**Xilinx FPGA复位逻辑处理小结**

[**https://blog.csdn.net/shanekong/article/details/44834381?utm\_medium=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.pc\_relevant\_is\_cache&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.pc\_relevant\_is\_cache**](https://blog.csdn.net/shanekong/article/details/44834381?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.pc_relevant_is_cache&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.pc_relevant_is_cache)

1. 为什么要复位呢？

（1）FPGA上电的时候对设计进行初始化；

（2）使用一个外部管脚来实现全局复位，复位信号将所有存储单元设置为一个已知的状态，这个全局复位管脚与任何其他的输入管脚没有什么差别，经常以异步方式作用于FPGA。因此，设计人员可以在FPGA内部采用异步或者同步的方式来复位他们的设计。

2. 复位是针对存储单元—触发器（Flip-Flops）

Xilinx7系列架构的FPGA每一个Slice里面包含8个寄存器，并且所有的这些寄存器都是D触发器，这些触发器共享控制集。控制集包括输入时钟信号（CLK），高电平有效使能信号（CE）和高电平有效SR信号。SR在触发器里用作同步置位/复位（Set/Reset）或者异步预置位/清零（Preset/Clear）。

（1）同步复位与异步复位

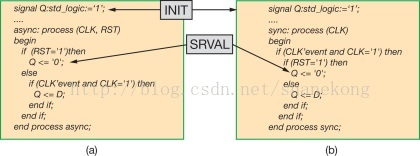
当复位信号出现在敏感信号列表中时，触发器将采用异步复位的方式。触发器的SR端口被配置为预置位/清零（Preset/Clear）端口，用FDCE/FDPE触发器来表示。一旦RST= ‘1’条件满足，触发器将**立即输出SRVAL属性。**

在同步复位的情形，触发器的SR端口被配置为**置位/复位**Set/Reset）端口，用FDSE/FDRE触发器来表示。一旦RST=‘1’条件满足，触发器将**在下一个时钟上升沿时输出SRVAL属性。**

（2）INIT 与SRVAL属性

INIT属性用来初始化触发器，INIT属性值将在配置过程中加载到触发器。

Xilinx FPGA 的触发器可以支持同步或者异步的复位/置位控制，但是关联的触发器一次只能实现Set/Reset/Preset/Clear中的一个。如果RTL代码中使用Set/Reset/Preset/Clear 里不止一种时，将导致一些条件下使用触发器的SR端口，而另一些条件下使用Fabric Logic，则将消耗更多FPGA资源。一般来说，最好避免使用超过一种Set/Reset/Preset/Clear。

****

3. 除了考虑复位类型，还需要考虑复位信号与时钟是否同步的问题

只要全局复位脉冲信号持续的时间够长，FPGA所有的触发器都将进入复位状态。然而，复位信号的释放必须满足触发器的时序要求，以确保触发器能够正确的从复位状态恢复到正常状态。不满足这些时序要求，将导致触发器进入亚稳态。并且为确保子系统（如状态机、计数器）正常工作，所有触发器必须在相同的时钟沿退出复位状态。

如果同一状态机不同的位在不同的时钟沿退出复位状态，则状态机将转移到一个非法的状态，这就说明了释放复位信号与时钟同步的必要性。

4. 两种复位机制

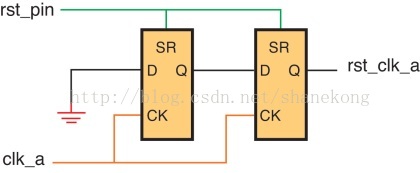
（1）同步复位机制

可以使用一种标准的亚稳态消除电路（两个触发器背靠背相连）来将全局复位同步到特定的时钟域。同步化的复位信号可以使用同步SR端口来初始化所有的存储单元。由于同步器和触发器在同一时钟域里被复位，标准的时钟周期约束包括了它们之间的路径的时间。在FPGA中每一个时钟域需要使用单独的同步器来为对应的时钟域产生同步化的全局复位信号。

***Tip 1:*** *When driving the synchronous SR port of Flip-Flops， every clock domain requires its own localized version of the global reset， synchronized to that domain*

（2）异步复位机制

设计中的存储单元可能需要使用触发器的异步SR port 来异步复位初始化。尽管这些存储单元使用了异步的SR端口，复位信号的释放必须与时钟信号同步。这与触发器的复位恢复时序要求有关，类似于异步SR释放信号边沿对时钟上升沿要满足建立时间的要求。不能满足这些时序要求时将导致触发器进入一个亚稳态，同步的子系统进入异常的状态。

