影响 FPGA 设计中时钟因素的探讨

时钟是整个电路最重要、最特殊的信号，系统内大部分器件的动作都是在时钟的跳变沿上进行，这就要求时钟信号时延差要非常小，否则就可能造成时序逻辑状态出错；因而明确FPGA设计中决定系统时钟的因素，尽量较小时钟的延时对保证设计的稳定性有非常重要的意义。

1、1 建立时间与保持时间

建立时间（Tsu：set up time）是指在时钟沿到来之前数据从不稳定到稳定所需的时间，如果建立的时间不满足要求那么数据将不能在这个时钟上升沿被稳定的打入触发器；保持时间（Th：hold time）是指数据稳定后保持的时间，如果保持时间不满足要求那么数据同样也不能被稳定的打入触发器。建立与保持时间的简单示意图如下图 1 所示。

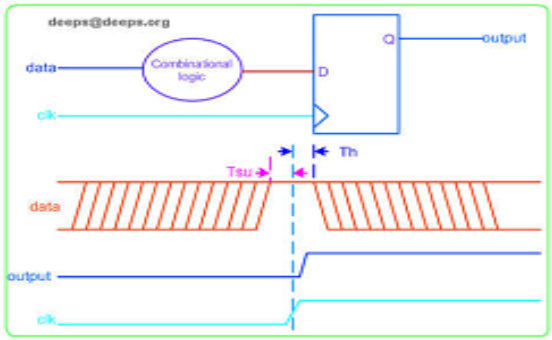

图 1 保持时间与建立时间的示意图

在 FPGA 设计的同一个模块中常常是包含组合逻辑与时序逻辑，为了保证在这些逻辑的接口处数据能稳定的被处理，那么对建立时间与保持时间建立清晰的概念非常重要。下面在认识了建立时间与保持时间的概念上思考如下的问题。

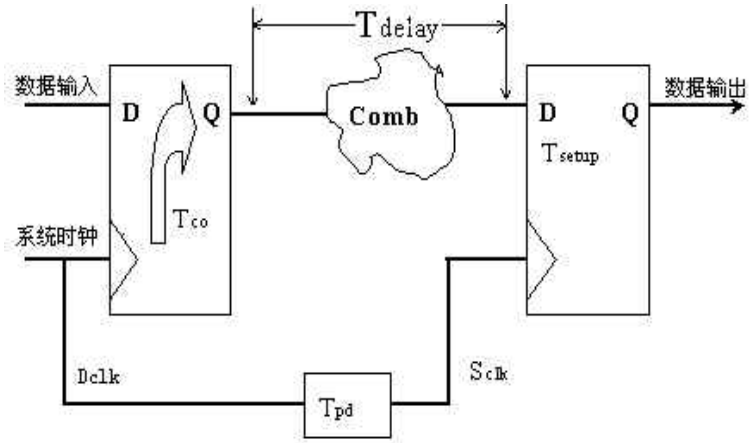

图 2 同步设计中的一个基本模型

图2为统一采用一个时钟的同步设计中一个基本的模型。图中Tco是触发器的数据输出

的延时；Tdelay是组合逻辑的延时；Tsetup是触发器的建立时间；Tpd ,为时钟的延时。如果第一个触发器D1建立时间最大为T1max，最小为T1min，组合逻辑的延时最大为T2max，最小为T2min。问第二个触发器D2立时间T3与保持时间T4应该满足什么条件，或者是知道了T3与T4那么能容许的最大时钟周期是多少。这个问题是在设计中必须考虑的问题，只有弄清了这个问题才能保证所设计的组合逻辑的延时是否满足了要求。

下面通过时序图来分析：设第一个触发器的输入为D1，输出为Q1,第二个触发器的输入为D2，输出为Q2；

时钟统一在上升沿进行采样，为了便于分析我们讨论两种情况即第一：假设时钟的延时Tpd为零，其实这种情况在FPGA设计中是常常满足的，由于在FPGA设计中一般是采用统一的系统时钟，也就是利用从全局时钟管脚输入的时钟，这样在内部时钟的延时完全可以忽略不计。这种情况下不必考虑保持时间，因为每个数据都是保持一个时钟节拍同时又有线路的延时，也就是都是基于CLOCK的延迟远小于数据的延迟基础上，所以保持时间都能满足要求，重点是要关心建立时间，此时如果D2的建立时间满足要求那么时序图应该如图3所示。

