**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

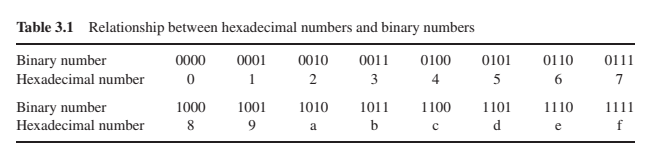
## 3.1.1 二进制和十六进制形式

**只给定一个二进制数是不知道其含义的**  
比如给定：00110011110111100000000100000000

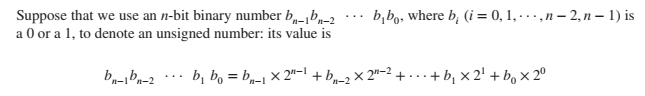
可以是一个整数：870187264  
可以是一个浮点数：0.000000103378624771721661090850830078125  
可以是一条MIPS指令： *addi $30, $30, 256*  
可以是一个地址  
可以是一个媒体数据等

**|** *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.*

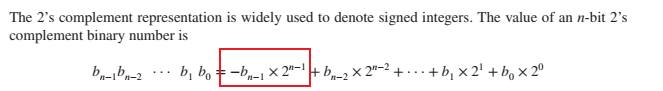
十六进制是为方便记忆产生的，每4位二进制看作一位十六进制



## 3.1.2 无符号二进制整数

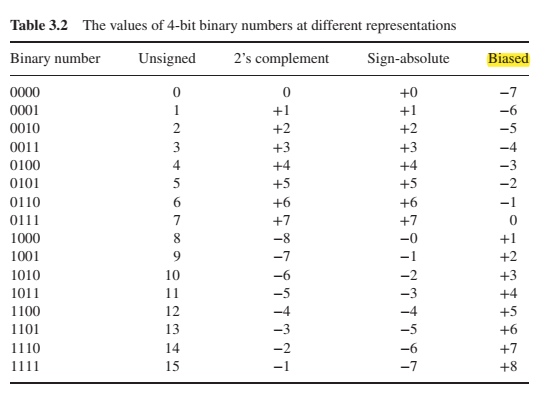


## 3.1.3有符号二进制整数



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**

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**()



# 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.*

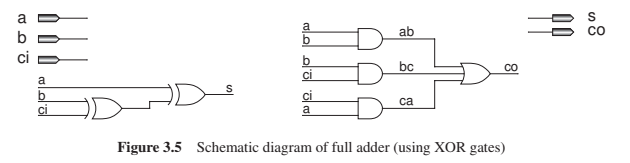
### 全加器——加低位的进位

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：

根据表达式实现电路是：



#### 逻辑门实现

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

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

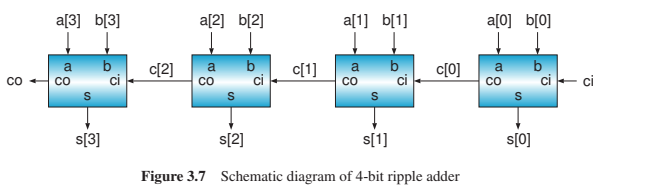
Verilog

`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 (  
 .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**



`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

`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

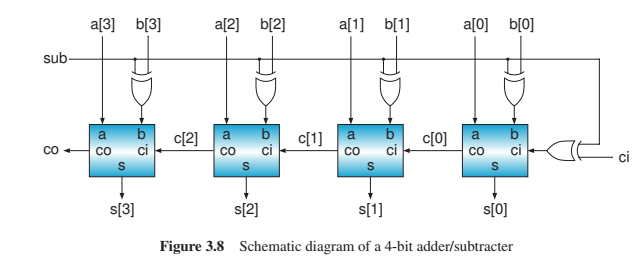
#### 减法器实现

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](https://www.wolai.com/2mgpd3SfCnzEawtAXnscCN#4rwxufWQ6jTfcnPVEpDgKp)，那么

因此，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，执行否则执行**

**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**



`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]);  
 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

