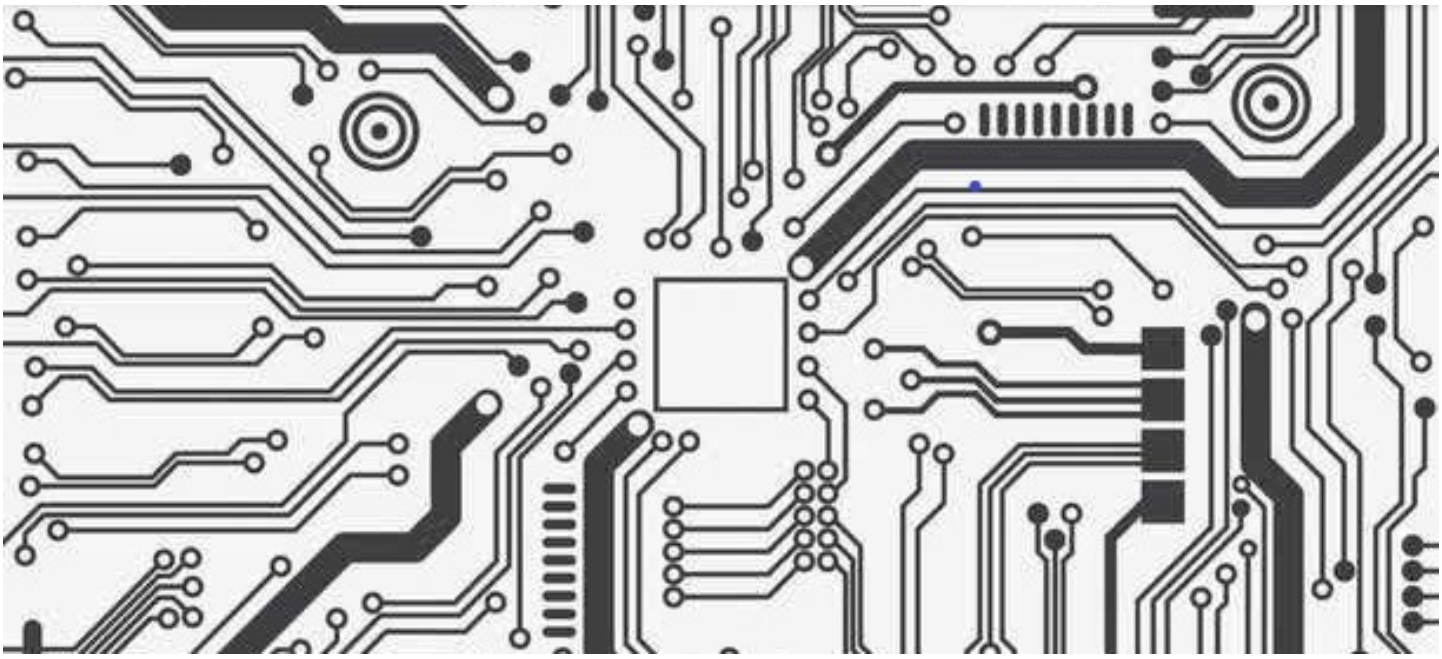


芯片设计进阶之路——Reset深入理解

烃围玮未
学习也是一种生活方式

已关注

ljgibbs 等 74 人赞同了该文章

Reset深入理解

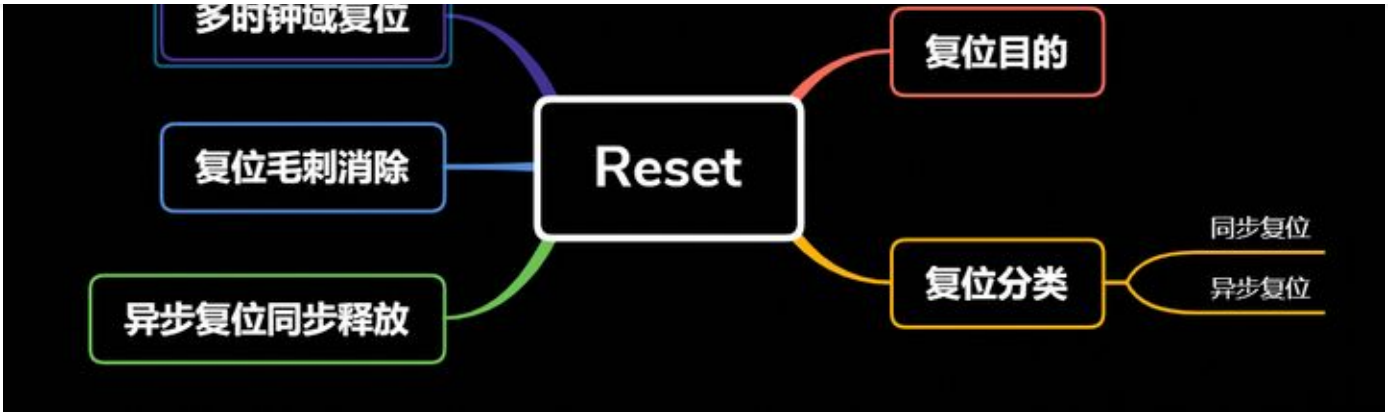
版权声明：

本文作者： 烃围玮未

微信公众号：芯片设计进阶之路

首发于知乎专栏《芯片设计进阶之路》，转发无需授权，请保留这段声明。

首先上思维导图：



如果要问“芯片中怎么复位才对？”

很多人都会回答“异步复位同步释放”。

但是为什么要用异步复位同步释放，是不是所有的芯片都必须采用这种方式，估计很少人能回答上来。那么让我们从为什么要复位开始。

1. 复位的目的

复位信号在数字电路里面的重要性仅次于时钟信号。对一个芯片来说，复位的主要目的是使芯片电路进入一个已知的，确定的状态。主要是触发器进入确定的状态。在一般情况下，芯片中的每个触发器都应该是可复位的。在某些情况下，当在高速应用程序中使用流水线触发器(移位寄存器触发器)时，为了实现更高的性能设计，可能会从某些触发器中消除复位。这种类型设计需要在复位激活期间，运行预先确定数量的时钟周期，以使ASIC处于已知的状态。

1.1 为什么需要复位呢?

1）复位可以使电路从确定的初始状态开始运行:

上电复位：上电的时候，为了避免上电后进入随机状态而使电路紊乱，这个时候就需要上电复位。

中间复位：有时候，要求电路从初始状态开始执行电路的功能，要对电路进行复位，让它从最初的状态开始运行。

2) 复位可以使电路从错误状态回到可以控制的确定状态:

3) 电路仿真时需要电路具有已知的初始值

在仿真的时候，信号在初始状态是未知状态(也就是所谓的x，不过对信号初始化之后的这种情况除外，因为仿真的时候对信号初始化就使信号有了初始值，这就不是x了)。

对于**数据通路**(数字系统一般分为数据通路和控制通路，数据通路一般是对输入的数据进行处理，控制通路则是对运行的情况进行操作)，在实际电路中，只要输入是有效数据(开始的时候可能不是有效的)，输出后的状态也是确定的;在仿真的时候，也是输入数据有效了，输出也就确定了。也就是说，初始不定态对数据通路的影响不明显。

对于**控制通路**，在实际电路中，只要控制通路完备(比如说控制通路的状态机是完备的)，即使初始状态即使是不定态，在经过一定的循环后，还是能回到正确的状态上;然而在仿真的时候就不行了，仿真的时候由于初始状态为未知态，控制电路一开始就陷入了未知态;仿真跟实际电路不同，仿真是“串行”的，仿真时控制信号的初始不定态会导致后续的控制信号结果都是不定态，也就是说，初始的不定态对控制通道是致命的。

1.2 不需要复位的情况

复位信号很重要，但是并不是每一部分的电路都需要复位电路，一方面是复位电路也消耗逻辑资源、占用芯片面积，另一方面是复位信号会增加电路设计的复杂性(比如要考虑复位的策略、复位的布局布线等等)。

当某个电路的输出在任何时刻都可以不受到复位信号的控制就有正确的值时，比如说数据通路中的对数据进行处理的部分。在某些情况下，当流水线的寄存器(移位寄存触发器)在高速应用中时，应该去掉某些寄存器的复位信号以使设计达到更高的性能，因为带复位的触发器比不带复位的触发器更复杂，反应也更慢。

2. 同步复位

2.1 同步复位的实现方式

同步复位的前提是，复位信号只会在时钟的有效边沿去影响或者复位flip-flop。Reset可以作为组合逻辑的一部分送给FF的D端。这种情况下，编码方式必须是if/else 优先级的方式，而且reset只能放在if条件下，其他组合逻辑放到else逻辑下。

```
module sync_resetFFstyle (
  output reg q,
  input      d, clk, rst_n);

  always @(posedge clk)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Correct way to model a flip-flop with synchronous reset using Verilog-2001

如果没有严格遵守这种方式，会有两个问题：

1、一些仿真器中，基于逻辑方程，逻辑可能组织复位到达触发器。这只是一个仿真问题，不是硬件问题。但是复位的一个主要目的，就是仿真的时候将电路置于一个已知的确定状态。

2、由于复位树的高扇出，复位信号可能是一个相对于时钟周期的“延迟到达信号”，尽管复位将从复位缓冲区树中进行缓冲，但明智的做法是限制复位到达本地逻辑后必须经过的逻辑量。就是说必须对复位信号少做逻辑。

下面的列子是一个同步复位的，带进位的计数器(loadable counter with synchronous reset)

