进位输出到前一个的进位输入并不需要与sub异或
    fullAdder_designBybehav a1(a[0],b[0]^sub,ci^sub,c[0],s[0]);
```

```verilog
    fullAdder_designBybehav a2(a[1],b[1]^sub,c[0],c[1],s[1]);
    fullAdder_designBybehav a3(a[2],b[2]^sub,c[1],c[2],s[2]);
    fullAdder_designBybehav a4(a[3],b[3]^sub,c[2],c[3],s[3]);

    assign co=c[3];
    //有符号只有溢出概念：符号位进位和数值位进位异或
    //无符号：加法时有进位输出则溢出，减法时小于则借位
    assign overflow=sign?(c[3]^c[2]):~sub&c[3];
    assign cf_take=~sign&sub&(a<b);//无符号数减法的借位
endmodule
```

Verilog

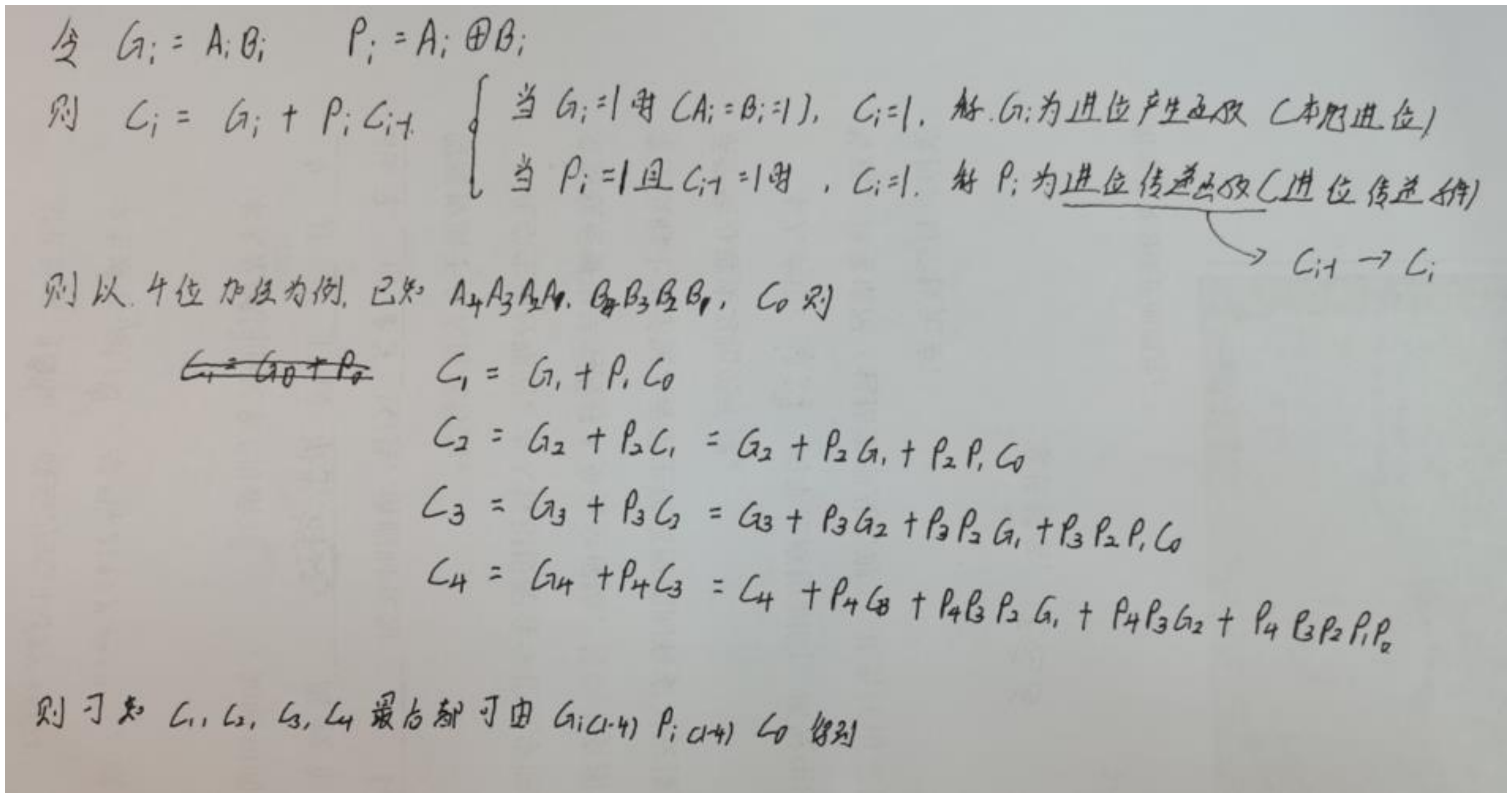
3.2.2 超前进位加法器

超前进位全加器CLA

行波进位加法器是缓慢的，因为**进位需要传递**

> *The ripple adder is area-efficient but is slow because the carry travels through the chain of the full adders*

CLA理想状态下可以实现进位的同时产生从而加快计算。推导过程如下：



令 $G_i = A_i B_i$ $P_i = A_i \oplus B_i$

则 $C_i = G_i + P_i C_{i-1}$
$\begin{cases} \text{当 } G_i=1 \text{ 时 } (A_i=B_i=1),\ C_i=1. \text{ 称 } G_i \text{ 为进位产生函数(本位进位)} \\ \text{当 } P_i=1 \text{ 且 } C_{i-1}=1 \text{ 时},\ C_i=1. \text{ 称 } P_i \text{ 为} \underline{\text{进位传递函数}} \text{(进位传递条件)} \end{cases}$

$C_{i-1} \rightarrow C_i$

则以4位加法为例. 已知 $A_4A_3A_2A_1$. $B_4B_3B_2B_1$, C_0 则

~~$C_1 = G_0 + P_0$~~

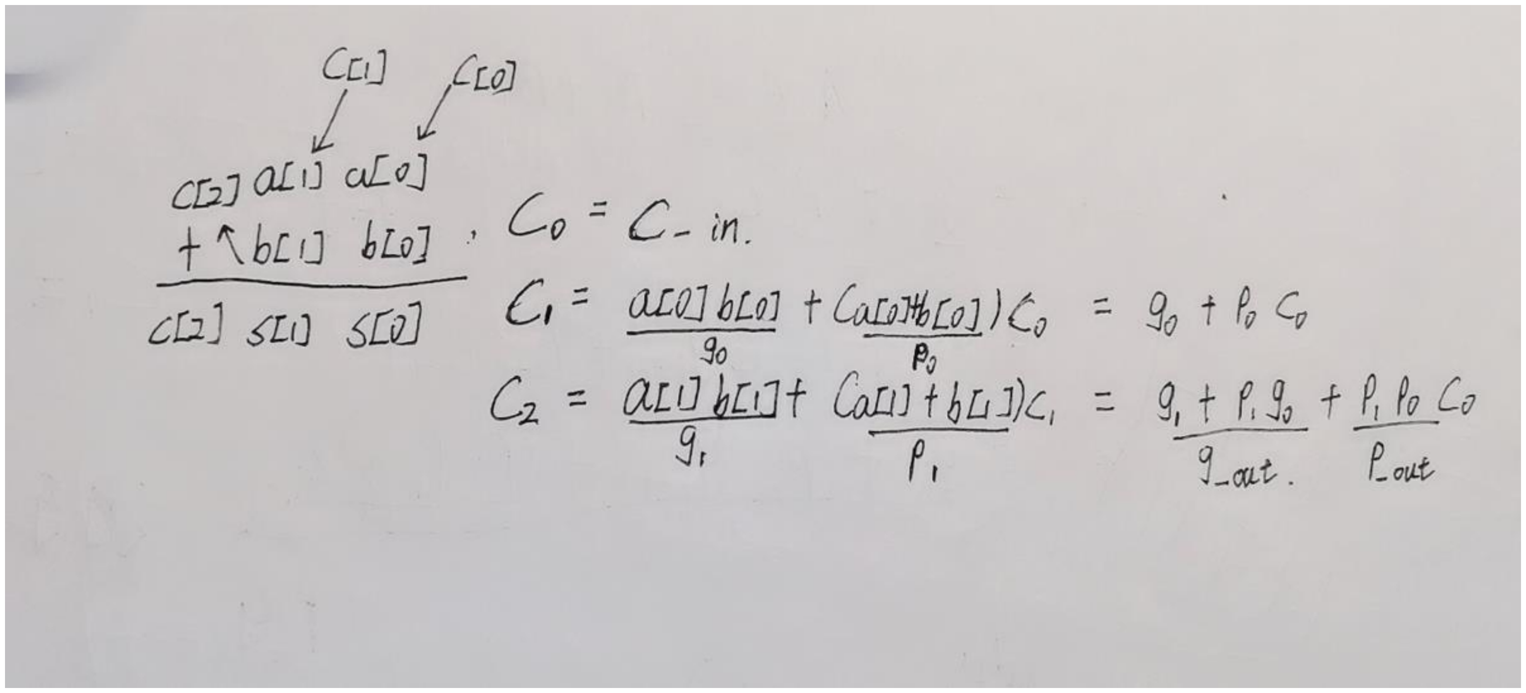
$$C_1 = G_1 + P_1 C_0$$
$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$
$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$
$$C_4 = G_4 + P_4 C_3 = C_4 + P_4 G_3 + P_4 P_3 P_2 G_1 + P_4 P_3 G_2 + P_4 P_3 P_2 P_1 P_0$$

则可知 C_1, C_2, C_3, C_4 最后都可由 $G_{i(1\text{-}4)}$ $P_{i(1\text{-}4)}$ C_0 得到

这里有本位进位（进位产生器g）和进位传递条件（进位传递者p）

两位CLA

输入a[1:0],b[1:0],c_in；输出s[1:0],c_out



C[1] C[0]

C[2] a[1] a[0]
+ b[1] b[0]
C[2] S[1] S[0]

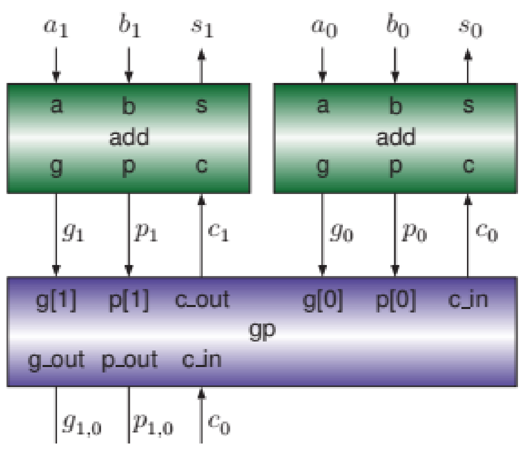
$C_0 = C\_in.$

$$C_1 = \underbrace{a[0]b[0]}_{g_0} + (\underbrace{a[0]+b[0]}_{p_0})C_0 = g_0 + p_0 C_0$$

$$C_2 = \underbrace{a[1]b[1]}_{g_1} + (\underbrace{a[1]+b[1]}_{p_1})C_1 = \underbrace{g_1 + p_1 g_0}_{g\_out.} + \underbrace{p_1 p_0}_{p\_out} C_0$$

首先需要一个部件去产生结果和g_i、p_i；然后p_i,g_i以及低位的c_in共同输入到一个部件中求高位进位，以及生成输出进位的p、g

如下图所示：



产生位结果和g、p的add部件

```verilog
module add (
    input a,b,c,
    output s,g,p
);
    assign s=a^b^c;
    assign g=a&b;
    assign p=a|b;
endmodule
```
Verilog

产生高位进位和进位输出的g、p的gp部件

