第九讲 SoC优化设计

- 同步设计优化问题
- 异步设计优化问题
- 高速设计的其他手段

声明

- 精选自一部分出版书籍、网络资料以及个人的一些设计经验，仅仅是众多设计原则和设计技巧中的沧海一粟，并且错误在所难免。
- 学无止境，讲授本部分的目的仅仅是带领入门，修行仍靠自身。要熟练使用这些技巧，并且有进一步的提高，必须经过大量的项目实践去积累。

同步设计优化问题

- 条件判断语句优化
- 多驱动与总线复用
- 毛刺的消除

条件判断语句优化

- 锁存的避免
- 无关态的使用
- 优先级问题

锁存的避免

- 例子：设计一个状态机，一共有两个状态s0, s1。输入信号为一个比特的din 。当din = ‘1’时，状态机进行状态翻转；当din = ‘0’时，状态保持为当前态。
- 描述方法1：可以将din当成状态机时钟clk的使能；本方法在此不再讲述。
- 描述方法2：按照状态机描述的三进程模板进行描述，以下是状态寄存器和次态译码进程。译码输出进程省略。

- 状态寄存器进程：
- Process(clk, reset)
- Begin
- if(reset = ‘1’)then
- Pst <= s0;
- elsif(clk’event and clk = ‘1’) then
- pst <= Nst;
- end if;
- End process;

- 次态译码进程
- Process(Pst, din)
- Begin
- case Pst is
- when s0 =>
- if(din = ‘1’) the
- Nst <= s1;
- else
- Nst <= s0;
- end if;
- when s1 => ….
- End process;

在**din = ‘0’**时，初学者往往认为状态机不翻转，则可不必再对**Nst**赋值，因为此时**Nst**和**Pst**的值一样，都为**s0**，再赋值一次显得多余。

- 事实上，这个“多余”的赋值非常致命，少了这一句，电路将出现惊人的复杂化，并且工作不正常。因为次态译码从纯组合电路变成了锁存电路。
- 在初始时刻，也就是在reset之后，如果din = ‘0’，则Nst没有获得明确的值。它只好“保持原来状态”。上述思路的漏洞在于没有考虑到初始时刻的情况，而是假定电路已经发生状态翻转之后再作分析。

锁存的避免 —— 总结

- 组合电路描述中，条件判断语句必须指明所有条件分支情况下，被赋值信号的值。
- 分支不完整，意味着电路需要在某种电平状态下，让被赋值的信号“保持原值”，这只能使用锁存电路实现。

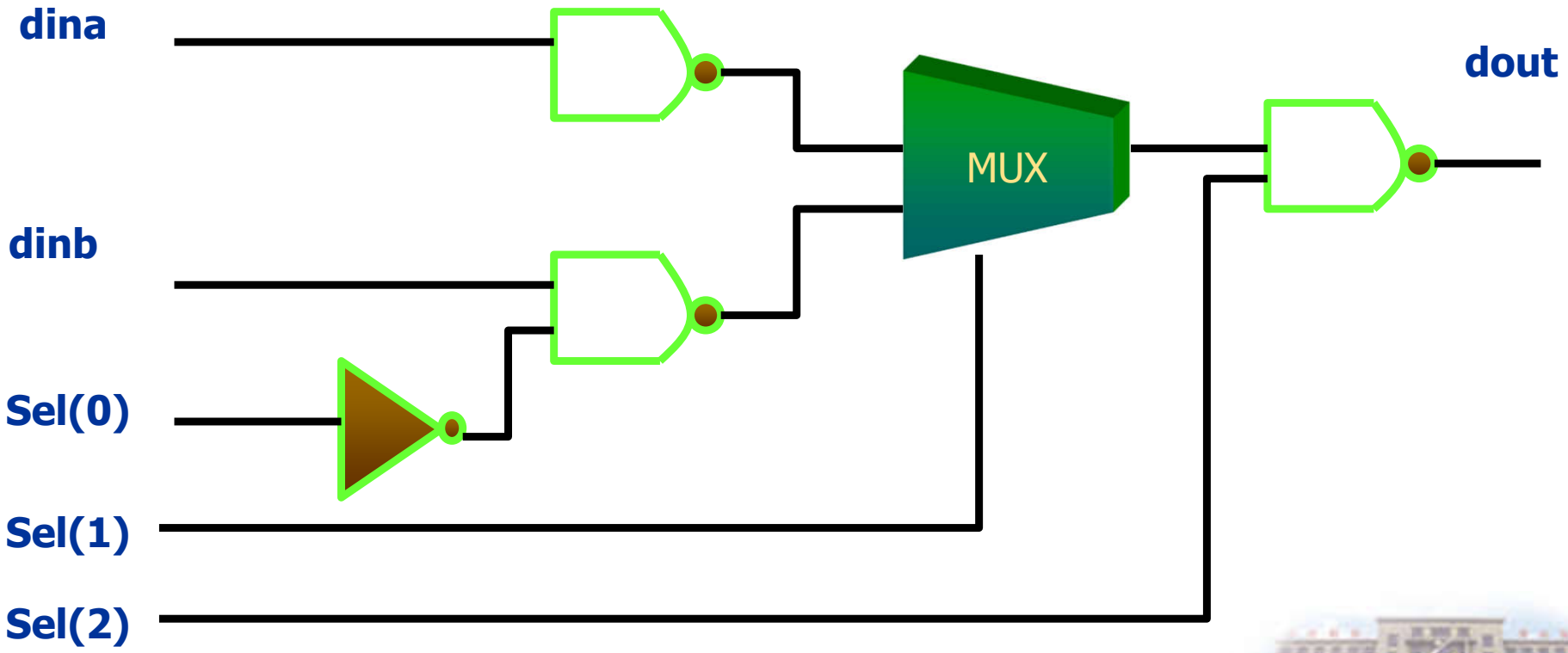
无关态的使用

- 代码段1：
- Case sel is
- 　　when “000” => dout <= dina;
- 　　when “010” => dout <= dinb;
- 　　when others => dout <= ‘0’;
- End case;

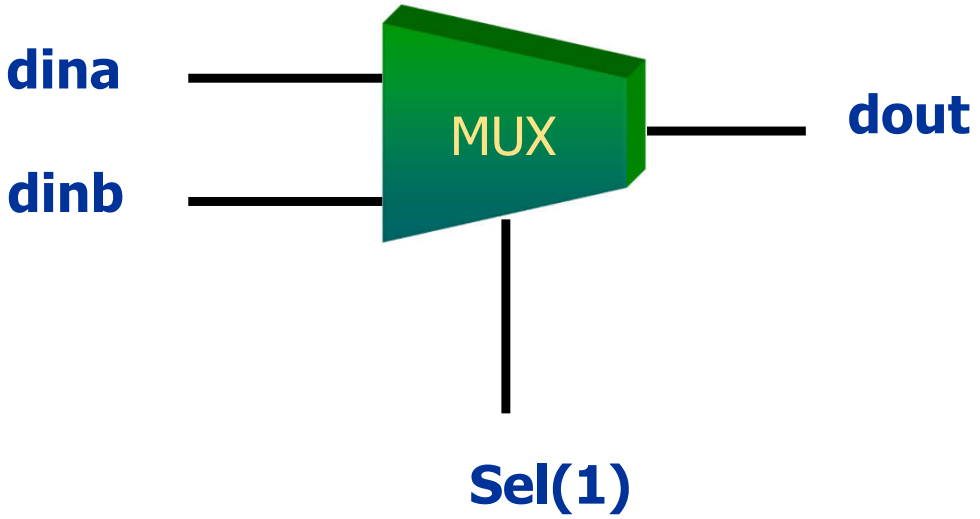
无关态的使用

- 代码段2：
- Case sel is
- 　　when “000” => dout <= dina;
- 　　when “010” => dout <= dinb;
- 　　when others => dout <= ‘—’;
- End case;

- 代码段1的综合结果：



- 代码段2的综合结果：



无关态的使用 —— 总结

- 对比以上两段代码的综合结果，可以发现，善于使用无关态‘-’来填补分支，可以引导综合工具生成很优化的电路。
- 无关态‘-’在本质上时起了冗余电路删简的作用。

优先级问题

1. Case语句无优先级，if语句有优先级。
2. Multiple if statement和single if statement具有两种不同的优先级顺序。

Single if

- Process(a)
- Begin
- if (a(0) = ‘1’) then b <= “001”;
- elsif(a(1) = ‘1’) then b <= “010”;
- elsif(a(2) = ‘1’) then b <= “100”;
- else b <= “000”;
- end if;
- End process;

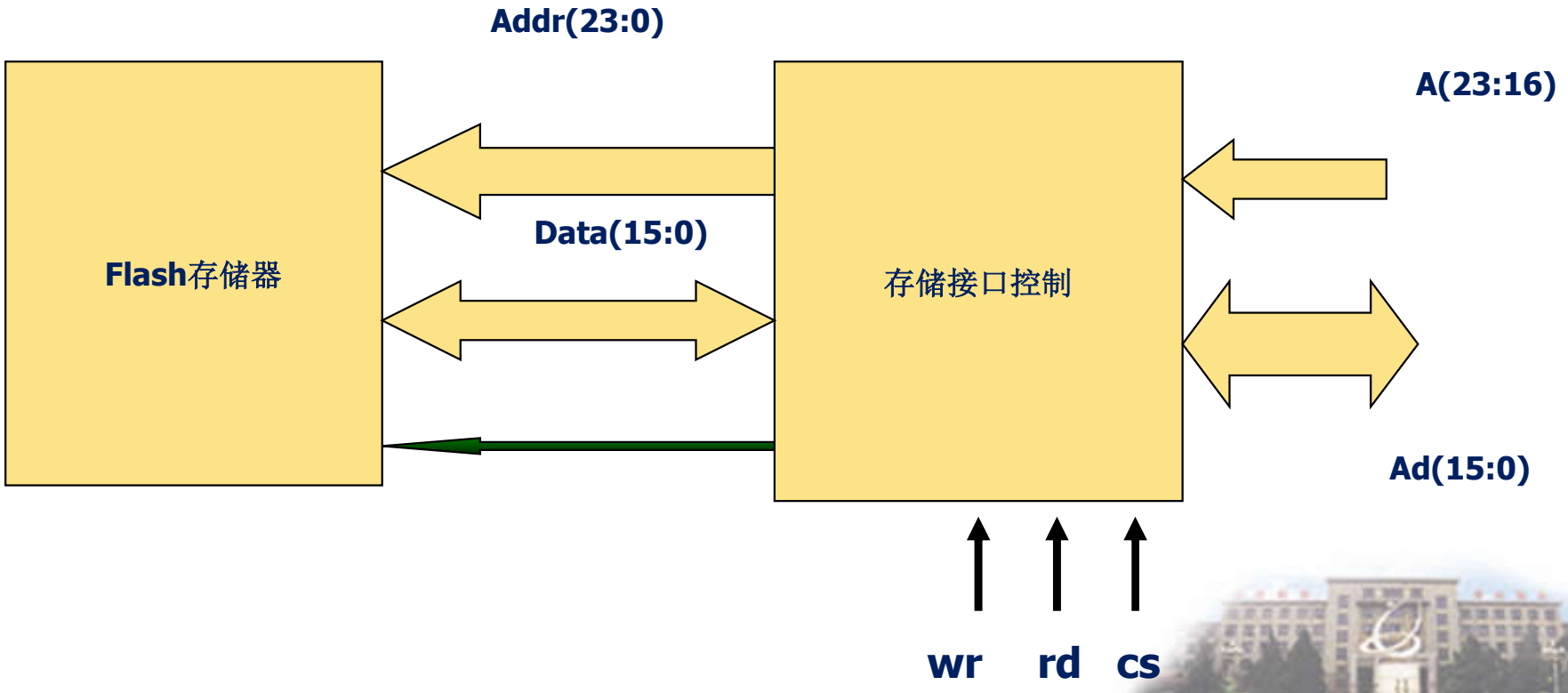
Multiple if

- Process(a)
- Begin
- b <= “000”;
- if(a (0) = ‘1 ‘) then b <= “001”; end if;
- if(a (1) = ‘1 ‘) then b <= “010”; end if;
- if(a (2) = ‘1 ‘) then b <= “100”; end if;
- End process;

多驱动与总线复用

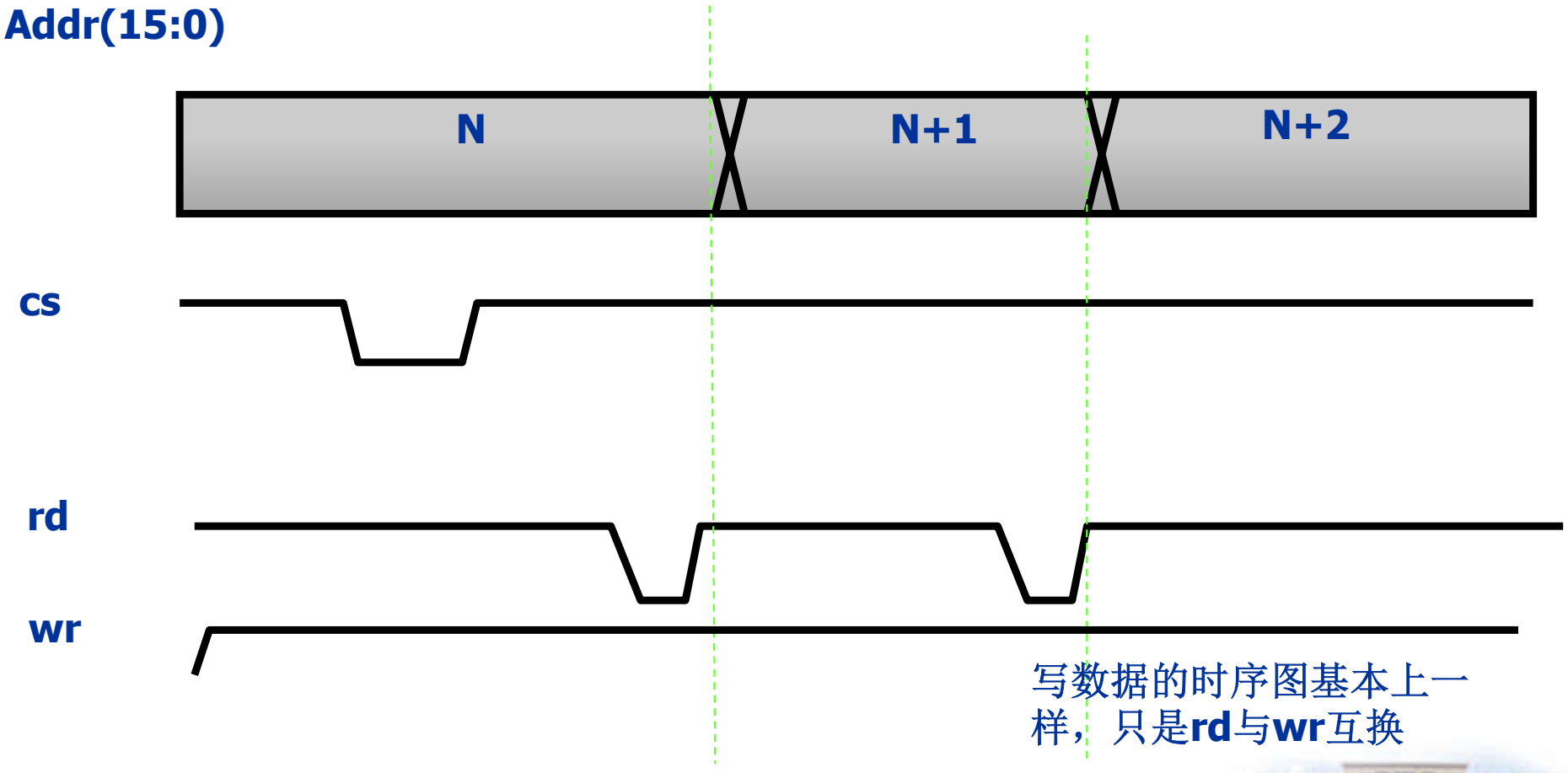
- 多驱动处理的要点：
  - 行为级思维 → 硬件思维；
  - 总线复用；
  - 线与；
  - 双向端口中的高阻态；

- 例子：任天堂游戏机的游戏卡(省略去游戏进度存储模块)内部的电路框图为：



- 利用CPLD代替接口控制芯片。请写出CPLD中，低16位地址控制部分的代码。
- 读数据时低16位地址的控制协议：当cs = ‘0’时，Addr(15:0) 即初始化为Ad(15:0)的值。Cs上升后，Addr(15:0)即被锁住，不再受Ad(15:0)的影响；在cs = ‘1’的期间，每一个rd上升沿均导致Addr(15:0)进行加一计数。
- 写数据时的控制也差不多，只需要将上面的rd换成wr即可。

读数据时的低16位地址控制时序图



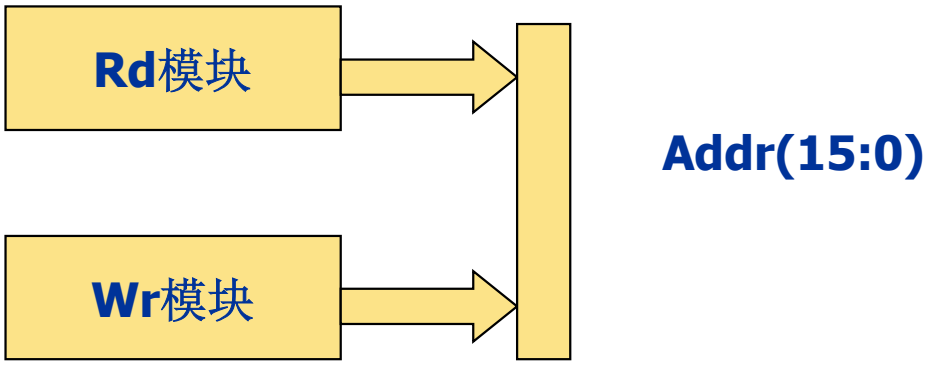
写数据的时序图基本上一样，只是rd与wr互换

- 代码实现方法1，包括读控制进程和写控制进程：

```
Rdproc: Process( cs, rd )
Begin
    if( cs = '0') then
      Addr( 15 downto 0 ) <= Ad(15 downto 0);
    elsif( rd'event and rd = '1' ) then
      Addr( 15 downto 0) <= Addr( 15 downto 0) + '1';
    end if;
End process;
```

- Wrproc: Process(cs, wr)
- Begin
- if(cs = ‘0’) then
- Addr(15 downto 0) <= Ad(15 downto 0);
- elsif(wr’event and wr = ‘1’) then
- Addr(15 downto 0) <= Addr(15 downto 0) + ‘1’;
- end if;
- End process;

- 以上代码的描述实际上“照搬”控制协议中的描述，并且符合人的“直觉思维”。从“软件”的角度来看，确实没有任何算法上的错误。
- 但是综合时，出现了多驱动的错误。
- 原因：代码被综合成如下形式：



- 这是典型的“直觉”描述，没能生成正确的电路。
- 修正方法如下：

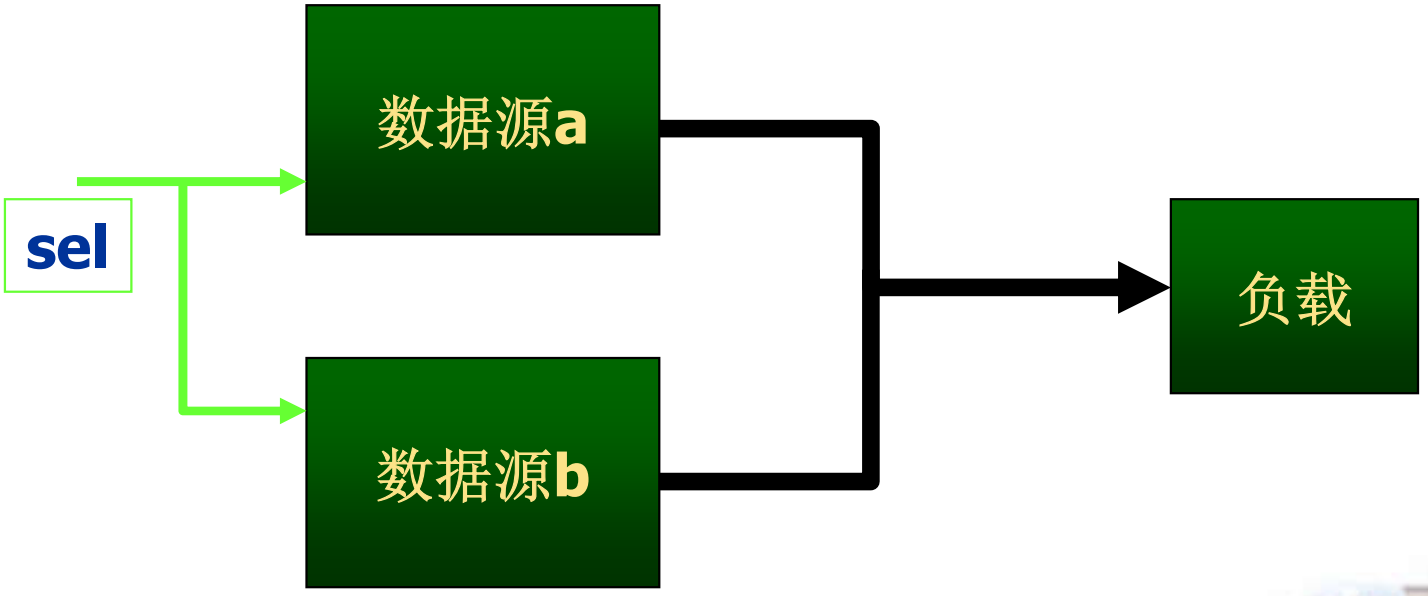
- rwproc: Process(cs, wr, rd)
- Begin
- cclk <= wr and rd;
- if(cs = ‘0’) then
- Addr(15 downto 0) <= Ad(15 downto 0);
- elsif(cclk’event and cclk = ‘1’) then
- Addr(15 downto 0) <= Addr(15 downto 0) + ‘1’;
- end if;
- End process;

软件思维→硬件思维 的转变

- 在电路描述时，必须摒弃软件思维方式，一切从硬件的角度去思考代码的描述。
- 在具体的项目实践中，必须先画好模块的接口时序图，然后画出或者在脑子里形成模块的内部原理框图，最后才是代码实现。
- 企图一开始就依靠“软件算法”思维进行代码实现，最后才分析时序和电路图，是非常不可取的。
- 硬件思维的形成，需要一定的硬件设计训练才能达到，熟练了之后才可能科学地在初始阶段完成模块划分和时序设计。

