Xilinx时序约束笔记

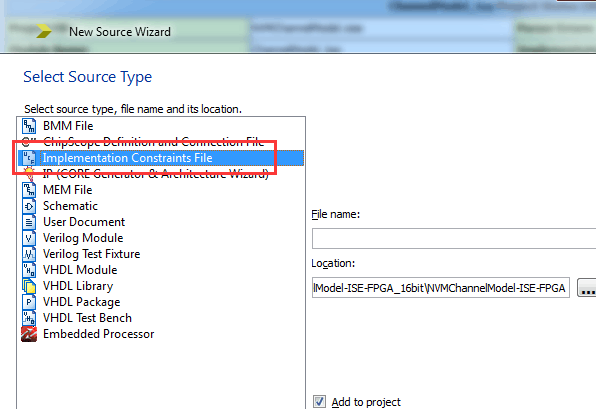
# 用户文件

FPGA设计中的约束文件有3类：用户设计文件（.UCF文件）、网表约束文件（.NCF文件）以及物理约束文件（.PCF文件），可以完成时序约束、管脚约束以及区域约束。**3类约束文件的关系为：用户在设计输入阶段编写UCF文件，然后UCF文件和设计综合后生成NCF文件，最后再经过实现后生成PCF 文件。**

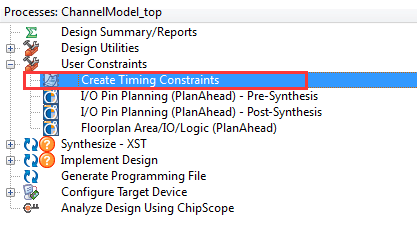
UCF文件是ASC 2码文件，描述了逻辑设计的约束，可以用文本编辑器和Xilinx约束文件编辑器进行编辑。NCF约束文件的语法和UCF文件相同，二者的区别在于： UCF文件由用户输入，NCF文件由综合工具自动生成，当二者发生冲突时，以**UCF**文件为准，这是因为UCF的优先级最高。PCF文件可以分为两个部分：一部分是映射产生的物理约束，另一部分是用户输入的约束，同样用户约束输入的优先级最高。一般情况下，用户约束都应在UCF文件中完成，不建议直接修改 NCF文件和PCF文件。

约束文件的后缀是.ucf，所以一般也被称为UCF文件。创建约束文件有两种方法，一种是通过新建方式，另一种则是利用过程管理器来完成。

第一种方法：新建一个源文件，在代码类型中选取“Implementation Constrains File”，在“File Name”中输入约束文件的名称。单击“Next”按键进入模块选择对话框，选择要约束的模块，然后单击“Next”进入下一页，再单击“Finish”按键完成约束文件的创建。



第二种方法：在工程管理区中，将“Source for”设置为“Synthesis/Implementation”。“Constrains Editor”是一个专用的约束文件编辑器，双击过程管理区中“User Constrains”下的“Create Timing Constrains”就可以打开“Constrains Editor”。



需要注意的是，UCF文件是大小敏感的，端口名称必须和源代码中的名字一致，且端口名字不能和关键字一样。但是关键字NET是不区分大小写的。

# UCF文件的语法说明：

## 1．语法

UCF文件的语法为：

{NET|INST|PIN} "signal\_name" Attribute;

其中，“signal\_name”是指所约束对象的名字，包含了对象所在层次的描述；“Attribute”为约束的具体描述；语句必须以分号“；”结束。可以用“#”或“”添加注释。需要注意的是：UCF文件是大小写敏感的，信号名必须和设计中保持大小写一致，但约束的关键字可以是大写、小写甚至大小写混合。例如：

NET "CLK" LOC = P30;

“CLK”就是所约束信号名，LOC = P30；是约束具体的含义，将CLK信号分配到FPGA的P30管脚上。

对于所有的约束文件，使用与约束关键字或设计环境保留字相同的信号名会产生错误信息，除非将其用" "括起来，因此在输入约束文件时，最好用" "将所有的信号名括起来。

## 2．通配符

在UCF文件中，通配符指的是“\*”和“?”。“\*”可以代表任何字符串以及空，“?”则代表一个字符。在编辑约束文件时，使用通配符可以快速选择一组信号，当然这些信号都要包含部分共有的字符串。例如：

NET "\*CLK?" FAST;

将包含“CLK”字符并以一个字符结尾的所有信号，并提高了其速率。

在位置约束中，可以在行号和列号中使用通配符。例如：

INST "/CLK\_logic/\*" LOC = CLB\_r\*c7;

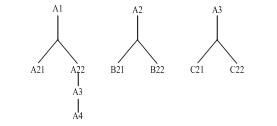
把CLK\_logic层次中所有的实例放在第7列的CLB中。

## 3．定义设计层次

在UCF文件中，通过通配符\*可以指定信号的设计层次。其语法规则为：  
\* 遍历所有层次

Level1/\* 遍历level1及以下层次中的模块

Level1/\*/ 遍历level1中的模块，但不遍历更低层的模块例4-5 根据图4-75所示的结构，使用通配符遍历表4-3所要求的各个模块。

[](http://www.openhw.org/data/11-10/19200027021056/1319682536_8c19ba69.jpg)

# 管脚和区域约束语法（或使用Floorplanner约束）

LOC约束是FPGA设计中最基本的**布局约束和综合约束**，能够定义基本设计单元在FPGA芯片中的位置，**可实现绝对定位、范围定位以及区域定位**。此外， LOC还能将一组基本单元约束在特定区域之中。LOC语句既可以书写在约束文件中，也可以直接添加到设计文件中。换句话说，**ISE中的FPGA底层工具编辑器（FPGA Editor）、布局规划器（Floorplanner）和引脚和区域约束编辑器的主要功能都可以通过LOC语句完成。**

LOC语句语法

INST "instance\_name " LOC = location;

其中“location”可以是FPGA芯片中任一或多个合法位置。如果为多个定位，需要用逗号“,”隔开，如下所示：

LOC = location1,location2,...,locationx;

目前，还不支持将多个逻辑置于同一位置以及将多个逻辑至于多个位置上。需要说明的是，多位置约束并不是将设计定位到所有的位置上，而是在布局布线过程中，布局器任意挑选其中的一个作为最终的布局位置。

范围定位的语法为：

INST “instance\_name” LOC=location:location [SOFT];

**表-常用的LOC定位语句**



使用LOC完成端口定义时，其语法如下：

NET "Top\_Module\_PORT" LOC = "Chip\_Port";

其中，“Top\_Module\_PORT”为用户设计中顶层模块的信号端口，“Chip\_Port”为FPGA芯片的管脚名。

**LOC语句中是存在优先级的，当同时指定LOC端口和其端口连线时，对其连线约束的优先级是最高的。例如，在下图中，LOC=11的优先级高于LOC=38。**

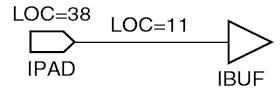
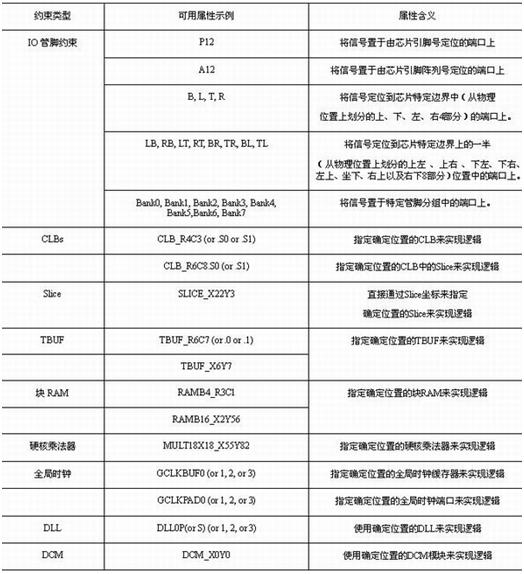
[](http://www.openhw.org/data/11-10/19200027021056/1319682511_6a0cc95e.jpg)

图 LOC优先级示意图

## LOC属性说明

LOC语句通过加载不同的属性可以约束管脚位置、CLB、Slice、TBUF、块RAM、硬核乘法器、全局时钟、数字锁相环（DLL）以及DCM模块等资源，基本涵盖了FPGA芯片中所有类型的资源。由此可见，LOC语句功能十分强大，下面列出了LOC的常用属性。

表-LOC语句常用属性列表



# 时序约束的语法（或使用constraint editor约束）

UCF文件中时序约束的语法

约束UCF文件，从**Constrains Editor直接输入是最方便、最直接的添加约束的方法了**。以下是几种常用的语法：

## 周期约束（PERIOD）

PERIOD约束是一个基本时序和综合约束，它附加在时钟网线上，时序分析工具根据PERIOD约束检查时钟域内所有同步元件的时序是否满足要求，它将检查与同步时序约束端口相连接的所有路径的延迟，但是不会检查PAD到寄存器的路径。

**附加时钟周期约束的首选方法（Preferred Method）语法如下：**

**TIMESPEC “TS\_identifier” = PERIOD “TNM\_reference” period {HIGH|LOW} [high\_or\_low\_time]**

其中“[ ]”内为可选项，“{ }”为必选项，参数period为要求的时钟周期，可以使用ps、ns、us或者ms等单位，大小写都可以，缺省单位为ns。HIGH|LOW关键词指出时钟周期里的第一个脉冲是高电平还是低电平，而high\_or\_low\_time为脉冲的延续时间，缺省单位也是ns，如果不提供该项，则缺省占空比为50%。

**TIMESPEC是一个基本时序相关约束标识**，表示本约束为时序规范。TSidentifier包括字母TS和一个标识符identifier（为ASCII码字符串）共同组成一个时序规范。

例如定义时钟周期约束时，首先在时钟网线clk上附加一个TNM\_NET约束，把clk驱动的所有同步元件定义为一个名为sys\_clk的分组，然后使用TIMESPEC约束定义时钟周期。

NET “clk” TNM\_NET=”sys\_clk”;

#定义clk驱动的所有同步元件为sys\_clk的分组

TIMESPEC “TS\_sys\_clk”= PERIOD “sys\_clk” 50 HIGH 30;

#定义可引用的时序规范TS\_sys\_clk，

#这个规范规定sys\_clk组的时钟情况

而定义派生时钟的语法如下：

TIMESPEC “TSidentifier\_2”=PERIOD “timegroup\_name” “TSidentifier\_1” [\*or/] factor PHASE [+|-] phase\_value [units];

#定义第二个时序规范TSidentifier\_2

#其内容是名为timegroup\_name的分组是第一个时序规范TSidentifier\_1派生的

其中TSidentifier\_2为要定义的派生时钟，TSidentifier\_1为已经定义的时钟，factor指出两者周期的倍数关系，是一个浮点数。phase\_value指出两者之间的相位关系，为浮点数。例如：

定义主时钟clk0：

TIMESPEC “TS01” = PERIOD “clk0” 10.0 ns;

定义派生时钟clk180，其相位与主时钟相差180°：

TIMESPEC “TS02” = PERIOD “clk180” TS01 PHASE + 5.0 ns;

定义派生时钟clk180\_2,其周期为主时钟的1/2，并延迟2.5ns：

TIMESPEC “TS03” = PERIOD “clk180\_2” TS01 /2 PHASE + 2.5 ns;