```verilog
module gp(
    input [1:0]g,p,
    input c_in,
    output g_out,p_out,
    output c_out
);
    assign p_out=p[1]&p[0];
    assign g_out=g[1]|p[1]&g[0];
    assign c_out=g[0]|p[0]&c_in;
endmodule
```

Verilog

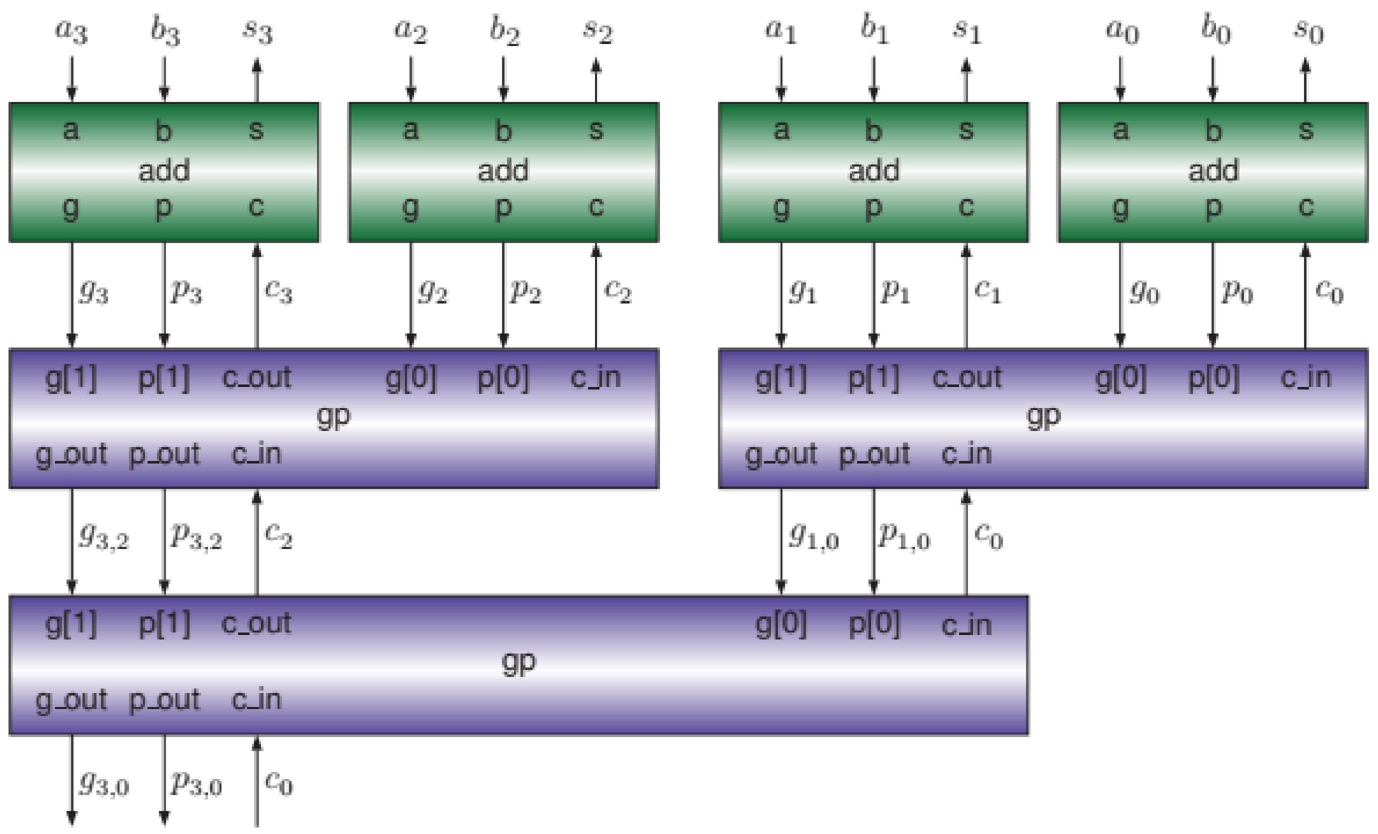
两位CLA

```
module cla_2(//用于级联的cla2
    input [1:0]a,b,
    input c_in,

    output [1:0]s,
    output g_out,
    output p_out,
    output data_co//判断溢出的数值最高位进位
);
    wire [1:0]g,p;
    wire c1;
    gp gp_init(g,p,c_in,g_out,p_out,c1);
    add add1(a[0],b[0],c_in,s[0],g[0],p[0]);//低位add
    add add2(a[1],b[1],c1,s[1],g[1],p[1]);//高位add

    assign data_co=c1;
endmodule
```

Verilog

4位CLA

4位CLA可以用两个两位CLA+1个gp集成，gp用来生成进位输出的g、p以及生成左侧CLA的低位进位输入



a_3 b_3 s_3 a_2 b_2 s_2 a_1 b_1 s_1 a_0 b_0 s_0
a b s add g p c
g_3 p_3 c_3 g_2 p_2 c_2 g_1 p_1 c_1 g_0 p_0 c_0
g[1] p[1] c_out g[0] p[0] c_in gp g_out p_out c_in
$g_{3,2}$ $p_{3,2}$ c_2 $g_{1,0}$ $p_{1,0}$ c_0
$g_{3,0}$ $p_{3,0}$ c_0

集成代码

```
module cla_4(
    input [3:0]a,b,
    input c_in,
    output [3:0]s,
    output g_out,p_out,
    output data_co
);//用于级联的cla4


    wire c1,co1,co2;
    wire [1:0]g,p;
    cla_2 cla_2_inst1(a[1:0],b[1:0],c_in,s[1:0],g[0],p[0],co1);//低位

    cla_2 cla_2_inst2(a[3:2],b[3:2],c1,s[3:2],g[1],p[1],co2);//高位

    gp gp_init(g,p,c_in,g_out,p_out,c1);
    assign data_co=c1;
endmodule
```
Verilog

8位CLA、16位CLA、32位CLA

8位CLA是用两个4位CLA+gp集成

```
module cla_8(
    input [7:0]a,b,
    input c_in,
    output [7:0]s,
    output g_out,p_out,
    output data_co
);//用于级联的cla8


    wire c1,co1,co2;
    wire [1:0]g,p;
    cla_4 cla_4_inst1(a[3:0],b[3:0],c_in,s[3:0],g[0],p[0],co1);//低位

    cla_4 cla_4_inst2(a[7:4],b[7:4],c1,s[7:4],g[1],p[1],co2);//高位

    gp gp_init(g,p,c_in,g_out,p_out,c1);

    assign data_co=c1;
endmodule
```
Verilog

16位CLA是用两个8位CLA+gp集成

```
module cla_16(
    input [15:0]a,b,
    input c_in,
    output [15:0]s,
    output g_out,p_out,
    output data_co
);//用于级联的cla8


    wire c1,co1,co2;
    wire [1:0]g,p;
    cla_8 cla_8_inst1(a[7:0],b[7:0],c_in,s[7:0],g[0],p[0],co1);//低位

    cla_8 cla_8_inst2(a[15:8],b[15:8],c1,s[15:8],g[1],p[1],co2);//高位

    gp gp_init(g,p,c_in,g_out,p_out,c1);
    assign data_co=c1;
endmodule
```
Verilog

32位CLA是用两个16位CLA+gp集成

```
module cla_32(
    input [32:0]a,b,
    input c_in,
    output [32:0]s,
    output g_out,p_out,
    output data_co
);//用于级联的cla8


    wire c1,co1,co2;
    wire [1:0]g,p;
    cla_16 cla_16_inst1(a[15:0],b[15:0],c_in,s[15:0],g[0],p[0],co1);//低位

    cla_16 cla_16_inst2(a[31:16],b[31:16],c1,s[31:16],g[1],p[1],co2);//高位

    gp gp_init(g,p,c_in,g_out,p_out,c1);
    assign data_co=c1;
endmodule
```
Verilog

32位CLA运算器

```verilog
`timescale 1ns / 1ps

module cla32AddSub_design(
    input [31:0]a,b,
    input c_in,sign,sub,

    output [31:0]s,
    output c_out,
    output overflow,//有符号数溢出，无符号数加法溢出
    output c_take//无符号数减法借位
);
    wire p_out,g_out,data_co;
    //以4位为例，sub^b是b和0001异或，但是b应该和1111异或
    cla_32 cla_32_init(a,{32{sub}}^b,sub^c_in,s,g_out,p_out,data_co);//加法时sub=0，减法时sub=1
    assign c_out=g_out+p_out&c_in;

    assign overflow=sign?c_out^data_co:(~sub&c_out);
    assign c_take=sub&~sign&(a<b);
endmodule
```
Verilog

```verilog
`timescale 1ns / 1ps

module cla32_alu_test();
    reg [31:0]a,b;
    reg ci,sign,sub;

    wire [31:0]s;
    wire c_out;
    wire overflow;//有符号数溢出，无符号数加法溢出
    wire c_take;//无符号数减法借位

    initial begin
        sign=0;sub=0;a=32'd5;b=32'd9;ci=0;//5+9+0
        #5; sign=0;sub=0;a=32'd7;b=32'd11;ci=1;//7+11+1
        #5; sign=0;sub=1;a=32'd8;b=32'd3;ci=0;//8-3-0
        #5; sign=0;sub=1;a=32'd9;b=32'd6;ci=1;//9-6-1
        #5; sign=0;sub=1;a=32'd9;b=32'd11;ci=0;//9-11-0

        #5; sign=1;sub=0;a=-32'd5;b=-32'd2;ci=0;//-5+(-2)+0
        #5; sign=1;sub=0;a=-32'd8;b=32'd7;ci=1;//-8+7+1

        #5; sign=1;sub=1;a=32'd5;b=-32'd2;ci=1;//5-(-2)-1
        #5; sign=1;sub=1;a=-32'd4;b=-32'd3;ci=1;//-4-(-3)-1
        #5; $finish;
    end

    cla32AddSub_design  cla32AddSub_design_inst (
        .a(a),
        .b(b),
        .c_in(ci),
        .sign(sign),
        .sub(sub),
        .s(s),
        .c_out(c_out),
        .overflow(overflow),
        .c_take(c_take)
    );
endmodule
```
Verilog

3.3二进制乘法

3.3.1无符号数乘法设计

和十进制乘法类似，无符号数乘法可以通过移位（被乘数左移、乘数右移）和加法来实现

$$
\begin{array}{cccccccccl}
 & & & & 1 & 1 & 1 & 0 & & (14) \\
 & & & \times & 1 & 0 & 1 & 0 & & (10) \\
\cline{4-8}
 & & & & 0 & 0 & 0 & 0 & & \\
 & & & 1 & 1 & 1 & 0 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
+ & 1 & 1 & 1 & 0 & & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & & (140)
\end{array}
$$

$$0 \times 1110 + 1 \times 11100 + 0 \times 111000 + 1 \times 1110000$$

C语言实现的无符号数乘法

```c
unsigned int mul16 (unsigned int x, unsigned int y) {
  unsigned int a, b, c;
  unsigned int i; // counter
  a = x; // multiplicand
  b = y; // multiplier
  c = 0; // product
  for (i = 0; i < 16; i++) { // for 16 bits
    if ((b & 1) == 1) { // LSB of b is 1
        c += a; // c = c + a
    }
    a = a << 1; // shift a 1-bit left
    b = b >> 1; // shift b 1-bit right
  }
  return(c); // return product
}
```

Verilog

Verilog实现的无符号数乘法

```
`timescale 1ns / 1ps

module nosigned_mul #(parameter WIDTH_A=4,WIDTH_B=4)(
    input [WIDTH_A-1:0]a,
    input [WIDTH_B-1:0]b,
    input enable,
    input clk,