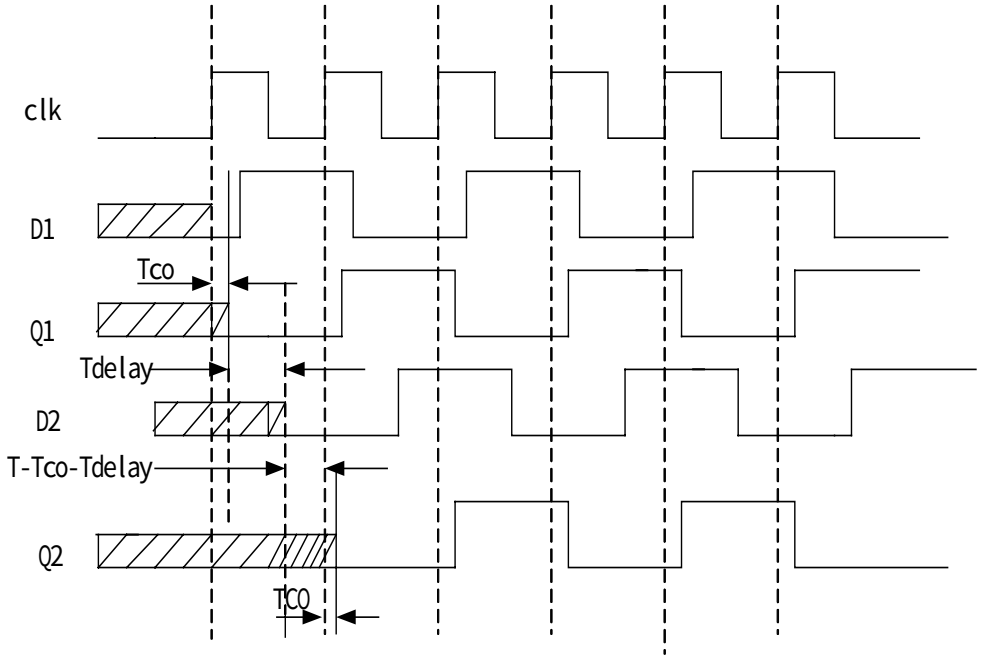

图3 符合要求的时序图

从图中可以看出如果：

$$T\text{-}Tco\text{-}Tdelay > T3$$

即：

$$Tdelay < T\text{-}Tco\text{-}T3$$

那么就满足了建立时间的要求，其中 T 为时钟的周期，这种情况下第二个触发器就能在第二个时钟的升沿就能稳定的采到 D2，时序图如图 3 所示。

如果组合逻辑的延时过大使得

$$T\text{-}Tco\text{-}Tdelay < T3$$

那么将不满足要求，第二个触发器就在第二个时钟的升沿将采到的是一个不定态，如图 4 所示。那么电路将不能正常的工作。

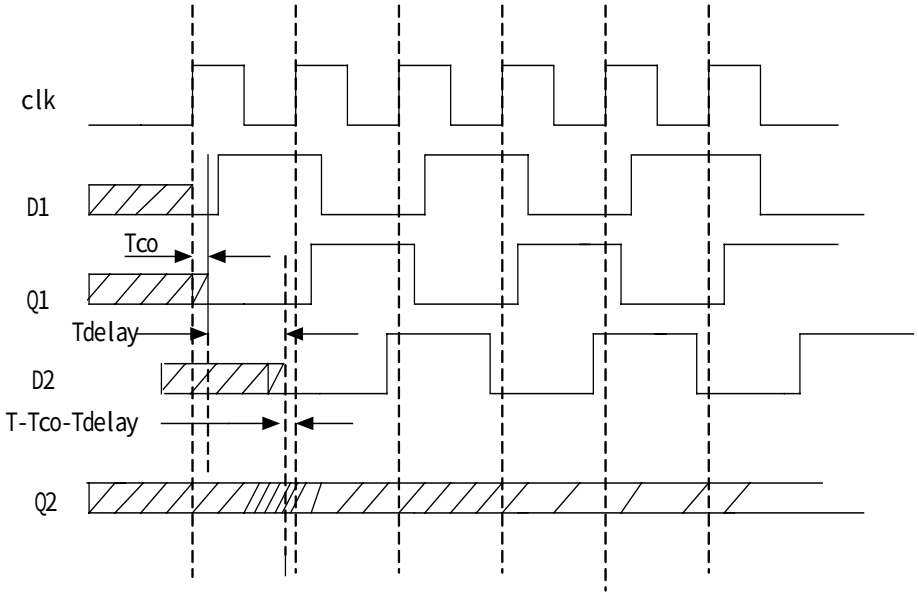

图 4　组合逻辑的延时过大时序不满足要求

从而可以推出

$$T\text{-}Tco\text{-}T2max>=T3$$

这也就是要求的 D2 的建立时间。

从上面的时序图中也可以看出，D2 的建立时间与保持时间与 D1 的建立与保持时间是没有关系的，而只和 D2 前面的组合逻辑和 D1 的数据传输延时有关，这也是一个很重要的结论。说明了延时没有叠加效应。

第二种情况如果时钟存在延时，这种情况下就要考虑保持时间了，同时也需要考虑建立时间。时钟出现较大的延时多是采用了异步时钟的设计方法，这种方法较难保证数据的同步性，所以实际的设计中很少采用。此时如果建立时间与保持时间都满足要求那么输出的时序如图 5 所示。