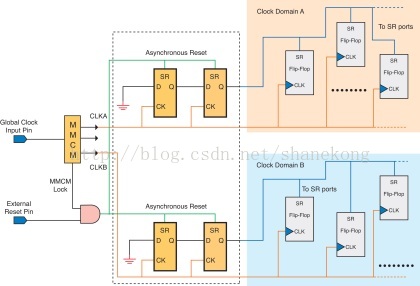


复位桥电路提供了异步复位（Assert）同步释放（Release）机制。在这个电路中，假定两个触发器的SR port 有一个异步设置的值，SRVAL= 1。每一个时钟域需要使用复位桥电路来产生它自己的本地化（localized）全局复位信号。***FPGA*的每一个时钟域需要一个单独的、同步化的全局复位信号。它可以由单独的复位桥电路来产生。**

图中电路假定：复位桥电路和相关逻辑的时钟源*clk\_a*是稳定的、无错的。在*FPGA*中，时钟可以来自芯片外的时钟源（最好通过一个*clock-capable*管脚），或者采用*MMCM*或者*PLL*内部产生。任何用于产生时钟信号的*MMCM or PLL*在复位后需要校准。因此，你可能需要在全局复位路径插入一些额外的逻辑来稳定时钟信号。

*Tip 2: A reset bridge circuit provides a safe mechanism to deassert an asynchronous reset synchronously. Every clock domain requires its own localized version of the global reset with the use of a reset bridge circuit.*

***Tip 3:****Ensure that the clock the MMCM or PLL has generated is stable and locked before deasserting the global reset to the FPGA*



FPGA内实现异步复位逻辑的典型框图

5. 为什么复位信号一般都使用高电平有效呢？

Xilinx 寄存器的SR控制端是高电平有效的。如果RTL代码采用了低电平有效的复位模式，综合器将在复位信号驱动寄存器SR控制端之前插入一个反相器（Interver）。你必须使用一个查找表（Look Up Table, LUT）来实现反向器，以利用LUT的输入端口。低电平有效的控制信号带来额外的逻辑可能拉长了执行时间（Runtime），将导致更低的FPGA资源利用率，也将影响时序和功耗。

 尽可能在 *HDL* 代码或者实例化的模块中使用高电平有效的控制信号。如果在设计中，你不能够改变这些控制信号的极性，你需要在代码的顶层文件反转这些控制信号。采用这种方式描述电路的话，这些反向器将被吸收到 *I/O* 逻辑中，而不需要使用额外的 *FPGA* 逻辑和路径。

*Tip 4: Active-high resets enable better device utilization and improve performance.*

6. 到底如何用呢？

应用场合：多个时钟域情况下的复位，针对每一个时钟域都得有对应的复位逻辑。参见tri\_mode\_mac 参考例子中的实现方法。

7. 复位桥电路到底怎么实现呢？

`timescale 1ps/1ps

module tri\_mode\_mac\_reset\_sync # (

**parameter INITIAL\_VAL = 2'b11**

)

(

   input       reset\_in，

   input       clk，

   input       enable，

   output      reset\_out

);

wire  reset\_stage1;

wire  reset\_stage2;

  (\* ASYNC\_REG = "TRUE"， RLOC = "X0Y0"，  SHREG\_EXTRACT = "NO"， INIT = "1" \*)

  FDPE #(

   .INIT (INITIAL\_VAL[0])

  ) reset\_sync1 (

  .C  (clk)，

  .CE (enable)，

  .PRE(reset\_in)，

  .D  (1'b0)，

  .Q  (reset\_stage1)

  );

  (\* ASYNC\_REG = "TRUE"， RLOC = "X0Y0"，  SHREG\_EXTRACT = "NO"， INIT = "1" \*)

  FDPE #(

   .INIT (INITIAL\_VAL[1])

  ) reset\_sync2 (

  .C  (clk)，

  .CE (enable)，

  .PRE(reset\_in)，

  .D  (reset\_stage1)，

  .Q  (reset\_stage2)

  );

assign reset\_out = reset\_stage2;

endmodule

参考文档：How do I reset FPGA. Srikanth Erusalagandi, Xilinx.