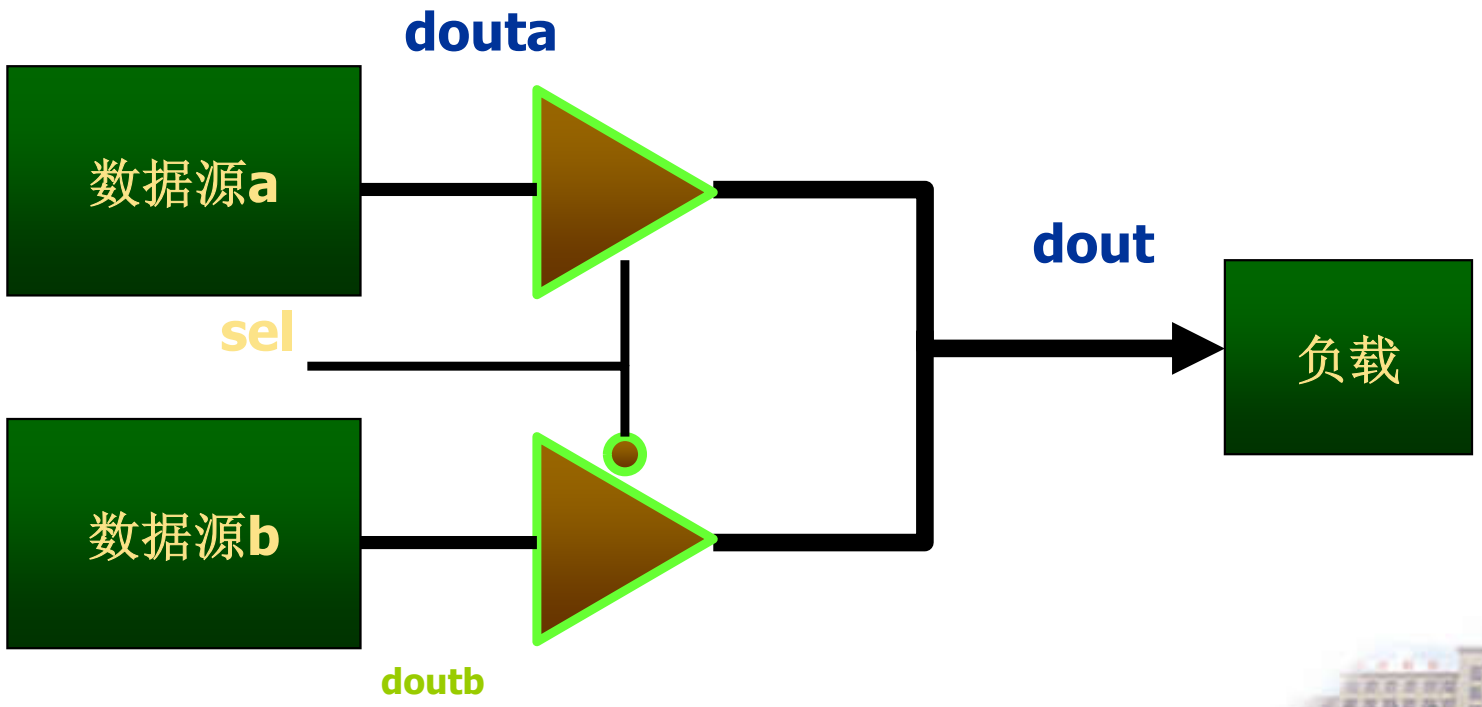
总线复用

- 两个模块输出数据给同一个负载模块的设计中，必须处理好总线复用。



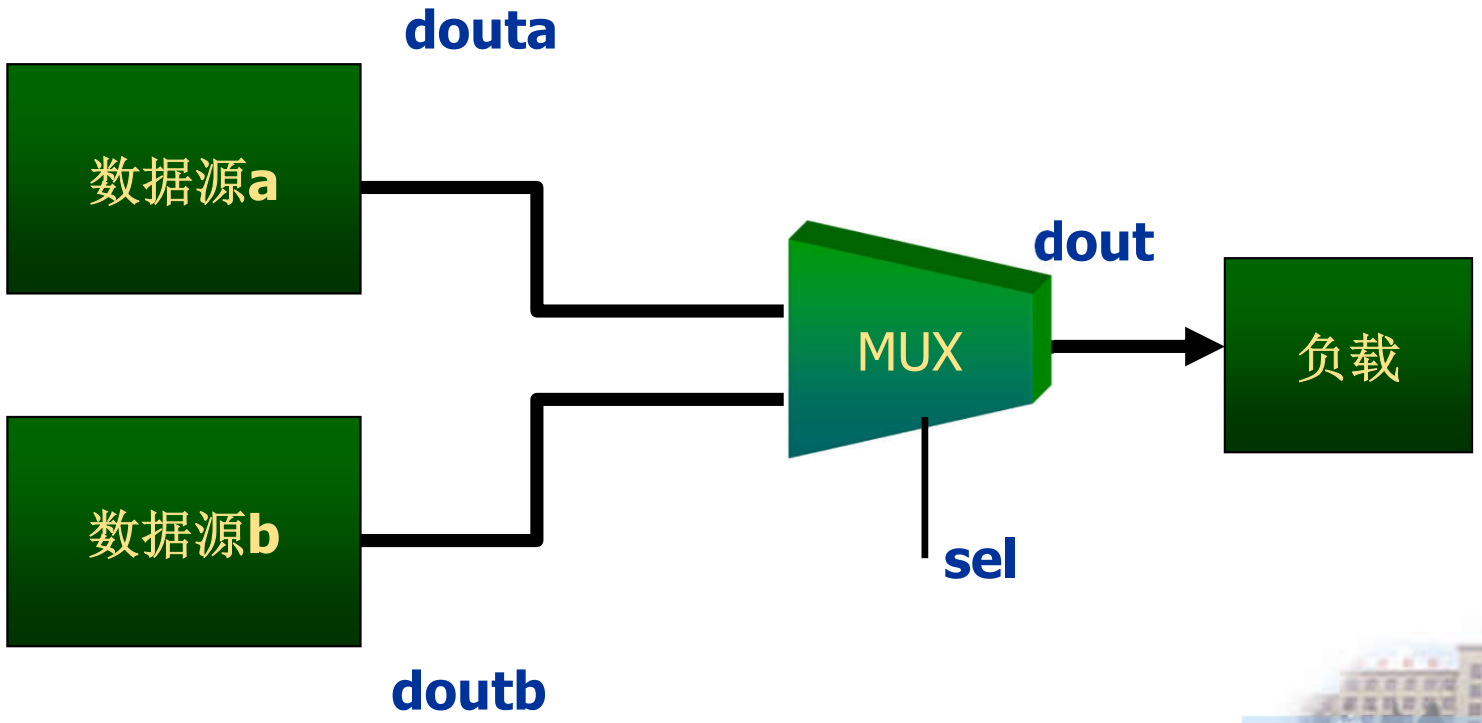
总线复用

- 加上sel的实质就是：



总线复用

- 或者



总线复用

- 代码描述：
- If (sel = ‘1’) then
- dout <= douta;
- Else
- dout <= doutb
- End if;

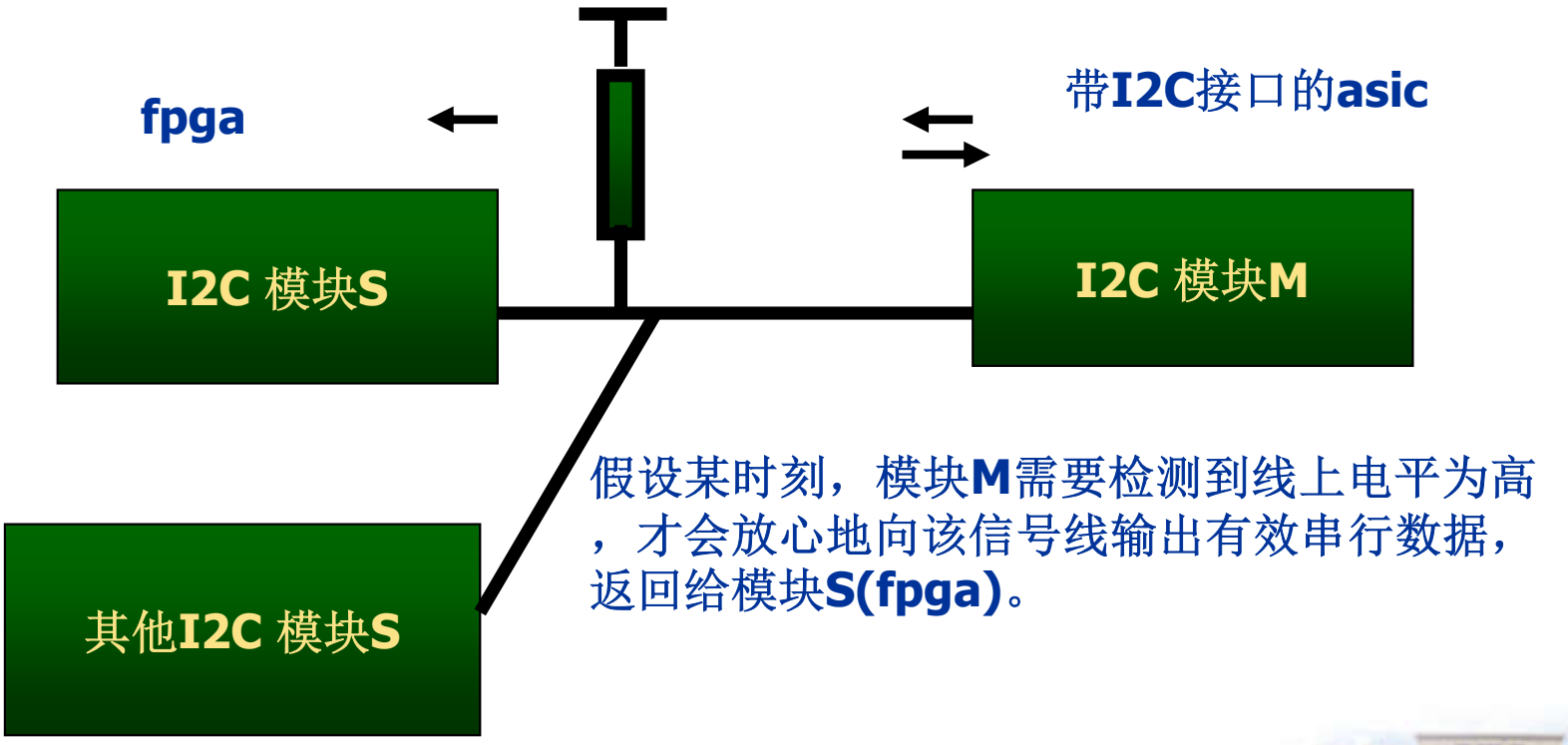
另外：尽量不要在芯片内部使用三态，某些**FPGA**不支持这种特性，在**ASIC**设计中也会带来一些测试上的问题。

线与

- CMOS工艺中只有漏极开路输出(Open drain)的电路才能实现线与；
- 对应于TTL工艺，则为集电极开路(Open collector)。
- 有部分FPGA可以将引脚设置为OD门输出，这个时候可以实现正常的线与逻辑；但是有部分FPGA没有这种输出逻辑，这时候可以用高阻输出或者直接切换到输入模式来代替OD模式，以实现安全的线与功能。当然这个时候要在芯片外部使用上下拉电阻来代替线与情况下的弱输出。

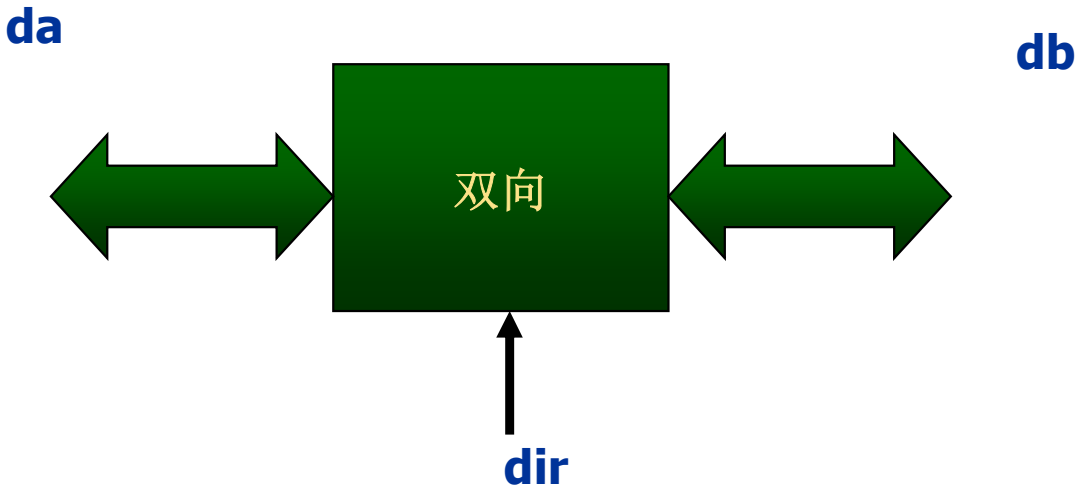
线与

- 典型案例：I2C控制模块。



双向端口中的高阻态

- 描述全双向端口，当dir = ‘1’时，数据从da流向db；dir = ‘0’时，数据从db流向da。



双向端口代码实现：分两个方向分别描述

- -- da → db:
- If(dir = ‘1’) then
- db <= da;
- Else
- db <= “ZZZZ”;
- End if;

-- db → da:

If(dir = ‘0’) then

da <= db;

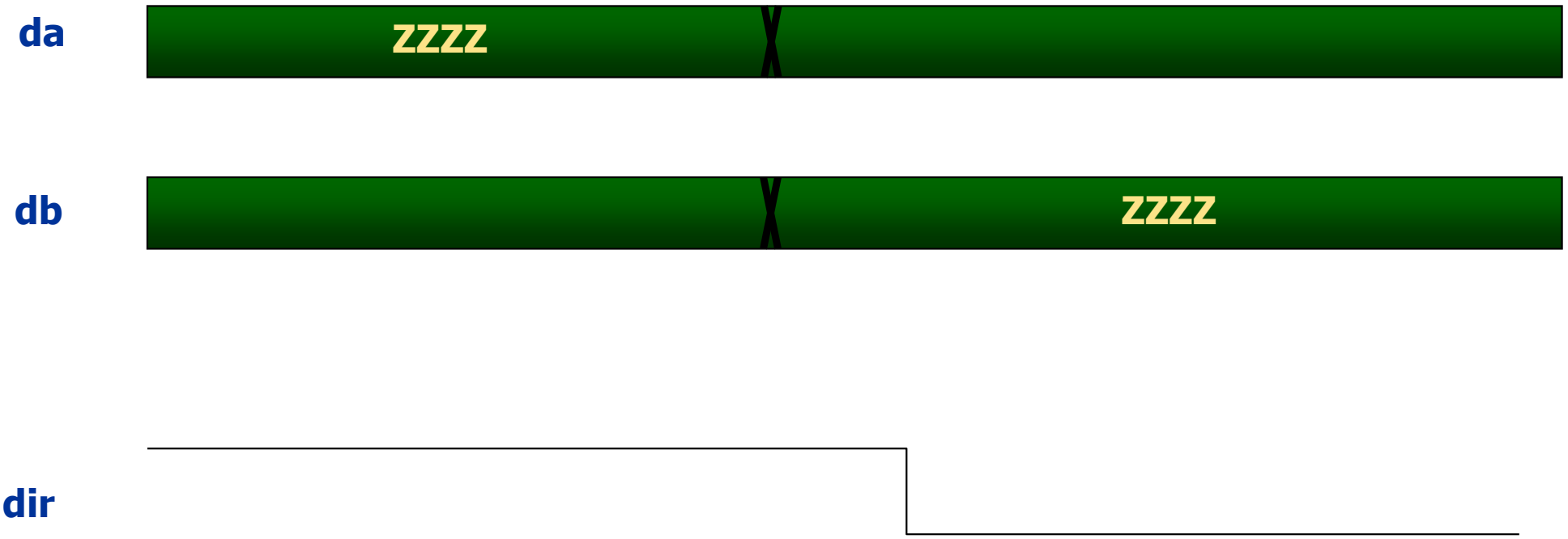
Else

da <= “ZZZZ”;

End if;

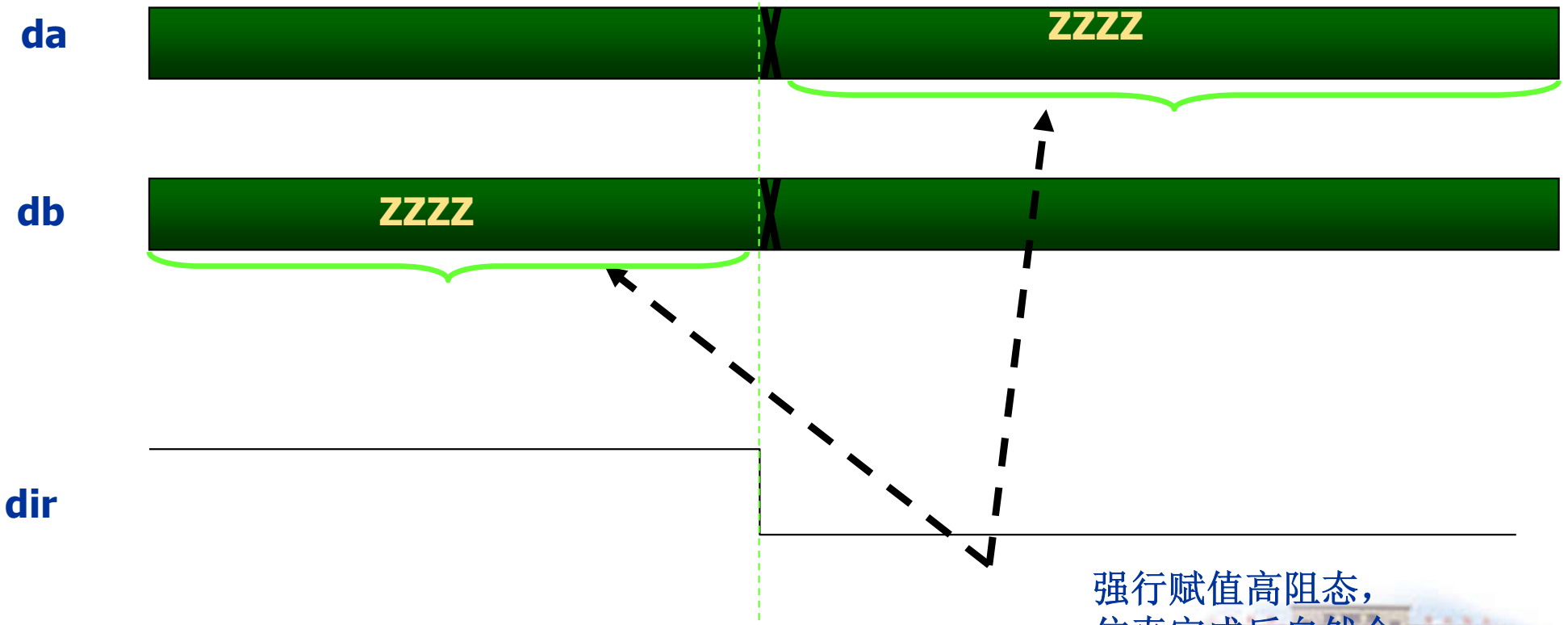
双向端口设计时序

- 时序为：



双向端口的测试矢量时序

- TestBench时序正好与RTL相反!!



强行赋值高阻态，
仿真完成后自然会
被覆盖

双向端口设计和仿真总结

- 双向的设计和描述对于初学者来说是比较难以理解的。总之
- 其设计时序规律为：可分为两个方向分别描述，对于作为被赋值者的任何一个端口，当它作为输出时，就被赋给输入的值；当它作为输入时，就被赋给高阻态；
- 其仿真时序规律为：对于任意一个端口，作为输出时，强行赋值高阻态。
- 以上规律的实质就是避免输出碰撞。(请自行画出双向口的内部电路图 ----事实上就是两个三态buffer反向并行，这就是以上描述方法的原因。)

毛刺的消除

- 毛刺产生的机理
  - 竞争和冒险
  - 延时不平衡
  - 线间干扰
- 毛刺消除的方法
  - 竞争冒险的避免
  - Gray Coding
  - 寄存器消除
  - 其他

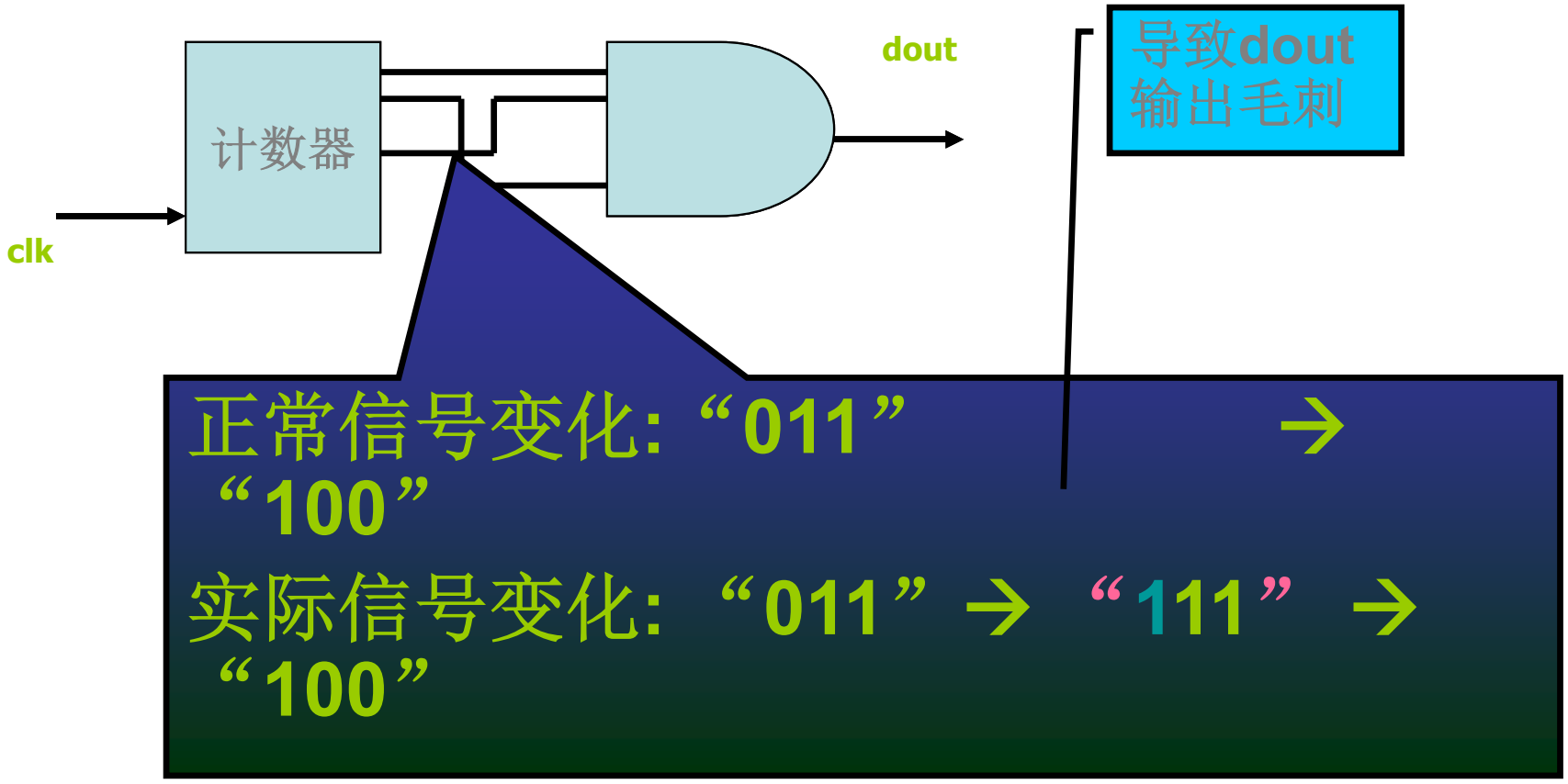
时序电路中，异步复位、时钟等输入端出现毛刺时，都会引起系统的误动作。

毛刺产生的机理

- 毛刺产生的机理
  - 竞争和冒险(请参考数字逻辑设计课程的教材)导致毛刺。
  - 组合延时，布线延时的不平衡，导致译码输出毛刺。
  - 线间的信号耦合，导致毛刺的产生。

实际上可以认为，不管是否出现竞争冒险，只要是纯组合译码输出的电路，就可能会产生毛刺。

延时不平衡导致的毛刺：



毛刺消除的方法(1)

- 竞争与冒险的避免(略)
- Gray编码方法

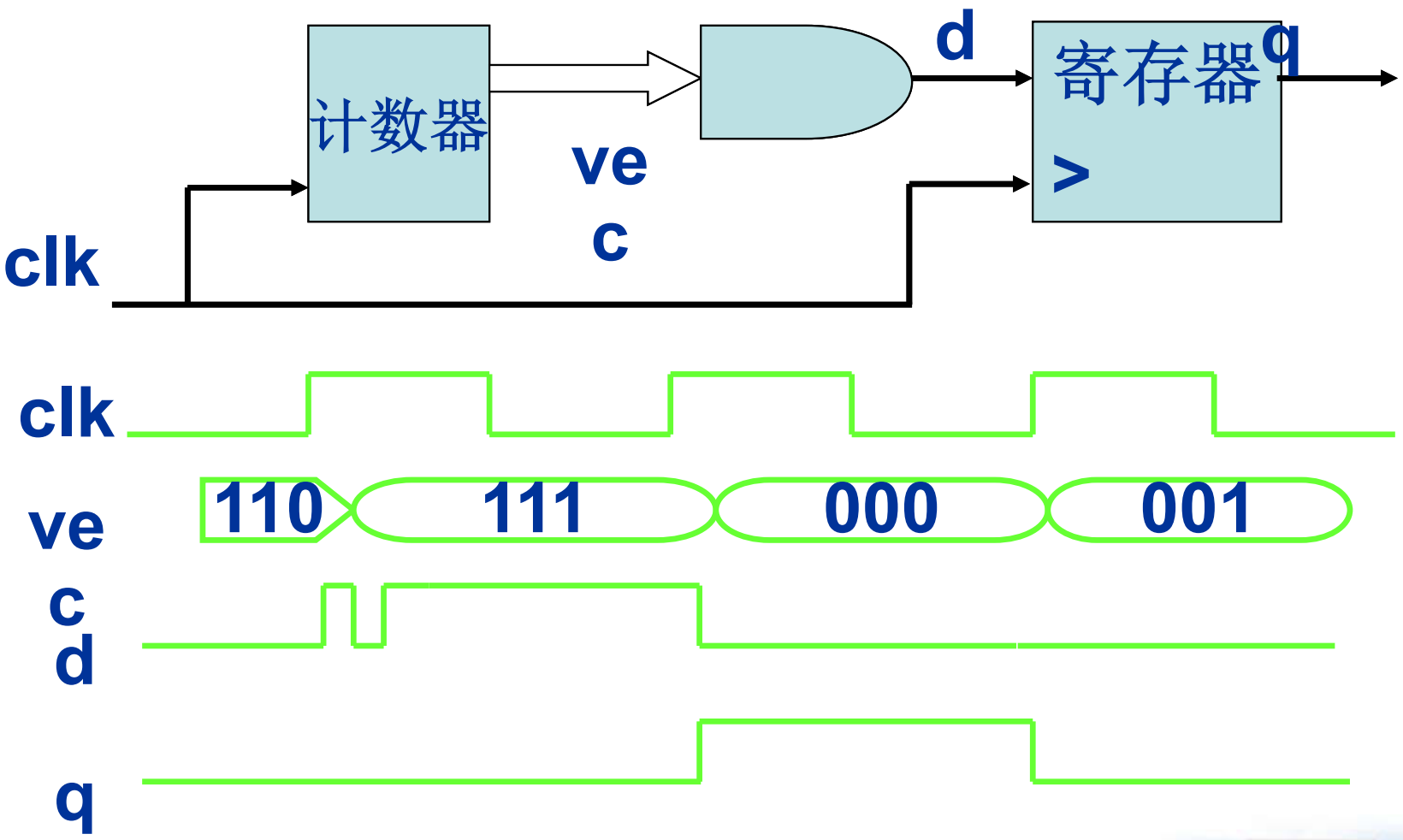
  计数器(状态机)电路中，采用Gray编码可以避免总线上的多个bit同时在一个时钟周期内翻转而导致毛刺。如上一页图中的计数器即可采用这种编码。(附：Gray码在任何相邻的两组代码中，仅有一位数码不同)

