第九章 设计与优化

主讲人：徐向民教授

设计与优化

- 设计与优化的重要思想
- 设计与优化的重要概念
- 跨时钟域的信号处理

设计与优化的重要思想

设计与优化的重要思想

数字系统设计与优化的重要思想：

先设计模块电路后设计代码，设计者须明确每一段代码生成的电路，否则优化无从谈起。

设计与优化的重要思想

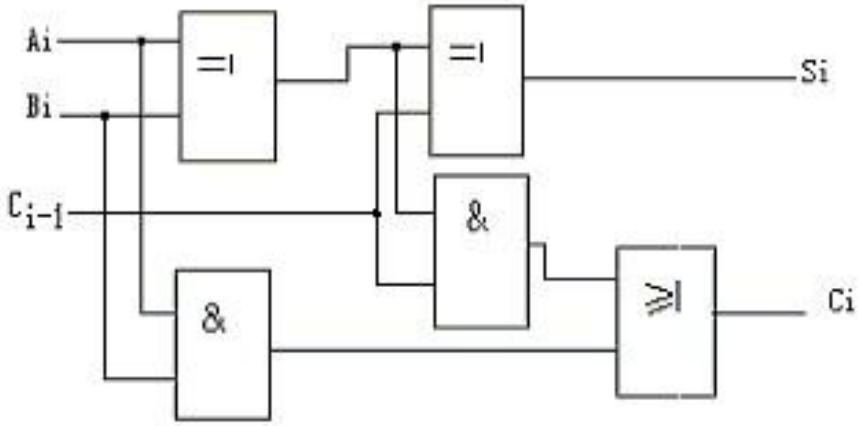
下面以一个简单的设计阐明这一思想：

例:全加器设计（FPGA）

逻辑函数：

$Si=Ai \oplus Bi \oplus Ci\text{-}1$

$Ci=AiBi+Ci\text{-}1(Ai \oplus Bi)$

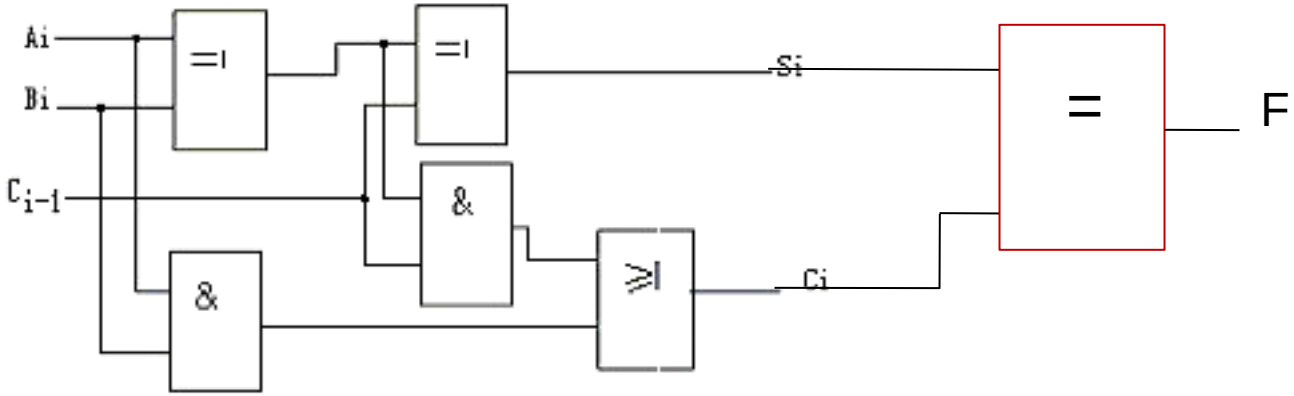


VHDL代码：

```
tmp <= Ai  xor  Bi ;
Si   <= tmp  xor  Ci-1 ;
Ci   <= (Ai and Bi) or (Ci-1  and  tmp) ;
```

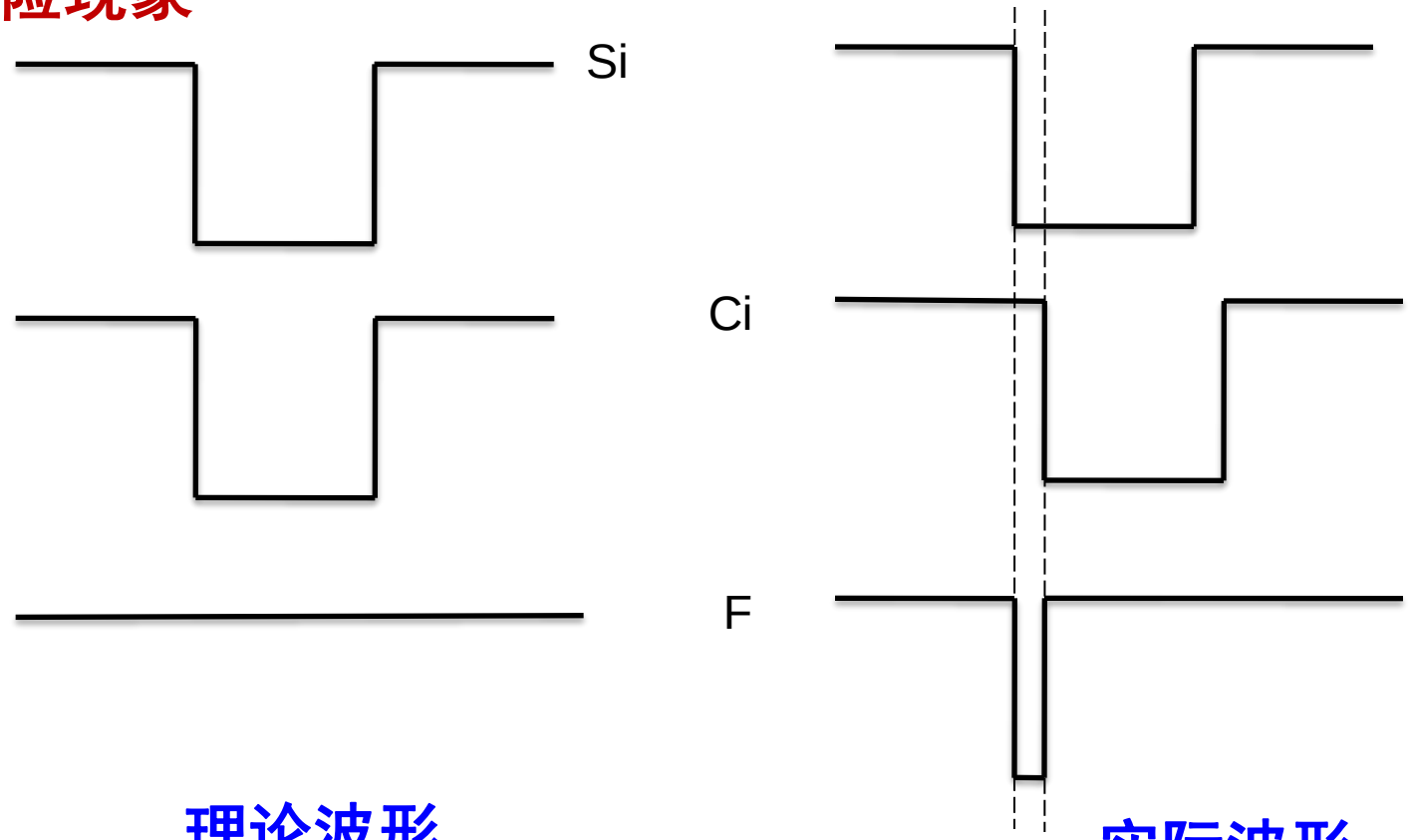
设计与优化的重要思想

如上述设计的全加器，若Ci 和Si做同或运算时：



设计与优化的重要思想

如上述设计的全加器，若Ci 和Si做同或运算时：将产生竞争冒险现象

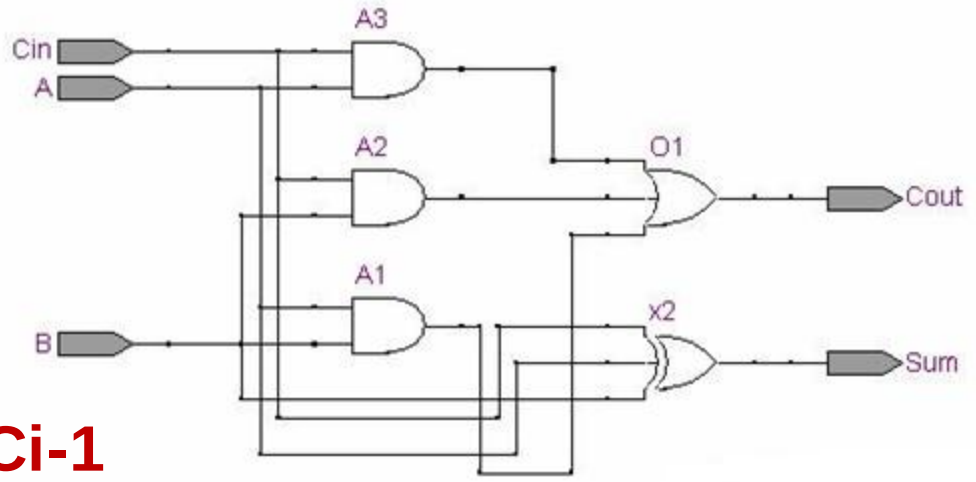


理论波形

实际波形

设计与优化的重要思想

消除上述全加器竞争冒险现象：



逻辑函数

$Si=Ai \oplus Bi \oplus Ci-1$

将：$Ci=AiBi+Ci-1(Ai \oplus Bi)$

改为：$Ci=AiBi + BiCi-1 + AiCi-1$

VHDL代码

```
Si <= Ai xor Bi  xor Ci-1;
Ci <=（Ai and Bi）or（Bi and Ci-1）or（Ai and Ci-1）;
```

设计与优化的重要思想

严格的说，VHDL代码不是程序。VHDL既称为硬件描述语言，则VHDL主要用于设计描述硬线电路以及对设计的抽象仿真。那么，对于一个数字系统设计者来说，能够将所设计的代码映射为相应的硬线电路是必须具备的能力。

下面以寄存器的引入为例详细阐述：

寄存器的引入方法

1. 触发器的引入：

简单来说，寄存器的引入方法有一下几点：

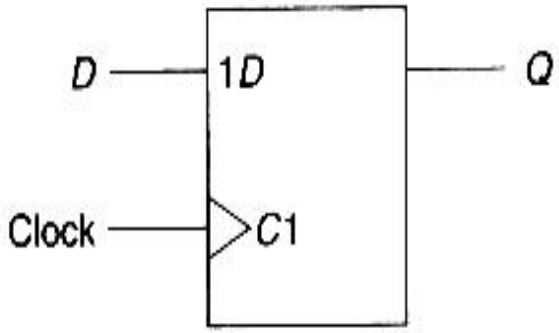
- ◆ 条件涵盖不完整的if语句会产生触发器
- ◆ 条件涵盖不完整的case语句会产生触发器

以下通过几个例子来说明。

寄存器的引入方法

1. 触发器的引入：

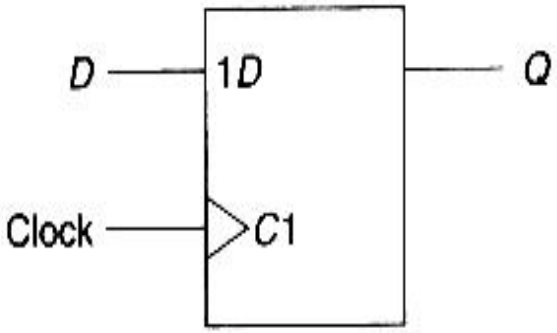
```
ENTITY DFF IS
PORT(a,clk:in std_logic;
        y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF DFF IS
BEGIN
PROCESS(clk)
BEGIN
    IF (clk'event AND clk='1') THEN
        y<=a;
    END IF;
END PROCESS;
```



寄存器的引入方法

1. 触发器的引入：

```
ENTITY DFF IS
PORT(a,clk:in std_logic;
          y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF DFF IS
BEGIN
PROCESS(clk)
 BEGIN
   IF (rising_edge(clk)) THEN    y<=a;
   END IF;
END PROCESS;
```

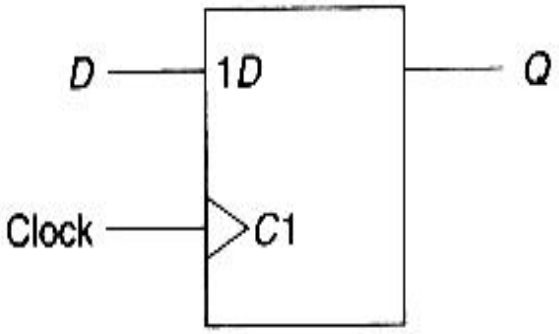


点评：此时clk必须为std_logic类型。

寄存器的引入方法

1. 触发器的引入：

```
ENTITY DFF IS
PORT(a,clk:in std_logic;
        y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF DFF IS
BEGIN
PROCESS
BEGIN
    WAIT UNTIL clk'evnt ANDclk='1';
    y<=a;
END PROCESS;
```

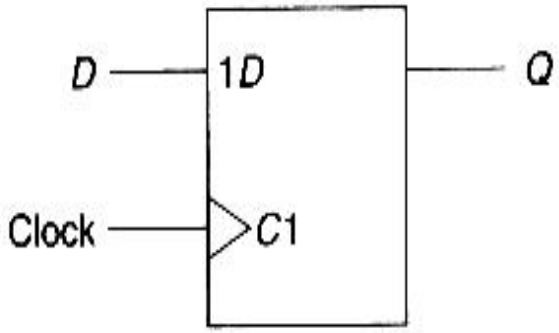


点评：Wait语句必须放在进程的首部或尾部，并且一个进程中的wait语句不能超过一个。

寄存器的引入方法

1. 触发器的引入：

```
ENTITY DFF IS
PORT(a,clk:in std_logic;
        y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF DFF IS
BEGIN
PROCESS
BEGIN
    IF clk='1' THEN
        y<=a;
    END IF;
END PROCESS;
```



点评：因为要启动进程必须要clk发生跳变，且仅当clk='1'时赋值才有效。所以综合后是一个D触发器。

寄存器的引入方法

1. 触发器的引入：

上例中用到了条件涵盖不完整的if语句形成触发器，当if语句涵盖完整时，综合后形成一般的组合逻辑。而一下为涵盖完整的if语句。

```
PROCESS (clk,a,b)
BEGIN
   IF clk='1' THEN
      y<=a;
   ELSE
      y<=b;
   END IF ;
END PROCESS;
```

