FPGA时序约束

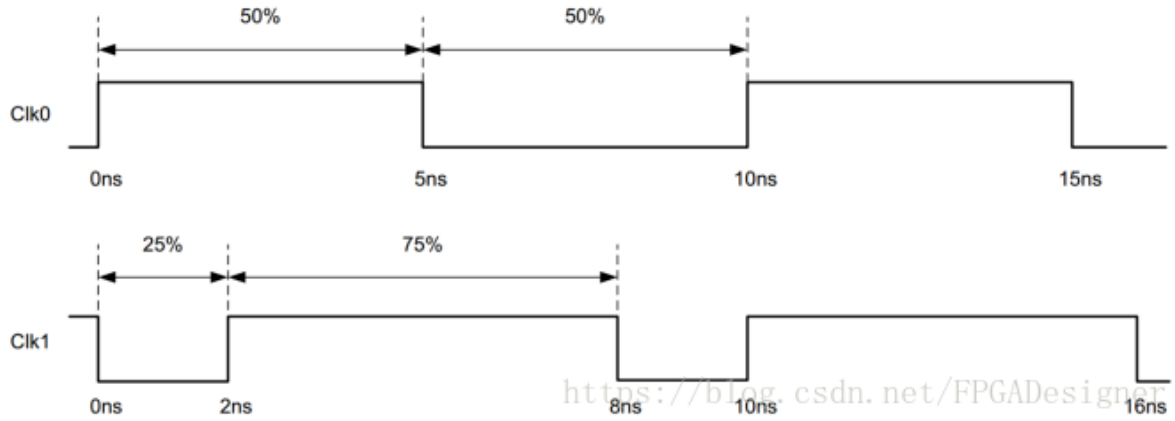
1.时钟约束

时钟的基础知识

数字设计中，“时钟”表示在寄存器间可靠地传输数据所需的参考时间。Vivado的时序引擎通过时钟特征来计算时序路径需求，通过计算裕量（Slack）的方法报告设计时序空余。时钟必须有合适的定义，包含如下特性：

- 定义时钟树的驱动管脚或端口，通常称作根或源点。
- 通过周期和波形属性来描述时钟边沿。
- 周期（period）以ns为单位进行设定，与波形重复率相关。
- 波形（waveform）以列表的形式给出，表中包含上升沿和下降沿在周期中的绝对时间，以ns为单位。

如下图给出了两个时钟Clk0: period=10, waveform={0 5}、Clk1: period=8, waveform = {2 8}。



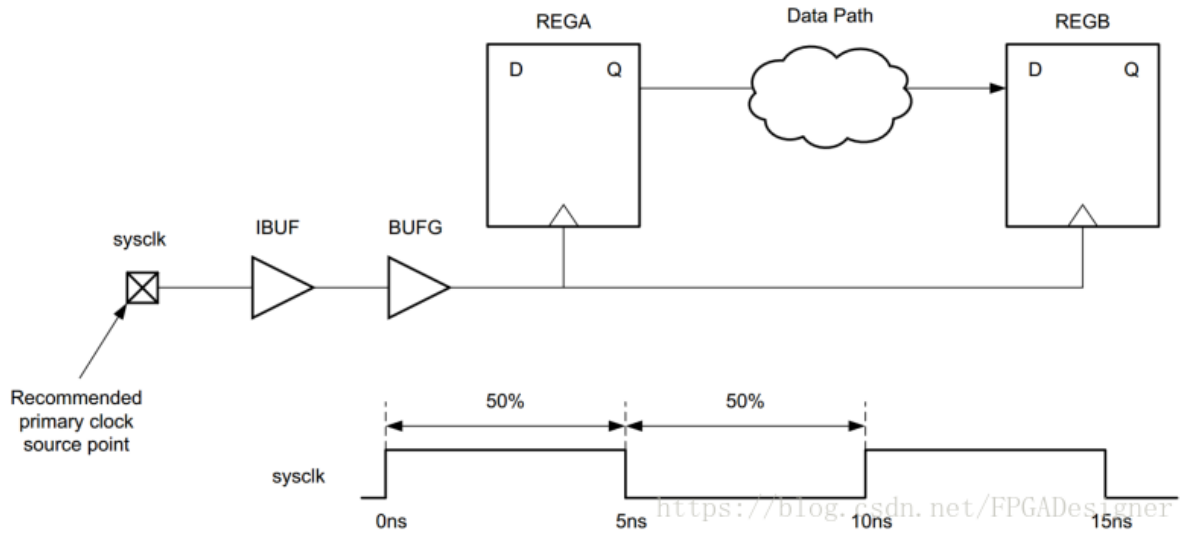
上述给出的只是时钟的理想特征。当时钟进入了FPGA器件，通过时钟树传递时，时钟边沿会有延时，通常称作时钟网络延迟；噪声或硬件表现会导致时钟随时可能发生变化，通常称作时钟不确定性，包括时钟抖动、相位错位等等。Vivado在时序分析时会考虑这些非理想因素以得到精确的时序裕量。

Xilinx FPGA器件内部有专用的硬件资源，支持大量设计时钟的使用。通常板子上有一个外部组件（如有源晶振）产生时钟信号，通过输入端口进入器件内部。外部时钟可以通过MMCM、PLL、BUFR等特殊原语生成其它时钟，也可以由LUT、寄存器等常规单元进行转换（通常称作门控时钟）。本文将讲述如何根据应用情况定义时钟。

主时钟Primary Clock

主时钟通常由两个来源：（1）.板级时钟通过输入端口进入设计；（2）.GT收发器的输出管脚（如恢复时钟）。主时钟必须与一个网表对象相连，该对象代表了所有时钟边沿的开始点，并且在时钟树中向下传递。也可以说，主时钟的源点定义了0时刻，Vivado靠此来计算时钟延迟和不确定性。

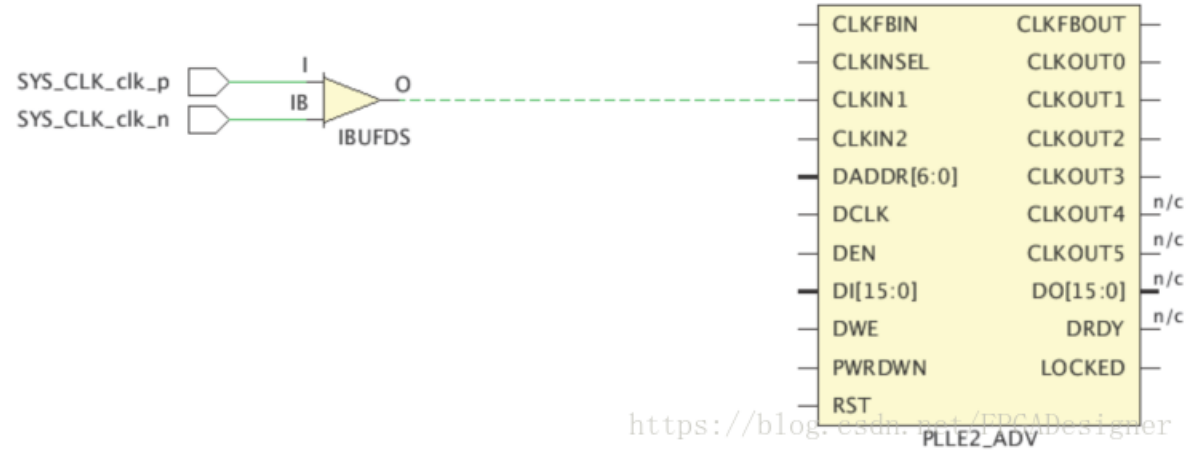
主时钟只能通过create_clock命令来定义，且必须放在约束的开始，这是因为其它时序约束几乎都要参考主时钟。下面给出两个主时钟的例子。第一个例子如下图所示，采用单端时钟输入：



板级时钟通过sysclk端口进入FPGA，通过一个输入缓冲器和一个时钟缓冲器后到达寄存器。使用如下命令定义：

```
create_clock -period 10 [get_ports sysclk]  #10ns周期，50%占空比，无相移
create_clock -name devclk -period 10 -wavefor {2.5 5} [get_ports sysclk]  #板级时钟名称devclk，10ns周期，25%占空比，90°相移
```

第二个例子如下图所示，采用差分时钟输入，这也是高速时钟的输入方式：



上图中差分时钟驱动一个PLL，定义主时钟时必须只创建差分缓冲器的正极输入。如果同时创建了正极、负极输入，将会导致错误的CDC路径。如“create_clock -name sysclk -period 3.33 [get_ports SYS_CLK_clk_p]”。

虚拟时钟Virtual Clock

这种类型的时钟对于初学者来说用的可能很少，虚拟时钟通常用于设定输入和输出的延迟约束。之所以称为“虚拟”，是因为这种时钟在物理上没有与设计中的任何网表对象相连。定义时使用create_clock命令，但无需指定源对象。在下列情况需要用到虚拟时钟：

所有设计时钟都不是外部器件I/O的参考时钟。
FPGA的I/O路径与一个内部生成的时钟相关，但是该时钟不能合适地通过对板级时钟计时来生成（如两个周期的比不是整数）。
希望为与I/O延迟约束相关的时钟设定不同的抖动和延迟，但是不希望修改内部时钟的特征。
比如时钟clk_virt的周期为10ns，且不与任何网表对象相连，可以这样定义“create_clock -name clk_virt –period 10”，没有指定objects参数。注意，虚拟时钟必须在使用之前便定义好。

生成时钟Generated Clock

生成时钟是指在设计内部由特殊单元（如MMCM、PLL）或用户逻辑驱动的时钟。生成时钟与一个上级时钟（注：官方称作master clock，为与primary clock作区分，这里称作上级时钟）相关，其属性也是直接由上级时钟派生而来。上级时钟可以是一个主时钟，也可以是另一个生成时钟。

生成时钟使用create_generated_clock命令定义，该命令不是设定周期或波形，而是描述时钟电路如何对上级时钟进行转换。这种转换可以是下面的关系：

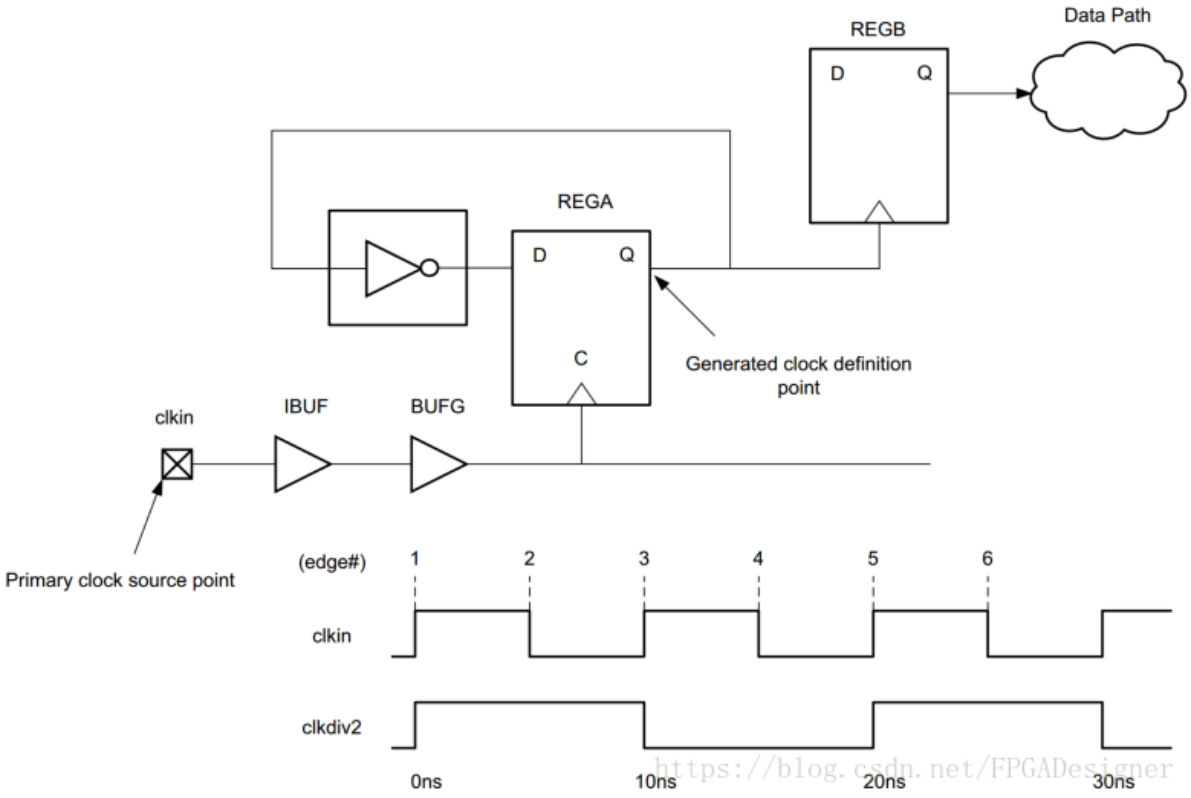
- 简单的频率分频
- 简单的频率倍频
- 频率倍频与分频的组合，获得一个非整数的比例，通常由MMCM或PLL完成
- 相移或波形反相
- 占空比改变
- 上述所有关系的组合

Vivado计算生成时钟的延迟时，会追踪生成时钟的源管脚与上级时钟的源管脚之间的所有组合和时序路径。某些情况下可能只希望考虑组合逻辑路径，在命令行后添加-combinational选项即可。