    output reg[WIDTH_A+WIDTH_B-1:0]res,
    output reg ready
);
    reg [WIDTH_A+WIDTH_B-1:0]temp_A,res_temp;
    reg [WIDTH_B-1:0]temp_B;
    integer i;
    always @(posedge clk) begin//采用时钟控制，每个上升沿计算一次；也便于流水
        if (enable) begin
            temp_A<=a;
            temp_B<=b;
            res_temp<=0;
            i<=0;
            res<=0;
            ready<=0;
        end else begin
            res_temp<=temp_B[0]?res_temp+temp_A:res_temp;
            temp_A<=temp_A<<1;
            temp_B<=temp_B>>1;
            i=i+1;
            if (i==WIDTH_B) begin
                ready<=1;
                res<=res_temp;
            end
        end
    end
endmodule
```
Verilog

3.3.2有符号数乘法设计

有符号数的乘法设计最简单的是可以先记录符号，转换为无符号数$-x = barx + 1$，然后按照无符号数的乘法计算，最后再对结果进行符号的处理

此外也可以根据乘法实例，直接对有符号数进行相乘，下面进行介绍

推导过程

以两个8位采用补码表示的有符号数相乘为例，根据行内引用A_8, B_8可以表示为下面的形式

则$A_8 \times B_8$可以表示如下，$A_7 \times B_7$即为无符号数的乘法

$$\begin{aligned} Z_{16} &= A_8 \times B_8 \\ &= (-a_7 \times 2^7 + A_7) \times (-b_7 \times 2^7 + B_7) \\ &= a_7 \times b_7 \times 2^{14} + (-a_7 \times B_7) \times 2^7 + (-b_7 \times A_7) \times 2^7 + A_7 \times B_7 \end{aligned}$$

如上式，最后的结果可以表示为4个因式的和，其中第一个因式和最后一个因式均是正值，而第二、第三则是负数

第一个因式，直接$a_7 \times b_7$左移14位即可

第四个因式，是一个无符号数相乘

第二、三个因式可以先表示$a_7 \times B_7$、$b_7 \times A_7$再取反

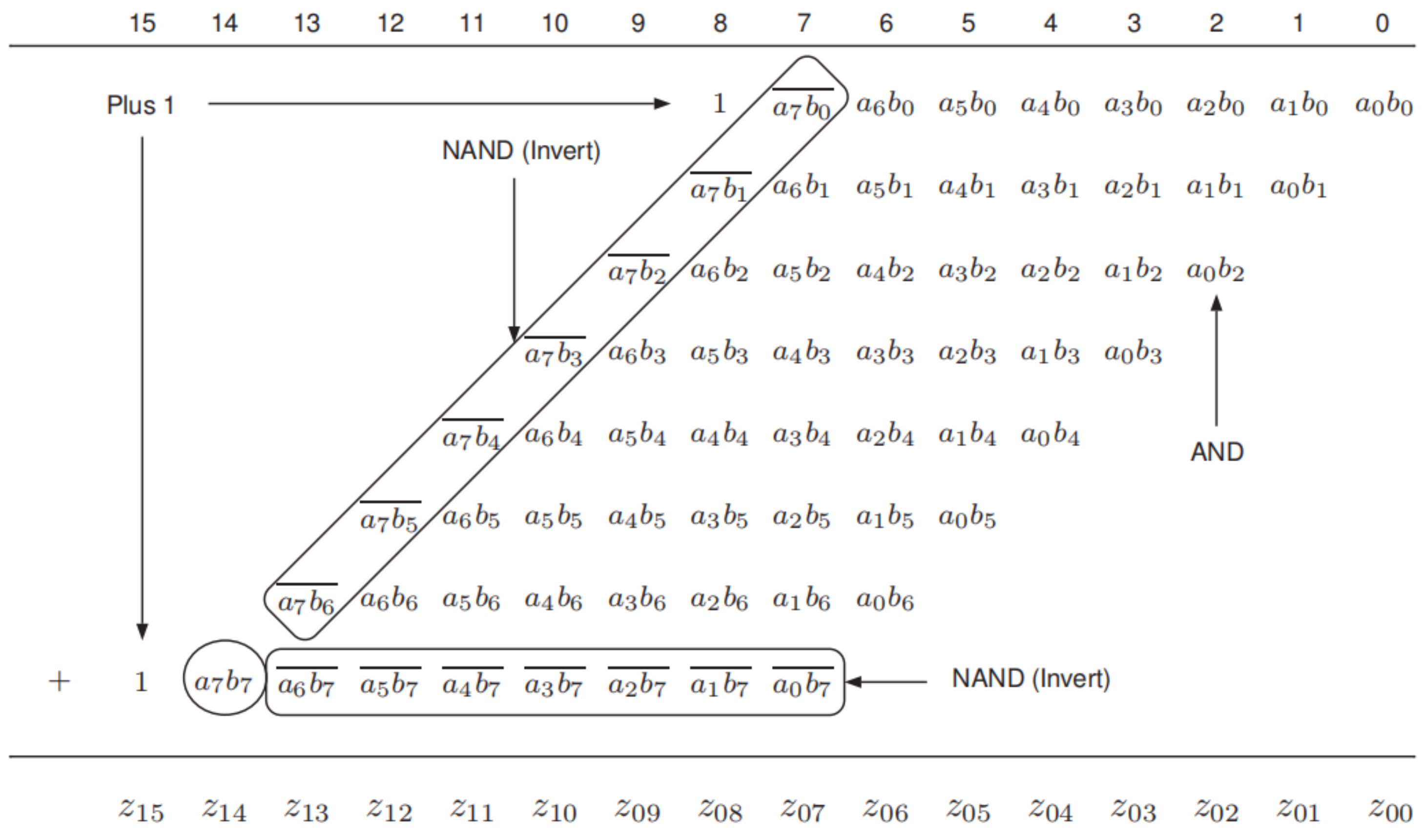
$$\begin{array}{lllllllll} a_7 \times B_7 = 0 & 0 & a_7b_6 & a_7b_5 & a_7b_4 & a_7b_3 & a_7b_2 & a_7b_1 & a_7b_0 \\ b_7 \times A_7 = 0 & 0 & a_6b_7 & a_5b_7 & a_4b_7 & a_3b_7 & a_2b_7 & a_1b_7 & a_0b_7 \end{array}$$

	15	14	13	12	11	10	9	8	7
	1	1	$\overline{a_7b_6}$	$\overline{a_7b_5}$	$\overline{a_7b_4}$	$\overline{a_7b_3}$	$\overline{a_7b_2}$	$\overline{a_7b_1}$	$\overline{a_7b_0}$
	0	0	0	0	0	0	0	0	1
	1	1	$\overline{a_6b_7}$	$\overline{a_5b_7}$	$\overline{a_4b_7}$	$\overline{a_3b_7}$	$\overline{a_2b_7}$	$\overline{a_1b_7}$	$\overline{a_0b_7}$
+	0	0	0	0	0	0	0	0	1

最后去掉进位，可以得到

	15	14	13	12	11	10	9	8	7
	0	0	$\overline{a_7b_6}$	$\overline{a_7b_5}$	$\overline{a_7b_4}$	$\overline{a_7b_3}$	$\overline{a_7b_2}$	$\overline{a_7b_1}$	$\overline{a_7b_0}$
	0	0	$\overline{a_6b_7}$	$\overline{a_5b_7}$	$\overline{a_4b_7}$	$\overline{a_3b_7}$	$\overline{a_2b_7}$	$\overline{a_1b_7}$	$\overline{a_0b_7}$
+	1	0	0	0	0	0	0	1	0

则最后的加法式为



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Plus 1
1
NAND (Invert)
AND
NAND (Invert)
+ 1
$\overline{a_7b_0}$ a_6b_0 a_5b_0 a_4b_0 a_3b_0 a_2b_0 a_1b_0 a_0b_0
$\overline{a_7b_1}$ a_6b_1 a_5b_1 a_4b_1 a_3b_1 a_2b_1 a_1b_1 a_0b_1
$\overline{a_7b_2}$ a_6b_2 a_5b_2 a_4b_2 a_3b_2 a_2b_2 a_1b_2 a_0b_2
$\overline{a_7b_3}$ a_6b_3 a_5b_3 a_4b_3 a_3b_3 a_2b_3 a_1b_3 a_0b_3
$\overline{a_7b_4}$ a_6b_4 a_5b_4 a_4b_4 a_3b_4 a_2b_4 a_1b_4 a_0b_4
$\overline{a_7b_5}$ a_6b_5 a_5b_5 a_4b_5 a_3b_5 a_2b_5 a_1b_5 a_0b_5
$\overline{a_7b_6}$ a_6b_6 a_5b_6 a_4b_6 a_3b_6 a_2b_6 a_1b_6 a_0b_6
a_7b_7 $\overline{a_6b_7}$ $\overline{a_5b_7}$ $\overline{a_4b_7}$ $\overline{a_3b_7}$ $\overline{a_2b_7}$ $\overline{a_1b_7}$ $\overline{a_0b_7}$
z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_{09} z_{08} z_{07} z_{06} z_{05} z_{04} z_{03} z_{02} z_{01} z_{00}

实现

Use of parentheses in the code ensures performing the additions in parallel

```
`timescale 1ns / 1ps

//实现8*8的有符号乘法
module signed_mul(
    input [7:0]a,b,

    output reg[15:0]res
);
    reg [7:0]ab[7:0];
    integer i,j;
    always @(*) begin
        for (i=0;i<7;i=i+1) begin
            for (j=0;j<7;j=j+1) begin
                ab[j][i]=a[i]&b[j];
            end
        end
        for (i = 0; i<7; i=i+1) begin
            ab[7][i]=~(a[i]&b[7]);
        end
        for (j=0;j<7;j=j+1)begin
            ab[j][7]=~(a[7]&b[j]);
        end
        ab[7][7]=a[7]&b[7];
        res={8'b1,ab[0][7:0]}+
            {7'b0,ab[1][7:0],1'b0}+
            {6'b0,ab[2][7:0],2'b0}+
            {4'b0,ab[3][7:0],3'b0}+
            {3'b0,ab[4][7:0],4'b0}+
            {2'b0,ab[5][7:0],5'b0}+
            {1'b0,ab[6][7:0],6'b0}+
            {1'b1,ab[7][7:0],7'b0};
    end
endmodule
```
Verilog

```
`timescale 1ns / 1ps

module signed_mul_test();
    reg [7:0]a,b;
    wire[15:0]res;

    initial begin
        a=8'd1;b=8'd7;
        #5;a=8'd2;b=8'd10;
        #5;a=-8'd5;b=8'd12;
        #5;a=-8'd11;b=-8'd13;
        #5;a=8'd13;b=-8'd14;
        #5;a=8'd17;b=8'd11;
        #5;$finish;
    end
    signed_mul  signed_mul_inst (
        .a(a),
        .b(b),
        .res(res)
    );
endmodule
```
Verilog

3.3.3 Wallace Tree

进位保存加法器CSA

首先来介绍进位保存加法器CSA

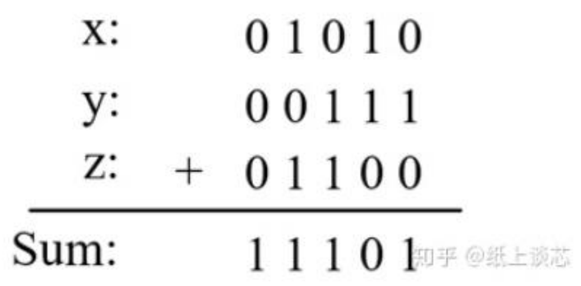
https://zhuanlan.zhihu.com/p/102387648书签：【HDL系列】进位保存加法器原理与设计

CSA思想

进位保存加法器CSA在执行多个数加法时具有极小的进位传播延迟。它的基本思想是将**3个加数的和**（不考虑进位端）减少为**2个加数的和**，将**进位c和和s分别计算保存**，并且每比特可以独立计算c和s，所以速度极快。

下面以（10+7+12=29）为例来讲解进位保存加法器的操作：

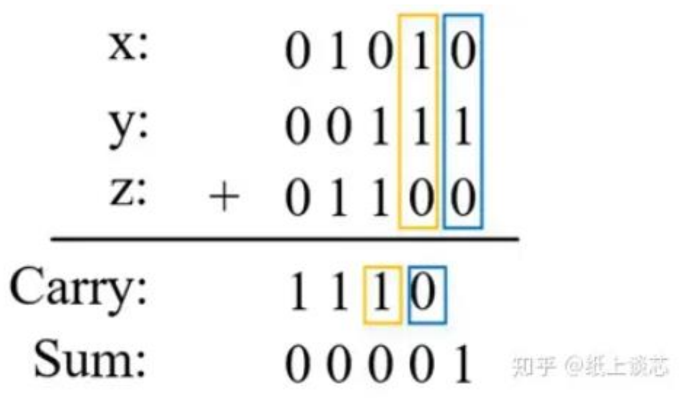
1. 正常的执行加法



竖式计算

先对10和7相加，再将其结果加12得到最终的结果

2. CSA执行

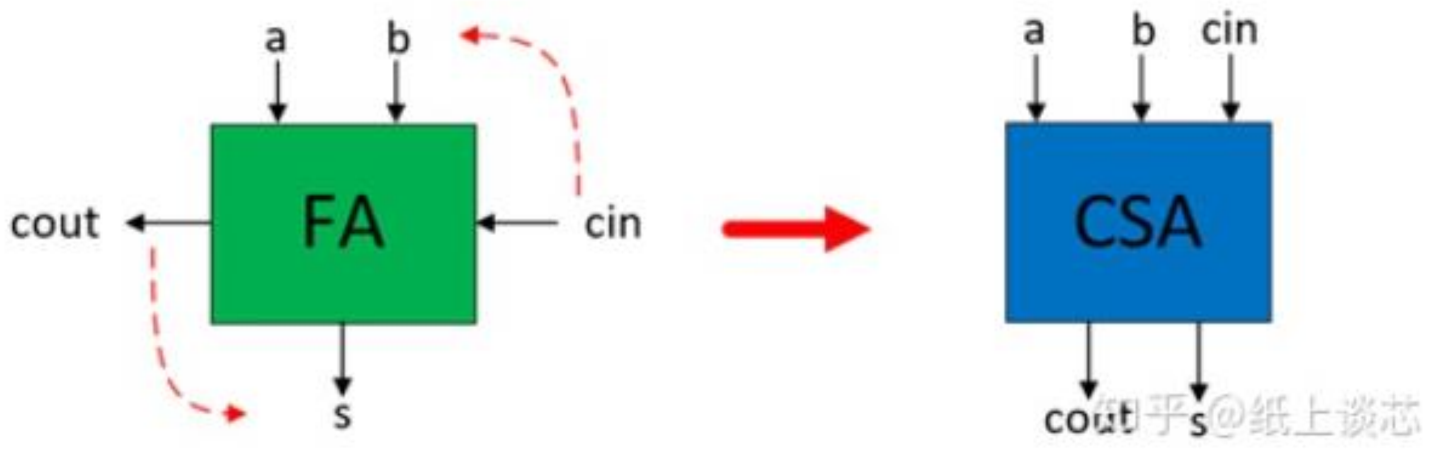


CSA进位Carry与和Sum竖式计算方法

对三个加数同时进行处理，分别计算产生sum和co，$sum = (ai + bi + zi)mod2$**，**$co = (ai + bi + ci)/2$**。之后再对sum和co再利用csa求和，注意co要左移一位**

Compressor

Compressors是进位保存加法器的一种，它就实现了上述的CSA的行为，其真值表和全加器一致，故对全加器FA进行一些变换即得到进位保存器CSA



设计代码如下：

```verilog
module compressors(
    input a,b,z,
    output co,
    output s
);
    assign {co,s}=a+b+z;
endmodule
```

Verilog

CSA实现10+7+12