点评：if语句条件完全覆盖，产生一个2选1的多路选择器。

寄存器的引入方法

以下例子为条件涵盖不完整的case语句引入寄存器的方法。

```
PROCESS( state, inA, inB)
  BEGIN
      CASE state IS
          WHEN s0 =>
              outA<='1'; --没有对outB赋值，所以outB保持原值
          WHEN  s1 =>
              outA<=inB ;
              outB<='1';
          WHEN s2 =>
              outB<=inA;    --没有对outA赋值，所以outA保持原值
      END CASE;
END PROCESS;
```

寄存器的引入方法

2. 锁存器的引入：

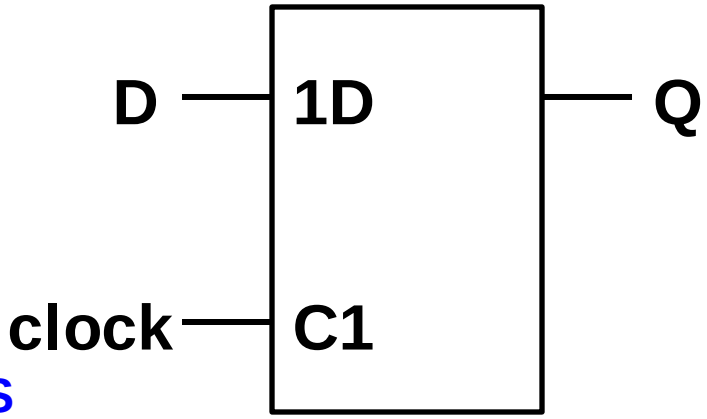
同样的道理，锁存器的引入方法有一下几点：

- ◆ 条件涵盖不完整的if语句会产生锁存器
- ◆ case语句中条件不完全覆盖会产生锁存器

以下通过几个例子来说明。

寄存器的引入方法

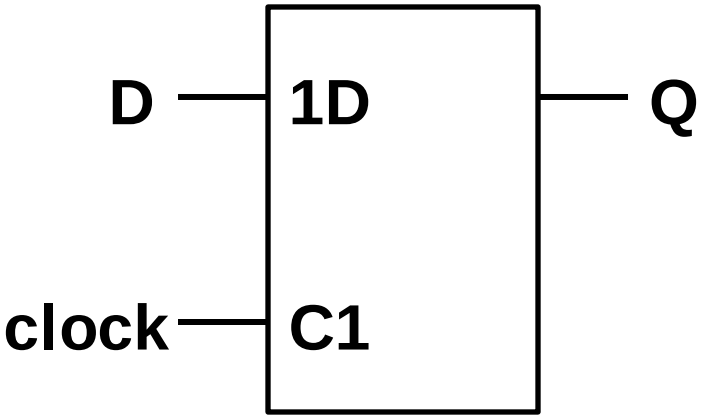
2. 锁存器的引入：




```
ENTITY rigister IS
 PORT(a,clk:in std_logic;
          y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF rigister IS
BEGIN
PROCESS(clk,a)
  BEGIN
    IF clk='1' THEN
          y<=a;
    ELSE          --VHDL默认保持先前的值，故引入高电平锁存器
    END IF;
 END PROCESS;
```

寄存器的引入方法

2. 锁存器的引入：

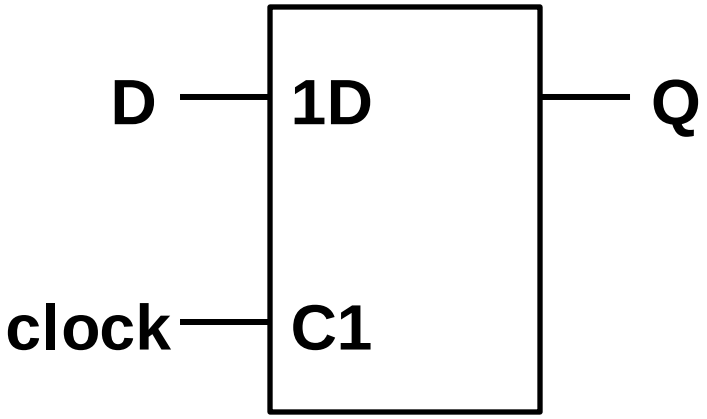



```
ENTITY rigister IS
 PORT(a,clk:in std_logic;
          y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF rigister IS
BEGIN
PROCESS(clk,a)
  BEGIN
    IF clk='1' THEN
          y<=a;
        --省去了ELSE分支，表示分支y值不发生跳变，同样引入高电平锁存器
    END IF;
 END PROCESS;
```

寄存器的引入方法

2. 锁存器的引入：

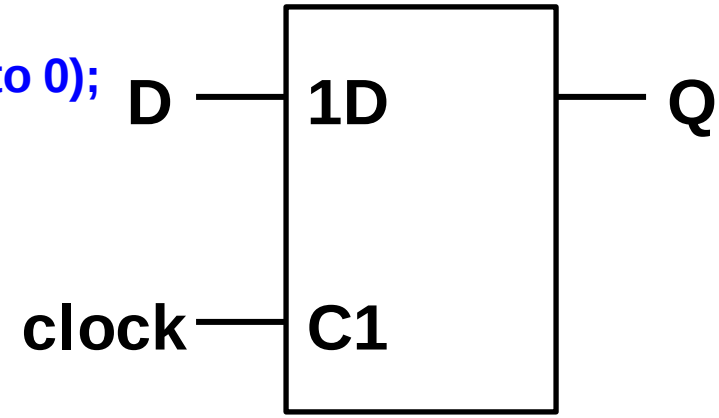
```
ENTITY rigister IS
 PORT(a,clk:in std_logic;
          y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF rigister IS
BEGIN
PROCESS(clk,a)
  BEGIN
    IF clk='0' THEN        --引入低电平锁存器
          y<=a;
    END IF;
END PROCESS;
```



寄存器的引入方法

2. 锁存器的引入：

```
ENTITY rigister IS
    PORT(a,b:in std_logic;
         sel:in std_logic_vector(1 downto 0);
         y:out std_logic);
END DFF;
ARCHITECTURE BEHAV OF rigister IS
BEGIN
PROCESS(sel,a,b)
 BEGIN
  CASE  sel  IS
    WHEN  “00”  => y<=a;
    WHEN  “01”  => y<=a;
    WHEN  OTHERS  =>  NULL;
  END CASE;
END PROCESS;
```

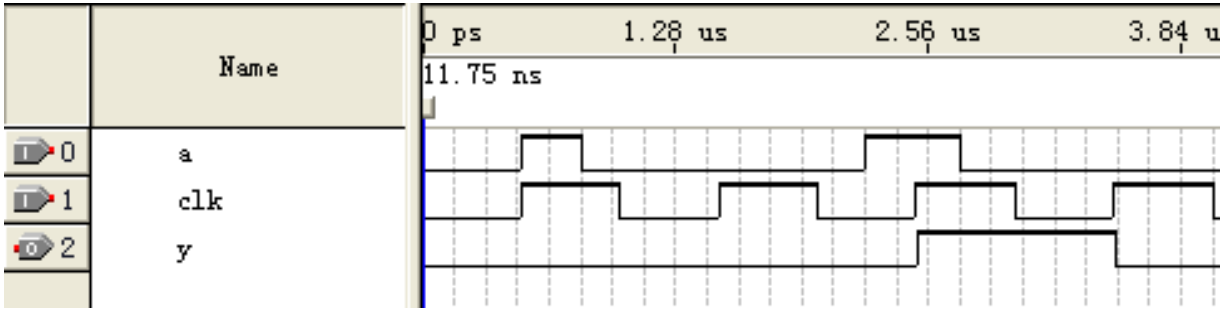


CASE语句中的条件不完全覆盖也将导致寄存器的引入。

在第五章里已经提到锁存器与触发器的区别，这里用两个程序来演示。

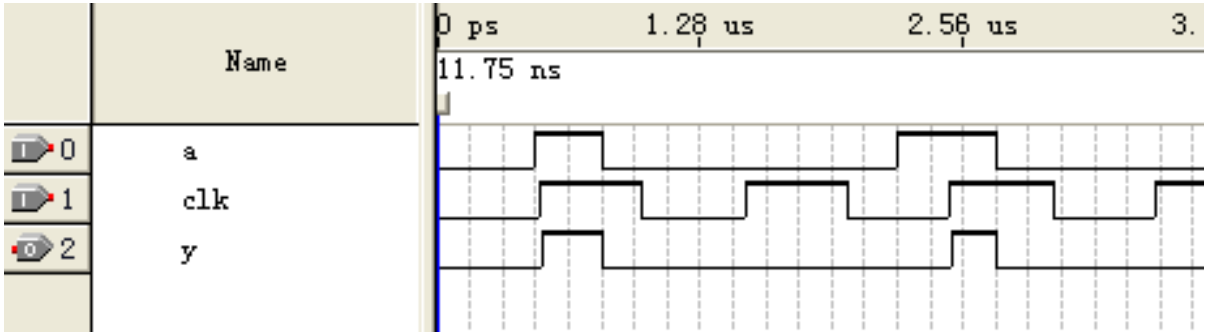
触发器：

```
PROCESS(clk)
   BEGIN
      IF clk='1' THEN
         y<=a;
      END IF;
END PROCESS;
```



锁存器：

```
PROCESS(clk,a)
   BEGIN
      IF clk='1' THEN
         y<=a;
      END IF;
END PROCESS;
```



寄存器的引入方法

3. 寄存器引入应注意的问题：

例1：

```
process(clk_a , clk_b)
begin
   if  (clk_a'event and clk_a='1') then
       a<=b;
   end  if;
   if  (clk_b'event and clk_b='1') then  --错误
       c<=b;
   end  if;
end  process;
```

错误点评：

因为在MAXPLUS2中，一个进程只能引入了一个边沿检测的语句。

在QUARTUS中就不会有这种错误。

寄存器的引入方法

3. 寄存器引入应注意的问题：

例2：

```
process(clock)
begin
   if (clock'event  and  clock='1') then
      sig<=b;
   else
      sig<=c;   --错误
   end if;
end process;
```

错误点评:

将用于产生寄存器的信号或变量赋值语句放在了ELSE条件分支上。

这种赋值方式相当于检测如果没有时钟信号，则赋新值。不可能有这样的硬件电路与之对应。

寄存器的引入方法

3. 寄存器引入应注意的问题：

例3：

```
process(clock)
   variable edge_var, any_var : bit;
begin
   if (clock'event  and  clock='1') then
      edge_signal<=x;   --此句用于产生寄存器
      edge_var :=y;     --此句用于产生寄存器
      any_var := edge_var ;   --错误
    end if ;
    any_var :=edge_var ;      --错误
end process;
```

错误点评：

如果一个变量已在IF的边 沿检测结构中作了赋值操作，就不能在同一进程中再作读操作。

此错误在QUARTUS2中修改为WARNING。

寄存器的引入方法

3. 寄存器引入应注意的问题：

例4：

```
if  not ( clock'event and clock='1') then   --错误
```

错误点评：将边沿表达式当成了操作数。

寄存器的引入方法

3. 寄存器引入应注意的问题：

在寄存器的引入时，另外需要注意以下4点：

- ◆在引入寄存器时，一般情况下采用异步复位的方式
- ◆寄存器中，复位的优先级一般要高于置位
- ◆在引入时钟具有使能作用的寄存器时，宜采用嵌套的if语句来描述
- ◆注意将时序进程和组合进程分开描述

详见课本例子7-16，P111

寄存器的引入方法

4. 具有时钟门控的触发器引入：

```
ENTITY D_FF IS
    PORT(d,clk,ena:in std_logic;
         q:out std_logic);
END ENTITY;
ARCHITECTURE BEHAV OF D_FF IS
BEGIN
PROCESS(clk,ena)
BEGIN
    IF  clk'event  AND  clk='1'   AND   ena='1'  THEN
        q<=d;
    END IF;
END PROCESS;
```

--这种写法相当于在时钟的输入通道上加了一个与非门，有可能导致不可靠的工作情况。

寄存器的引入方法

4. 具有时钟门控的触发器引入：

```
ENTITY D_FF IS
     PORT(d,clk,ena:in std_logic;
          q:out std_logic);
END ENTITY;
ARCHITECTURE BEHAV OF D_FF IS
BEGIN
PROCESS(clk,ena)
BEGIN
   IF  clk'event  AND  clk='1' THEN
      IF  ena='1'  THEN
         q<=d;
      END IF;
   END IF;
END PROCESS;
```

--这种写法有比较好的可靠性，而且节省资源。

寄存器的引入方法

5. 同步复位/置位功能引入：

```
process(clk)
begin
    if  clk'event and clk='1' then
        if  set='1'  then
            y<='1';
            --注意，输入 '1'（或true）才能引入硬件置位功能
        else
            y<=a and b;
        end if ;
    end if;
end process;
```

寄存器的引入方法

6. 异步复位/置位功能引入：

```
process(clk , reset , set)
begin
    if reset='1'then
        y<='0';      --必须是一个常量值0才能引入硬件复位机制
     elsif  set='1'then
        y<='1';      --必须是一个常量值1才能引入硬件置位机制
    elsif  rising_edge(clk)  then
        y<=a and b;
    end if ;
end process;
```

点评：复位的优先级比置位高，这是符合常规硬件电路结构的。

避免引入不必要的寄存器

由于综合在硬件电路中需要一定的空间，为了节省资源，让硬件更快的工作，平时编写程序的时候，尽可能的避免不必要的寄存器的引入，要注意以下几点：

1） 组合逻辑进程中不能存在边沿触发状态。（ PA111-112，例7-17）

2) IF语句，CASE语句涵盖要完整。（ PA113-114，例7-18）

3) 如果信号或变量在一个CASE分支有赋值，就必须在每个分支都有赋值操作(或者在CASE语句前面有赋值)。
（ PA113-114，例7-18）

避免引入不必要的寄存器

下例通过条件涵盖不完整的case语句产生寄存器。

```
process ( present_state, inA, inB)
begin
      case present_state is
        when s0 =>
             outA<='1';       --没有对outB赋值，所以outB保持原值
        when  s1 =>
             outA<=inB ;
             outB<='1';
        when s2 =>
             outB<=inA;           --没有对outA赋值，所以outA保持原值
      end case;
end process;
```