这里先解释一下本文甚至本系列大量使用的两个词，端口（Port）和管脚（Pin）。端口通常用get_ports命令获取，管脚使用get_pins命令获取。二者的含义是不同的，但管脚的范围更广泛，比如设计中用到的一个寄存器都有3个管脚：clk、D和Q。下面给出几个定义生成时钟的例子：

1.简单的2分频

下图中，主时钟clkin通过端口进入FPGA，使用一个寄存器REGA对其2分频，得到的生成时钟clkdiv2驱动其它的寄存器管脚。



可以采用如下两种方法对生成时钟进行约束：

```
#定义主时钟，周期10ns，50%占空比
create_clock -name clkin -period 10 [get_ports clkin]
#约束方法1，主时钟作为源点
create_generated_clock -name clkdiv2 -source [get_ports clkin] -divide_by 2
[get_pins REGA/Q]
#约束方法2，REGA的时钟管脚作为源点
create_generated_clock -name clkdiv2 -source [get_pins REGA/C] -divide_by 2
[get_pins REGA/Q]
```

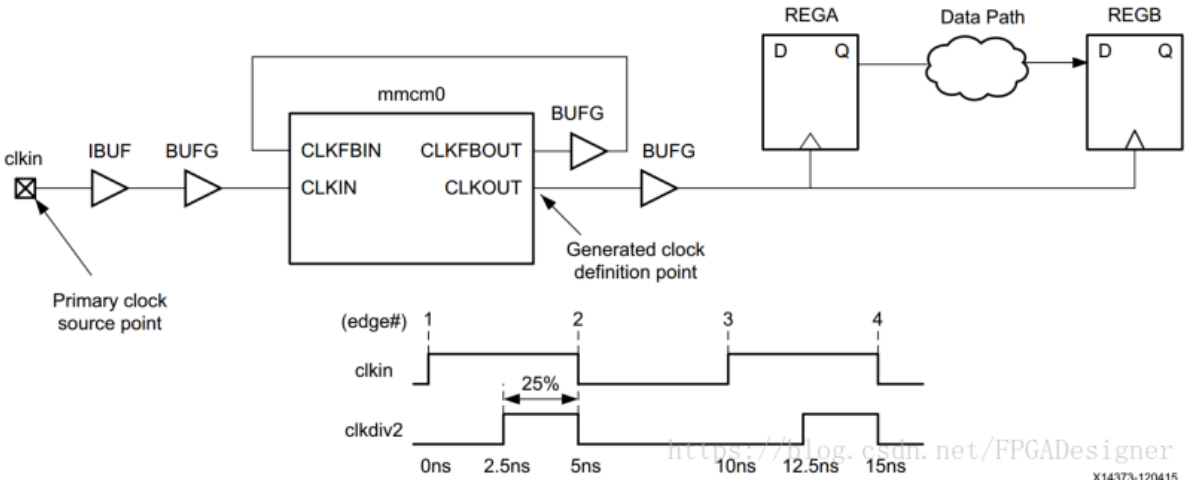
约束命令中使用**-source**选项来设定上级时钟，但如上所示，该选项只能设定为一个端口或管脚类型的网表对象，不能直接设置为时钟类型对象。上面约束使用**-divide_by**选项设置分频系数，此外还可以使用**-edges**选项，如下所示：

```
#该约束与上面等效
create_generated_clock -name clkdiv2 -source [get_pins REGA/C] -eedges {1 3 5}
[get_pins REGA/Q]
```

-edges的参数为一个列表，该列表通过主时钟的边沿来描述生成时钟的波形。列表中的值为主时钟边沿的序号（注意观察上图），由时钟上升沿开始，定义了生成时钟边沿的时间点。

2.改变占空比与相移

如果仅需要改变时钟的相移，使用**-edge_shift**选项可以正向或反向设定每一个生成时钟波形的相移量。注意，-edge_shift选项不能与-devide_by、-multiply_by、-invert选项同时使用。下图中上级时钟为clkin，进入mmcm0单元，产生一个25%占空比、相移90°的时钟：



可以采用如下方法对生成时钟进行约束。使用上级时钟的1、2、3标号边沿（即0ns、5ns、10ns）定义生成时钟，为了得到预期波形，1和3标号边沿要分别移动2.5ns，得到2.5ns、5ns、12.5ns的波形。

```
#定义主时钟，周期10ns，50%占空比
create_clock -name clkin -period 10 [get_ports clkin]
#定义生成时钟，周期10ns，25%占空比，90°相移
create_generated_clock -name clkshifit -source [get_pins mmcm0/CLKIN] -edges {1
2 3} -edge_shift {2.5 0 2.5} [get_pins mmcm0/CLKOUT]
```

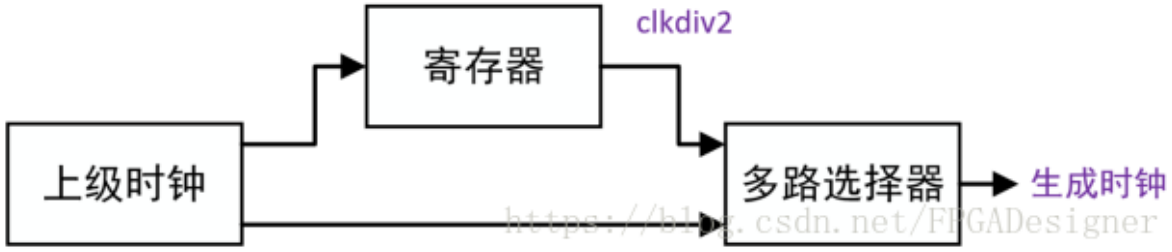
3.同时倍频与分频

这种情况通常用于定义MMCM或PLL的输出，一般使用这些IP核时会自动创建相应约束。考虑上例中的图，假设MMCM将上级时钟倍频到4/3倍，无法直接倍频，需要同时使用-divede_by和-multiply_by选项来实现：

```
create_clock -name clkin -period 10 [get_ports clkin] #定义主时钟
#定义生成时钟，4/3倍频
create_generated_clock -name clk43 -source [get_pins mmcm0/CLKIN] -multiply_by 4
-divide_by 3 [get_pins mmcm0/CLKOUT]
```

4.仅通过组合路径追踪上级时钟

前面简单介绍了-combinational选项的使用，为了更好理解，这里举一个具体例子。下图中，上级时钟同时传递到寄存器和多路选择器中，寄存器对时钟进行2分频。多路选择器从寄存器的2分频时钟和上级时钟中选择一个作为生成时钟输出。



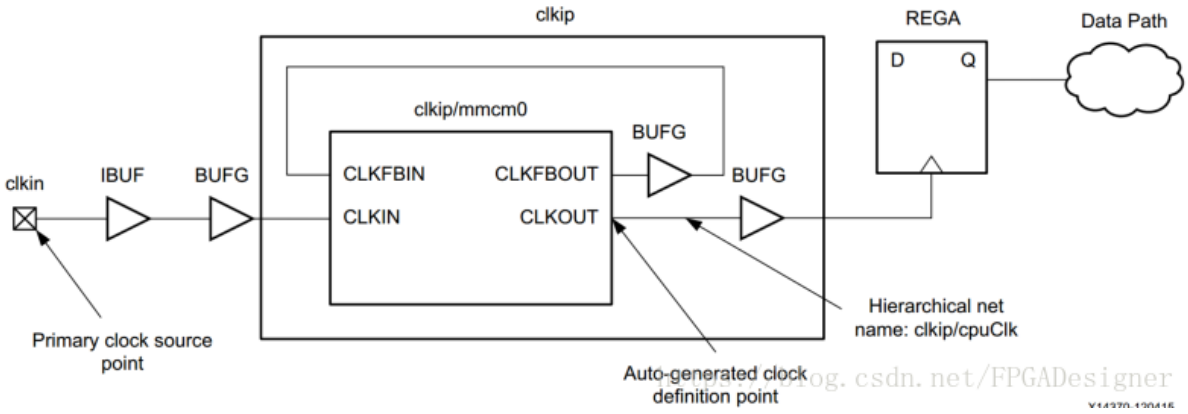
显而易见，从上级时钟到生成时钟有两条路径，一条为时序路径，一条为组合路径。如果我们只希望考虑组合路径上的延迟时，定义生成时钟时就需要使用-combinational选项。

自动生成时钟

这种类型时钟算是生成时钟的一种特例，“自动”是指在已经定义了上级时钟的情况下，Vivado会自动为时钟管理单元**CMBs**（Clock Modifying Blocks）的输出管脚创建约束。官方称作Automatically Derived Clocks或Auto-generated Clock。

7系列FPGA的CMB单元包括MMCM、PLL、BUFR、PHASER；UltraScale系列FPGA的CMB单元种类与数量更多，这里不陈列。如果约束中已经存在用户在某一网表对象上定义的时钟，则不会创建相同对象上的自动生成时钟。

下面给出一个具体例子。下图中上级时钟clkin驱动clkip/mmcm0单元的CLKIN输入，该单元是一个MMCME2资源的实例。则自动生成时钟的定义源点为clkip/mmcm0/CLKOUT，顶层与此源点连接的网络名为clkip/cpuClk，自动生成时钟的名字便是cpuClk。



如上所述，Vivado会自动创建自动生成时钟的名称（Name），如果两个名称发生冲突也会自动添加后缀，如usrclk、usrclk_1等等。Vivado也支持对已经创建好的自动生成时钟重新命名，但很少用到，这里不做介绍。

时钟组Clock Group

很多初学者应该也没有接触过时钟组这个概念。默认情况下，Vivado会测量设计中所有时钟之间的路径时序。添加如下两种约束可以控制该功能：

- set_clock_groups：建立时钟组，Vivado不会对不同时钟组的时钟之间进行时序分析。
- set_false_path：将两个时钟之间的路径设置为false path后，不会对该路径进行任何时序分析。

划分时钟组通常有两个依据：（1）.原理图或时钟网络报告中的时钟树拓扑图，判断哪些时钟不应该放在一起做时序分析；（2）.时钟交互报告查看两个时钟间存在的约束，判断它们是否有共享的主时钟（代表是否有已知的相位关系）或者是否有公共周期。

但要明白，我们设定时钟组的目的还是为了保证设计在硬件中能正常工作，因此我们必须确保这些忽略了时序分析的路径有合适的再同步电路或异步数据传输协议。根据时钟间的关系，可以做如下分类：

- 同步时钟：即两个时钟间有可预知的相对相位，通常它们的时钟树源自网表中的同一个根，且有一个公共周期。
- 异步时钟：两个时钟间有无法预知的相对相位。比如两个独立的晶振信号通过两个输入端口进入FPGA中，生成两个时钟。由于两个主时钟没有明确的相位关系，两个生成时钟间便是异步的。
- 不可扩展时钟：官方称作Unexpandable Clocks，是指时序引擎在1000个周期内无法判断两个时钟是否有公共周期。这种情况通常发生在两个时钟周期比是一个特殊的分数，比如一个主时钟通过MMCM生成一个周期为5.125ns的时钟clk1和一个周期为6.666ns的时钟clk2，尽管它们在时钟树的根上有一个确定的相位关系，但是在1000个周期内时钟上升沿无法再次对齐。

1.异步时钟组

同步时钟可以安全地进行时序分析。异步时钟和不可扩展时钟虽然通过时序分析也会得到一个裕量值，但这个值不可作为可靠结果。从这个角度出发，不可扩展时钟也可以视作一种特殊的异步时钟。这就需要通过设置时钟组来忽略异步时钟的时序路径上的时序分析。

这里举个例子，一个主时钟clk0通过MMCM生成两个时钟usrclk和itfclk；另一个主时钟clk1通过另一个MMCM生成两个时钟clkrx和clktx。用如下命令创建异步时钟组：

```
set_clock_groups -name async_clk0_clk1 -asynchronous -group {clk0 usrclk itfclk}
-group {clk1 clkrx clktx}
#如果时钟名称事先不知道，可以用如下写法
set_clock_groups -name async_clk0_clk1 -asynchronous -group [get_clocks -
include_generated_clocks clk0] -group [get_clocks -include_generated_clocks
clk1]
```

2.互斥时钟组

下面再介绍另一种会用到时钟组的情况。某些设计会有几个操作模式，不同操作模式使用不同的时钟。这些时钟通常由专用的时钟选择器进行选择，如BUFGMUX和BUFGCTRL，最好不要用LUT作时钟选择器。

这些单元都是组合逻辑单元，Vivado会将所有输入传递到输出。在Vivado IDE中，几个时序时钟可以同时存在时钟树上，方便地同时报告所有操作模式。但是在硬件中这是不可能的，它们之间是互斥的，这些时钟便称作互斥时钟。

举个例子，一个MMCM实例生成的两个时钟clk0和clk1，与一BUFGMUX实例clkmux相连，clkmux的输出驱动设计时钟树。默认情况下，虽然clk0和clk1共享同一时钟树，且不能同时存在，Vivado还是会分析clk0和clk1之间的路径。这个问题要通过设置互斥时钟组来解决，达到禁止分析这两个时钟间路径 的目的。约束如下：

```
set_clock_groups -name exclusive_clk0_clk1 -physically_exclusive -group clk0 -
group clk1
```

在ASIC工艺中使用-physically_exclusive和-logically_exclusive代表不同的信号完整性分析模式，但对于Xilinx FPGA而言，二者是等价的，都可以使用。

时钟延迟、抖动与不确定性

本文的上述约束可以说都是对时钟的理想特征进行约束，为了更精确地进行时序分析，设计者还必须设定一些与运行环境相关的可预测变量和随机变量。这部分也称作时钟的不确定性特征。