- 注：Gray码方法也是降低设计功耗的一个常用手段，因为它降低了寄存器的电平翻转率。

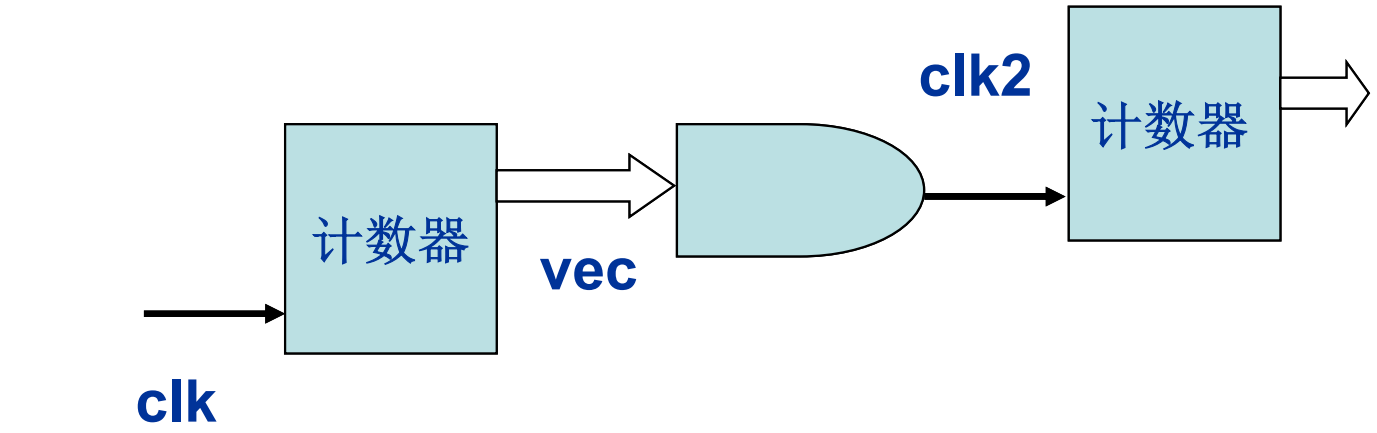
毛刺消除的方法(2)

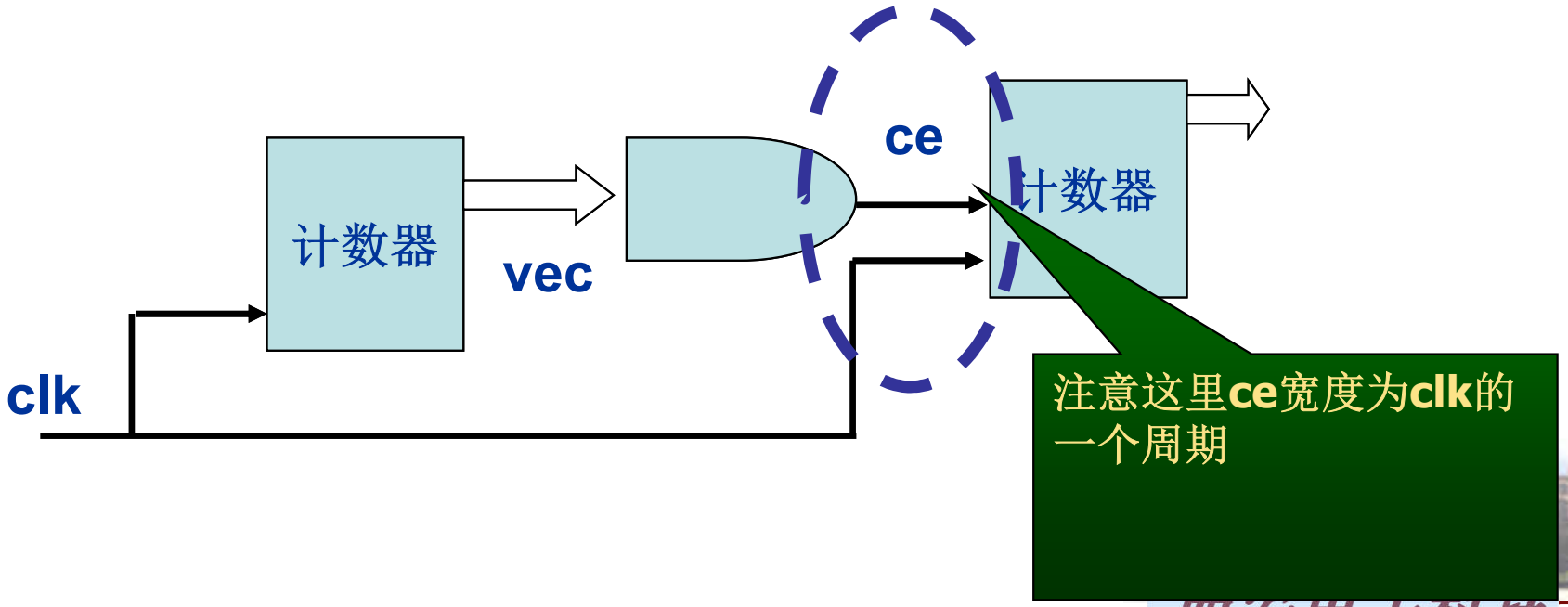
- 寄存器消除：
  - 寄存器的数据输入端D和时钟使能端EN对毛刺不敏感，因此可以利用D和EN来吸收毛刺信号。

寄存器采样(利用D端)



时钟使能吸收





同步设计优化

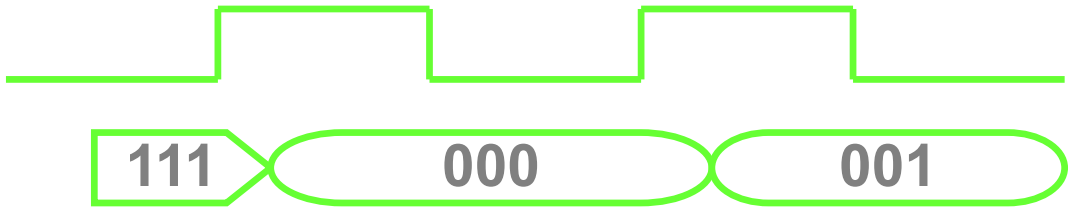
- 同步设计的概念
- 时钟质量的保证
- 路径延迟及其优化
- 时钟驱动的TestBench

同步设计的概念

- 整个系统有一个时钟，最多还有一些派生(分频，倍频等，保证与源时钟有确定的相位关系) 时钟。
- 系统中的主要存储元件大都是时钟沿敏感的元件(即寄存器)，而不是电平敏感的元件(即锁存器)。
- 数字系统设计中，应该尽量地采用纯粹的同步系统(单时钟系统)设计。异步设计会给电路带来很多不安全的因素。

同步系统的时序特点

- 信号变化都是发生在时钟沿(之后的微小时间处)，即系统中的动作基本上都是“绑定”在时钟沿上。
- 信号在敏感的时钟沿之后可能会有一段不稳定时间，随后将保持一段时间的稳定，等待下一个敏感的时钟沿的采样。



时钟质量的保证

减少时钟的**skew**
避免时钟毛刺

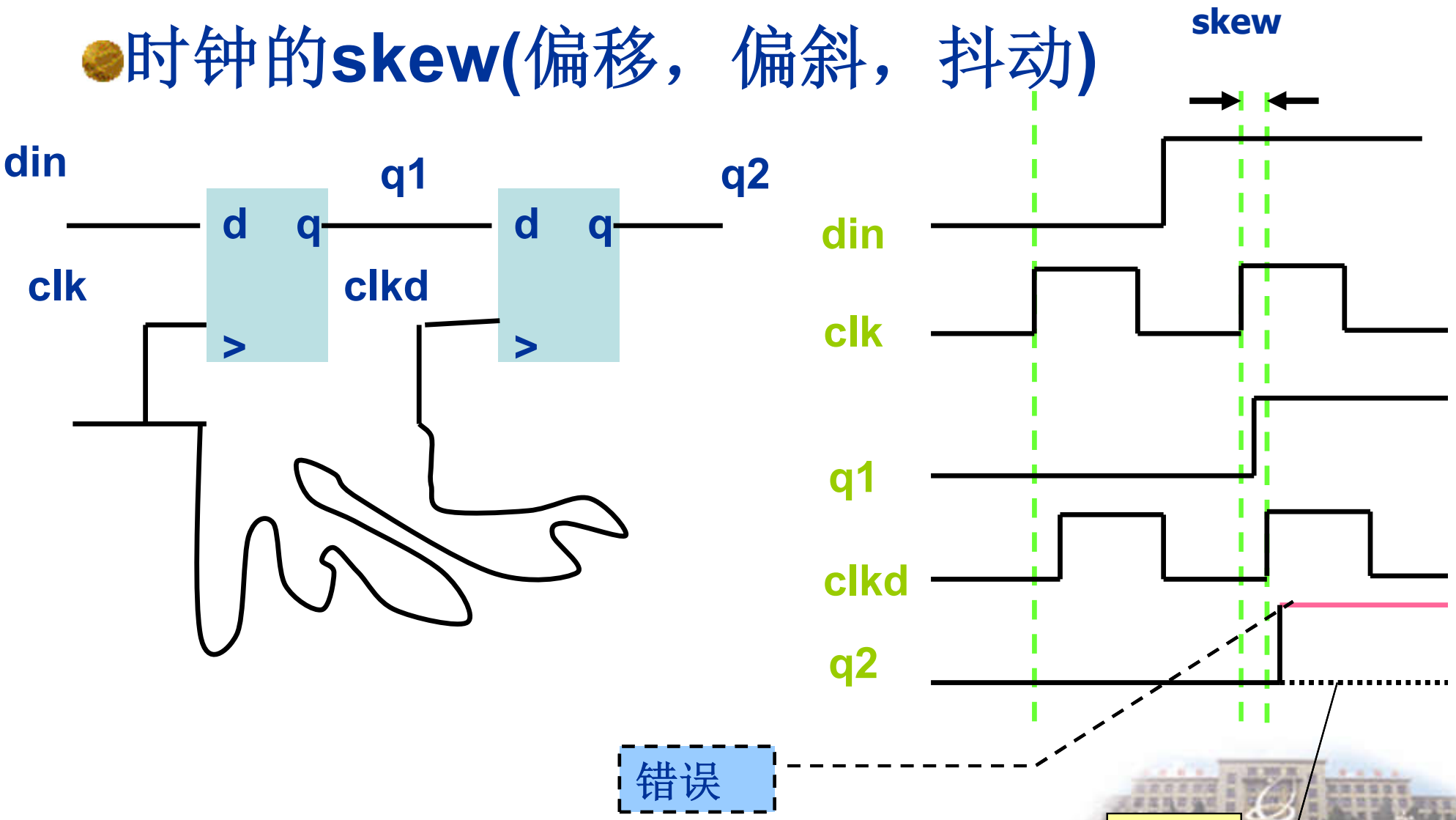
1. 时钟Skew的最小化；
2. PLL的使用；
3. 门控时钟与时钟使能；
4. 派生时钟与派生使能；
5. 其他…

同步系统中的时钟skew

- 时钟的skew(偏移，偏斜，抖动)
  - 正偏移和负偏移。当信号传输的目标寄存器在接收寄存器之前捕获正确的时钟信号，电路发生正偏移；反之，当信号传输的目标寄存器在接收寄存器之后捕获正确的时钟信号，电路发生负偏移。
  - 时钟偏移可能会造成两种时序违背：保持时间违背、建立时间违背。

同步系统中的时钟skew

- 时钟的skew(偏移，偏斜，抖动)



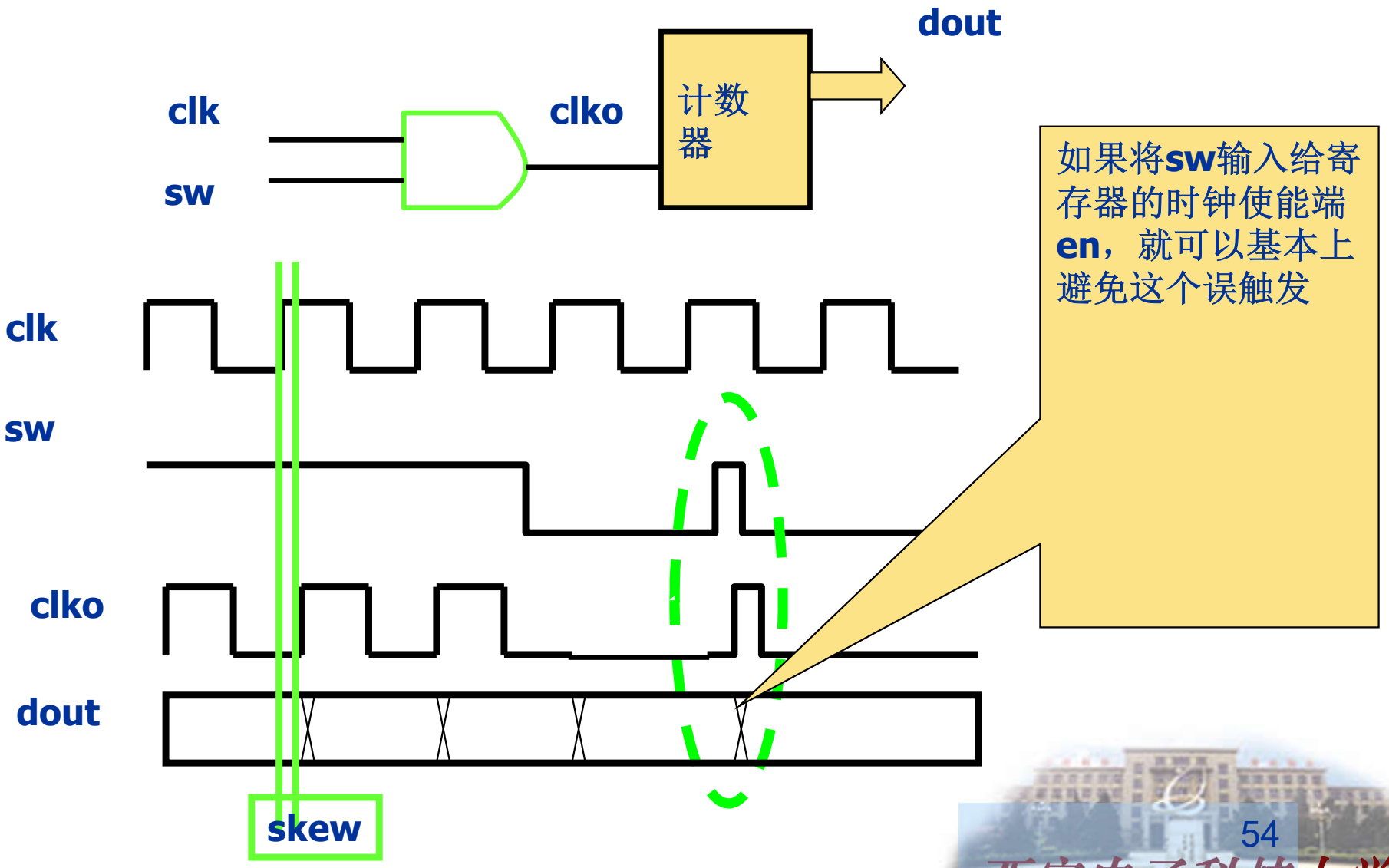
同步系统设计中应该使时钟的skew最小化

- 使用快速的导线类型来对时钟布线(如在fpga中，采用全铜层工艺来实现全局通道)。
- 使用时钟树(Distribution Tree)
- 在前端设计上，应遵循一定的设计原则来避免时钟的过分偏移(接下去一部分内容中讲述)。

PLL的使用

- 目前所使用的中、高端FPGA内部均集成了DPLL，甚至模拟锁相环。
- 这些锁相环提供对时钟的分频、倍频以及移相功能，并且保证skew最小。

门控时钟(gated clk)



Gated clk的危害

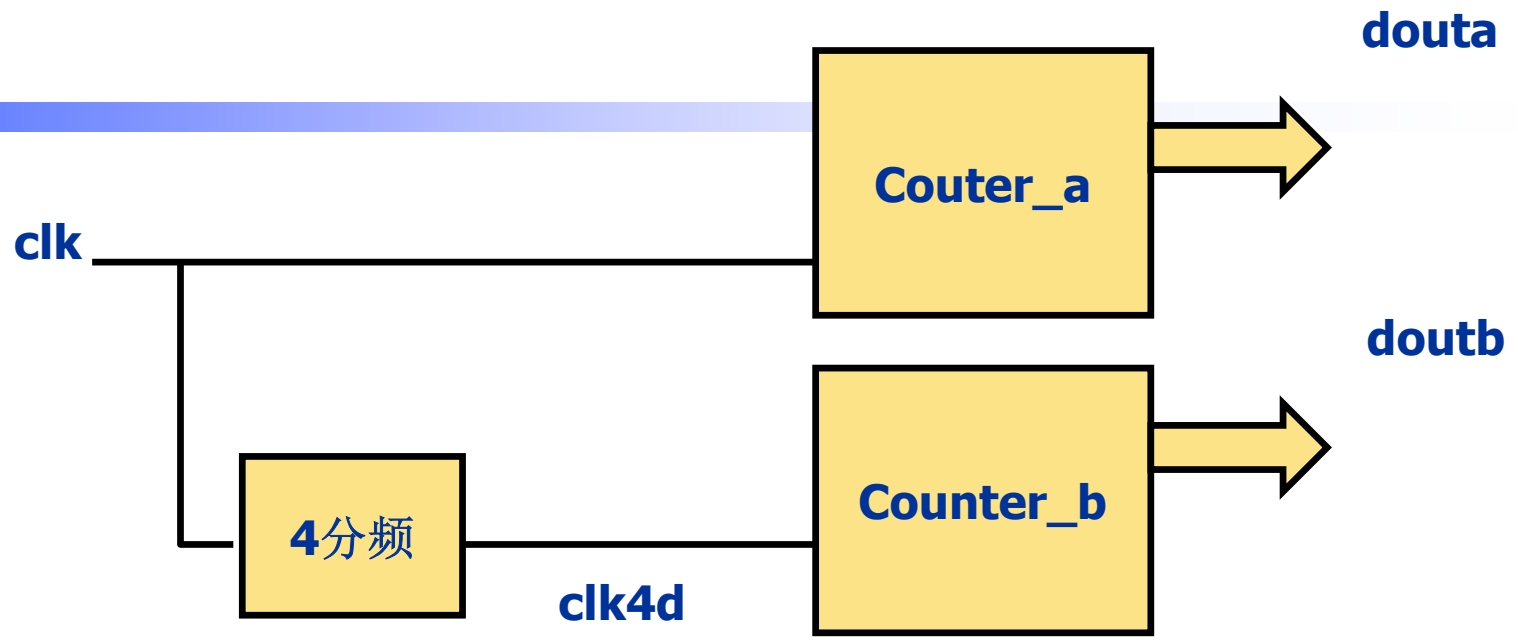
1. 容易导致时序电路的误触发；
2. 增大了clk的skew。
3. 在某些情况下，它可以作为低功耗设计的手段。

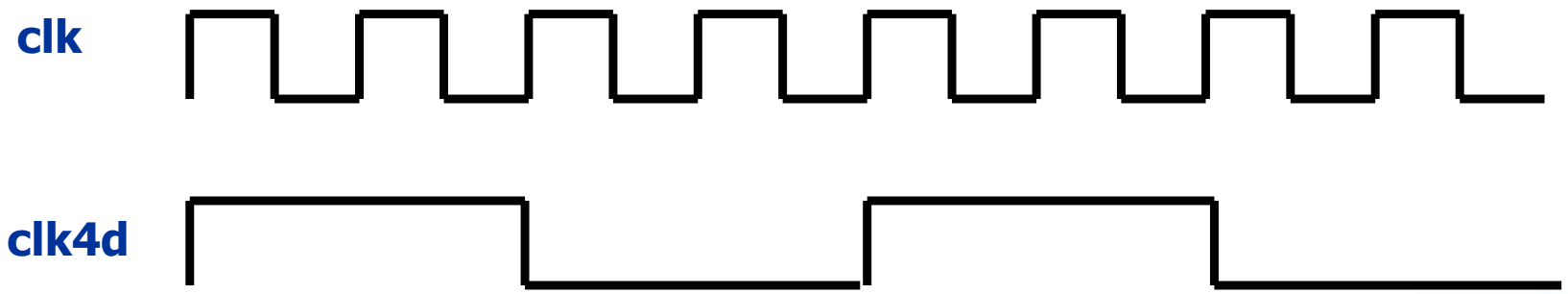
- 因此在设计中，尽量避免时钟通过组合电路，避免使用组合电路来产生时钟。

派生时钟与派生使能

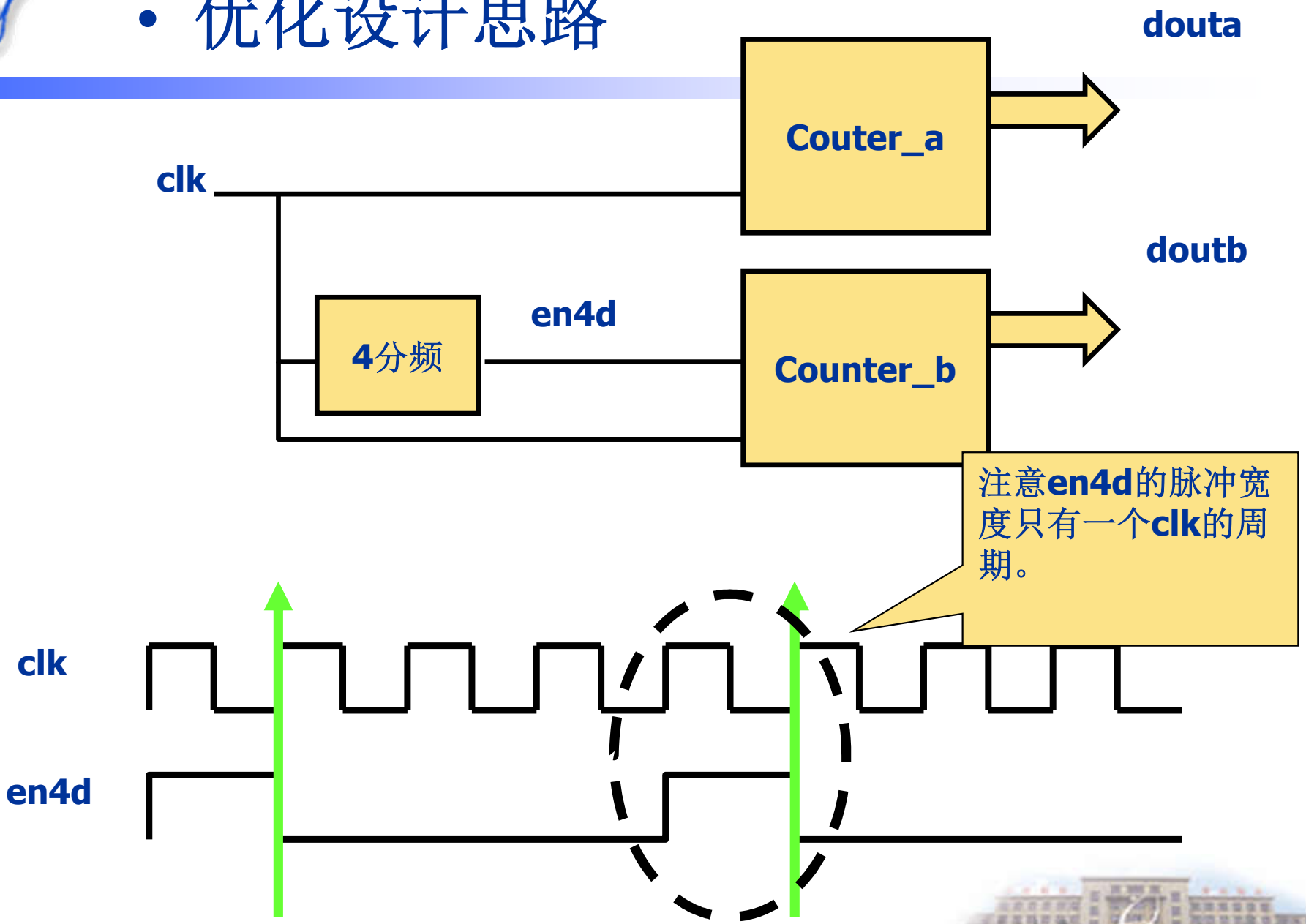
- 例1：设计一个计数器系统，该系统包含两个计数器，其中一个计数器以系统时钟clk频率计数；另一个计数器以clk的1/4频率计数。