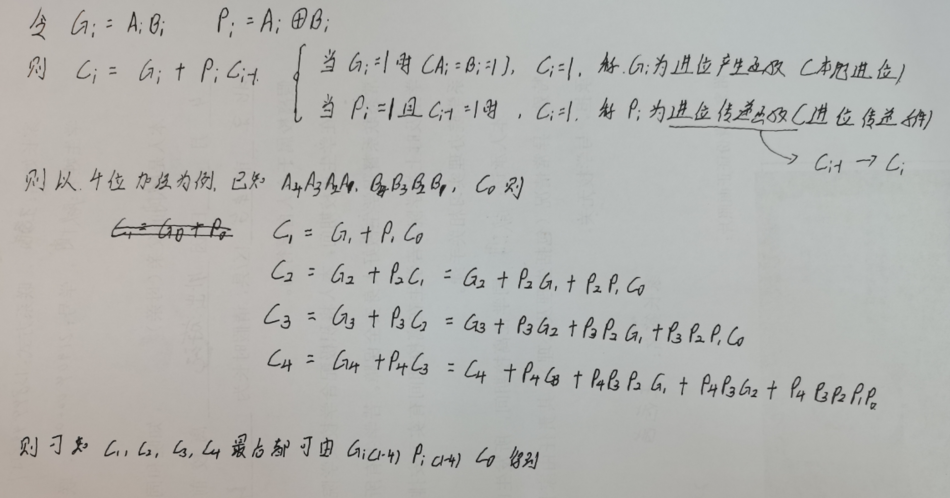
## 3.2.2 超前进位加法器

[超前进位全加器CLA](https://www.wolai.com/5NLCUoxzgi8trKfUqoy58T#aMwNRPK79o3AVBpN2DhopD)

