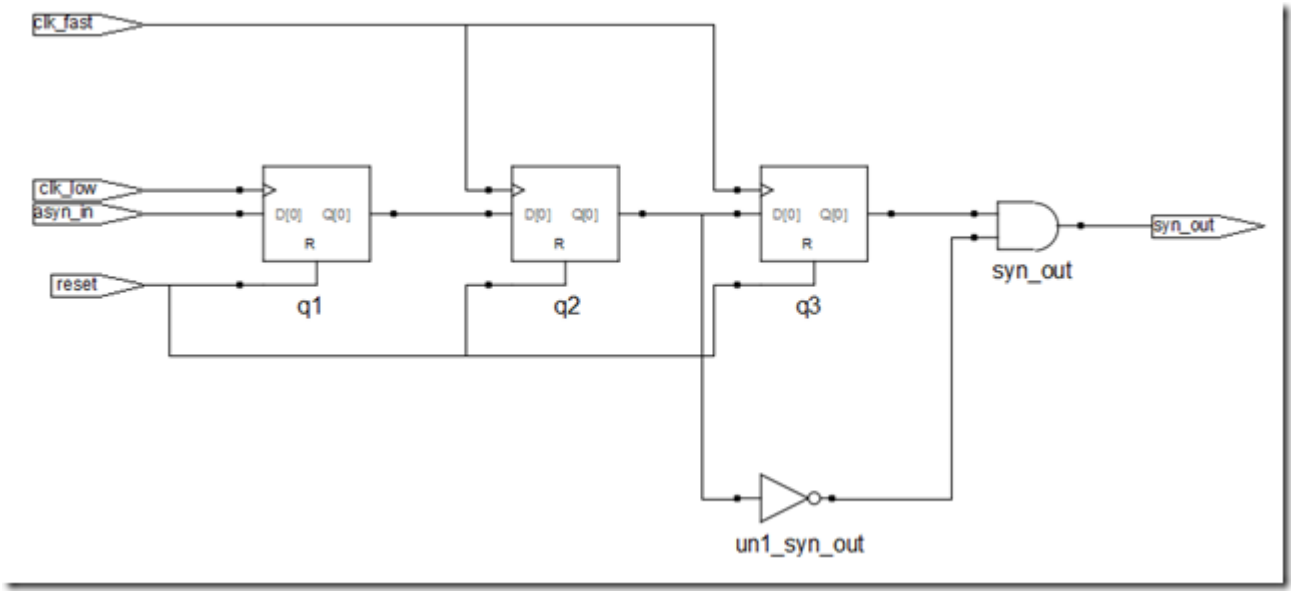
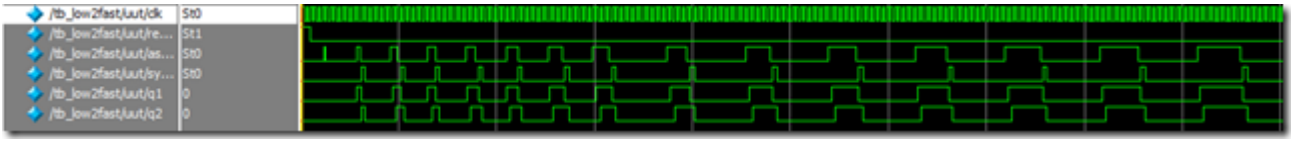
多时钟域数据同步

对于不同的时钟域要传递数据的话，需要采用一定的手段，来防止数据传递时产生亚稳态等问题

1、慢时钟域向快时钟域传递数据

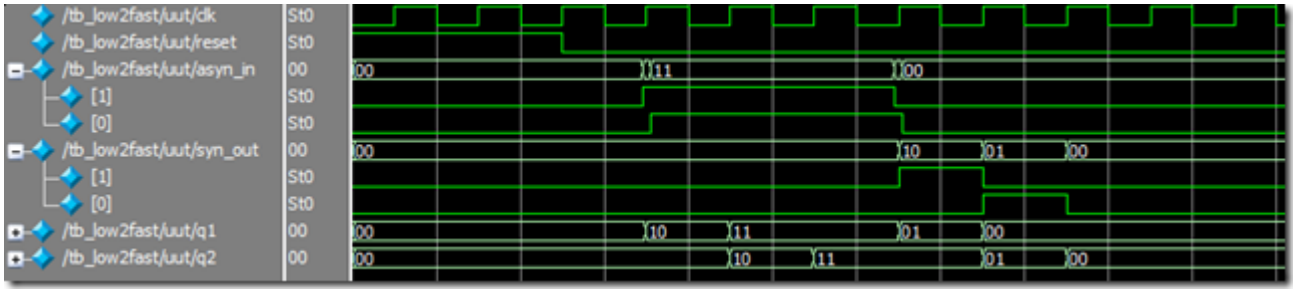
```
module low2fast(clk,reset,asyn_in,syn_out,);
input clk,reset;
input asyn_in;
output syn_out;

reg q1,q2;
always@(posedge clk or posedge reset)
if(reset)
            q1<=2'b0;
else
            q1<=asyn_in;

always@(posedge clk or posedge reset)
if(reset)
            q2<=2'b0;
else
            q2<=q1;

assign syn_out=!q1 & q2;

endmodule
```