# [你真的会Xilinx FPGA的复位吗？](https://www.cnblogs.com/ninghechuan/p/9969879.html)

# https://www.cnblogs.com/ninghechuan/p/9969879.html

# Get Smart About Reset: Think Local, Not Global.

　　对于复位信号的处理，为了方便我们习惯上采用全局复位，博主在很长一段时间内都是将复位信号作为一个I/O口，通过拨码开关硬件复位。后来也看了一些书籍，采用异步复位同步释放，对自己设计的改进。

　　不过自从我研读了Xilinx的White Paper后，让我对复位有了更新的认识。

　　One of the commandments of digital design states, "Thou shalt have a master reset for all Flip-Flops so that the test engineer will love you, and your simulations will not remain undefined for time eternal."

　　这句是用来装逼的。使用全局复位有利于我们仿真，所有的寄存器都是有初始值的，也可以在任意时刻让你的寄存器恢复初值，所以验证工程师很喜欢这样的设计，但是Xilinx的建议是尽量避免使用全局复位，这是为什么呢。

## 全局复位并不是好的处理方式

　　我们习惯上通常使用的复位有三种。

　　硬件开关：复位信号接一个拨码开关。

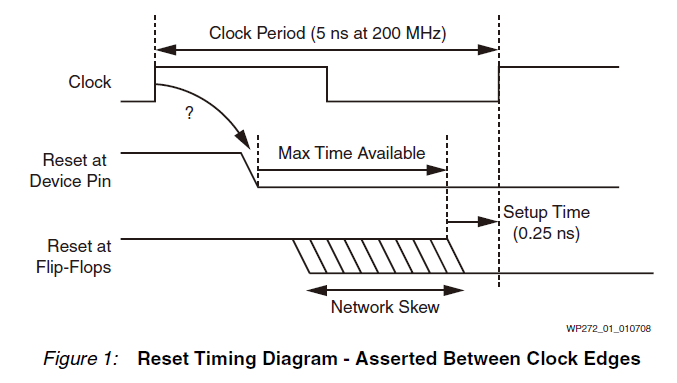
　　电源芯片：上电时候电源芯片产生，可以长时间维持，直到稳定。

　　控制芯片：控制芯片产生复位脉冲。

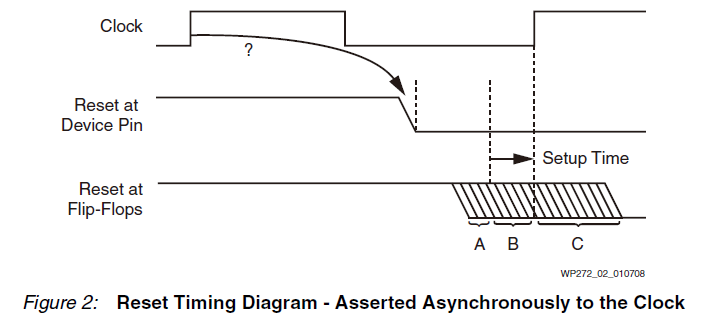
　　这些复位信号和FPGA内部信号的变化比起来是比较慢的。复位按钮最快也会到达毫秒级别，而FPGA内部信号都是纳秒级别的变化。全局复位的周期远大于系统时钟的周期，是完全可以保证所有的触发器被成功复位的。

　　但随着系统的频率越来越快，全局复位信号的释放形成一个高扇出的网络。

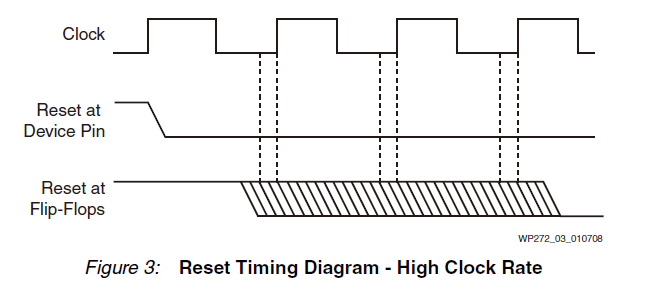
　　Fan-out即扇出，模块直接调用的下级模块的个数，如果这个数值过大的话，在FPGA直接表现为net delay较大，不利于时序收敛。因此，在写代码时应尽量避免高扇出的情况。



 　　如图，全局复位的释放需要传输到不同的触发器。每一个触发器需要被释放复位，但是随着时钟频率的提高，加上复位路径网络的延迟，而且全局复位还是一个高扇出的网络，所以这对系统的时序是一个大挑战。