1.时钟延迟latency

经过板子上和FPGA器件内部的传输，时钟边沿到达目的地后会有一个确定的延迟。这个延迟可以分为两个部分看待：

- 网络延迟：也称作插入延迟，指再FPGA内部传输带来的延迟。Vivado会自动分析计算该延迟，布线过程前只是一个粗略的估计，布线后便可以得到一个精确的值。对于生成时钟，包含其本身的网络延迟和上级时钟的网络延迟两部分。
- 源端延迟：通常指FPGA器件外，时钟进入源点前的传输延迟，这部分延迟与PCB设计相关，需要用set_clock_latency命令进行约束。

下面给出一个约束源端时钟延迟的例子：

```
#设定最小源端延迟值
set_clock_latency -source -early 0.2 [get_clocks sysclk]
#设定最大源端延迟值
set_clock_latency -source -late 0.5 [get_clocks sysclk]
```

2.时钟抖动jitter

对于ASIC器件来说，时钟抖动通常代表了时钟不确定性特征；但对于Xilinx FPGA而言，抖动属性被当作可预测变量看待。抖动有的需要单独设置，有的在时序分析过程中自动计算。抖动分为两种：

- 输入抖动：指实际时钟边沿与理想时钟边沿到达时刻之间的差值，使用set_iput_jitter命令为每个主时钟单独设置输入抖动。但是不能直接为生成时钟设置输入抖动，这部分由工具自动计算，如果（1）.生成时钟由一个组合或时序单元创建，生成时钟的抖动与上级时钟相同；（2）.生成时钟由MMCM或PLL驱动，生成时钟的抖动为一个自动计算的值。
- 系统抖动：指电源噪声、板级噪声或其它原因引起的整体的抖动，对于整个设计，使用set_system_jitter命令设置一个值即可，会应用到所有时钟。

下面给出一个约束输入抖动的例子：

```
#主时钟传输过程中有±100ps的抖动
set_input_jitter [get_clocks -of_objects [get_clocks sysclk]] 0.1
```

不过，时钟抖动对整个时钟不确定性计算的影响不是太大。计算时钟不确定性时对每条路径都是独立的，且主要依赖于时钟拓扑结构、路径上的时钟对、时钟树上是否存在MMCM/PLL单元等其它因素。

3.附加的时钟不确定性

使用set_clock_uncertainty命令可以根据需要为特定的时钟关系定义附加的时钟不确定性，这样在时序分析时，可以为设计中的某些部分增加额外裕量。

前面文章说过XDC约束带有顺序性，后面的约束会重写前面的约束。但在这里，时钟间的不确定性总是优先于单个时钟的不确定性，不管约束顺序如何。看下面的例子：

```
set_clock_uncertainty 2.0 -from [get_clocks clk1] -to [get_clocks clk2]
set_clock_uncertainty 1.0 [get_clocks clk1]
```

这里首先约束从clk1到clk2有一个2ns的时钟不确定性，接着又约束clk1有1ns的时钟不确定性，但是后面这条约束不会改动从clk1到clk2之间的关系。

2.IO延迟的约束方法

为了对设计外部的时序情况进行精确建模，设计者必须设定输入和输出端口的时序信息。Vivado只能识别出FPGA器件范围内的时序，因此必须使用set_input_delay和set_output_delay命令来设置FPGA范围外的延迟值。两者在含义、约束命令等方面有很多地方是相似的，只不过一个是输入，一个是输出，本文还是分开对两者进行讲述。

输入延迟

set_input_delay命令设定输入端口上相对于设计接口时钟边沿的输入路径延迟。输入延迟既指数据从外部芯片经过板级传输到FPGA输入管脚间的相位差，也指相对参考板级时钟间的相位差。输入延迟值可以是正的，也可以是负的，由时钟和数据在FPGA接口处的相对相位决定。

约束时的相对时钟可以是一个设计时钟，也可以是一个虚拟时钟。输入延迟命令的选项包括：

- -min和-max：-min设定的值用于最小延迟分析（保持时间、移除时间）；-max设定的值用于最大延迟分析（建立时间、恢复时间）。如果约束命令中没有使用这两个选项，输入延迟值会同时应用于min和max。
- -clock_fall：用于指定由相对时钟的下降沿启动的时序路径上的输入延迟约束。如果没有这个选项，Vivado只假定使用相对时钟的上升沿。
- -add_delay：该选项通常用于约束与多个时钟沿相关的输入端口（比如DDR接口），而且必须已经存在一个最大或最小输入延迟约束，设计者使用该命令为同一端口设置其它相对时钟沿的最大或最小输入延迟约束。

输入延迟约束只能应用于输入端口或双向端口（不包括时钟输入端口），不能用于设计内部的管脚。下面给出几个使用输入延迟约束的典型例子。

1.示例一

```
create_clock -name sysClk -period 10 [get_ports CLK0]
set_input_delay -clock sysClk 2 [get_ports DIN]
```

定义一个主时钟sysClk作为输入延迟的相对时钟，设定的值同时作为最小值（min）分析和最大值（max）分析。

2.示例二

```
create_clock -name clk_port_virt -period 10
set_input_delay -clock clk_port_virt 2 [get_ports DIN]
```

该例子约束目的与上例相同，只是相对时钟换为一个虚拟时钟。使用虚拟时钟的好处是可以在不改变内部设计时钟的情况下，设定任意的抖动和延迟。

3.示例三

```
create_clock -name sysClk -period 10 [get_ports CLK0]
set_input_delay -clock sysClk -max 4 [get_ports DIN]
set_input_delay -clock sysClk -min 1 [get_ports DIN]
```

该例中最小值分析和最大值分析采用不同的输入延迟值。

4.示例四

```
create_clock -name sysClk -period 10 [get_ports CLK0]
set_input_delay -clock sysClk 4 [get_ports DIN]
set_output_delay -clock sysClk 1 [get_ports DOUT]
```

第30篇中介绍过用虚拟时钟对纯组合逻辑进行约束的相关知识，这里给出一个例子。如果两个I/O端口之间仅有组合逻辑路径，没有任何时序单元，必须相对于虚拟时钟为I/O端口定义输入与输出延迟。上例中DIN和DOUT之间的组合逻辑路径约束为5ns（10-1-4）。

5.示例五

```
create_clock -name clk_ddr -period 6 [get_ports DDR_CLK_IN]
set_input_delay -clock clk_ddr -max 2.1 [get_ports DDR_IN]
set_input_delay -clock clk_ddr -max 1.9 [get_ports DDR_IN] -clock_fall -
add_delay
set_input_delay -clock clk_ddr -min 0.9 [get_ports DDR_IN]
set_input_delay -clock clk_ddr -min 1.1 [get_ports DDR_IN] -clock_fall -
add_delay
```

这里相对时钟为DDR的时钟，最小值分析和最大值分析采用不同的输入延迟值。约束的一端是器件外部时钟的上升沿和下降沿启动的数据，另一端是器件内部同时对上升沿和下降沿敏感的触发器的输入数据。

输出延迟

set_output_delay命令设定输出端口上相对于设计接口时钟边沿的输出路径延迟。输出延迟既指数据从FPGA的输出管脚通过板级传输到另一个器件间的相位差，也指相对参考板级时钟间的相位差。输出延迟值同样也可以是正的或负的，由时钟和数据在FPGA器件外的相对相位决定。

约束时的相对时钟可以是一个设计时钟，也可以是一个虚拟时钟。输出延迟命令的选项基本与输入延迟约束相同，还是陈列如下：

- -min和-max：-min设定的值用于最小延迟分析（保持时间、移除时间）；-max设定的值用于最大延迟分析（建立时间、恢复时间）。如果约束命令中没有使用这两个选项，输入延迟值会同时应用于min和max。

- -clock_fall：用于指定由相对时钟的下降沿捕获的时序路径上的输出延迟约束。如果没有这个选项，Vivado只假定使用相对时钟的上升沿。
- -add_delay：该选项通常用于约束与多个时钟沿相关的输出端口（比如DDR接口同时使用上升沿和下降沿，或者输出端口与几个使用不同时钟的器件相连），而且必须已经存在一个最大或最小输入延迟约束，设计者使用该命令为同一端口设置其它相对时钟沿的最大或最小输入延迟约束。

同样，输出延迟约束只能应用于输出端口或双向端口，不能用于设计内部的管脚。下面给出几个使用输出延迟约束的典型例子。

1.示例一

```
create_clock -name sysClk -period 10 [get_ports CLK0]
set_output_delay -clock sysClk 6 [get_ports DOUT]
```

定义一个主时钟sysClk作为输出延迟的相对时钟，设定的值同时作为最小值（min）分析和最大值（max）分析。

2.示例二

```
create_clock -name clk_port_virt -period 10
set_output_delay -clock clk_port_virt 6 [get_ports DOUT]
```

该例子约束目的与上例相同，只是相对时钟换为一个虚拟时钟。使用虚拟时钟的好处是可以在不改变内部设计时钟的情况下，设定任意的抖动和延迟。

3.示例三

```
create_clock -name clk_ddr -period 6 [get_ports DDR_CLK_IN]
set_output_delay -clock clk_ddr -max 2.1 [get_ports DDR_OUT]
set_output_delay -clock clk_ddr -max 1.9 [get_ports DDR_OUT] -clock_fall -
add_delay
set_output_delay -clock clk_ddr -min 0.9 [get_ports DDR_OUT]
set_output_delay -clock clk_ddr -min 1.1 [get_ports DDR_OUT] -clock_fall -
add_delay
```

这里相对时钟为DDR的时钟，最小值分析和最大值分析采用不同的输出延迟值。约束的一端是器件外部时钟的上升沿和下降沿启动的数据，另一端是器件内部同时对上升沿和下降沿敏感的触发器的输出数据。

最后再补充一句，虽然上面说输入延迟约束和输出延迟约束不能应用于FPGA内部管脚，但也有特例。UltraScale+系列FPGA的STARTUPE3内部管脚就可以进行输入延迟和输出延迟约束。不过博主连UltraScale+的芯片都没摸过，本文便不做介绍。

3.时序例外/异常（Timing Exception）

英文名为Timing Exception，可以认为是时序例外或时序异常（本系列文章 的称法），“例外”或“异常”是指这部分时序的分析与大多数常规时序分析不同。下表给出了Vivado支持的时序异常命令及功能：

命令	功能
set_multicycle_path	设置路径上从起点到终点传递数据需要的时钟周期数
set_false_path	指示设计中的某条逻辑路径不进行时序分析
set_max_delay、set_min_delay	设置最小与最大路径延迟值，会重写默认的建立与保持约束
set_case_analysis	使用端口或管脚上的逻辑常量或逻辑转换进行时序分析，以限制信号在设计间的传递

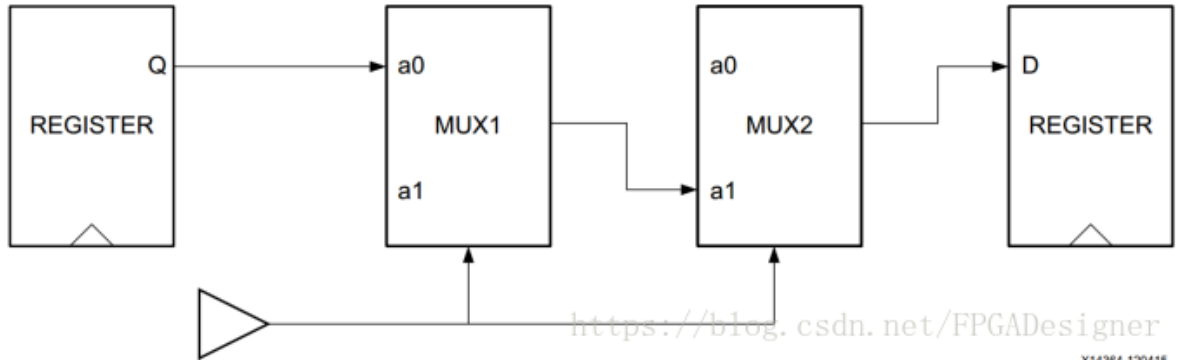
Vivado不支持即时分析有矛盾的时序异常，需要运行**report_exceptions**进行完整的分析，报告所有时序异常。多周期路径的情况有很多，比较复杂，单独放在第35篇中讲述。本文介绍其余三种时序异常的相关知识。

虚假路径false path

某些在拓扑结构上看存在于设计中的路径，但是没有工作或者不需要被计时，便被称作虚假路径。虚假路径在时序分析过程中应该被忽略不计。下面这些情况都属于虚假路径：

- 在有双同步器逻辑的地方有时钟域交叉
- 可能只在上电时写入一次的寄存器
- 复位或测试逻辑
- 异步分布式RAM的写端口于异步读时钟之间的路径

举个具体的例子加深对虚假路径的理解，如下图：



两个多路选择器MUX控制两个寄存器间的数据传输，但是两个MUX采用同一个选择信号。仔细分析会发现，无论如何Q端数据都无法传递到D端。虽然结构图上看起来Q和D之间存在一条路径，但是没有起到任何功能，因此应该定义为虚假路径。

为何要在时序分析中移除掉虚假路径？理由如下：

- 减少运行时间：工具不需要为这些虚假路径计时和做优化，可以减少很多运行时间。
- 增加结果质量：移除虚假路径可以极大增加结果质量QOR（Quality of Results）。综合、布局和优化设计的质量很大程度上受到时序问题的影响，因为工具总会尝试解决这些问题，包含虚假路径会导致不理想的结果（比如工具将过多注意力放在解决虚假路径的时序违背上，而忽视了真正需要解决的问题）。