避免引入不必要的寄存器

对于前面的例5，如何避免引入不必要的锁存器呢？
有下面两种方法：

1、在每个case分支都对outA和outB赋值；
2、在case语句前面，先对outA和outB赋初值；

这两种解决办法详见课本p113-P117

避免引入不必要的寄存器

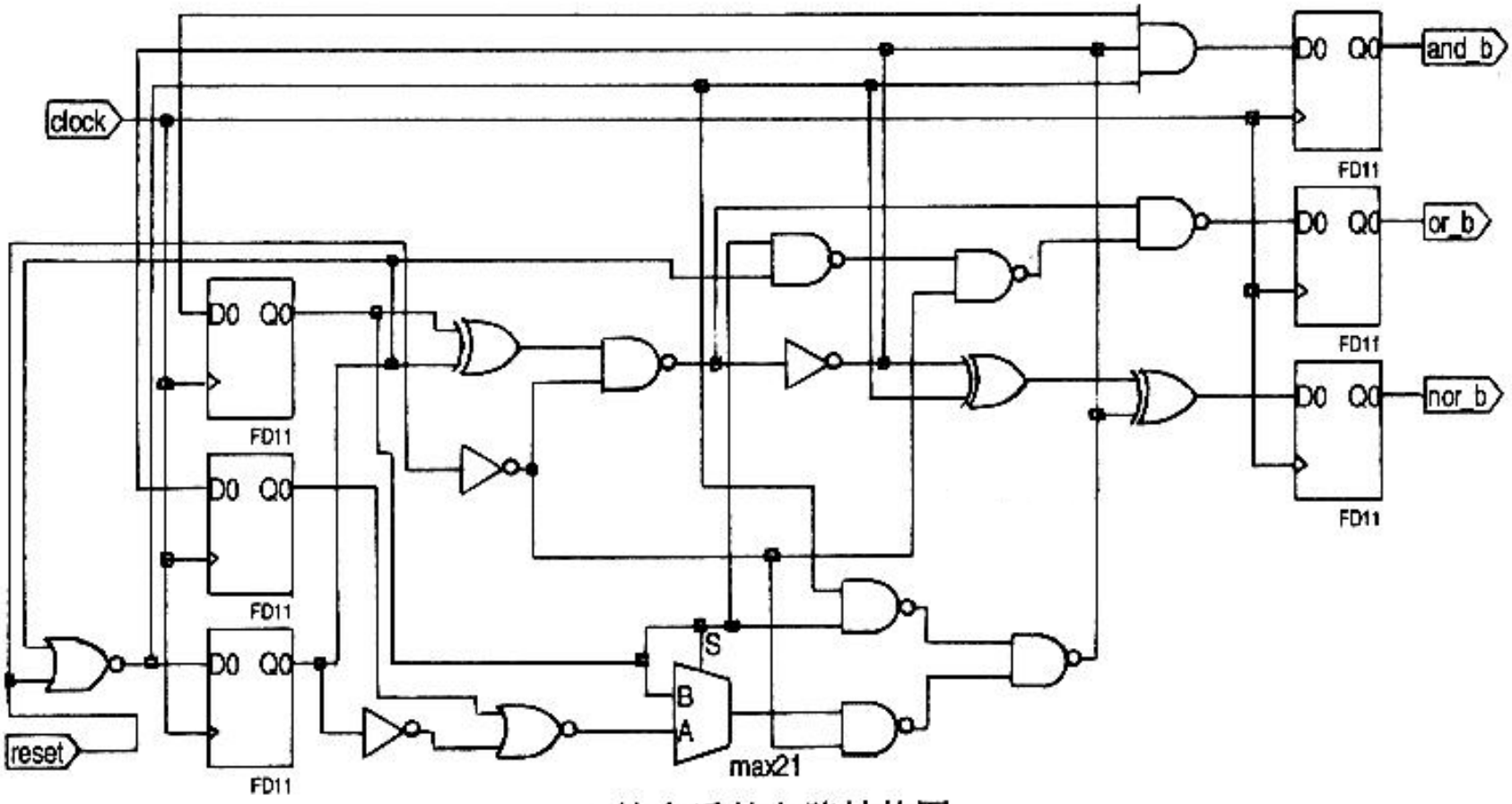
课本例7-16：基于3位二进制计数器的3个逻辑输出

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity exmp is
    port (  clock, reset : in std_logic ;
            and_b, or_b, xor_b : out std_logic) ;
end exmp;
architecture rtl of exmp is
begin
```

避免引入不必要的寄存器

```
process
    variable count: std_logic_vector(2 downto 0);
begin
    wait until clock'event and clock='1';
     if  reset='1' then
         count :="000";
     else count :=count+1;
     end if;
     and_b<=count(2) and count(1) and count(0);
     or_b<=count(2) or count(1) or count(0);
     xor_b<=count(2) xor count(1) xor count(0);
end process;
end rtl;
```

例7-16 综合后的电路结构如图：



综合后的电路结构图

避免引入不必要的寄存器

由图知，引入了6个D触发器。其实三个输出只依赖于count的计数值，由于count作为累加器，已具有存储功能，3个输出变量没有必要利用别的寄存器另加存储。例7-16的问题在于将3个输出赋值语句放在了同一个具有wait语句的进中。

为了解决这些问题，以免引入过多的寄存器，可将这3个输出赋值语句放在另外一个没有wait或if语句的进程中。下面的例7-17中有两个进程，一个进程具有Wait语句，用于产生具有寄存器性质的计数器，另一个只作输出赋值用。

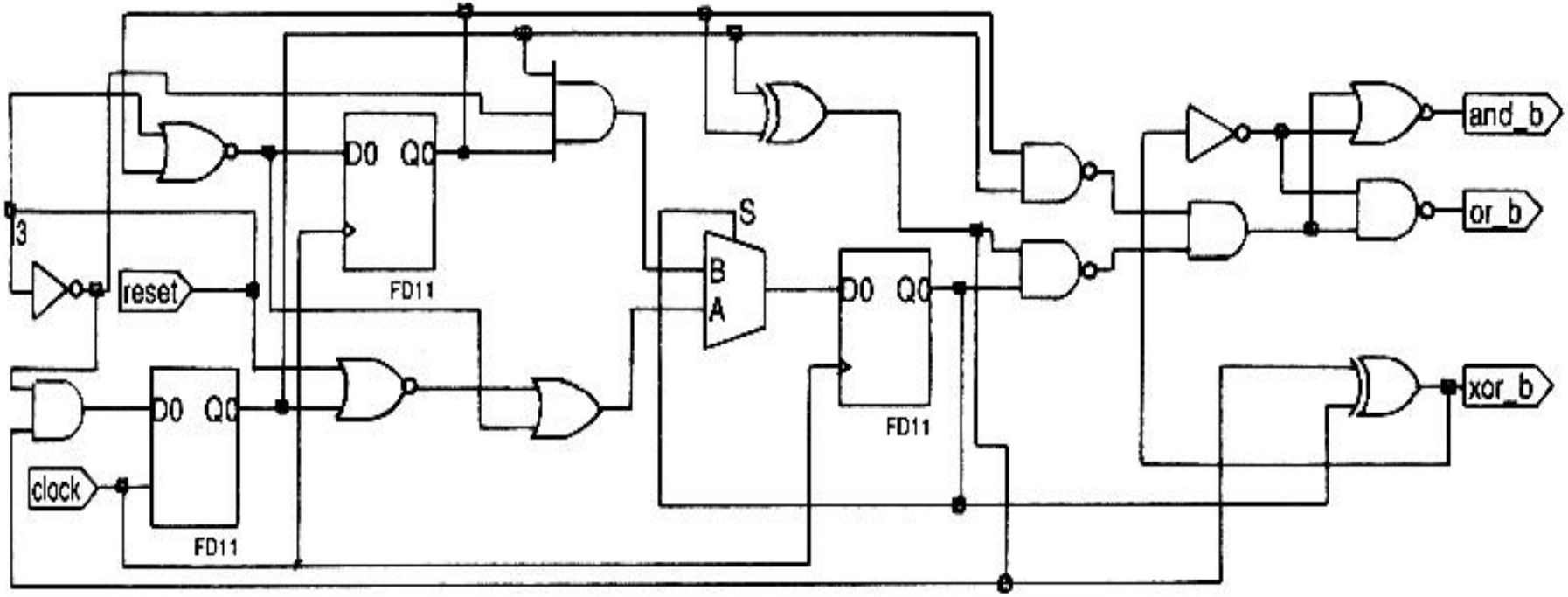
例7-17（例7-16的改进）：

