verilog设计乘法器

2019-08-09 14:24:59

# 概述

移位乘法器、原码乘法器、补码乘法器、流水线乘法器

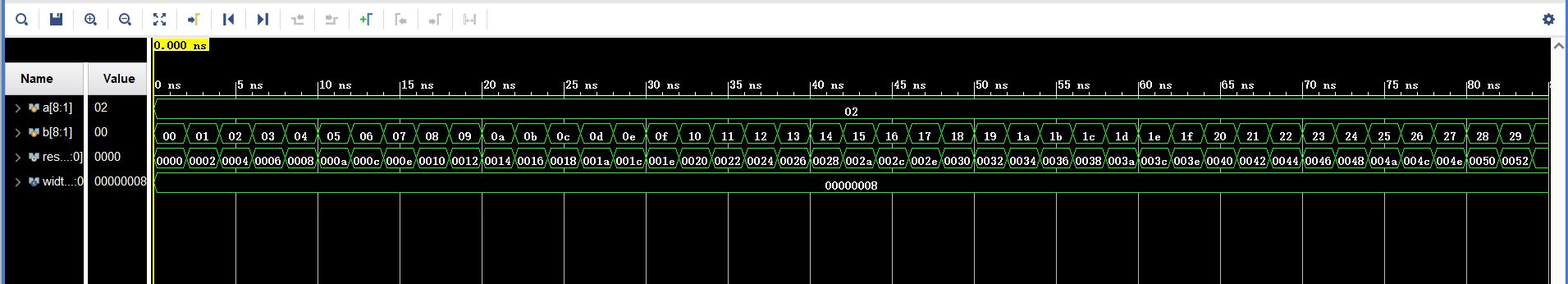
# 左移位乘法器

## 设计文件

module cfq(result,a,b);  
parameter width = 8;  
input [width:1] a,b;  
output [2\*width-1:0] result;  
 // wire [width:1] a,b; 一开始定义为reg类型报错，输入端口为非线型变量，不可定义为reg类型  
reg [2\*width-1:0] result;  
integer index;  
always @(a or b)  
 begin  
 result = 0;  
 for(index = 0;index <= width;index = index+1)  
 begin  
 if(b[index])  
 begin  
 result = result + (a << (index-1));  
 end  
 end  
 end  
endmodule

## 仿真文件

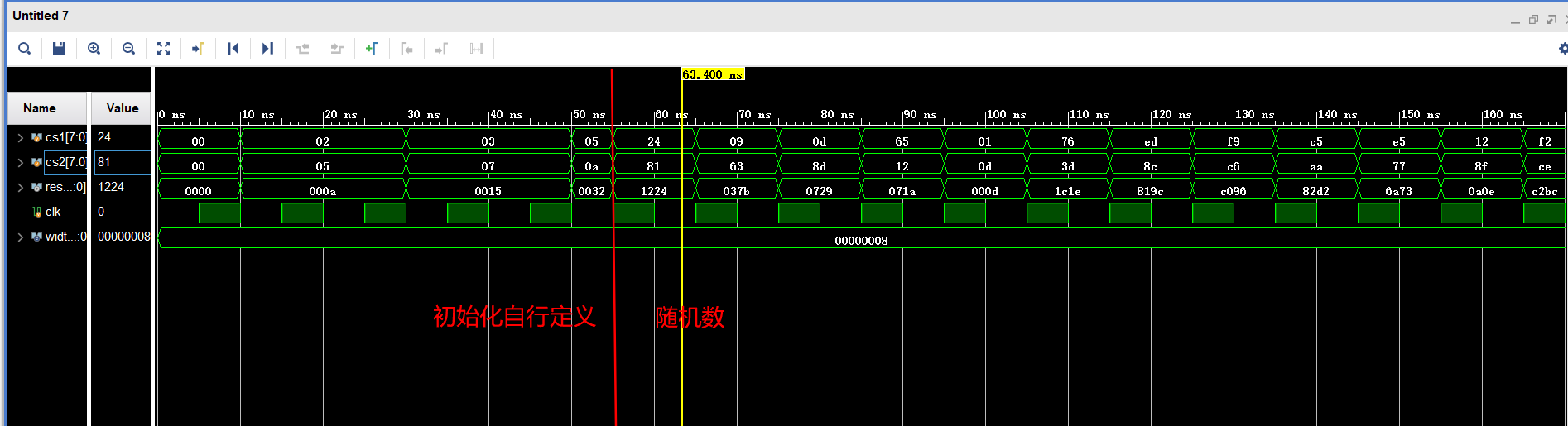
module cfqsimu;  
parameter width = 8;  
reg [width:1] a,b;  
wire [2\*width-1:0] result;  
cfq sl(.a(a),.b(b),.result(result));  
initial  
 begin  
 a=8'b00000010;b=8'b00000000;  
 end  
always #2 assign {a,b} = {a,b} + 1;  
endmodule