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*

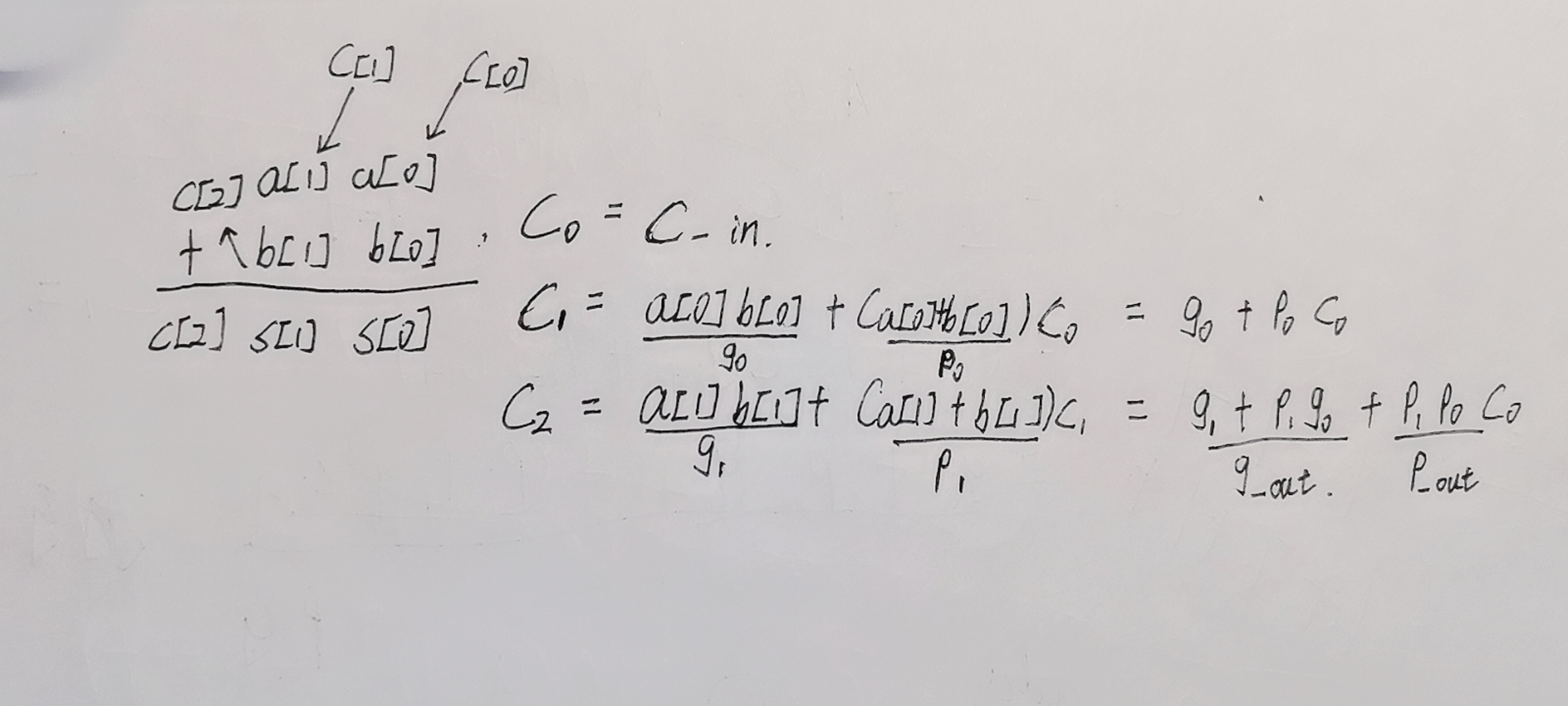
CLA理想状态下可以实现进位的同时产生从而加快计算。推导过程如下：



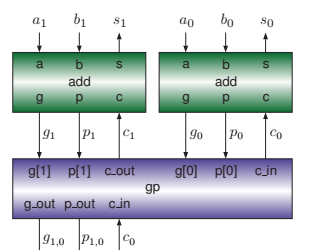
这里有**本位进位（进位产生器g）和进位传递条件（进位传递者p）**

### 两位CLA

输入a[1:0],b[1:0],c\_in；输出s[1:0],c\_out



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

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

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

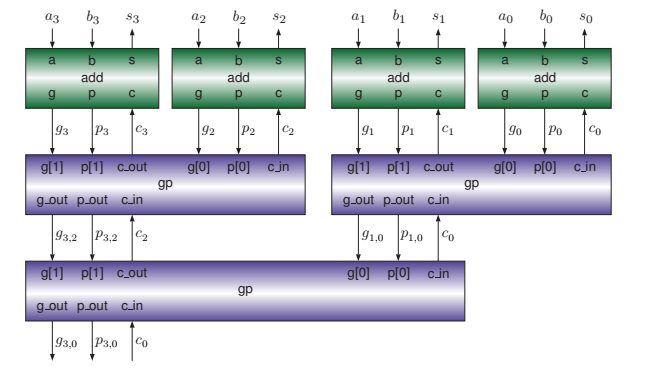
#### 两位CLA

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



#### 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

`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

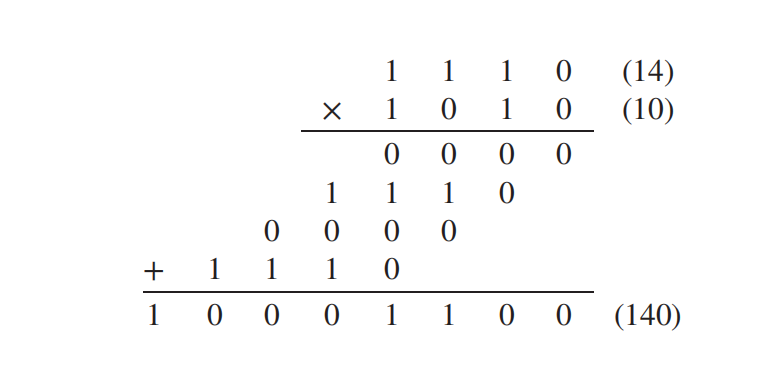
`timescale 1ns / 1ps  
  
module cla32\_alu\_test();  
 reg [31:0]a,b;  
 reg ci,sign,sub;  
  
 wire [31:0]s;  
 wire c\_out;  
 wire overflow;//有符号数溢出，无符号数加法溢出  
 wire c\_take;//无符号数减法借位  
  
 initial begin  
 sign=0;sub=0;a=32'd5;b=32'd9;ci=0;//5+9+0  
 #5; sign=0;sub=0;a=32'd7;b=32'd11;ci=1;//7+11+1  
 #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



#### C语言实现的无符号数乘法

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，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