```verilog
`timescale 1ns / 1ps

module csa4_design(
    input [3:0]a,b,z,
    output [3:0]s,
    output co
);
    wire [4:0]c_temp;
    wire [4:0]s_temp;
    assign c_temp[0]=0;
    assign s_temp[4]=0;
    compressors c1_1(a[0],b[0],z[0],c_temp[1],s_temp[0]);
    compressors c1_2(a[1],b[1],z[1],c_temp[2],s_temp[1]);
    compressors c1_3(a[2],b[2],z[2],c_temp[3],s_temp[2]);
    compressors c1_4(a[3],b[3],z[3],c_temp[4],s_temp[3]);

    wire [3:0]co_temp;
    compressors c2_1(s_temp[0],c_temp[0],0,co_temp[0],s[0]);
    compressors c2_2(s_temp[1],c_temp[1],co_temp[0],co_temp[1],s[1]);
    compressors c2_3(s_temp[2],c_temp[2],co_temp[1],co_temp[2],s[2]);
    compressors c2_4(s_temp[3],c_temp[3],co_temp[2],co_temp[3],s[3]);

    wire p;
    compressors c2_5(s_temp[4],c_temp[4],co_temp[3],p,co);
endmodule
```
Verilog

```verilog
`timescale 1ns / 1ps

module csa4_test();
    reg [3:0]a,b,z;
    wire [3:0]s;
    wire co;

    initial begin
        a=4'd10;b=4'd7;z=4'd12;
        #5;a=4'd7;b=4'd5;z=4'd2;
        #5;$finish;
    end
    csa4_design  csa4_design_inst (
        .a(a),
        .b(b),
        .z(z),
        .s(s),
        .co(co)
    );
endmodule
```
Verilog

Wallac Tree思想

https://zhuanlan.zhihu.com/p/130968045书签：【HDL系列】乘法器(4)——图解Wallace树

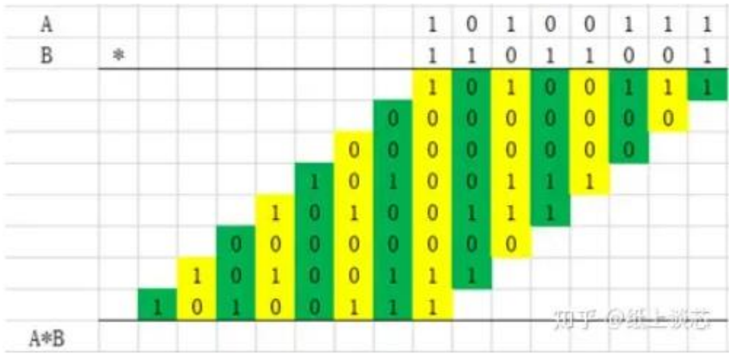
✍Wallac Tree就是利用CSA“全加器3-2压缩的特性”对乘法矩阵中的数进行快速的相加——每列3个加数分为一组，压缩至两个加数，循环往复至最后只有两行使用进位传播加法器相加即可得到最终的结果

示例理解

🎉一个Wallac Tree Multiplier包括三个部分：

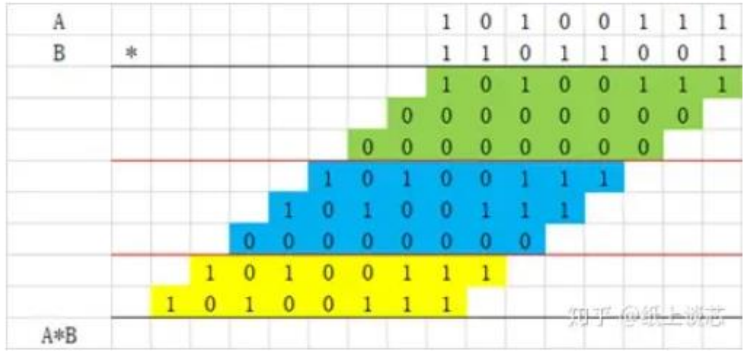
1. 与门（有符号数则还要使用与非门）计算乘数阵列
2. CSA阵列得到求和sum、进位c
3. 进位传播加法器对sum和c相加，形成最终结果

以计算无符号数乘法“10100111*11011001”为例，其乘法阵列如下：



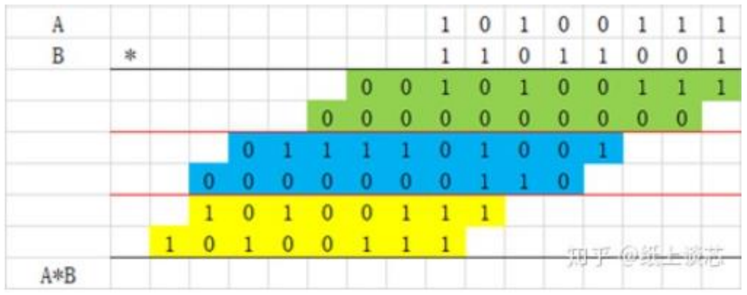
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
									1	0	1	0	0	1	1	1
								0	0	0	0	0	0	0	0	
							0	0	0	0	0	0	0	0		
						1	0	1	0	0	1	1	1			
					1	0	1	0	0	1	1	1				
				0	0	0	0	0	0	0	0					
			1	0	1	0	0	1	1	1						
		1	0	1	0	0	1	1	1							
A*B																

1. 第一级：按每列3个一组进行分组，不足3个的保持到下一级



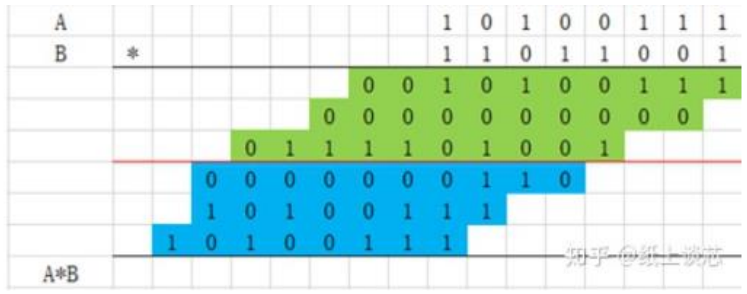
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
									1	0	1	0	0	1	1	1
								0	0	0	0	0	0	0	0	
							0	0	0	0	0	0	0	0		
						1	0	1	0	0	1	1	1			
					1	0	1	0	0	1	1	1				
				0	0	0	0	0	0	0	0					
			1	0	1	0	0	1	1	1						
		1	0	1	0	0	1	1	1							
A*B																

产生结果：



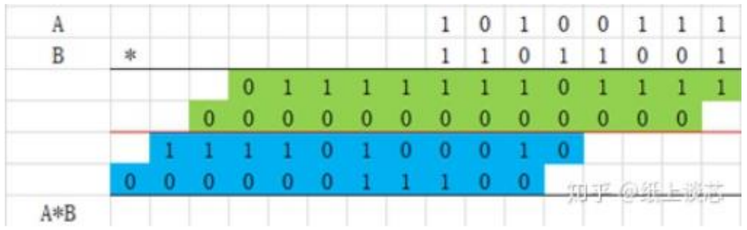
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
							0	0	1	0	1	0	0	1	1	1
						0	0	0	0	0	0	0	0	0	0	
				0	1	1	1	1	0	1	0	0	1			
			0	0	0	0	0	0	0	1	1	0				
			1	0	1	0	0	1	1	1						
		1	0	1	0	0	1	1	1							
A*B																

2. 第二级：再次按照每列3个进行分组



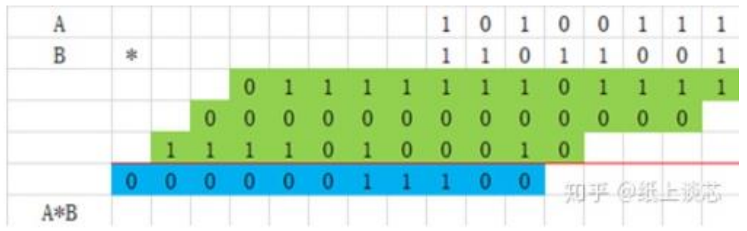
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
							0	0	1	0	1	0	0	1	1	1
						0	0	0	0	0	0	0	0	0	0	
				0	1	1	1	1	0	1	0	0	1			
			0	0	0	0	0	0	0	1	1	0				
			1	0	1	0	0	1	1	1						
		1	0	1	0	0	1	1	1							
A*B																

产生结果：



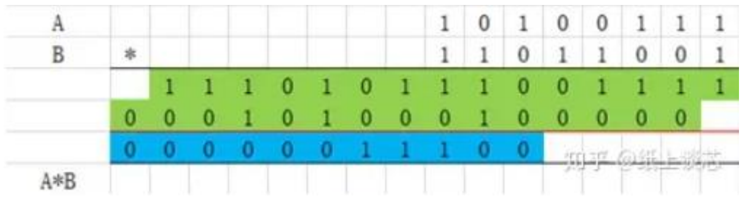
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
				0	1	1	1	1	1	1	1	0	1	1	1	1
			0	0	0	0	0	0	0	0	0	0	0	0	0	
		1	1	1	1	0	1	0	0	0	1	0				
	0	0	0	0	0	0	1	1	1	0	0					
A*B																

3. 第三级：再次分组



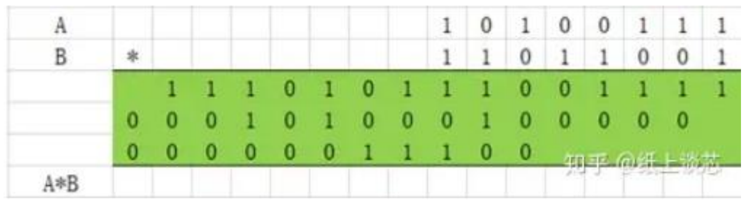
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
				0	1	1	1	1	1	1	1	0	1	1	1	1
			0	0	0	0	0	0	0	0	0	0	0	0	0	
		1	1	1	1	0	1	0	0	0	1	0				
	0	0	0	0	0	0	1	1	1	0	0					
A*B																

产生结果：



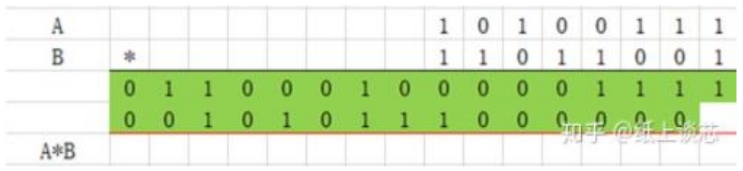
A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
		1	1	1	0	1	0	1	1	1	0	0	1	1	1	1
	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	
	0	0	0	0	0	0	1	1	1	0	0					
A*B																

4. 第四级：再次分组



A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
		1	1	1	0	1	0	1	1	1	0	0	1	1	1	1
	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	
	0	0	0	0	0	0	1	1	1	0	0					
A*B																

产生结果：



A									1	0	1	0	0	1	1	1
B	*								1	1	0	1	1	0	0	1
	0	1	1	0	0	0	1	0	0	0	0	0	1	1	1	1
	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	
A*B																

得到最后的sum和c由传递进位加法器计算最终结果

进位传播加法器的使用也更方便于去流水，采用两段流水线（第一段产生sum和c，第二段产生最终结果）将原来两个时钟周期一个结果变为一个时钟周期一个结果

实现

Wallac树

```
`timescale 1ns / 1ps

module wallac8_8(
    input [7:0]a,b,

    output [15:0]sum,c
);
    reg [15:0]ab[7:0];
    integer i,j;
    //第一部分 与/与非阵列
    always @(*) begin
        for (i = 0; i<8; i=i+1) begin
            for (j = 0; j<8; j=j+1) begin
                ab[i][j]=a[j]&b[i];
            end
            ab[i][15:8]=8'b0;
        end
    end
    //level1
    wire [15:0]sumLevel1_1,coLevel1_1,sumLevel1_2,coLevel1_2;
    csa csa_level1_1(ab[0],ab[1]<<1,ab[2]<<2,sumLevel1_1,coLevel1_1);
    csa csa_level1_2(ab[3]<<3,ab[4]<<4,ab[5]<<5,sumLevel1_2,coLevel1_2);
```

```verilog
    //level2
    wire [15:0]sumLevel2_1,coLevel2_1,sumLevel2_2,coLevel2_2;
    csa csa_level2_1(sumLevel1_1,coLevel1_1<<1,sumLevel1_2,sumLevel2_1,coLevel2_1);
    csa csa_level2_2(coLevel1_2<<1,ab[6]<<6,ab[7]<<7,sumLevel2_2,coLevel2_2);

    //level3
    wire [15:0]sumLevel3_1,coLevel3_1;
    csa csa_level3_1(sumLevel2_1,coLevel2_1<<1,sumLevel2_2,sumLevel3_1,coLevel3_1);