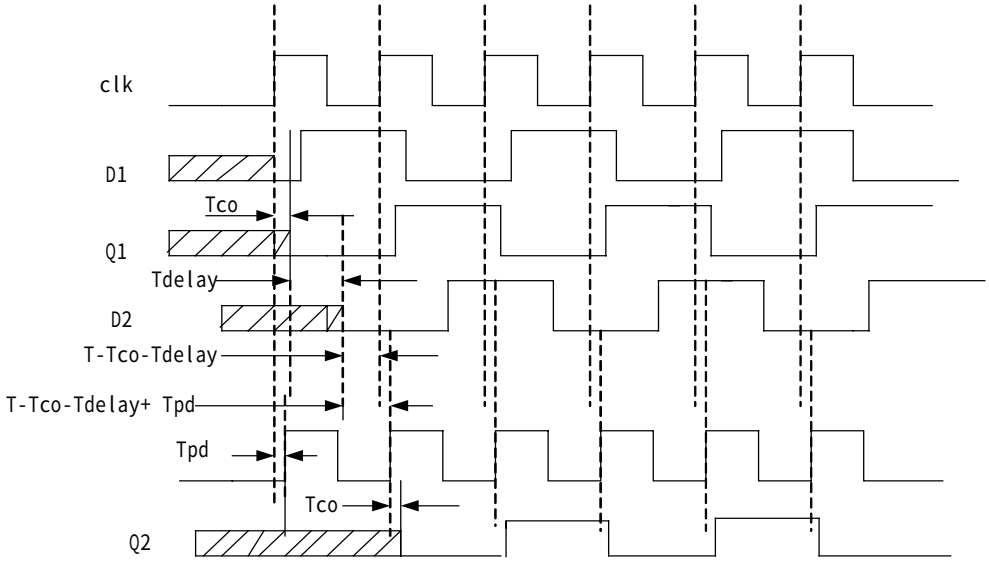

图 5　时钟存在延时但满足时序

从图 5 中可以容易的看出对建立时间放宽了 Tpd，所以 D2 的建立时间需满足要求：

$$Tpd+T\text{-}Tco\text{-}T2max>=T3$$

由于建立时间与保持时间的和是稳定的一个时钟周期，如果时钟有延时，同时数据的延时也较小那么建立时间必然是增大的，保持时间就会随之减小，如果减小到不满足 D2 的保持时间要求时就不能采集到正确的数据，如图 6 所示。

这时即 T－（Tpd－Tco-T2min）<T4,就不满足要求了，所以 D2 的保持时间应该为：

T－（Tpd+T－Tco-T2min）>＝T4　　即 Tco＋T2min-Tpd>＝T4

从上式也可以看出如果 Tpd＝0 也就是时钟的延时为 0 那么同样是要求 Tco＋T2min>T4，但是在实际的应用中由于 T2 的延时也就是线路的延时远远大于触发器的保持时间即 T4 所以不必要关系保持时间。