说明：初始化时a为1，b为0，自增时需等到b增为ff，a才会继续增1，虽然仿真无误，但是在正确性的表现上并不十分直观。

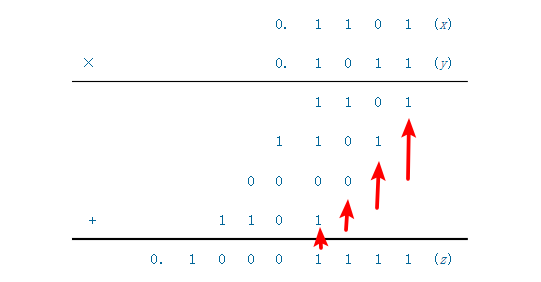
而采用如下Testbench，则会在一开始定义三组数据，随后自行产生随机数验证，在波形图中也更加直观

module cfqsimu;  
parameter width=8;  
reg [width-1:0] cs1;  
reg [width-1:0] cs2;  
wire [2\*width-1:0] result;  
reg clk;  
initial  
begin  
 clk=0;  
 cs1=0;  
 cs2=0;  
 #10  
 cs1=2;  
 cs2=5;  
 #20  
 cs1=3;  
 cs2=7;  
 #20  
 cs1=5;  
 cs2=10;   
 repeat(20) @ (posedge clk)  
 begin  
 cs1[width-1:0]<=($random);  
 cs2[width-1:0]<=($random);  
 end  
end  
always #5 clk=~clk;  
cfq ul(.a(cs1),.b(cs2),.result(result));  
endmodule



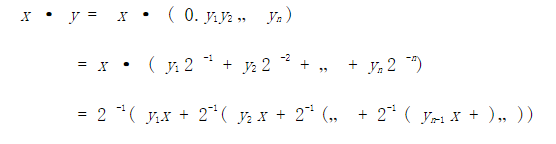
# 右移位乘法器

由于左移位乘法器在计算时不断将乘得的数字进行左移做加法，这时会因为左移增大数字计算的内存，而通过部分积的方式，将最后一位不参与加法的数字右移单独存起，相较于左移位乘法器，大大的节省了运算空间。

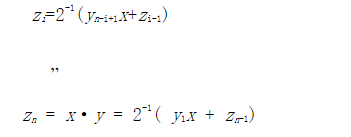


由图可以看出，最后一位的结果并不参与移位相加的运算，可直接右移转存，则部分积占用的空间显然比左移运算小多了。

具体理论思路如下：x=x1x2x3....xn，y=y1y2y3....yn，



再另zi表示第i次的部分积，上式可整理成下式

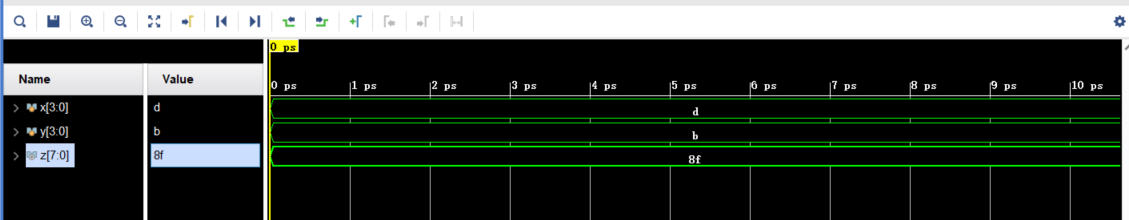


## 设计文件

//右移乘法器  
module cfq(z,x,y);  
 input [3:0] x;  
 input [3:0] y;  
 output [7:0] z;  
 reg [4:0] A; //部分积  
 integer i;  
 reg [7:0] z;  
 reg [3:0] temp\_x; //乘法运算时，被乘数与乘数最后一位的积的最后一位不参与最后的加法运算，将其右移另存。  
 always @ (\*)  
 begin  
 A=0;  
 for (i=0;i<4;i=i+1)  
 begin  
 if(y[i]) A=A+{1'b0,x}; //部分积移位前可能会比乘数多一位  
 else A=A+5'b00000;  
 temp\_x[i]=A[0];  
 A=A>>1;  
 end  
 z={A,temp\_x};  
 end   
endmodule