```
architecture  rtl  of  exmp  is
signal count : std_logic_vector(2 downto 0);
begin
  process
  begin
    wait until clock'event and clock='1' ;
    if   (reset='1')  then
          count<="000";
    else  count<=count+1;
    end if;
  end process;
  process(count)
  begin
    and_b<=count(2) and count(1) and count(0);
    or_b<=count(2) or count(1) or  count(0);
    xor_b<=count(2) xor count(1) xor count(0);
  end process;
end  rtl;
```

--count定义为signal类型，

用于两个进程间信号的传递

例7-17 综合后的电路结构如图：



综合后的电路结构图

设计与优化的重要概念

设计与优化的重要概念

建立时间：

指在触发器的时钟信号上升沿到来以前，数据稳定不变的时间。如果 建立时间不够，数据将不能正确输入触发器。

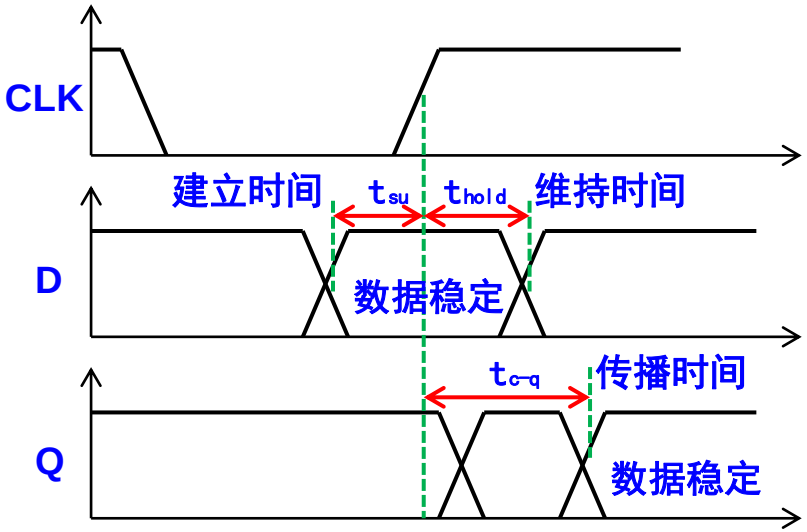
保持时间：

指在触发器的时钟信号上升沿到来以后，数据稳定不变的时间。如果 保持时间不够，数据同样不能正确输入触发器。

设计与优化的重要概念

传播延时：

指信号播路径上传播所需要的时间。



时序电路工作的时钟周期T必须能容纳电路中任何一级的最长延时

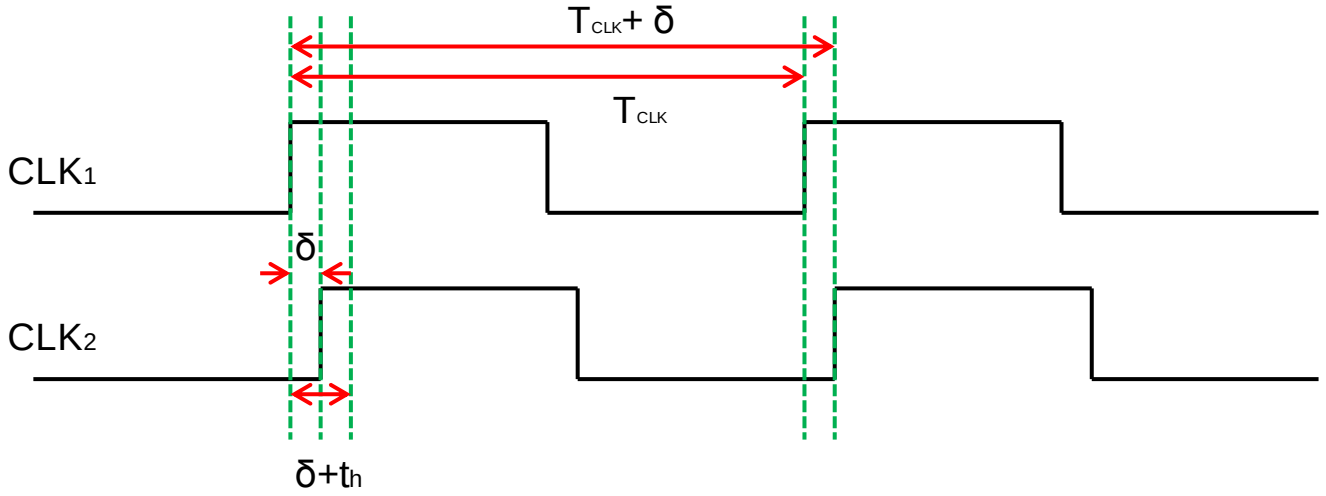
$$T > t_{c-q} + t_{plogic} + t_{su}$$

其中tplogic表示一个逻辑最坏情形的延时。

设计与优化的重要概念

时钟偏差：

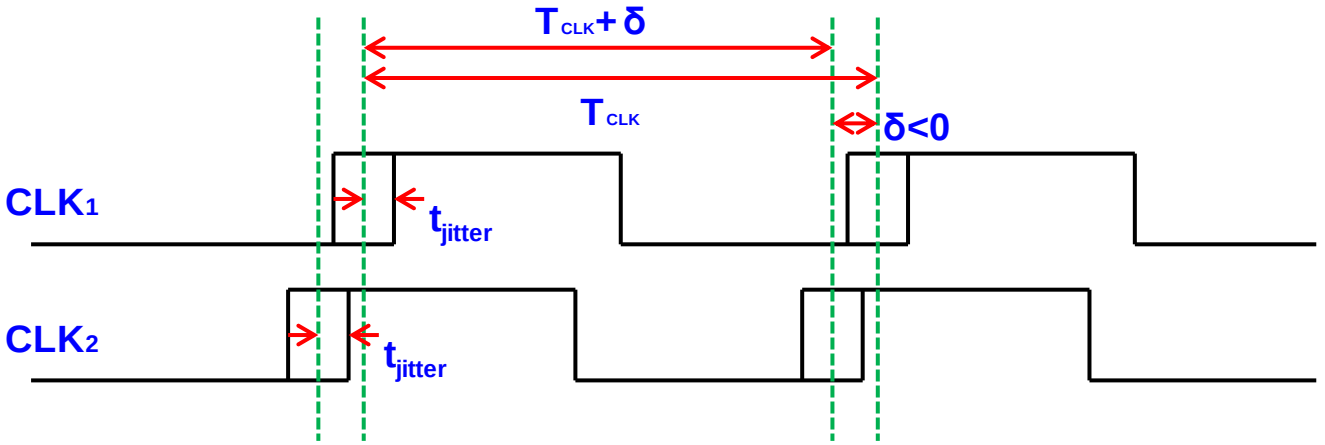
一个时钟翻转的到达时间在空间上的差别通常称为时钟偏差。时钟偏差是由时钟路径的表态不匹配以及时钟在负载上的差异千万的。



设计与优化的重要概念

时钟抖动：

是指在芯片的某一个定点上时钟周期发生暂时的变化，即时钟周期在每个不同的周期上可以缩短或加长。



设计与优化的重要概念

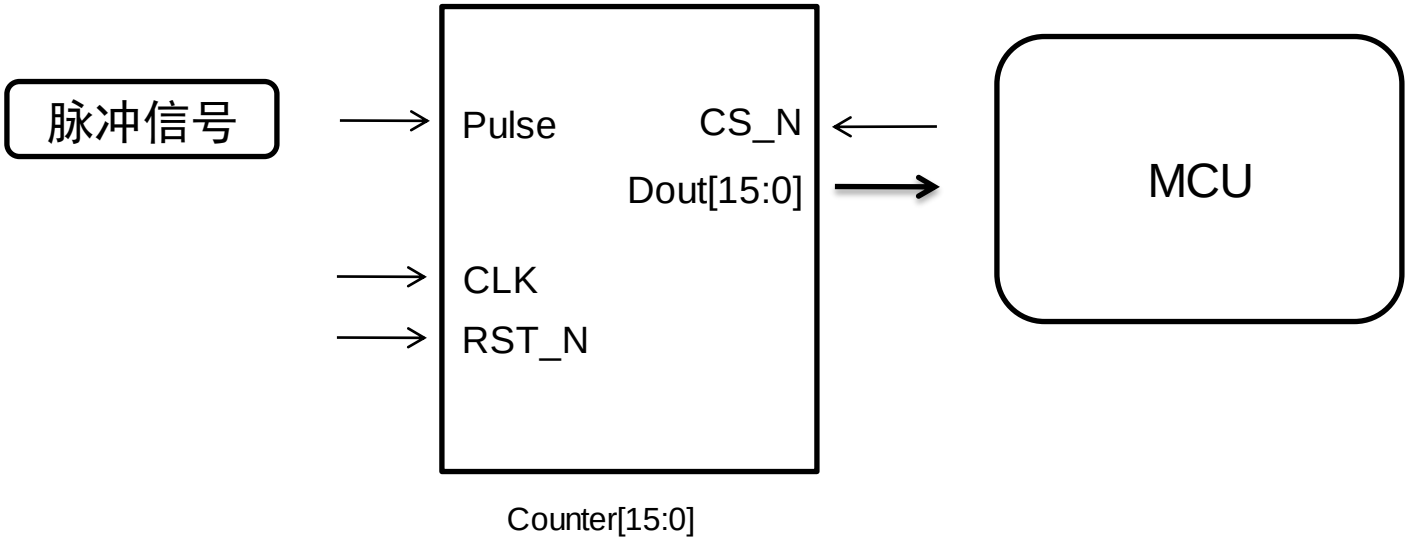
偏差和抖动的共同影响：