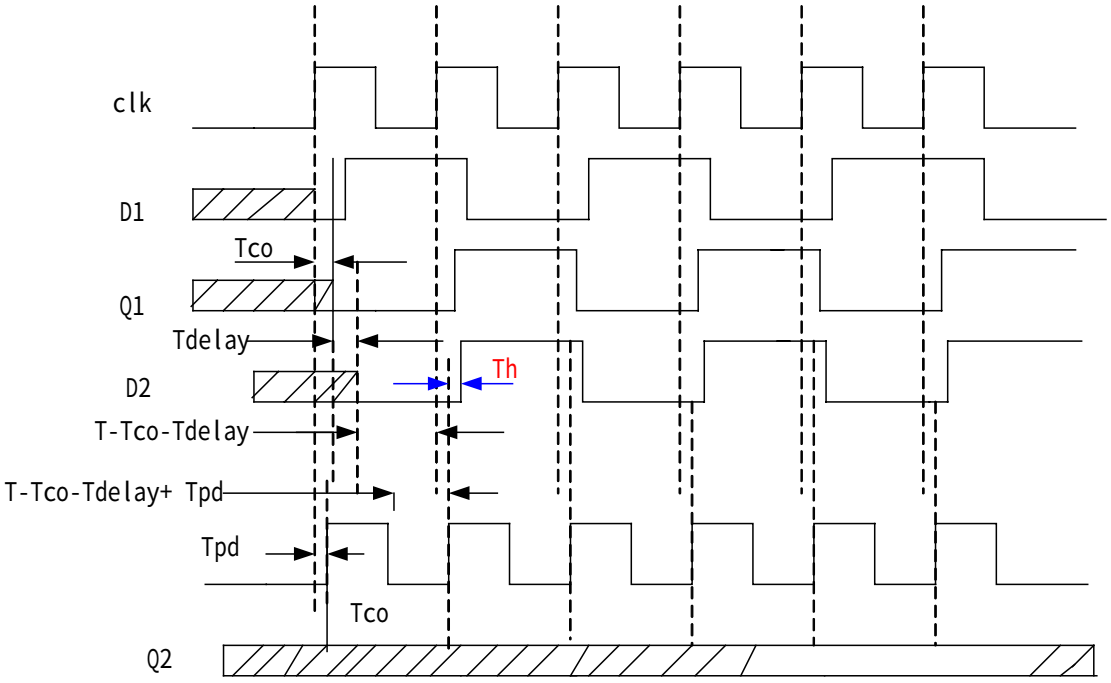

图 6　时钟存在延时且保持时间不满足要求

综上所述，如果不考虑时钟的延时那么只需关心建立时间，如果考虑时钟的延时那么更需关心保持时间。下面将要分析在 FPGA 设计中如何提高同步系统中的工作时钟。

1.2　如何提高同步系统中的工作时钟

从上面的分析可以看出同步系统时对 D2 建立时间 T3 的要求为：

T-Tco-T2max>＝T3

所以很容易推出 T ＞＝T3+Tco+T2max,其中 T3 为 D2 的建立时间 Tset，T2 为组合逻辑的延时。在一个设计中 T3 和 Tco 都是由器件决定的固定值，可控的也只有 T2 也就时输入端组合逻辑的延时，所以通过尽量来减小 T2 就可以提高系统的工作时钟。为了达到减小 T2 在设计中可以用下面不同的几种方法综合来实现。

1.2.1　通过改变走线的方式来减小延时

以 altera 的器件为例，我们在 quartus 里面的 timing closure floorplan 可以看到有很多条条块块，我们可以将条条块块按行和按列分，每一个条块代表 1 个 LAB，每个 LAB 里有 8 个或者是 10 个 LE。它们的走线时延的关系如下：同一个 LAB 中（最快） ＜ 同列或者

同行 < 不同行且不同列。

我们通过给综合器加适当的约束（约束要适量，一般以加 5%裕量较为合适，比如电路工作在 100Mhz，则加约束加到 105Mhz 就可以了，过大的约束效果反而不好，且极大增加综合时间）可以将相关的逻辑在布线时尽量布的靠近一点，从而减少走线的时延。

1.2.2 通过拆分组合逻辑的方法来减小延时

由于一般同步电路都不止一级锁存（如图 8），而要使电路稳定工作，时钟周期必须满足最大延时要求，缩短最长延时路径，才可提高电路的工作频率。如图 7 所示：我们可以将较大的组合逻辑分解为较小的几块，中间插入触发器，这样可以提高电路的工作频率。这也是所谓“流水线”（pipelining）技术的基本原理。