```
module ctr8sr (
  output reg [7:0] q,
  output reg       co,
  input      [7:0] d,
  input            ld, clk, rst_n);

  always @(posedge clk)
    if      (!rst_n) {co,q} <= 9'b0;      // sync reset
    else if (ld)     {co,q} <= d;         // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Verilog-2001 code for a loadable counter with synchronous reset

它的电路图如下：

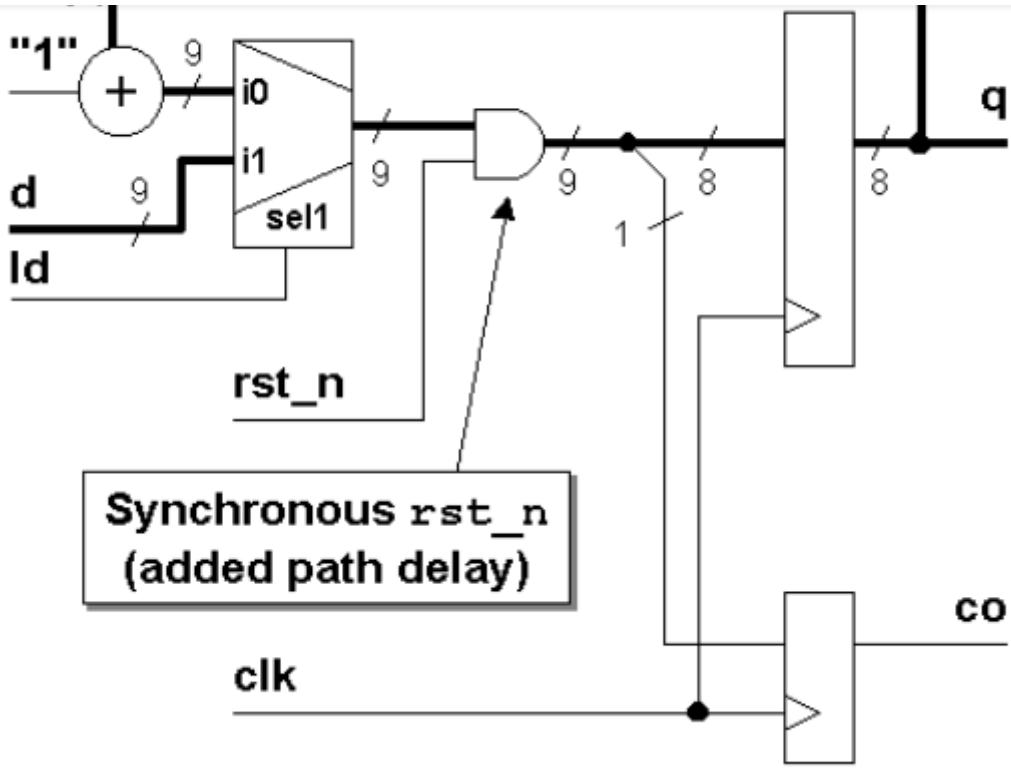


图1 Loadable counter with synchronous reset

同步复位的有一个问题就是，综合工具不能很好的把reset信号和其他的信号区分开。比如上面的电路也可能被综合成另外的电路，如下所示：

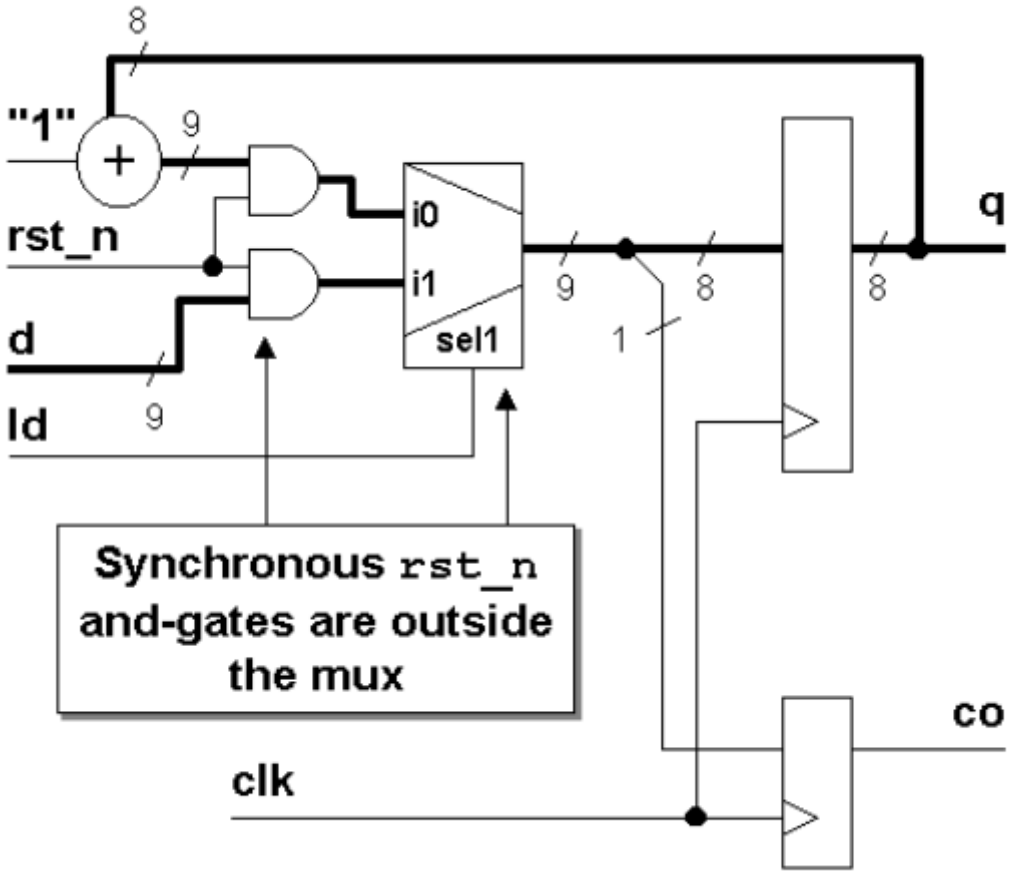


图1和图2是完全相同的。唯一的区别是图2的reset信号被提前到了MUX之前。通过rst_n拉低，可以强制MUX的两个分支的输入为0，但是如果ld是未知的(X)，并且MUX模型是悲观的，则会保持未知(X) 不会被复位。注意，这只是在仿真过程中出现的问题! 实际的电路可以工作正确并将其重置为0。

Synopsys提供编译器指令**sync_set_reset**，该指令告诉综合工具给定信号是同步reset(or set)。合成工具将这个信号“拉”到尽可能接近触发器，防止这种初始化问题的发生：

// synopsys sync_set_reset "rst_n"

这个命令只会影响综合，不会影响逻辑行为，所以推荐在同步复位每个模块都加上这个信号。

另外，可以在读取RTL之前将合成变量**hdlin_ff_always_sync_set_reset**设置为-true，这样就可以得到相同的结果，而不需要在代码本身中执行任何指令。

2.2 同步复位的优点和缺点

同步复位的优点如下

(1) 同步复位会综合成更小的触发器，特别当reset生成逻辑电路作为触发器D输入，但是这种情况下组合逻辑电路的数量变多，所以总的门电路节省不是那么显著。

(2) 同步复位确保电路100%是同步的。

(3) 同步复位确保复位只发生在时钟有效边沿，对小的复位毛刺来说，时钟就像滤波器。

(4) 在一些设计中，复位必须由内部条件产生。同步复位能过滤时钟间逻辑等式的毛刺。

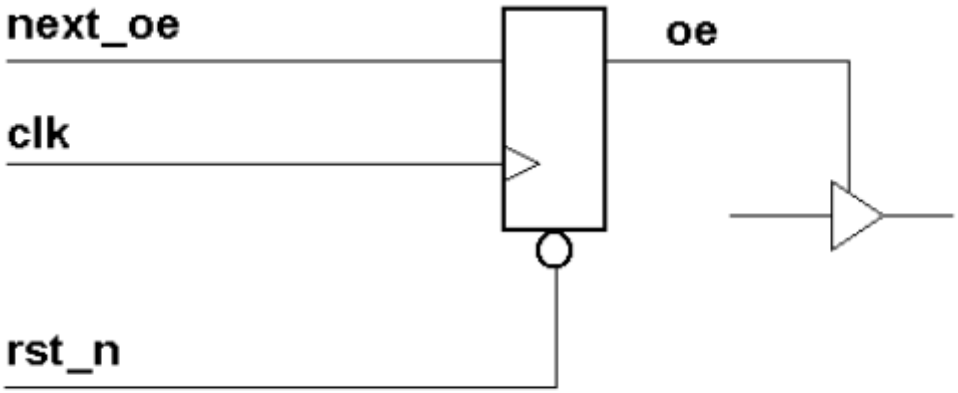
(5) 通过使用同步重置和预先确定的时钟数量作为复位过程的一部分。可以在复位缓冲区树中使用触发器，来帮助将缓冲树的时序保持在一个时钟周期以内。