对于持续时间较短的脉冲一般情况下无法捕捉到，只有脉冲宽度较时钟周期大才可被捕捉到，该电路实际上我一般用来作为控制信号的边沿检测，此处为下降沿检测，把反相器放到q3的输出即可用来检测上升沿。

但是需要注意的是该方法只适用于单个数据的同步，如果是多位数据的话可能会出现问题：

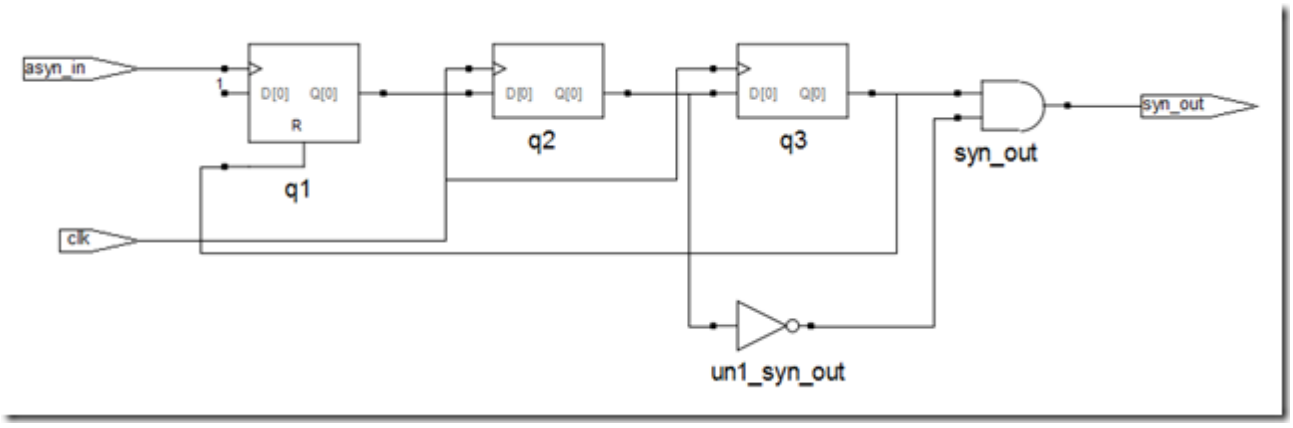


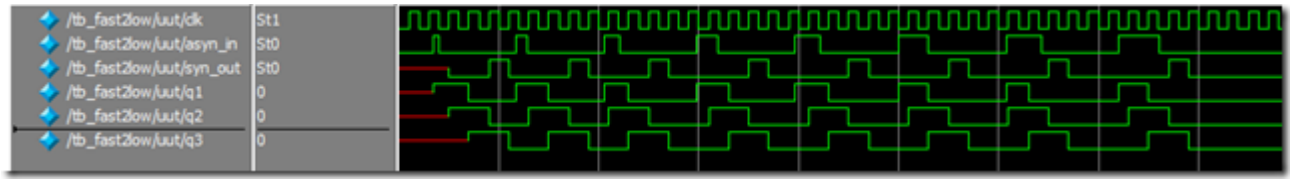
多路数据传输的时候由于不同数据的路径不一样，因此到达寄存器输入端的时间不同，如果一个数据的到达时间满足建立时间而例外一个不满足寄存器的建立时间，则会导致数据与数据间相差一个或多个时钟。