## 仿真文件

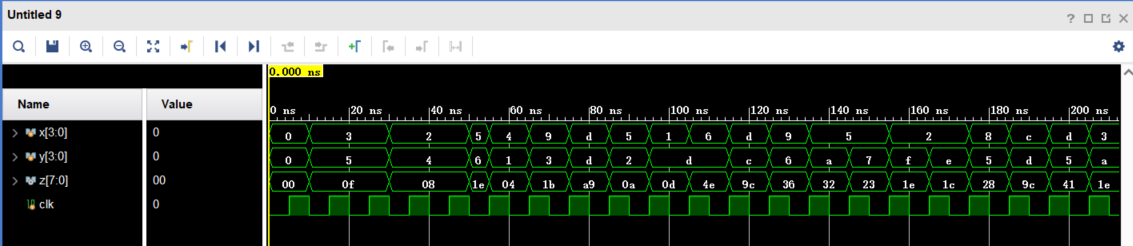
//右移  
module cfqsimu;  
reg [3:0] x,y;  
 wire [7:0] z;  
initial  
 begin  
 x=4'b1101;y=4'b1011; //x=1101;y=1011;  
 end  
 cfq sl(.x(x),.y(y),.z(z));  
endmodule



为了防止运算结果的偶然性，将仿真文件稍作修改：

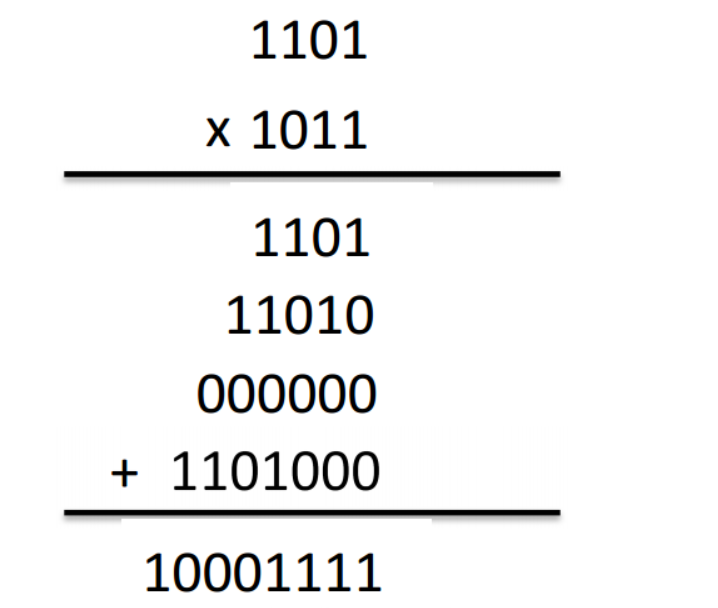
module cfqsimu;  
 reg [3:0] x;  
 reg [3:0] y;  
 wire [7:0] z;  
 reg clk;  
 initial  
 begin  
 clk=0;  
 x=0;  
 y=0;  
 #10  
 x=3;  
 y=5;  
 #20  
 x=2;  
 y=4;  
 #20   
 x=5;  
 y=6;  
 repeat (20) @(posedge clk)  
 begin  
 x={$random};  
 y={$random};  
 end  
 end  
 always #5 clk=~clk;  
 cfq ul(.z(z),.x(x),.y(y));  
endmodule