同步复位的缺点如下：

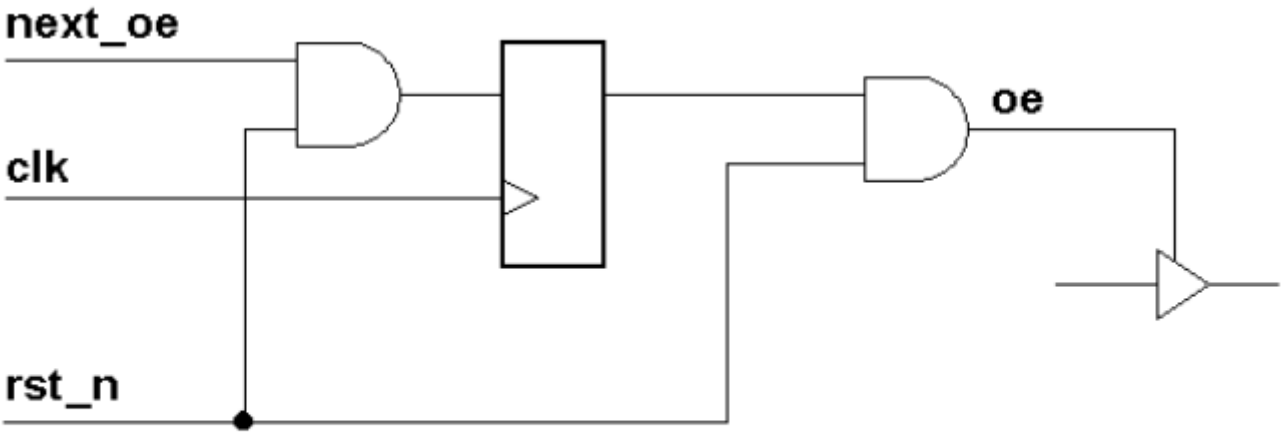
(1) 不是所有的库都有自带同步reset的FF, 但是可以通过把reset当作数据输入来解决；

(2) 同步复位需要一个脉冲延伸器保证复位脉冲足够宽能，够被有效时钟沿采集到；特别是在多时钟设计中；

(4) 如果电路中有三态总线，那么上电时必须用异步复位，如果用同步复位，reset必须能够复位三态信号的enable信号。如下所示



Asynchronous reset for output enable

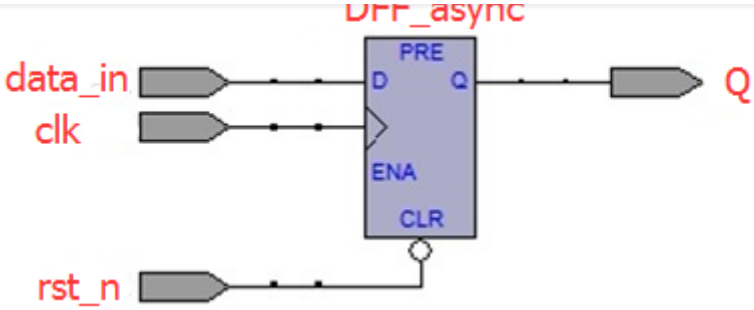


Synchronous reset for output enable

3. 异步复位

3.1 异步复位的实现

异步复位触发器则是在设计触发器的时候加入了一个复位引脚，也就是说**复位逻辑集成在触发器里面。**(一般情况下)低电平的复位信号到达触发器的复位端时，触发器进入复位状态，直到复位撤离。带异步复位的触发器电路图和RTL代码如下所示:




```
module async_resetFFstyle (
  output reg q,
  input      d, clk, rst_n);

  // Verilog-2001: permits comma-separation
  // @(posedge clk, negedge rst_n)
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Correct way to model a flip-flop with asynchronous reset using Verilog-2001

关于异步复位的综合约束，推荐将输入reset设置为

set_ideal_network + false path

3.2 异步复位的优点

1. 异步复位的最大优点是，vendor库里面有异步复位FF， 这样datapath就十分干净。不用把reset与数据做逻辑，这样复位路径上就不会有额外的延时，也不会受外部信号的干扰。

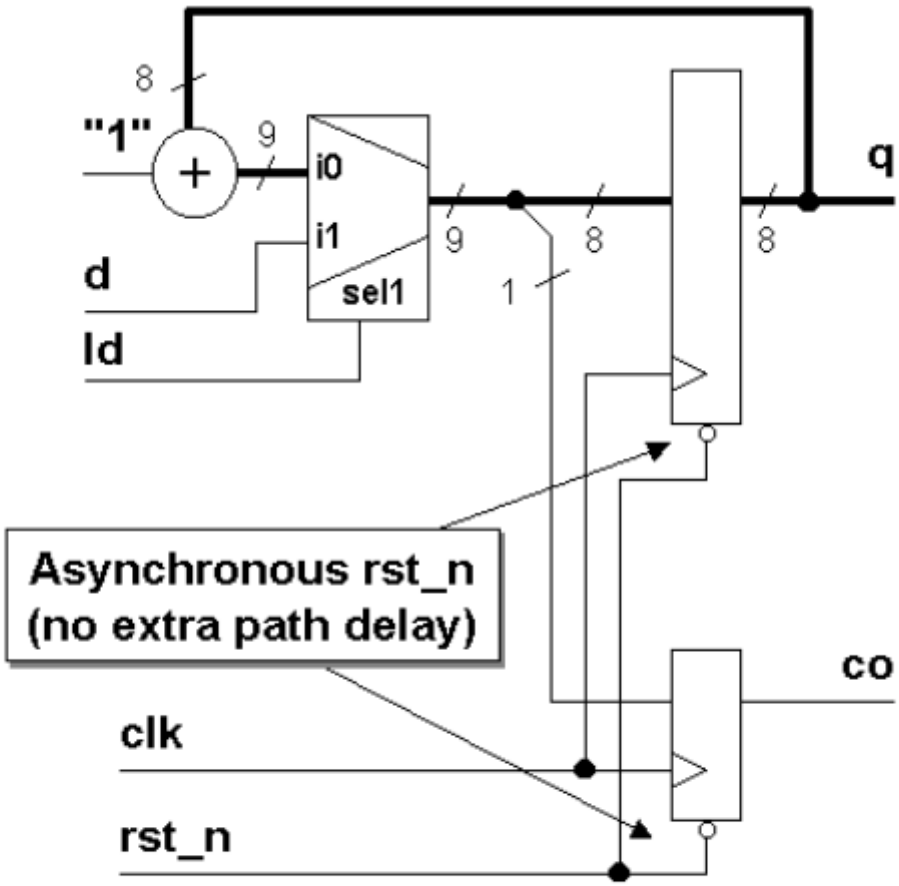
异步复位的进位计数器如下：

```
  output reg        co;
  input      [7:0] d;
  input            ld, rst_n, clk;