    //level4
    csa csa_level4_1(sumLevel3_1,coLevel3_1<<1,coLevel2_2<<1,sum,c);

endmodule

module csa #(parameter  WIDTH = 8)(
    input [WIDTH*2-1:0]a,b,z,
    output [WIDTH*2-1:0]s,
    output [WIDTH*2-1:0]co
);
    genvar i;
    generate
        for (i = 0; i<WIDTH*2; i=i+1) begin
            assign {co[i],s[i]}=a[i]+b[i]+z[i];
        end
    endgenerate
endmodule
```
Verilog

进位传播加法器

```verilog
`timescale 1ns / 1ps
```

```verilog
module wallacCarryAdder8_8(
    input [7:0]a,b,
    output [15:0]res
);
    wire [15:0]sum,c;
    assign res=sum+(c<<1);
    wallac8_8  wallac8_8_inst (
        a,b,sum,c
    );
endmodule
```
Verilog

测试

```verilog
`timescale 1ns / 1ps

module wallacCA_test();
    reg [7:0]a,b;
    wire [15:0]res;

    initial begin
        a=8'b10100111;b=8'b11011001;
        #5;a=8'd7;b=8'd15;
        #5;a=8'd17;b=8'd24;
        #5;$finish;
    end
    wallacCarryAdder8_8  wallacCarryAdder8_8_inst (
        .a(a),
        .b(b),
        .res(res)
    );
endmodule
```
Verilog

有符号乘法

有符号乘法与无符号数乘法Wallace的区别在于：

1. 在生成乘数阵列中，不仅仅使用了与门，还使用了与非门
2. csa队列中，ab[0][8]=1,ab[7][8]=1

```
`timescale 1ns / 1ps

module wallaceSignedMul(
    input [7:0]a,b,

    output [15:0]sum,c,res
);
    reg [15:0]ab[7:0];
    integer i,j;
    //第一部分 与/与非阵列
    always @(*) begin
        for (i=0;i<7;i=i+1) begin
            ab[i][15:0]=16'b0;
            for (j=0;j<7;j=j+1) begin
                ab[i][j]=a[j]&b[i];
            end
        end
        ab[7]=16'b0;
        for (i = 0; i<7; i=i+1) begin
            ab[7][i]=~(a[i]&b[7]);
        end
        for (j=0;j<7;j=j+1)begin
            ab[j][7]=~(a[7]&b[j]);
        end
        ab[7][7]=a[7]&b[7];
        ab[0][8]=1;
        ab[7][8]=1;
    end

    //level1
    wire [15:0]sumLevel1_1,coLevel1_1,sumLevel1_2,coLevel1_2;
    csa csa_level1_1(ab[0],ab[1]<<1,ab[2]<<2,sumLevel1_1,coLevel1_1);
    csa csa_level1_2(ab[3]<<3,ab[4]<<4,ab[5]<<5,sumLevel1_2,coLevel1_2);

    //level2
    wire [15:0]sumLevel2_1,coLevel2_1,sumLevel2_2,coLevel2_2;
    csa csa_level2_1(sumLevel1_1,coLevel1_1<<1,sumLevel1_2,sumLevel2_1,coLevel2_1);
    csa csa_level2_2(coLevel1_2<<1,ab[6]<<6,ab[7]<<7,sumLevel2_2,coLevel2_2);

    //level3
    wire [15:0]sumLevel3_1,coLevel3_1;
    csa csa_level3_1(sumLevel2_1,coLevel2_1<<1,sumLevel2_2,sumLevel3_1,coLevel3_1);

    //level4
    csa csa_level4_1(sumLevel3_1,coLevel3_1<<1,coLevel2_2<<1,sum,c);
    assign res=sum+(c<<1);
endmodule
```
Verilog

3.4二进制除法

3.4.1恢复余数算法

给定被除数a和除数b，恢复除数算法计算商q和余数r（$a = b \times q + r, r < b$）是**通过从剩余余数（初始化为a的MSB最高有效位）中减去除数b**

如果差值为正数，则设置q的当前位为1；如果差值是负数，则恢复剩余余数

然后左移剩余余数和被除数a一位，再重复计算直至a的所有位均已被移出

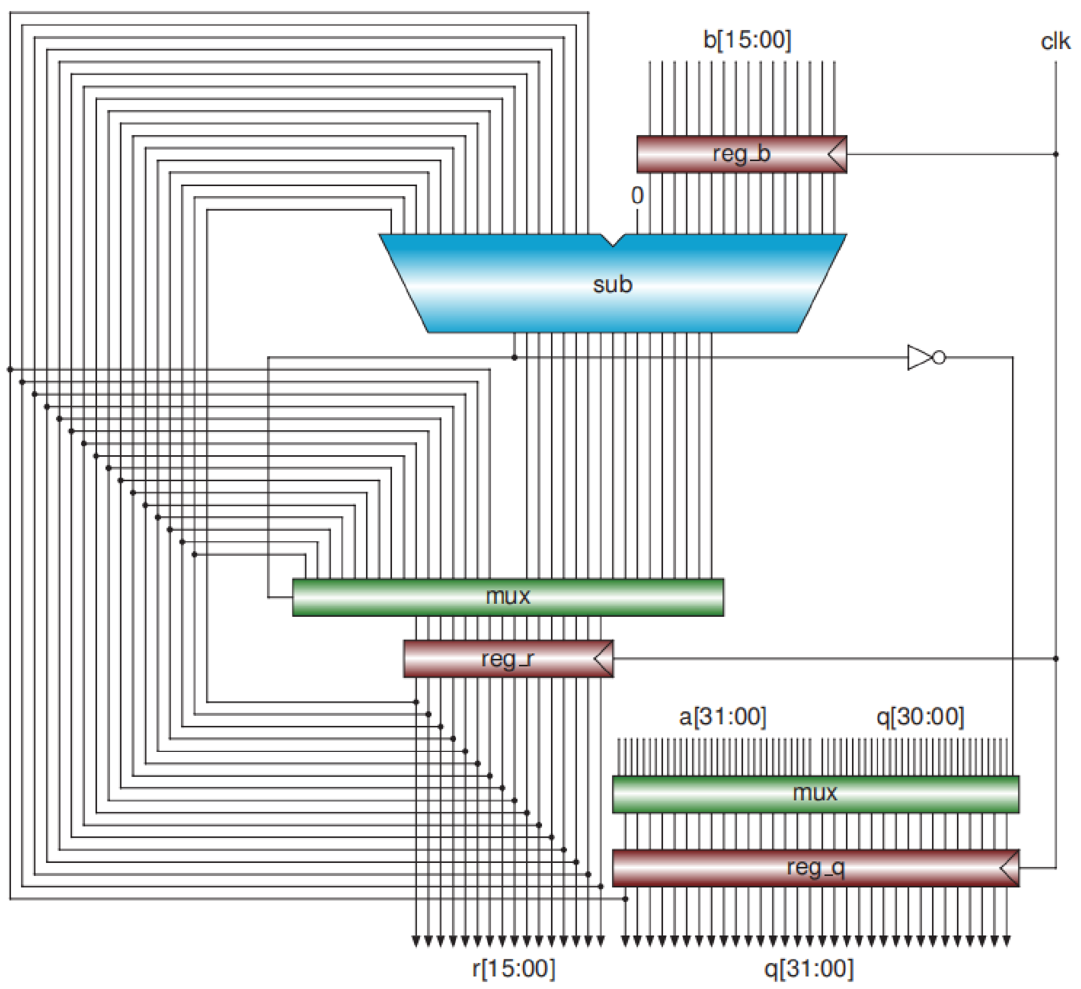


Figure 3.20 Schematic diagram of a restoring divider

reg_r、reg_b、reg_q分别用来存储剩余余数r、除数b和商q(初始值为a)

reg_q上的多路选择器的选择信号是start，是选择初值a还是选择左移结果（左移的LSB是填充减法器减法结果的MSB符号位）

reg_r上的多路选择器的选择信号是减法器的MSB符号位，判断是选择恢复还是选择差值

sub执行的减法的两个操作数一个是剩余玉树，一个是B，**减法的位宽比输入的位宽>1**

剩余余数的更新是原左移，LSB由q的MSB填充

busy表示正在工作，ready表示计算完毕，count用于计数→记录a是否全被移出

> *Given a dividend a and a divisor b, the restoring division algorithm calculates the quotient q and theremainder r such that a = b × q + r and r < b, by subtracting b from the partial remainder (initially the MSB of a).If the result of the subtraction is not negative, we set the quotient bit to 1. Otherwise, b isadded back to the result to restore the partial remainder. Then we shift the partial remainder with theremaining bits of a to the left by one bit for the calculation of the next quotient bit. This procedure is repeated until all the bits of a are shifted out.*

```
`timescale 1ns / 1ps

module restore_Diver(
    input clk,start,
    input [3:0]a,
    input [3:0]b,

    output reg busy,ready,
    output reg[3:0]q,//q的位宽取决于a
    output reg[3:0]r//r位宽取决于b
);
    reg [3:0]reg_b,reg_q,reg_r;

    wire [4:0]sub_res={reg_r[3:0],reg_q[3]}-{1'b0,reg_b};
    wire [3:0]temp_r=sub_res[4]?{reg_r[2:0],reg_q[3]}:sub_res[3:0];

    integer count=0;

    always @(posedge clk) begin
        if (start) begin
            reg_q<=a;
            reg_b<=b;
            reg_r<=0;
            q<=0;
            r<=0;
            busy<=1;
            count<=0;
            ready<=0;
        end else begin
            reg_q={reg_q[2:0],~sub_res[4]};
            reg_r=temp_r;
            count=count+1;
            if (count>=4) begin
                busy<=0;
                ready<=1;
                q=reg_q;
                r=reg_r;
            end
        end
    end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module restoreDivder_test();
    reg clk,start;
    reg [3:0]a;
    reg [3:0]b;

    wire  busy,ready;
    wire [3:0]q;//q的位宽取决于a
    wire [3:0]r;//r位宽取决于b

    initial begin
        clk=0;start=0;
        #4; start=1;a=4'd8;b=4'd2;//4ns
        #2; start=0;//6ns
    end
    always @(*) begin
        if (ready) begin
            #5;start=1;a=a+1;b=b+2;
            if (b==0) begin
                b=1;
            end
        end
        if (busy) begin
            start=0;
        end
    end
    always #5 clk=~clk;
    restore_Diver  restore_Diver_inst (
        .clk(clk),
        .start(start),
        .a(a),
        .b(b),
        .busy(busy),
        .ready(ready),
        .q(q),
        .r(r)
    );
endmodule
```

Verilog

3.4.2不恢复余数算法

恢复余数法是在差值为负时，将r+b再写回r继续后面的运算。

如果不恢复余数直接将负值写入r，那么按照恢复余数法进行的步骤，需要进行r+b然后左移1位再减去b即$2\times(r+b)-b=2\times r+b$。也就是**只需要左移r然后执行加法即可。最后如果余数是负数，则需要加b恢复为正**

> *In the restoring division algorithm described in the previous section, if the result of the subtraction r is negative, b is added back to r. That is, the remainder is restored by r + b, where r is the remainder in the current iteration. The restored remainder r + b is then shifted to the left by one bit, that is, 2(r + b). Then b is subtracted from the shifted remainder, that is, 2(r + b) − b. Because 2(r + b) − b = 2r + b, we can use the negative remainder r directly for the calculation of thenext iteration. This is the idea of the nonrestoring division algorithm. That is, if the partial remainder is negative, we shift it to the left directly and add b to the shifted partial remainder. If the partial remainder is not negative, we shift it to the left and subtract b from the shifted partial remainder (same as the restoring algorithm).*

不恢复余数法的电路图如下所示：

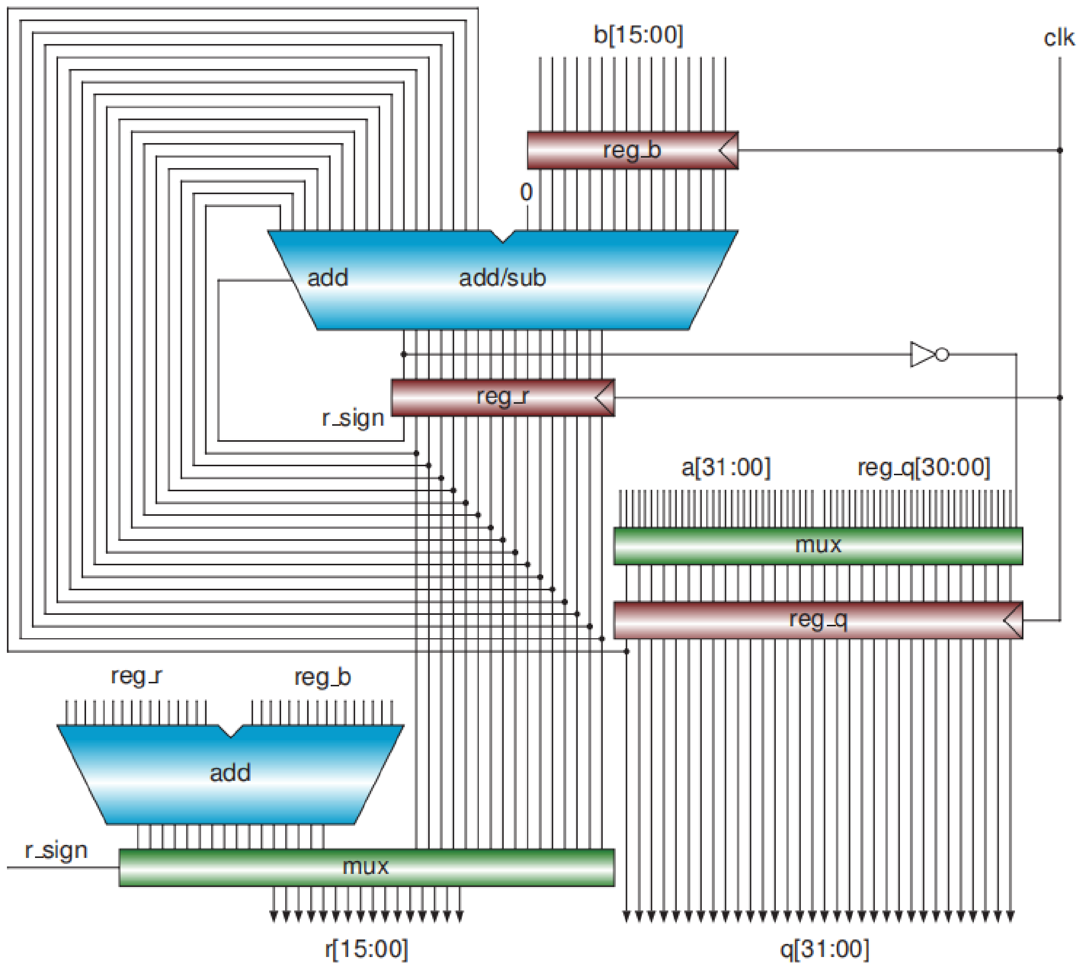


Figure 3.22 Schematic diagram of a nonrestoring divider

reg_r、reg_b、reg_q存储的是剩余余数、除数和商

add/sub源操作数位宽比输入位宽多1，该部件将运算结果写入reg_r，并根据reg_r的符号位决定执行加法还是减法

reg_q上的多路选择器根据start信号选择加载被除数还是加载移位数

当计算完成时，若reg_r为负，那么需要加回b恢复为正