  两个计数器同时复位。假设仅要求两个计数器速度满足4倍关系，对相位无任何要求。


douta
Couter_a
clk
doutb
Counter_b
4分频
clk4d


clk
clk4d

• 优化设计思路


douta
Couter_a
clk
doutb
4分频
en4d
Counter_b
注意en4d的脉冲宽度只有一个clk的周期。
clk
en4d

选择派生使能方案的原因

- 派生时钟方案增加了全局通道的耗费，这在全局通道比较稀缺的FPGA中是相当致命的。ASIC实现时导致了时钟树耗费增加。
- 派生使能不但可以消除以上的缺点，而且其时钟skew比派生时钟方案的skew更小。

注意派生使能的产生

- Process(clk, reset)
- Begin
- if(reset = ‘1’) then
  clk_vec <= (others => ‘0’);
- elsif(clk’event and clk = ‘1’) then
  clk_vec <= clk_vec + ‘1’;
  end if;
- End process;

- Process(clk_vec)
- Begin
- if(clk_vec = “00”) then
- en4d <= ‘1’;
- else
- en4d <= ‘0’;
- end if;
- End process;

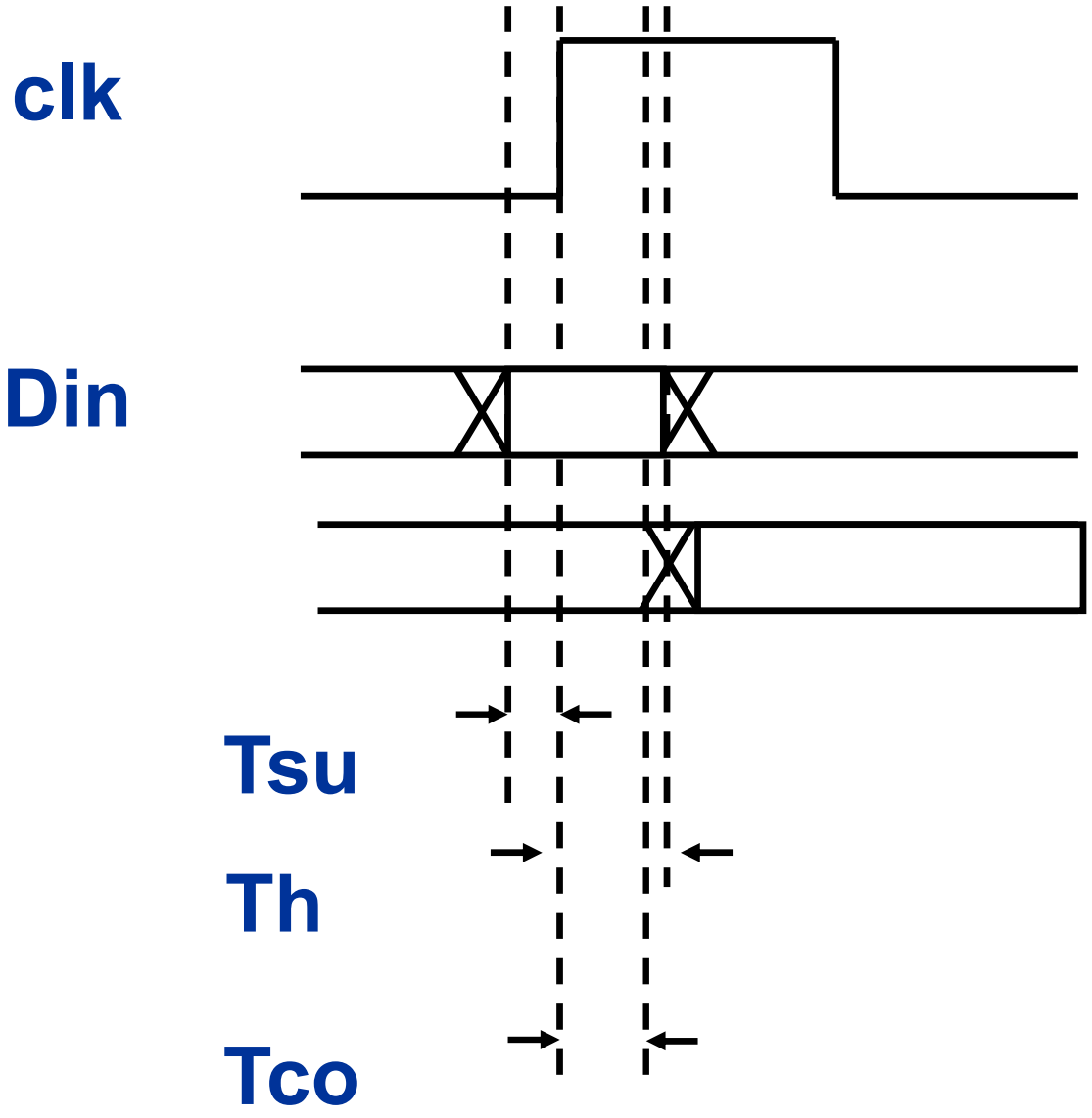
注意，这里是组合电路输出**en4d**，难免产生毛刺；但是因为**en4d**是使能信号，所以这仍然是安全的。

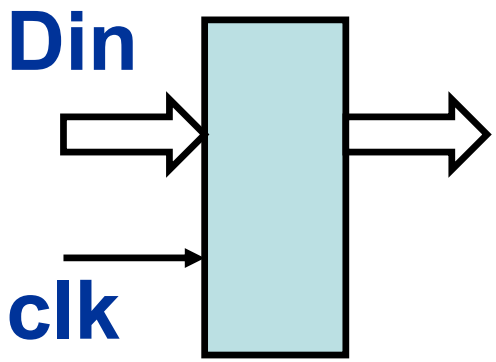
(时钟质量保证)小结

(以下要点均针对前端设计)

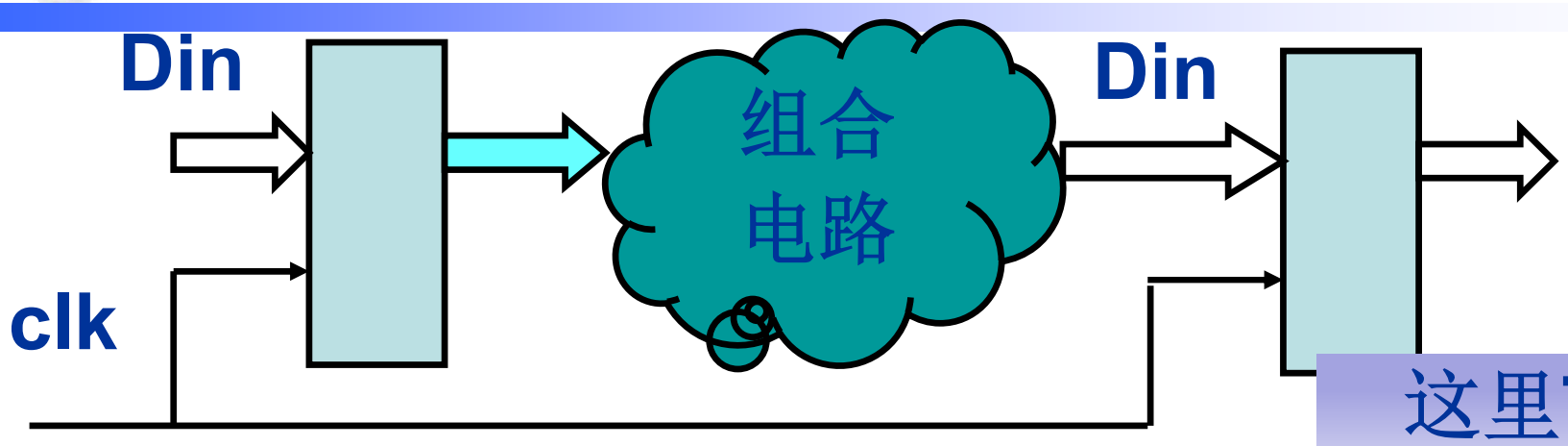
- 时钟的纯净：时钟最好不要通过任何组合电路，或者不要用组合电路产生时钟；
- 时钟的单一：数字系统设计时，应该尽量减少时钟的数目，最好整个系统只有一个时钟；当需要派生时钟时，尽量用派生使能去代替。

时序电路的主要时序参数

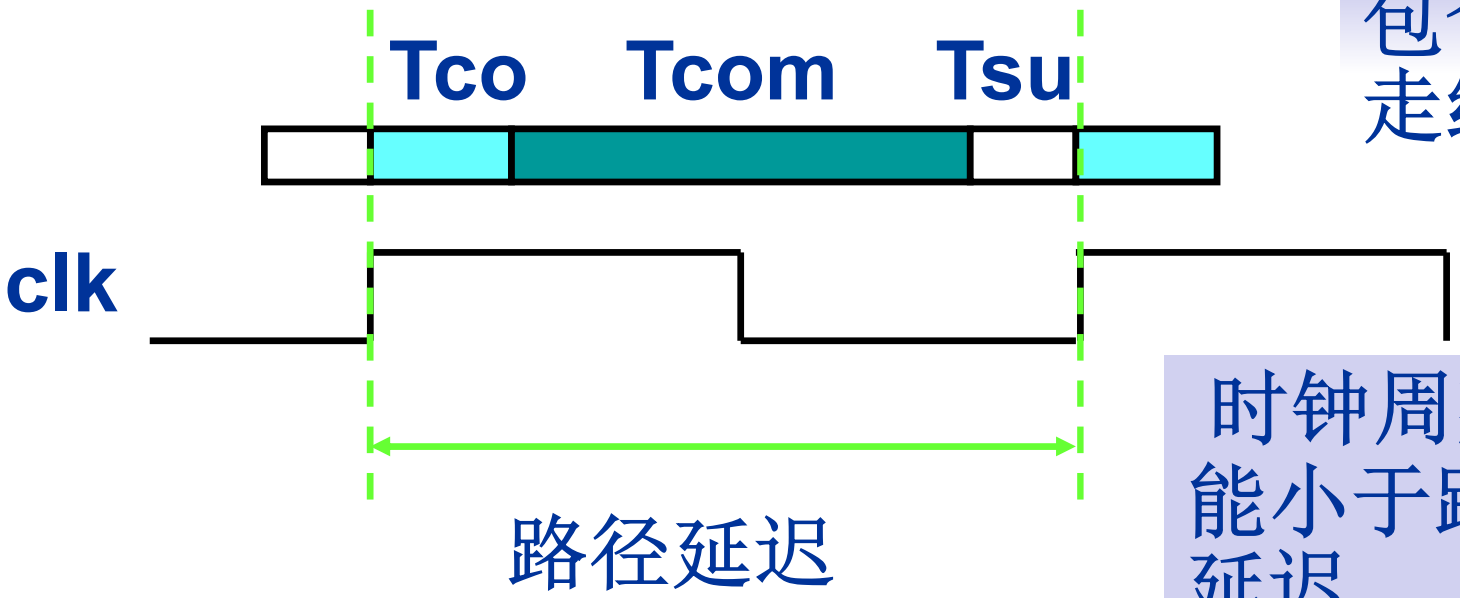




路径延迟



这里Tcom包含了网络走线延迟



时钟周期不能小于路径延迟

路径的定义

- 路径：是一系列标识一条电路的逻辑路线的元素；
- 路径可能包含一个信号网络(net)或一组信号网络(net)，以及相关的元件；
- 当一个元件被包含在一条路径中时，它的输入和输出也包含在这个路径中。
- 路径从一个pad或者一个同步元件(触发器)的输出端开始，一直到遇到一个pad或一个同步元件(触发器)的输入端时终止。

关键路径

- 一个同步系统中的关键路径，就是它所有的路径中，路径延迟最长的那一条。
- 显然，这个同步系统的最高工作频率，等于关键路径延迟的倒数。

延时优化的几个要点

1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

长路径的避免

- 实际上是一个很泛的技巧，总之，在设计中，时序能走短路径就尽量走短路径；组合电路能缩小就尽量缩小。以下仅举两例来说明。

延时优化

- 状态机设计中，状态编码采用**Binary**编码和**One-hot**编码对系统会造成什么样的性能影响？这两种编码对**FPGA/CPLD**的适用情况如何？

	Binary	One-hot
S0	00	0001
S1	01	0010
S2	10	0100
S3	11	1000

延时优化

- 解答：one-hot编码方式只用一个bit来表示一个状态，这大大缩小了状态译码的组合电路规模，使得路径延时更小，因此状态机的时钟可以运行在更高的频率上。
- 特例：不妨想象该状态机就是一个循环计数器，如果采用binary编码，则该计数器存在明显的组合电路；而如果采用one-hot编码，该计数器的综合结果就是一个移位寄存器序列，根本不存在任何组合门！

扇入系数与组合规模

- 这个例子实际上也说明了一个问题：组合电路的规模往往受影响于其扇入系数：
- 扇入↑ → 组合电路规模↑ → 从而路径延时↑ → 系统工作速率↓
- 减少组合逻辑的扇入是提高系统工作速率的基本手段。

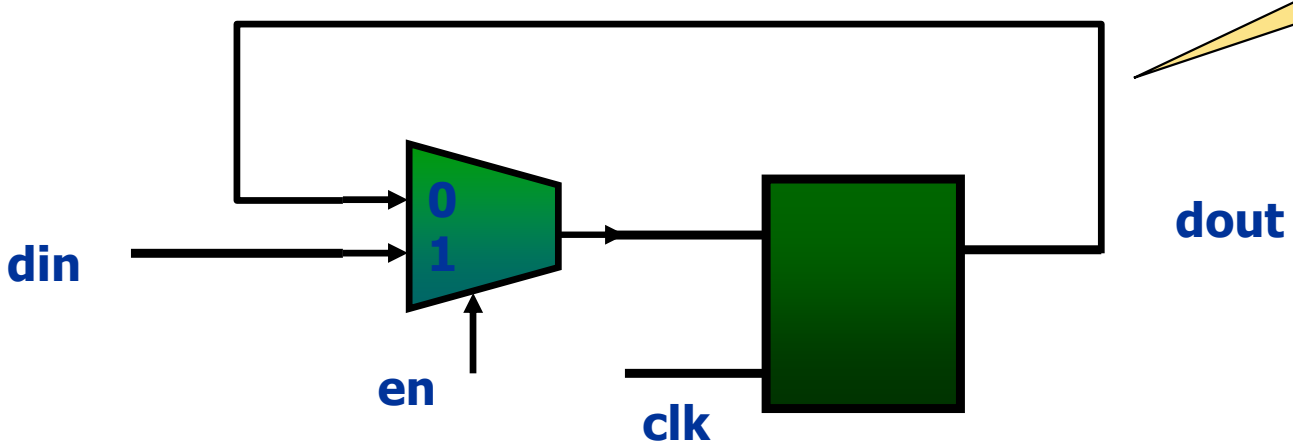
One-hot资源消耗对比

- One-hot因为寄存器消耗量比较大，所以往往用在寄存器资源比较丰富的FPGA中，CPLD中使用得比较少。

反馈多路选择与专用时钟使能

- 时钟使能的两种实现方式：
  - 寄存器内部的专用时钟使能电路；
  - 反馈多路选择，如下图：

因为增加了额外的路径，因此降低了寄存器的最高工作速率。



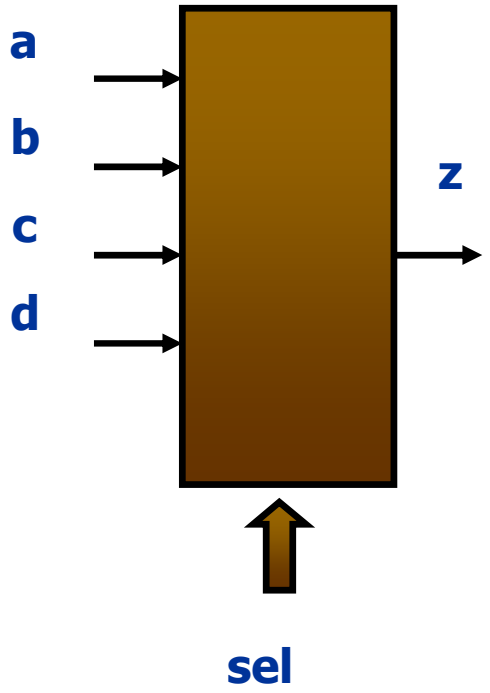
优化时序的设计调整

- 设计时，往往需要针对具体的情况来进行设计的调整，以使系统的时序得到优化。
- 时序优化的原则，最主要的还是在保证正确逻辑的基础上，尽量缩小关键路径的延迟。

例：带优先级的多路选择器

- 假设该选择器的真值表如下：

Sel(0)	Sel(1)	Sel(2)	Sel(3)	z
x	x	x	1	d
x	x	1	0	c
x	1	0	0	b
1	0	0	0	a
0	0	0	0	0



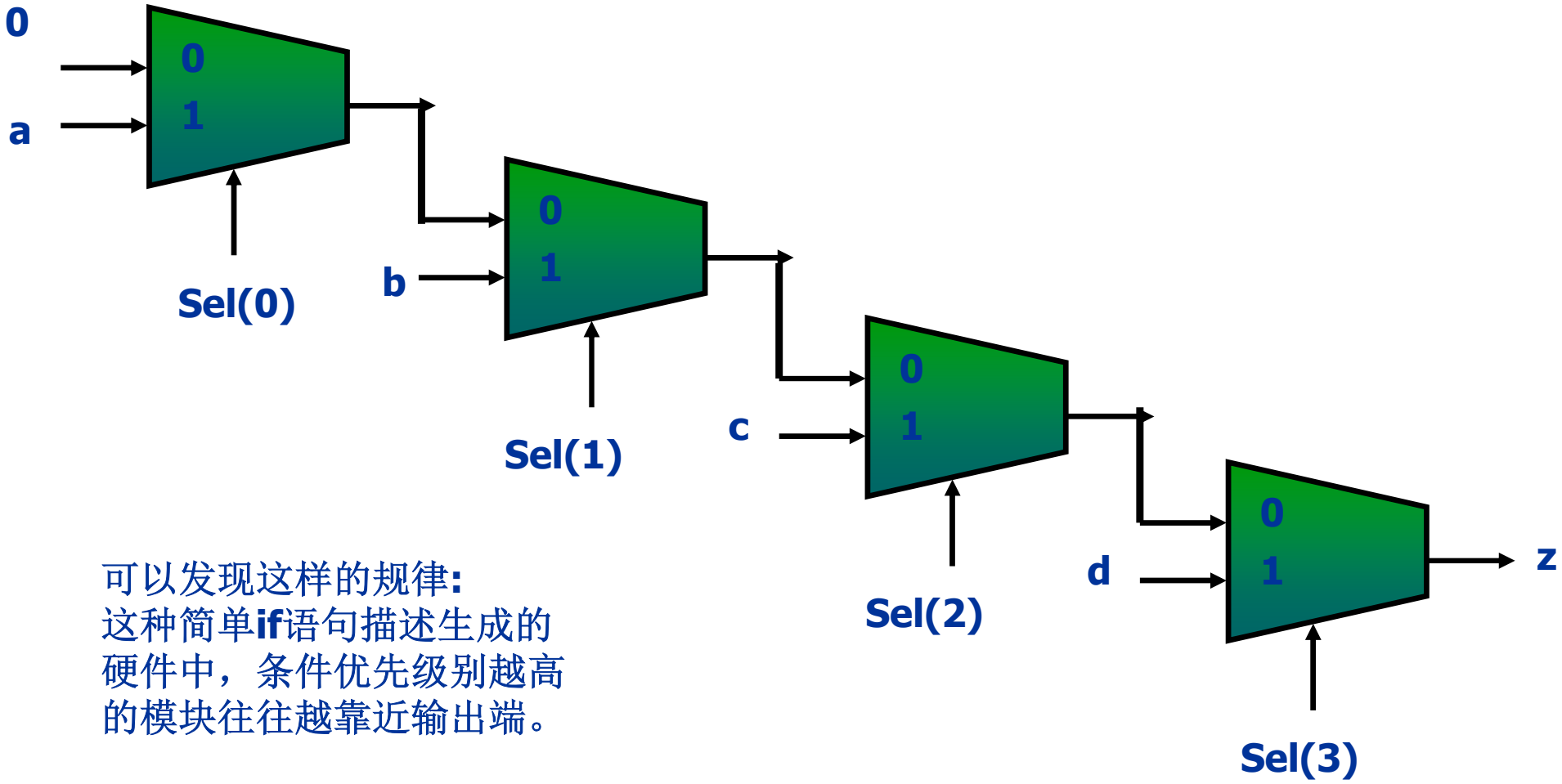
正常的描述方法1：多if语句

- Process(a, b, c, d, sel)
- Begin
- z <= ‘0’; -- 要记得初始化!不推荐此风格。
- if(sel(0) = ‘1’) then z <= a; end if;
- if(sel(1) = ‘1’) then z <= b; end if;
- if(sel(2) = ‘1’) then z <= c; end if;
- if(sel(3) = ‘1’) then z <= d; end if;
- End process;

正常的描述方法2：单if语句

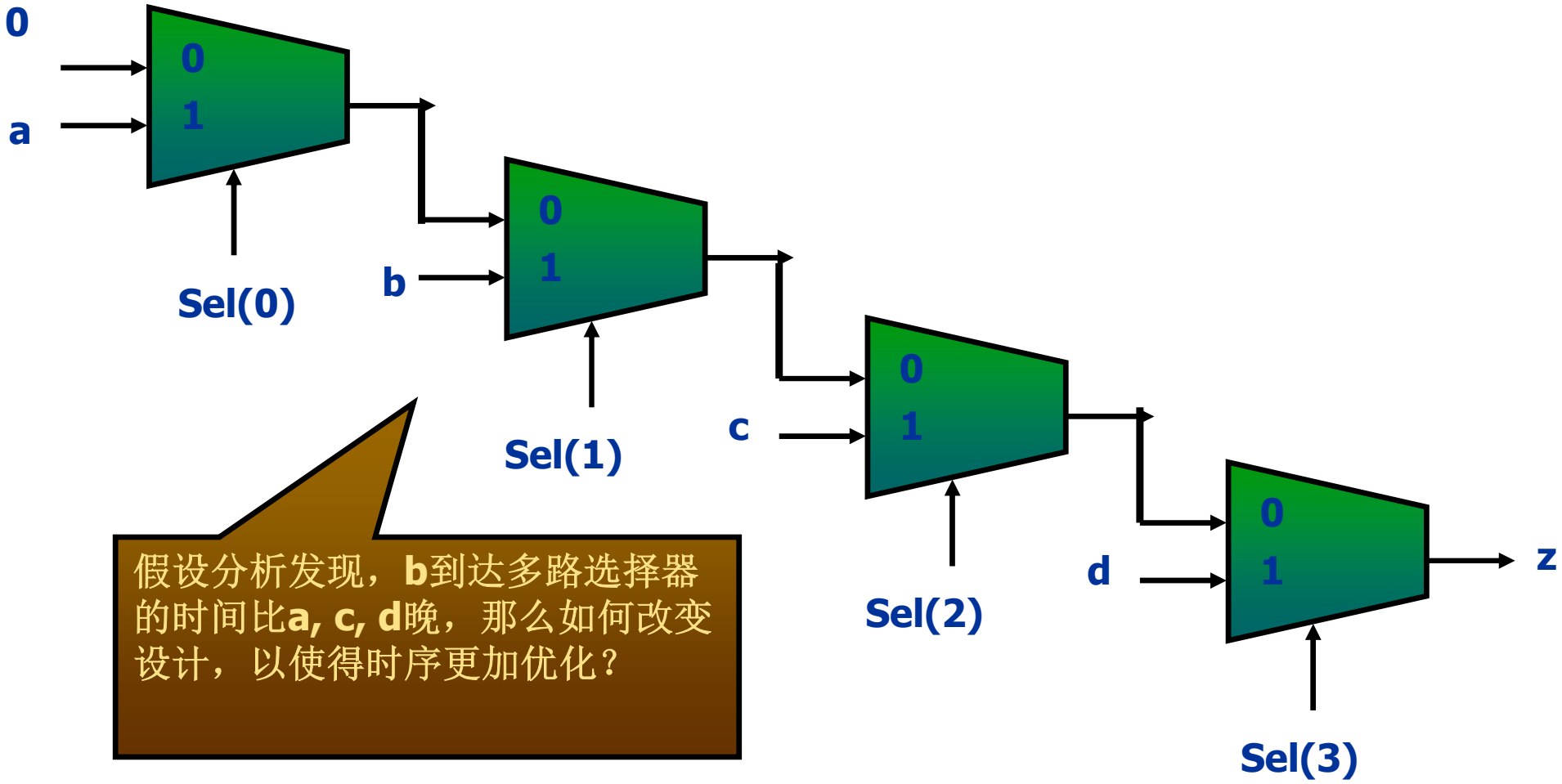
```
Process(a, b, c, d, sel)
Begin
    if    ( sel(3) = '1' ) then  z <= d;
    elsif( sel(2) = '1' ) then  z <= c;
    elsif( sel(1) = '1' ) then  z <= b;
    elsif( sel(0) = '1' ) then  z <= a;
    else                        z <= '0';
    end if;
End process;
```