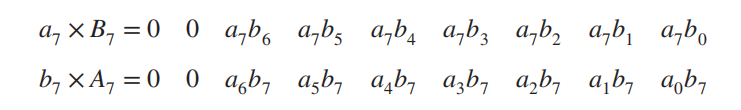
以两个8位采用补码表示的有符号数相乘为例，根据[行内引用](https://www.wolai.com/2mgpd3SfCnzEawtAXnscCN#r3bgw9yBcBZru5Koq5aJ53)可以表示为下面的形式

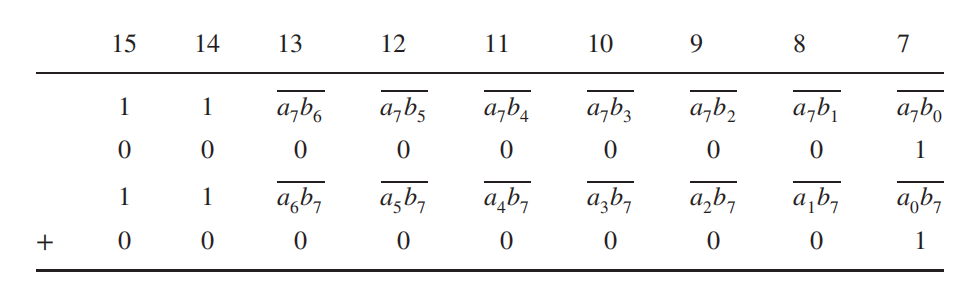
则可以表示如下，即为无符号数的乘法

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

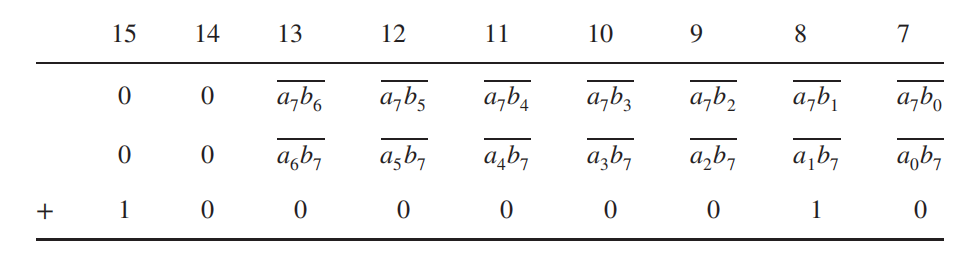
第一个因式，直接左移14位即可  
第四个因式，是一个无符号数相乘

第二、三个因式可以先表示再取反

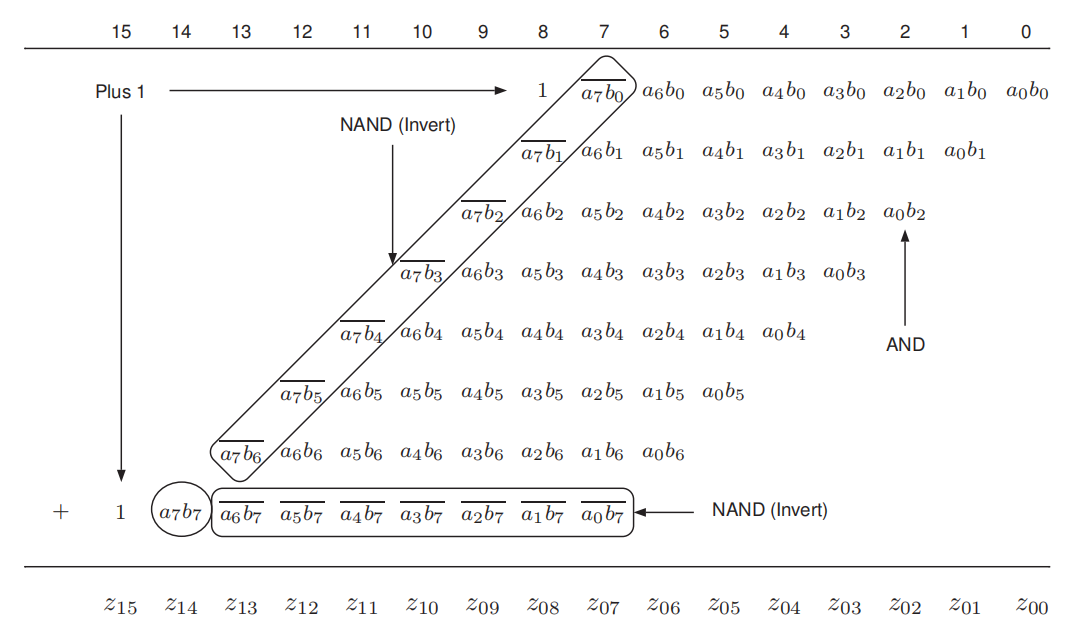




最后去掉进位，可以得到



则最后的加法式为



### 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

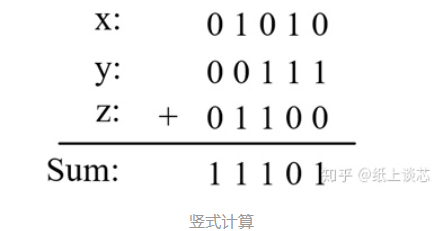
https://zhuanlan.zhihu.com/p/102387648书签：[【HDL系列】进位保存加法器原理与设计](https://zhuanlan.zhihu.com/p/102387648)

#### 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。

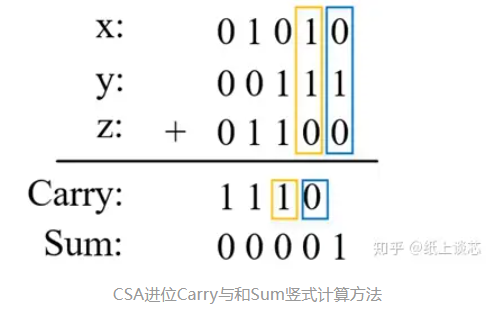
下面以（10+7+12=29）为例来讲解进位保存加法器的操作：

* 1. 正常的执行加法