## 偏移约束(OFFSET)

**偏移约束规定了外部时钟和数据输入输出引脚之间的时序关系，只用于与PAD相连的（端口）信号，不能用于内部信号**。使用该约束可以为综合实现工具指出输入数据到达的时刻，或者输出数据稳定的时刻，从而在综合实现中调整布局布线过程，使正在开发的FPGA/CPLD的输入建立时间以及下一级电路的输入建立时间满足要求。

基本语法如下：

OFFSET = {IN|OUT} “offset\_time” [units] {BEFORE|AFTER} “clk\_name” [TIMEGRP “group\_name”];

其中{IN|OUT}说明约束的是输入还是输出，offset\_time为FPGA引脚数据变化与有效时钟沿之间的时间差，BEFORE|AFTER说明该时间差在有效时钟沿的前面还是后面，TIMEGRP “group\_name”定义了约束的触发器组，缺省时约束该时钟驱动的所有触发器。

## 分组约束

使用TNM（Timing Name）约束可以选出构成一个分组的元件，并赋予一个名字，以便给它们附加约束。TNM\_NET（timing name for nets）约束只加在网线上，其作用与TNM加在网上时基本相同，即把该网线所在路径上的所有有效同步元件作为命名组的一部分。不同之处在于当TNM约束加在PAD NET上时，TNM的值将被赋予PAD，而不是该网线所在的路径上的同步元件，即TNM约束不能穿过IBUF。而用TNM\_NET约束就不会出现这种情况。

## 专门约束

附加约束的一般策略是首先附加整体约束，例如PERIOD、OFFSET等，然后对局部的电路附加专门约束，这些专门约束通常比整体约束宽松，通过在可能的地方尽量放松约束可以提高布线通过率，减小布局布线的时间。

FROM\_TO约束在两个组之间定义时序约束，对两者之间的逻辑和布线延迟进行控制，这两个组可以是用户定义的，也可以是与定义的。用户可以使用TNM\_NET、TNM和TIMEGRP定义组，而与定义组主要包括FFS、LATCHES、PADS和RAMS等。语法如下：

TIMESPEC “TSname” = FROM “group1” TO “group2” value;

其中value为延迟时间，可以使具体数值或表达式。

MAXDELAY约束定义了特定网线上的最大延迟，其语法如下：

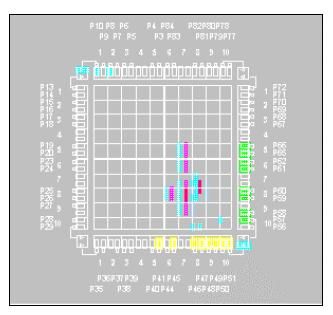
NET “net\_name” MAXDELAY = value units;

# 时序约束概念和效果(特权)

执行工具不会试图寻找达到最快速的布局&布线路径。——取而代之的是，执行工具会努力达到你所期望的性能要求。

性能要求和时序约束相关——时许约束通过将逻辑元件放置的更近一些以缩短布线资源从而改善设计性能。

**没有时序约束的例子**

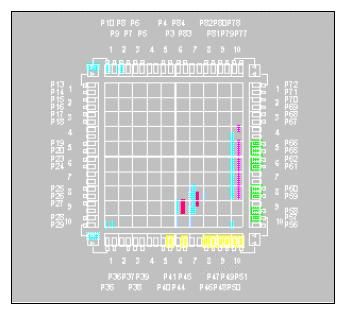


该工程没有时序约束和管脚分配

——注意它的管脚和放置

——该设计的系统时钟频率能够跑到50M

**添加时序约束的例子**



  和上面是相同的一个设计，但是加入了3个全局时序约束。

——它最高能跑到60M的系统时钟频率

——注意它**大部分的逻辑的布局更靠近器件边沿其相应管脚的位置**

## 更多关于时序约束

         时序约束应该用于界定设计的性能目标

1.       太紧的约束将会延长编译时间

2.       不现实的约束可能导致执行工具罢工

3.       查看综合报告或者映射后静态时序报告以决定你的约束是否现实

执行后，查看布局布线后静态时序报告以决定是否你的性能要求达到了——如果约束要求没有达到，查看时序报告寻找原因。

## 路径终点

         有两种类型的路径终点：

1.       I/O pads

2.       同步单元（触发器，锁存器，RAMs）

时序约束的两个步骤：

1.       路径终点生产groups（顾名思义就是进行分组）

2.       指点不同groups之间的时序要求

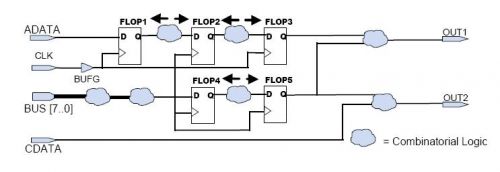
全局约束使用默认的路径终点groups——即所有的触发器、I/O pads等

**问题思考**

单一的全局约束可以覆盖多延时路径

如果箭头是待约束路径，那么什么是路径终点呢？

所有的寄存器是否有一些共同点呢？



**问题解答**

什么是路径终点呢？

         ——FLOP1,FLOP2,FLOP3,FLOP4,FLOP5。

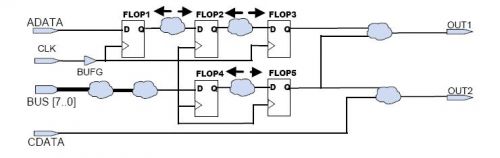
         所有的寄存器是否有一些共同点呢？

         ——它们共享一个时钟信号，约束这个网络的时序可以同时覆盖约束这些相关寄存器间的延时路径。

## 周期约束

周期约束覆盖由**参与网络钟控的的同步单元之间的路径延时**。

         周期约束不覆盖的路径有：input pads到output pads之间的路径（纯组合逻辑路径），input pads到同步单元之间的路径，同步单元到output pads之间的路径。



周期约束**使用最准确的时序信息**，使其能够自动的计算：

1.       源寄存器和目的寄存器之间的时钟偏斜（Clock Skew）

2.       负沿钟控的同步单元

3.       不等同占空比的时钟

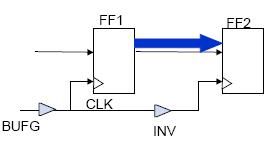
4.       时钟的输入抖动（jitter）

假设：

1.       CLK信号占空比为50%

2.       周期约束为10ns

3.       由于FF2将在CLK的下降沿触发，两个触发器之间的路径实际上将被约束为10ns的50%即5ns



### 时钟输入抖动（Clock Input Jitter）

         时钟输入抖动是源时钟的不确定性（clock uncertainty）之一

         时钟的不确定时间必须从以下路径扣除：

         ——周期约束建立时间路径

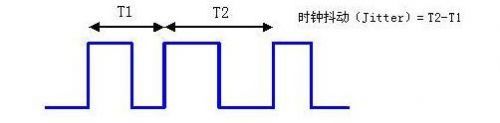
         ——OFFSET IN约束的建立时间路径

         时钟的不确定时间必须添加到以下路径中：

         ——周期约束保持时间路径

         ——OFFSET IN约束保持时间路径

         ——OFFSET OUT约束路径



## Pad-to-Pad约束

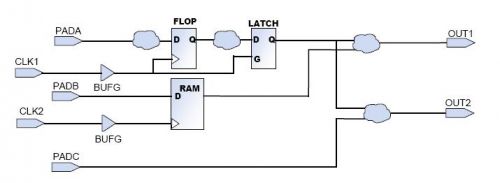
         ——不包含**任何同步单元的纯组合逻辑电路**

         ——**纯组合逻辑延时路径开始并结束于I/O pads**，所以通常会被我们遗漏而未约束

**问题思考**

         哪些路径是由CLK1进行周期约束？

         哪些路径是由pad-to-pad进行约束？

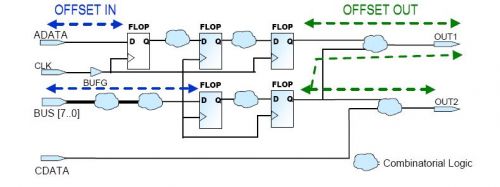


**OFFSET约束**

         OFFSET约束覆盖以下路径：

——从input pads到同步单元（OFFSET IN）

——从同步单元到output pads（OFFSET OUT）



### OFFSET约束特性

 OFFSET约束自动计算时钟分布延时

1.       提供最准确的时序信息

2.       大量增加输入信号到达同步单元的时间（时钟和数据路径并行）

3.       大量减少输出信号到达输出管脚的时间（时钟和数据路径先后）

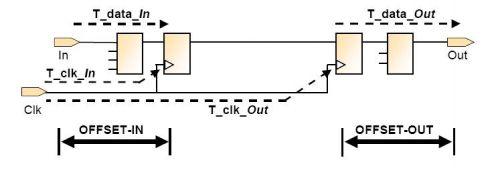
OFFSET约束也可以解释时钟输入抖动——使用抖动确定关联的周期约束

### 时钟延时

         数据路径延时和时钟分布延时都需要在OFFSET计算中使用到

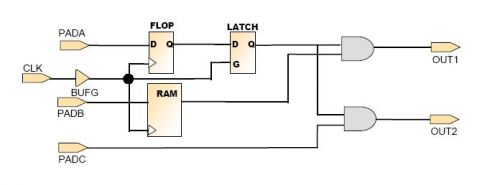
         ——OFFSET IN = T\_data\_in – T\_clk\_in

         ——OFFSET OUT = T\_data\_out + T\_clk\_out



**问题思考**

         在这个电路中哪些路径是由OFFSET IN 和 OFFSET OUT来约束的？



**问题解答：**

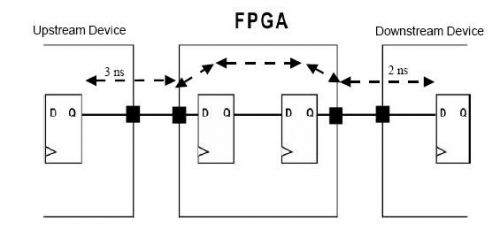
         ——OFFSET IN：PADA to FLOP and PADB to RAM

——OFFSET OUT：LATCH to OUT1, LATCH to OUT2, and RAM to OUT1

**问题思考**

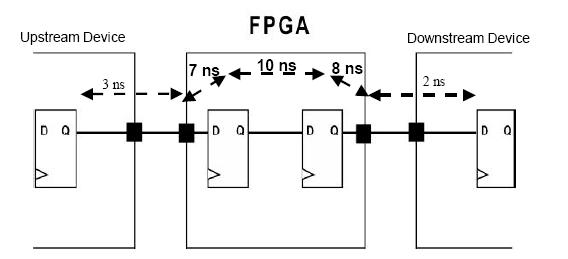
         下面给出的系统框图里，你将给出什么样的约束值以使系统能够跑到100MHz？