综合后的电路图

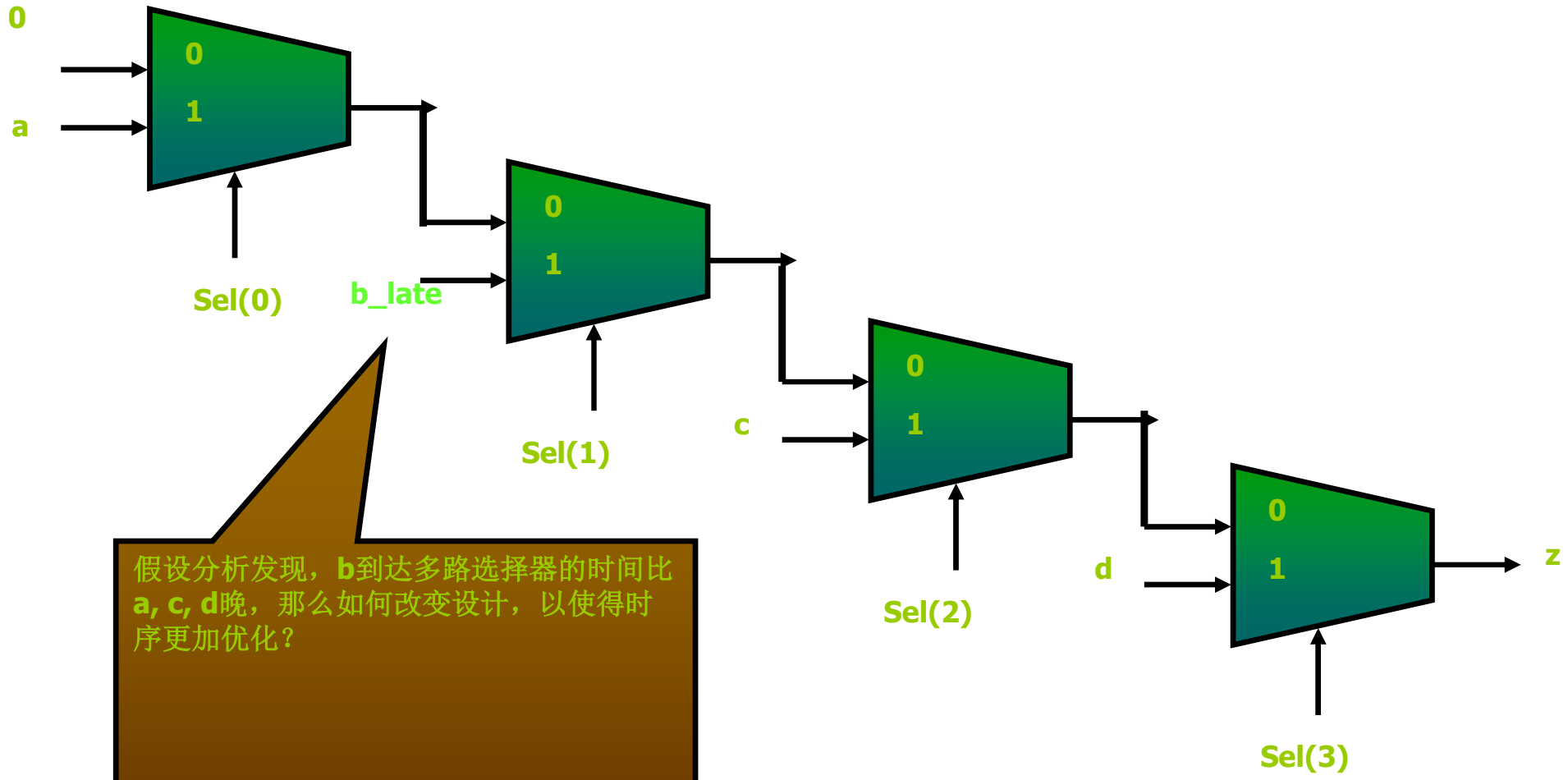


可以发现这样的规律:
这种简单**if**语句描述生成的硬件中，条件优先级别越高的模块往往越靠近输出端。

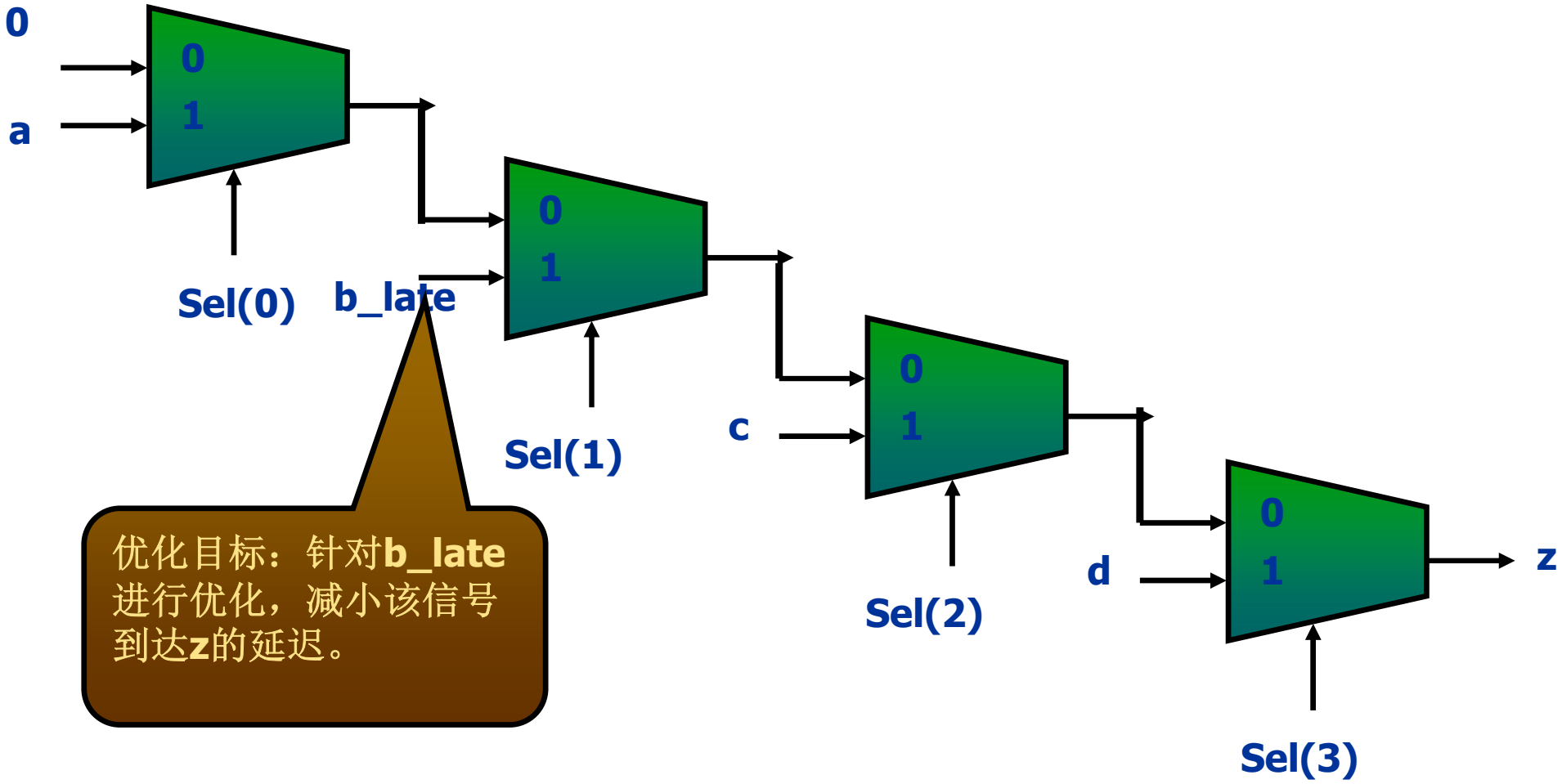
调整情况1:数据到达延迟
(b ➔ b_late)



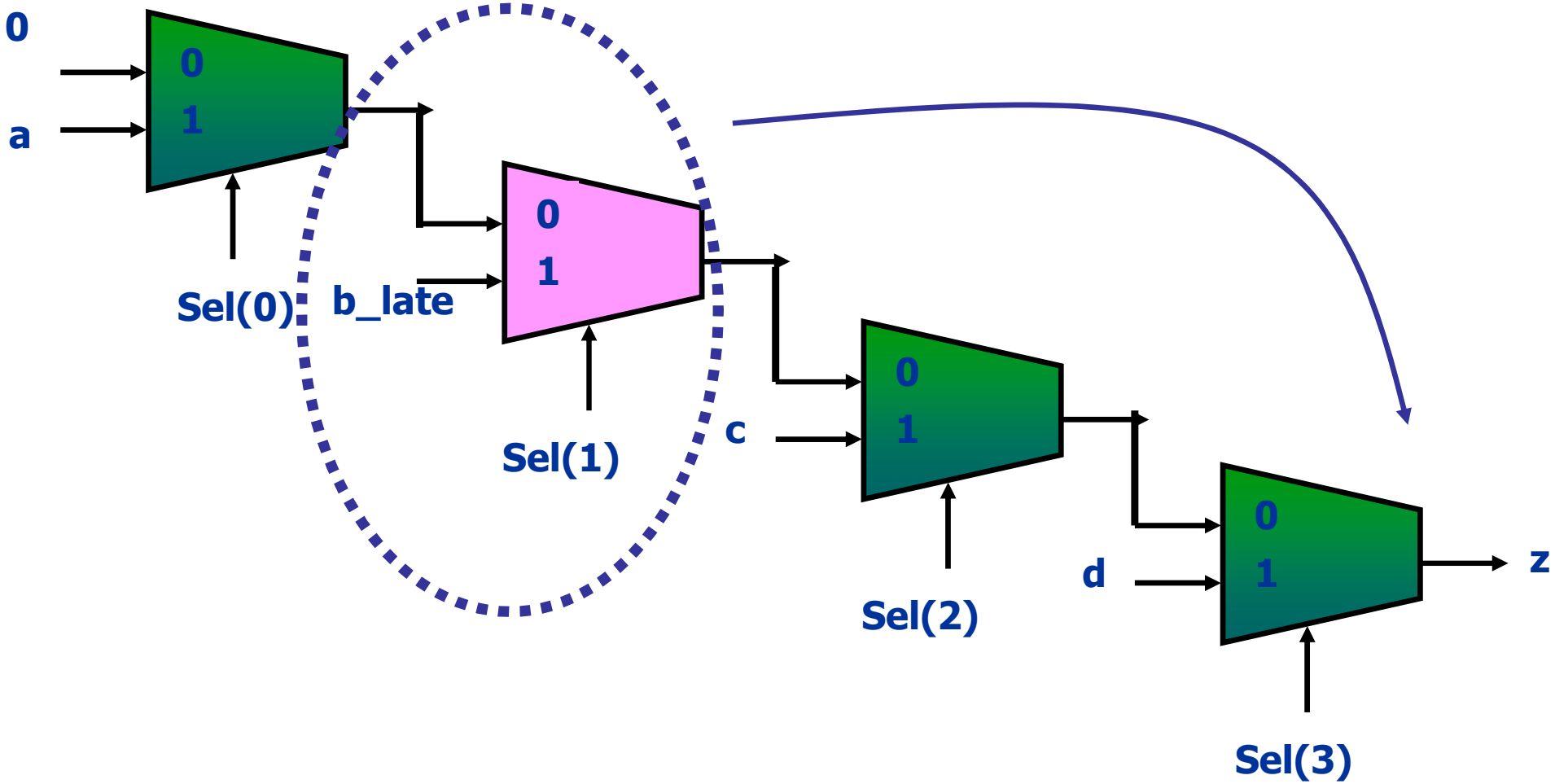
调整情况1:数据到达延迟
(b ➔ b_late)



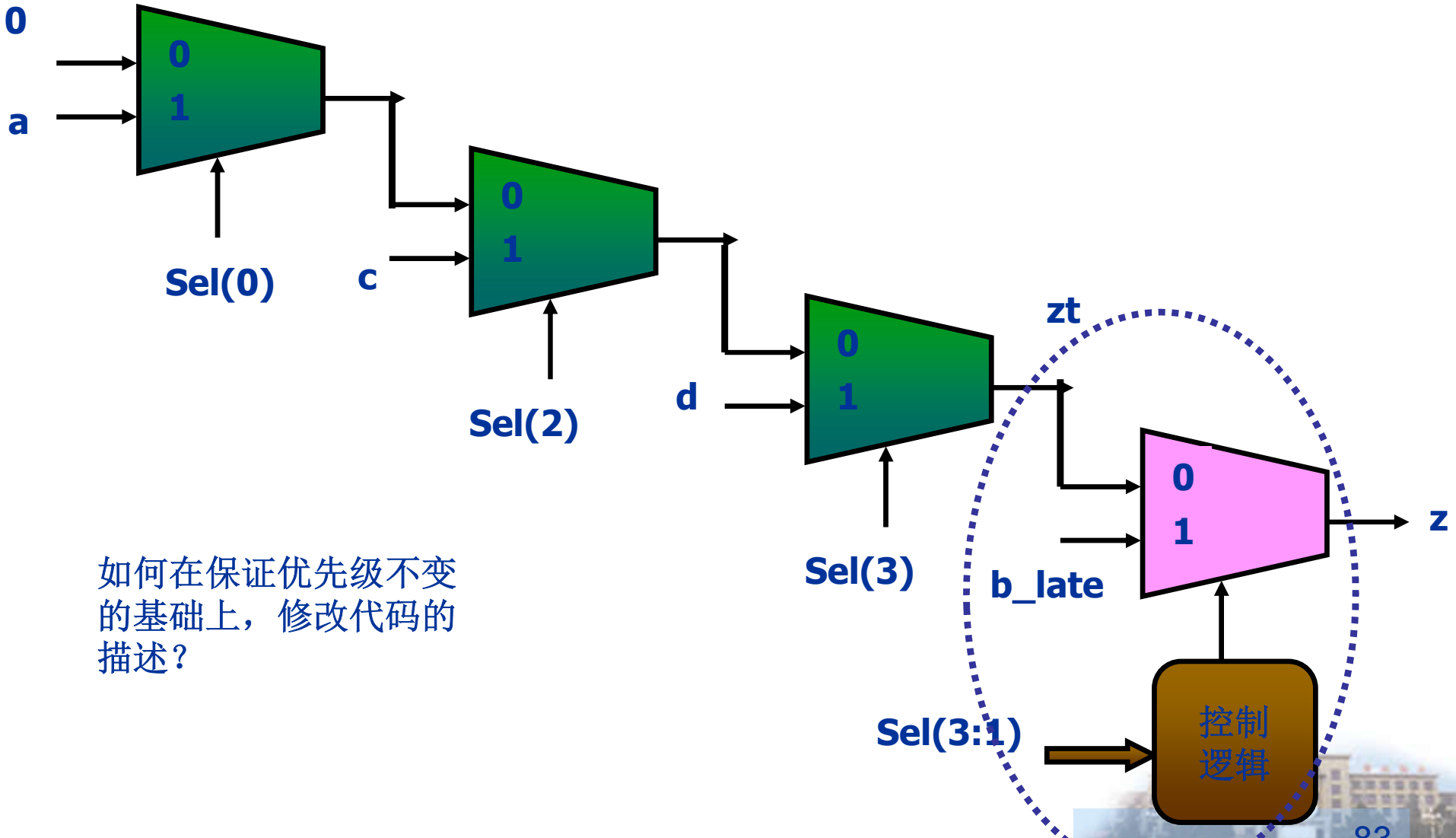
优化目标


0
a
0
1
Sel(0)
b_late
0
1
Sel(1)
c
0
1
Sel(2)
d
0
1
Sel(3)
z
优化目标：针对b_late进行优化，减小该信号到达z的延迟。

优化思路



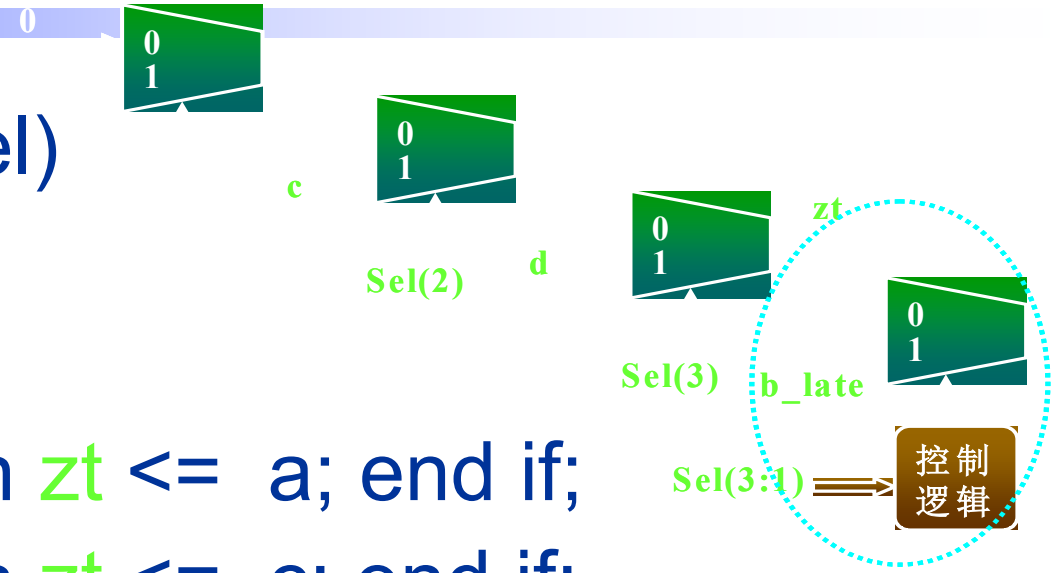
优化后的电路图



如何在保证优先级不变的基础上，修改代码的描述？

优化描述方法1：多if语句

```
Process( a, b, c, d, sel)
Begin
    zt <= '0';
    if( sel(0) = '1' ) then zt <=  a; end if;
    if( sel(2) = '1' ) then zt <=  c; end if;
    if( sel(3) = '1' ) then zt <=  d; end if;
    if( ( sel(1) = '1' ) and ( sel(2) = '0' ) and ( sel(3) =
   '0')   )                then  z <= b_late;
    else                          z <= zt;
    end if;
End process;
```



优化描述方法2：双if语句

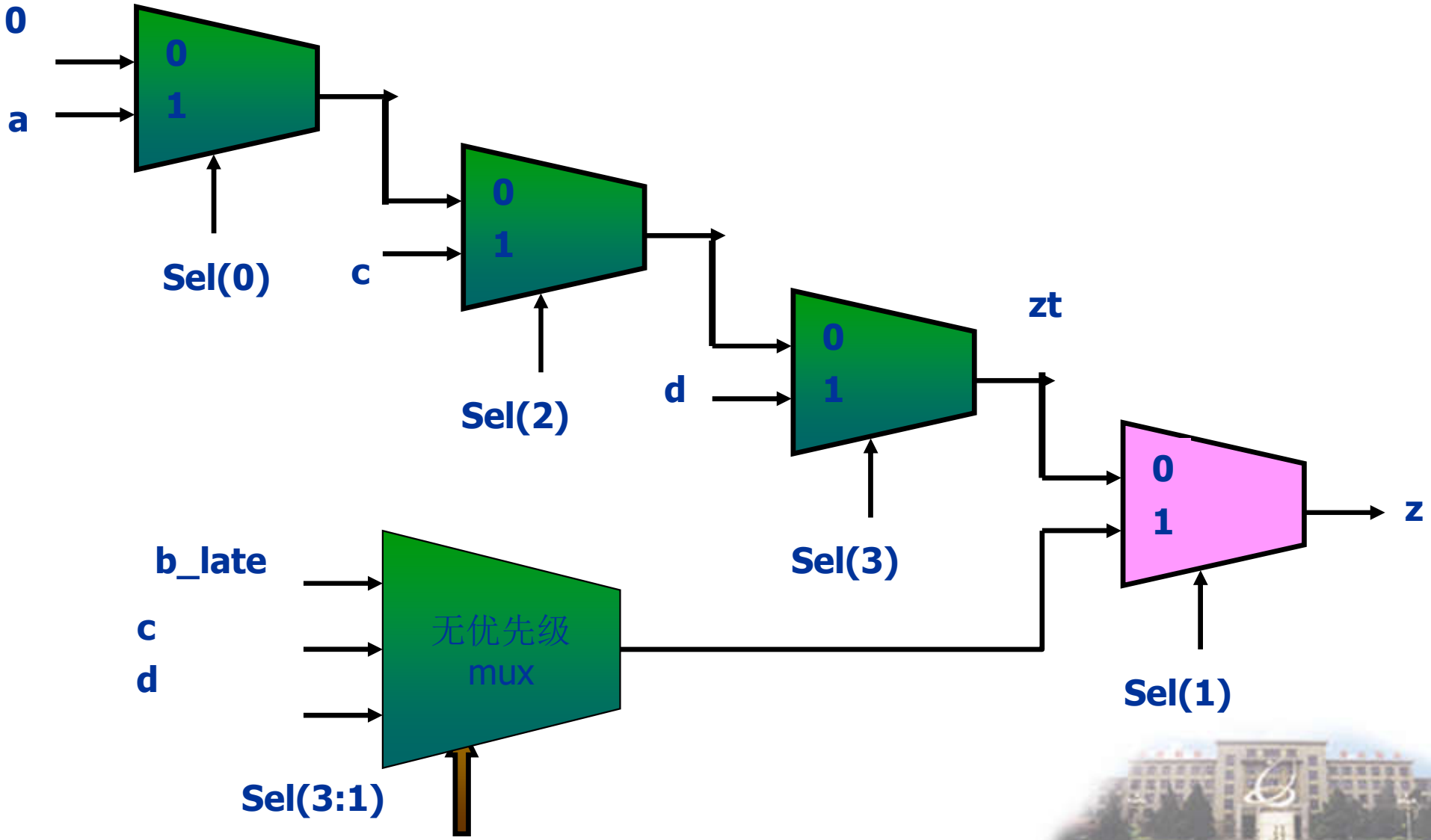
```
Process( a, b, c, d, sel)
Begin
    if    ( sel(3) = '1' ) then zt <= d;
    elsif( sel(2) = '1' ) then zt <= c;
    elsif( sel(0) = '1' ) then zt <= a;
    else                                zt <= 0;
    end if;
    if     ( ( sel(1) = '1' ) and ( sel(2) = '0' ) and (
   sel(3) = '0' ) then                        z <= b_late;
    else                                  z <= zt;
    end if;
End process;
```

!优化描述方法3：

- Process(a, b, c, d, sel)
- Begin
- if (sel(1) = ‘1’)
- case ….. End case;
- elsif(sel(3) = ‘1’) then z <= d;
- elsif(sel(2) = ‘1’) then z <= c;
- elsif(sel(0) = ‘1’) then z <= a;
- else z <= 0;
- end if;
- End process;

```
Case sel( 3 downto 2 ) is
   when “00” => z <= b;
   when “01” => z <= c;
   when “10” => z <= d;
   when others => z <=d;
End case;
```

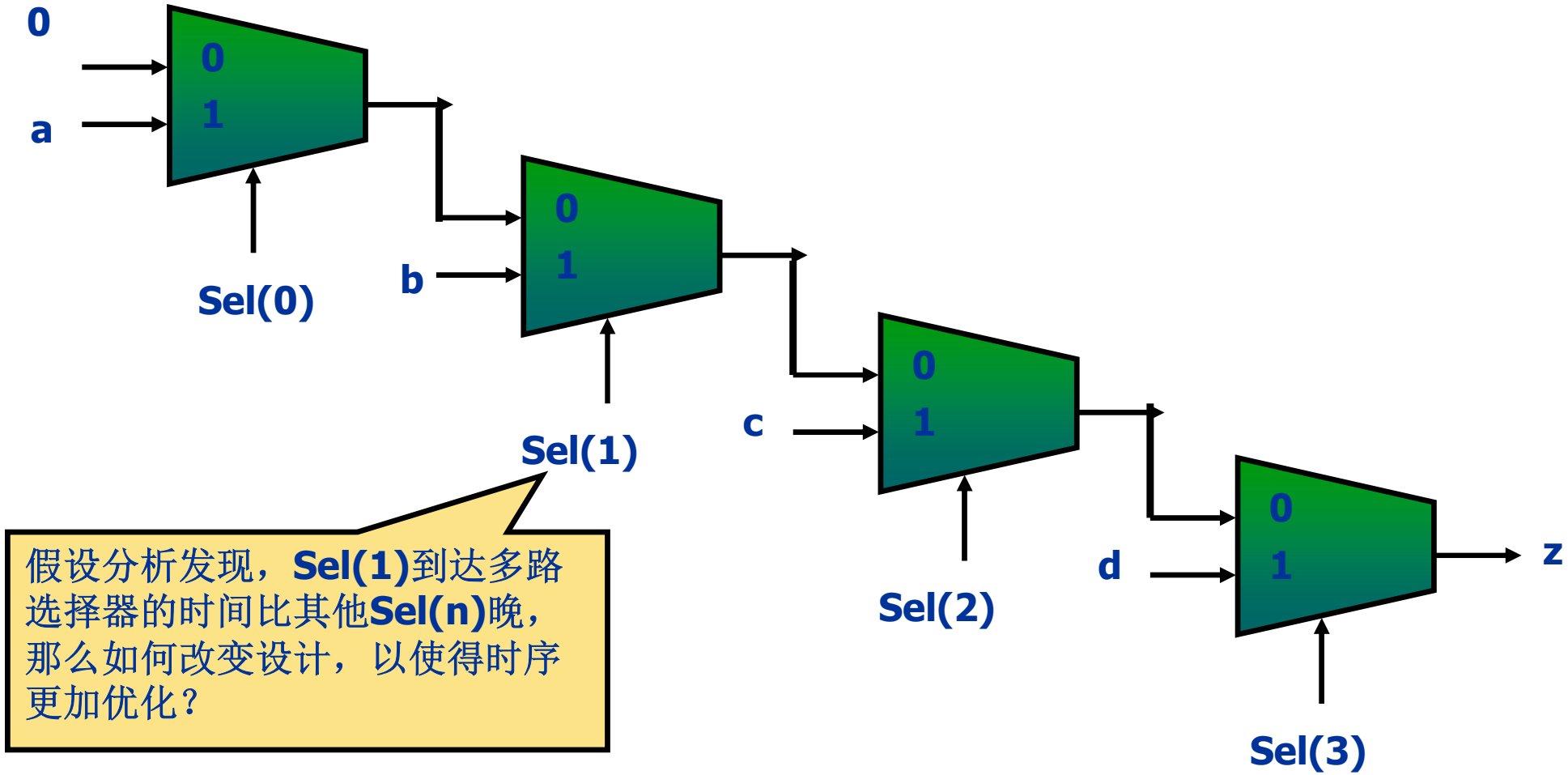
方法3综合后的电路图


0
a
0
1
Sel(0)
c
0
1
Sel(2)
d
0
1
Sel(3)
zt
0
1
z
Sel(1)
b_late
c
d
无优先级
mux
Sel(3:1)