先对10和7相加，再将其结果加12得到最终的结果

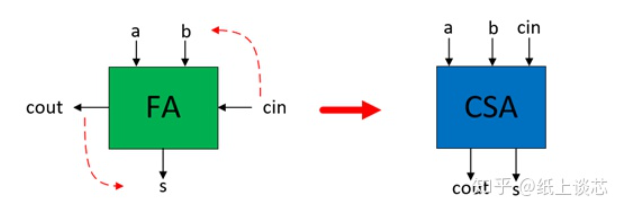
* 1. CSA执行



**对三个加数同时进行处理，分别计算产生sum和co，，。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：

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

Verilog

#### CSA实现10+7+12

`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

`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](https://zhuanlan.zhihu.com/p/130968045)

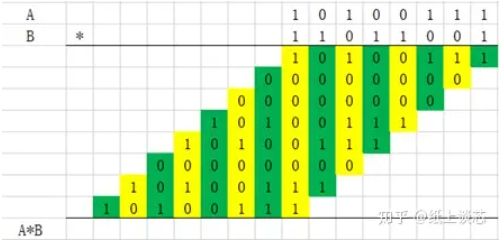
✍🏽**Wallac Tree就是利用CSA“全加器3-2压缩的特性”对乘法矩阵中的数进行快速的相加——每列3个加数分为一组，压缩至两个加数，循环往复至最后只有两行使用进位传播加法器相加即可得到最终的结果**

#### 示例理解

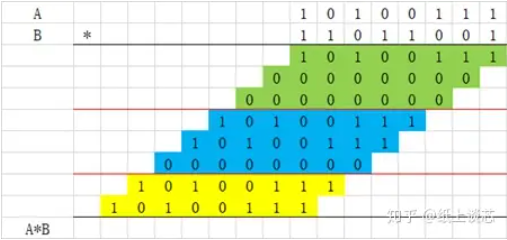
🎉一个Wallac Tree Multiplier包括三个部分：

* 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

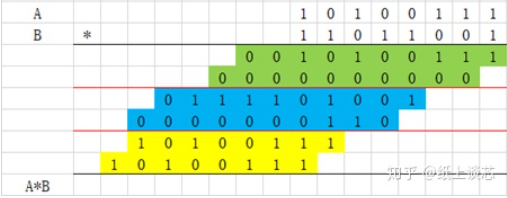
以计算无符号数乘法“10100111\*11011001”为例，其乘法阵列如下：