         ——假设在下面的器件之间没有时钟偏斜



**问题解答：**

         PERIOD = 10 ns , OFFSET IN (BEFORE) = 7 ns and OFFSET OUT (AFTER) = 8 ns



**小结**

1.       性能期望和时序约束相关联

2.      **周期约束覆盖同步单元之间的延时路径**

3.       OFFSET约束覆盖从输入管脚到同步单元和从同步单元到输出管脚之间的延时路径

## 3. 特定路径时序约束

         使用全局时序约束（PERIOD,OFFSET,PAD-TO-PDA）将约束整个设计

         仅仅使用全局约束通常会导致过约束

         ——约束过紧

——编译时间延长并且可能阻止实现时序目标

——通过综合工具或者映射后时序报告重新审视性能评估

**特定路径约束能够覆盖全局时序约束在特定路径上的约束**

——这就允许设计者放宽特定路径的时序要求

### 更多关于特定路径约束

         你的设计器件的内部面积将会从特定路径约束收益

1.       多周期路径Multi-cycle paths

2.       跨时钟域路径

3.       双向总线

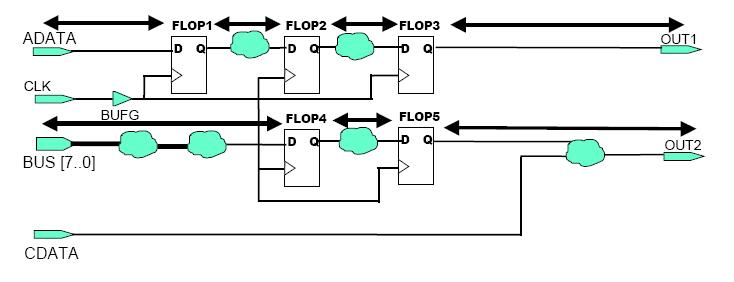
4.       I/O时序

特定路径约束应该由你的性能目标来界定，不能够不加限制的随意放置

## 全局约束回顾

         使用全局PERIOD,OFFSET IN和OFFSET OUT约束将约束所有以下的路径

         这使得控制设计的总体性能更加容易



## 特定路径约束实例

         一条特定路径约束对于路径本身的优化微乎其微

         这有助于你更好的控制设计性能，并带给执行工具更大的灵活性以达到你的性能和使用要求

         生成特定路径约束需要两个步骤：

1.       多个有共同时序要求的特定路径终点生成一个groups

2.       关联两个groups，指定它们的特定路径的时序要求

## 生成终点路径的Groups

**特定路径时序约束在终点路径较好的分组后会更加高效——否则，约束一个大的工程将极其耗时耗力。**

         约束编辑有助于你更容易的进行路径终点（pads, flip-flops, latches, and RAMs）进行Groups分组。

使用约束编辑器，终点路径的分组有以下选项：

– Group by nets

– Group by instance name

– Group by hierarchy

– Group by output net name

– Timing THRU Points option

– Group by clock edge

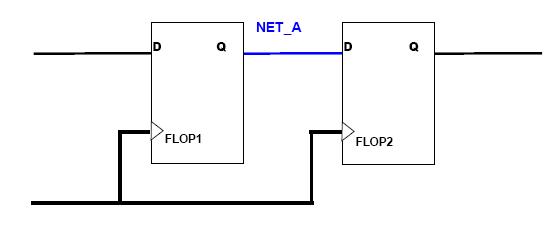
## Nets 分组与output net name分组对比

由net分组的 “NET\_A”将生成一个只包含FLOP2的group

——Group包含选择网络所驱动的寄存器

由output net name分组的“NET\_A”将生成一个只包含FLOP1的group

——Group包含选择网络的源寄存器



## 回顾全局OFFSET约束

在时钟行中使用Pad-to-Setup和Clock-to-Pad列为所有出于该时钟域的I/O路径指定OFFSETs。为大多数I/O路径进行约束的最简单方法——然而，这将会导致一个过约束的设计。

## 指定管脚的OFFSET约束

         使用Pad-to-Setup和Clock-to-Pad列为每个I/O路径指定OFFSETs。

         这种约束方法适用于只有少数管脚需要不同的时序约束。

         更常用的方法是：

1.       为Pads生成Groups

2.       对生成的指定Groups进行OFFSET IN/OUT约束

## 双沿时钟的OFFSET约束

         OFFSET约束指明了FPGA管脚的输入数据和初始时钟之间的关系。

         初始时钟沿在周期约束定义中出现关键词“高”和“低”。

                  ——高：初始时钟上升沿（默认），即上升沿锁存数据

                  ——低：初始时钟下降沿

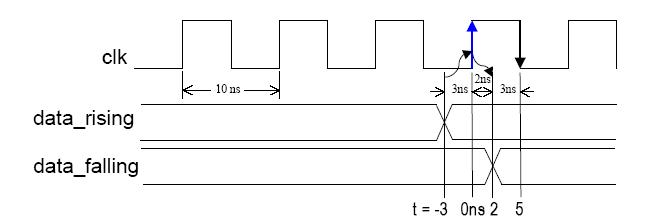
         如果所有的I/O都由时钟的一个沿控制，那么你可以使用这个关键字高或低进行周期约束。

如果两个沿都用到，你就必须进行两个OFFSET的约束。

                  ——每个OFFSET对应一个时钟沿

                  ——DDR寄存器也是这样使用的一个例子

## 双沿时钟的OFFSET IN约束



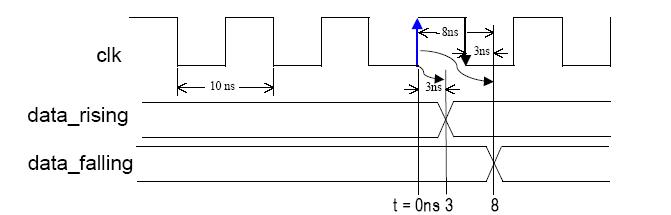
输入数据在上升沿或者下降沿之前3ns有效——周期约束为10ns，初始上升沿，占空比为50%。

         为每个时钟沿生成一个时钟Groups

                  ——输入时钟的上升沿，OFFSET = IN 3ns BEFORE CLK;

                  ——输入时钟的下降沿，OFFSET = IN -2ns BEFORE CLK;（在初始时钟的上升沿后2ns = 时钟下降沿前3ns）

## 双沿时钟的OFFSET OUT约束



输出数据必须在时钟的上升沿或者下降沿后3ns内有效——周期约束为10ns，初始上升沿，占空比为50%。

         为每个时钟沿生成一个时钟Groups

                  ——输入时钟的上升沿，OFFSET = OUT 3ns AFTER CLK;

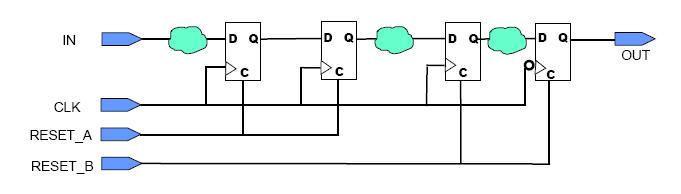
                  ——输入时钟的下降沿，OFFSET = OUT 8ns AFTER CLK;（在初始时钟的上升沿后8ns = 时钟下降沿后3ns）

**问题思考**

         特定路径时序约束如何改善了设计性能？

你如何约束这个设计使其内部时钟频率达到100 MHz？

输入（数据）将在时钟CLK的上升沿到达前3ns内有效。输出数据必须在时钟CLK的下降沿后4ns内稳定下来。写出合适的OFFSET约束？



**问题解答**

         特定路径时序约束如何改善了设计性能？

                  ——它使得执行工具更加灵活的达到你的时序要求。

你如何约束这个设计使其内部时钟频率达到100 MHz？

——给时钟信号CLK施加一个10ns的全局周期约束。

写出合适的OFFSET约束？

OFFSET = IN 3 ns BEFORE CLK;

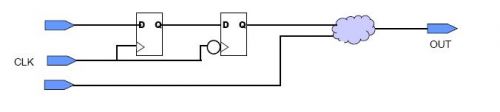
OFFSET = OUT 4 ns AFTER CLK;

## 时钟上升沿和下降沿之间的时序约束

         周期约束可以自动计算两个沿的约束——包括调整非50%占空比的时钟。

         例：一个CLK时钟周期约束为10ns，能够应用5ns的约束到两个寄存器之间。

         不需要特定路径应用到这个例子中。

[](http://maxim.eefocus.com/data/11-09/1316679158_fca764fb.jpg)

## 相关时钟域的约束

**为一个时钟进行周期约束——以这个周期约束确定相关的时钟。**

         执行工具将根据它们的关系来决定如何处理跨时钟域。

         DCM有多个输出：

                   ——确定DCM输入时钟的周期约束

                   ——执行工具将会从这个周期约束推导出其输出的约束

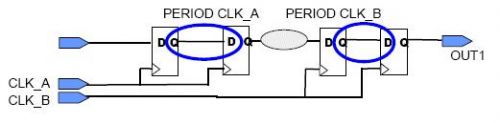
                   ——所有的约束将会和原始的周期约束相关

## 不相关时钟域的约束

         在这个例子中，周期约束不覆盖到处于两个时钟域之间的任何延时路径。——这是默认的处理方式。

         你必须添加一个约束覆盖到相关时钟域之间的路径中。——例如，频率相同，但是CLK\_B有一些相位偏移。

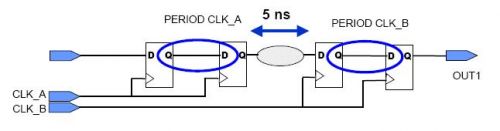
**在两个不相关的时钟域你就必须添加一些同步电路。**

[](http://maxim.eefocus.com/data/11-09/1316679161_4d62f931.jpg)

 约束两个时钟域之间的路径。

         ——使用Groups by NETs选项为CLK\_A和CLK\_B定义groups，如果你为每个时钟添加完周期约束，这个步骤将自动完成。

         ——在这个寄存器的groups之间指定快速/慢速例外约束。

[](http://maxim.eefocus.com/data/11-09/1316679165_7e14d7be.jpg)

## 多周期路径约束

         多周期约束应用在连续几个时钟周期内寄存器不需要更新的情况。

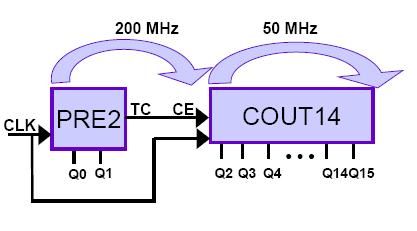
                   ——总是至少需要一个时钟周期才更新。

                   ——通常的，这样的寄存器由时钟使能信号控制。

         一个分段计数器就是这样的一个例子。

                   ——COUT14每隔4个时钟周期才更新一次。

                   ——这些寄存器间的路径就算是多周期路径。

[](http://maxim.eefocus.com/data/11-09/1316679168_e73f0f07.jpg)

## False 路径

         False路径选项将用于防止约束覆盖到特定路径

## 时序约束优先级

         从高到低为：

1.       False路径——将会覆盖任何其它的约束路径

2.       FROM THRU TO

3.       FROM TO

4.       管脚指定OFFSETs

5.       Groups OFFSETs（由寄存器或者PADS生产的groups）

6.       全局PERIOD和OFFSETs——最低优先级约束

这里特权同学提醒大家注意的是，通常类似下面这样的计数器绝对不可以归为多周期约束：

         reg[15:0] counter;

         always @ (posedge clk or negedge rst\_n)

         begin

                   if(!rst\_n) counter <= 16’d0;

                   else counter <= counter+1’b1;

         end

虽然我们想想似乎counter[1]也是2个clk变化一次，counter[2]也是4个clk变化一次……但是，我们想想看，如果从counter=1到counter=2没有在一个clk完成，那么肯定就会影响到counter=2到counter=3的变化，对吧？所以，这样的计数器不能算做多周期约束例外。