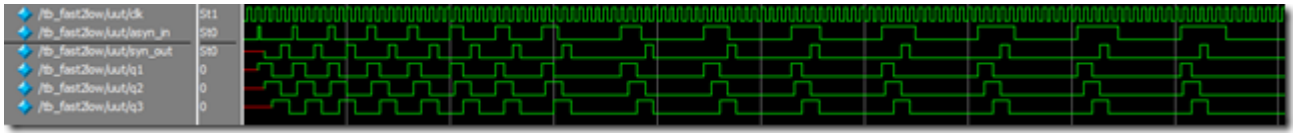
2、快时钟域向慢时钟域传递数据

对于单个的数据:锁存反馈法

```
module fast2low(clk,asyn_in,syn_out);//pulse asyn_in is short than one clock time

input clk;
input asyn_in;
output syn_out;

reg q1,q2,q3;
always@(posedge asyn_in or posedge q3)
if(q3)
            q1<=0;
else
            q1<=1;

always@(posedge clk)
            q2<=q1;

always@(posedge clk)
            q3<=q2;

assign syn_out=!q2 & q3;

endmodule
```







可见在同步电路的作用下，较窄的脉冲也可以被电路捕捉到。

在不同时钟域间传递数据一般不采用以上方法，因为多位数据传递时会使同步器的采样率错误率大大增加，一般采用的是异步FIFO.

跨时钟域时可以采用的方法：

1 如果时钟间存在着固定的频率倍数，这种情况下它们的相位一般具有固定关系，可以采用下 述方法处理：

1）使用高频时钟作为工作时钟，使用低频时钟作为使能信号，当功耗不作为首要因素时建议使用这种方式。

2）在仔细分析时序的基础上描述两个时钟转换处的电路。

2 如果电路中存在两个不同频率的时钟，并且频率无关，可以采用如下策略：

1）利用高频时钟采样两个时钟，在电路中使用高频时钟作为电路的工作时钟，经采后的低频时钟作为使能。

2）在时钟同步单元中采用两次同步法

3）使用握手信号

4）使用双时钟FIFO 进行数据缓冲

在构件由两个不同系统时钟控制工作的模块之间的同步模块时，应该遵守下面原则：两个采用不同时钟工作的寄存器之间不应该再出现逻辑电路，而应该仅仅是一种连接关系,这种方法有利于控制建立保持时间的满足。

verilog 不可综合语句 总结 汇总

（1）所有综合工具都支持的结构：always, assign, begin, end, case, wire, tri, aupply0, supply1, reg, integer, default, for, function, and, nand, or, nor, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1, if, inout, input, instantitation, module, negedge, posedge, operators, output, parameter。

（2）所有综合工具都不支持的结构：time, defparam, $finish, fork, join, initial, delays, UDP, wait。

(3)有些工具支持有些工具不支持的结构:casex, casez, wand, triand, wor, trior, real, disable, forever, arrays, memories, repeat, task, while。

建立可综合模型的原则

要保证Verilog HDL赋值语句的可综合性，在建模时应注意以下要点：

(1)不使用initial。

(2)不使用#10。

(3)不使用循环次数不确定的循环语句，如forever、while等。

(4)不使用用户自定义原语(UDP元件)。

(5)尽量使用同步方式设计电路。

(6)除非是关键路径的设计，一般不采用调用门级元件来描述设计的方法，建议采用行为语句来完成设计。

(7)用always过程块描述组合逻辑，应在敏感信号列表中列出所有的输入信号。

(8)所有的内部寄存器都应该能够被复位，在使用FPGA实现设计时，应尽量使用器件的全局复位端作为系统总的复位。

(9)对时序逻辑描述和建模，应尽量使用非阻塞赋值方式。对组合逻辑描述和建模，既可以用阻塞赋值，也可以用非阻塞赋值。但在同一个过程块中，最好不要同时用阻塞赋值和非阻塞赋值。

(10)不能在一个以上的always过程块中对同一个变量赋值。而对同一个赋值对象不能既使用阻塞式赋值，又使用非阻塞式赋值。

(11)如果不打算把变量推导成锁存器，那么必须在if语句或case语句的所有条件分支中都对变量明确地赋值。

(12)避免混合使用上升沿和下降沿触发的触发器。

(13)同一个变量的赋值不能受多个时钟控制，也不能受两种不同的时钟条件(或者不同的时钟沿)控制。

(14)避免在case语句的分支项中使用x值或z值。

不可综合verilog语句

1、initial

只能在test bench中使用，不能综合。(我用ISE9.1综合时，有的简单的initial也可以综合，不知道为什么)