假设由于时钟分布的原因，两个寄存器上的时钟信号之间存在一个静态偏差δ。而且是两个时钟都有抖动t_{jitter}。为了确定对最小时钟周期的限制，我们必须先看一下完成所要求的计算可用的时间是最坏情况发生在CLK1的当前时钟周期的上升沿推迟出现而CLK2的下一个时钟周期的上升沿提前出现。由此得到下列约束条件：

$$T_{CLK} + \delta - 2t_{jitter} > t_{c-q} + t_{plogic} + t_{su}$$

习题

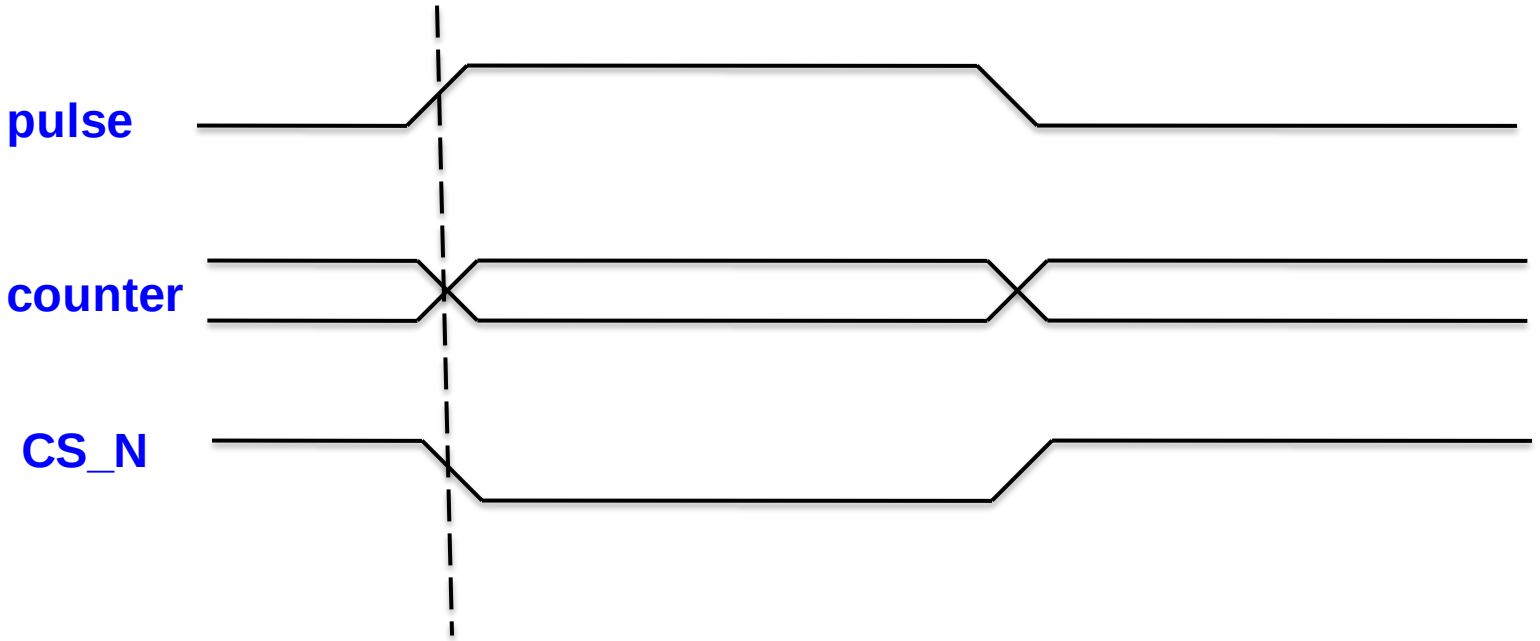
设计一个脉冲计数器，在脉冲上升沿触发计数，上位机MCU在片选的下降沿到来时读取一个计数值：



直观的初步设计：

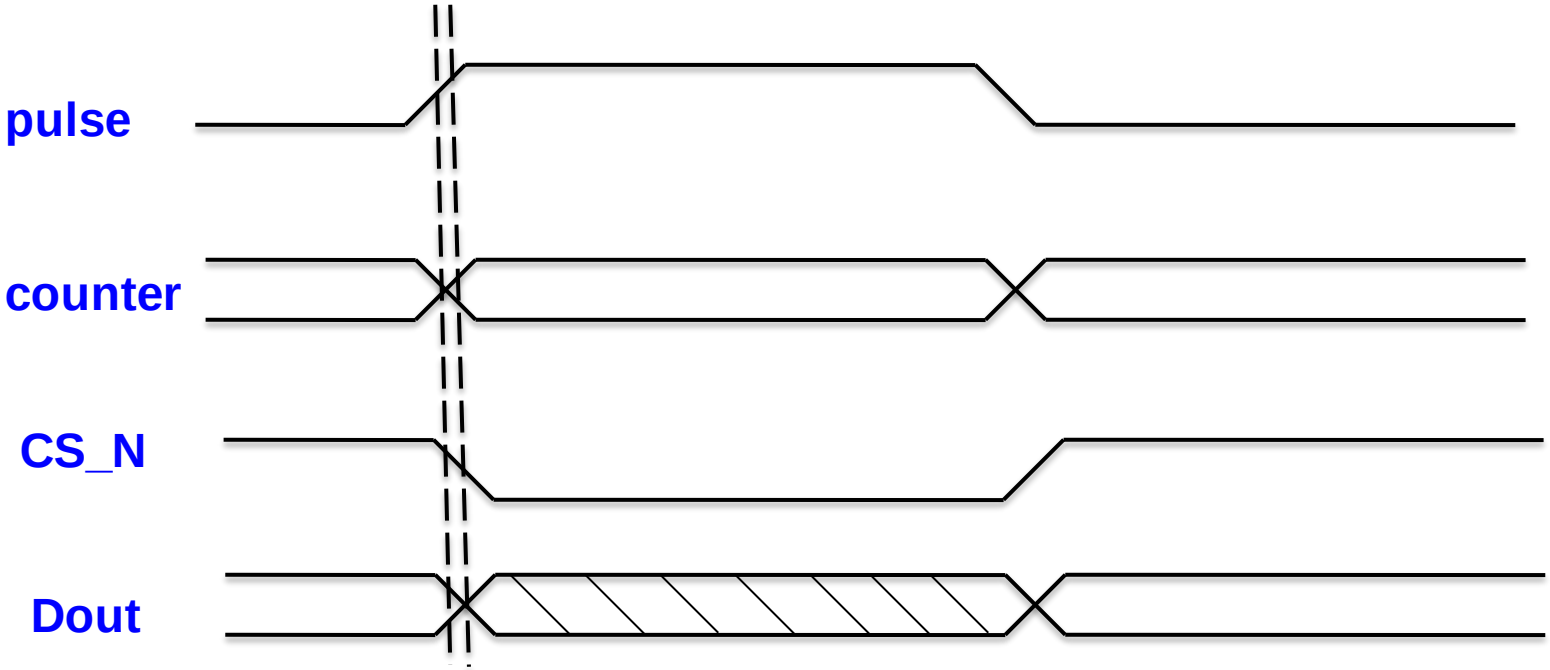
```
pulse:process(rst_n,pulse)
        begin
            if(rst='0') then
                counter <= (others=>'0');
            elsif(pulse 'event and pulse = '1')
                counter <= counter + 1;
             endif;
        endprocess;
cs:process(rst_n,cs_n)
        begin
           if(rst='0') then
                dout <= 0;
           elsif(cs_n 'event and cs_n = '0')
                 dout <= counter;
           endif;
      endprocss;
```

问题： 由于脉冲信号PULSE和读片选信号CS_N是异步信号，PULSE何时出现上升沿和CS_N何时出现下降沿是不可控。若两者同时出现?



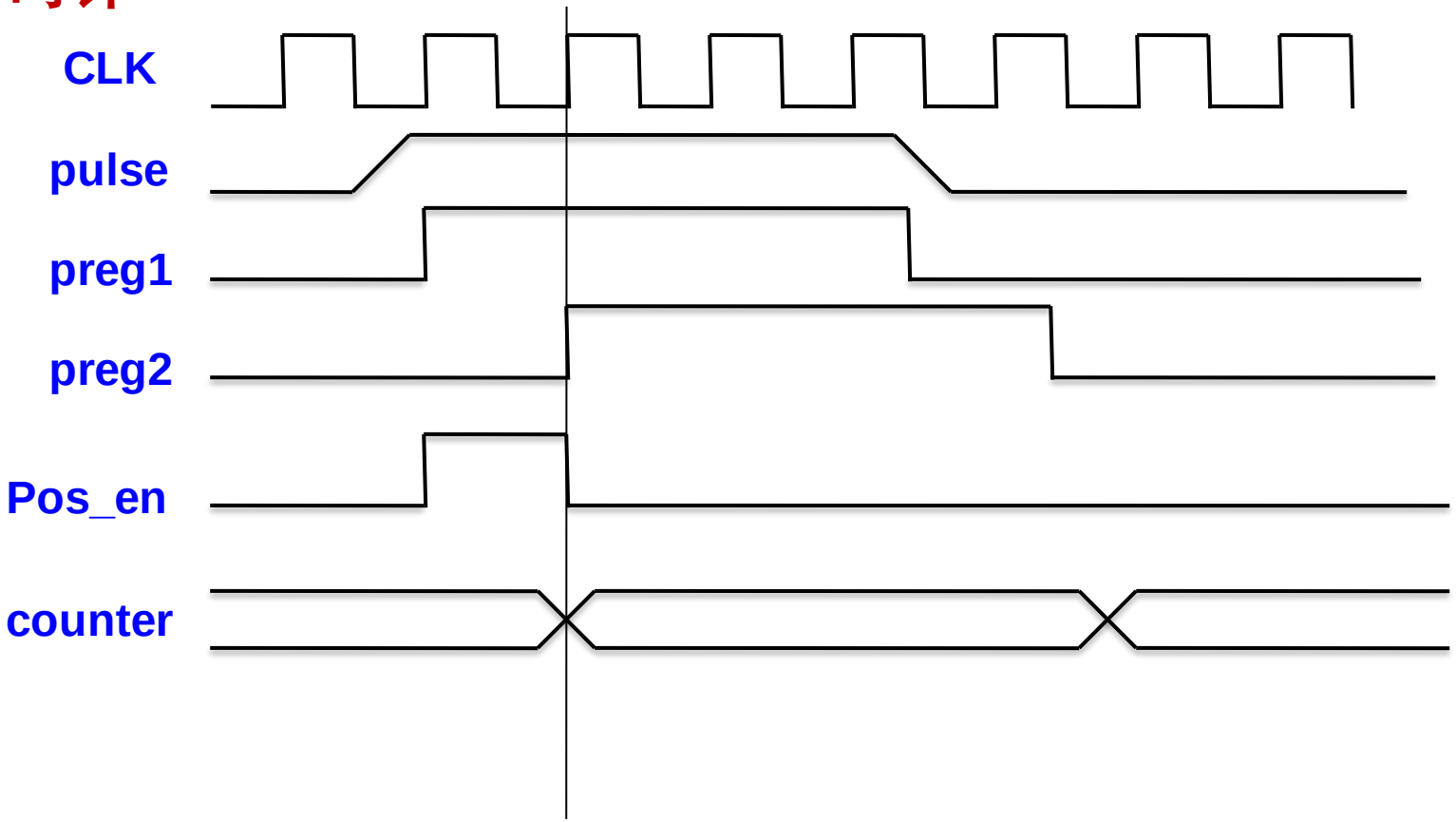
则无法预测得到的结果是加1前还是加1后

问题： 若CS_N下降沿出现时Dout <= counter的建立时间不足?

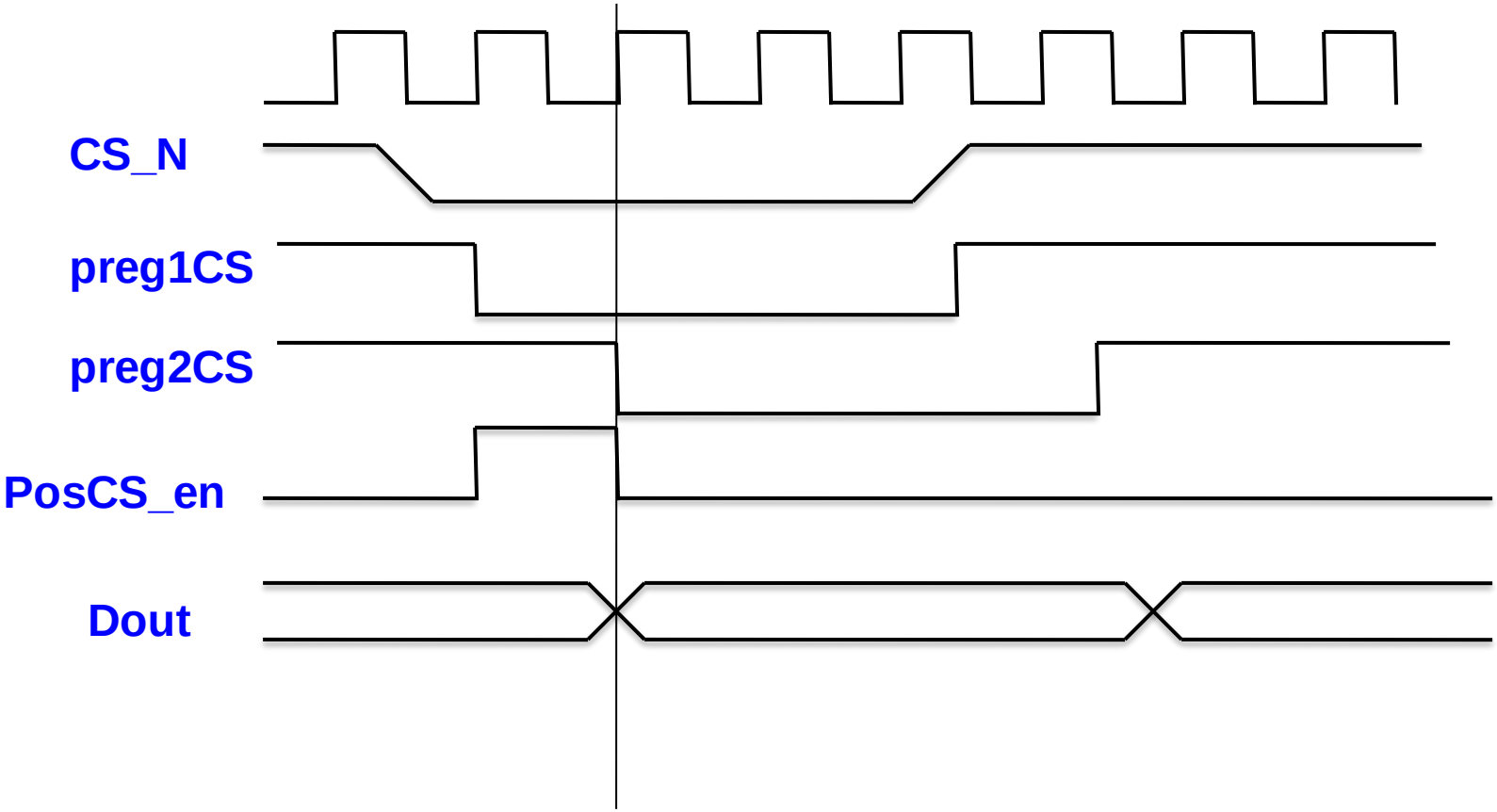


则会出现亚稳态，得到的结果不可预测

解决方法：跨时钟域的同步设计。将PULSE和CS_N同步于时钟



解决方法：跨时钟域的同步设计。将PULSE和CS_N同步于时钟



VHDL代码：