提纲里描述的多周期例外的计数器应该是这样一个模型：

         reg[15:0] counter;

         always @ (posedge clk or negedge rst\_n)

         begin

                   if(!rst\_n) counter[1:0] <= 2’d0;

                   else counter[1:0] <= counter[1:0]+1’b1;

         end

         always @ (posedge clk or negedge rst\_n)

         begin

                   if(!rst\_n) counter[15:2] <= 14’d0;

                   else if(counter[1:0] == 2’b11) counter[15:2] <= counter[15:2]+1’b1;

         end

         上面两个always块里的数据互不干扰，并且都正常工作，只有下一个always块检测到前一个always块里的counter[1:0]==2’b11时才进位加1。

写到这里，特权发现单从功能上来说，这两个例子是没有差别的，说白了，任何一个计数器都可以建模成后面的形式。或者说，我的问题其实没有说明白，反而被自己的例子给驳倒了。

呵呵，换个角度思考这个问题，到底什么时候是多周期例外，什么时候不是？这个其实还是要看情况的，如果你的系统实时性较高，可能会在某一特定的时钟周期用到16位计数器的计数值（如a = (counter == 16’hffff)?1b’1:1’b0; ），那么这个计数器的高位就不能算作多周期例外。而如果比如在我的一个工程里，有这样的计数器用法：cuonter[2:0]没16个clk的后8个clk需要自增加（从0到15），而高位counter[18:3]当然只有在16个clk变化一次，因为这里counter是用于作为一个地址产生器，也就是说，我的地址是每16个clk的后8个clk用到，那么这里的counter[18:3]就是一个8clk的多周期例外实例。

         说白了，还是要具体问题具体分析。

# 总结

题记：achieving timing closure即达到时序收敛，这是一个很具有挑战性的任务。因为实际的工程项目往往不会像我们用一个资源超大（相对于你的设计来说）的FPGA来做几个数码管串口实验那么简单。设计者往往需要达到成本、速度、资源等各个方面的平衡，即使是一个小设计，有时候也是很费神的。特权同学前几周在饱经ISE4里才有的老器件的折磨后，感慨良多。

## 关于时序报告

**ISE中的时序报告分为两种：**

**– Post-Map Static Timing Report**

**– Post-Place & Route Static Timing Report**

**所谓Post-Map是布局后（没有布线）的静态时序报告，主要用于估计设计的性能，然后提前对设计做一些必要的修改。因为设计的实现（布局布线）是很消耗时间的。Post-Place & Route就是布局布线后的一个比较接近实际板级的一个静态时序报告了，这算是设计者进行时序分析的最终依据。**

## 关于性能估计

**综合报告**

1.       准确的逻辑延时；

2.       基于扇出的布线延时估计

3.       报告的性能是实际的20%误差内

**Post-Map静态时序报告**

1.       准确的逻辑延时

2.       基于最快的可能的布线资源的布线延时估计

3.       使用了60/40规则来计算更趋近于实际的性能估计

**60/40法则：**

1.       这是一个时序约束合理性的经验法则；

2.       打开Post-Map静态时序报告，查看时序报告中关于逻辑延时的百分比

——低于60%，时序很有机会到达时序约束要求

——60%到80%，如果使用了高级选项，时序也很有机会达到时序要求

——80%以上，基本上很难（回到综合部分，或者重新优化你的代码）

**Post-Place & Route时序分析**

         找出时序违规的因素有很多：

1.       设计的综合不当或者代码风格太烂；

2.       糟糕的综合结果（路径中的逻辑太繁杂）

3.       不准确或者不完整的时序约束

4.       糟糕的逻辑映射和布局

每个问题都用不同的解决方案：

1.       重写代码

2.       添加时序约束（注意应该是一些时序例外）

3.       使用不同的软件选项重新综合或者实现

准确的定位时序报告能够解决大多数问题。

**一些可能的问题和解决**

布线延时太长——似乎有些路径扇出很低，但是延时却很大，那么很可能这个地方的布线比较拥堵。解决办法：如果是不相关的逻辑布局到一块，可以到Floorplanner中查看。（这个问题特权同学还没有完全领会也是比较头疼的，希望看到更多更好的资料或者自己在工程中有更多体会时再和大家分享）

         高扇出问题——解决办法是复制高扇出的网络。如果是组合逻辑，那么就比较难了。

         逻辑级数太高——这个问题综合工具无法做太多优化。首先查看是否该路径为多周期路径，如果是，添加多周期例外；使用retiming选项更加均匀的分配触发器之间的逻辑；确定一个比较好的代码技巧被运用到了你的设计代码中；使用流水线设计。

         I/O时序问题——使用DCM移除时钟分布延时； 将输入输出相关的寄存器放入IOB寄存器中。

另外在实际应用中，其实很是有很多可以应用的技巧的，比如实现属性选项里其实是可以设置布局布线的努力程度，还有布局布线的次数等待，对于大多数设计而言这些工具都是有用的。

# Xilinx Timing Analyzer使用札记

虽说当初刚接触FPGA的时候学的是ISE，但是真正深入学习还是在Quartus II平台上。现在项目需要，又得重新来玩转ISE了，虽说QII和ISE很多东西是相通的，不过还是有那么点操作上的差异在里头。用惯了TimeQuest的特权同学折腾起这个Xilinx Timing Analyzer也是一愣一愣的，好不适应。

         花了点时间在Help上，还好这个Help的说明还是蛮到位的。列几个常用的技巧或者说是简单的操作以及特权同学遇到的问题在这里和大家一起学习和探讨。

1.       四种时序分析查看方式。在同等条件下，其实这四种查看方式得到的时序报告结果是一致的，因为它是基于一个时序结果进行的。如图1所示，工具栏上的这四个时序分析按钮分别代表了（从左到右）：

|  |  |
| --- | --- |
| 序号 | 功能 |
| 1 | 根据时序约束产生的时序分析报告。 |
| 2 | 自动产生的时序分析报告。在同等的时序约束条件下，那么该报告的结果肯定和1的结果是大同小异的。 |
| 3 | 用户指定的路径的时序报告，主要是路径端点的时序报告。 |
| 4 | 用户指定的路径的时序报告，主要是时钟或者IO的时序报告。 |

打开Xilinx Floorplanner或Xilinx FPGA Editor后，在详细时序报告中点击任何一条路径或者net等，这里点击了Data Path:dsp\_addr<10> to uut dspc/Mtridata dsp\_out\_dbr\_5（如图2），相应的可以看到其在Floorplanner（如图3）或FPGA Editor中的位置（如图4）。

http://maxim.eefocus.com/data/11-09/1316679383_f40aa98f.jpg

图1

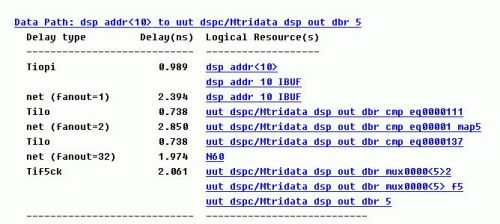
[](http://maxim.eefocus.com/data/11-09/1316679388_b870d0bc.jpg)

图2

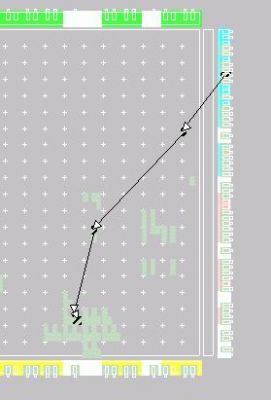
[](http://maxim.eefocus.com/data/11-09/1316679406_58c0e619.jpg)

图3

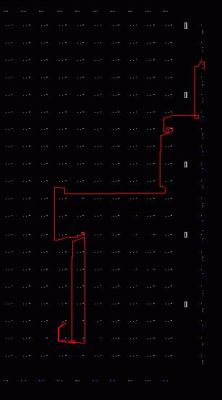
[](http://maxim.eefocus.com/data/11-09/1316679403_af4e138b.jpg)

图4

在查看unconstrained path ayalysis时，遇到了一些问题，clk是FPGA片外晶振输入的时钟，从PAD输入后直接连到DLL，而不直接钟控任何寄存器。但是在unconstrained path报告中却出现了clk到内部所有寄存器的路径。详细的查看这些路径，发现这些路径无外乎是内部DLL输出的两个使用到的时钟的路径。如图5和图6所示。

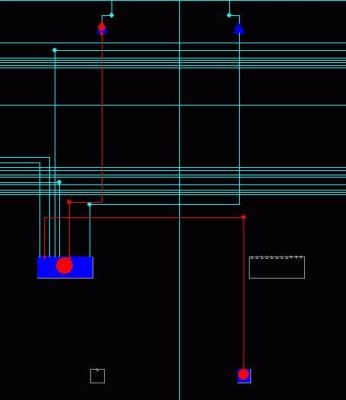
[](http://maxim.eefocus.com/data/11-09/1316679412_481a614b.jpg)

图5

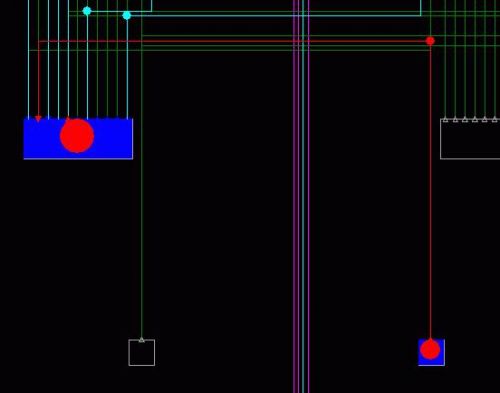
[](http://maxim.eefocus.com/data/11-09/1316679423_93853969.jpg)

图6

         而特权同学希望添加一个clk到all FFT的Flase path，但是

**TIMESPEC "TS\_CLK" = FROM "clk" TO "FFS"  TIG;**

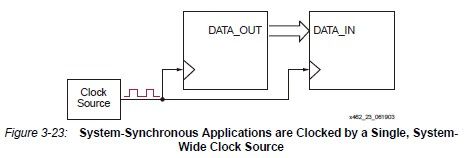
这条语句约束得到的是约束工具的报错信息，因为之前有clk的时钟约束：

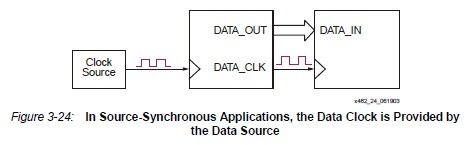
**NET "clk" TNM\_NET = "clk";**

**TIMESPEC "TS\_clk" = PERIOD "clk" 40 ns HIGH 50 %;**

         不知道他们之间是否有冲突，虽然最后的unconstrained报告大可不必介意，但是这个false路径总应该有个约束的办法，只是特权同学现在还没有搞定其中的问题所在。

# System Synchronous & Source Synchronous





# Clock

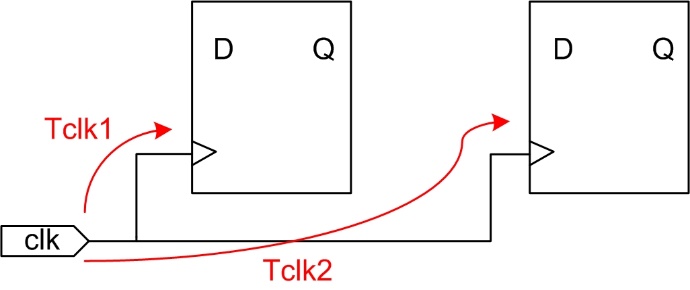
时钟是数字电路的动力系统，可以说数字电路中最重要的信号就是时钟信号了。一般时钟信号的时序特性分为：