```
module notRestore_Diver(
    input clk,start,
    input [3:0]a,
    input [3:0]b,

    output reg busy,ready,
    output reg[3:0]q,//q的位宽取决于a
    output reg[3:0]r//r位宽取决于b
);
    reg[3:0]reg_q,reg_r,reg_b;
    integer i;
    reg sign;
    wire [4:0]temp_res=sign?{reg_r,reg_q[3]}+{1'b0,reg_b}:{reg_r,reg_q[3]}-{1'b0,reg_b};

    always @(posedge clk) begin
        if (start) begin
            reg_q<=a;
            reg_b<=b;
            reg_r<=0;
            sign<=0;
            i<=0;
            q<=0;r<=0;
            busy<=1;ready=0;
        end else begin
            reg_q={reg_q[2:0],~temp_res[4]};
            reg_r=temp_res[3:0];
            sign=temp_res[4];
            i=i+1;
            if (i>=4) begin
                q=reg_q;
                r=reg_r[3]?reg_r+reg_b:reg_r;
                ready=1;
                busy=0;
            end
        end
    end
endmodule
```

Verilog

带符号的除法器设计

带符号的除法器最简单的实现是和有符号乘法一样，先将有符号数中的负数转换为正数，并保存符号。然后按照无符号数的除法进行处理，最后再恢复符号➞余数符号和除数相同，商符号为两者异或

除了这种方法外，带符号的除法还可以直接使用不恢复余数的方法进行计算：

https://blog.csdn.net/weixin_44611096/article/details/105905241#:~:text=%E6%81%A2%E5%A4%8D%E4%BD%99%E6%95%B0%E6%B3%95%201%20%E8%A1%A5%E7%A0%81%E8%BF%90%E7%AE%97%E9%9C%80%E8%A6%81%E7%AC%A6%E5%8F%B7%E4%BD%8D%E5%92%8C%E6%95%B0%E5%80%BC%E4%BD%8D%E4%B8%80%E8%B5%B7%E5%8F%82%E4%B8%8E%E8%BF%90%E7%AE%97%E3%80%82%202%20%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E6%98%AF%E5%90%A6%E5%A4%9F%E5%87%8F%E9%99%A4%E6%95%B0%EF%BC%8C%E4%B8%8D%E7%9C%8B%E6%96%B0%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%EF%BC%88%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E5%87%8F%E9%99%A4%E6%95%B0%E7%9A%84%E7%BB%93%E6%9E%9C%EF%BC%89%E7%9A%84%E7%AC%A6%E5%8F%B7%E4%BD%8D%EF%BC%8C%E7%9C%8B%E7%9A%84%E6%98%AF%E6%96%B0%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E4%B8%8E%E5%8E%9F%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E7%9A%84%E7%AC%A6%E5%8F%B7%E4%BD%8D%E6%98%AF%E5%90%A6%E4%B8%80%E8%87%B4%EF%BC%8C%E4%B8%80%E8%87%B4%E8%A1%A8%E7%A4%BA%E5%A4%9F%E5%87%8F%EF%BC%8C%E4%B8%8D%E4%B8%80%E8%87%B4%E8%A1%A8%E7%A4%BA%E4%B8%8D%E5%A4%9F%E5%87%8F%E3%80%82,3%20%E4%B8%8D%E5%83%8F%E6%97%A0%E7%AC%A6%E5%8F%B7%E6%95%B0%E6%AF%8F%E6%AC%A1%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E5%87%8F%E9%99%A4%E6%95%B0%E3%80%82%20%E7%9C%8B%E7%9A%84%E6%98%AF%E8%A2%AB%E9%99%A4%E6%95%B0%E6%88%96%E4%B8%AD%E9%97%B4%E4%BD%99%E6%95%B0%E4%B8%8E%E9%99%A4%E6%95%B0%E7%9A%84%E7%AC%A6%E5%8F%B7%EF%BC%8C%E5%90%8C%E5%8F%B7%E5%81%9A%E5%87%8F%E6%B3%95%EF%BC%8C%E5%BC%82%E5%8F%B7%E5%81%9A%E5%8A%A0%E6%B3%95%204%20%E5%A6%82%E6%9E%9C%E9%99%A4%E6%95%B0%E5%92%8C%E8%A2%AB%E9%99%A4%E6%95%B0%E5%90%8C%E5%8F%B7%EF%BC%8C%E5%95%86%E7%9A%84%E7%AC%A6%E5%8F%B7%E5%BA%94%E8%AF%A5%E6%98%AF%E6%AD%A3%EF%BC%9B%E5%A6%82%E6%9E%9C%E9%99%A4%E6%95%B0%E5%92%8C%E8%A2%AB%E9%99%A4%E6%95%B0%E5%BC%82%E5%8F%B7%EF%BC%8C%E5%95%86%E7%9A%84%E7%AC%A6%E5%8F%B7%E5%BA%94%E8%AF%A5%E6%98%AF%E8%B4%9F%E3%80%82%20%E5%A6%82%E6%9E%9C%E5%95%86%E7%9A%84%E7%AC%A6%E5%8F%B7%E9%94%99%E4%BA%86%EF%BC%8C%E5%88%99%E9%9C%80%E8%A6%81%E5%AF%B9%E5%95%86%E8%BF%9B%E8%A1%8C%E6%B1%82%E8%A1%A5%EF%BC%88%E5%90%84%E4%BD%8D%E5%8F%96%E5%8F%8D%EF%BC%8C%E6%9C%AB%E5%B0%BE%E5%8A%A0%E4%B8%80%EF%BC%89%E3%80%82书签：【学习计算机组成原理】补码的除运算_补码除法原理_程序鸡的博客-CSDN博客

实现步骤:

1. 除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和商寄存器Q
2. R和Q同时左移一位
3. 若R与Y同号，则R=R-Y。否则，R=R+Y。
    1. 若R=0或R操作前后符号未变，表示够减，上商为1。
    2. 若R操作前后符号变了，表示不够减，上商为0。恢复余数R值。
4. 重复操作2，3直至取得n位商。
5. 若商符号正确，则Q中的就是商。否则，对Q中的值求补才是正真的商。
6. R中的就是余数。

1. 除数存入reg_b，被除数的符号扩展存入reg_r，被除数存入reg_q
2. 若reg_r与reg_b同号，则执行减法{reg_r[位宽-1:0],reg_q[位宽-1]}-{reg_b[位宽-1],reg_b}，否则执行加法{reg_r[位宽-1:0],reg_q[位宽-1]}+{reg_b[位宽-1],reg_b}
3. 根据运算得到的结果符号和reg_r的符号，若相同则置reg_qLSB为1，否则置0
4. 重复reg_q位宽次，至reg_q所有位均被处理
5. 商处理：若商符与被除数符号不同则商取反+1，否则不变输出

余数处理：若余数为负，则需要加上除数的绝对值，否则正常输出

```
module notRestore_comDivider(
    input clk,start,
    input [3:0]a,
    input [3:0]b,

    output reg busy,ready,
    output reg[3:0]q,//q的位宽取决于a
    output reg[3:0]r//r位宽取决于b
);
    reg[3:0]reg_q,reg_r,reg_b;
    integer i;
    reg signA,sign;
```

```verilog
    wire [4:0]temp_res=(signA==reg_b[3])?{reg_r[3:0],reg_q[3]}-{reg_b[3],reg_b}:
    {reg_r[3:0],reg_q[3]}+{reg_b[3],reg_b};

    always @(posedge clk) begin
        if (start) begin
            reg_q<=a;
            reg_b<=b;
            reg_r<={4{a[3]}};
            signA<=a[3];
            sign<=a[3]^b[3];
            i<=0;
            q<=0;r<=0;
            busy<=1;ready=0;
        end else begin
            reg_q={reg_q[2:0],~temp_res[4]};
            reg_r=temp_res[3:0];
            signA=temp_res[4];
            i=i+1;
            if (i>=4) begin
                q=(sign!=reg_q[3])?~reg_q+1:reg_q;
                r=signA?(reg_b[3]?reg_r+~reg_b+1:reg_r+reg_b):reg_r;
                ready=1;
                busy=0;
            end
        end
    end

endmodule
```
Verilog

3.4.3 Goldschmidt除法算法

https://zhuanlan.zhihu.com/p/404613667书签：Goldschmidt近似除法

GoldSchmidt算法用于计算a、b(形式是0.1xxxxx)的除法，0.5≤a,b≤1，利用下面的比例公式将除法转换为乘法计算商q

$$\frac{a \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}{b \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}$$

当上式中的分母趋近于1时，分子即为商q

推导过程如下：

1. 令$\frac{1}{b} = \frac{1}{1+y}$，然后对后式进行麦克劳林公式展开

$$\frac{x}{b} = \frac{x}{1+y} \approx x(1-y)(1+y^2)(1+y^4)(1+y^8)ldots$$

2. 因此每次迭代系数rk可以表示为

$$\{.$$

3. 因为$b_0 = 1 + y$那么，分母的迭代模型如下所示

$$\{.$$

4. 因此可以得到迭代系数与b的关系为

$$r_k = 2 - b_k$$

5. 因此可以得到

初始时：$x_0 = a_0, y_0 = b_0, r0 = 2 - b0 = barb\_0 + 1$

之后：$x_n = x_{n-1} \times r_{n-1}, y_n = y_{n-1} \times r_{n-1}, r_n = 2 - b_n = 2 - y_n = bary\_n + 1$

电路图如下所示：

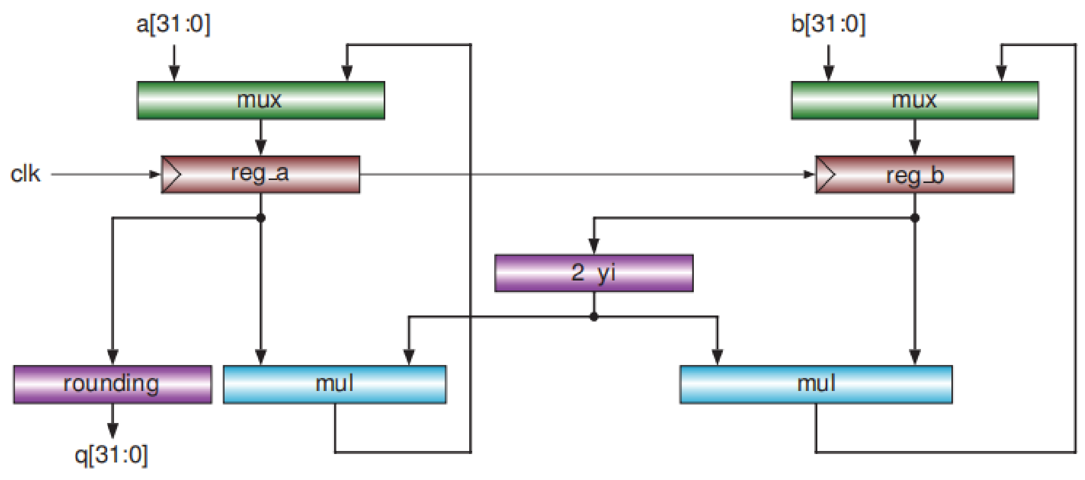


Figure 3.24 Schematic diagram of the Goldschmidt divider

🏵四舍五入实现方式：reg[63:0]64位小数，前32位有效，reg[63:32]+|reg[31,29]

通过实验观察，分母趋近于1最少在第5次，所以下面手动设置循环，当循环到第五次时输出