方法分析

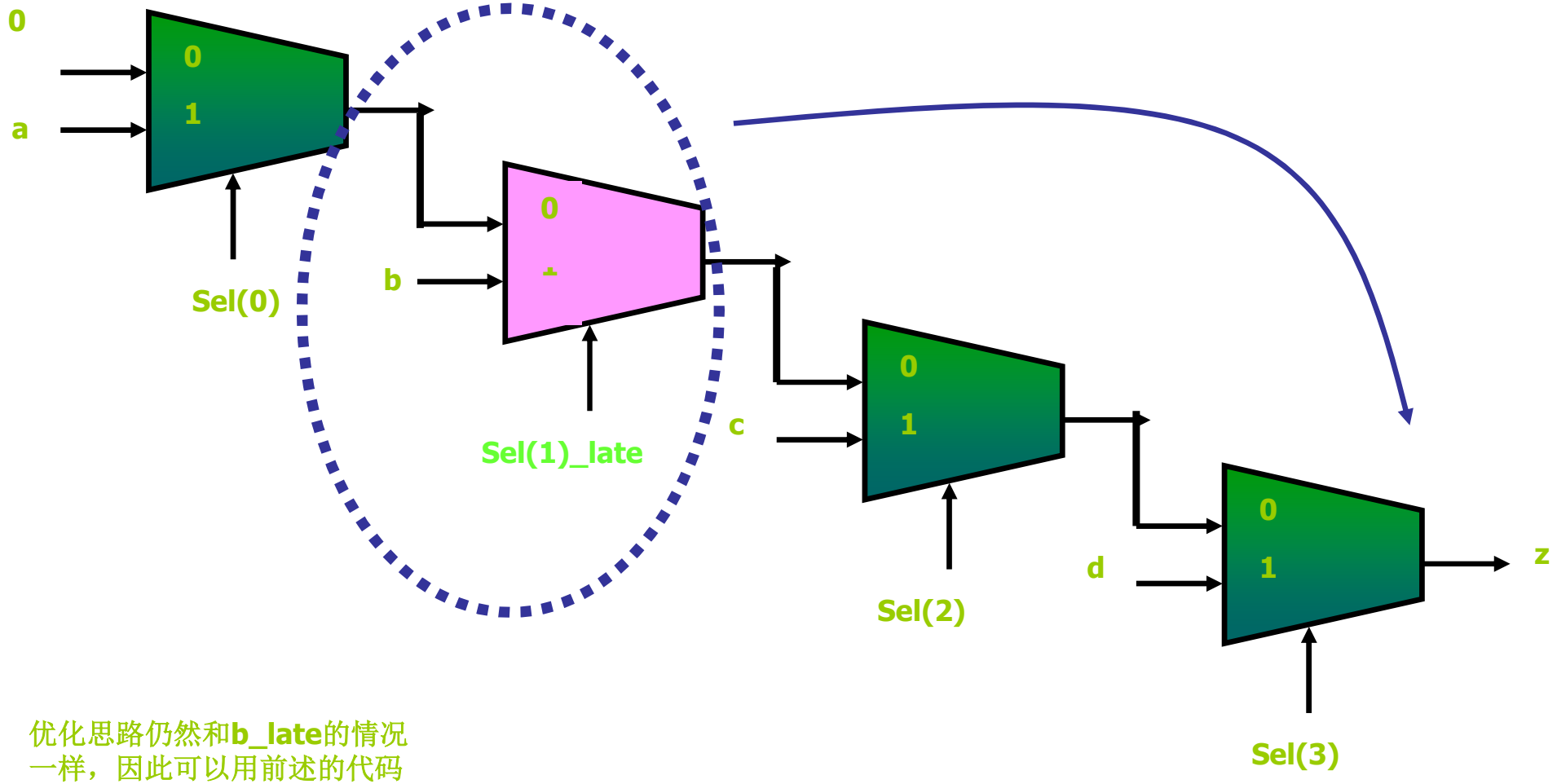
- 方法1和方法2实际上是等效的，都可以生成前述的优化电路；而方法3的结果并不是优化的，b_late到输出z的延时仍然很大。
- 所以要注意，并不是将b_late对应的if条件写在最优先级别处就一定可以获得最小的延时，方法3中b_late被读的条件处理不当会导致b_late进入复杂的硬件从而又增大延迟。
- 代码设计时，必须弄清楚所用的综合工具对特定描述风格的综合结果。
- 一般来说，代码描述的风格不要太抽象化，应该遵循硬件一一对应的模块化原则，这样有利于综合器生成与人脑直观相符合的硬件电路。特别在优化描述时更是应该如此。

调整情况2：控制到达延迟
(Sel(1) -> Sel(1)_late)



假设分析发现，**Sel(1)**到达多路选择器的时间比其他**Sel(n)**晚，那么如何改变设计，以使得时序更加优化？

优化思路



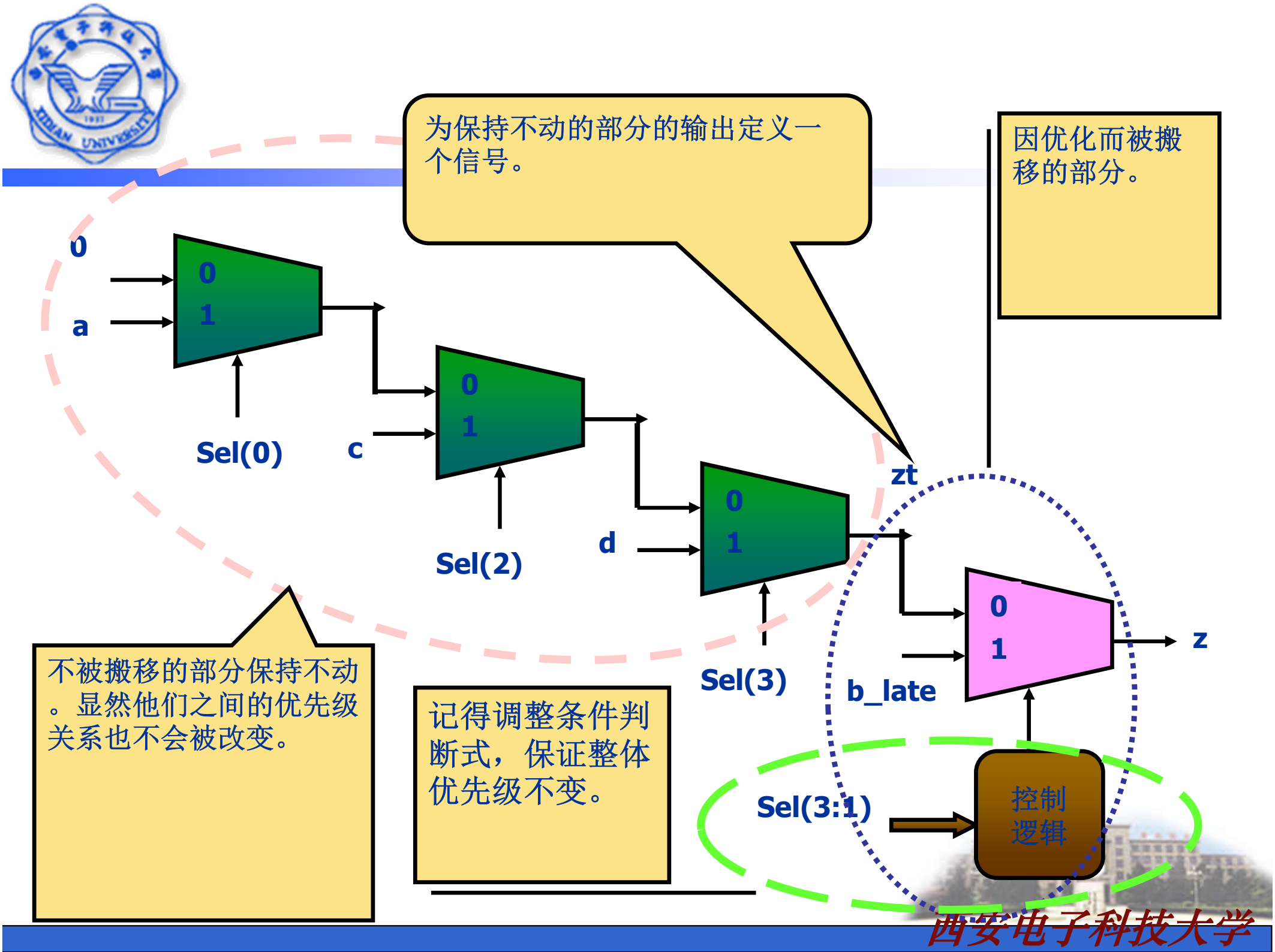
优化思路仍然和**b_late**的情况一样，因此可以用前述的代码进行优化。

带优先级的电路优化方法总结

- 对于带有优先级别的电路，在进行延时优化时，要兼顾好延时和优先级。
- 对于单纯的if语句描述(single if statement OR multiple if statement)，一般生成的硬件都是越高优先级条件判断对应的模块越靠近输出端。但这不意味着对应的单个控制信号(如Sel(1))的优先级越高。

带优先级的电路优化方法总结(续)

- 如果到达比较晚(即延时较大)的信号(不管是data信号还是control信号)所对应的硬件模块，并不是处于最高优先级处时，那么就应该想办法将该模块向靠近输出端的位置搬移，以减小该信号所在的路径(很可能成为关键路径)的延迟。
- 可以通过搬移if条件模块的方法搬移需要移动的模块；
- 搬移之后，要调整if条件判断式，保证原来的优先级不变。


为保持不动的部分的输出定义一个信号。
因优化而被搬移的部分。
0
a
0
1
Sel(0)
c
0
1
Sel(2)
d
0
1
Sel(3)
zt
0
1
z
b_late
不被搬移的部分保持不动。显然他们之间的优先级关系也不会被改变。
记得调整条件判断式，保证整体优先级不变。
Sel(3:1)
控制逻辑
西安电子科技大学

代码结构

- Process(…)
- Begin
- -- 固定不动部分的代码描述(对zt赋值);
- -- 被搬移部分的代码描述(对z赋值);
- End process;

延时优化的几个要点

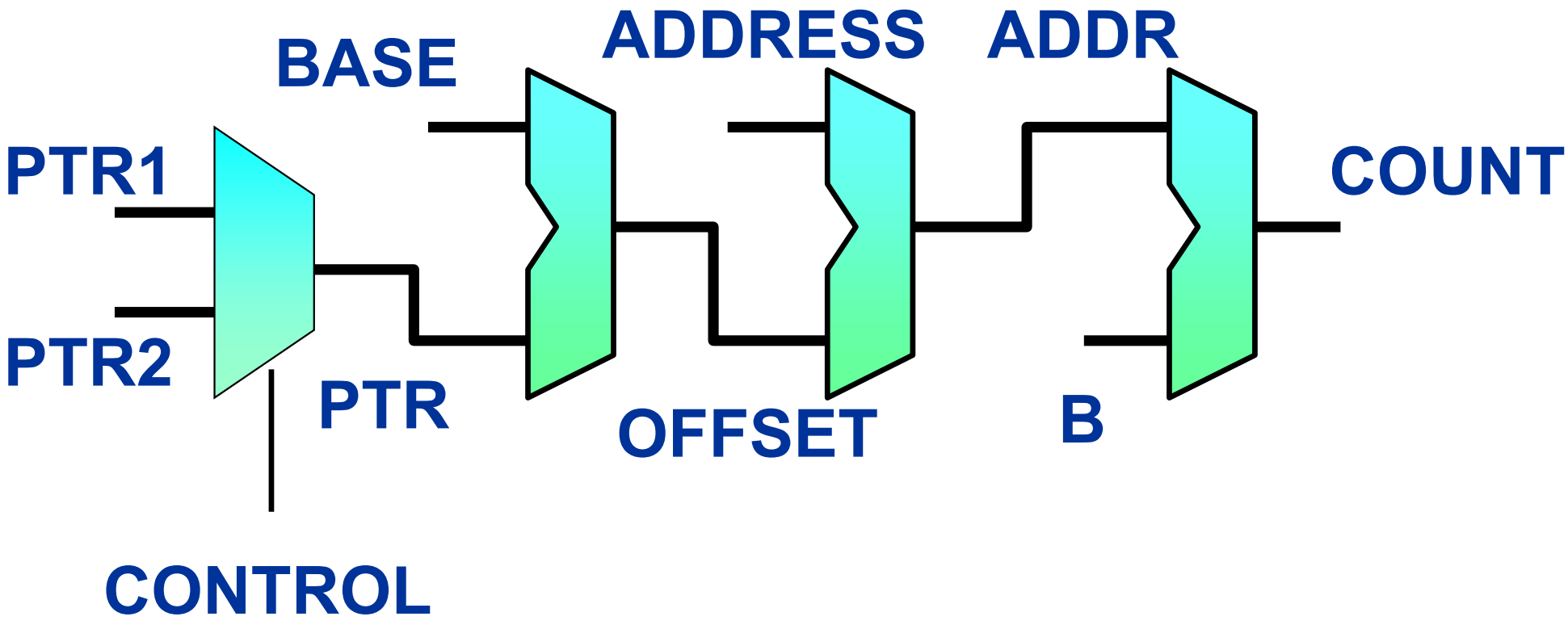
1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

例：数据通路复用

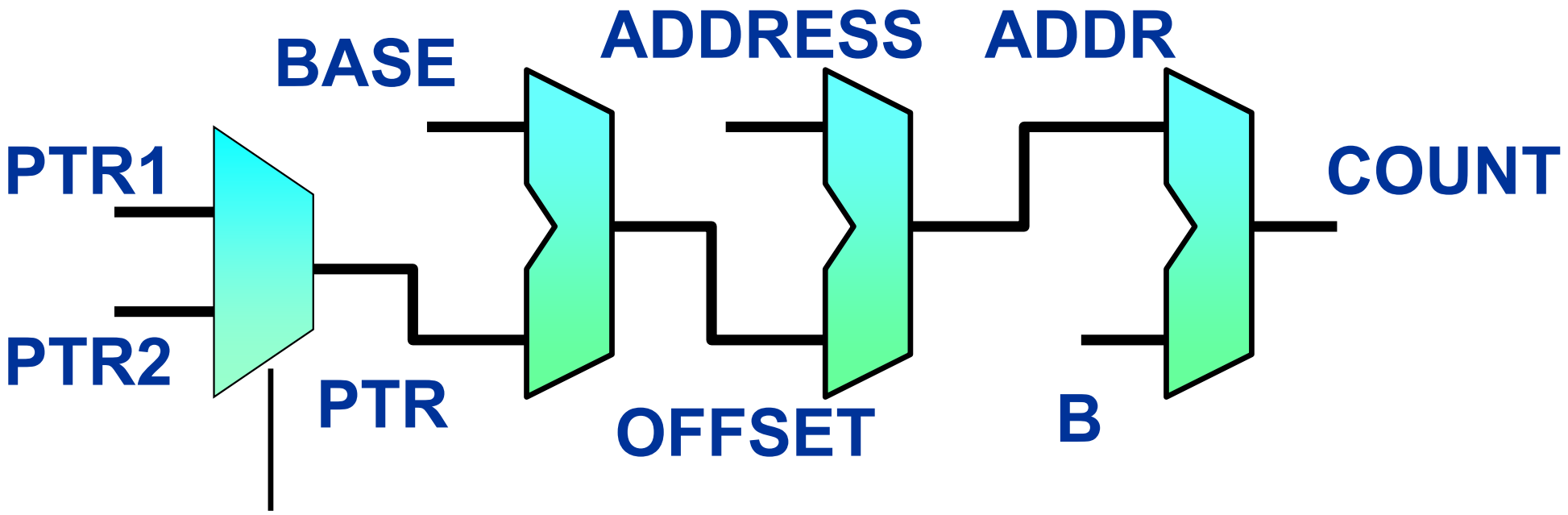
考虑下面的代码：

```
If( CONTROL = ’1’ )then
    PTR := PTR1;
else
    PTR := PTR2;
end if;
OFFSET := BASE - PTR;
ADDR := ADDRESS - OFFSET;
COUNT <= ADDR + B;
```

综合后的电路图

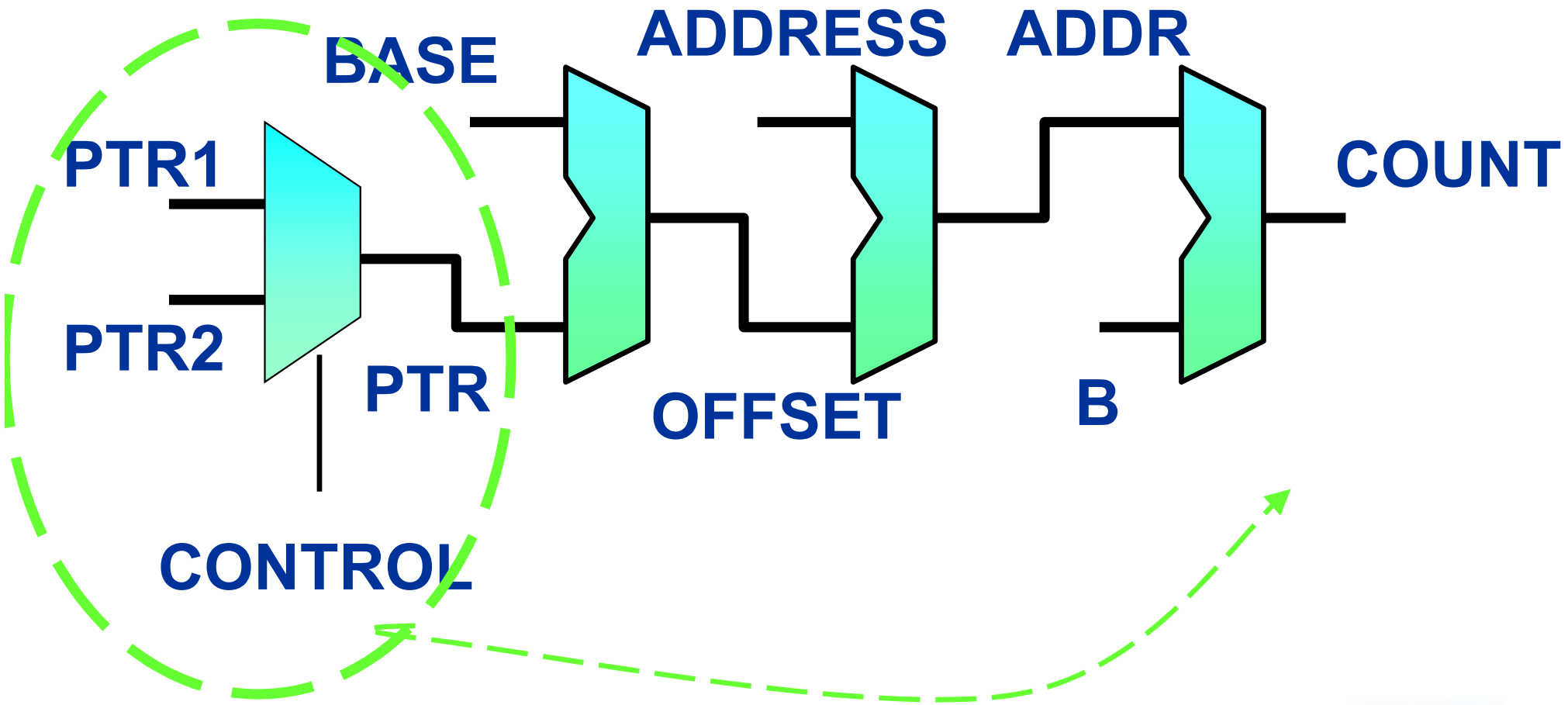


优化目标

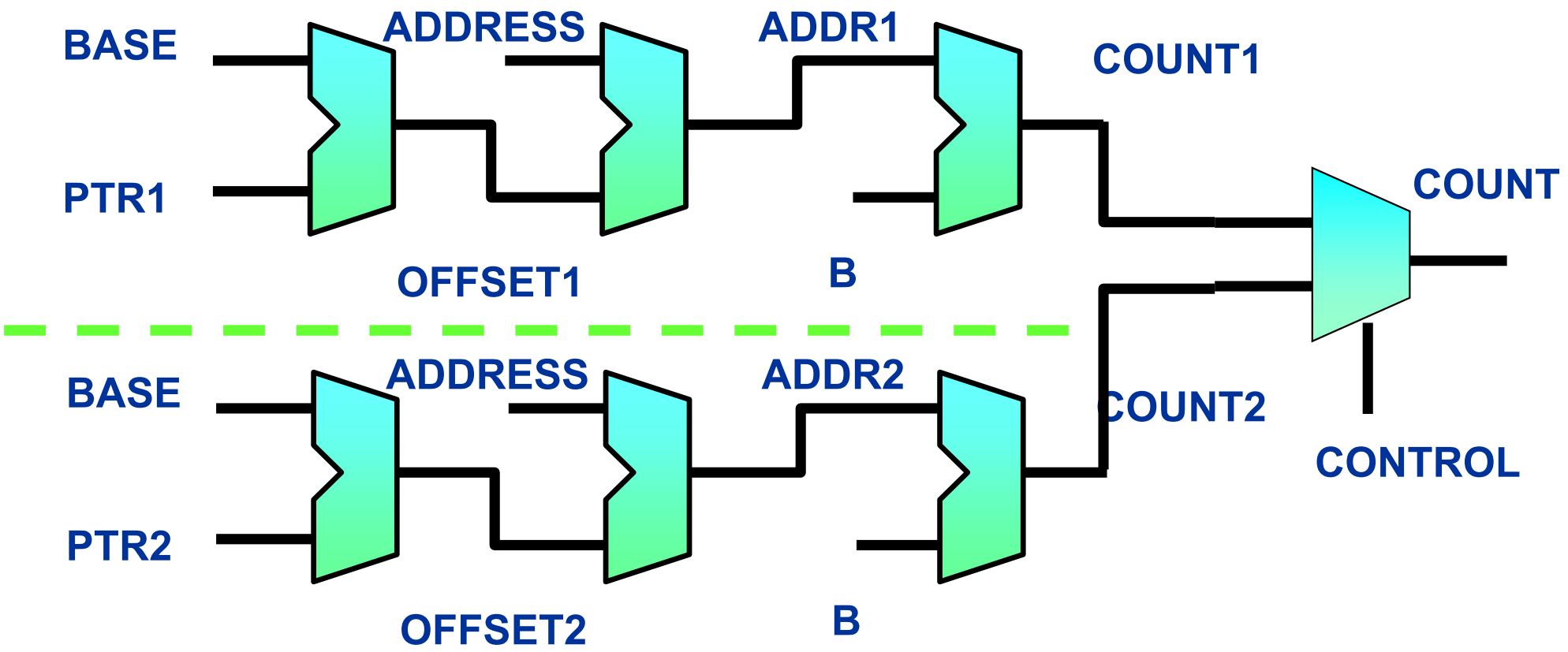


假设**CONTROL**是一个**late arriving input signal**。现在要对该信号进行优化，减小该信号到达**COUNT**的延时。

优化思路



优化后的电路图


BASE
ADDRESS
ADDR1
COUNT1
PTR1
COUNT
OFFSET1
B
BASE
ADDRESS
ADDR2
COUNT2
CONTROL
PTR2
OFFSET2
B

Data-Path Duplication方法代码实现

- 在此略去，请自行完成该代码的编写。

性能对比

可见，逻辑复用方法在提高时序性能的同时，引起资源耗费增大。

	Data Arrival Time	Area
传统方法：	5.23	1057
数据通路复用方法：	2.33	1622

注：这里的**Data Arrival Time**指的是**CONTROL**到达输出端**COUNT**的延时。

延时优化的几个要点

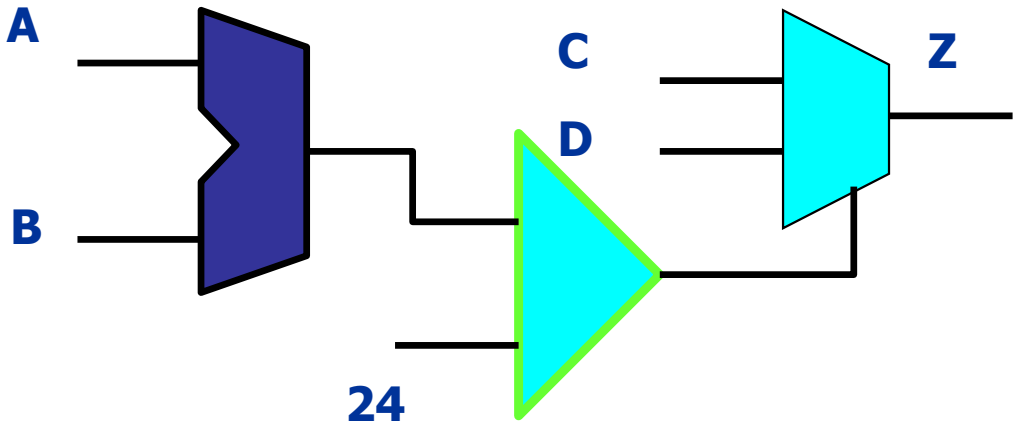
1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