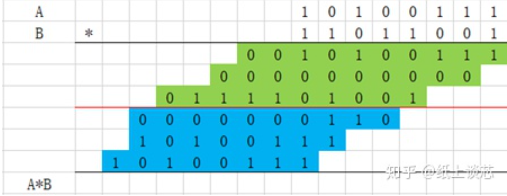
1. 第一级：按每列3个一组进行分组，不足3个的保持到下一级



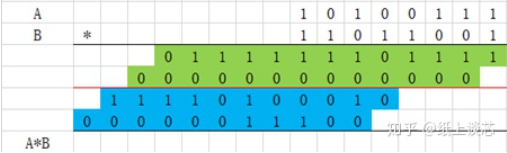
产生结果：



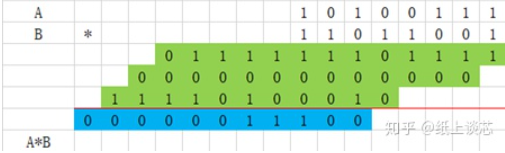
1. 第二级：再次按照每列3个进行分组



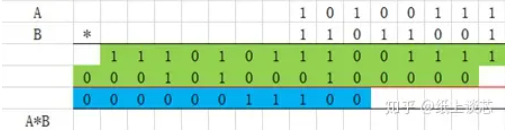
产生结果：



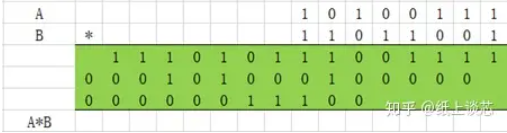
1. 第三级：再次分组



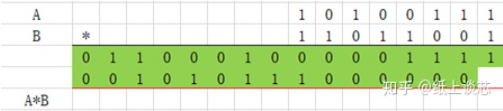
产生结果：



1. 第四级：再次分组



产生结果：



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);  
  
 //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

`timescale 1ns / 1ps  
  
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

`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（）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**

|  |  |
| --- | --- |
|  | 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。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：

|  |  |
| --- | --- |
|  | 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 |

`timescale 1ns / 1ps  
  
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;  
  
 wire [4:0]temp\_res=reg\_r[3]?{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;  
 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];  
 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

## 3.4.3 Goldschmidt除法算法

https://zhuanlan.zhihu.com/p/404613667书签：[Goldschmidt近似除法](https://zhuanlan.zhihu.com/p/404613667)

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**

当上式中的分母趋近于1时，分子即为商q

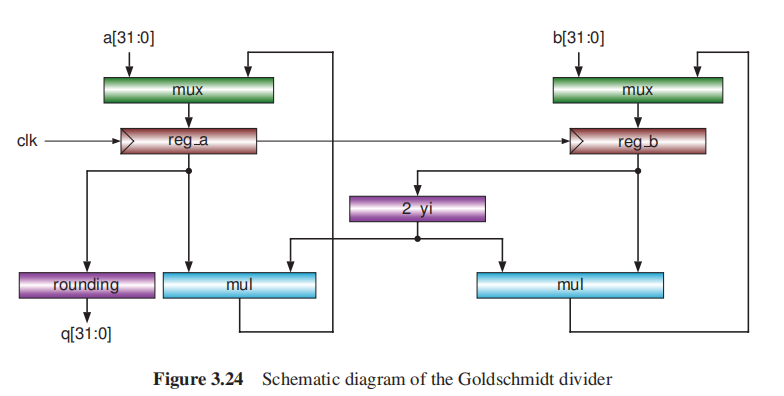
推导过程如下：

1. 令，然后对后式进行麦克劳林公式展开
2. 因此每次迭代系数rk可以表示为
3. 因为那么，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
5. 因此可以得到

初始时：

之后：

电路图如下所示：



🎉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](https://zhuanlan.zhihu.com/p/400064205)

#### 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：

由前两项即可得零点的迭代公式为：

要求a/b的商，可以先求出，然后商

建立方程求1/b如下，当y=0时，x=1/b

f(x)的导数为，则迭代公式为

🎉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 |

`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 | 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]} |

`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：

**当分母趋近于1时，分子恰为**

迭代公式是：

//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，零点时解的的x为，然后再乘以d即为最终得到的平方根

迭代方程为//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  
 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