虚假路径由**set_false_path**命令定义，该命令模板如下：

```
set_false_path [-setup] [-hold] [-from <node_list>] [-to <node_list>] [-through <node_list>]
```

几个节点列表选项的含义如下：

- -from：一组合法的起点列表，包括时钟对象、时序元素的时钟管脚、输入主端口或双向主端口。
- -to：一组合法的终点列表，包括时钟对象、时序元素的数据输入管脚、输出主端口或双向主端口。
- -through：一组合法的管脚或端口，注意节点的顺序很重要。如果约束中仅使用了-through，没有使用-from和-to选项，Vivado会从时序分析中移除所有通过该列表的路径，使用时要特别小心。

下面给出几个定义虚假路径的例子：

```
#-through的顺序表示路径穿过节点的顺序，因此下面是两条不同的约束
set_false_path -through cell1/pin1 -through cell2/pin2
set_false_path -through cell2/pin2 -through cell1/pin1

#上图中的虚假路径应该用下面这条命令约束
set_false_path -through [get_pins MUX1/a0] -through [get_pins MUX2/a1]
#使用-through而不用-from和-to的好处是可以确保所有通过此节点的路径都会被移除，而不用考虑起点和终点

#移除复位端口到所有寄存器间的时序路径
set_false_path -from [get_port reset] -to [all_registers]

#禁用两个异步时钟域间的时序路径，从CLKA到CLKB
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
#注意，上述命令并没有禁用从CLKB到CLKA的路径，还需要补充如下约束
set_false_path -from [get_clocks CLKB] -to [get_clocks CLKA]
```

从最后一个例子可知，我们需要双向地禁用时序路径，但是如果设计中有多个异步时钟域，编写起来就非常麻烦。不知道您是否还记得第31篇中讲过的时钟约束方法，这种情况最好其实应该使用set_clock_groups设置不同的异步时钟组。

上面还说到了异步分布式RAM的情况，这里也举一个约束例子。假设一个异步双口分布式RAM，其写操作于RAM时钟同步，但是读操作是异步的，这种情况下应该在写和读时钟间设置一个虚假路径。约束如下：

```
#在RAM前的写寄存器和RAM后的读寄存器间设置虚假路径
set_false_path -from [get_cells <write_registers>] -to [get_cells
<read_registers>]
```

个例分析

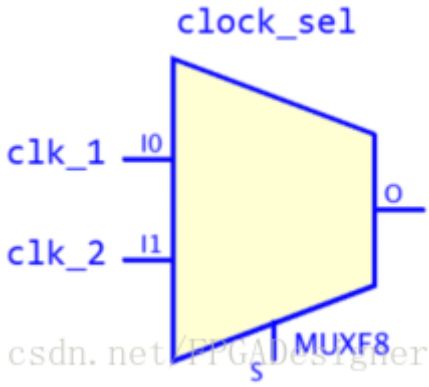
设计中，某些信号在特定模式中为常数值，比如：（1）.某些测试信号不会变换，直接连接在VSS或VDD上；（2）.某些信号上电后便不再发生变化；（3）.如果设计有多种功能模式，某些信号在部分模式下为活跃状态，但在其它模式下为不活跃状态。这些情况便属于“个例分析”。

我们必须告诉静态时序分析引擎，哪些信号为常数值，从而减少分析范围、运行时间和内部占用率，并且不必报告那些不工作的和不相关的路径。通常，设计者使用set_case_analysis命令将信号（管脚和端口）申明为不活跃状态。该命令的语法如下：

```
set_case_analysis <value> <pins or ports objects>
```

参数值value可以是0、1、zero、one、rise、rising、fall或falling。作用对象可以是端口（port）、子单元（英文名为leaf cell）的管脚或层次模块的管脚。下面举两个例子加强对这种时序异常的理解。

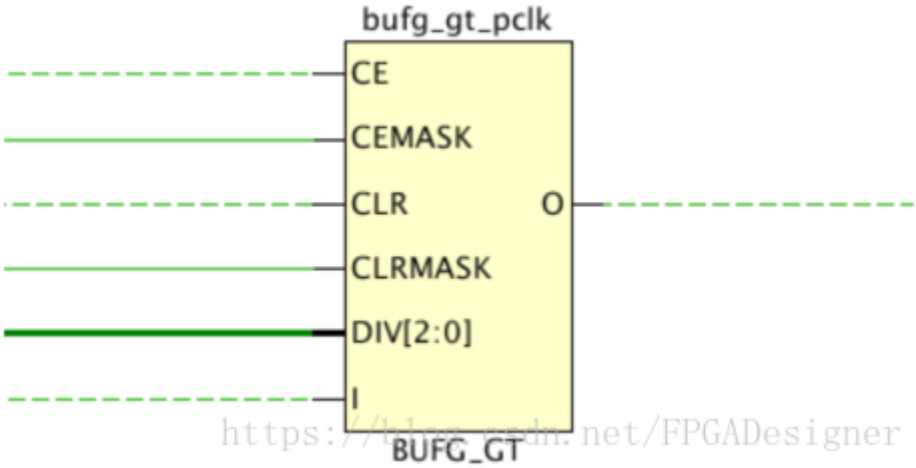
第一个例子如下图，clock_sel是一个时钟选择器，通过选择管脚s对两个输入时钟clk_1和clk_2进行选择输出：



如果我们只希望分析一种个例，比如只分析选择clk_2时的情况。通过将 s管脚设置为常数即可只把clk_2传递到输出端口o。约束如下：

```
set_clock -name -clk_1 -period 10.0 [get_pins clock_sel/I0]
set_clock -name -clk_2 -period 15.0 [get_pins clock_sel/I1]
set_case_analysis 1 {get_pins clock_sel/S}
```

对某一管脚设置了个例分析，会导致禁用经过该管脚的路径上的时序分析，也不会报告相关信息。第二个例子如下图，BUFG_GT有一个动态时钟分频控制管脚DIV[2:0]，由其它逻辑驱动而不是直接连接到VCC/GND：



默认情况下，Vivado会假设输出时钟的最坏可能情况，即1分频（相当于不分频，此时频率最高）。然而如果设计中根本不会出现DIV取1的情况，这就成了过度约束。为了更合理地约束设计，我们可以对DIV[2:0]总线进行个例分析约束，比如可能出现的最差情况为DIV取3，约束如下：

```
set_case_analysis 0 {get_pins bufg_gt_pclk/DIV[0]}
set_case_analysis 1 {get_pins bufg_gt_pclk/DIV[1]}
set_case_analysis 0 {get_pins bufg_gt_pclk/DIV[2]}
```

最小/最大延迟

最大延迟约束**set_max_delay**用于改写路径的默认建立时间（或恢复时间）需求；最小延迟约束**set_min_delay**用于改写路径的默认保持时间（或移除时间）。两条约束命令的语法模板如下：

```
set_max_delay <delay> [-datapath_only] [-from <node_list>] [-to <node_list>] [-
through <node_list>]
set_min_delay <delay> [-from <node_list>] [-to <node_list>] [-through
<node_list>]
```

-from、-to和-through和虚假路径中的用法相同。set_max_delay命令中如果添加了-datapath_only，那么计算裕量时便不会考虑时钟斜率。使用最小/最大延迟约束要注意如下三点：

- 路径上仅设置最大延迟约束（不使用-datapath_only选项），不会修改该路径上的最小延迟需求，保持时间检查仍采用默认值，相反也成立。但如果加入了-datapath_only，就会导致该路径上的保持时间需求被忽略。
- 通常输入端口到第一级寄存器间的约束用set_input_delay命令；最后一级寄存器到输出端口之间的约束用set_output_delay命令（详情见第32篇）。但输入端口到输出端口之间的纯组合逻辑路径可以用set_max_delay和set_min_delay命令进行约束（通常称为in-to-out I/O路径）。
- 某些异步信号间没有时钟关系，但是需要最大延迟约束。比如我们通常用set_clock_groups划分两个异步时钟域，但有时我们需要确保两个时钟域之间的路径延迟不要太高。这种情况下，我们就要用set_max_delay和set_false_path的命令组合（因为set_clock_groups的优先级更高，会取代set_max_delay，因此不能和其一块使用）。

另外在约束最小延迟和最大延迟时，如果-from和-to中的节点选择不合理，会出现**路径分割**（Path Segmentation）现象。第34篇给出了路径分割的具体实例及说明。

禁用Timing Arcs

最后再介绍一种时序异常的特例：Timing Arcs，字面翻译为时序弧，之所以没有单独列为一种时序异常，是因为它与其它时序异常有着千丝万缕的关系。其实很多情况下计时器为了处理一些特殊情况会自动禁用某些时序弧，比如：

- 组合逻辑反馈环不能被正确地计时（因此也不推荐使用），计时器会通过禁用环内的某条时序弧来打破环路。
- 据前文所述，默认情况下MUX的所有输入数据都会传递端口，但是个例分析时将MUX的选择信号设为常数值，此时仅有一个数据输入端口会传递到输出端口。其实这正是由计时器打断了其它数据端口到输出端口间的时序弧实现的。

Vivado提供了**set_disable_timing**命令，可以人为打断一个单元输入端口到输出端口之间的时序弧。考虑如下应用情况：

- 比如对于上面第一个例子的情况，可以人工设定打断组合反馈环中的哪条时序弧，而不是让工具自动完成。
- 假设有多个时钟同时到达LUT的输入管脚，但是只能有一个时钟传递到LUT的输出端口。此时需要打断与其它时钟相关的时序弧。

当禁用了时序弧后，通过该时序弧的所有时序路径都不会在时序分析中报告。因此使用时要额外小心，避免禁用了必要的时序弧，导致隐藏的时序违背或时序问题使设计在硬件中不能正常工作。
set_disable_timing的语法及示例如下：

```
#语法，-from和-to只能设置为库单元的管脚名称（不是设计管脚名称）
set_disable_timing [-from <arg>] [-to <arg>] [-quiet] [-verbose] <objects>

#禁用所有基于LUTRAM的异步FIFO的WCLK到O之间的时序弧
set_disable_timing -from WCLK -to O [get_cells inst_fifo_gen/
gdm.dm/gpr1.dout_i_reg[*]]

#指定对象的所有以O管脚为终点的时序弧都被禁用
set_disable_timing -to O <objects>
```

```
#指定对象的所有以WCLK管脚为起点的时序弧都被禁用
set_disable_timing -from WCLK <objects>

#指定对象内的所有时序弧都被禁用
set_disable_timing <objects>
```

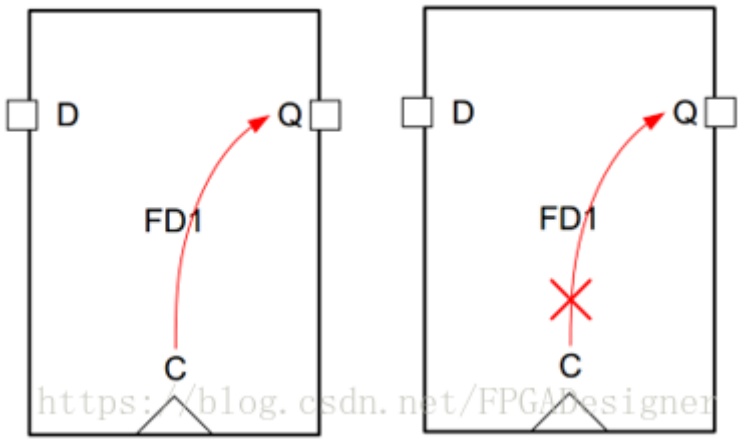
使用report_disable_timing命令可以查看所有自动禁用和手动禁用了的时序弧。注意这个列表可能会非常大，最好加上-file选项保存结果到文件中查看。

3.路径分割现象

上文提到，进行最小/最大延迟约束时，**set_max_delay**和**set_min_delay**命令要设置-from和-to选项。但是如果起点和终点设置的不合理（具体见第33篇），便会导致出现路径分割（Path Segmentation）。

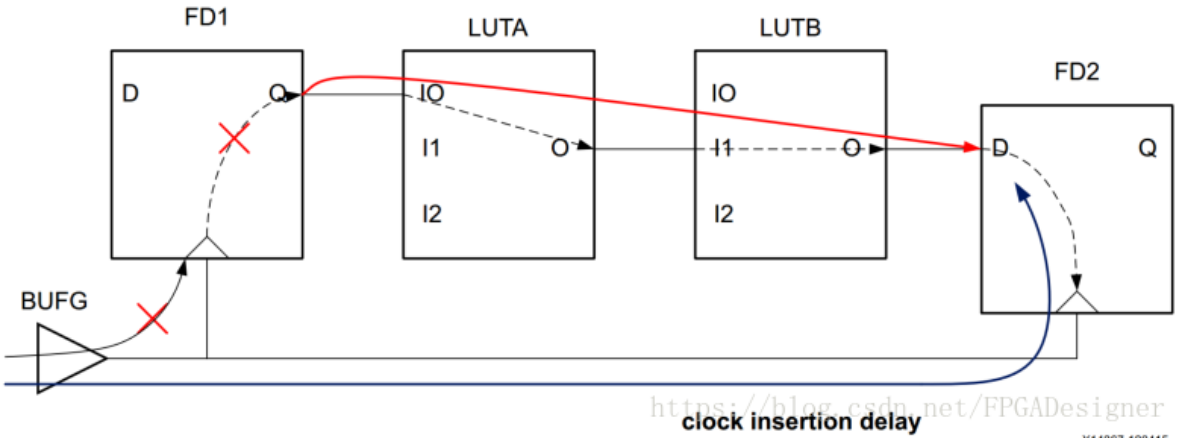
非法的起点

下面举一个例子说明，如果-from设置了一个非法的起点，时序引擎会阻止经过该节点的时序的传递，从而这个“非法的”起点变得“合法”。如下图：



FD1单元中唯一合法的起点只有C端口，即“set_max_delay 5 -from [get_pins FD1/C]”。但如果将起点设置为Q端口，时许引擎便会阻止通过C->Q的时序弧，从而Q变成了一个合法的起点。这个阻止时序传递并创建一个合法起点的过程便是路径分割。