1. 偏移 Skew
2. 抖动 Jitter
3. 占空比时钟 Duty Cycle Distortion

对于低速设计，基本不用考虑这些特征，但是对于高速设计，由于时钟本身造成的问题越来越普遍，因此有必要关注高速设计中的时序特性。

## Skew(时钟偏移)

时钟信号要提供给整个电路的时序单元，所以时钟信号线非常长，并构成分布式的RC网路。它的延时与时钟线的长度、时序单元的负载电容、个数有关，由于时钟线长度及负载不同，会导致时钟信号到达相邻两个时序单元的时间不同，这个时间上的偏差就是 时钟偏移Skew。如下图所示：



假设时钟信号达到两个 DFF 的延时分别为 Tc1 和 Tc2，用 Tskew 来表示它们之间的时钟偏移，则计算公式如下：

Tskew = Tc2 - Tc1

根据差值可以分为正偏移和负偏移：

当时钟到达 DFF1 的延时更大时，也就是 C1 > C2 时，Tskew 为正

当时钟到达 DFF2 的延时更大时，也就是 C1 < C2 时，Tskew 为负

需要注意到是，时钟偏移永远存在，当其大到一定程度时，就会严重影响电路的时序。

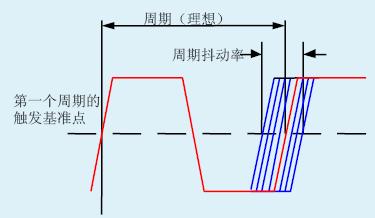
FPGA 在设计架构时，专门针对这种现象进行优化，采用全铜工艺和树状结构，**并且设计了专用的时钟缓冲和驱动网络，这么做的目的就是尽量使时钟到达不同时序单元的路径一样长，从而使时钟偏移非常小**，可以忽略不计。

所以，**Skew 问题的解决方法就是：设计中的主要信号应该走全局时钟网络。**

即使采用了这样的设计，在实际电路中，时钟信号到达每个 DFF 的时间也不可能完全相等，Skew 是肯定存在的。所以 STA 仍然需要考虑该因素。在 PAR 之前，STA 只能根据设计的面积来粗略估计 Skew，在 PAR 之后，因为有了更具体的信息（线段长度、宽度、信号分布情况）STA 的估计值更加精确。

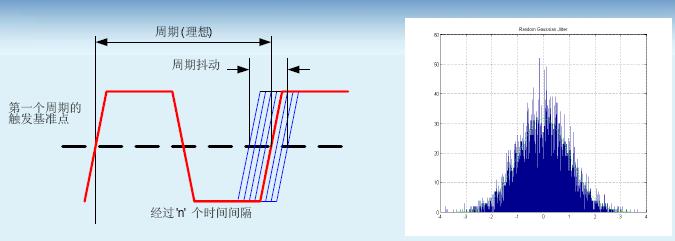
## Jitter(时钟抖动)

理想的时钟信号是方波，但是实际中的时钟信号边沿不可能是瞬间变化的，是个斜坡，如下图所示：



时钟抖动 Jitter 的定义很多，最常见的有 3 种：

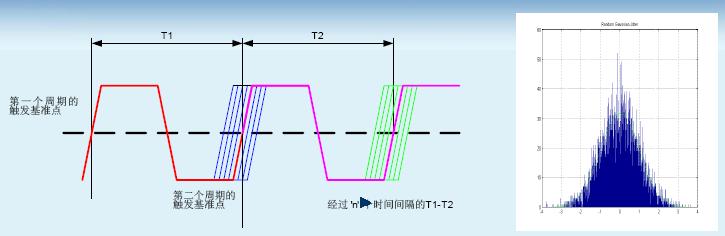
### 周期抖动 Period Jitter



**实际时钟信号周期与理想时钟周期**的差值的变化。这是最早最直接的一种衡量抖动的方式，这个指标说明了时钟信号每个周期的变化。

**因为这个差值是个随机变量，并且满足高斯分布，所以可以用期望和方差来描述。一般随机选择很多个周期，然后计算平均周期、标准差、峰峰值。标准差称为 “RMS 抖动”，峰峰值称为 “Pk-Pk 周期抖动”。知道 Pk-Pk 周期抖动，对于恰当配置系统和保持时间很有用。**

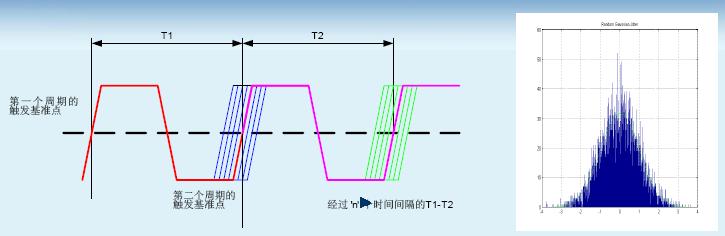
### 周期差抖动 Cycle-to-cycle Jitter



**两个相邻时钟周期的差值**的变化。根据定义可知，对周期抖动做一阶差分，就可以得到周期差抖动。

这个差值也是一个服从高斯分布的随机变量。

* 相位抖动 Phase Jitter



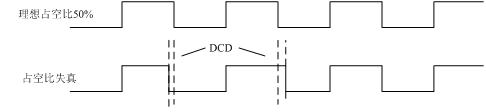
**一个时钟沿相对于基准对齐之后，经过一段时间后，与理想位置的偏差**。这个指标说明了周期抖动在各个时期的累计效应。

因为需要累积一段时间，所以这个误差又称为 时间间隔误差（TIE, Timer Interval Error）。

由于周期抖动和周期差抖动是单个周期或者相邻周期的偏差，所以表征为短期抖动行为。而相位抖动需要累积一段时间，所以表征为长期抖动行为。

时钟抖动的原因就是噪声。**时钟抖动是永远存在的**，当其大到可以和时钟周期相比拟的时候，会影响到设计，这样的抖动是不可接受的。

### 占空比失真Duty Cycle Distortion



占空比失真，即时钟不对称，有脉冲的时间和无脉冲的时间发生了变化。DCD 会吞噬大量的时序裕量，造成数字信号的失真，使过零区间偏离理想的位置。DCD通常是由信号的上升沿和下降沿之间时序不同而造成的。

# Others

除去时钟信号，还有一些其他的相关定义。

## Fan-out/Fan-out

### **Fan-out**

在数字电路中，逻辑门相互连接，组成更加复杂的电路，所以大多数逻辑门的**输出端都连接着多个别的单元的输入**。所以需要一个术语来描述逻辑门的驱动能力的大小，也就是扇出 Fan-out。最大扇出数 maximum fan-out 定义为一个逻辑门可以驱动的同类逻辑门的最大数。

大多数 TTL 逻辑门能够为 10 个其他数字门或驱动器提供信号。因而，一个典型的 TTL 逻辑门有 10 个扇出信号。在一些数字系统中，必须有一个单一的 TTL 逻辑门来驱动 10 个以上的其他门或驱动器。这种情况下，被称为缓冲器（buf）的驱动器可以用在 TTL 逻辑门与它必须驱动的多重驱动器之间。这种类型的缓冲器有 25 至 30 个扇出信号。逻辑反向器（非门）在大多数数字电路中能够辅助这一功能。

**模块的扇出** 是指模块的直属下层模块的个数。一般认为，设计得好的系统平均扇出是 3 或 4。一个模块的扇出数过大或过小都不理想，过大比过小更严重。一般认为扇出的上限不超过 7。扇出过大意味着管理模块过于复杂，需要控制和协调过多的下级。解决的办法是适当增加中间层次。一个模块的扇入是指有多少个上级模块调用它。扇入越大，表示该模块被更多的上级模块共享。这当然是我们所希望的。但是不能为了获得高扇入而不惜代价，例如把彼此无关的功能凑在一起构成一个模块，虽然扇入数高了，但这样的模块内聚程度必然低。这是我们应避免的。

设计得好的系统，上层模块有较高的扇出，下层模块有较高的扇入。其结构图像清真寺的塔，上面尖，中间宽，下面小。

### **Fan-in**

与扇出相对的概念是 扇入 Fan-in，**它描述的是一个逻辑门能够处理的外部输入的能力**。扇入大的逻辑门的速度要比扇入小的慢，原因是增加扇入相当于增加逻辑门的输入电容。我们可以用使用多级逻辑门来代替高扇入的设计。

## Setup/Hold/Recovery/Removal Time

建立/保持时间是在同步设计中的概念：

* 建立时间 setup time ：触发器在时钟信号上升沿到来以前，要求输入数据必须保持稳定不变一段时间，这段时间就是器件需要的建立时间。如不满足 setup time，这个数据就不能被这一时钟打入触发器。
* 保持时间 hold time ：触发器在时钟信号上升沿到来以后，要求数据保持稳定不变一段时间，以便能够稳定读取，这段时间就是器件需要的保持时间。如果不满足 hold time，数据同样不能被打入触发器。

恢复/撤销时间是在异步设计中的概念：

* 恢复时间 recovery time : 对于异步信号（比如异步复位/置位），信号变无效的边沿和下一个时钟沿之间必须满足一个最小的间隔。其意义在于，如果保证不了这个最小时间，也就是异步信号无效边离时钟边沿太近了，异步信号解除（无效）之后，没有给 DFF 足够的时间来恢复（recovery）到正常状态，那么就不能保证在时钟沿到来时 DFF 可以正常工作。
* 撤销时间 removal time : 对于异步信号（比如异步复位/置位），信号变有效的边沿和前一个时钟沿之间必须满足一个最小的间隔。其意义在于，如果保证不了这个最小时间，也就是异步信号的有效沿离时钟太近了，在时钟信号去除（无效）之前，异步信号提前有效了，可能会造成 DFF 处于不确定状态。

# STA静态时序分析

STA 中的一些术语定义如下：

## timing path

[FPGA STA(三) --- STA的基本概念](http://blog.sina.cn/dpool/blog/s/blog_72c14a3d01013tpi.html?type=-1)中说的很明白：

在做 STA 时，首先要把电路分解为一条条的 timing path。实际上我们也可以把 timing path 称为 data path，其本质就是指信号传播的途径。每一条 timing path 都具有一个起始点和一个终点。起始点是指电路中信号被时钟沿锁存的点；而信号经过一系列的组合逻辑的通道或者走线后被另外一个时钟沿捕获，这个点被称为终点。信号从起始点到终点所经过的通道就被称为 timing path。

起点有两种：

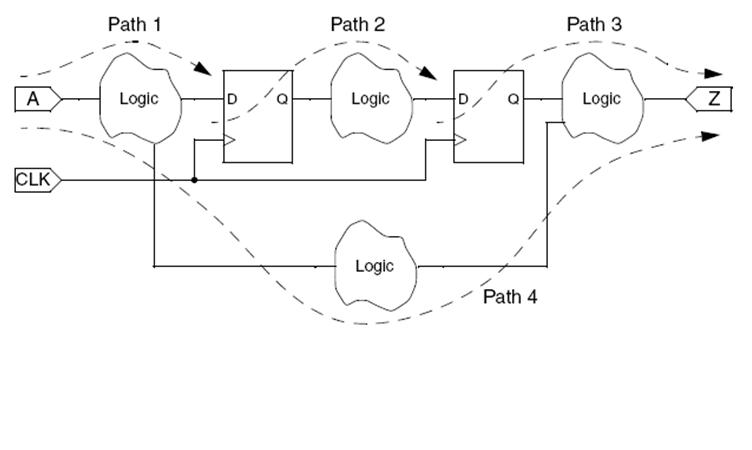
* 时序器件的 时钟输入端
* 电路的 输入端口

终点也有两种：

* 时序器件的 数据输入端
* 电路的 输出端口

输入和输出排列组合一共就有 4 种 path：

1. 电路输入端口 -> 触发器的数据D端 (Pad-to-Setup)
2. 触发器的clk端 -> 触发器的数据D端 (Clock-to-Setup)
3. 触发器的clk端 -> 电路输出端口 (Clock-to-Pad)
4. 电路输入端口 -> 电路输出端口 (Pad-to-Pad)