  always @(posedge clk or negedge rst_n)
    if       (!rst_n) {co,q} <= 9'b0;      // async reset
    else if (ld)      {co,q} <= d;         // sync load
    else              {co,q} <= q + 1'b1; // sync increment
endmodule
```

Verilog-2001 code for a loadable counter with asynchronous reset

综合后的电路如下：



Loadable counter with asynchronous reset

可以看到，reset路径上十分干净，直接由外部pin来控制，不受其他信号影响。

2. 异步复位的另一个优点是电路reset和时钟无关，不管有没有时钟，都可以reset。好处是可以实时复位，也可以加在门控时钟里面。门控时钟是低功耗设计的重要方法。

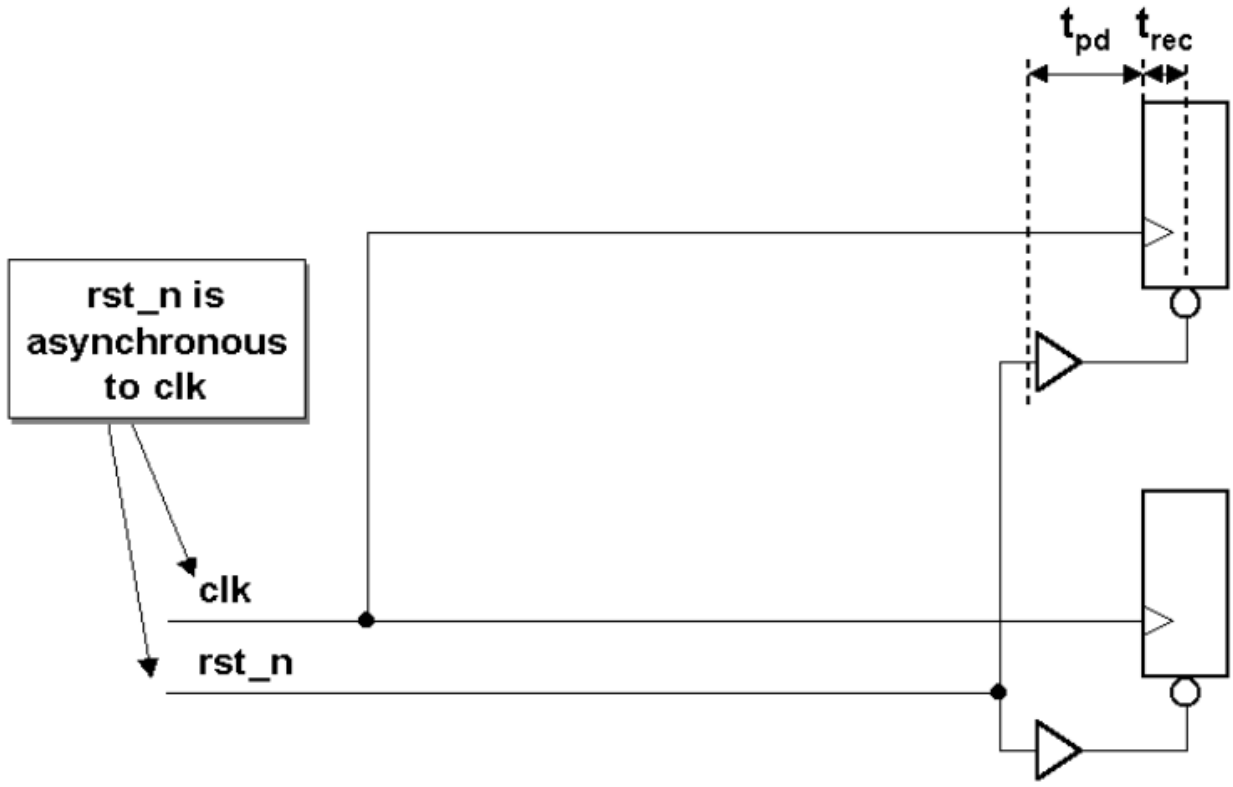
3.3 异步复位的缺点

1. 异步重置的最大问题是它们是异步的，在复位阶段和解复位阶段（复位撤离）都是异步的。复位阶段不是问题，解复位才是问题。如果在触发器的活动时钟边缘或附近释放异步复位，则触发器的输出可能变为亚稳态，这样电路的复位状态可能会丢失，解复位失败。

如下图所示， 异步复位信号复位解除时是和时钟信号完全异步的。

这种情况下由两个问题：（1）复位恢复时间（reset recovery time） 违例

(2) 复位解除（reset removal) 发生在不同的触发器的不同时钟周期



Asynchronous reset removal recovery time problem

下面先解释一下两个概念：**复位恢复时间**(reset recovery time)和**复位解除时间**(reset removal time)

复位恢复时间： 解除复位信号时，**复位边沿**(当然是从有效变成无效的跳变时刻，通常是0->1那个时间点）与**下一个有效时钟沿**之间的这段时间。对应**建立时间**

复位解除时间： 解除复位信号时，**复位边沿**与**上一个有效时钟沿**之间的这段时间。对应**保持时**

1). 不满足复位恢复时间或者撤离时间，**可能**会导致亚稳态问题。（注意是可能）因为如果输出本身就是复位后的值，即使当前时钟沿不能判断是否复位，输出也是复位值，这时候就不会产生亚稳态，因为已经是复位态了。

2). 不满足复位恢复时间或者撤离时间可能会导致不同FF复位状态不一致的问题。复位信号和时钟信号一样，通过复位网络到达各个触发器。复位网络具有非常大的扇出和负载，到达不同的触发器存在不同的延时，不满足复位恢复或者解除时间的情况下，就有可能在不同的触发器的不同时钟周期内进行解复位。注意，这里的假设条件是复位树和时钟树已经做成立平衡状态，不再考虑复位树和时钟树没做好的情况。

既然同步复位和异步复位都有问题，那么到底应该怎么复位呢？能不能即有同步复位和异步的优点，而没有同步复位和异步复位的缺点呢？小孩子才做选择，成年人就是我都要。所以解决方案就是： **异步复位同步释放。**

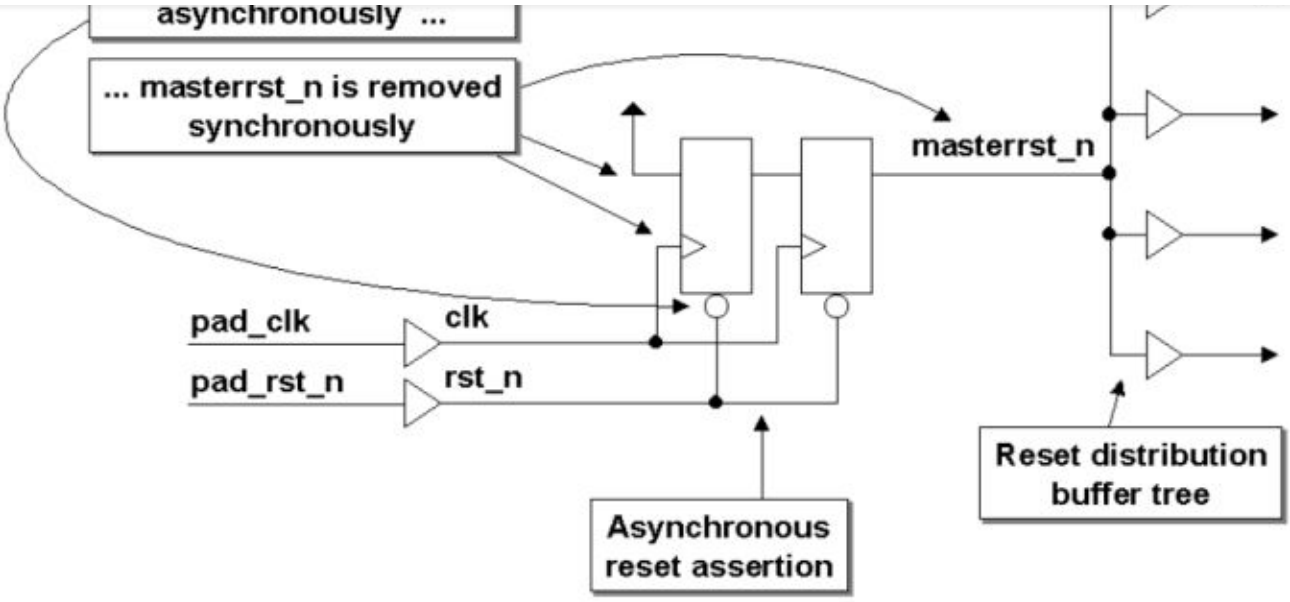
4. 异步复位同步释放

异步复位的同步释放电路也称为**复位同步器。**

规则：每个异步复位的电路，必须包含一个复位同步器。代码和电路如下：

```
module async_resetFFstyle2 (
  output reg rst_n,
  input      clk, asyncrst_n);
  reg        rff1;