对于图 8 的上半部分，它时钟频率受制于第二个较大的组合逻辑的延时，通过适当的方法平均分配组合逻辑，可以避免在两个触发器之间出现过大的延时，消除速度瓶颈。

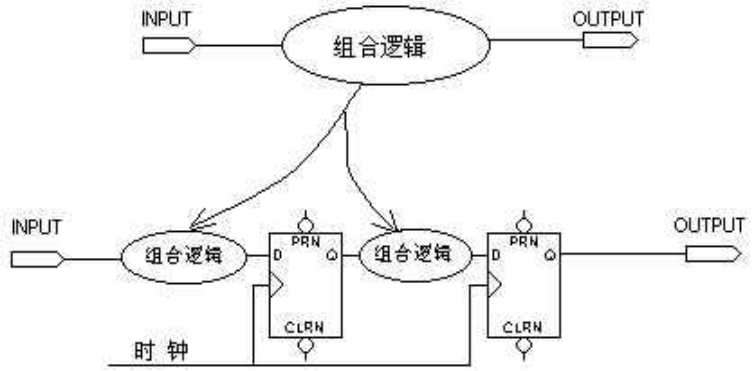

图 7 分割组合逻辑

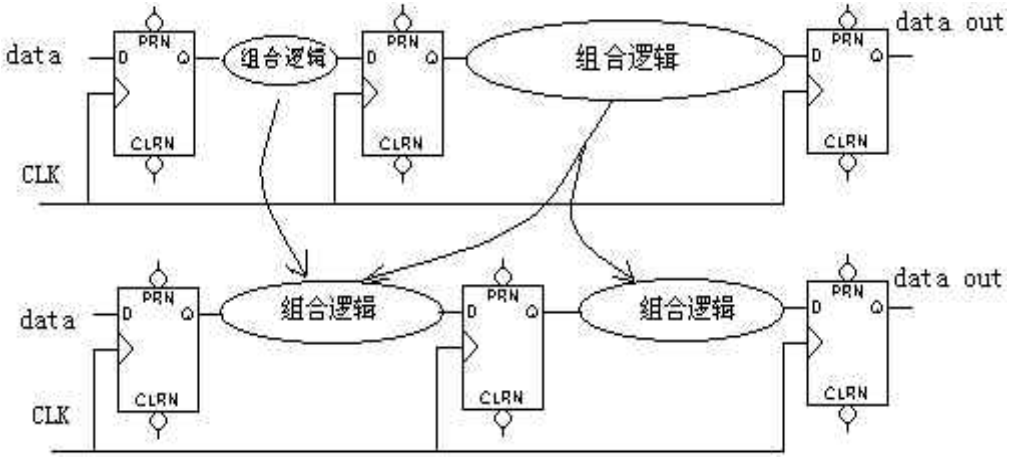

图 8 转移组合逻辑

那么在设计中如何拆分组合逻辑呢，更好的方法要在实践中不断的积累，但是一些良好的设计思想和方法也需要掌握。我们知道，目前大部分 FPGA 都基于 4 输入 LUT 的，如果一个输出对应的判断条件大于四输入的话就要由多个 LUT 级联才能完成，这样就引入一级组合逻辑时延，我们要减少组合逻辑，无非就是要输入条件尽可能的少，这样就可以级联的 LUT 更少，从而减少了组合逻辑引起的时延。

我们平时听说的流水就是一种通过切割大的组合逻辑（在其中插入一级或多级 D 触发

器，从而使寄存器与寄存器之间的组合逻辑减少）来提高工作频率的方法。比如一个 32 位的计数器，该计数器的进位链很长，必然会降低工作频率，我们可以将其分割成 4 位和 8 位的计数，每当 4 位的计数器计到 15 后触发一次 8 位的计数器，这样就实现了计数器的切割，也提高了工作频率。

在状态机中，一般也要将大的计数器移到状态机外，因为计数器这东西一般是经常是大于 4 输入的，如果再和其它条件一起做为状态的跳变判据的话，必然会增加 LUT 的级联，从而增大组合逻辑。以一个 6 输入的计数器为例，我们原希望当计数器计到 111100 后状态跳变，现在我们将计数器放到状态机外，当计数器计到 111011 后产生个 enable 信号去触发状态跳变，这样就将组合逻辑减少了。状态机一般包含三个模块，一个输出模块，一个决定下个状态是什么的模块和一个保存当前状态的模块。组成三个模块所采用的逻辑也各不相同。输出模块通常既包含组合逻辑又包含时序逻辑；决定下一个状态是什么的模块通常又组合逻辑构成；保存现在状态的通常由时序逻辑构成。三个模块的关系如下图 9 所示。