路径分割并不是一个好的现象，它会影响到最大和最小延迟分析，以及所有经过这些节点的时序约束。如下图所示：



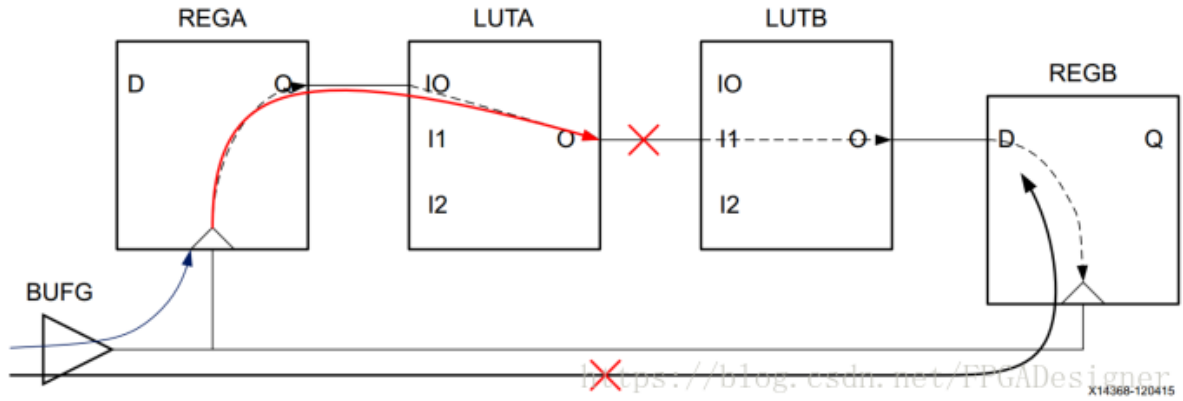
对于以Q为起点的路径，启动时钟不会考虑时钟插入延迟。但是在路径终点处（FD2/D）仍会计算插入延迟，这就可能导致比较大的时钟斜率。总之，应该尽量避免出现路径分割现象，如果不得不必须使用也要非常小心。如果出现了路径分割，Vivado会提示一个严重警告。

路径分割会导致该路径上没有保持时间需求，如果需要的话，可以使用set_min_delay命令为该路径设置一个保持时间需求（在没有用-datapath_only的情况下）。一般来说，路径分割总是能避免的，比如上例中，如果只是不想考虑时钟斜率而把FD1/Q作为起点，完全可以用-datapath_only选项达到此目的：

```
set_max_delay 5 -from [get_pins FD1/C] -datapath_only
```

非法的终点

与上例相同，如果-to设置了一个非法的终点，时序引擎会阻止该节点之后的传递，从而这个“非法的”终点变得“合法”。如下图：

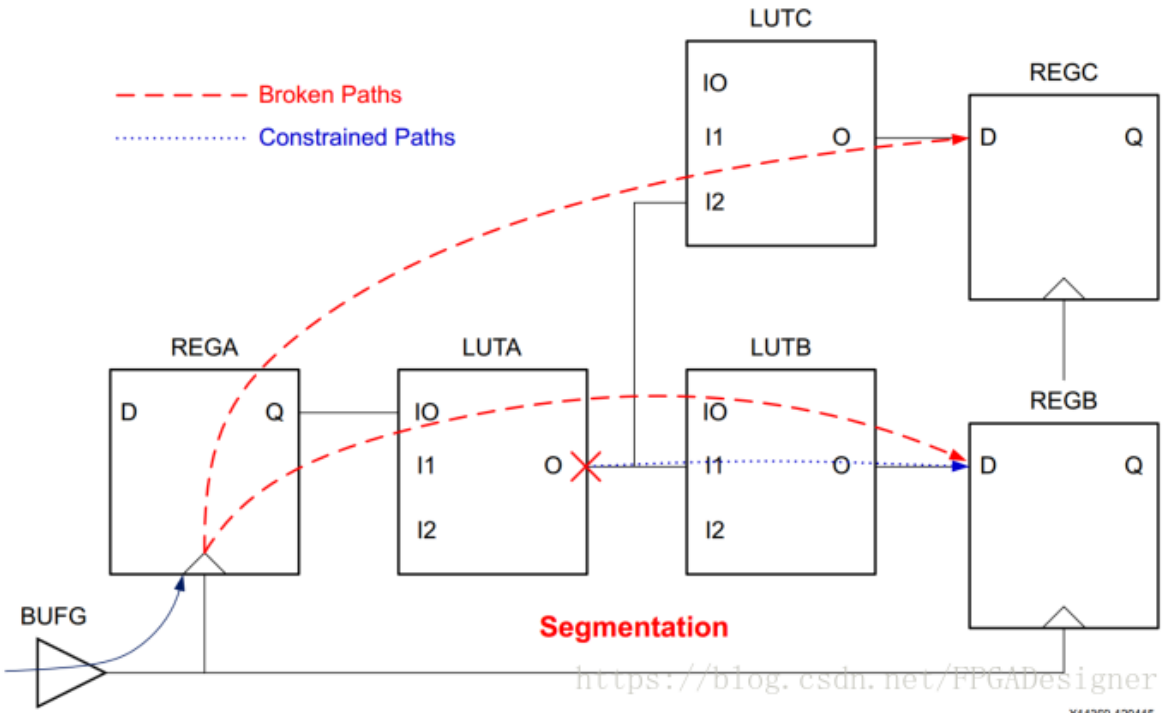


如果约束为“set_max_delay 5 -to [get_pins LUTA/O]”，LUTA/O作为组合逻辑单元的数据输出管脚并不是一个合法的终点。但是时序引擎会阻止LUTA/O之后的传递，使其成为一个合法终点。

所有经过LUTA/O的时序路径的建立和保持都会受到影响。从上图看到，计算时只会考虑从REGA/C到LUTA/O之间的时钟插入延迟，这同样也会导致比较大的斜率。

路径分割的影响

再次举例强调一下路径分割的缺点，由于路径分割会阻止时序弧的传递，所有经过这些节点的路径都会受到影响，如下图：



假设在LUTA/O和REGB/D之间设置了最大延迟“set_max_delay 6 -from [get_pins LUTA/O] -to [get_pins REGB/D]”，如上图蓝线。

由于LUTA/O是一个非法的起点，发生了路径分割，所有从LUTA/I*到LUTA/O之间的时序弧都被打断。尽管上面设置的最大延迟约束可以起作用，但是其它经过此节点的路径（REGA/C到REGC/D；REGA/C到REGB/D）由于路径分割的影响也被打断了。

路径分割与时序异常

第33篇中也简单提到过set_max_delay约束会被set_clock_groups约束替代的问题。当发生路径分割时，可能会导致感觉上时序异常的优先级发生了改变，但事实上并非如此。考虑如下两个约束：

```
#情景1
set_max_delay <ns> -datapath_only -from <instance> to <instance>
#情景2
set_max_delay <ns> -datapath_only -from <pin> to <pin | instance>
```

第一个情景中，-from和-to提供实例名称，set_max_delay约束总会被set_clock_groups -asynchronous约束替代。这是因为在给出实例的情况下，Vivado总会优先选择合法的起点。

第二个情景中，如果-from提供的管脚名称导致路径分割现象出现，此时set_max_delay约束不会被set_clock_groups -asynchronous约束替代。这是因为路径分割将该管脚名称强制作为路径起点，Vivado无需自己选择，导致set_max_delay不会被set_clock_groups覆盖。

4.多周期路径

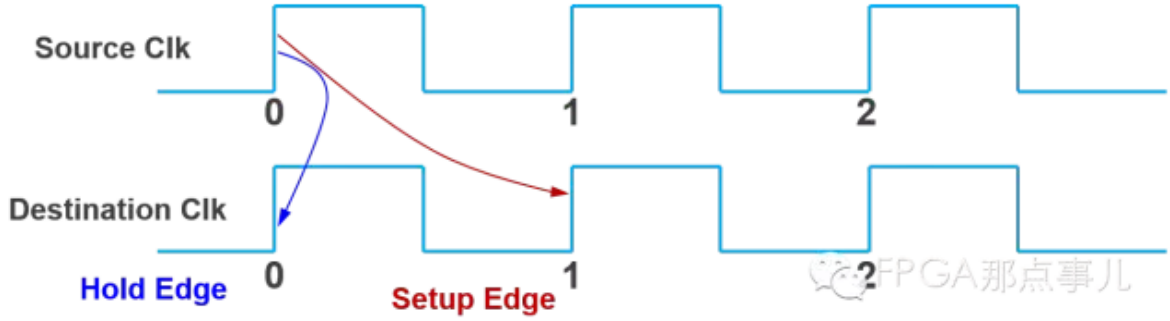
多周期路径 Multicycle Paths

默认情况下，vivado时序引擎是按照单周期关系分析数据关系的，即数据在发起沿发送，在捕获被捕获，发起沿和捕获沿相差一个周期；但是很多情况是，数据路径逻辑较为复杂，导致延时较大，使得数据无法在一个时钟周期内稳定下来， 或者数据可以在一个时钟周期内稳定下来，但是在数据发送几个周期之后才使用；在这些情况中，设计者的意图都是使数据的有效期从发起沿为起始直至数个周期之后的

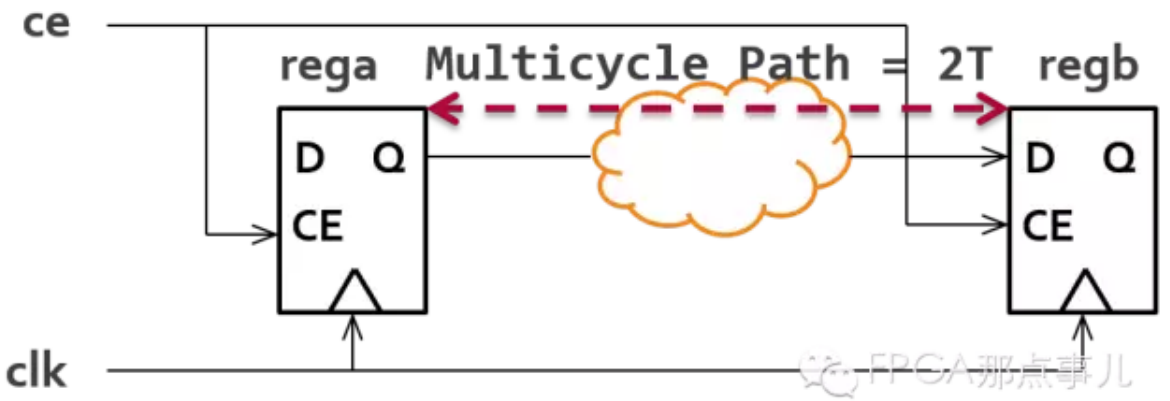
捕获沿，这样的意图无法被时序分析工具猜度出来，必须由设计者在时序约束中指明；否则时序分析工具会按照单周期路径检查的方式执行，往往会误报出时序违规；此时，我们可以将这样的path约束为multicycle path，多周期路径的约束我们会用到。set_multicycle_path；set_multicycle_path 可以用来修改与源时钟或者目的时钟相关的路径所需要的时钟周期数！

如何理解多周期路径约束?