```
en1:process(clk,rst_n,pulse)
        begin
            if(rst_n = '0') then
              preg1 <= '1'; preg2 <= '1';
            elsif(clk 'event and clk='1')
              preg1 <= pulse; preg2 <= preg1;
            endif;
            pos_en <= NOT preg2 AND preg1;
         endprocess;
en1:process(clk,rst_n,cs_n)
        begin
            if(rst_n = '0') then
              preg1CS <= '1'; preg2CS<= '1';
            elsif(clk 'event and clk='1')
              preg1CS <= pulse; preg2CS <= preg1;
            endif;
            PosCS_en <= NOT preg1CS AND preg2CS;
         endprocess;
```

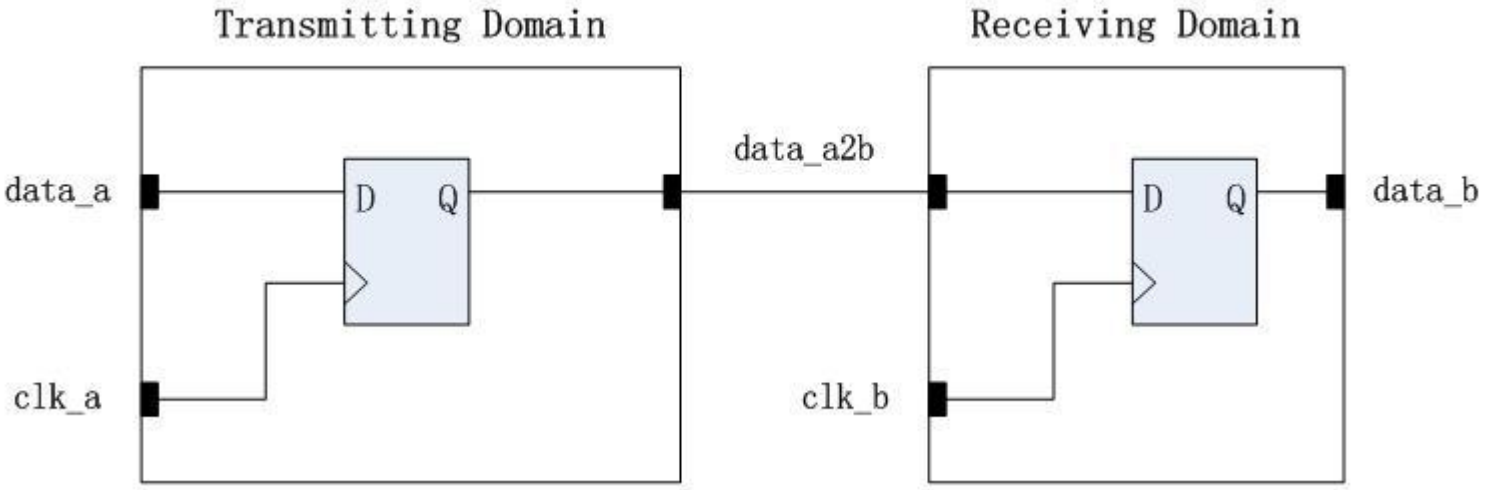
跨时钟域信号处理

跨时钟域信号处理

- ✓何谓跨时钟域：
- ✓在逻辑设计领域，只涉及单个时钟域的设计并不多。尤其对于一些复杂的应用，FPGA往往需要和多个时钟域的信号进行通信。跨时钟域所涉及的两个或多个时钟之间可能存在相位差，也可能没有任何频率关系，即通常所说的不同频不同相。

跨时钟域信号处理

- ✓ 如图是一个跨时钟域的异步通信实例，发送域和接收域的时钟分别是clk_a和clk_b。这两个时钟频率不同，并且存在一定的相位差。对于接收时钟域而言，来自发送时钟域的信号data_a2b有可能在任何时刻变化。



跨时钟域信号处理

- ✓ 对于上述的异步时钟域通信，设计者需要做特殊的处理以确保数据可靠的传输。由于两个异步时钟域的频率关系不确定，触发器之间的建立时间和保持时间要求也无法得到保证。如果出现建立时间或者保持时间违规，接收域将会采样到处于亚稳态数据，那么后果可想而知。

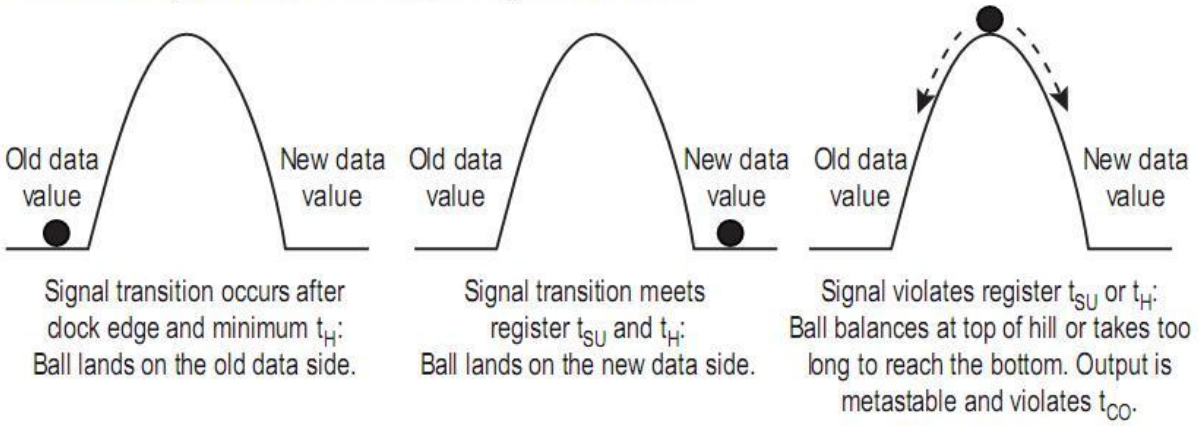
跨时钟域信号处理

- ✓ 什么是亚稳态：

  如果数据输入不满足建立时间和保持时间，那么触发器的晶体管就不能被可靠地设置成代表逻辑0或逻辑1的电压。不是确定在高或低电平就是被设到有效水平之前，晶体管就可能停留在一个中间电压，这就叫亚稳态。

- ✓ 在同步系统中，输入信号总是能够达到寄存器的时序要求，所以亚稳态不会发生。亚稳态问题通常发生在一些跨时钟域信号的传输上。

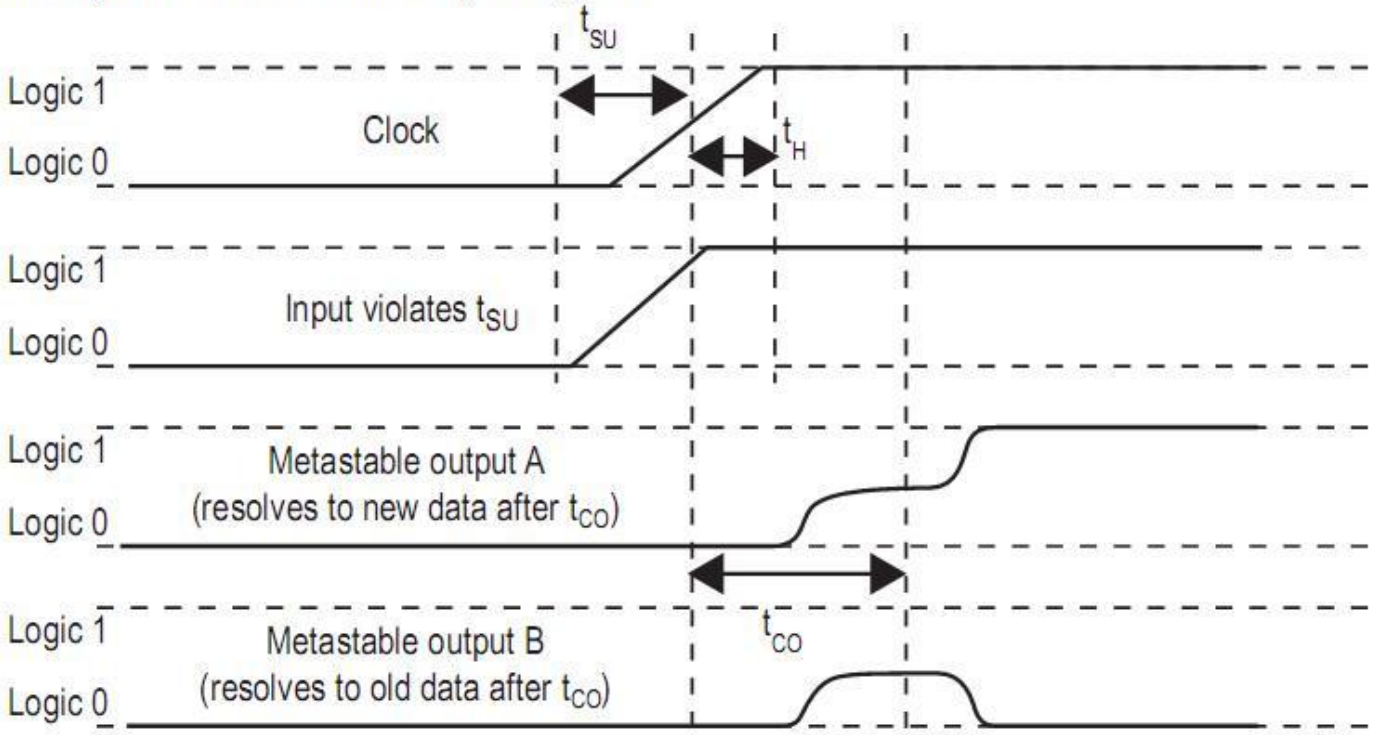
Figure 1. Metastability Illustrated as a Ball Dropped on a Hill



跨时钟域信号处理

如图很好地阐释了亚稳态信号