```
`timescale 1ns / 1ps

module goldSchmdit_Divder(
    input clk,start,
    input [31:0]a,b,

    output reg busy,ready,
    output reg[31:0]q,//q的位宽取决于a
    output reg [31:0]y //看小数点后的位
);
    reg [63:0]reg_a,reg_b;//存储x，y,初值是a,b
    wire [63:0]r_iter=~reg_b+1;//迭代r
    wire [127:0]temp_x=r_iter*reg_a;//最高位是0. /1.
    wire [127:0]temp_y=r_iter*reg_b;//最高位是0. /1.

    integer i;
    always @(posedge clk) begin
        if (start) begin
            reg_a<={1'b0,a,31'b0};//0.1xxxxxx
            reg_b<={1'b0,b,31'b0};//0.1xxxxxx
            busy<=1;ready<=0;
            i<=0;
            q<=0;y<=0;
        end else begin
            reg_a=temp_x[126:63];//取除去小数点后的64位
            reg_b=temp_y[126:63];//取除去小数点后的64位
            i=i+1;
            if (i>=5) begin
                q=reg_a[63:32]+|reg_a[31:29];
                y=reg_b[62:31];
                ready=1;
                busy=0;
            end
        end
    end
endmodule
```
Verilog

3.4.4 Newton-Raphson除法算法

https://zhuanlan.zhihu.com/p/400064205书签：Newton-Raphson Method

推导

Newton-Raphson即牛顿迭代法，该方法**利用f(x)的泰勒级数的前两项求解f(x)=0的根**，f(x)在x0的泰勒级数如下：

$$f(x)=f(x0)+f^{prime}(x0)(x-x0)+\frac{f^{primeprime}(x0)}{2!}(x-x0)^2+\cdots+\frac{f^n(x0)}{n!}(x-x0)^n+R_n(x)$$

由前两项即可得零点x_{n+1}的迭代公式为：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

要求a/b的商，可以先求出$\frac{1}{b}$，然后商$q=\frac{a}{b}$

建立方程求1/b如下，当y=0时，x=1/b

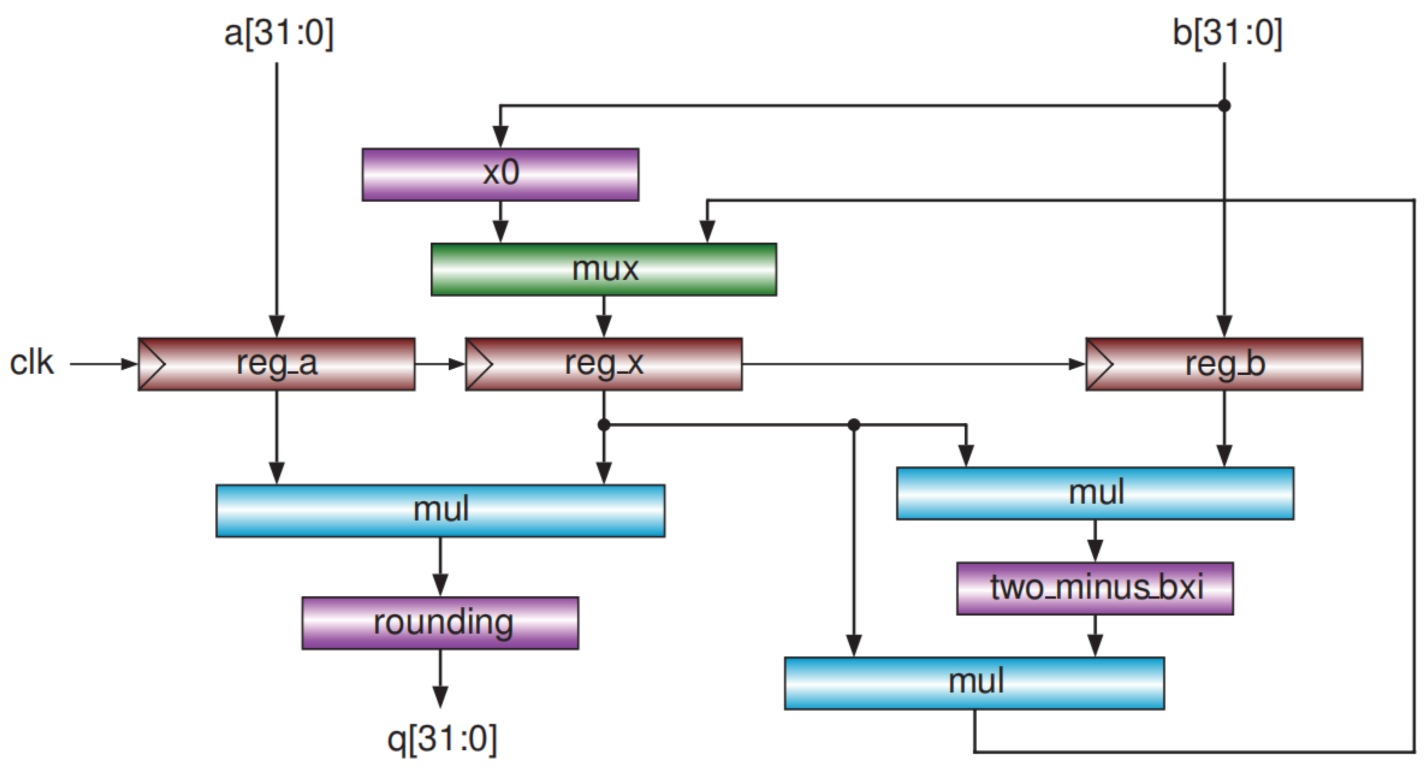
$$y=\frac{1}{x}-b$$

f(x)的导数为$-\frac{1}{x^2}$，则迭代公式为$x_{n+1}=x_n(2-x_nb)$

🎉同GoldSchmdit算法，牛顿迭代法也要求0.5≤a,b≤1

实现

其电路结构如下所示：



reg_a存储的是x，reg_b存储的即为除数b
也设置迭代次数，初值x0从ROM中读取，设置2次迭代即足够

Figure 3.26 Schematic diagram of 32-bit Newton–Raphson divider

```
`timescale 1ns / 1ps

module newton_divider(
    input clk,start,
    input [31:0]a,b,

    output reg busy,ready,
    output reg[31:0]q//q的位宽取决于a
);

    reg [33:0]reg_a,reg_x,reg_b;
    wire [65:0]mul_res=reg_a*reg_x;
    wire [65:0]temp=reg_b*reg_x;
    wire [33:0]two_minus_x=~temp[64:31]+1;
    wire [67:0]x_n=reg_x*two_minus_x;
    integer i;
    always @(posedge clk) begin
        if (start) begin
            reg_a<=a;
            reg_b<=b;
            reg_x<={2'b1,rom(b[30:27]),24'b0};//小数点后4位
            busy<=1;
            ready<=0;
            i<=0;
        end else begin
            reg_x=x_n[66:33];
            i=i+1;
            if (i>=2) begin
                q=mul_res[64:33]+|mul_res[32:30];
                busy=0;
                ready=1;
            end
        end
    end
    function [7:0] rom; // a rom table
        input [3:0] b;
        case (b)
            4'h0: rom = 8'hff; 4'h1: rom = 8'hdf;
            4'h2: rom = 8'hc3; 4'h3: rom = 8'haa;
            4'h4: rom = 8'h93; 4'h5: rom = 8'h7f;
            4'h6: rom = 8'h6d; 4'h7: rom = 8'h5c;
            4'h8: rom = 8'h4d; 4'h9: rom = 8'h3f;
            4'ha: rom = 8'h33; 4'hb: rom = 8'h27;
            4'hc: rom = 8'h1c; 4'hd: rom = 8'h12;
            4'he: rom = 8'h08; 4'hf: rom = 8'h00;
        endcase
    endfunction
endmodule
```
Verilog

3.5二进制平方根

3.5.1恢复平方根算法

对于32位根和d，假设q是部分根，r是部分余数

$$
\begin{array}{llcccccll}
 & & & q_1 & q_2 & q_3 & q_4 & & \\
 & & & 1 & 0 & 1 & 1 & ;\ \text{Square root } (11_{10}) & \\
\cline{4-7}
 & 1 & \sqrt{} & 01 & 11 & 11 & 11 & ;\ \text{Radicand } (127_{10}) & \\
(q_1=1) & 1 & & 1 & & & & ;\ 1\times 1=1 & (\text{guess } 1) \\
\cline{4-7}
 & 100 & & 0 & 11 & & & ;\ 1+1=10 & \\
(q_2=0) & 0 & & & 0 & & & ;\ 100\times 0=0 & (\text{guess } 0) \\
\cline{4-7}
 & 1001 & & & 11 & 11 & & ;\ 100+0=100 & \\
(q_3=1) & 1 & & & 10 & 01 & & ;\ 1001\times 1=1001 & (\text{guess } 1) \\
\cline{5-7}
 & 10101 & & & 1 & 10 & 11 & ;\ 1001+1=1010 & \\
(q_4=1) & 1 & & & 1 & 01 & 01 & ;\ 10101\times 1=10101 & (\text{guess } 1) \\
\cline{5-7}
 & & & & & 01 & 10 & ;\ \text{Remainder } (6_{10}) &
\end{array}
$$

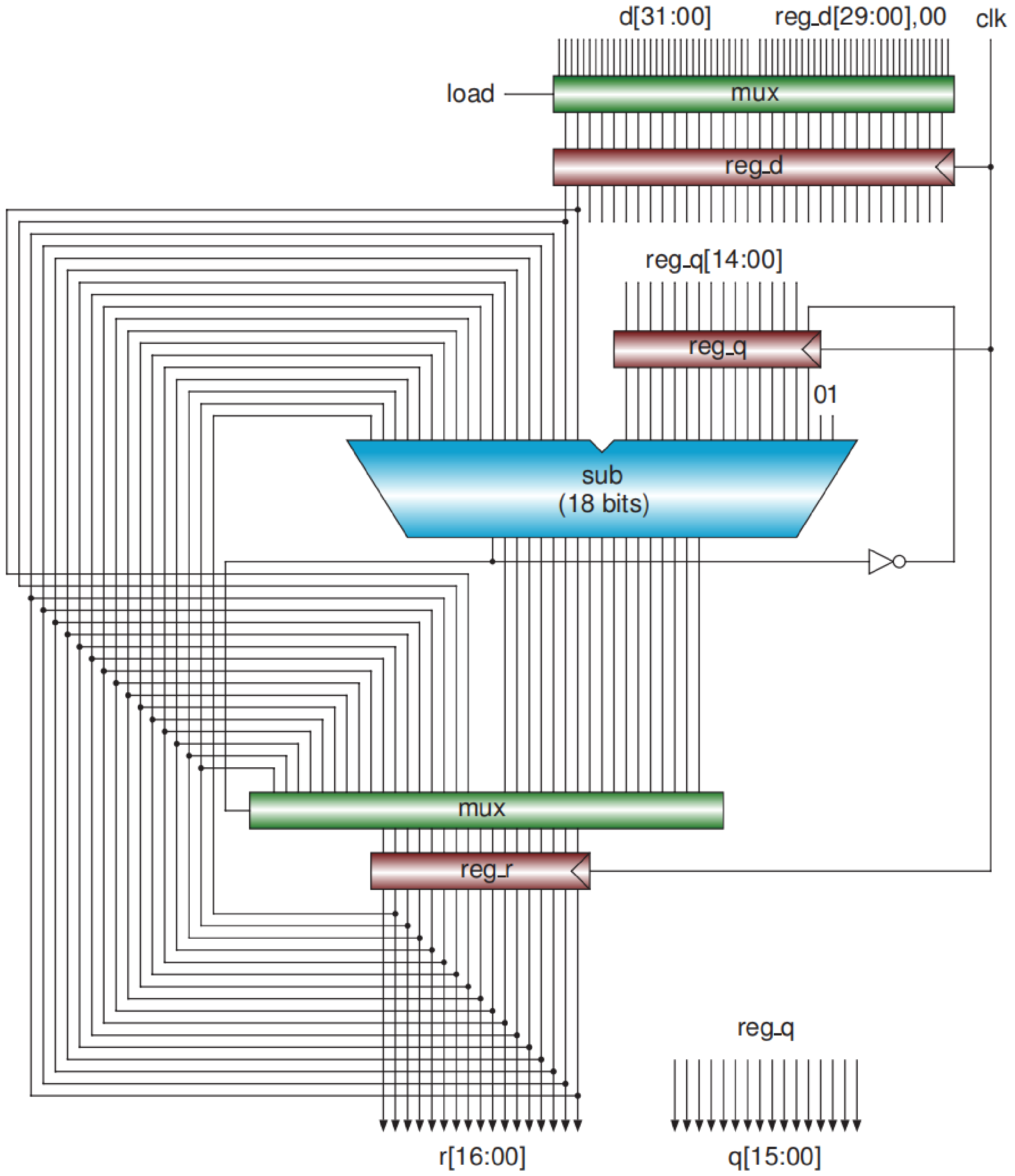
以左图为例进行介绍：

1. 首先将被开方数每两位为一对进行划分
2. 初始时q置0，r为d的两位MSB
3. 恢复被开方数算法将从r中减去q01 q01即4q+1
4. 若结果为正，则设置当前q位为1否则，设置为0并恢复r

5. r接下来取剩余d的两位MSB——d左移两位

> *For a 32-bit radicand d, assume that q is a partial root and ris a partial remainder. Initially, let q = 0 and let r be the two MSBs of d. The restoring square root algorithm subtracts q01 from r, where q01 means q ≪ 2 + 1, or 4q + 1. If the result of the subtraction is not negative, we set the root bit to 1. Otherwise, the root bit is reset to 0 and q01 is added back to the result to restore the original r. Then r is obtained by concatenating the next two MSBs of d. This procedure is repeated until all the bits of d are dealt with*

其电路结构如下:



左图中reg_r、reg_q、reg_d分别存储余数、根、被开方数

load信号有效时加载数据至reg_d，无效时reg_d=reg_d<<2;以便于余数取d的两位MSB

根存储在reg_q中，减法时是reg_q<<2+1因此减数即{reg_q[位宽-2:0],01}；根据结果对reg_q最低位置1或复位0→reg_q={reg[位宽-1],~减法符号}

减法的被减数是{reg_r[左移两位],reg_d[高两位]}若结果为正则reg_r写结果，为负则写{reg_r[左移两位],reg_d[高两位]}

Figure 3.28 Schematic diagram of a restoring square rooter