* critical path

关键路径：从输入到输出，延时最大的那条路径称为 critical path。关键路径是系统中延时最大的路径，它决定了系统所能达到的最大时钟频率。

* arrival time

到达时间：信号到达某个特定位置所消耗的时间。一般将时钟信号到达的时刻作为参考的 0 时刻，为了计算到达时间，需要对路径中的所有组件的延时都进行计算。

* required time

需求时间：所能容忍的路径最大延时，也就是信号到达的最晚的时间。如果路径上的延时再大一些，则必须降低时钟频率，否则会产生 setup/hold time violation。

* slack

时序余量：slack = required time - arrival time。如果计算出某条路径的 slack 是正数，说明这条路径的时延是满足要求的；如果计算出某条路径的 slack 是负数，则表示路径上的延时太大了，必须做出修改（修改设计 or 修改约束 or 换芯片），否则包含它的电路不能以预期的频率工作。

## Purpose

在同步设计中，数据的流动是统一步伐的，即时钟信号每改变一次，数据跟随改变一次。这种运作方式是基于同步器件（DFF 或者 Latch）来实现的，这类器件以时钟信号作为指示，将其输入端的数据复制到输出端。在同步设计中只存在两种时序错误：

* setup time violation

输入数据和时钟的关系不满足 setup time 的要求，即在时钟有效沿之前，输入数据没有保持稳定足够长的时间，数据将不能被这个时钟沿记录下来。

* hold time violation

输入数据和时钟的关系不满足 hold time 的要求，即在时钟有效沿之后，输入数据没有保持稳定足够长的时间，数据将不能被时钟信号记录下来。

导致数据和时钟不同步的原因很多，比如数据本身和时钟不同步、或者是电路进行了不同的操作，器件的温度、电压、制造工艺等因素也会产生影响。

**静态时序分析 STA 的主要目的是在上述可能的电路偏移情况存在的情况下，验证所有信号能够准时到达，并保证电路的正常功能。**

引自[Xilinx FPGA开发实用教程](http://book.douban.com/subject/11523088/)：

“工作频率对数字电路而言至关重要。提高工作频率意味着更强大的处理能力，但是也带来了时序瓶颈：时序冲突的概率变大，电路的稳定性降低。所以为了使电路的性能达到设计的预期目标，并满足电路工作环境的要求，必须对一个电路设计进行时序、面积、负载等多方面的约束，并自始至终使用这些约束来驱动EDA软件工作。

ISE 具有一定的自动优化能力，对于一般的低速设计（处理时钟不超过100MHz），基本上不需要时序方面的任何手动分析和处理；但是对于高速和大规模设计，需要设计人员自行添加时序方面的控制和处理，通过多次反复操作，根据反馈结果逐步调整设定，直到满足要求为止。

以前小规模FPGA设计，只需要做动态的门级时序仿真就课同时完成逻辑功能验证和时序验证；随着FPGA设计规模和速度的提升，有必要将逻辑功能验证和时序验证分开：首先，逻辑功能的正确性，可以通过RTL级或者门级的功能仿真来验证；其次，时序分析通过STA（Static Timing Analysis，静态时序分析）验证。

**时序分析的主要作用就是查看FPGA内部逻辑和布线的延迟，验证其是否满足设计者的约束。**

* **确定芯片最高工作频率**

控制工程的综合、映射、布局布线等关键环节，减少逻辑和布线的延迟，从而尽可能提高工作频率。一般情况下，处理时钟高于100MHz的时候，必须添加合理的时序约束文件。

* **检查时序约束是否满足**

检查目标模块是否满足约束，若不满足，通过时序分析器定位程序中不满足的部分，并给出具体原因，然后设计人员修改程序，直到满足约束。

* **分析时钟质量**

当采用了全局时钟等优质资源后，仍然不满足目标约束，则需要降低所约束的时钟频率。

* **确定分配引脚的特性**

通过时序分析可以指定I/O引脚所支持的接口标准、接口速率和其他电气特性。”

**STA的目的就是要保证 DUT（Device Under Test）中所有的路径满足内部时序单位对 setup time 和 hold time 的要求。信号可以及时的从任一时序路径的起点传递到终点，同时要求在电路正常工作所需的时间内保持恒定。**

## Theory

STA 是基于前面介绍的时序路径的，在分析时，计算时序路径上数据信号的到达时间和要求时间的差值，以判断是否存在违反设计规则的错误。

Slack = Trequired\_time - Tarrival\_time

如果时序裕量 Slack 为正，表示满足时序，负值表示不满足时序。STA 按照上式分析设计中所有路径，如果 Slack为负值，则该路径为影响设计的关键路径，需要修改设计以达到时序要求。

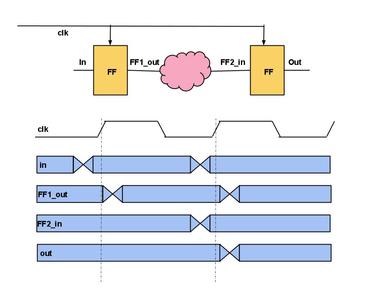
STA 是通过“穷举法”抽取整个设计电路的所有时序路径，按照约束条件分析电路中是否有违反设计规则的问题，并计算出设计的最高频率。

## 计算最大时钟频率的题

首先，电路中的元件一般分为两类：

1. **组合逻辑**，比如与非门、或非门等组合逻辑
2. **时序逻辑**，受时钟驱动的元件（一定要有时钟），比如 flip-flop

一般时序分析都是分析前面介绍的 Path2 的路径，如下图所示：



在这个图中，第一级的 DFF 的输出经过组合逻辑进入第二级 DFF，图中的时序是满足条件的，电路可以正常工作；但是如果违反 setup / hold time 的话，电路就无法正常工作，下面分别说明：

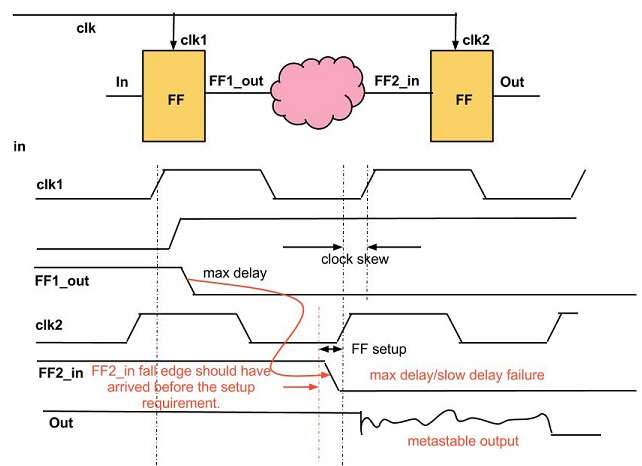
### Setup Time Failure

前面已经提到了，对于任何 DFF，都必须满足 setup time 的要求：**数据信号要在时钟信号之前到达 DFF，并保持稳定至少 setup time 时间，才能被成功打入 DFF**。

分类讨论一下：

1. 如果中间的组合逻辑较少，产生的时延很小，那么在下一个时钟沿到来之前，数据满足 setup time，没有问题。
2. 如果中间的组合逻辑较多，产生的时延足够大，导致数据和时钟的关系不满足 setup time，则有问题。

下图是一个 setup time failure 的具体例子：



第一级的 DFF 在时钟的上升沿对输入数据 In 进行采样，并经过一段时间（Tco，clock to output delay）后输出为 FF1\_out，输出数据经过中间的组合逻辑变为 FF2\_in 输入到第二级 DFF，因为中间的组合逻辑的时延太大，FF2\_in 违反了 setup time 的要求（图中橙色线条所示）。由于第二级 DFF 的输入不满足 setup time，所以这个 DFF 会进入亚稳态，它将花费 1 个或多个时钟周期才能脱离亚稳态，在这期间它的输出都是不确定的，那么下游逻辑将会在这期间采样到错误数值，电路会发生错误。

如果中间的组合逻辑过多，产生的时延过大，以至于 FF2\_in 的变化落在了第二个时钟沿的后面，这时所然没有违反 setup time 的要求，第二个 DFF 不会进入亚稳态，但是它采样的数据是旧数据，这个数据可能是错误的，同样会导致下游逻辑发生错误。

**如果我们将时钟频率降到足够低，那么就可以避免 setup time failure 的问题。**

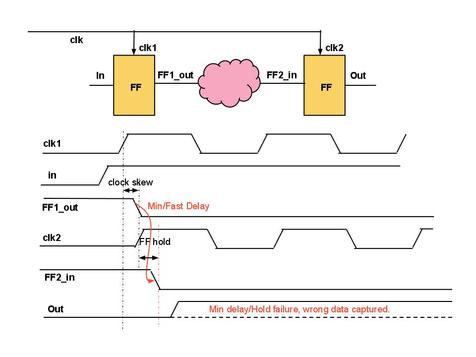
通过上面的分析，可以知道 setup time 限制了中间组合逻辑的最大时延，所以这个 setup time constraint 也叫做 max delay constraint。为了保证在最恶劣的情况下设计仍然能够满足要求，在计算 setup time 时中应该使用最大的数据时延路径。

P.S. 上面的图中还包含了 clock skew，clk1 和 clk2 没有对齐，这加剧了 setup time failure 的机率。在实际中，时钟信号到达各个 DFF 的时间是不同的，设计者必须将这个因素考虑在内。

### Hold Time Failure

和 setup time 对应，对于任何时序器件（DFF)，都必须满足 hold time 的要求：**数据信号要在时钟信号达到之后保持稳定至少 hold time 的时间，否则输出是错误**。

下图是一个 hold time failure 的具体例子：



第一级 DFF 在时钟的上升沿对输入数据 In 进行采样，经过一段时间（Tco）后，输出为 FF1\_out，输出数据经过中间的组合逻辑变为 FF2\_in，输入到第二级 DFF。因为中间的组合逻辑时延非常小，**而且由于 clock skew** 的原因，导致第二级 DFF 在第一个时钟周期的 hold time 内输入的 FF2\_in 发生了变化，违法了 hold time 要求，DFF2 进入亚稳态，它将花费 1 个或多个时钟周期才能退出亚稳态，在此期间输出的是错误数据。

在现实中，导**致这个问题的因素有可能是设计的问题（比如两个 DFF 邻接，中间没有其他逻辑），也有可能是器件的原因。**

因为**传输时延太小**，从而导致在第一个时钟沿信号就被 DFF2 采样了，而设计本意是在第二个时钟沿采样，看起来好像是信号跑（racing）的太快了，所以 hold time failure 也叫做 race。