2、events

event在同步test bench时更有用，不能综合。

3、real

不支持real数据类型的综合。

4、time

不支持time数据类型的综合。

5、force 和release

不支持force和release的综合。

6、assign 和deassign

不支持对reg 数据类型的assign或deassign进行综合，支持对wire数据类型的assign或deassign进行综合。

7、fork join

不可综合，可以使用非块语句达到同样的效果。

8、primitives

支持门级原语的综合，不支持非门级原语的综合。

9、table

不支持UDP 和table的综合。

10、敏感列表里同时带有posedge和negedge

如:always @(posedge clk or negedge clk) begin...end

这个always块不可综合。

11、同一个reg变量被多个always块驱动

12、延时

以#开头的延时不可综合成硬件电路延时，综合工具会忽略所有延时代码，但不会报错。

如:a=#10 b;

这里的#10是用于仿真时的延时，在综合的时候综合工具会忽略它。也就是说，在综合的时候上式等同于a=b;

13、与X、Z的比较

可能会有人喜欢在条件表达式中把数据和X(或Z)进行比较，殊不知这是不可综合的，综合工具同样会忽略。所以要确保信号只有两个状态:0或1。

如:

```
module synthesis_compare_xz (a,b);
output a;
input b;
reg a;

always @ (b)
 begin
  if ((b == 1'bz) || (b == 1'bx))
   begin
    a = 1;
   end
  else
   begin
     a = 0;
   end
 end

endmodule
```

流水线技术原理和Verilog HDL实现

所谓流水线处理，如同生产装配线一样，将操作执行工作量分成若干个时间上均衡的操作段，从流水线的起点连续地输入，流水线的各操作段以重叠方式执行。这使得操作执行速度只与流水线输入的速度有关，而与处理所需的时间无关。这样，在理想的流水操作状态下，其运行效率很高。

如果某个设计的处理流程分为若干步骤，而且整个数据处理是单流向的，即没有反馈或者迭代运算，前一个步骤的输出是下一个步骤的输入，则可以采用流水线设计方法来提高系统的工作频率。

下面用8位全加器作为实例，分别列举了非流水线方法、2级流水线方法和4级流水线方法。

（1）非流水线实现方式

```
module adder_8bits(din_1, clk, cin, dout, din_2, cout);
    input [7:0] din_1;
    input clk;
    input cin;
```

```
    output [7:0] dout;
    input [7:0] din_2;
    output cout;

            reg [7:0] dout;
            reg       cout;

            always @(posedge clk) begin
                        {cout,dout} <= din_1 + din_2 + cin;
            end

endmodule
```

(2)2级流水线实现方式:

```
module adder_4bits_2steps(cin_a, cin_b, cin, clk, cout, sum);
    input [7:0] cin_a;
    input [7:0] cin_b;
    input cin;
    input clk;
    output cout;
    output [7:0] sum;

            reg cout;
            reg cout_temp;
            reg [7:0] sum;
            reg [3:0] sum_temp;

            always @(posedge clk) begin
                        {cout_temp,sum_temp} = cin_a[3:0] + cin_b[3:0] + cin;
            end

            always @(posedge clk) begin
                        {cout,sum} = {{1'b0,cin_a[7:4]} + {1'b0,cin_b[7:4]} + cout_temp, sum_temp};
            end
endmodule
```

注意：这里在always块内只能用阻塞赋值方式，否则会出现逻辑上的错误！

(3)4级流水线实现方式:

```
module adder_8bits_4steps(cin_a, cin_b, c_in, clk, c_out, sum_out);
    input [7:0] cin_a;
    input [7:0] cin_b;
    input c_in;
    input clk;
    output c_out;
    output [7:0] sum_out;
```

```
        reg c_out;
        reg c_out_t1, c_out_t2, c_out_t3;

        reg [7:0] sum_out;
        reg [1:0] sum_out_t1;
        reg [3:0] sum_out_t2;
        reg [5:0] sum_out_t3;
        always @(posedge clk) begin
                {c_out_t1, sum_out_t1} = {1'b0, cin_a[1:0]} + {1'b0, cin_b[1:0]} + c_in;
        end

        always @(posedge clk) begin
                {c_out_t2, sum_out_t2} = {{1'b0, cin_a[3:2]} + {1'b0, cin_b[3:2]} + c_out_t1,
sum_out_t1};
        end
        always @(posedge clk) begin
                {c_out_t3, sum_out_t3} = {{1'b0, cin_a[5:4]} + {1'b0, cin_b[5:4]} + c_out_t2,
sum_out_t2};
        end
        always @(posedge clk) begin
                {c_out, sum_out} = {{1'b0, cin_a[7:6]} + {1'b0, cin_b[7:6]} + c_out_t3, sum_out_t3};
        end
endmodule
```