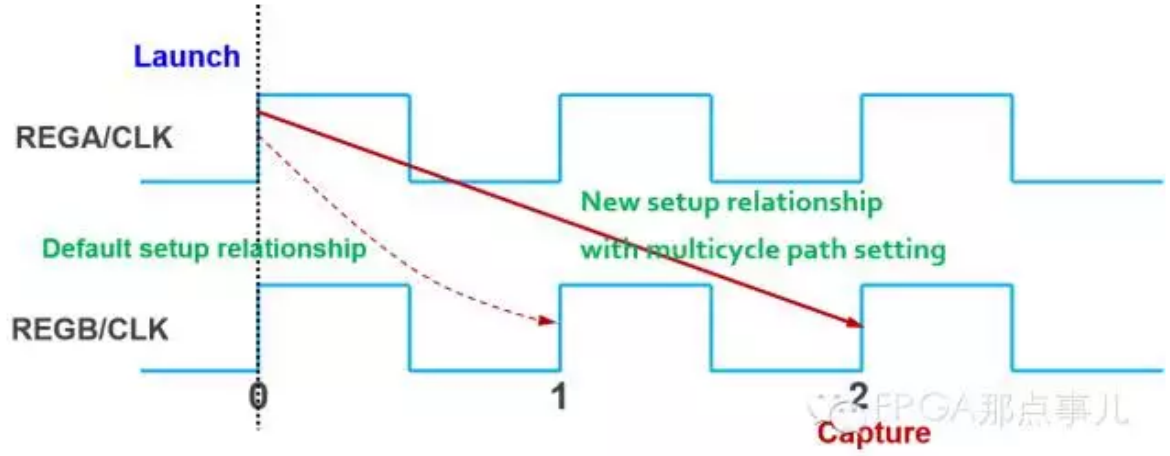
先看看单时钟周期的情形，如下图所示：红色标记为默认情况下的建立时间检查，蓝色标记为默认情况下的保持时间检查，且**注意保持时间的检查是以建立时间的检查为前提，即总是在建立时间检查的前一个时钟周期确定保持时间检查**：



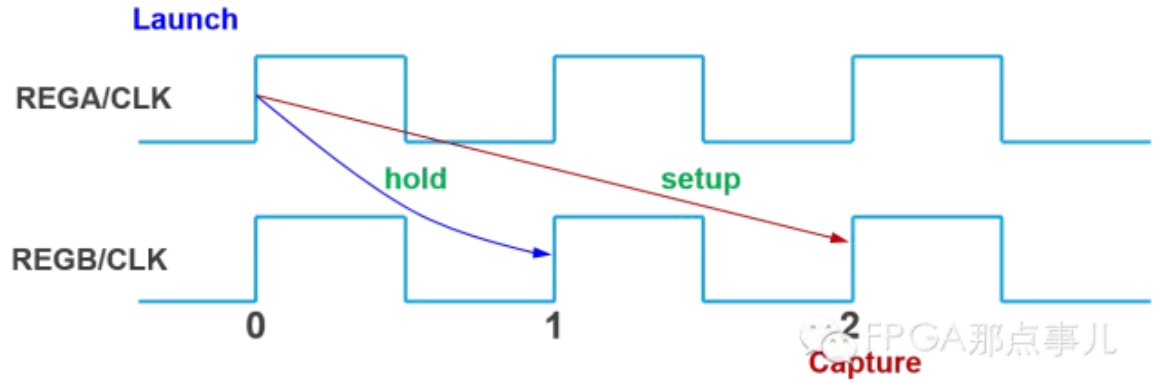
如果是多周期，如下图所示，此时两个寄存器之间尽管使用同一个时钟但因为使能信号的作用，使得两者数据率变为时钟频率的一半，意味着发起沿和捕获沿相隔2个时钟周期：



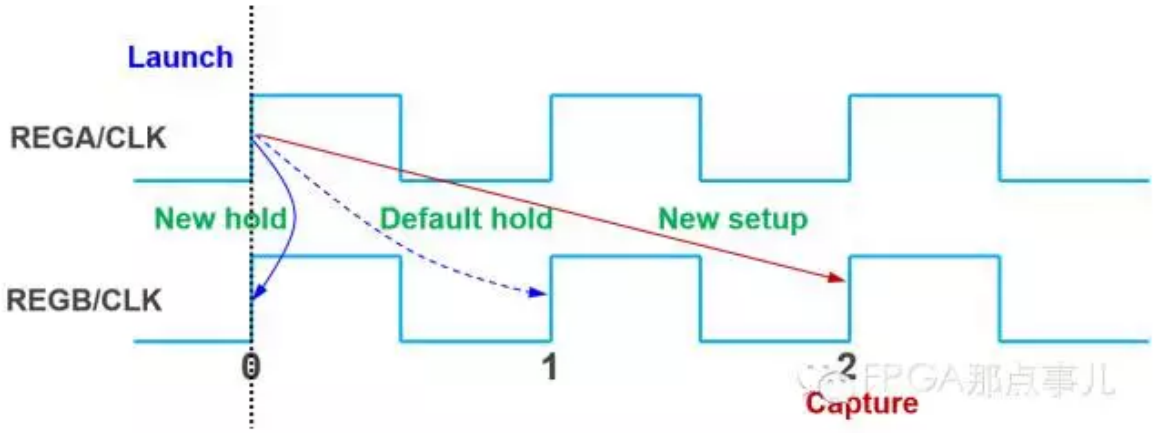
此时新的建立时间检查如下图中的红色实线所示，红色虚线为默认情况下的建立时间检查：



在此基础上，保持时间的检查变为下图中的蓝色实线（建立时间检查前移一个时钟周期）：



但显然，此时的保持时间检查是不对的，可想象一下，如果是一个时钟频率为原始时钟频率的一半的时钟，保持时间检查应该如下图中的蓝色实线所示：



多周期路径的约束我们会用到set_multicycle_path，常用的约束语法格式：

```
set_multicycle_path <path_multiplier> [-setup|-hold] [-start|-end]
[-from <startpoints>] [-to <endpoints>] [-through <pins|cells|nets>]
```

对于大多数情况，可以采用如下公式计算路径的周期数：

```
Hold cycles = <setup path multiplier> - 1 - <hold path multiplier>
```

这里我们结合上述分析重点介绍一下相应参数的含义：

- <path_multiplier>：周期数，对于建立时间检查默认为1，保持时间检查默认为0；

*语句中数字的含义：

- 对于-setup：表示该多周期路径建立时间所需要的时钟周期个数；
- 对于-hold：表示相对于缺省捕获沿（图中的Default hold）,实际捕获沿（图中的New hold）**应回调的时钟周期个数**；

*参考时钟周期的选取：

- -end表示参考时钟为捕获端（收端）所用时钟，对于-setup缺省为-end；
- -start表示参考时钟为发送端（发端）所用时钟，对于-hold缺省为-start；

Multicycles in Single Clock Domain 单时钟域的多周期路径

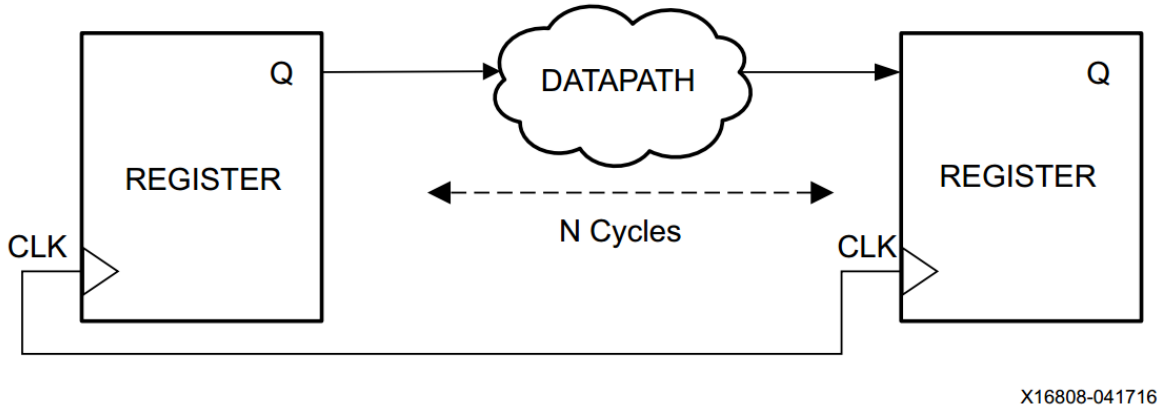


Figure 5-2: **Multicycle Constraint in Single Clock Domain**

如上图所示，两个寄存器之间的数据路径为N个周期，默认情况下的建立和保持时间检查如下所示：

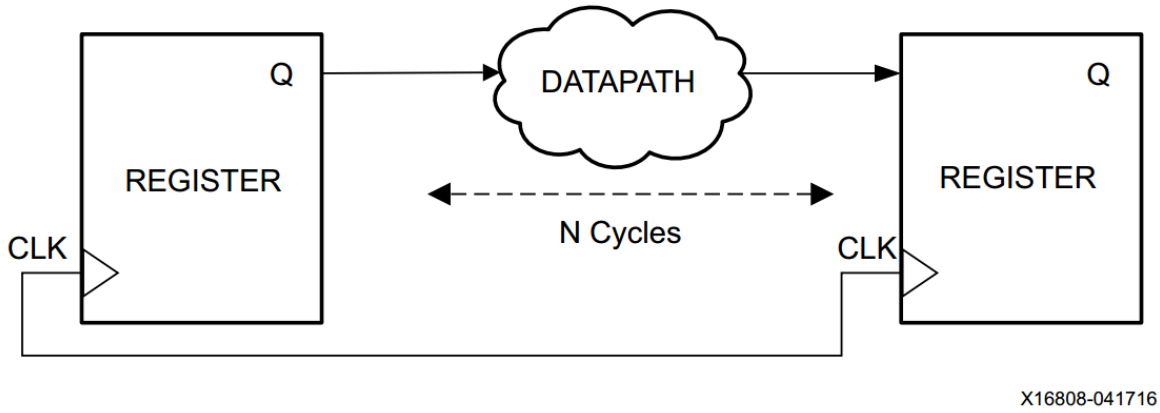


Figure 5-2: **Multicycle Constraint in Single Clock Domain**

例1.Relaxing Setup While Maintaining Hold 放宽建立检查维持保持检查

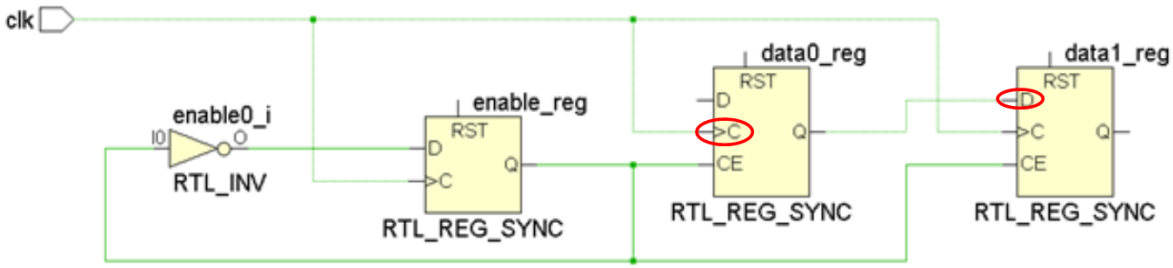
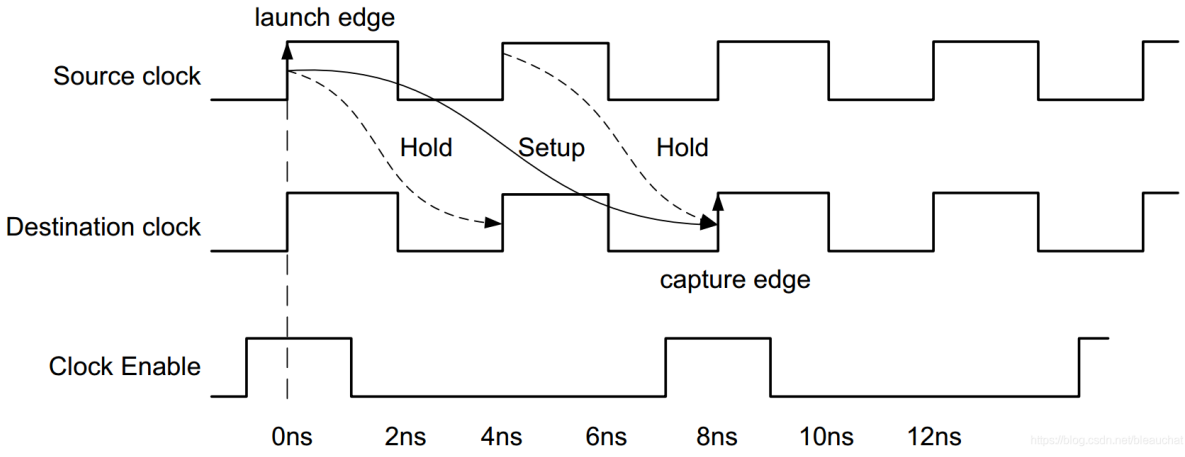


Figure 5-4: **Registers Enabled Every Two Cycles**

上图中两个寄存器之间的数据路径周期数为2，因此下面的语句设定了一个新的建立检查关系：

```
set_multicycle_path 2 -setup -from [get_pins data0_reg/C] -to [get_pins
data1_reg/D]
```

新的建立检查关系如下所示：



即默认的保持时间检查总是建立时间检查往前推一个周期

但是很显然，此时的保持时间检查是不对的，它应该在最开始的位置，即上图中的发起沿的位置，所以要用到第二条多周期路径约束：

```
set_multicycle_path 1 -hold -end -from [get_pins data0_reg/C] \
-to [get_pins data1_reg/D] #保持检查往前移动1个周期
# end表示参考时钟为捕获端时钟，这里对默认进行了修改，因为保持检查的默认时钟为发起端时钟；
# 当然，这里由于是单一的时钟域，两个时钟是等同的，所以 end为选择项
```

对于建立路径周期数为4，多周期路径约束为：

```
set_multicycle_path 4 -setup -from [get_pins data0_reg/C] -to [get_pins
data1_reg/D]
set_multicycle_path 3 -hold -from [get_pins data0_reg/C] -to [get_pins
data1_reg/D]
```