仿真波形如下，可见结果正确。



# 加法树乘法器

与单纯的左右移位乘法器不同，加法树乘法器是将每一位乘得的数后面补充0至最后一位，再直接利用加法来计算，参考下图：

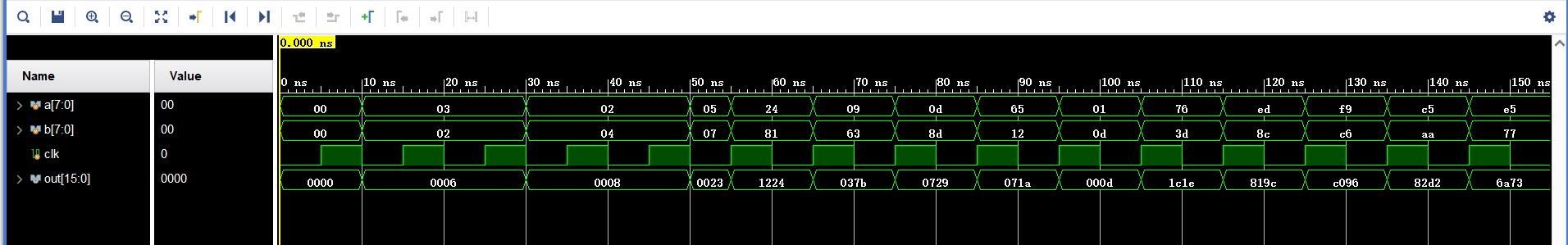


## 设计文件

module cfq(out,a,b,clk);  
output [15:0] out;  
input [7:0] a,b;  
input clk;  
wire [15:0] out;  
wire [15:0] out1，c1;  
wire [13:0] out2;  
wire [11:0] out3;  
wire [12:0] c2;  
wire [9:0] out4;  
reg [14:0] temp0 = 0;  
reg [13:0] temp1 = 0;  
reg [12:0] temp2 = 0;  
reg [11:0] temp3 = 0;  
reg [10:0] temp4 = 0;  
reg [9:0] temp5 = 0;  
reg [8:0] temp6 = 0;  
reg [7:0] temp7 = 0;  
  
function [7:0] mux8\_1; //被乘数与乘数的对应位相乘得到的积函数  
input [7:0] operand;  
input sel;  
begin  
 mux8\_1=(sel)?(operand):8'b00000000;  
end   
endfunction  
always @(\*)  
begin  
 temp7<=mux8\_1(a,b[0]); //把每一次乘得的数移位填零赋值给temp\_i  
 temp6<=((mux8\_1(a,b[1]))<<1);  
 temp5<=((mux8\_1(a,b[2]))<<2);  
 temp4<=((mux8\_1(a,b[3]))<<3);  
 temp3<=((mux8\_1(a,b[4]))<<4);  
 temp2<=((mux8\_1(a,b[5]))<<5);  
 temp1<=((mux8\_1(a,b[6]))<<6);  
 temp0<=((mux8\_1(a,b[7]))<<7);  
end  
assign out1 = temp0 + temp1; //两个相近位数的结果相加，可据此判断out\_i的位数  
assign out2 = temp2 + temp3;  
assign out3 = temp4 + temp5;  
assign out4 = temp6 + temp7;  
assign c1 = out1 + out2;  
assign c2 = out3 + out4;  
assign out = c1 + c2;  
endmodule

## 仿真文件

module cfqsimu;  
reg [7:0] a,b;  
reg clk;  
wire [15:0] out;  
cfq sl(.a(a),.b(b),.out(out),.clk(clk));  
initial  
begin  
 clk=0;a=0;b=0;  
 #10  
 a=3;b=2;  
 #20  
 a=2;b=4;  
 #20  
 a=5;b=7;  
 repeat(20) @(posedge clk)  
 begin  
 a={$random};  
 b={$random};  
 end   
end  
always #5 clk=~clk;  
endmodule



说明：如果将上述设计代码中的wire类型所对应的变量全部改为reg类型，并对其延时赋值，就相当于加法器之间插入寄存器以实现流水线，从而再次提高时钟频率。

# 查找表乘法器

查找表乘法器是将乘积直接存放在存储器中，将操作数作为地址访问存储器，得到的输出数据就是乘法器的结果，其速度只取决于存储器的存取速度。

## 思路

首先欲利用操作数地址访问存储器，应当想到将操作数拆分，因为如果不拆分的话，就直接合并只能得到一个地址，只能查找到一个结果，下面以1101 x 1011为例，首先想到的拆分应该是从中间位置拆分，1101拆分为11和01，1011拆分为10和11，再通过拆分得到的部分想办法与结果联系起来，即：1101 x 1011 = （1100 + 0001）x（1000 + 0011），再因式分解一下，得到1100 x 1000 + 1100 x 0011 + 0001 x 1000 + 0001 x 0011，另firsta=11，seconda=01，firstb=10，secondb=11，上述结果又可表示为((firsta x firstb)<<4) + ((firsta x secondb)<<2) + ((seconda x firstb)<<2) + (seconda x secondb),每部分移位的位数也就是应该放大的倍数，自行理解就好。