总结:利用流水线的设计方法，可大大提高系统的工作速度。这种方法可广泛运用于各种设计，特别是大型的、对速度要求较高的系统设计。虽然采用流水线会增大资源的使用，但是它可降低寄存器间的传播延时，保证系统维持高的系统时钟速度。在实际应用中，考虑到资源的使用和速度的要求，可以根据实际情况来选择流水线的级数以满足设计需要。

这是一种典型的以面积换速度的设计方法。这里的"面积"主要是指设计所占用的FPGA逻辑资源数目，即利用所消耗的触发器(FF)和查找表(LUT)来衡量。"速度"是指在芯片上稳定运行时所能达到的最高频率。面积和速度这两个指标始终贯穿着FPGA的设计，是设计质量评价的最终标准。

乘法器的Verilog HDL实现

1. 串行乘法器

两个N位二进制数x、y的乘积用简单的方法计算就是利用移位操作来实现。

```
module multi_CX(clk, x, y, result);

    input clk;
    input [7:0] x, y;
    output [15:0] result;

    reg [15:0] result;

    parameter s0 = 0, s1 = 1, s2 = 2;
```

```
    reg [2:0] count = 0;
    reg [1:0] state = 0;
    reg [15:0] P, T;
    reg [7:0] y_reg;

    always @(posedge clk) begin
        case (state)
            s0: begin
                count <= 0;
                P <= 0;
                y_reg <= y;
                T <= {{8{1'b0}}, x};
                state <= s1;
            end
            s1: begin
                if(count == 3'b111)
                    state <= s2;
                else begin
                    if(y_reg[0] == 1'b1)
                        P <= P + T;
                    else
                        P <= P;
                    y_reg <= y_reg >> 1;
                    T <= T << 1;
                    count <= count + 1;
                    state <= s1;
                end
            end
            s2: begin
                result <= P;
                state <= s0;
            end
            default: ;
        endcase
    end

endmodule
```

乘法功能是正确的，但计算一次乘法需要8个周期。因此可以看出串行乘法器速度比较慢、时延大，但这种乘法器的优点是所占用的资源是所有类型乘法器中最少的，在低速的信号处理中有着广泛的应用。

2.流水线乘法器

一般的快速乘法器通常采用逐位并行的迭代阵列结构，将每个操作数的N位都并行地提交给乘法器。但是一般对于FPGA来讲，进位的速度快于加法的速度，这种阵列结构并不是最优的。所以可以采用多级流水线的形式，将相邻的两个部分乘积结果再加到最终的输出乘积上，即排成一个二叉树形式的结构，这样对于N位乘法器需要lb(N)级来实现。

```
module multi_4bits_pipelining(mul_a, mul_b, clk, rst_n, mul_out);

    input [3:0] mul_a, mul_b;
    input       clk;
    input       rst_n;
    output [7:0] mul_out;

    reg [7:0] mul_out;

    reg [7:0] stored0;
    reg [7:0] stored1;
    reg [7:0] stored2;
    reg [7:0] stored3;

    reg [7:0] add01;
    reg [7:0] add23;

    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            mul_out <= 0;
            stored0 <= 0;
            stored1 <= 0;
            stored2 <= 0;
            stored3 <= 0;
            add01 <= 0;
            add23 <= 0;
        end
        else begin
            stored0 <= mul_b[0]? {4'b0, mul_a} : 8'b0;
            stored1 <= mul_b[1]? {3'b0, mul_a, 1'b0} : 8'b0;
            stored2 <= mul_b[2]? {2'b0, mul_a, 2'b0} : 8'b0;
            stored3 <= mul_b[3]? {1'b0, mul_a, 3'b0} : 8'b0;

            add01 <= stored1 + stored0;
            add23 <= stored3 + stored2;

            mul_out <= add01 + add23;
        end
    end

endmodule
```

从图中可以看出，流水线乘法器比串行乘法器的速度快很多很多，在非高速的信号处理中有广泛的应用。至于高速信号的乘法一般需要利用FPGA芯片中内嵌的硬核DSP单元来实现。