通过上面的分析，可以知道 hold time 限制了中间组合逻辑的最小时延，所以这个 hold time constraint 也叫做 min delay constraint。为了保证在最恶劣的情况下设计仍然能够满足要求，**在计算 hold time 时中应该使用最小的数据时延路径。**

**P.S. 上面的例子中 clock skew 加剧了 hold time failure 的机率，如果两个时钟是对齐的，那么就不会出现 hold time failure，输出也就不会出错。**

## Summary(非常重要)

假设理想的时钟为 clk，它的周期为 T，它连接到 DFF1 和 DFF2 的时钟为 clk1 和 clk2，它们相对于理想时钟的时延分别为 Tc1 和 Tc2；DFF 的参数分别为 Tsu，Th 和 Tco，中间组合逻辑的时延为 Tcomb。

数据的时延：Tc1 + Tco + Tcomb

时钟的时延：Tc2 + T

要满足 setup time 的要求，则有 Tc2 + T - (Tc1 + Tco + Tcomb) >= Tsu，整理一下即：

T >= Tsu + Tco + Tcomb - Tskew --Eq1

所以可以确定出系统的最大工作频率：

Fmax <= 1 / (Tsu + Tco + Tcomb - Tskew) -- Eq2

因为 Tc1, Tc2, Tco, Tsu 都是固定值，所以一个系统的最大工作频率取决 Tcomb。

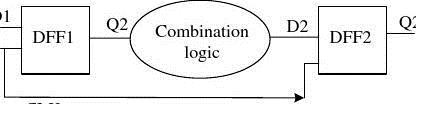
而对于 hold time，则有不等式：Tc1 + Tco + Tcomb > Tc2 + Th，整理一下就有：

Tco + Tcomb - Tskew > Th -- Eq3

上一级 DFF 的输出数据不能传输太快，侵占了下一级 DFF 正在锁存的 hold time，也就是说限制了数据的传输速度的上限。

有了前面的 3 个公式，就可以解决大部分问题了。**比如华为某年的笔试题：**

下图中时钟周期为T,触发器D1的建立时间最大为T1max，最小为T1min。组合逻辑电路最大延迟为T2max,最小为T2min。问，触发器D2的建立时间T3和保持时间应满足什么条件?



答：

题目中 Tco = 0，Tskew = 0

对于 setup time 的要求，将 Tco = 0，Tskew = 0 代入 Eq1 得：

Tsu <= {T - Tcomb}min = T - T2max

对于 hold time 的要求，将 Tco = 0，Tskew = 0 代入 Eq2 得：

Th <= {Tcomb}min = T2min

以上转自 <http://guqian110.github.io/pages/2015/03/18/static_timing_analysis_1_basic.html>

# STA静态时序分析(2)

按照路径所覆盖的范围，可以将时序路径要求分为 4 大类：

1. Input paths
2. Register-to-register paths
3. Output paths
4. Path specific exceptions

**添加约束最有效的方法就是先添加全局约束，然后根据需求考虑是否添加指定路径上的特殊约束。在很多案例中，只需要添加全局约束就可以了。**

FPGA器件执行工具都是由指定的时序要求驱动的。如果时序约束过头的话，就会导致内存使用增加，工具运行时间增加。更重要的是，过约束还会导致性能下降。因此，推荐使用实际设计要求的约束值。

下面分别讨论每种路径上的约束。

## Input paths

所谓 “输入路径 input paths”，指的是从 “FPGA 外部引脚 ---> 内部读取这个数的寄存器” 之间的路径。

在输入路径模型中，发送端是一个外部设备（当然也可以是 FPGA），接收端是 FPGA 芯片，输入路径讨论的就是以接收端的 FPGA 为视角，如何正确接收输入的数据。

依据接口类型，可以将输入路径的时序分为 2 类：

1. System Synchronous Inputs
2. Source Synchronous Inputs

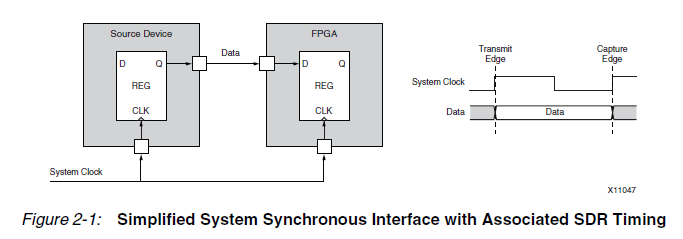
而对于输入路径的约束方法就是使用 OFFSET IN 来约束，它约束了输入数据和用于捕获这个数据的时钟沿 之间的关系。（顾名思义，输入路径约束的是方向为 IN 的数据和时钟的相对偏移 OFFSET 的关系）

下面分别对两类输入路径进行讨论：

### System Synchronous Inputs

**所谓 System Synchronous Inputs 其实就是指 “发送端 和 接收端 使用同一个系统**（system）时钟”。布线延时和时钟倾斜会限制这种接口的工作时钟频率，由于这个原因，这种接口一般应用在 SDR 中。

system synchronous SDR 应用示例如下图：



其时序为：

1. 发送器件在某个时钟的上升沿将数据发送出去
2. FPGA 在下一个时钟的上升沿捕获到数据

**对于这种接口的时序，使用 OFFSET IN 是最有效方便的添加约束的方法。对于接口中的每个时钟，都有一个对应的 offset in 约束，这个约束覆盖了所有的使用该时钟来捕获输入数据的路径。**

**添加约束的方法：**

**1. 首先对接口的时钟添加周期约束（period constraint）**

**2. 其次为接口添加全局 offset in 约束（global Offset In）**

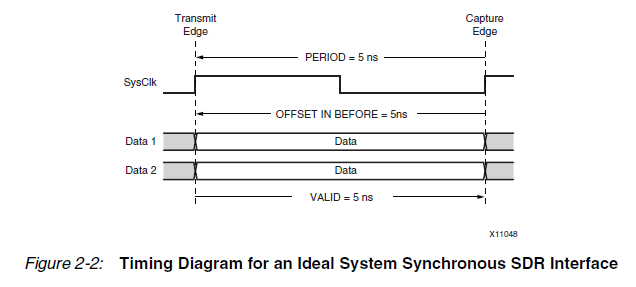
**语法如下：**

OFFSET = IN value VALID value BEFORE clock;

* OFFSET=IN 约束了数据变有效的沿和时钟的捕获数据的沿之间的距离
* VALID 约束了数据保持有效的时间长度

举例：

例1. 在下面理想的 system synchronous SDR interface 时序图中：



数据在时钟沿的前 5 ns 变为有效，并且保持了 5 ns 时间，所以在这个实例中的时序约束应该如下：

NET "SysClk" TNM\_NET = "SysClk";

TIMESPEC "TS\_SysClk" = PERIOD "SysClk" 5 ns HIGH 50%;

OFFSET = IN 5 ns VALID 5 ns BEFORE "SysClk";

而且这个约束对于 data1 和 data2 共同有效。

例2. 不是理想的 system synchronous SDR interface 中，假设时钟周期为 5 ns，并且占空比为 50%，数据在发送时钟上升沿之后的 500 ps 之后变有效，并且持续 4 ns。则时序约束应该如下：

NET "clock" TNM\_NET = CLK;

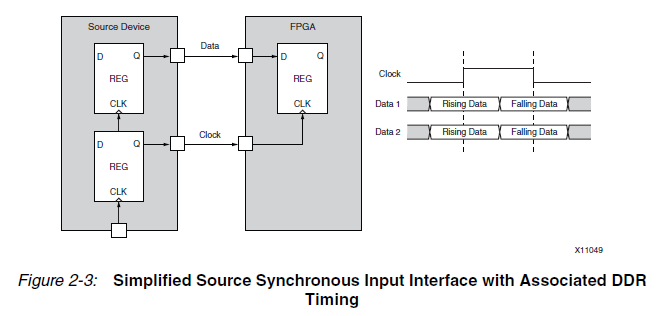
TIMESPEC TS\_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN 4.5 ns VALID 4 ns BEFORE clock;

### Source Synchronous Inputs

所谓 Source Synchronous Inputs 其实就是指 “发送端重新生成一个时钟信号，并且将数据和时钟信号有着相似的布线，两者的延时基本相同，在接收数据的 FPGA 端，使用这个时钟来捕获这个数据”。布线延时和时钟倾斜不再是限制这种接口工作速度的因素，所以这种接口一般应用在双倍数据速率的 DDR(Double data rate) 中。

source synchronous DDR 应用示例如下图：



其时序为：

1. 发送器件在某个时钟的**上升沿和下降沿都会发送一个独立的数据**
2. 接收端的 FPGA 使用发送端传递过来的这个再生时钟来捕获数据

**对于这种接口的时序，使用 OFFSET IN 是最有效方便的添加约束的方法。对于接口中的每个时钟，都有一个对应的 offset in 约束，这个约束覆盖了所有的使用该时钟来捕获输入数据的路径。**

**添加约束的方法：**

**1. 首先对接口的时钟添加周期约束（period constraint）**

**2. 其次为时钟的上升沿添加全局 offset in 约束（global Offset In）**

**3. 最后为时钟的下降沿添加全局 offset in 约束（global offset in）**

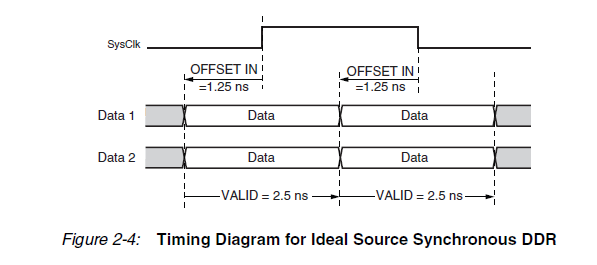
**语法如下：**

OFFSET = IN value VALID value BEFORE clock RISING;

OFFSET = IN value VALID value BEFORE clock FALLING;

举例：

例3. 在下面理想的 source synchronous DDR interface 时序图中：



输入时钟的周期为 5 ns，并且占空比为 50%，两 bit 的数据的有效时间都为 1/2 时钟周期，所以在这个实例中的时序约束应该如下：

TIMESPEC (time specification) TNM\_NET (Time name NET)

NET "SysClk" TNM\_NET = "SysClk";

TIMESPEC "TS\_SysClk" = PERIOD "SysClk" 5 ns HIGH 50%;

OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE "SysClk" RISING;

OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE "SysClk" FALLING;

例4. 不是理想的 source synchronous DDR（数据和时钟的边沿对齐），假设时钟周期为 5 ns，并且占空比为 50%，上升沿和下降沿的数据都保持有效 2ns，并且位于时钟波形的高低电平的中间位置，也就是说，在数据有效的前后各有 250 ps 的空白。

对于上升沿，因为数据相对于捕获它的时钟沿后了 250 ps，并且有效时间持续了 2 ns；对于下降沿，数据也沿后了 250 ps， 并且有效时间持续了2 ns，所以时序约束如下：

NET "clock" TNM\_NET = CLK;

TIMESPEC TS\_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN -250 ps VALID 2 ns BEFORE clock RISING;

OFFSET = IN -250 ps VALID 2 ns BEFORE clock FALLING

例5. 不是理想的 source synchronous DDR（数据和时钟的中间位置对齐），假设时钟周期为 5 ns，并且占空比为 50%，上升沿和下降沿的数据都保持有效 2 ns，并且时钟沿对齐数据的中间位置，可以得出结论，在数据有效的前后各有 250 ps 的空白。