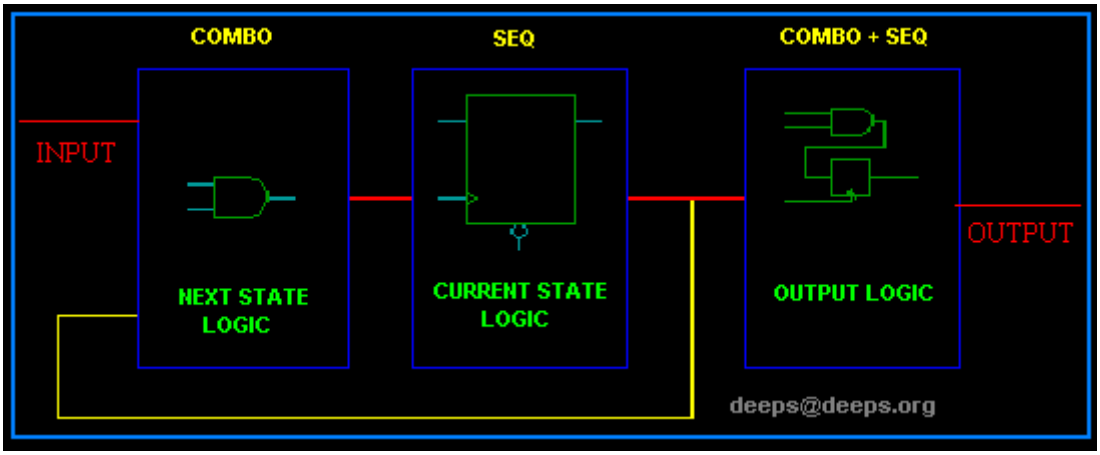

图 9 状态机的组成

所有通常写状态机时也按照这三个模块将状态机分成三部分来写，如下面就是一种良好的状态机设计方法：

```
/*-------------------------------------------------------
  This is FSM demo program
  Design Name : arbiter
  File Name       : arbiter2.v
-------------------------------------------------------*/
module arbiter2 (
clock , // clock
reset , // Active high, syn reset
req_0 , // Request 0
req_1 , // Request 1
gnt_0 ,
gnt_1);
//-------------Input Ports----------------------------
input clock    ;
input reset ;
input req_0 ;
input req_1 ;
```

```
//-------------Output Ports----------------------------
output gnt_0 ;
output gnt_1 ;
//-------------Input ports Data Type-------------------
wire clock ;
wire reset ;
wire req_0 ;
wire req_1 ;
//-------------Output Ports Data Type------------------
reg gnt_0 ;
reg gnt_1 ;
//-------------Internal Constants--------------------------
parameter SIZE = 3 ;
parameter IDLE = 3'b001 ,
          GNT0 = 3'b010 ,
          GNT1 = 3'b100 ;
//-------------Internal Variables----------------------------
reg     [SIZE-1:0]    state ;// Seq part of the FSM
wire    [SIZE-1:0]    next_state ;// combo part of FSM
//----------Code startes Here-----------------------
assign next_state = fsm_function(req_0, req_1);
function [SIZE-1:0] fsm_function;
   input       req_0;
   input       req_1;
   case(state)
     IDLE : if (req_0 == 1'b1)
                 fsm_function = GNT0;
            else if (req_1 == 1'b1)
                 fsm_function= GNT1;
            else
                 fsm_function = IDLE;
     GNT0 : if (req_0 == 1'b1)
                 fsm_function = GNT0;
            else
                 fsm_function = IDLE;
     GNT1 : if (req_1 == 1'b1)
                 fsm_function = GNT1;
            else
                 fsm_function =IDLE;
     default : fsm_function = IDLE;
endcase
endfunction
always@(posedge clock)
begin
```

```
            if    (reset == 1'b1)
                state <=IDLE;
              else
                state <=next_state;
end
//----------Output Logic----------------------------
always @ (posedge clock)
begin
if (reset == 1'b1) begin
   gnt_0 <= #1 1'b0;
   gnt_1 <= #1 1'b0;
end
else begin
   case(state)
      IDLE : begin
                  gnt_0 <= #1 1'b0;
                  gnt_1 <= #1 1'b0;
                end
      GNT0 : begin
                  gnt_0 <= #1 1'b1;
                  gnt_1 <= #1 1'b0;
                end
    GNT1 : begin
                  gnt_0 <= #1 1'b0;
                  gnt_1 <= #1 1'b1;
                end
    default : begin
                    gnt_0 <= #1 1'b0;
                    gnt_1 <= #1 1'b0;
                    end
   endcase
end
end // End Of Block OUTPUT_
endmodule
```

状态机通常要写成 3 段式，从而避免出现过大的组合逻辑。

上面说的都是可以通过流水的方式切割组合逻辑的情况，但是有些情况下我们是很难去切割组合逻辑的，在这些情况下我们又该怎么做呢？

状态机就是这么一个例子，我们不能通过往状态译码组合逻辑中加入流水。如果我们的设计中有一个几十个状态的状态机，它的状态译码逻辑将非常之巨大，毫无疑问，这极有可能是设计中的关键路径。那我们该怎么做呢？还是老思路，减少组合逻辑。我们可以对状态的输出进行分析，对它们进行重新分类，并根据这个重新定义成一组组小状态机，通过对输入进行选择(case 语句)并去触发相应的小状态机，从而实现了将大的状态机切割成小的状态机。在 ATA6 的规范中（硬盘的标准），输入的命令大概有 20 十种，每一个命令又对应很多种状态，如果用一个大的状态机（状态套状态）去做那是不可

想象的，我们可以通过 case 语句去对命令进行译码，并触发相应的状态机，这样做下来这一个模块的频率就可以跑得比较高了。

总结：提高工作频率的本质就是要减少寄存器到寄存器的时延，最有效的方法就是避免出现大的组合逻辑，也就是要尽量去满足四输入的条件，减少 LUT 级联的数量。我们可以通过加约束、流水、切割状态的方法提高工作频率。

在 FPGA 中进行时钟设计时也要注意一下几点：

1、一个模块尽量只用一个时钟，这里的一个模块是指一个 module 或者是一个 entity。在多时钟域的设计中涉及到跨时钟域的设计中最好有专门一个模块做时钟域的隔离。这样做可以让综合器综合出更优的结果。