Figure 2. Examples of Metastable Output Signals

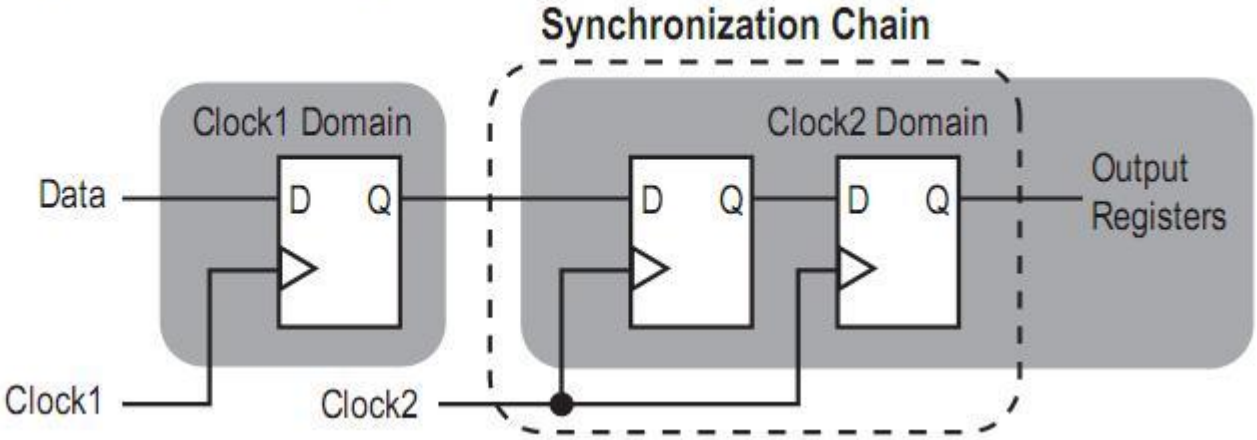


在A和B两种情况下，输出变化稳定的时间都远超寄存器固有的传播延时。

跨时钟域信号处理

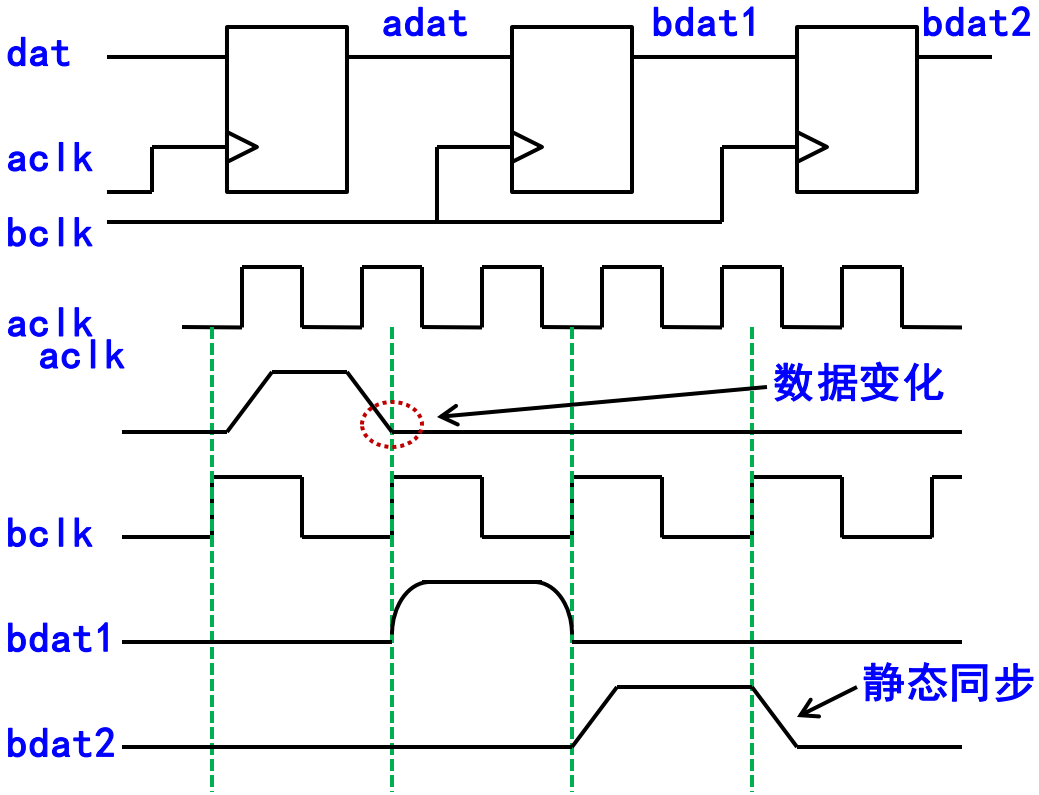
- ✓ 为了尽可能减少异步信号传输中由于亚稳态引发的问题，设计者通常在目的时钟域中使用一串连续的寄存器（同步寄存器链或者同步装置）将信号同步

Figure 3. Sample Synchronization Register Chain



跨时钟域信号处理

- 同步寄存器到寄存器路径的时序余量，也就是亚稳态信号达到稳定的最大时间，也被认为是亚稳态持续时间。



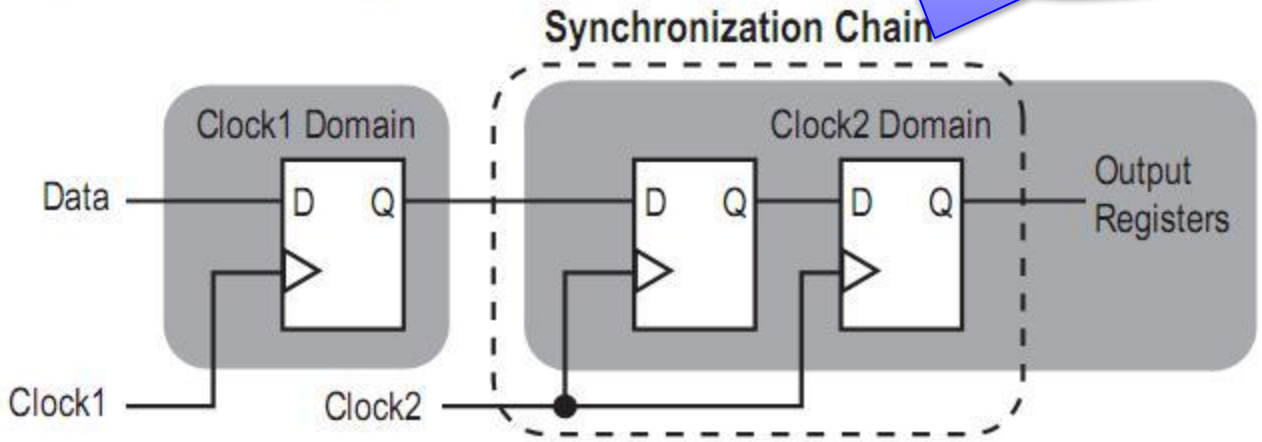
双锁存器解决亚稳态

跨时钟域信号处理

- ✓ 从本质将，增加一级寄存器是用来防止其他电路发现这个亚稳态信号，并且一旦同步信号有机会稳定下来，就把它传输下去。

只增加一级够吗？

Figure 3. Sample Synchronization Register Chain



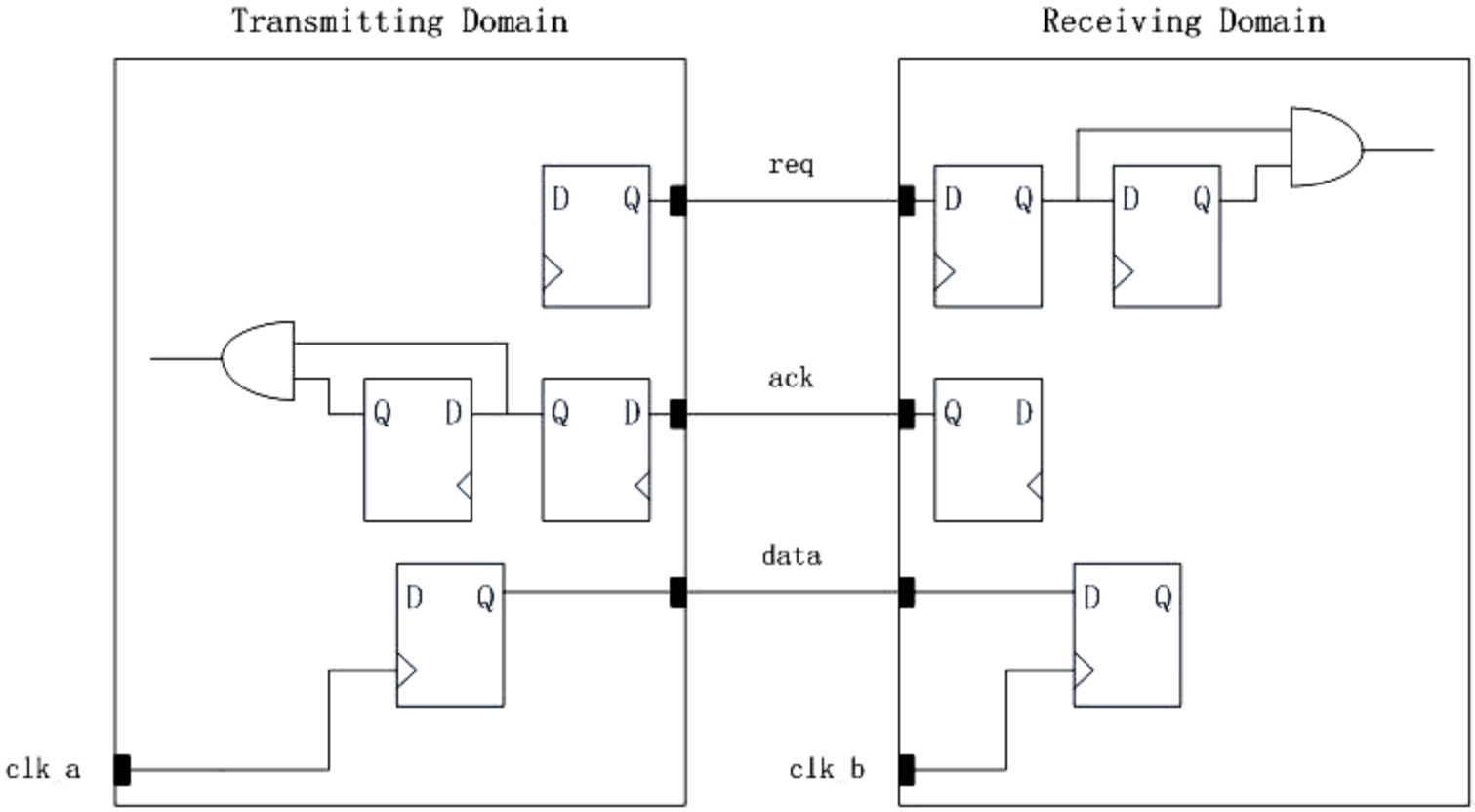
跨时钟域信号处理

如何有效的进行跨时钟域的信号传输呢？

- 同步寄存器链
- 时钟同步设计
- 专用握手信号
- 借助存储器（FIFO结构）
- 相位控制
- 分割同步模块

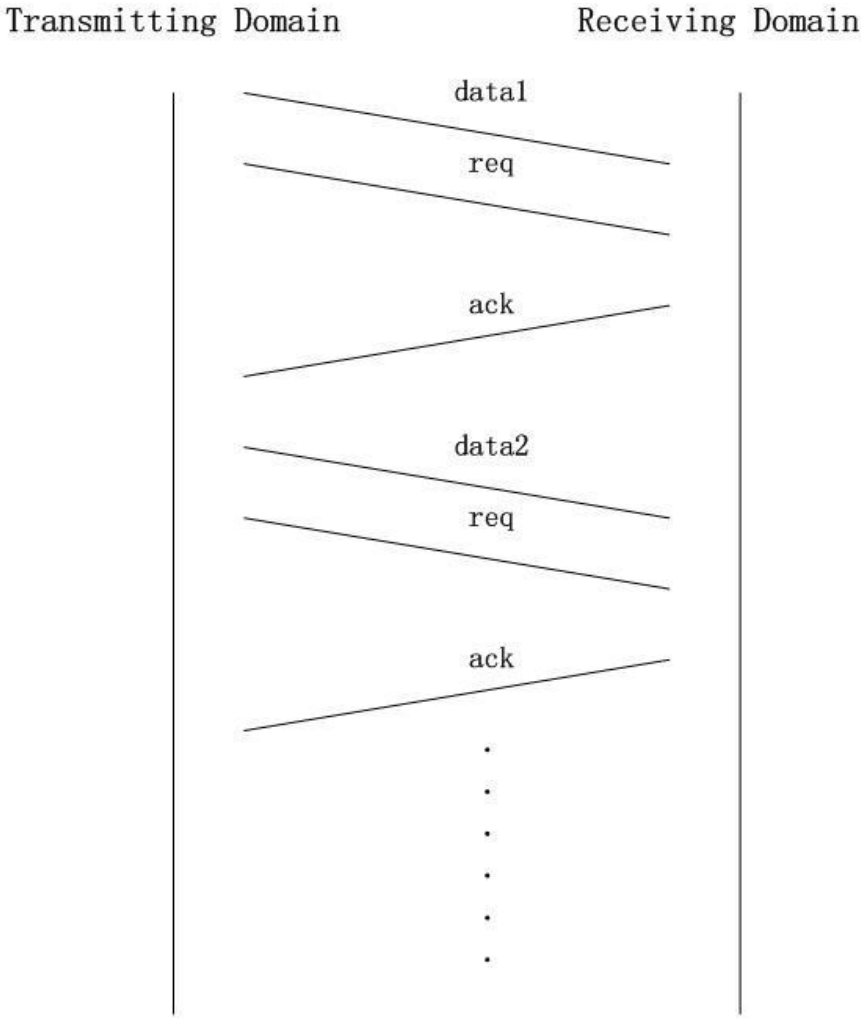
专用握手信号

- ✓ 如图是一个基本的握手通信方式。所谓握手，意即通信双方使用了专用控制信号进行状态指示。这个控制信号既有发送域给接收域的，也有接收域给发送域的。



专用握手信号

使用握手协议方式处理跨时钟域数据传输，只需要对双方的握手信号（req和ack）分别使用脉冲检测方法进行同步。在具体实现中，假设req、ack、data总线在初始化时都处于无效状态，发送域先把数据放入总线，随后发送有效的req信号给接收域。接收域在检测到有效的req信号后锁存数据总线，然后回送一个有效的ack信号表示读取完成应答。发送域在检测到有效ack信号后撤销当前的req信号，接收域在检测到req撤销后也相应撤销ack信号，此时完成一次正常握手通信。



专用握手信号

优点：

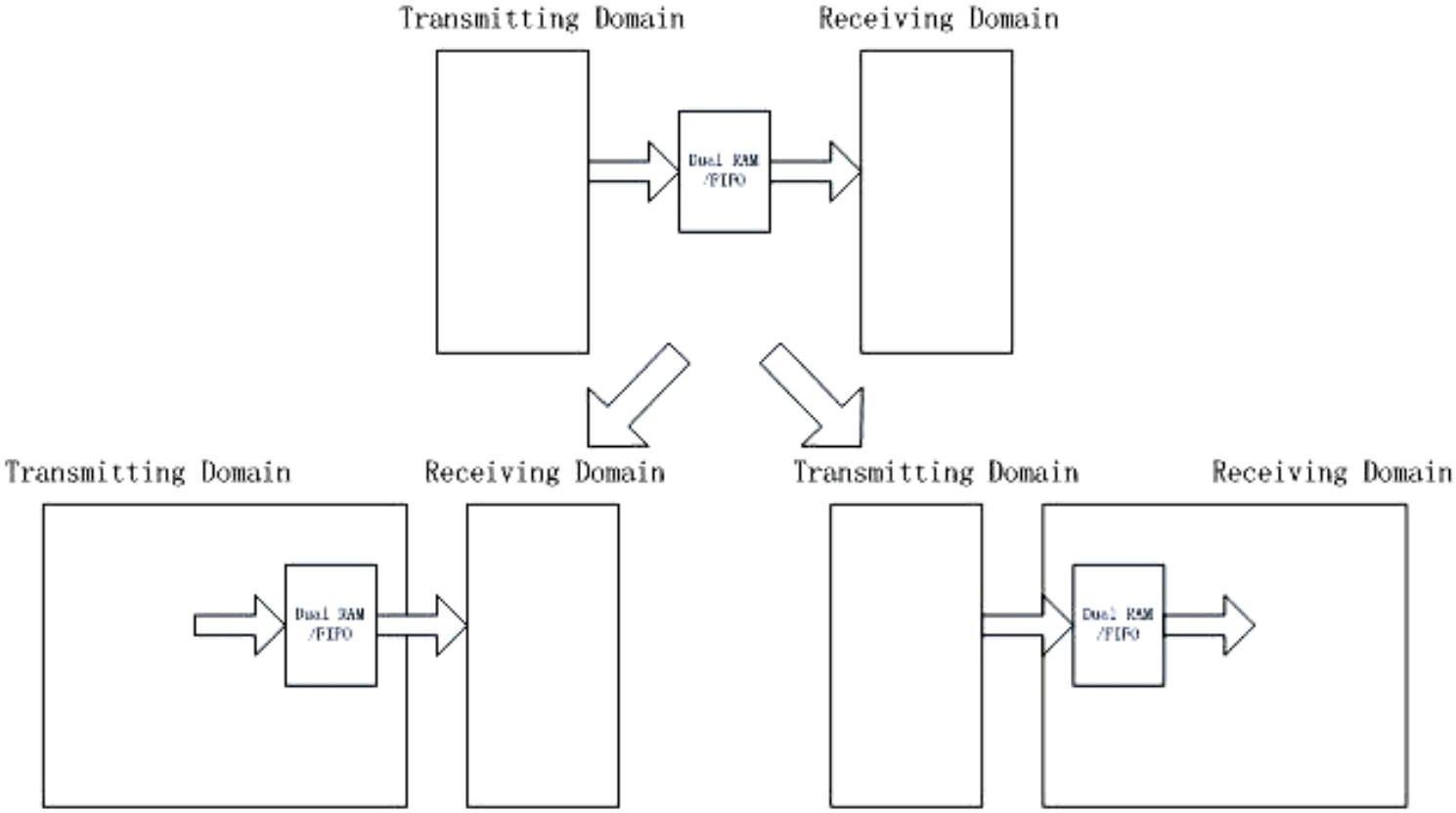
该方式能够使接收到的数据稳定可靠，有效的避免了亚稳态的出现。

限制：

但控制信号握手检测会消耗通信双方较多的时间，不适于高速设计。

借助存储器(FIFO结构)

✓为了达到可靠的数据传输，借助存储器来完成跨时钟域通信也是很常用的手段。在两个处理器间添加一个双口RAM或者FIFO来完成相互间的数据交换是很常见的做法。



借助存储器(FIFO结构)

优点：

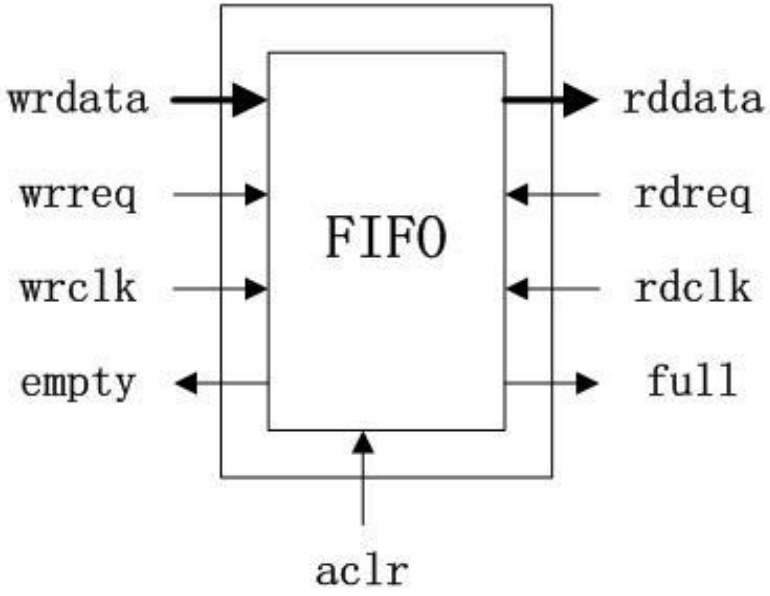
- ✓双口RAM更适合于需要互通信的设计，只要双方对地址做好适当的分配，那么剩下的工作只是控制好存储器的读写时序。
- ✓设计者不需要再花时间和精力考虑如何处理同步问题。
- ✓大大提高通信双方的数据吞吐率，它不像握手信号和逻辑同步处理机制那样在同步设计上耗费太多的时钟周期。