    通常情况下，复位信号的异步释放，没有办法保证所有的触发器都能在同一时间内释放。触发器在A时刻接收到复位信号释放是最稳定的，在下一个时钟沿来临被激活，但是如果在C时刻接收到复位信号释放无法被激活，在B时刻收到复位信号释放，则会引起亚稳态。

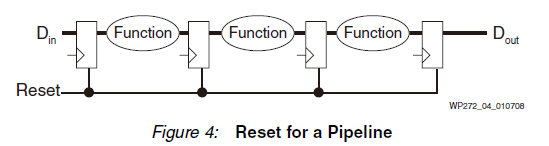


　　随着系统时钟频率的提高，并不是所有的触发器都能在同一个时刻从复位状态被释放。

## 复位真的有那么重要吗？

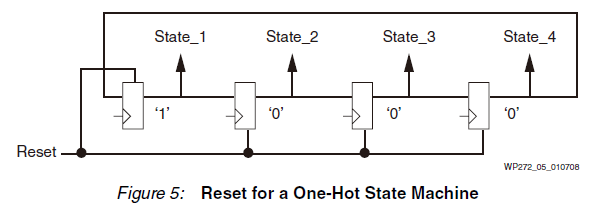
　　白皮书上又说了，好的消息是99.9%的情况下，全局复位的异步释放并不会出现问题。所以大多数电路都可以正常工作。但是，如果你有了第一次就不能工作的电路，那你就是遇到那0.01%的情况，很不幸你的复位信号被在错误的时刻重置。（哈哈，皮）

　　在一些情况下，复位释放的时间并不重要。



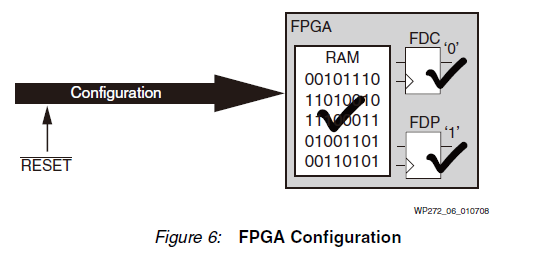
　　当你的数据采用流水线操作的时候，复位释放的时间并不重要，因为不管你流水线后面的触发器复位释放后是否出错，只需要一些周期后，整个流水线就又会正常工作了。白皮书上还说，这样的复位也是没有意义的。

　　但是有一些情况下，复位的释放后是很重要的。



　　比如独热码状态机。如果表示独热码状态的第一个触发器比第二个触发器早释放了一个时钟周期，那状态机的状态机会跳转到一个无效的状态。如果所有的表示独热码的寄存器无法在同一个周期内被释放，那状态机肯定会跳转到一个无效的状态。还有一些例子，请读者自己看白皮书。

## 解决99.99%的情况

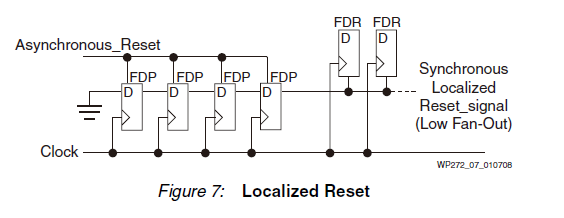


　　其实当Xilinx FPGA配置或重新配置时，所有的单元都会被初始化。白皮书称为master reset，因为这可比你复位一些D触发器要强得多，它甚至初始化了片内RAM。

　　Xilinx的器件也有嵌入处理的系列，软核或硬核。在程序执行第一条指令前，程序和数据区域已经定义好了。有了上电复位，还用专门消耗逻辑资源去复位触发器是没有意义的。

## 解决0.01%的方法

　　当然，白皮书也给出了处理复位的建议和方法。



　　Think Local, Not Global。异步复位同步释放的方法。用内部定义复位信号的方法来复位触发器，而不是全都直接使用全局复位信号。当进行复位操作时，所有的触发器被预设为1。如上图，移位寄存器的最后一个触发器去操作模块内部定义的复位网络。当复位信号释放时，移位寄存器经过移位，当最后一个触发器由高电平变为低电平时，对本地的复位网络进行复位操作。也就是异步复位同步释放。