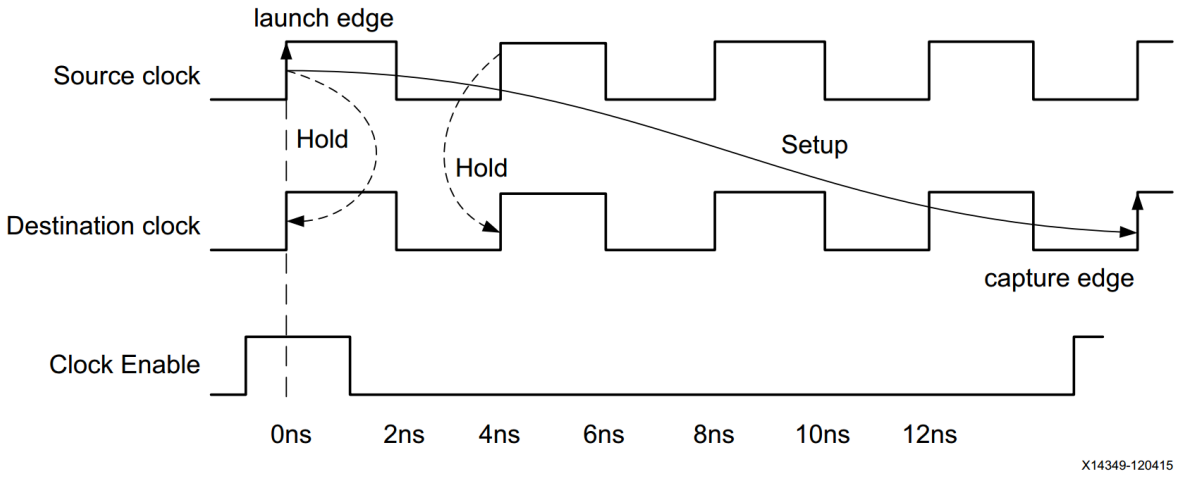


Figure 5-7: **Multicycle Path with Setup Multiplier of Four (4)**

一般情况下， 对于建立路径周期数为N，保持时间建立周期数为N-1，多周期路径约束为：

```
set_multicycle_path N -setup -from [get_pins data0_reg/C] -to [get_pins
data1_reg/D]
set_multicycle_path N-1 -hold -from [get_pins data0_reg/C] -to [get_pins
data1_reg/D]
```

Multicycle Paths and Clock Phase-Shift 时钟相位偏移中的多周期路径

一些情况下，必须对两个具有相同周期但是带相位偏移时钟进行时序约束；这种情况下，最重要的理解时序引擎在默认的情况下设定的建立保持关系，如果不做适当的调整就会导致在两个时钟域之间产生过度约束，情形如下图所示：

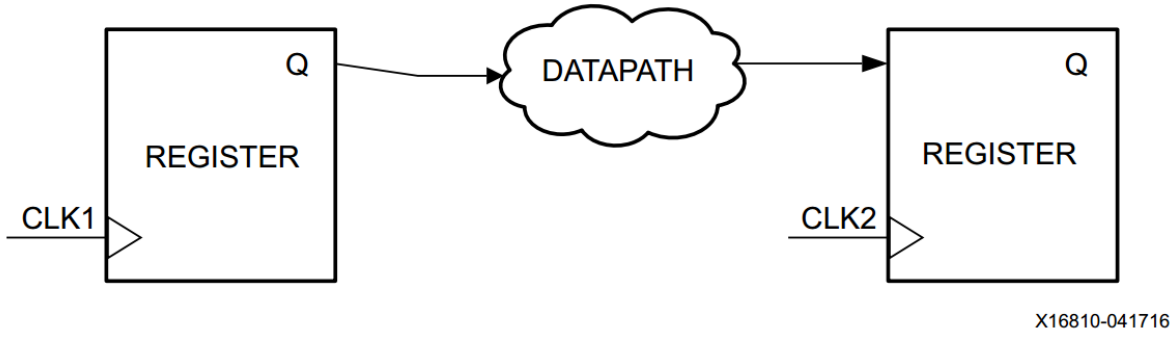
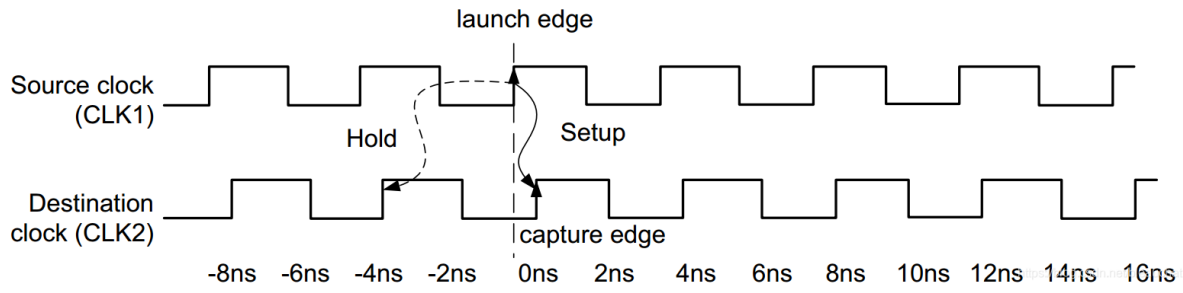


Figure 5-11: **Multicycle Paths and Clock Phase-Shift**

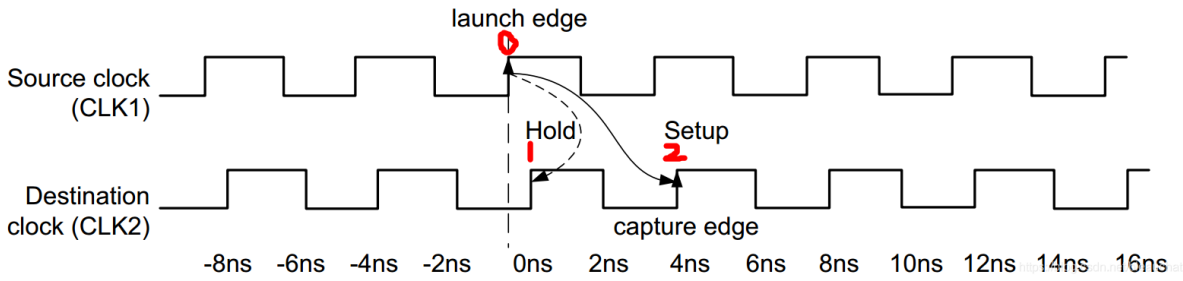
默认情况下不带多周期路径约束的建立保持关系：



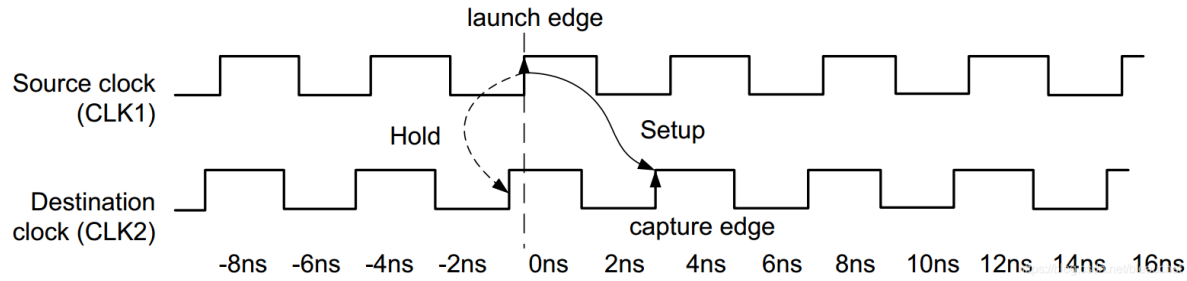
上图中捕获时钟相对于发起时钟发生了0.3ns的相移，这导致按照单周期时序检查的情况时序根本无法收敛；这时，必须调整建立检查时间和保持检查时间，可用如下的约束：

```
set_multicycle_path 2 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
# 数字量为1时为默认的单周期约束，即相对于发起时钟沿往前1个捕获沿
```

这导致相对于建立需求时间的 捕获沿往后移动了一个周期，保持时间会由建立时间得到默认值，如下图所示：



对于捕获时钟反向偏移相位的情况，如下图所示：



如果偏移量较小，不需要设置多周期路径约束来权衡偏移量；但是如果偏移量较大，此时需要调整发起沿或者捕获沿来满足建立保持需求；

Multicycles Between SLOW-to-FAST Clocks 慢时钟到快时钟的多周期路径

如下图所示，clk2的频率是clk1频率的3倍：

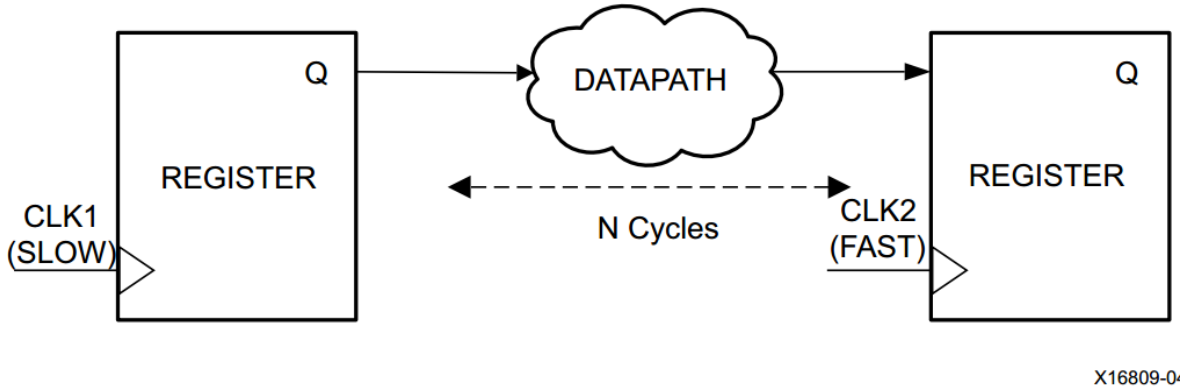
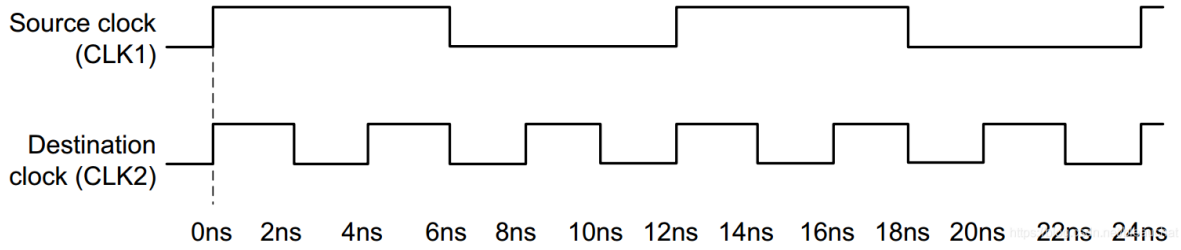
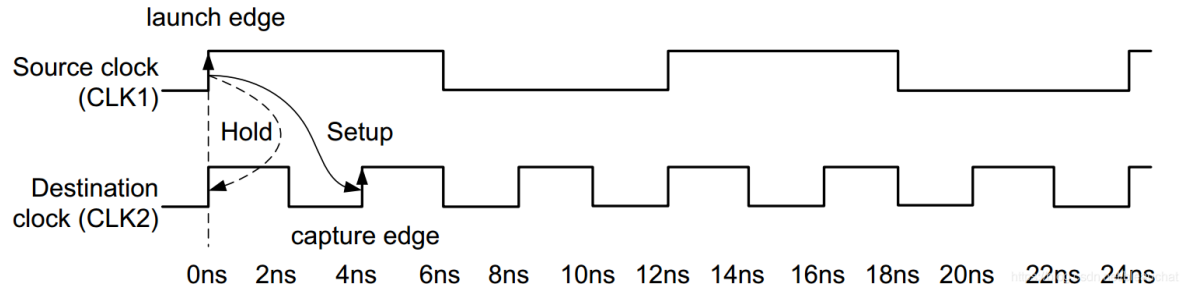


Figure 5-15: **Multicycles Between SLOW-to-FAST Clocks**

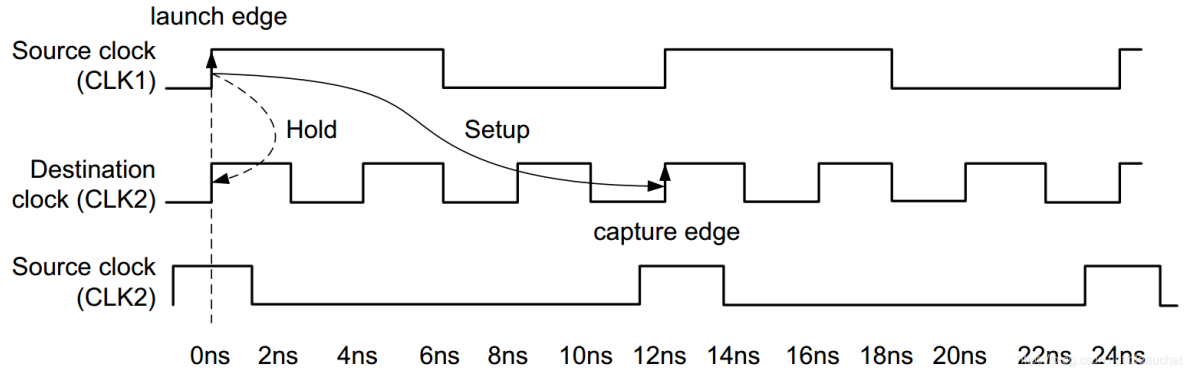


默认的建立保持关系如下图：



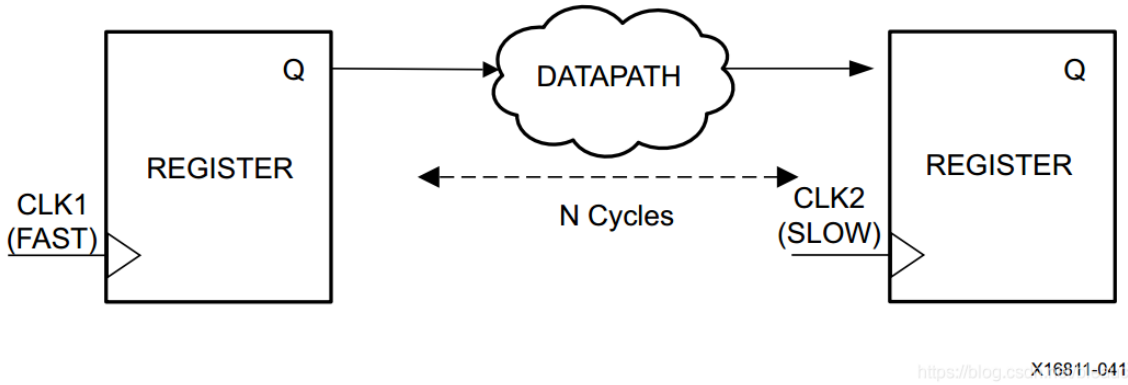
可以看出，捕获沿总是在于发起时钟对齐后的下一个上升沿；需要设置的多周期路径约束为：

```
set_multicycle_path 3 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
set_multicycle_path 2 -hold -end -from [get_clocks CLK1] -to [get_clocks CLK2]
# 保持检查默认的参考时钟为发起时钟，这里加入了end,表明了参考时钟为捕获时间，即相对于捕获沿往前回调2个周期
```