2、除非是低功耗设计，不然不要用门控时钟--这会增加设计的不稳定性，在要用到门控时钟的地方，也要将门控信号用时钟的下降沿 打一拍再输出与时钟相与。

3、禁止用计数器分频后的信号做其它模块的时钟，而要用改成时钟使能的方式，否则这种时钟满天飞的方式对设计的可靠性极为不利，也大大增加了静态时序分析的复杂性。

1.4 不同时钟域之间的同步

当一个设计中的两个模块分别用的是两个工作时钟，那么在它们的接口处就工作在异步模式，这时为了保证数据能正确的处理那么就要对两个模块进行同步。

这里的不同的时钟域通常是以下的两种情况：

1、 两个时钟的频率不同；

2、 虽然两个时钟的频率相同，但是它们是两个独立的时钟，其相位没有任何关系。

分别如下两个图所示：

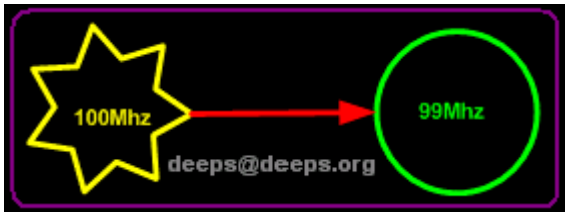

图 10 两个时钟的频率完全不同

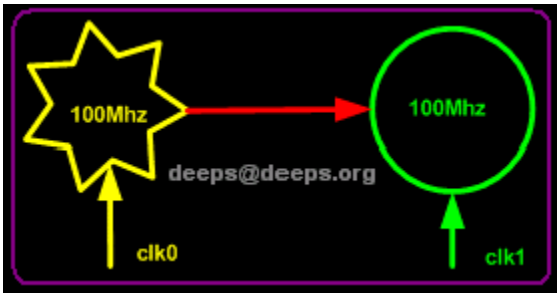

图 11 两个时钟的频率相同，但相位不相关

两个时钟域之间传输的数据根据不同的位宽通常采用不同的同步的方法。

1、单 bit 之间的同步且发送的每个 pulse 至少有 1 个周期宽度的情况

这类同步主要是用于一些控制信号自己的同步。通常的采用方法就是输出数据在接收的模块中利用两个触发器采用系统时钟打两拍，如下图 12 所示。对于这种同步需要说明以下几点。

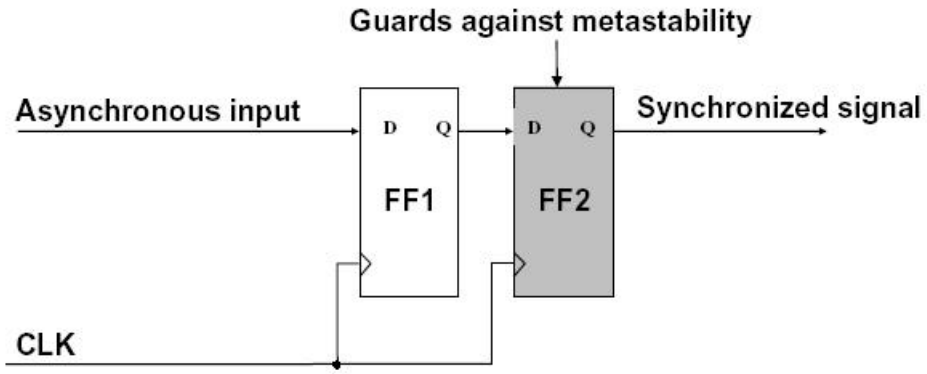

图 12 一位同步器设计

(1) 图 12 中的同步电路其实叫"一位同步器",它只能用来对一位异步信号进行同步,而且这个信号的宽度必须大于本级时钟的脉冲宽度,否则有可能根本采不到这个异步信号。

(2) 为什么图一中的同步电路只能用来对一位异步信号进行同步呢?

(a) 当有两个或更多的异步信号(控制或地址)同时进入本时域来控制本时域的电路时,如果这些信号分别都用图 12 中的同步电路来同步就会出现问题,由于连线延迟或其他延迟使两个或更多的异步信号(控制或地址)之间产生了 skew,那么这个 skew 经过图 12 的同步器同步进入本时域后,会产生很大的 skew 或产生竞争,导致本时域电路出错。