## 复位消耗的资源比你想象的要多

　　复位网络占用大量布线资源。

　　提高了布局布线时间。

　　使用器件的逻辑资源。

　　会使你的设计变得更大。

　　占用更多的逻辑资源肯定会影响你的性能。

　　具体请参阅白皮书。

## 总结

　　并不是所有部分的设计都需要复位，所以设计者在设计过程中应该准确判断需要被复位的部分，从而采用异步复位同步释放的方法进行对复位的严格处理。

　　当我们在设计每一个部分的时候，都要内心问问自己，这个bit需要被复位吗？

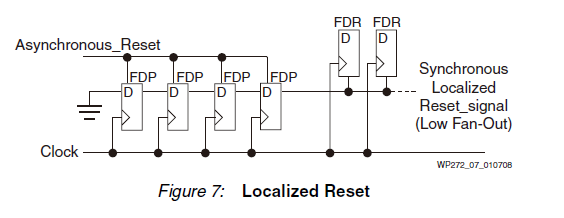
　　那么怎样处理Xilinx FPGA中的复位呢？Xilinx的工程师也给出了解释和方法。

<https://www.eetimes.com/document.asp?doc_id=1278998>

<http://xilinx.eetrend.com/d6-xilinx/article/2013-07/4343.html>

　　简单总结就是，Xilinx的FPGA 应该尽量避免全局复位，有些部分的设计都可以不用复位，必需要复位的设计采用同步高电平有效复位。

　　代码如下。设计中需要复位的部分，使用产生的sys\_rst信号进行同步复位，而且是同步高复位。再次贴出下图。



1 module Sys\_Rst(

2 input clk,

3 input rst,

4 output sys\_rst

5

6 );

7

8 reg rst\_r0;

9 reg rst\_r1;

10

11 always @(posedge clk or posedge rst)begin

12 if(rst)begin

13 rst\_r0 <= 1'b1;

14 rst\_r1 <= 1'b1;

15 end

16 else begin

17 rst\_r0 <= 1'b0;

18 rst\_r1 <= rst\_r0;

19 end

20 end

21

22 assign sys\_rst = rst\_r1;

23

24 endmodule

# Xilinx FPGA复位浅析

# https://blog.csdn.net/u010161493/article/details/52939224

# 1. 普通逻辑的复位

对于Altera的FPGA而言，因为里面的Flip-Flop只支持低电平有效的异步复位，所以推荐使用低电平有效的异步复位。对于Xilinx 7系列的FPGA而言，Flip-Flop支持高电平有效的异步复/置位和同步复位/置位。对普通逻辑设计，同步复位和异步复位没有区别，当然由于器件内部信号均为高电平有效，因此推荐使用高电平有效的控制信号，最好使用高电平有效的同步复位。输入复位信号的低电平有效在顶层放置反相器可以被吸收到IOB中。

先比较一下同步复位和异步复位的区别。首先是同步复位，同步的控制信号包括同步置位/复位和使能，可以被吸收到LUT中，目的是为了防止控制集不同的LUT不能被综合到同一个Slice中，这样虽然提高了了LUT的使用率，却降低了Slice的使用率，得不偿失。控制信号扇出不大于16的情况都可以尽可能地被吸收到同一个Slice的LUT中去，在Vivado中可以使用control\_set\_opt\_threshold进行设置。

## 同步复位

always @(posedge I\_sys\_clk)

begin

if(I\_rst)

begin

OUT1 <= 0;