例：if条件判断式中的数据运算

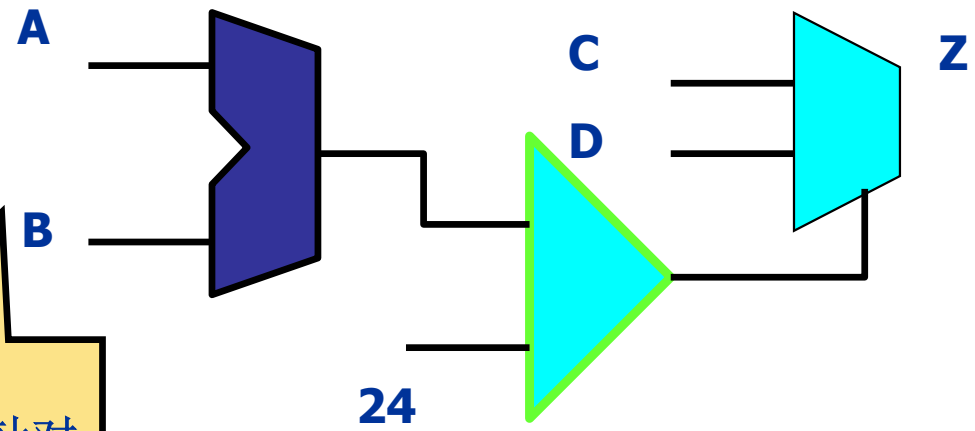
- 考虑下面的代码：

```
if (A + B < 24) then
    Z <= C;
else
    Z <= D;
end if;
```

综合后的电路图如下：

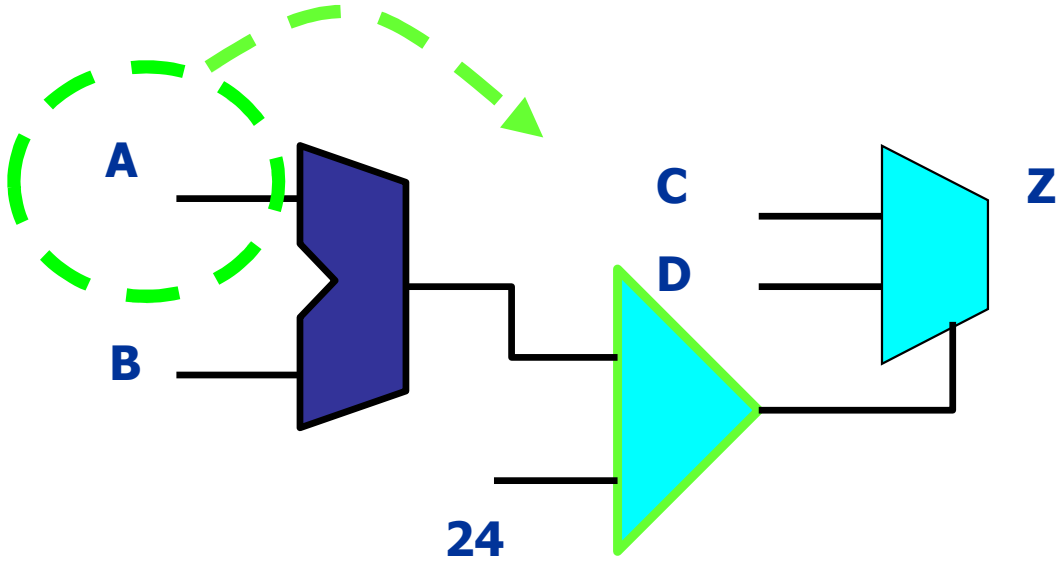


优化目标



假设A为late arrival signal。则考虑如何针对A进行优化，减小A到达Z的延时。

优化思路

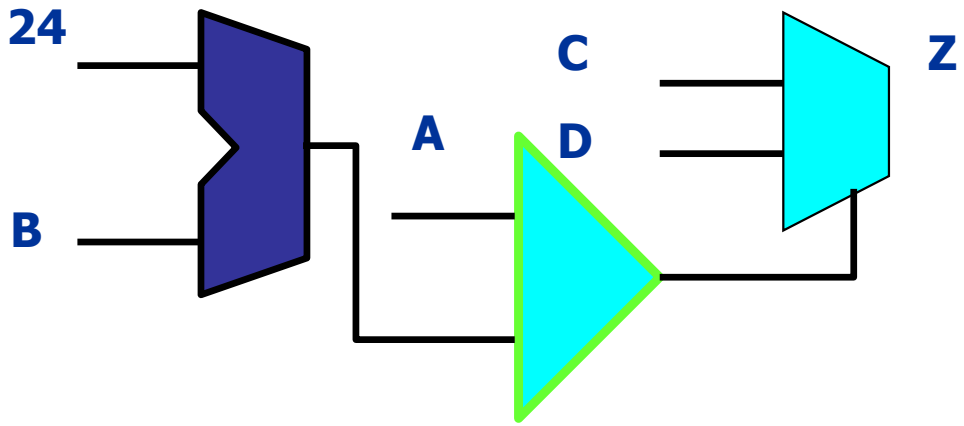


If(A+B < 24)

关键：尽量减小A”穿过”的运算步骤(级数)！！

If(A < 24-B)

优化思路



If(A+B < 24)

关键：尽量减小A”穿过”的运算步骤(级数)！！

If(A < 24-B)

优化后的代码

```
if (A < 24 - B) then
    Z <= C;
else
    Z <= D;
end if;
```

阶段性总结

- 以上“优先级电路的延时优化”、“数据通路拷贝”和“数据运算式变换”三种方法实际上都属于“信号搬移”方法。
- 其要点是想办法把形成关键路径的“迟到”信号尽量往路径输出端(路径末端)搬移，以减少路径延时。

延时优化的几个要点

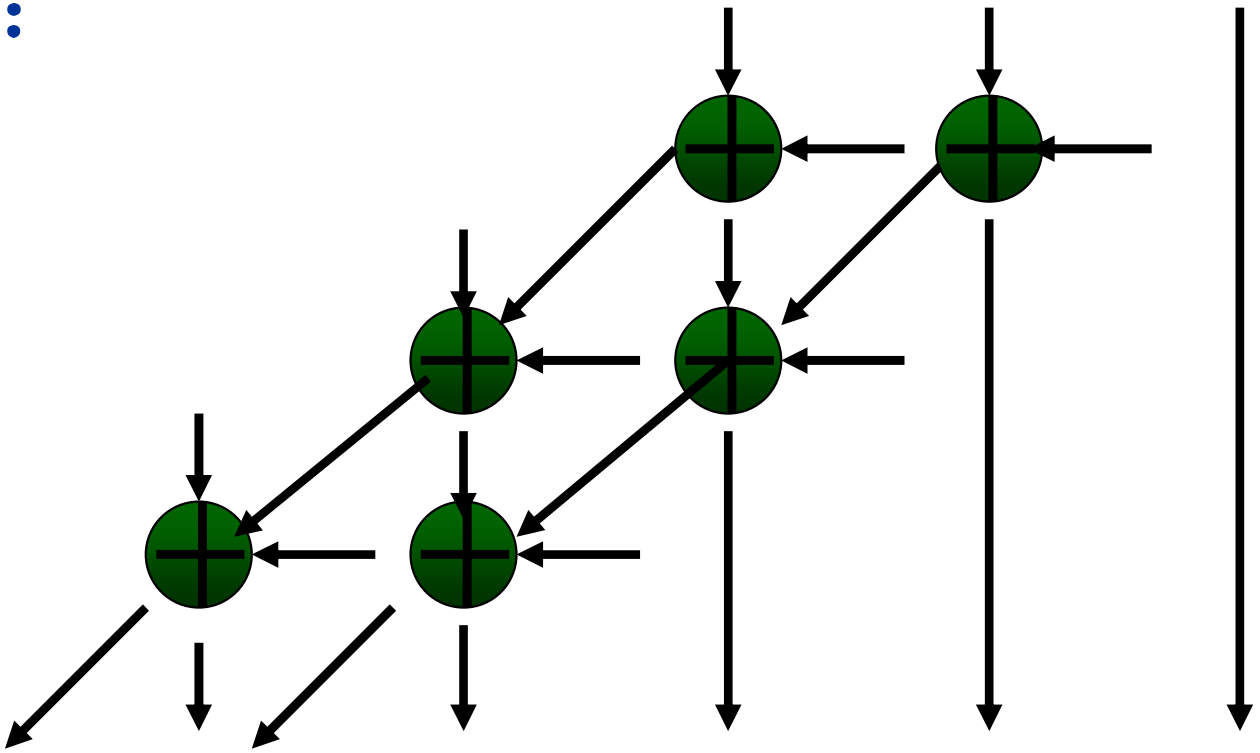
1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

变量运算优化

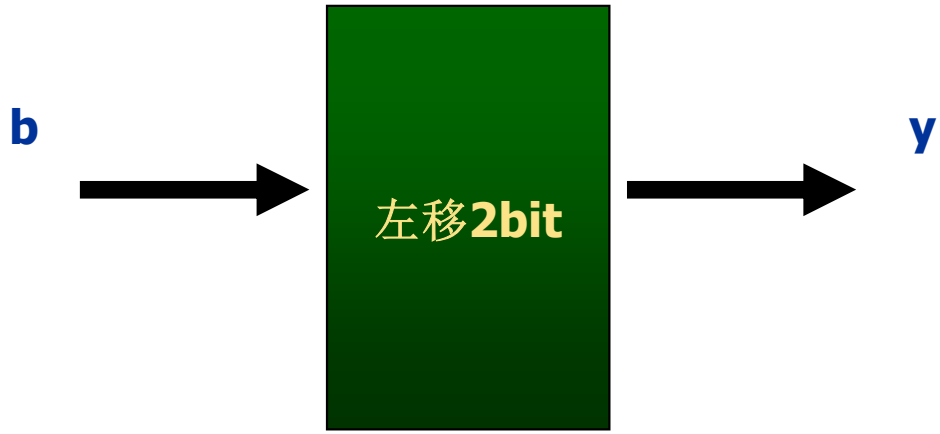
- 注意这里所说的“变量”指的是“非常量”，其包含VHDL中的“变量”和“信号”。
- 变量计算和常量计算的综合结果在组合逻辑规模上有着天壤之别。

例：变量乘法与常量乘法

- 假设a, b均为3bit信号(变量)，则y <= a * b的结果可能为(假设a, b，y均为无符号数):



- 假如在硬件运行的过程中，a始终为一个常数100，则可以将代码改成 y <= b * “100”。此时综合得到的电路为：



- 可见，如果能够用常量运算，就尽量用常量运算表达式。误用变量表达式虽然在逻辑上并没有错误，但是会导致组合电路规模过大，不但浪费芯片资源，而且降低系统工作速率。

延时优化的几个要点

1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

组合路径切割

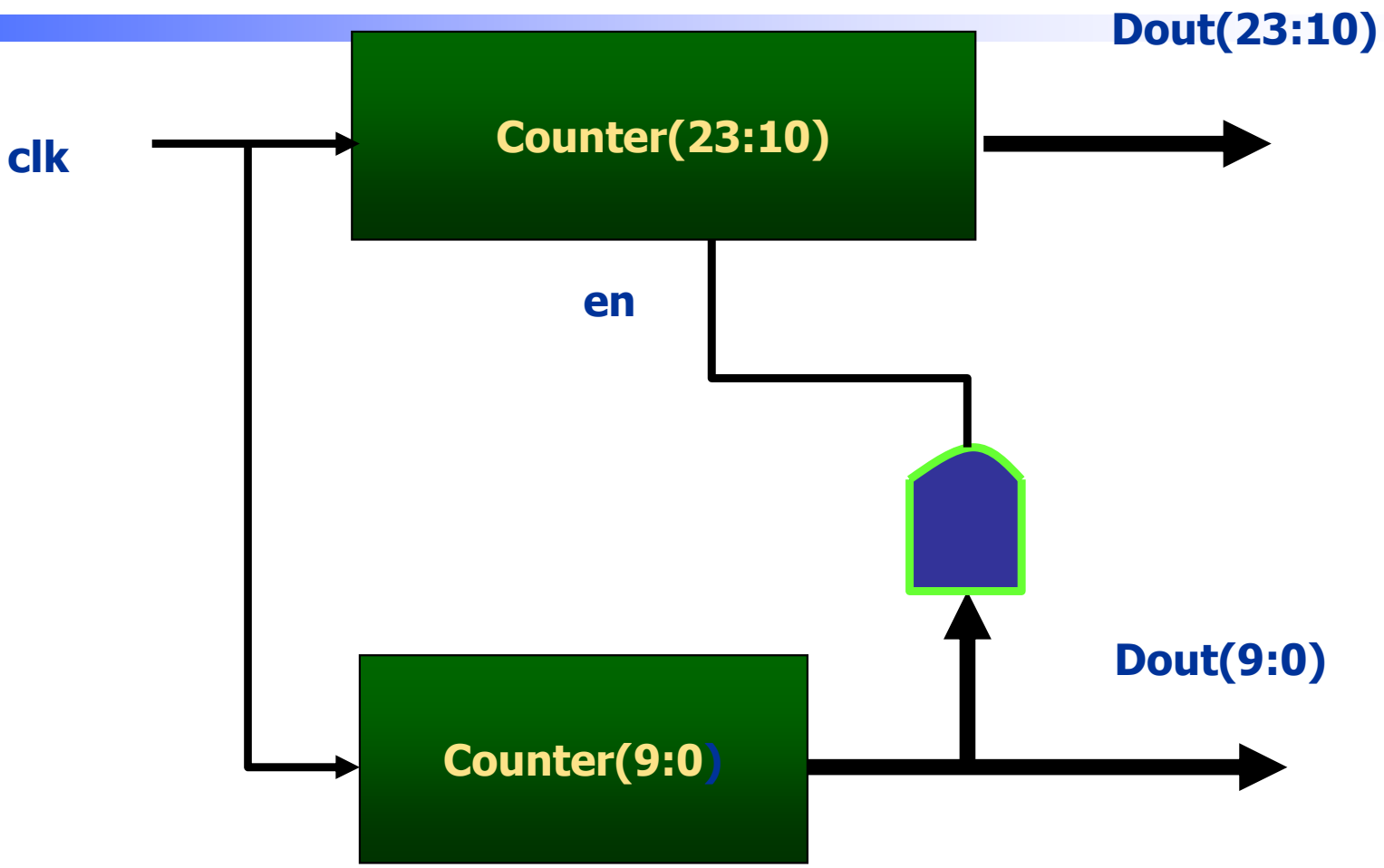
- 常用手段有：
  1. 状态机拆分；
  2. 流水线技术；
  3. 其它等。

状态机拆分

- 经典案例：预定标计数器。
- 设计一个32bit的计数器，采用binary编码。按照常规的设计：
- If(reset = ‘1’) then
- dout <= (others => ‘0’);
- Elsif(clk’event and clk = ‘1’) then
- dout <= dout + ‘1’;
- End if;

综合后的速度

- 用Mentor Graphics LeonadoSpectrum针对SCL05u ASIC库进行综合，得出关键路径延时为约14.07 ns，则clk的最高工作频率约为69.6 MHz。
- 为了提高该设计的最高工作频率，修改方案：


Dout(23:10)
clk
Counter(23:10)
en
Dout(9:0)
Counter(9:0)

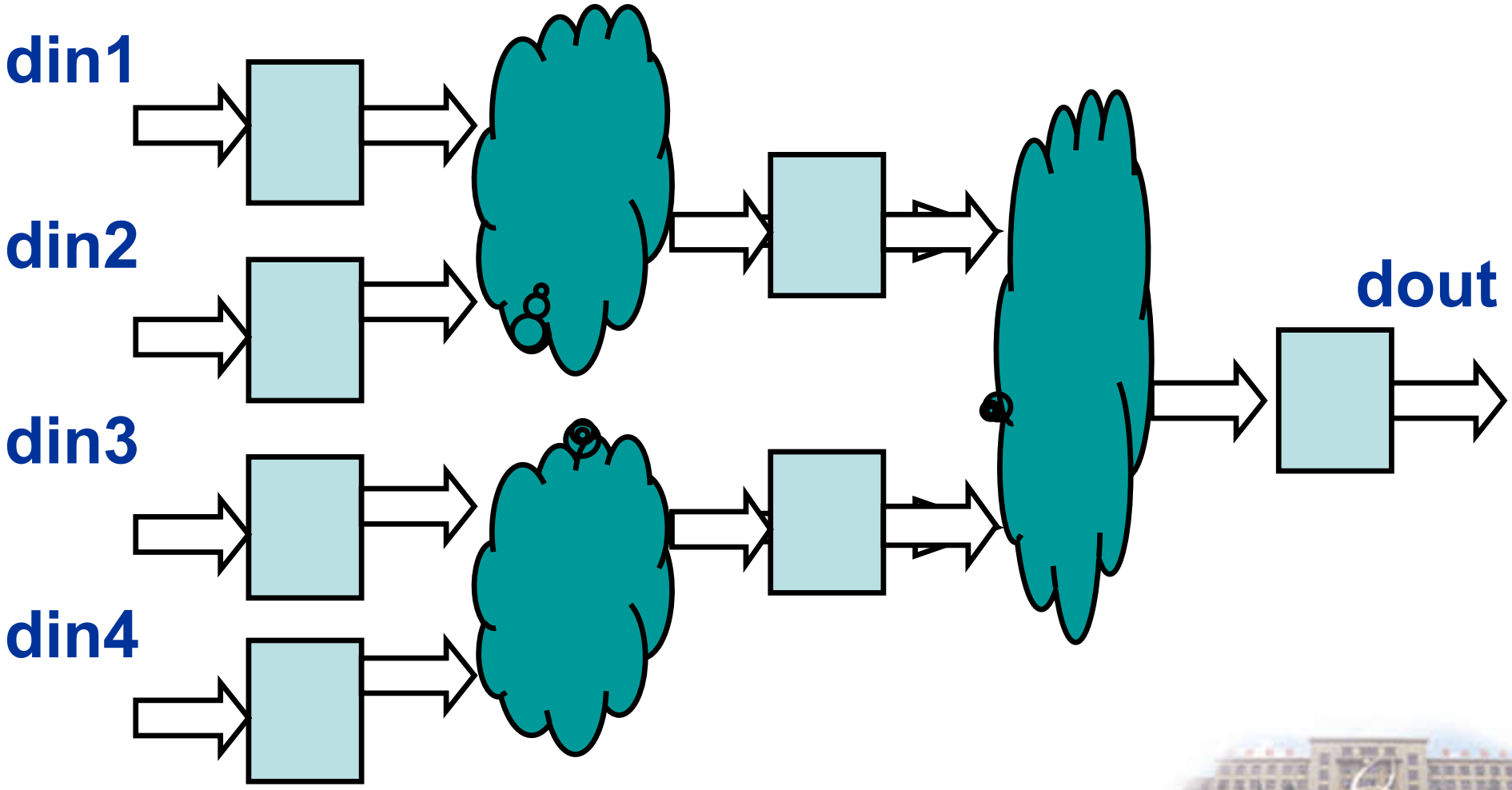
- 注意上图中en的高脉冲宽度为一个clk时钟周期。
- 综合后，其关键路径延时为6.40 ns，clk的最高工作频率可达140.8 MHz !!
- 可见，进行状态机拆分后，各状态机的状态译码的组合电路规模都比原来小，从而减小了关键路径的延时。

流水线技术

- 流水线技术几乎是最常用的提供系统工作速率的强有力手段。

流水线设计例子

- 设计din1 + din2 + din3 + din4结果输出给dout。



流水线技术

- 其思想是利用寄存器将一条长路径切分成几段小路径，从而达到提高工作速率的作用。
- 假设原路径延时为t，加入2级流水线并且假设路径切割均匀，则路径延时可以减少到约t/3，从而系统速率可以提高到原来的3倍左右。
- 当然要注意的是输出同时会往后推迟3个时钟周期。所以采用流水线技术时，要记得进行时序调整。

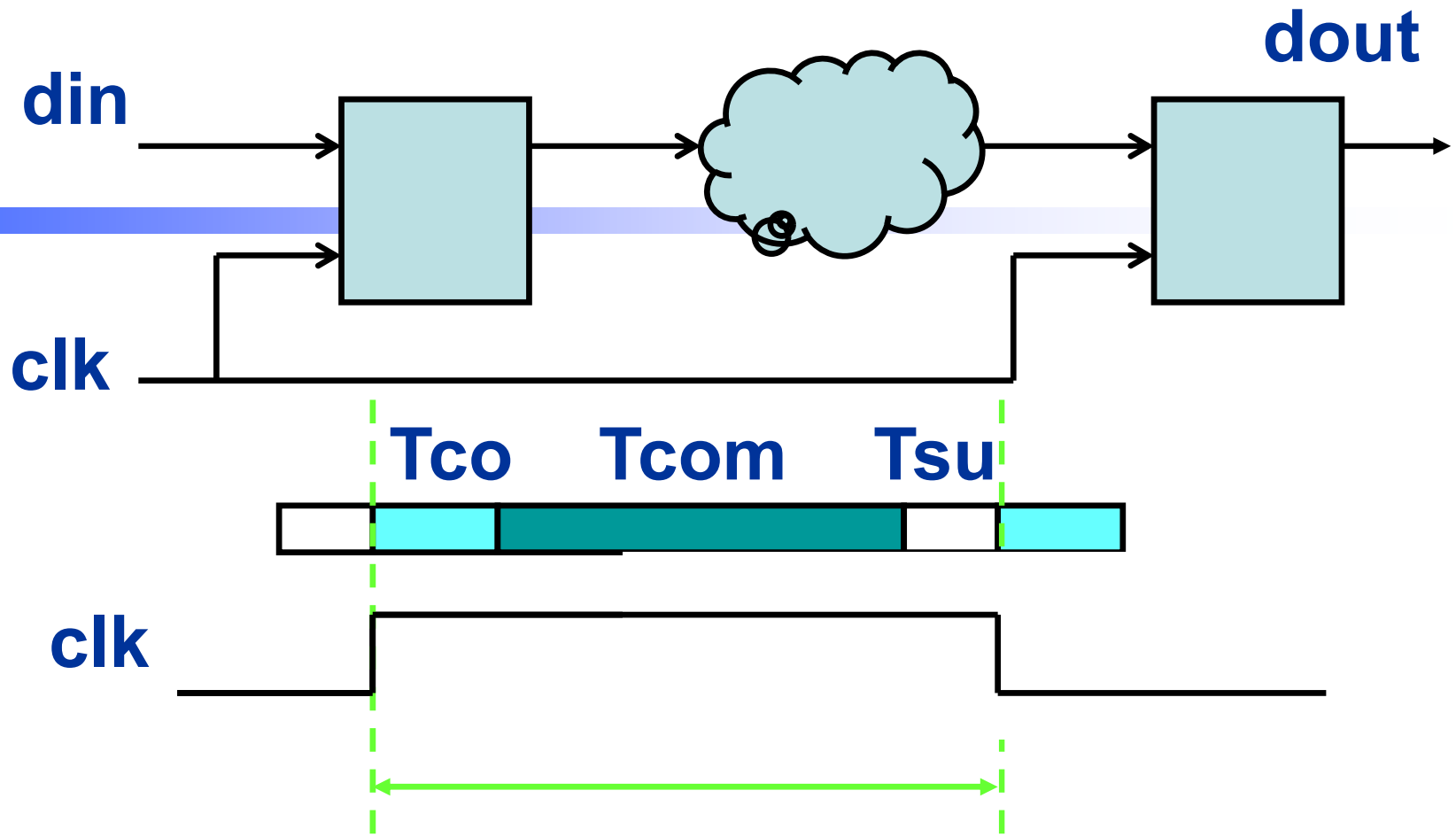
关于寄存输入和寄存输出

- 数字系统中，各模块应采取（寄存输入和）寄存输出，这样做有如下优点：

1. 模块化清晰(特别是寄存输出)；
2. 提高系统最高工作速率；
3. 有利于整个系统和单个模块分别进行静态时序分析。

延时优化的几个要点

1. 长路径的避免
2. 优先级电路的延时优化
3. 数据通路拷贝
4. 数据运算式变换
5. 变量运算优化
6. 组合路径切割
7. 双时钟沿问题
8. 其他

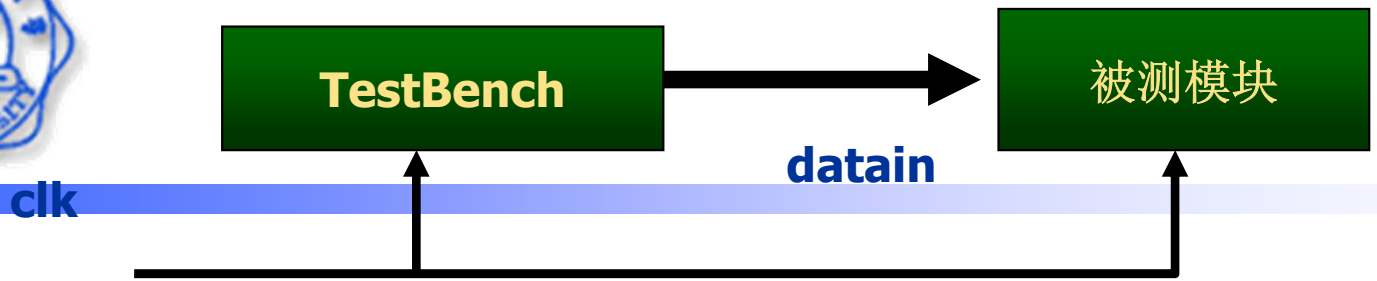


路径延迟

- 有人认为，以上系统可以设计为上升沿和下降沿都使用的系统，这样可以增大时钟沿的使用率，并且可以使得第二个寄存器能在数据稳定中间采样，从而保证数据的安全。这个说法对吗？为什么？