```
`timescale 1ns / 1ps
```

```
module restore_rooter(
    input clk,load,
    input [31:0]d,

    output reg busy,ready,
    output reg [15:0]q,
    output reg [16:0]r
);
    reg [31:0] reg_d;
    reg [15:0] reg_q;
    reg [16:0] reg_r;
    integer i;
    wire [17:0]sub_res={reg_r[15:0],reg_d[31:30]}-{reg_q[15:0],2'b1};
    wire [16:0]temp_r=sub_res[17]?{reg_r[14:0],reg_d[31:30]}:sub_res[16:0];

    always @(posedge clk) begin
        if (load) begin
            reg_d<=d;reg_q<=0;reg_r<=0;
            busy=1;ready=0;
            q<=0;r<=0;i<=0;
        end else begin
            reg_d={reg_d[29:0],2'b0};
            reg_q={reg_q[14:0],~sub_res[17]};
            reg_r=temp_r;
            i=i+1;
            if (i>=16) begin
                q=reg_q;
                r=reg_r;
                ready=1;
                busy=0;
            end
        end
    end
endmodule
```

Verilog

3.5.2不恢复平方根算法

和不恢复余数一致，不恢复平方根法的r也需要先加上之前的减数，然后再进行下一个处理：

```
`timescale 1ns/1ps

module noRestore_root(
    input clk,load,
    input [3:0]d,

    output reg busy,ready,
    output reg [1:0]q,
    output reg [3:0]r
);
    integer i;
    reg[3:0]reg_d;
    reg[1:0]reg_q;
    reg[3:0]reg_r;
    reg[1:0]reg_q_old;
    wire [2:0]r_old={1'b1,reg_r}+{reg_q_old[1:0],2'b01};
    wire [3:0]addSub=reg_r[3]?{r_old[1:0],reg_d[3:2]}-{reg_q[1:0],2'b01}
                    :{reg_r[1:0],reg_d[3:2]}-{reg_q[1:0],2'b01};
    reg [2:0]r_n;
    always @(posedge clk) begin
        if (load) begin
            q<=0;r<=0;
            reg_d<=d;reg_q<=0;reg_r<=0;reg_q_old<=0;
            i<=0;busy<=1;ready<=0;
        end else begin
```

```
            reg_d={reg_d[1:0],2'b0};
            reg_q_old=reg_q;
            reg_q={reg_q[0],~addSub[3]};
            reg_r=addSub;
            r_n=reg_r+{reg_q_old[1:0],2'b01};
            i=i+1;
            if (i>=2) begin
                q=reg_q;
                r=reg_r[3]?r_n:reg_r;
                ready=1;
                busy=0;
            end
        end
    end
endmodule
```
Verilog

3.5.3 GoldSchmdit平方根算法

用GoldSchmdit平方算法求解根式则是使用下面的比例式：

$$\frac{d \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}{d \times {r_0}^2 \times {r_1}^2 \times {r_2}^2 \times \cdots \times {r_{n-1}}^2}$$

当分母趋近于1时，分子恰为$sqrt$

迭代公式是：

$$r_i = 1 + (1 - x_i)/2 = \frac{3 - x_i}{2} = 1.5 - x_i/2$$
$$d_{i+1} = d_i \times r_i$$
$$x_{i+1} = x_i \times {r_i}^2$$

```
//1.5用二进制表示是1.1000000->c000


`timescale 1ns / 1ps

```

```
module goldSchmidt_rooter(
    input clk,load,
    input [31:0]d,

    output reg busy,ready,
    output reg [31:0]q,
    output reg [31:0]x//分母0.111111111时
);
    reg [63:0]reg_d,reg_x;//reg_d中存储分子，reg_x存储分母 0.xxxxxxx
    reg [63:0]reg_r;//1.xxxxxxxx
    wire [63:0]r=64'hc000_0000-{1'b0,reg_x[62:0]};//1.1-0.xxx
    wire [127:0]di=reg_d*reg_r;//xx.xxxxxxx
    wire [127:0]temp_i=reg_x*reg_r*reg_r;//xx.xxxxxx
    wire [63:0]xi={1'b0,temp_i[126:64]+|temp_i[63:0]};//x.xxxxxx

    integer i;
    always @(posedge clk) begin
        if (load) begin
            reg_d<={1'b0,d,31'b0};reg_x<={1'b0,d,31'b0};//最高位是符号位 0.xxxxxx
            i<=0;busy<=1;ready<=0;
        end else begin
            reg_r=r;
            reg_x=xi;
            reg_d=di[126:63]+|di[62:0];//x.xxxxxx
            i=i+1;
            if (i>=5) begin
                q=reg_d[62:31]+|reg_d[31:29];//xxxxxxx
                x=reg_x[62:31];//xxxxxxxx
                ready=1;
                busy=0;
            end
        end
    end
endmodule
```

Verilog

3.5.4 Newton-Raphson平方根算法

建立的方程f(x)为$f(x)=\frac{1}{x^2}-d$，零点时解的的x为$\frac{1}{sqrt}$，然后再乘以d即为最终得到的平方根

迭代方程为$x_{n+1}=\frac{x_n}{2}(3-{x_n}^2d)$//1100

```
`timescale 1ns / 1ps

module newton_rooter(
    input clk,load,
    input [31:0]d,

    output reg busy,ready,
    output reg [31:0]q//xxxxxxxx
);
    reg [31:0] reg_d;//xxxxxxxx
    reg [33:0] reg_x;//xx.xxxxxxxx
    integer i;
    // x_{i+1} = x_i * (3 - x_i * x_i * d) / 2
    wire [67:0] x_2 = reg_x * reg_x; // xxxx.xxxxx...x 34+34
    wire [67:0] x2d = reg_d * x_2[67:32]; // xxxx.xxxxx...x 32 36
    wire [33:0] b34 = 34'h300000000 - x2d[65:32]; // xx.xxxxx...x
    wire [67:0] x68 = reg_x * b34; // xxxx.xxxxx...x 34+34
    wire [65:0] d_x = reg_d * reg_x; // xx.xxxxx...x 24+32
    wire [7:0] x0 = rom(d[31:27]);
    always @ (posedge clk) begin
        if (load) begin
            reg_d <= d;
            reg_x <= {2'b1,x0,24'b0}; // 01.xxxx0...0
```

```verilog
            busy <= 1;
            ready <= 0;
            i <= 0;q<=0;
        end else begin
            reg_x <= x68[65:32];
            i=i+1;
            if (i == 2'h2) begin
                busy <= 0;
                ready <= 1;
                q<=d_x[63:32] + |d_x[31:0];
            end
        end
    end
    function [7:0] rom; // about 1/d ^{1/2}
        input [4:0] d;
        case (d)
            5'h08: rom = 8'hff; 5'h09: rom = 8'he1;
            5'h0a: rom = 8'hc7; 5'h0b: rom = 8'hb1;
            5'h0c: rom = 8'h9e; 5'h0d: rom = 8'h9e;
            5'h0e: rom = 8'h7f; 5'h0f: rom = 8'h72;
            5'h10: rom = 8'h66; 5'h11: rom = 8'h5b;
            5'h12: rom = 8'h51; 5'h13: rom = 8'h48;
            5'h14: rom = 8'h3f; 5'h15: rom = 8'h37;
            5'h16: rom = 8'h30; 5'h17: rom = 8'h29;
            5'h18: rom = 8'h23; 5'h19: rom = 8'h1d;
            5'h1a: rom = 8'h17; 5'h1b: rom = 8'h12;
            5'h1c: rom = 8'h0d; 5'h1d: rom = 8'h08;
            5'h1e: rom = 8'h04; 5'h1f: rom = 8'h00;
            default: rom = 8'hff;
        endcase
    endfunction
endmodule
```
Verilog

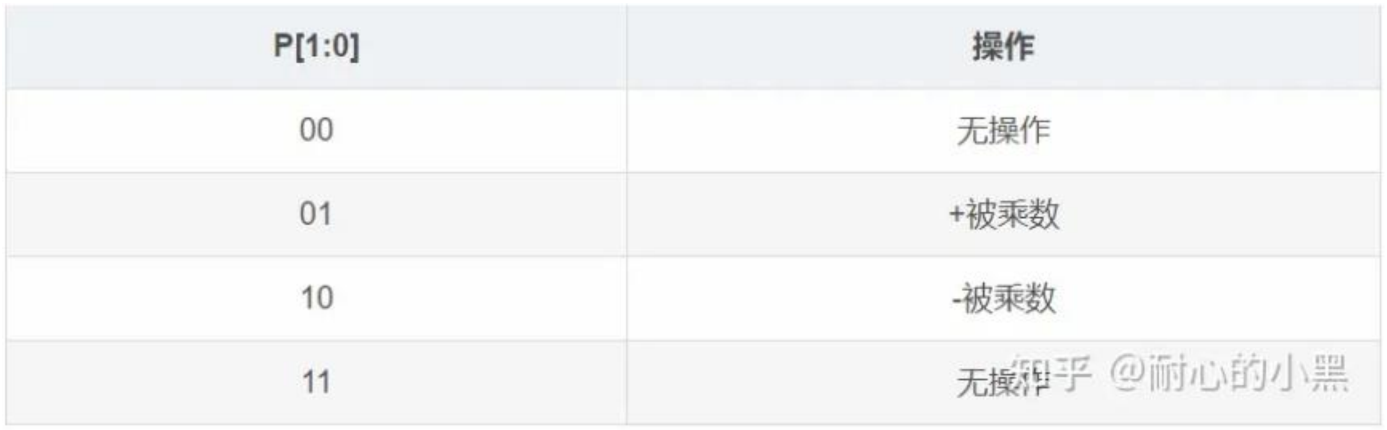
习题

1. Booth's multiplication

https://zhuanlan.zhihu.com/p/452236186书签：verilog Booth算法乘法器的实现(有无符号)

根据被乘数和乘数的位数，设置部分积位数为“被乘数位数+乘数位数”

在[乘数位宽:1]位置上存放乘数，然后判断[1:0]执行下图相应的操作写部分积的高被乘数位，然后右移1位。循环处理移位乘数位次，部分积[MSB:1]即为最终结果



P[1:0]	操作
00	无操作
01	+被乘数
10	-被乘数
11	无操作

```verilog
`timescale 1ns / 1ps


module booths_multiplier(
    input [3:0]a,b,
    output reg[7:0]res
);
    reg [4+4:0]p;
    reg [3:0]temp;
    integer i;
    always @(*) begin
        p[0]=1'b0;
        p[4:1]=b;
        p[8:5]=4'b0;
        for ( i= 0; i<4;i=i+1 ) begin
            case (p[1:0])
```

```verilog
                2'b11,
                2'b00:p=$signed(p)>>>1;
                2'b01:begin
                    temp=p[8:5]+a;
                    p[8:5]=temp;
                    p=$signed(p)>>>1;
                end
                2'b10:begin
                    temp=p[8:5]+~a+1;
                    p[8:5]=temp;
                    p=$signed(p)>>>1;
                end
            endcase
        end
        res=p[8:1];
    end
endmodule
```

Verilog

2. SRT除法平方根法

之后再补充