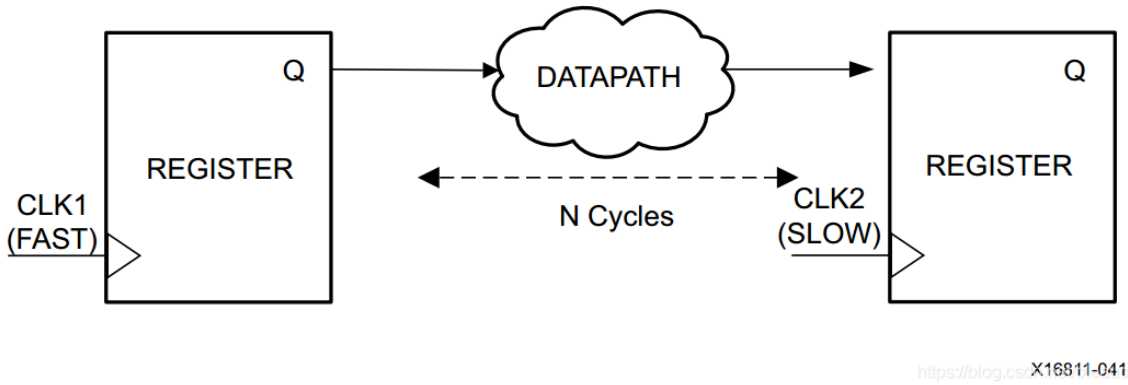


Multicycles Between FAST-to-SLOW Clocks 快时钟到慢时钟的多周期路径

如下图所示，clk1的频率是clk2频率的3倍：

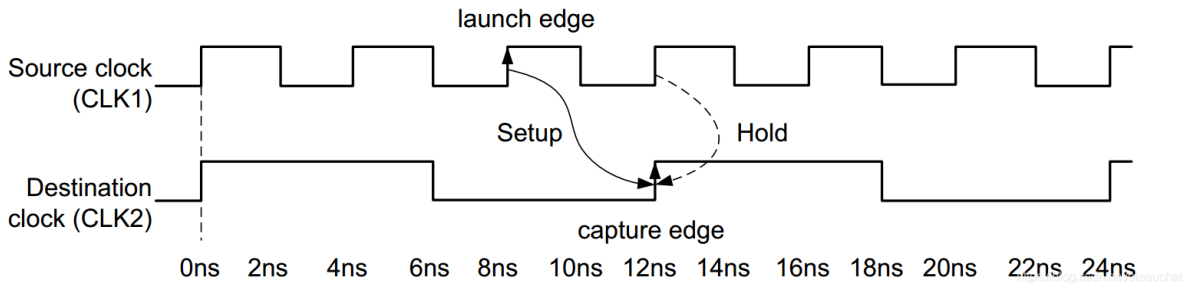


X16811-041716



X16811-041716

默认的建立保持关系如下图：



需要设置的多周期路径约束为：

```
set_multicycle_path 3 -setup -start -from [get_clocks CLK1] -to [get_clocks
CLK2]
set_multicycle_path 2 -hold -from [get_clocks CLK1] -to [get_clocks CLK2]
# 这里的参考时钟为发送时钟，所以在建立时间检查的约束中加入start
```

下表是对以上几种多周期路径约束语法的总结：

Table 5-3: **To define a multicycle path with a Setup of N**

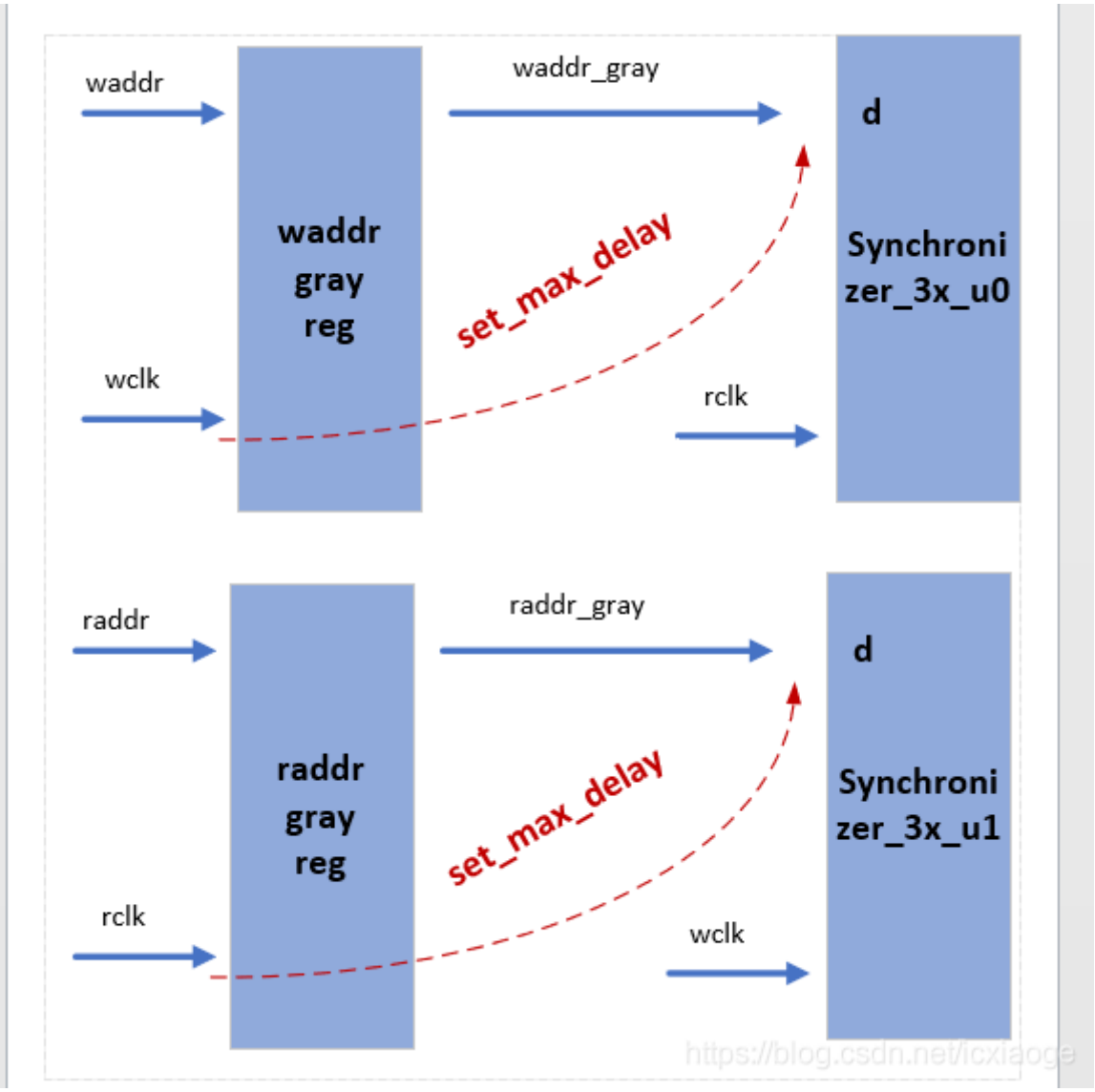
Scenario	Multicycle Constraints
Same clock domain or between synchronous clock domains with same period and no phase-shift	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -from CLK1 -to CLK2
Between SLOW-to FAST synchronous clock domains	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -end -from CLK1 -to CLK2
Between FAST-to SLOW synchronous clock domains	set_multicycle_path N -setup -start -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -from CLK1 -to CLK2

Note: The `get_clocks` command has been omitted in Table 5-3 to simplify the expressions.

异步FIFO中的格雷码设置max_delay

1、异步fifo中格雷码约束

为了保证异步fifo的功能和性能保证，需要在综合约束文件sdc中，约束异步FIFO格雷码的最大延时。约束如图所示：从格雷码寄存器的时钟端口---->到3级同步器的输入端口的最大延时。写地址waddr和读地址raddr格雷码同步都需要设置set_max_delay，延时可设置为读写时钟中最快时钟周期的一半。



set_max_delay [expr 0.5*$period_fast_clk] -from [get_pins "详细路径1/waddr_gray_reg_*/clocked_on"] -to [get_pins "详细路径1"/synchronizer_3x_u0/同步器件名称/d"]

set_max_delay [expr 0.5*$period_fast_clk] -from [get_pins "详细路径2/raddr_gray_reg_*/clocked_on"] -to [get_pins "详细路径2"/synchronizer_3x_u1/同步器件名称/d"]

NOTE:路径填写的是模块例化名称，同步器_u*不能丢。

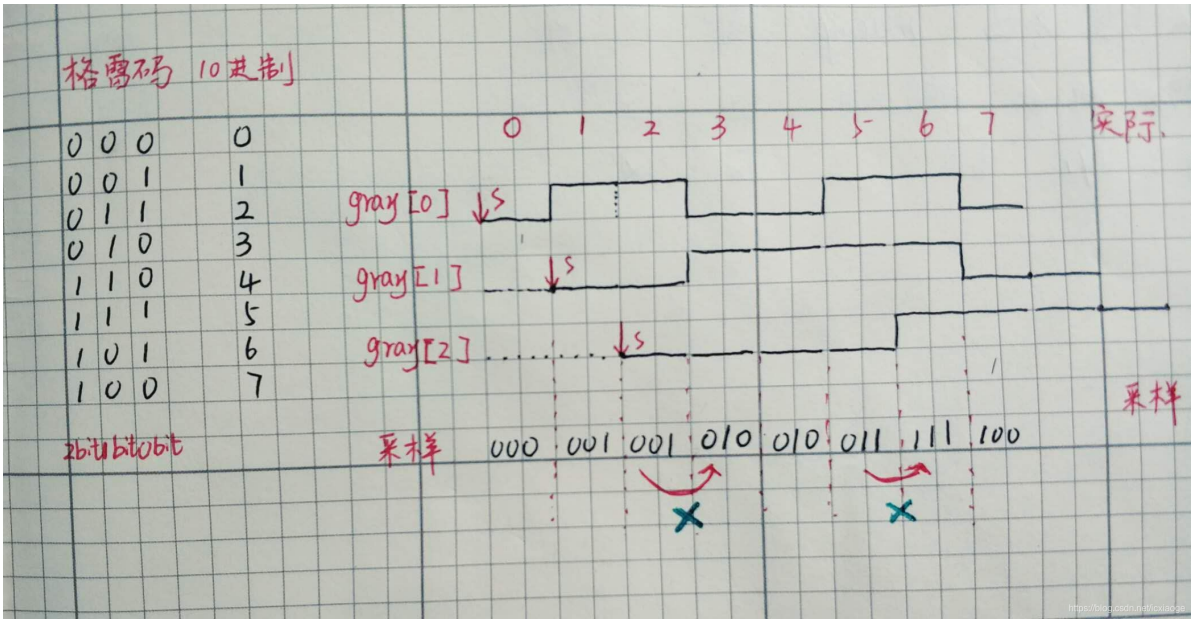
2、为什么要设置读写地址格雷码的max_delay?

如果不设置读写地址格雷码约束：会出现以下两种情况：

- 格雷码各bit位延时不一致---导致同步器采样的地址不符合gray规律，afifo功能异常。
- 格雷码到同步器的延时有好多个周期----异步afifo性能下降

情况1：格雷码各bit位延时不一致

格雷码各bit位延时不一致，导致afifo功能异常



假设3bit的gray码各比特位延时不一致，比如gray[1]延时比gray[0]多一个采样周期，比如gray[2]延时比gray[1]多一个采样周期，如图所示同步器syn_3x采样端的数据入口处的波形。

虽然源端格雷码是符合要求的，但是由于格雷码延时不一致，导致采样端采样的格雷码不符合要求，如图所示：采样后的格雷码由001（1）跳转到了010（3），又由011（2）跳转到了111（5），这会造成溢出和读空信号有效，afifo功能异常。

情况2：格雷码到同步器的延时有好多个周期

格雷码到同步器的延时越长，流水间隔越大，afifo性能越差。

假如afifo深度为16，写地址waddr_gray码到同步器的延时为8个周期，加上同步器3个周期，写数据侧写入数据后，至少需要11个read_clk后读数据侧empty信号无效，也就是说至少11个read_clk后读侧才能读数据。而如果写地址waddr_gray码到同步器的延时为1个周期，则写数据侧写入数据后，只需要4个周期，读侧就能读数据了。