出现的问题如下图 13 所示：

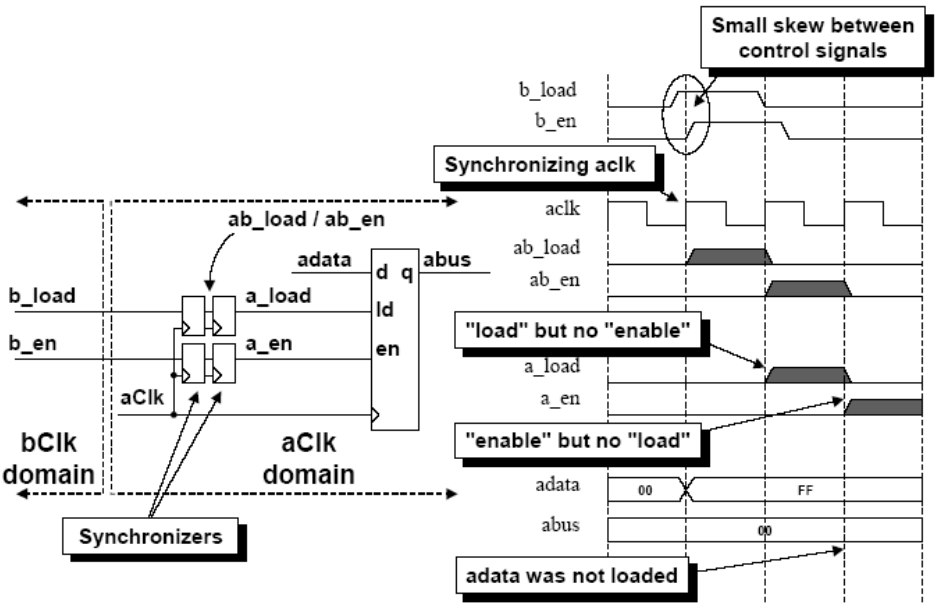

Figure 8 - Problem - Passing multiple control signals between clock domains

图 12 同步多个控制信号时出错

(b) 如果是异步数据总线要进入本时域,同样不能用图 12 的电路,因为数据的变化是很随机的,其 0 的宽度或 1 的宽度和本时域时钟脉冲无关,所以图 12 的电路可能会采不到正确数据。

(3) 注意，第二个触发器并不是避免“亚稳态的发生”，确切的说，该电路能够防止亚稳态的传播。也就是说，一旦第一个触发器发生了亚稳态（可能性存在），由于有了第二个

触发器，亚稳态不会传播到第二个触发器以后的电路中去。

（4）第一级触发器发生了亚稳态，需要一个恢复时间来稳定下来，或者叫退出亚稳态。当恢复时间加上第二级触发器的建立时间（更精确的，还要减去 clock skew）小于等于时钟周期的时候（这个条件还是很容易满足的，一般要求两级触发器尽量接近，中间没有任何组合逻辑，时钟的 skew 较小），第二级触发器就可以稳定的采样，得到稳定的确定的数据了，防止了亚稳态的传播。

（5）FF2 是采样了 FF1 的输出，当然是 FF1 输出什么，FF2 就输出什么。仅仅延迟了 1 个周期。注意，亚稳态之所以叫做亚稳态，是指一旦 FF1 进入，其输出电平不定，可能正确也可能错误。所以必须说明的是，虽然这种方法可以防止亚稳态的传播，但是并不能保证两级触发器之后的数据是正确的，因此，这种电路都有一定数量的错误电平数据，所以，仅适用于少量对于错误不敏感的地方。对于敏感的电路，可以采用双口 RAM 或 FIFO。

2　输入 pulse 有可能小于一个时钟周期宽度情况下的同步电路

对 2 的情况通常采用如下图 14 的反馈电路。该电路的分析如下：假设输入的数据是高电平，那么由于第一个触发器 FF1 是高电平清零，所有输出也是高电平，采用正确。如果输入是第电平那么被 FF1 被强制清零，这个时候输出位零。这样就保证了输出的正确性。

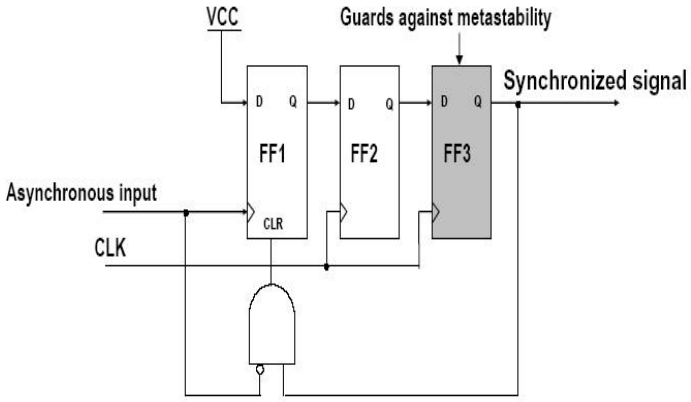

图 14 输入 pulse 有可能小于一个时钟周期宽度情况下的同步电路

对于要控制多个信号的情况可以参考详细的分析：www.fpga.com.cn中的设计异步多时钟系统的综合以及描述技巧.pdf。不过下面也介绍几种常用的方法。

(1)可采用保持寄存器加握手信号的方法同步(多数据,控制,地址)；

(2)特殊的具体应用电路结构,根据应用的不同而不同；

(3)异步 FIFO。

下面介绍一下利用异步 FIFO 来进行同步。

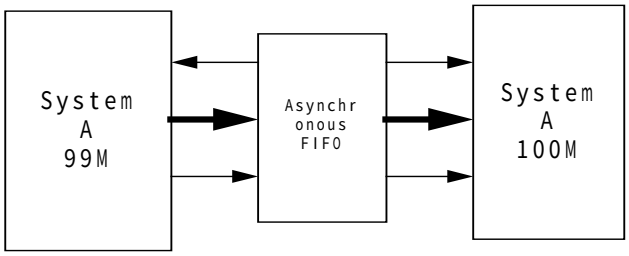

图 15　利用异步 FIFO 实现数据的同步

控制方法：