时钟驱动的TestBench

- TestBench是用行为风格的代码来设计的，
- 所以很多初学者喜欢用一堆after语句来生成同步系统所用的输入数据。
- 实际上，应该将TestBench当成一个也受同步系统时钟驱动的元件。



- 以下TestBench的写法一般情况下不可取：

  Datain <= datain + ‘1’ after clk_cycle;

- 推荐的写法应该是

  If(clk’event and clk = ‘1’) then

  datain <= datain + ‘1’ <after …>;

  end if;

异步设计优化

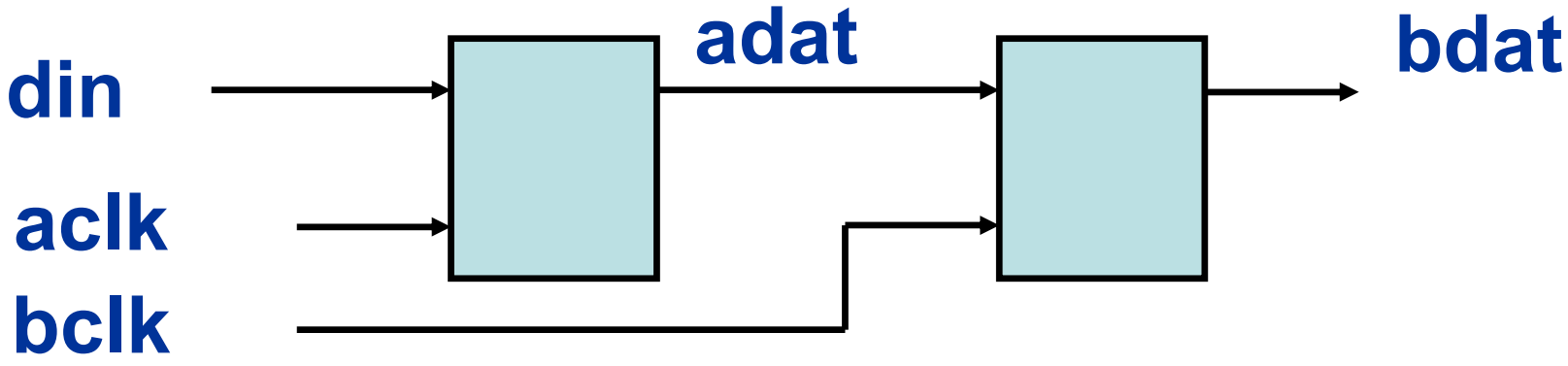
- 异步系统包括：
  - 全异步系统：硬件的行为不受时钟绑定，完全由自定时技术来完成信号的交互和握手。
  - 异步多时钟系统：系统的行为由多个异步的时钟来驱动。可包含“全局同步，局部异步”和“全局异步，局部同步”等。

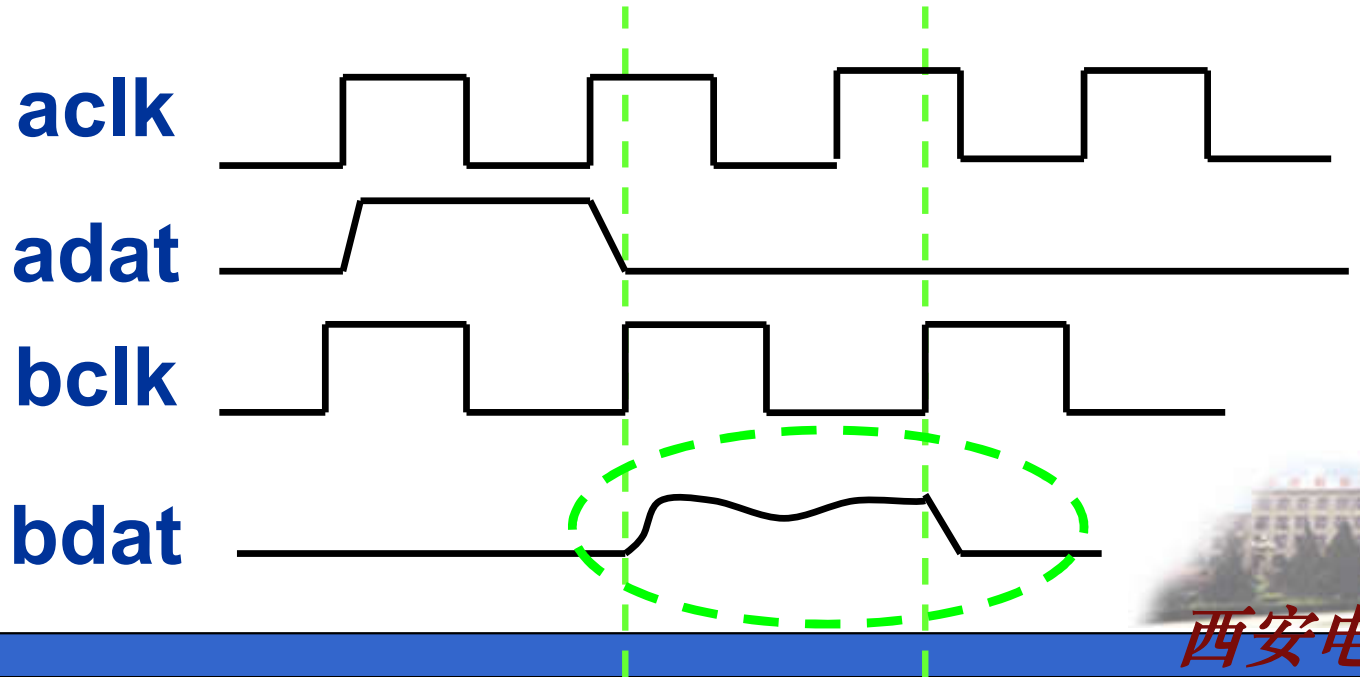
异步设计

- 全异步设计的特点：
  - 由于系统的行为不绑定在时钟上，因此不用考虑同步系统难度很大的全局时钟同步问题；
  - 避免了由时钟信号造成的连续漏电，以及为负责的功率管理系统而付出的开销。
  - 没有全局同步产生的电流瞬变，因此在低功耗设计方面有先天的优势。
  - 能达到的处理速度比同步系统高，因为它不用考虑最坏情况下的所谓“关键路径”。
  - 采用自定时技术，设计难度过高，在FPGA中无法实现。

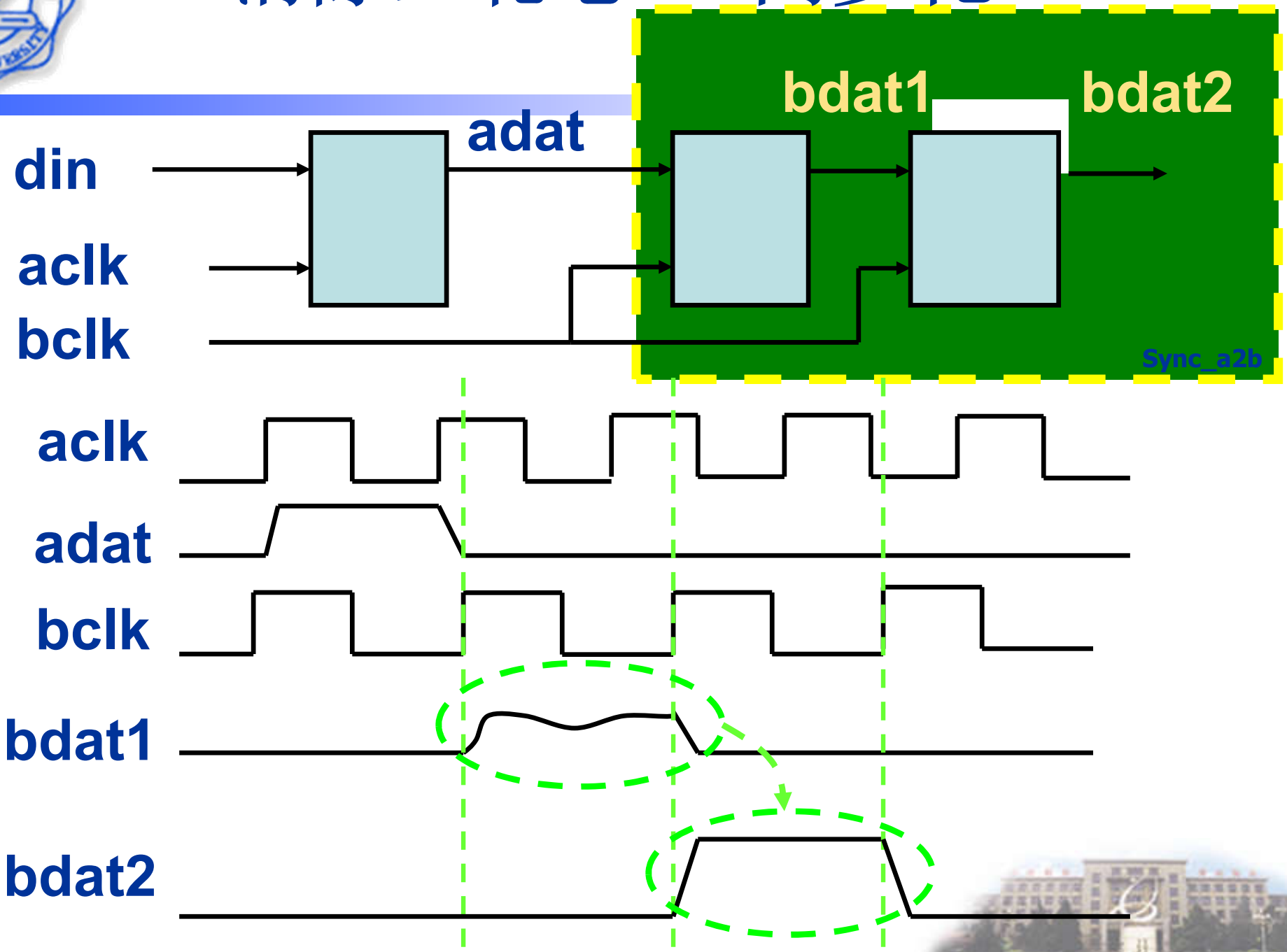
异步多时钟系统设计简介

- 如果一个系统中存在多个独立(异步)时钟，并且存在多时钟域(clock domain)之间的信号传输，那么电路会出现亚稳态。

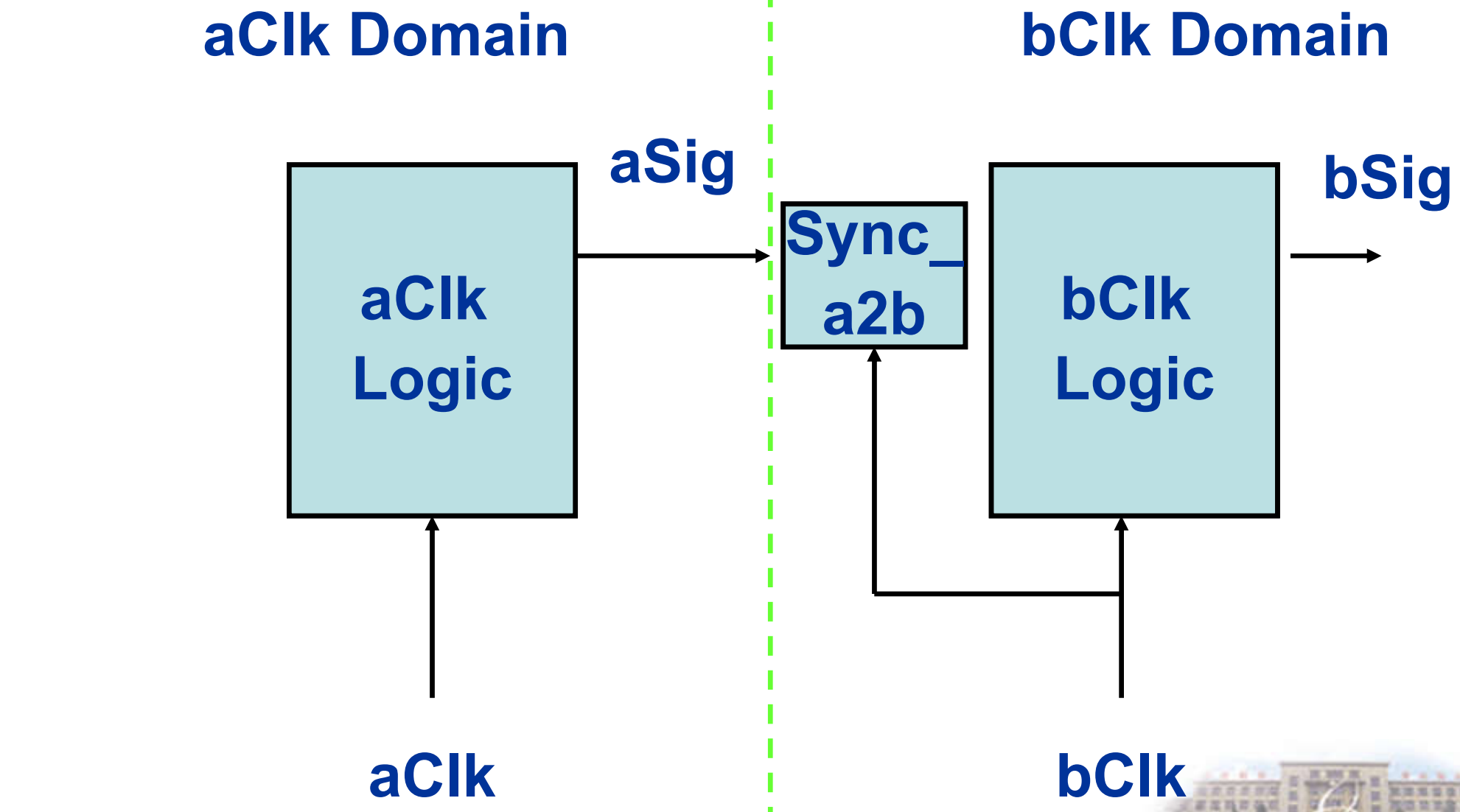




消除亚稳态----同步化


bdat1
bdat2
adat
din
aclk
bclk
Sync_a2b
aclk
adat
bclk
bdat1
bdat2

异步多时钟系统模型


aClk Domain
bClk Domain
aSig
bSig
Sync_
a2b
aClk
Logic
bClk
Logic
aClk
bClk

注意其信号命名和模块划分方法

- 这种信号命名和模块划分的方法有如下优点：
  - 有利于检查信号所通过的时钟域；
  - 有利于各模块进行单独的静态时序分析；
  - 有利于在静态时序分析中快速地设定false path；

异步信号穿越时钟域时，这些信号与异步时钟之间的相位关系数是无穷的，所以在整个系统静态时序分析时必须忽略这些信号路径。

多时钟域系统设计的经典案例：异步FIFO

- 数字系统设计当中，应该尽量避免使用异步多时钟，否则会带来很多潜在的问题(不仅仅是亚稳态的问题)。
- 关于多时钟域数字系统设计的方法，可以参考

《Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs》，Clifford E. Cummings。

以下只给出大概的结论。

异步设计的注意事项小结

- 注意使用同步化电路来对异步信号进行同步；
- 进行科学的模块划分和信号命名；
- 尽量减少握手控制信号的数目，以避免同步化造成的信号拉伸而破坏控制信号之间的相位关系；
- 快时钟域信号进入慢时钟域时，要注意信号丢失的避免和检测；
- 计数器要尽量采用Gray编码，以避免同步化造成的信号拉伸。

附：Gray ←→Bin

- gray[0] = bin[0] ^ bin[1];
- gray[1] = bin[1] ^ bin[2];
- gray[2] = bin[2] ^ bin[3];
- gray[3] = bin[3];

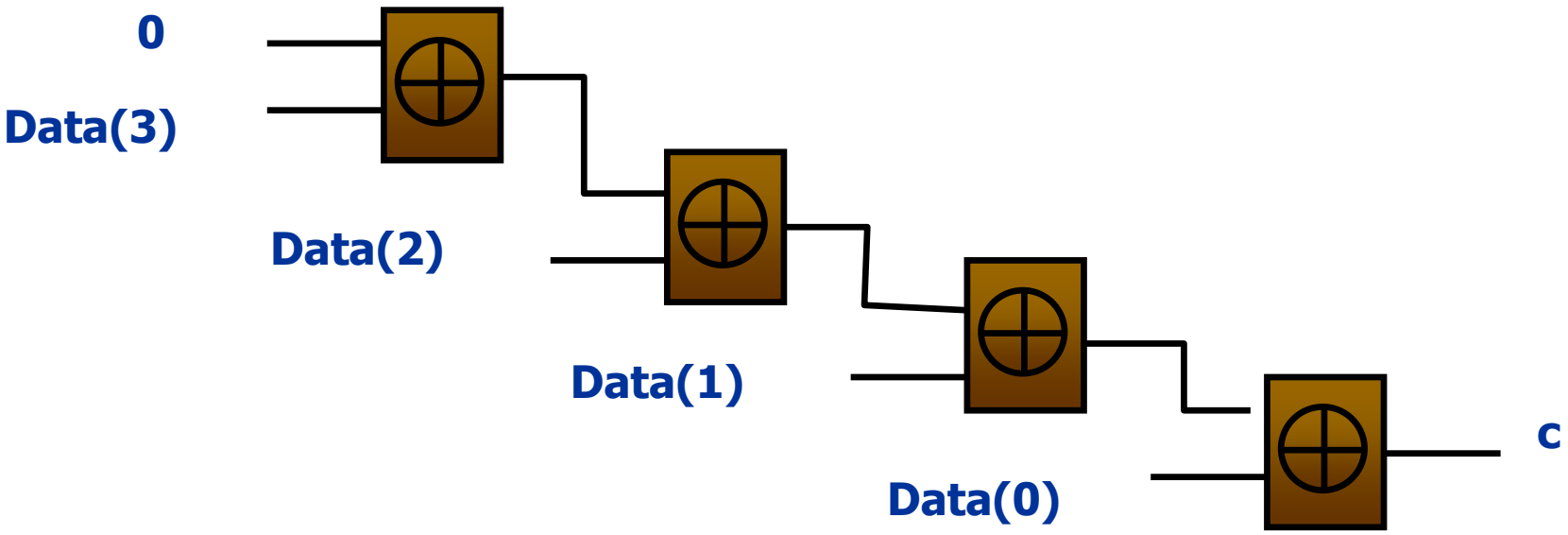
- `bin[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0];`
- `bin[1] = gray[3] ^ gray[2] ^ gray[1];`
- `bin[2] = gray[3] ^ gray[2];`
- `bin[3] = gray[3];`

串行转并 行：cascade → Tree (for循环的优化)

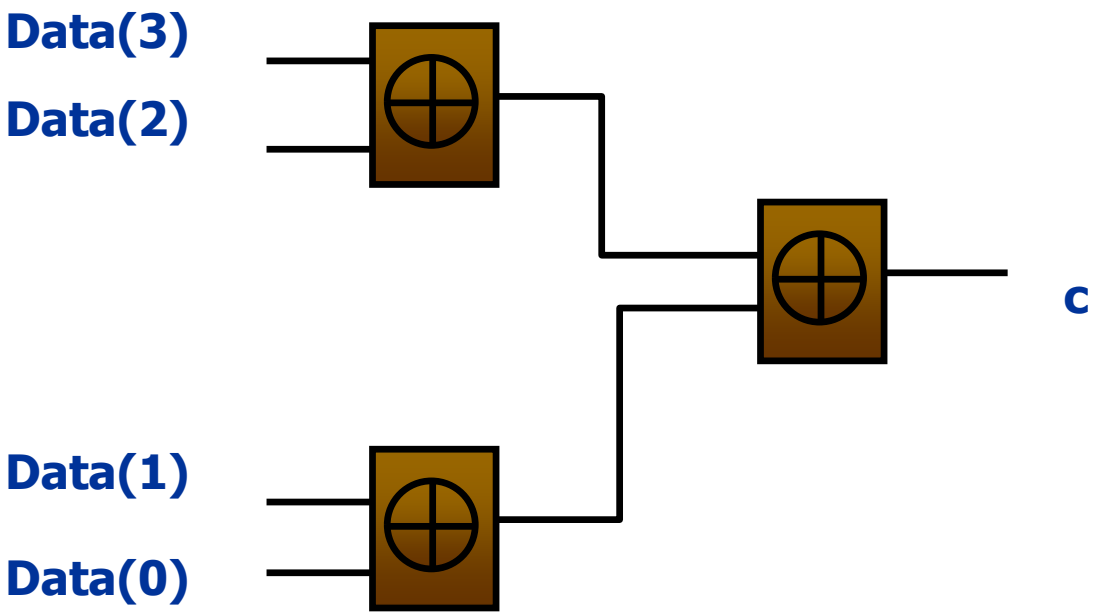
- 设计奇偶校验位生成电路：对数据data(3:0)生成校验位c。
- 一种思路如下：

```
result := '0';
for I in data'RANGE loop
result := result XOR data(I);
end loop;
C <= result;
```

综合后的电路图



改进思路(目标电路图)



改进的描述方法

```
Sig1 <= Data(3) xor Data(2);
Sig2 <= Data(1) xor Data(0);
C <= sig1 xor sig2;
```

问题: 对于Data(n:0)，代码该如何设计？

注：在串转并中，适当地使用平衡的二叉树结构，不但可以减少延时，而且有利于方便地引入流水线结构。

优化总结

- 正确设计：
  - VHDL结构体有三种描述风格：行为描述风格，RTL描述风格和结构描述风格；
  - RTL风格的描述中，要注意一个进程里只能判断一次时钟沿；注意避免X状态的传递；
  - 敏感信号表不完整，往往会造成前仿真和后仿真结果不一致；
  - 条件判断语句中，要注意对所有分支进行输出信号的赋值，以避免锁存；

优化总结

- 巧妙地使用无关态，可以引导综合器综合出优化的电路；
- case无优先级；single if和multiple if语句的优先级顺序；
- 硬件描述中，应加强硬件思维，打破软件思维；
- 注意要正确地使用多路开关或者三态缓冲来进行总线复用，避免总线冲突；

优化总结

- 双向端口描述中，注意高阻态的赋值；注意其测试矢量生成时，也要注意设置高阻态；
- 毛刺消除的要点：竞争和冒险的避免；Gray编码；寄存器消除方法(数据输入吸收，时钟使能吸收)。

优化总结

- 同步设计
  - 尽量避免使用门控时钟，应以时钟使能来代替；
  - 尽量避免使用派生时钟，应以派生使能来代替；
  - 路径与路径延时的含义；路径延时与系统工作时钟的最高频率的关系；
  - 缩小组合电路的规模，可以降低系统的路径延时从而提高最高工作速率；

优化总结

- 善于使用“信号搬移"技术进行延时优化，可以提高系统速度；手段包括优先级别电路中的模块搬移、数据通路拷贝、数据运算式变换等；
- 在优先级别电路中进行信号搬移时，要注意修正条件判断式，使得优先级别不变；
- 可能的情况下，应该尽量使用常量运算来代替变量(信号)运算；

优化总结

- 可以使用状态机拆分、流水线等技术进行路径切割，达到延时优化的目的；
- （寄存器输入和）寄存输出不仅可以提高系统工作速率，而且有利于模块和整体进行分别的静态时序分析；

优化总结

- 异步设计
  - 尽量避免使用异步设计；
  - 异步多时钟系统中，注意做好时钟域之间的同步；
  - 引入时钟域同步后，容易造成信号相位拉伸；
  - 为避免矢量拉伸对数据矢量的破坏，可以考虑采用Gray码进行计数器编码；
  - 为了在静态时序分析中快速地进行false path的设置，最好根据时钟域来进行信号命名。

高速设计的其他手段

（1）串转并不但可以降低系统的延时，而且可以大幅度提高系统的数据吞吐率。二叉树结构可以有助于引入平衡的流水线。

（2）安全的状态机描述：注意使用others分支，并且该分支不能以null来规定其行为，以处理状态机的“跑飞”。

课程结束语

- SoC关键技术分析
- SoC总线结构
- 逻辑电路设计基础
- 加法器/乘除法器设计
- 存储器设计
- CPU设计
- 优化设计

继续努力，不畏艰难，
勇攀科学技术高峰