  always @(posedge clk or negedge asyncrst_n)
    if (!asyncrst_n) {rst_n,rff1} <= 2'b0;
    else             {rst_n,rff1} <= {rff1,1'b1};
endmodule
```

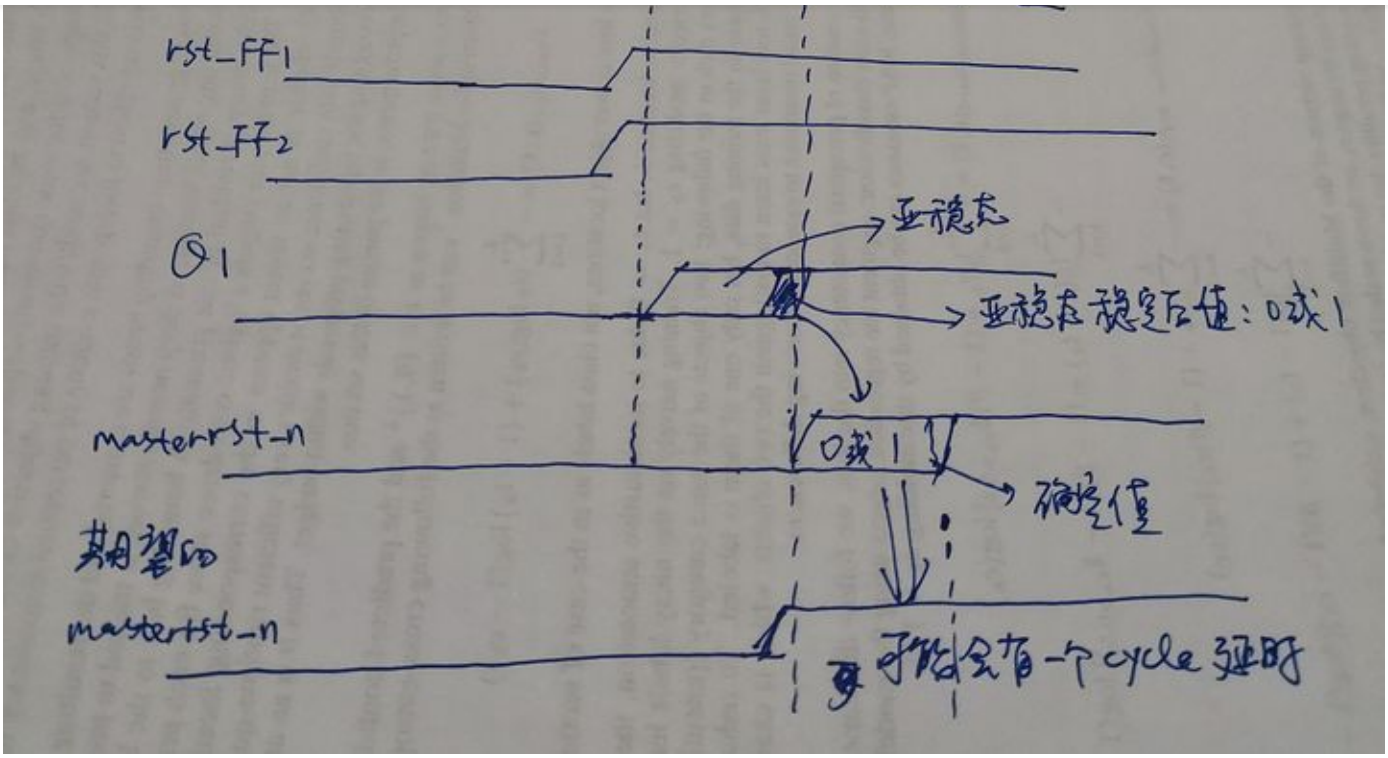


Reset Synchronizer block diagram

复位和解复位都是通过pad_rst_n来实现的，第一级FF的输入是拉成高(固定为“1”)，第二级的FF用来消除解复位时可能带来的的亚稳态。

为什么两级FF就不会出现亚稳态呢？第一级FF输入是1， 输出的reset值是0，而reset又是异步的，如果reset刚好在clock边沿附近，就会出现亚稳态。

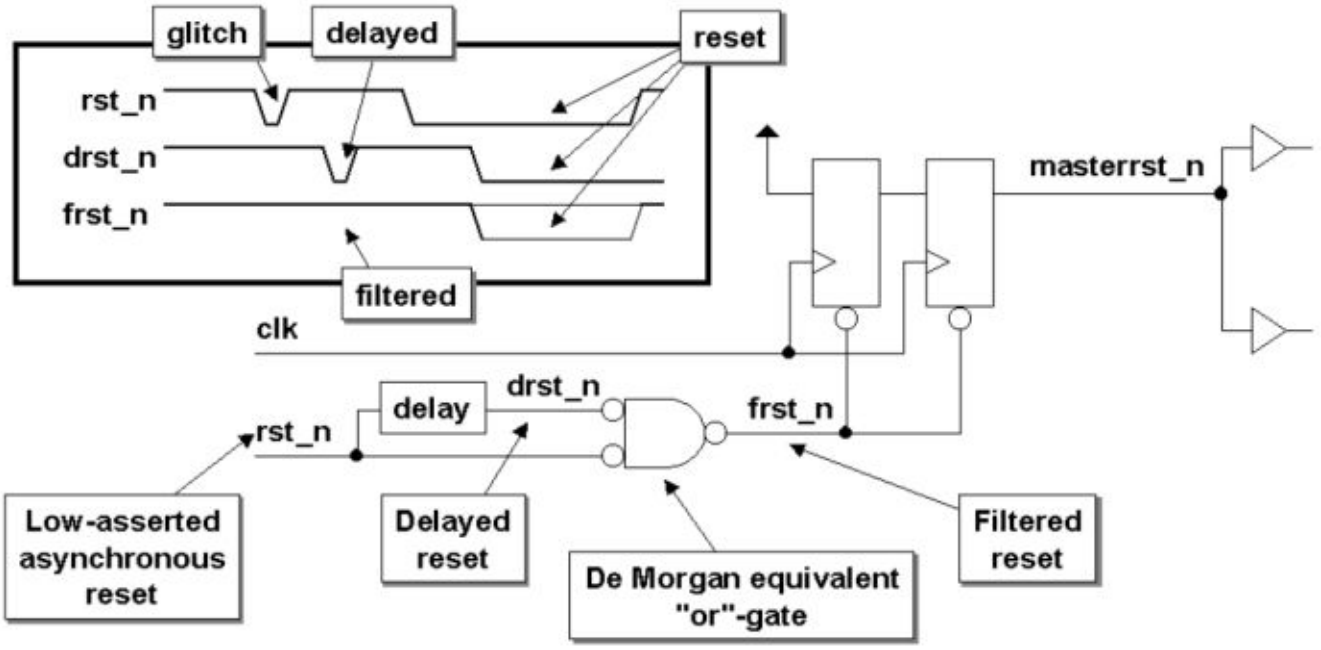
但是加上了第二级FF就不会出现亚稳态。这是因为第二级的FF的输入比第一个FF延时了一个cycle，这样第一个FF在解复位时候，即使有亚稳态，那么也只是影响下一个周期的Q1输出，下一个周期Q1可能为0也可能为1，但是当前Q1还是为低(0), 因为这时第一个FF还是复位状态。这样第二个FF当前周期的输入是0，复位输出也是0，所以能不能复位成功都不会改变输出为0，即当前周期不会产生亚稳态。下一个周期的Q1输出虽然可能是0也可能是1，但是已经稳定，是一个确定的值，所以第二个FF输出也是稳定的值。如下图所示：



（手画的，见谅）

5. 异步复位的抖动或毛刺

由于异步复位和时钟无关，任何一个毛刺都可以引起复位。这是一个reset源的问题。下面的电路可以过滤毛刺，主要原理是把输入源与上它的延时来消除毛刺。



Reset glitch filtering

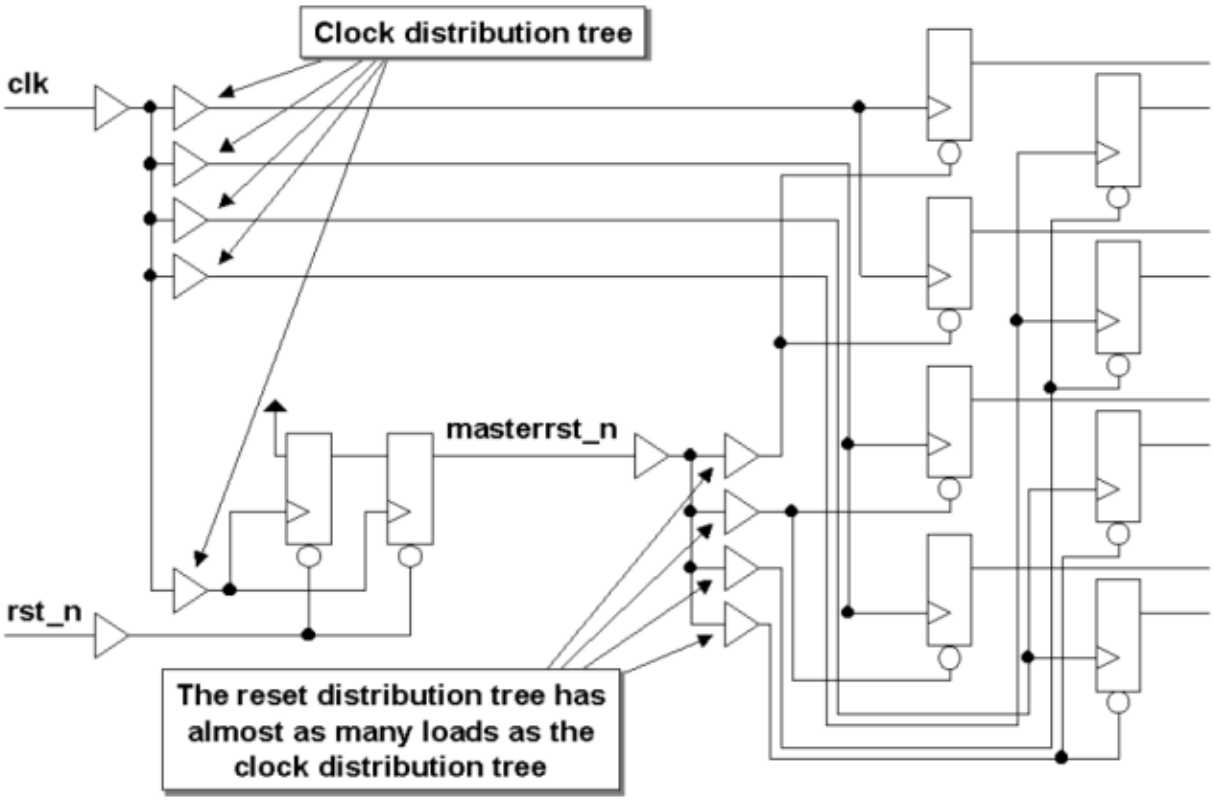
有的库里面包含Delay宏，有的没有；没有的话就需要手动增加delay或者插buffer，同时增加约束让delay不被综合掉。