限制：

- ✓速度受限于存储器本身的速度上限。
- ✓使用的便利以付出更多的存储单元为代价。

借助存储器(FIFO结构)

常见的异步FIFO接口如图所示，FIFO两侧会有相对独立的两套控制总线。若写入请求wrreq在写入时钟wrclk的上升沿处于有效状态，那么FIFO将在该时钟沿将锁存写入数据总线wrdata。同理，若读请求rdreq在读时钟rdclk的上升沿处于有效状态，那么FIFO将把数据放置到读数据总线rddata上，外部逻辑一般在下一个有效时钟沿读取该数据。



FIFO一般还会有指示内部状态的一些接口信号，如图中的空标志位empty、满标志位full，甚至还会有用多位数据线表示的FIFO当前数据量，这些状态标志保证了读写控制不出现空读和满写的情况。

借助存储器(FIFO结构)

FIFO的重要参数

FIFO的宽度：指的是FIFO一次读写操作的数据位。

FIFO的深度：指的是FIFO可以存储多少个N位的数据（如果宽度为N）。

满标志：用以阻止FIFO的写操作继续向FIFO中写数据而造成溢出。

空标志：用以阻止FIFO的读操作继续从FIFO中读出数据而造成无效数据的读出。

读时钟：读操作所遵循的时钟，在每个时钟沿来临时读数据。

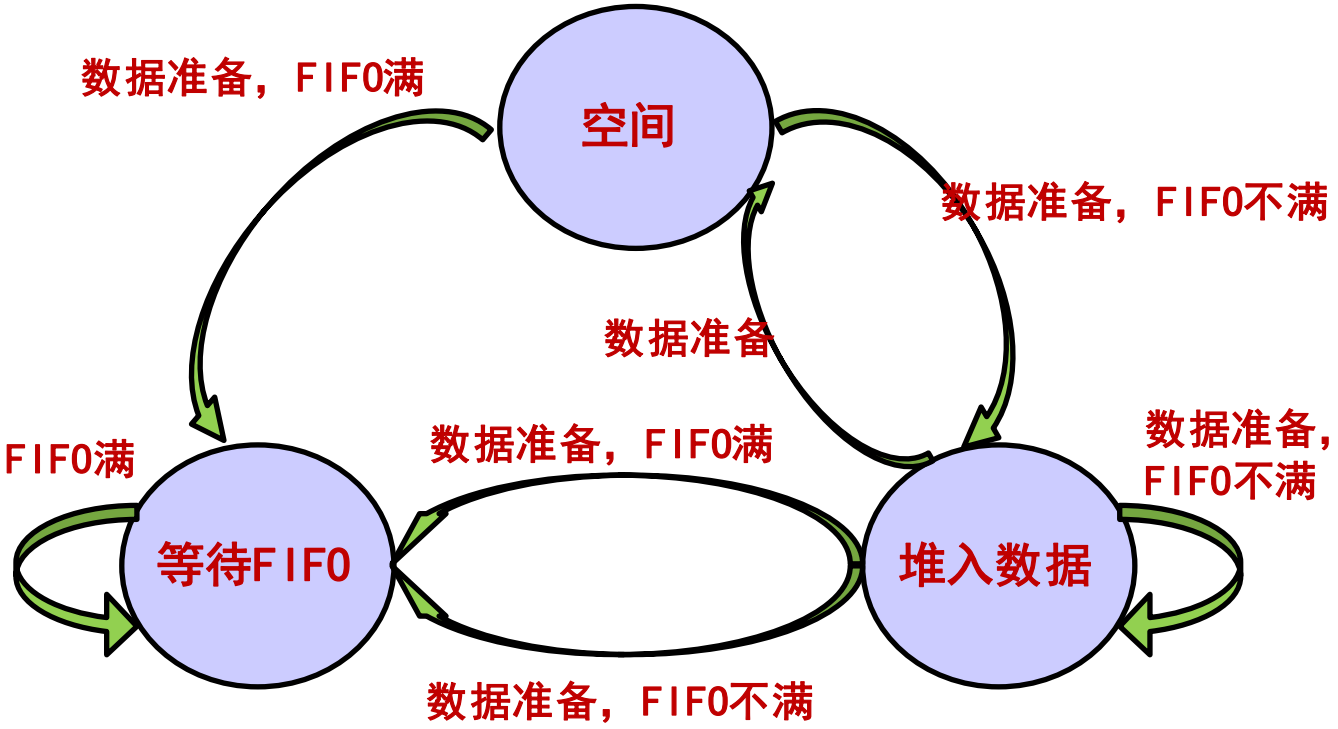
写时钟：写操作所遵循的时钟，在每个时钟沿来临时写数据。

读指针：指向下一个读出地址。读完后自动加1。

写指针：指向下一个要写入的地址的，写完自动加1。

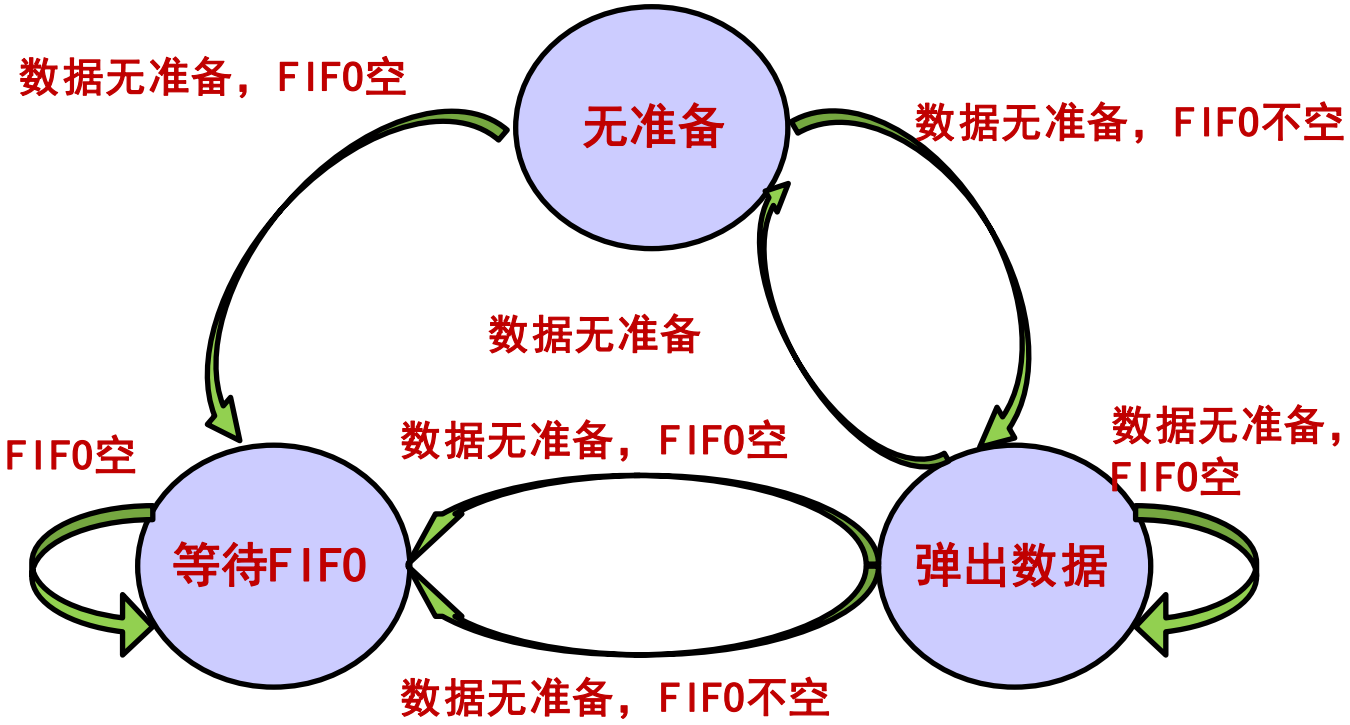
借助存储器(FIFO结构)

FIFO内部写数据握手控制



借助存储器(FIFO结构)

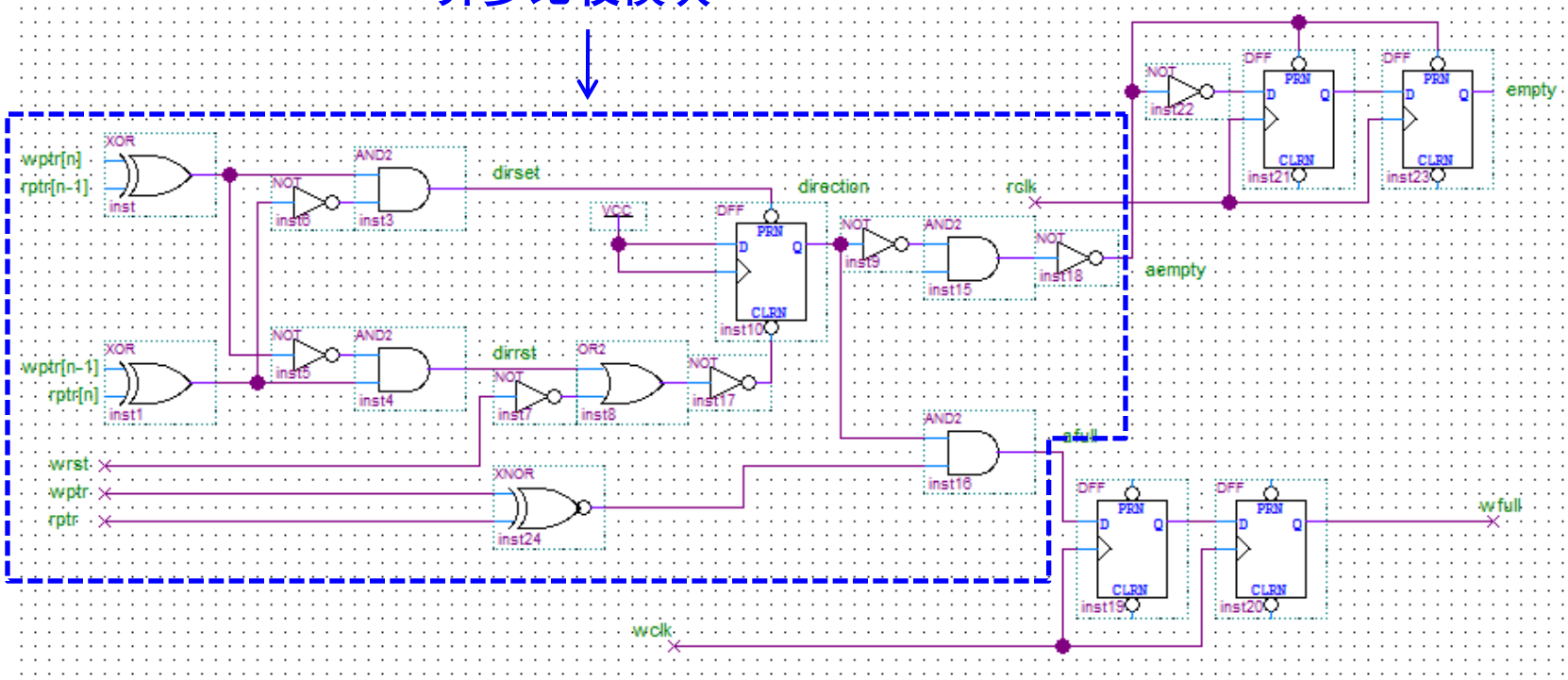
FIFO内部读数据握手控制



借助存储器(FIFO结构)

空满信号产生：

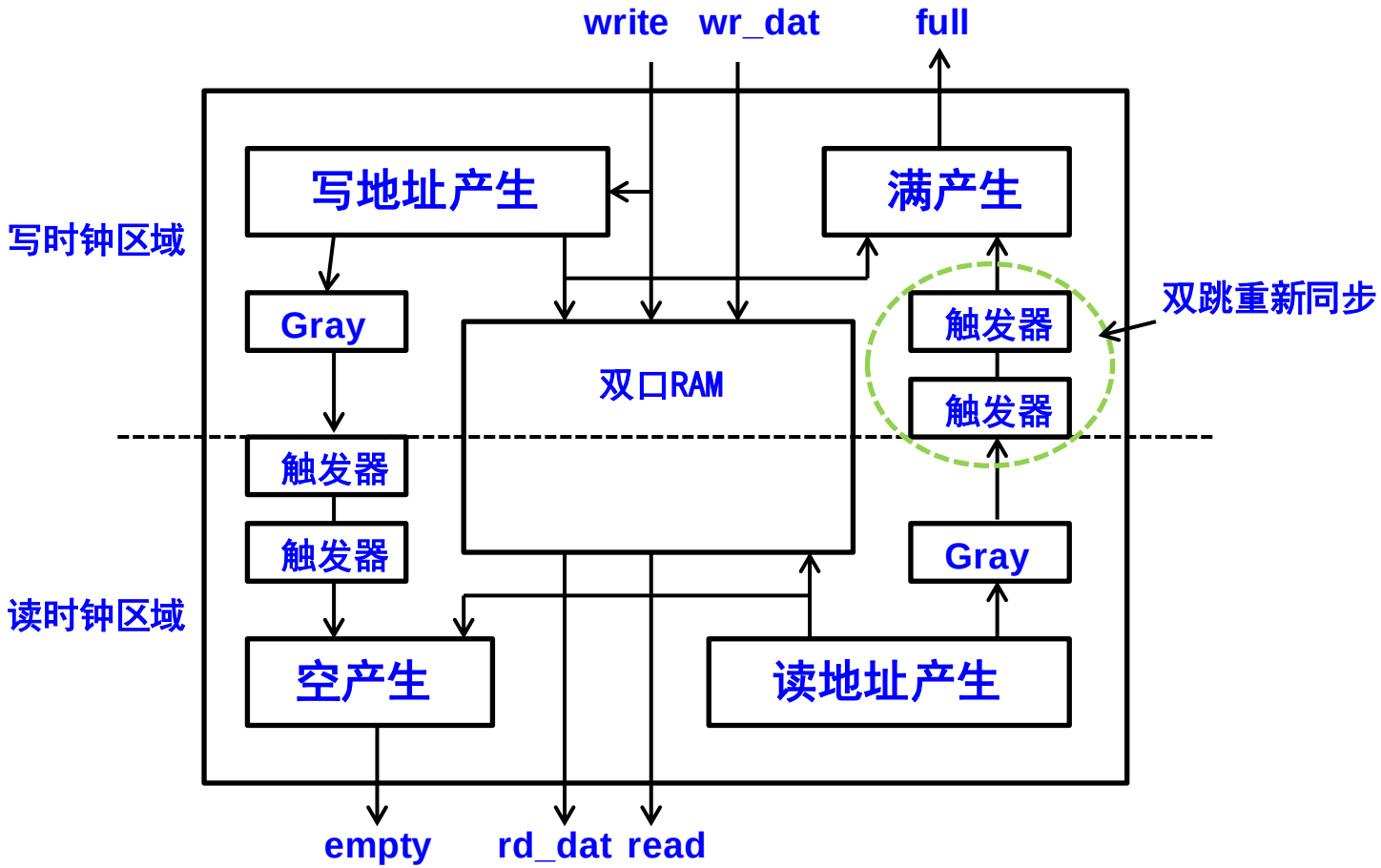
异步比较模块



空满信号产生逻辑框图

借助存储器(FIFO结构)

简化的异步FIFO

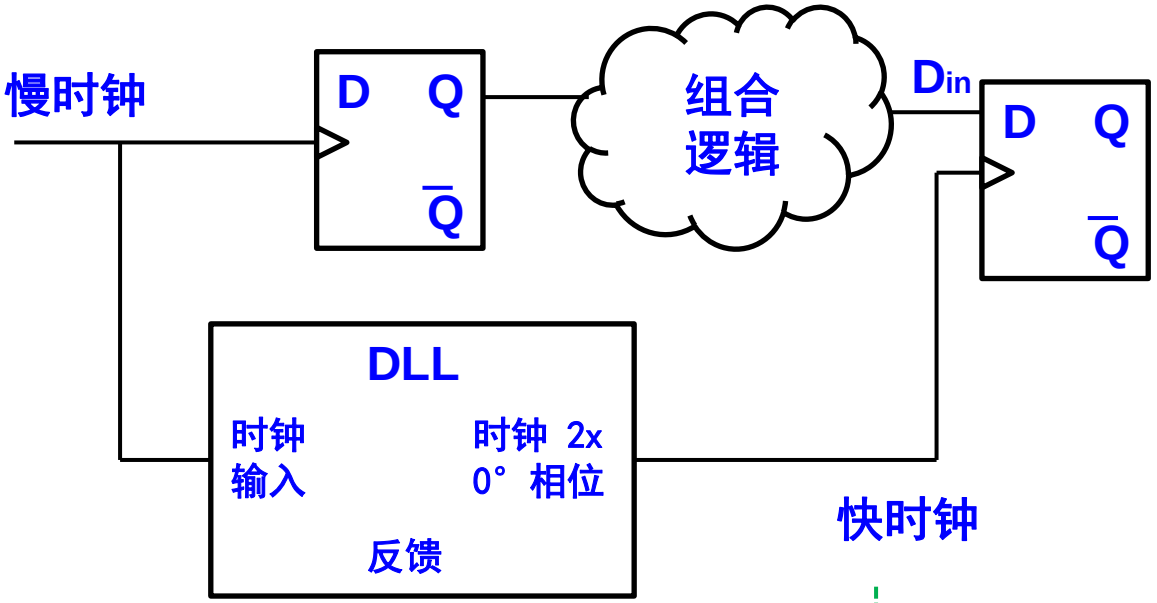


相位控制

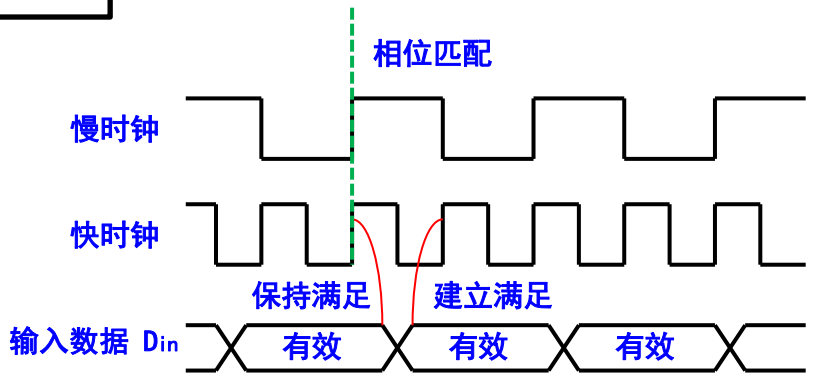
考虑不同周期的有任意相位关系的两个时钟区域。如果其中至少一个时钟是在FPGA中内部通过PLL（锁相环）或DLL（延迟锁相环）可控制的，另一个时钟与在PLL或DLL解决方案中那个时钟周期有倍数关系，那么可采用相位匹配来消除冲突。

相位控制

✓DLL调整较快的时钟区域来与较慢的时钟区域相匹配。

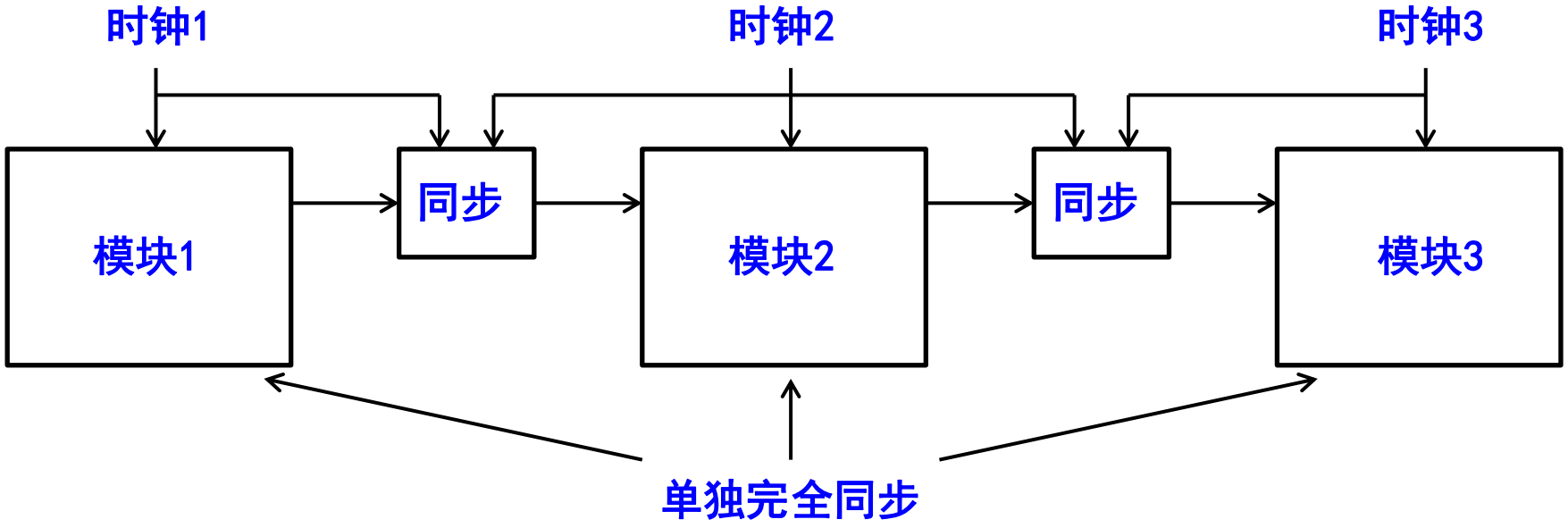


注意：必须在慢寄存器的触发电路到快寄存器的触发电路之间的传播延迟小于最快时钟的周期。



分割同步模块

✓一个好的设计，应该分割顶层设计使得同步模块包含在任何功能模块外部的各个模块中。这将助于达到在逐个模块基础上理想的时钟区域。



✓同步寄存器应该分割为功能模块外部的独立模块。

分割同步模块

✓每个基本模块的时序分析变为一致，因为它是完全同步。
✓时序在其应用到整个同步模块时很容易确定。