end

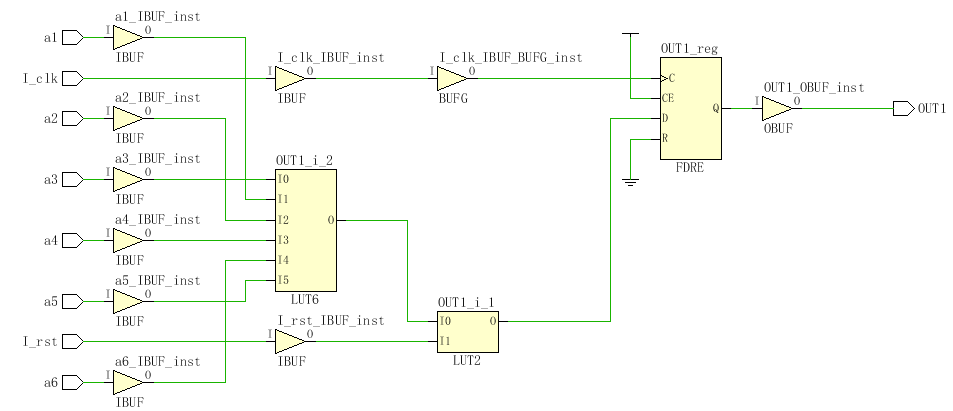
else

begin

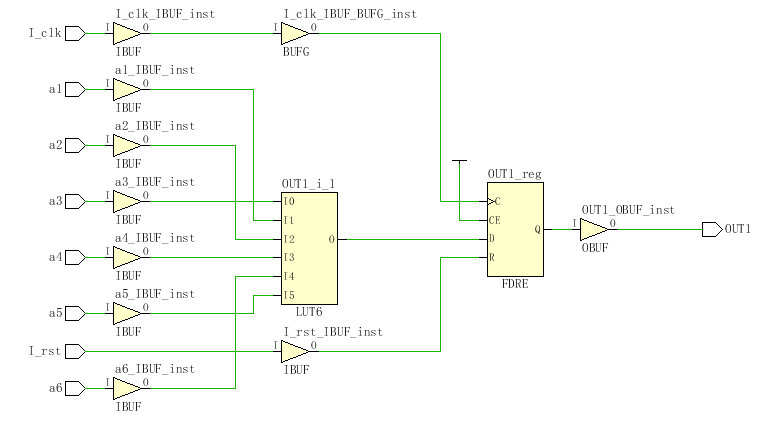
OUT1 <= &{a1,a2,a3,a4,a5,a6};

end

end



控制集优化为2时，可以看出复位信号中插入了LUT，但是由于在同一个Slice中，使用造成的延时并不显著。注意对于扇出大的信号比如全局复位来说，这种控制集优化是不起作用的。



控制集合优化为0时，可以看出不会使用多余的LUT逻辑。

## 异步复位

always @(posedge I\_sys\_clk or posedge I\_rst)

begin

if(I\_rst)

begin

OUT1 <= 0;

end

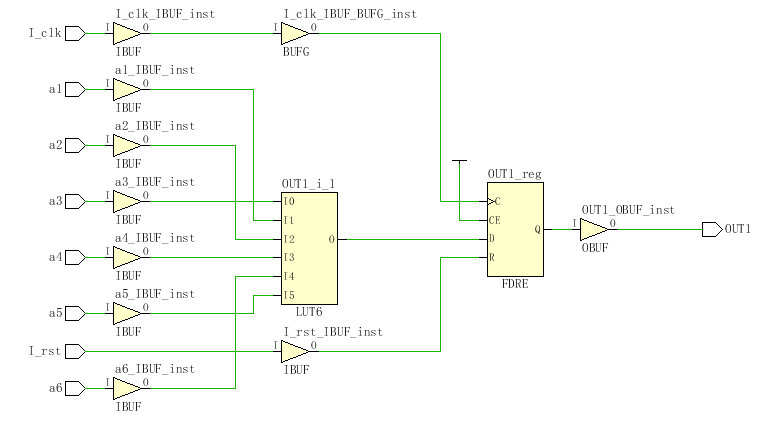
else

begin

OUT1 <= &{a1,a2,a3,a4,a5,a6};

end

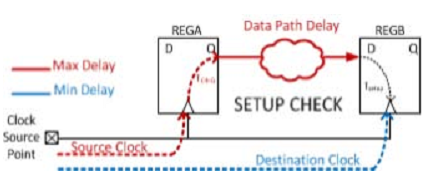
end



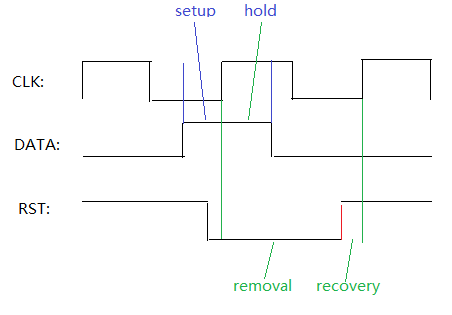
异步复位跟没有使用控制集综合出的结果相同，可以看出不会使用额外的LUT逻辑。

## 分析Recovery/Removal

异步复位有Recover time 和Remove time 的风险，也就是说recovery time和removal time都是检查异步信号（reset或preset或set）的释放沿，释放沿必须在时钟沿前面提前recovery time释放，或者在时钟沿后removal time之后释放。