同时rst_n 输入也必须是一个史密斯触发器pad，进一步消除抖动。

并不是所有的系统都需要增加防抖，要根据应用范围来判断。

6. 复位树

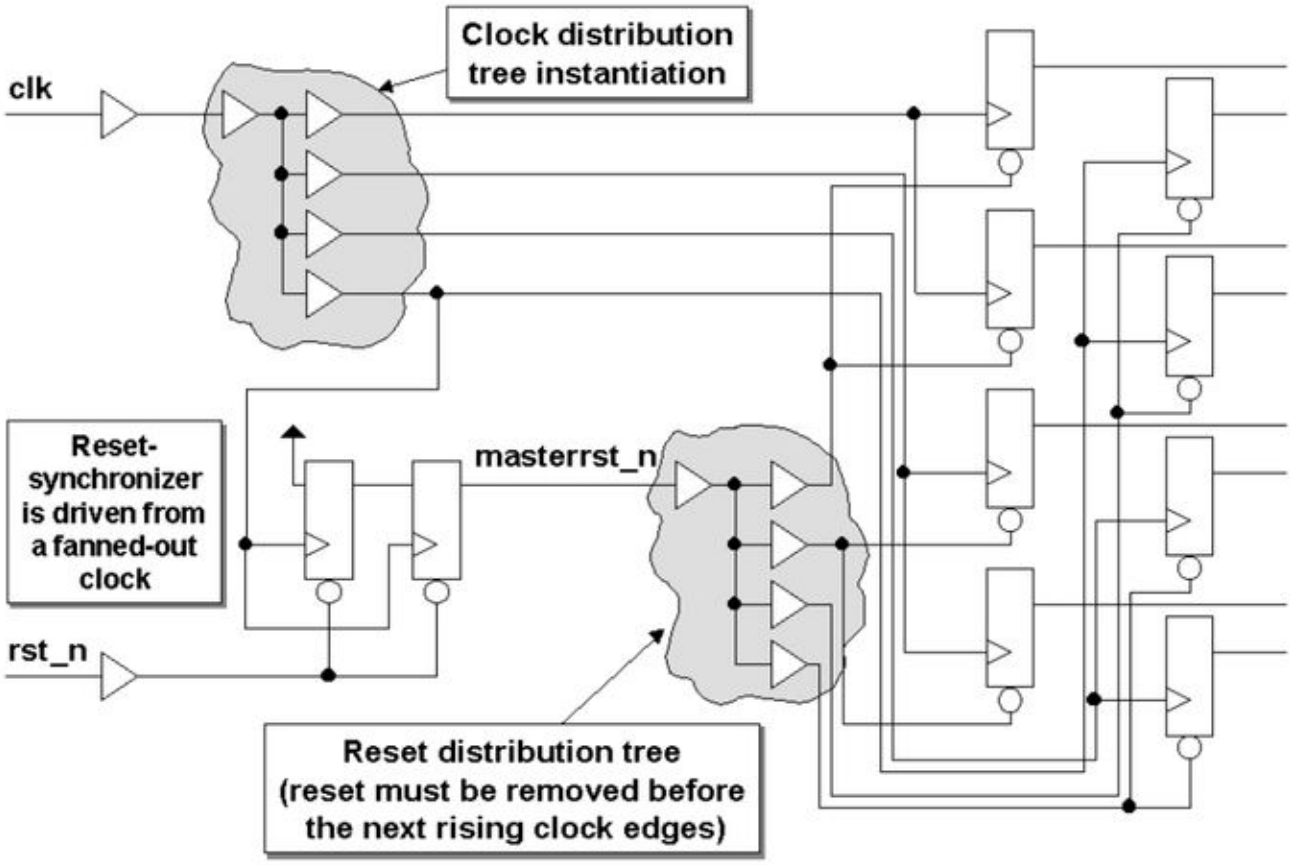
复位树和时钟树一样应该引起重视，因为典型的数字电路中，reset的负载和clock的负载数量一般是相当的。不管是同步复位还是异步复位，对复位树都是有时序要求的。一个典型的复位树如下图所示：



Reset distribution tree

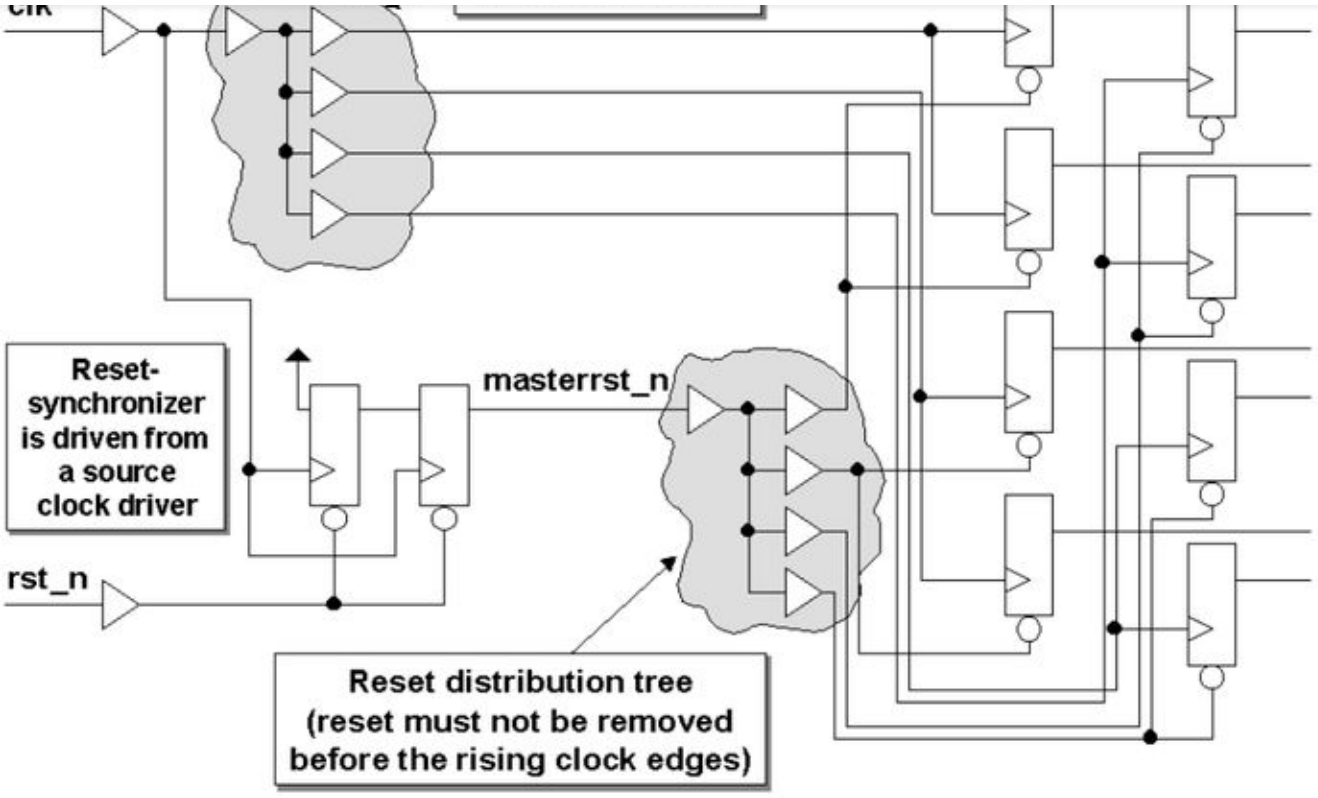
时钟分布树和重置分布树之间的一个重要区别是时钟树需要紧密平衡分布重置之间的偏差(skew)。与时钟信号不同，只要与复位信号相关的延迟足够短，允许在一个时钟周期内传播所有复位负载，并且仍然满足所有目标寄存器和触发器的恢复时间，那么复位信号之间的偏差(skew)就不是关

让时钟脉冲遍历时钟树、时钟复位驱动触发器，然后让复位遍历复位树，所有这些都在一个时钟周期内完成。这种情况如下图所示：



Reset tree driven from a delayed, buffered clock

为了能让reset到达所有逻辑的速度更快，更好的方式是用一个更早的clock来驱动reset ff，即用源时钟来驱动。必须进行Post layout timing分析，以保证复位同步器的FF不会出现setup/hold违例。通常情况下，两个tree之间详细的时序调整必须等到layout完成才能进行。电路结构如下图所示：



Reset synchronizer driven in parallel to the clock distribution tree

上述的复位树对同步复位树和异步复位树都是适用的。

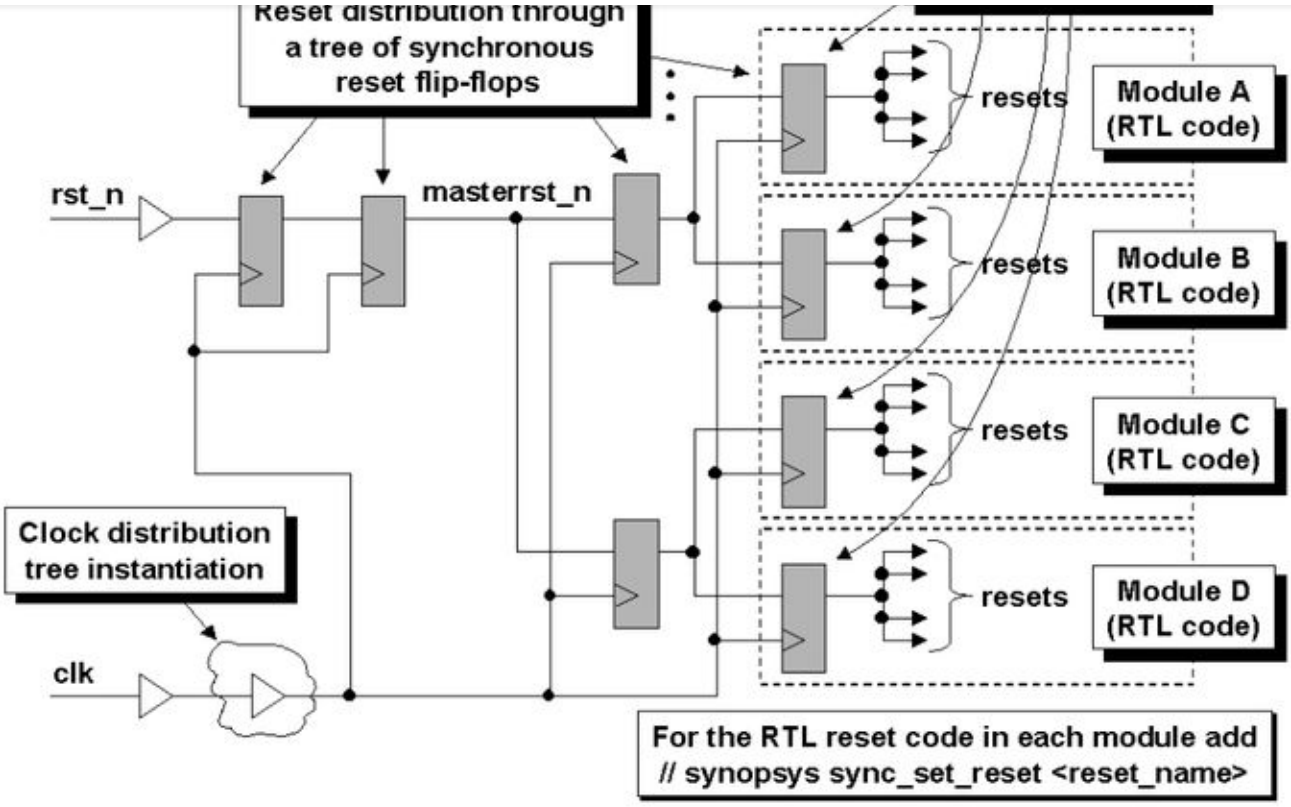
6.1 同步复位树

对同步复位，一种技术是通过插入FF来实现复位树。这样的好处是，reset不需要在一个时钟周期内到达所有的FF。所以需要几个时钟才能把整个设计复位掉。每个模块都需要包含如下代码：