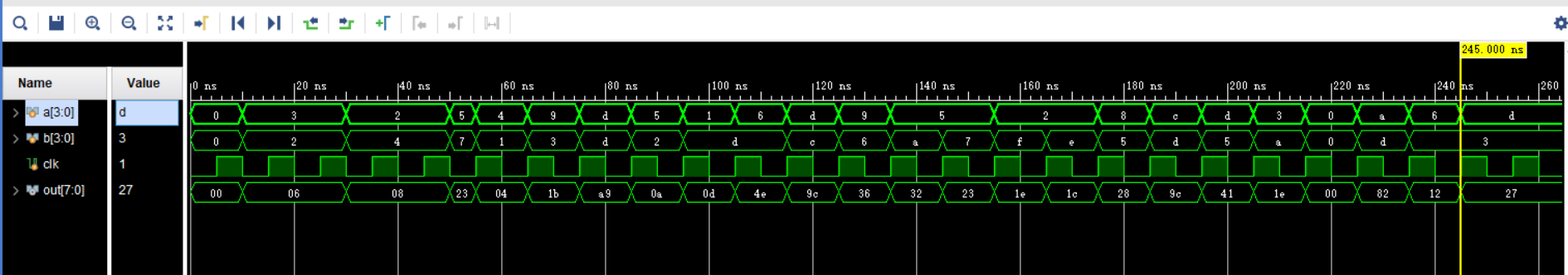
## 设计文件

module cfq(out,a,b,clk);  
output [7:0] out;  
input [3:0] a,b;  
input clk;  
reg [7:0] out;  
reg [1:0] firsta,firstb;   
reg [1:0] seconda,secondb;  
wire [3:0] outa,outb,outc,outd; //将上面说的因式分解的各个部分分别储存  
always @(\*)  
begin  
 firsta = a[3:2]; seconda = a[1:0]; //将a，b从中间分成两部分  
 firstb = b[3:2]; secondb = b[1:0];  
end  
  
lookup m1(outa,firsta,firstb,clk), //引用lookup模块  
 m2(outb,firsta,secondb,clk),  
 m3(outc,seconda,firstb,clk),  
 m4(outd,seconda,secondb,clk);  
  
always @(\*)  
begin  
 out = (outa<<4) + (outb<<2) + (outc<<2) + (outd);  
end  
endmodule

module lookup(out,a,b,clk); //新建设计文件用于引用  
output [3:0] out;  
input [1:0] a,b;  
input clk;  
reg [3:0] out;  
reg [3:0] address;  
always @(\*)  
begin  
 address = {a,b}; //将拆分的a，b组成的地址作为索引，注意：与乘数因子含义不同  
 case(address)  
 4'h0: out =4'b0000;  
 4'h1: out =4'b0000;  
 4'h2: out =4'b0000;  
 4'h3: out =4'b0000;  
 4'h4: out =4'b0000;  
 4'h5: out =4'b0001;  
 4'h6: out =4'b0010;  
 4'h7: out =4'b0011;  
 4'h8: out =4'b0000;  
 4'h9: out =4'b0010;  
 4'ha: out =4'b0100;  
 4'hb: out =4'b0110;  
 4'hc: out =4'b0000;  
 4'hd: out =4'b0011;  
 4'he: out =4'b0110;  
 4'hf: out =4'b1001; //拆分之后前半部分和后半部分各四种可能，分别为00，01，10，11，相乘即有16种可能，分别分析做出case事件  
 default: out = 'bx;  
 endcase  
end  
endmodule

## 仿真文件

module cfqsimu;  
reg [3:0] a,b;  
reg clk;  
wire [7:0] out;  
cfq sl(.a(a),.b(b),.out(out),.clk(clk));  
initial  
begin  
 clk=0;a=0;b=0;  
 #10  
 a=3;b=2;  
 #20  
 a=2;b=4;  
 #20  
 a=5;b=7;  
 repeat(20) @(posedge clk)  
 begin  
 a={$random};  
 b={$random};  
 end   
end  
always #5 clk=~clk;  
endmodule



# 布尔乘法器