顺便简单分析一下setup/hold, 对于同一个时钟，肯定满足data\_path > clk\_path, 而hold的要求很小，所以肯定满足。当data\_path过长时，setup可能不满足。



假如不作同步的复位信号，那么recovery/removal都有可能不满足。

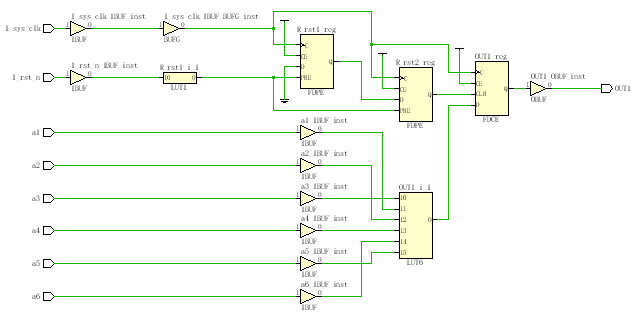
再考虑被本地时钟同步后的recovery/removal，同理复位信号对removal的时间要求很小，同步后的复位信号的延迟就能满足removal。由于设计中的全局复位信号一般有很大的扇出，布线的延时会很严重，因此recovery的要求变得严格。最好采用全局布线资源如BUFG，尽量用局部复位或者避免使用复位。

因此不管是同步复位还是异步复位，都要使用同步后的复位信号。复位信号进来后先用本地时钟打两拍。在多个时钟域的时候注意是本地时钟，不是全局时钟。并在RTL中注明是异步寄存器，使得综合工具把简单同步器综合在同一个Slice中以减少延时，提高MTBF。

*(\* ASYNC\_REG = "TRUE" \*)* *(\* keep = "true" \*)* reg system\_reset\_r;

*(\* ASYNC\_REG = "TRUE" \*)* *(\* keep = "true" \*)* reg system\_reset\_r2;

综合的结果如图，同步化的异步复位。

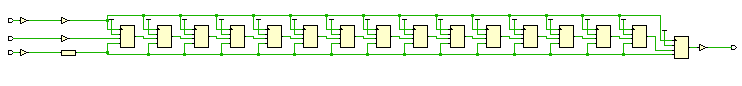


# 2. 特殊资源的复位

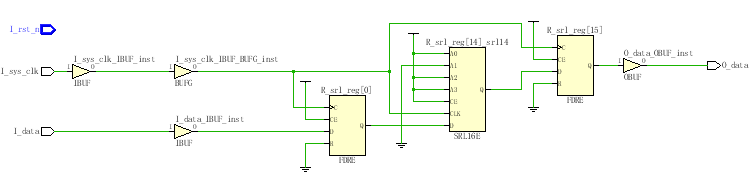
## 使用Xilinx 原语SRL16、SRL32、LUTRAM

由于是利用LUT完成的上述功能，没有复位接口。只能依靠 GSR方法来完成，不能使用复位。因此，在为以上这些资源编写代码时，应注意避免在编码中使用复位，以此来保证综合工具综合出相应的电路。

使用复位信号



没有使用复位信号，可以看出综合工具自动把输入和输出寄存化，以满足更好的时序。



## 使用DSP48E1或BRAM

使用同步复位可以允许综合工具使用 DSP48E1或BRAM等专用资源内部的寄存器。这样能够改善设计中相应部分的器件总体使用率和性能，同时降低总体功耗。

下面以18X18的乘法器为例

module multi\_18x18(

input I\_rst,

input I\_sys\_clk,

input [17:0]I\_data1,

input [17:0]I\_data2,

output reg [35:0] O\_data

);

reg[17:0] R\_data1, R\_data2;

always @(posedge I\_sys\_clk)

begin

if(I\_rst)

begin

R\_data1 <= 18'd0;

R\_data2 <= 18'd0;

end

else

begin

R\_data1 <= I\_data1;

R\_data2 <= I\_data2;

end

end

always @(posedge I\_sys\_clk)

begin

if(I\_rst)

begin

O\_data <= 36'd0;

end

else

begin

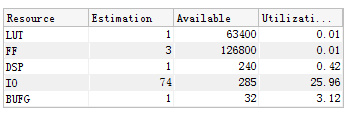
O\_data <= R\_data1\*R\_data2;

end

end

endmodule

同步复位



异步复位，多使用了35个FF和18个LUT