```
input reset_raw;

// synopsys sync_set_reset "reset"
always @ (posedge clk) reset <= reset_raw;
```

同步复位时钟树如下所示：

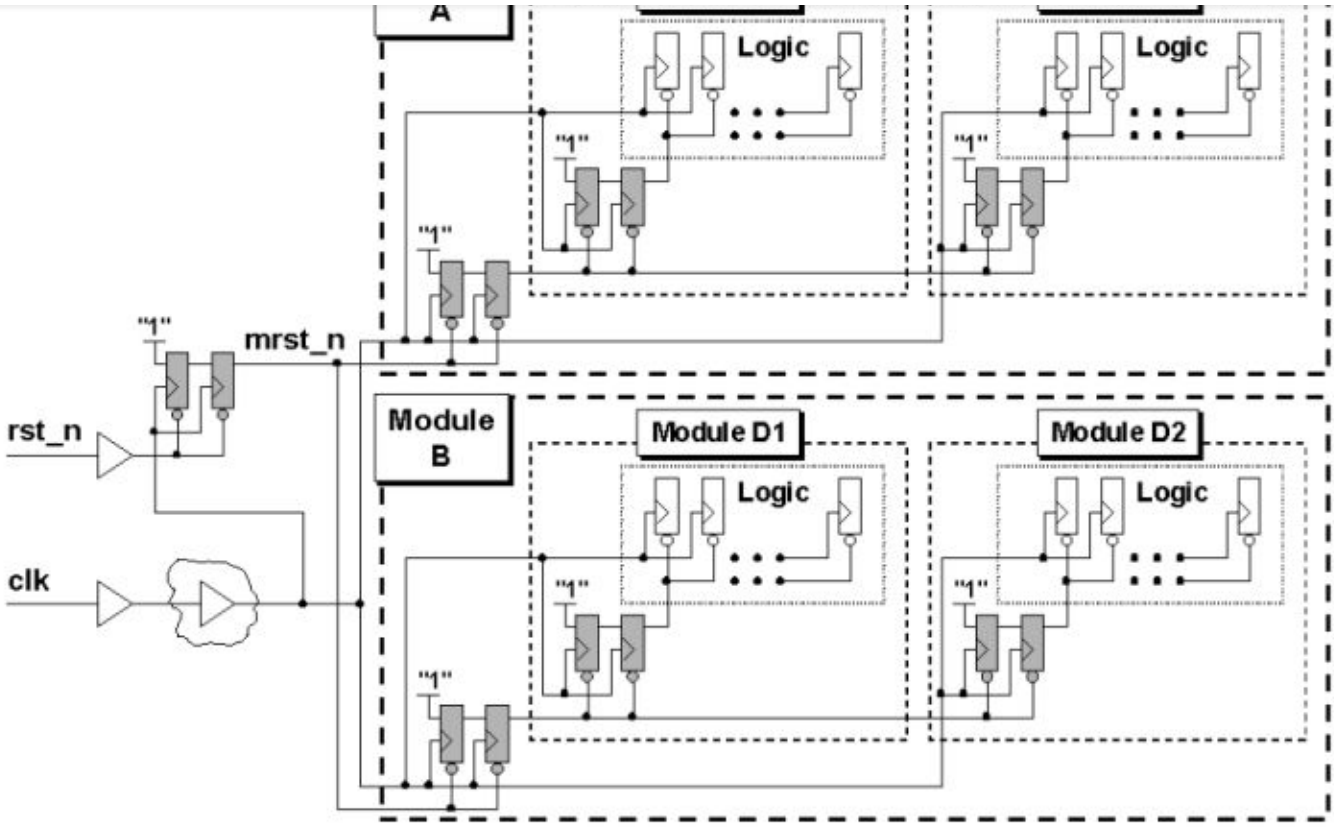


Synchronous reset distribution method using distributed synchronous flip-flops

采用这种技术，同步复位信号可以像其他数据信号一样处理，设计中每个模块的timing分析都很容易，复位树的每个阶段都有合理的扇出。

6.2 异步复位树

对异步复位，是通过复位同步器来完成复位树的建立。即每个层级都加上一个异步复位同步器。如下图所示：



Asynchronous reset distribution method using distributed reset synchronizers

异步复位树和同步复位树有点相似，但是这里的异步复位同步器是两级的FF。这种异步复位树，复位时可以所有的FF都同时复位，但是解复位必须要几个cycle才能完成。

这种结构的问题就是，不同层级解复位的时间点可能是不一样的。如果设计要求整个芯片在同一个cycle来解复位，那么就需要对复位同步器做平衡设计，保证到达每个复位终点是同一个时钟。同步时钟树也存在这个问题。

使用这种结构的好处是，不用等到P&R之后才去手动调整timing，完全可以交给综合工具(DC/PT)去插入buffer。

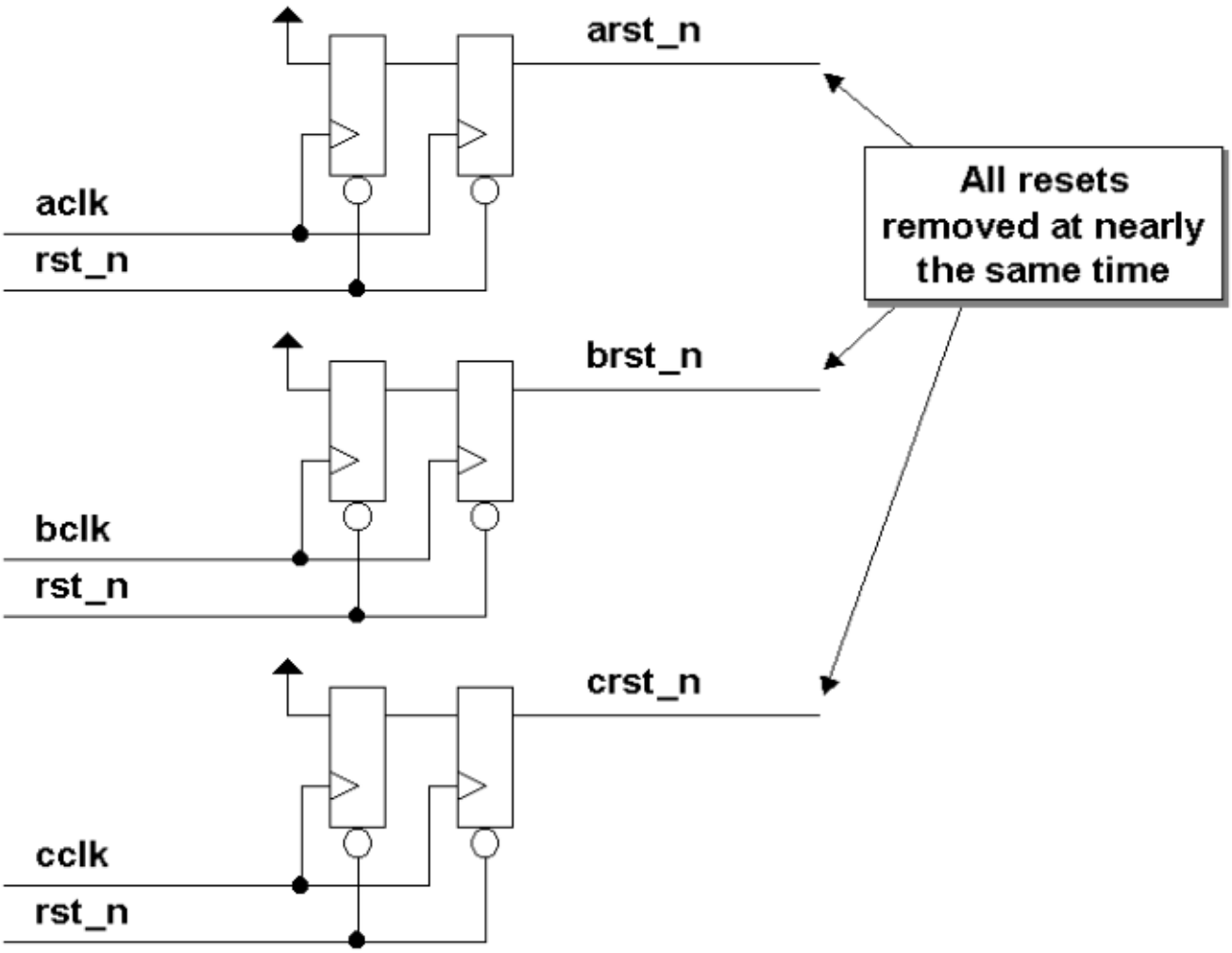
在使用异步重置时，至关重要的是，设计人员要在DC和PT中使用设置为适当设置的适当变量，以确保从复位同步触发器的q-output驱动的异步复位得到缓冲(如果需要的话)和timed。有关这些设置的详细信息可以在SolvNet文章#901989[43]中找到。文章指出，DC和PT都可以并且将时间按照本地时钟进行异步复位输入，如果设置了以下变量:

```
pt_shell> set timing_disable_recovery_removal_checks "false"
dc_shell> enable_recovery_removal_arcs "true"
```

这些设置应该是Synopsys的默认设置(只要确保它们是环境设置)。正确设置这些标志和使用分布式复位同步器后，就可以不用类似时钟树去构建缓冲复位树了。

7. 多时钟域复位

对于多时钟域的设计，每个时钟域必须有自己单独的复位同步器和分布式复位树。这样才能保证reset能满足不用时钟域的reset recovery time。如下图所示：

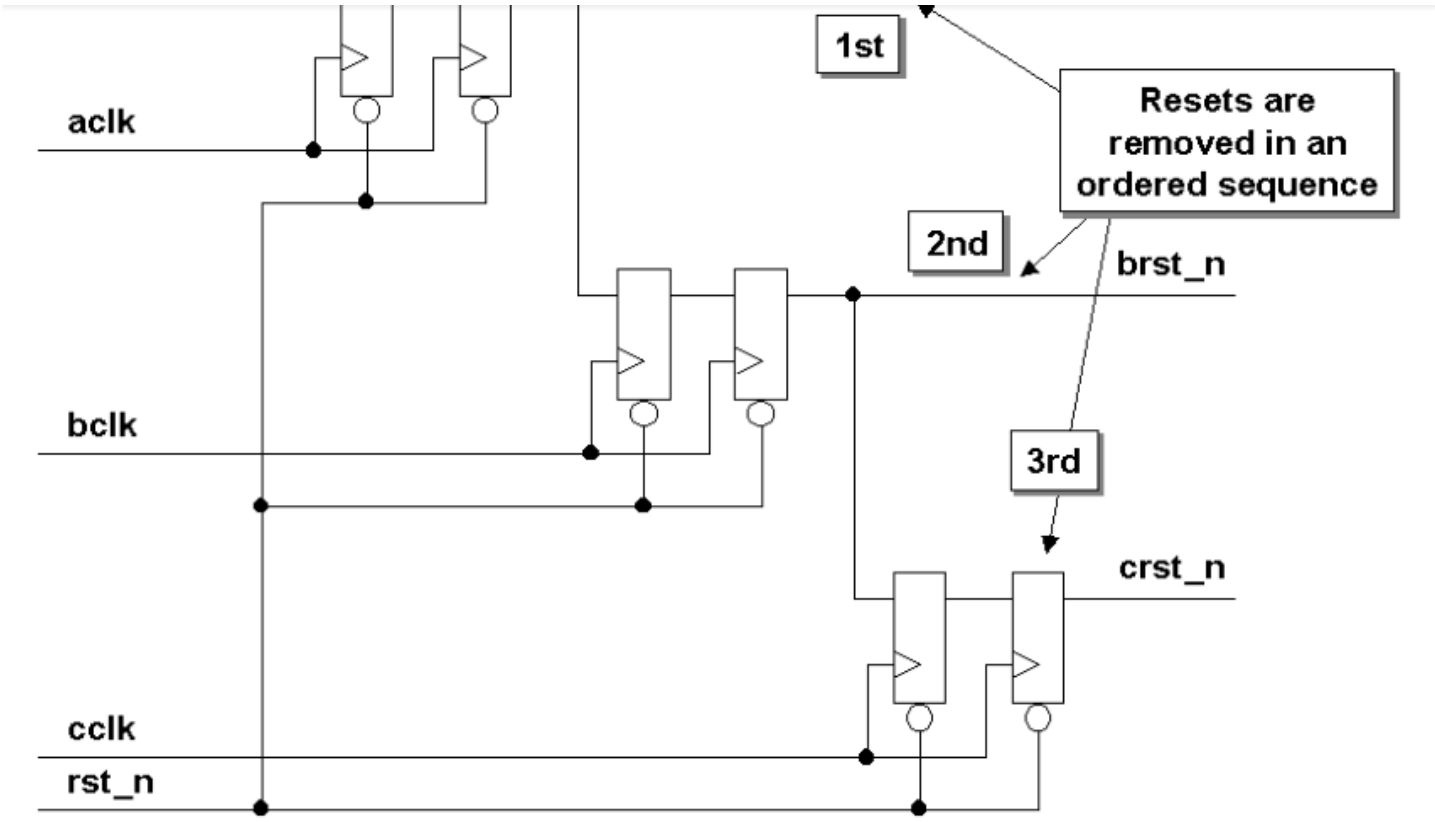


Multi-clock reset removal

对于多个时钟域的解复位顺序有两种情况：

1）大部分多时钟域的设计，解复位的顺序并没有特殊要求。即当前时钟域解复位的时间点，在另一个时钟域的精确时间点并不重要。通常来说，跨时钟域的设计，本身就会带来延时的不确定行。这种情况下，上面的单独建立不同时钟域的复位结构就足够了。

2) 有些多时钟设计，复位解除必须按顺序进行。这种设计，可以使用优先级的复位结构如下所示：



Multi-clock ordered reset removal

这种结构，除了最高优先级的同步器输入是 tied 1， 其他的输入都是上一优先级的输出。

8. 异步复位的DFT

在做DFT的时候，如果异步复位信号不能直接被I/O引脚驱动，就必须将异步复位信号和后面的的被驱动电路断开，用来保证DFT扫描和测试能够正确进行。两个同步复位触发器不应该包含在扫描链种，需要手动测试。

9. 后记

复位是芯片设计的一个重要主题，只有深入理解了为什么要复位，同步复位和异步复位的优缺点，才能深入理解工程设计中的各种解决方法。

赠人玫瑰，手有余香。原创不易，如果你有所得，麻烦花一秒时间帮我“**点赞**”吧，谢谢！

微信公众号：芯片设计进阶之路

参考资料：

SOC中的复位电路。blog.csdn.net/l47109484...

Clifford E. Cummings，"Asynchronous & Synchronous Reset Design Techniques"

编辑于 2020-05-03

「桃花仙人种桃树，又摘桃花换酒钱」

赞赏

2 人已赞赏

芯片设计

文章被以下专栏收录

芯片设计进阶之路
通向芯片设计专家的征途，我们一起前行

关注专栏

推荐阅读

4 条评论 ⇌ 切换为时间排序

写下你的评论...

空白的贝塔君 2020-04-29

写得太好了，受益匪浅

赞

米果 2020-05-18

您好，请问这些配图来自哪本书

赞

大郎不想吃药 2020-06-22

异步复位同步释放

赞

klmcu 2020-09-08

如何联系您?

赞