对于上升沿，因为数据相对于捕获它的时钟提前了 1 ns，并且有效时间持续了 2 ns；对于下降沿，数据也提前了 1 ns，并且有效时间持续了 2 ns，所以时序约束如下：

NET "clock" TNM\_NET = CLK;

TIMESPEC TS\_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN 1 ns VALID 2 ns BEFORE clock RISING;

OFFSET = IN 1 ns VALID 2 ns BEFORE clock FALLING;

### Register-to-register paths

这部分讨论寄存器-寄存器之间同步路径上的周期约束（period constraint）。

period constraint

* 定义了时钟域的时序
* 覆盖了内部寄存器之间的同步数据路径
* 分析单个时钟域内的路径
* 分析两个相关联的时钟域之间的所有路径
* 在分析时考虑了时钟域之间的相位、频率、不确定性因素

同步时钟域的约束可以分为以下 3 类：

1. Automatically Related Synchronous DLL, DCM, PLL, and MMCM Clock Domains
2. Manually Related Synchronous Clock Domains
3. Asynchronous Clock Domains

**使用工具对 DCM、PLL 和 MMCM 的输出时钟自动添加时钟关系，并且手动定义外部的相关时钟的关系。通过这种方法，可以保证所有的跨时钟域的同步路径都被正确约束、分析，使用这种方法来添加 period constraint 可以避免再添加额外的跨时钟域约束。**

#### Automatically Related Synchronous DLL, DCM, PLL, and MMCM Clock Domains

最常见的时钟信号就是下面两个：

1. 输入到 DCM、PLL 或者是 MMCM 的时钟信号
2. 从这些单元输出，用来驱动内部的同步路径的时钟信号

推荐的约束方法是 **对输入到 DCM、PLL 或者是 MMCM 的时钟信号添加周期约束（period constraint）。** 通过对输入时钟添加周期约束，Xilinx 工具会

* 自动为 DCM、PLL 或者是 MMCM 的输出生成一个新的周期约束
* 确定输出时钟之间的关系
* 分析这些同步域之间的任何路径

**语法如下：**

NET "ClockName" TNM\_NET = "TNM\_NET\_Name";

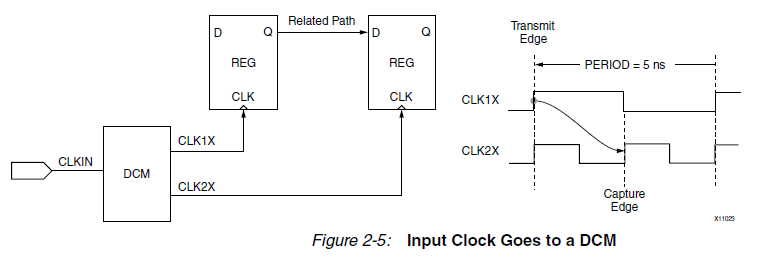
TIMESPEC "TS\_name" = PERIOD "TNM\_NET\_Name" PeriodValue HIGH HighValue%;

其中

* PeriodValue 定义时钟周期
* HighValue 定义时钟的占空比

举例：

例6. 在下图的例子中



输入时钟连接到 DCM 的输入端，因为输入时钟的时钟周期为 5 ns，并且占空比为 50%，所以添加的约束为：

NET "ClkIn" TNM\_NET = "ClkIn";

TIMESPEC "TS\_ClkIn" = PERIOD "ClkIn" 5 ns HIGH 50%;

在上面的例子中，我们给出上面的约束条件之后，DCM 会自动为它的两个输出添加约束，并且分析这两个时钟域

#### Manually Related Synchronous Clock Domains

在有些情况中，Xilinx 工具无法自动分析指定同步时钟域之间的关系（比如相关的时钟信号从两个不同的管脚输入进入到 FPGA），

**这种情形下，Xilinx 推荐的约束方法是：**

**1. 为每一个输入时钟都创建一个周期约束**

**2. 手动定义时钟之间的关系**

一旦我们定义了时钟约束，工具会自动分析两个同步域之间的所有路径，并且在分析时会把频率、相位、不确定因素都考虑进去。

Xilinx 的约束系统可以通过在周期约束中加入频率和相位信息来添加更加复杂的周期约束（complex manual relationship）。

**约束方法：**

**1. 为主时钟定义周期约束**

**2. 以主时钟的约束为参考，为其他的相关时钟添加周期约束**

**语法如下：**

NET "PrimaryClock" TNM\_NET = "TNM\_Primary";

NET "RelatedClock" TNM\_NET = "TNM\_Related";

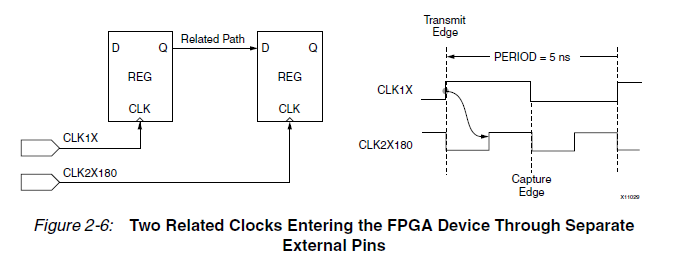
TIMESPEC "TS\_primary" = PERIOD "TNM\_Primary" PeriodValue HIGH HighValue%;

TIMESPEC "TS\_related" = PERIOD "TNM\_Related" TS\_Primary\_relation PHASE value;

在 related PERIOD 的约束中，PERIOD 的值定义了相关时钟和主时钟之间的关系（以时钟周期为单位），这种关系用主时钟的 TIMESPEC 形式来定义；PHASE 的值定义了主时钟和相关时钟的上升沿之间的关系。

举例：

例7. 如下图所示



CLK2X180 的频率是 CLK1X 的 2 倍，所以 PERIOD 的值为 1/2；CLK2X180 的相位相比于主时钟，偏移了 180度，所以它的上升沿比主时钟的上升沿晚了 1.25 ns；所以这个例子的约束如下：

NET "Clk1X" TNM\_NET = "Clk1X";

NET "Clk2X180" TNM\_NET = "Clk2X180";

TIMESPEC "TS\_Clk1X" = PERIOD "Clk1X7 5 ns HIGH 50%;

TIMESPEC "TS\_Clk2X180" = PERIOD "Clk2X180" TS\_Clk1X/2 PHASE +1.25 ns;

#### Asynchronous Clock Domains

异步时钟是指频率或相位有一个不同或者都不相同的时钟。因为时钟是不相关的，所以在进行 setup/hold time 分析时，是无法确定时钟的最终关系的。因此，**Xilinx 推荐在设计时使用一些特殊的方法来确保数据能被正确捕获。**然而，有时候设计者希望不考虑频率和相位之间的关系，在孤立的条件下限制数据传输的最大时延。

Xilinx 约束系统允许不考虑源和目的时钟之间的频率、相位关系，直接约束数据路径的最大时延。语法就是使用带 DATAPATHONLY 关键字的 From-To 语句。

**约束方法：**

**1. 为源同步寄存器定义时钟组**

**2. 为目的同步寄存器定义时钟组**

**3. 用带 DATAPATHONLY 关键字的 From-To 来约束两个时钟域之间的最大数据时延**

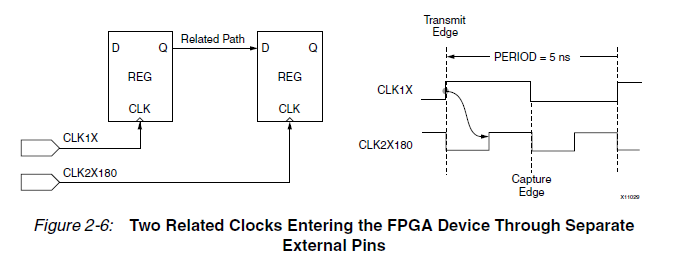
**语法如下：**

NET "CLKA" TNM\_NET = FFS "GRP\_A";

NET "CLKB" TNM\_NET = FFS "GRP\_B";

TIMESPEC TS\_Example = FROM "GRP\_A" TO "GRP\_B" Delay DATAPATHONLY

举例：



例8. 以前面的图为例，假设 CLKA 输入到第一个寄存器 R1，CLKB 输入到第二个寄存器 R2，R1 的输出连接到 R2 的输入，能忍受的最大的数据时延为 5 ns，则约束为：

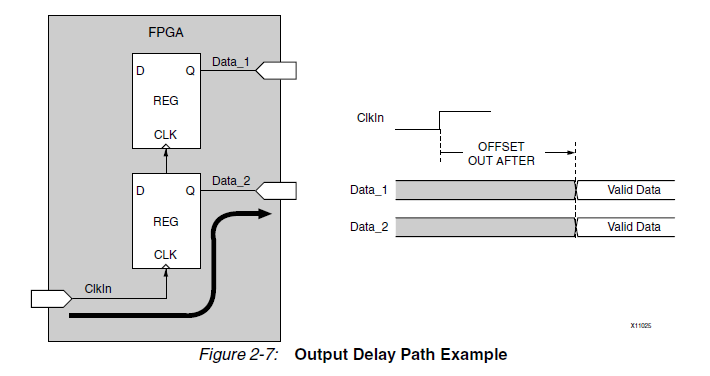
NET "CLKA" TNM\_NET = FFS "GRP\_A";

NET "CLKB" TNM\_NET = FFS "GRP\_B";

TIMESPEC TS\_Example = FROM "GRP\_A" TO "GRP\_B" 5 ns DATAPATHONLY

## Output paths

这部分讨论如何为输出路径添加约束。输出约束覆盖了从 “内部同步单元/寄存器 ---> FPGA 输出管脚” 之间的所有路径。如下图所示：



在输出路径模型中，发送端是 FPGA 芯片，接收端是一个外部设备（当然也可以是 FPGA），输入路径讨论的就是以发送端的 FPGA 为视角，如何将待发送定数据正确发送出去。

和输入路径对应，输出路径使用 OFFSET OUT 来约束以达到时序要求。

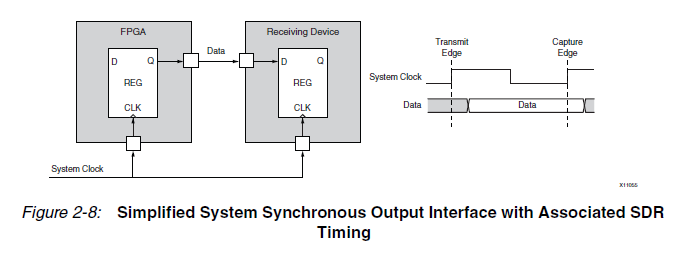
OFFET OUT 定义了输出数据和将该数据发送到输出管脚的时钟之间的关系。OFFSET OUT 的分析会自动将影响输出数据/输出时钟的内部因素考虑在内：

1. 时钟的频率和相位畸变
2. 时钟的不确定性
3. 数据时延的调整

和输入路径类似，输出路径的时序要求也可以依据接口的类型（system/source synchronous）和数据速率（SDR/DDR）来分类讨论。

### System Synchronous Output

在 system synchronous output 中，发送端和接收端使用同一个时钟，所以发送端的 FPGA 只需要发生数据部分就可以了。如下图所示：



**对于 system synchronous output 接口，使用全局 OFFSET OUT 是最有效的方法。每个 OFFSET OUT 都约束了一个对应的输出时钟，并且所有使用这个时钟来触发的输出数据路径都被这个约束所覆盖。**

**约束方法：**

**1. 为输出时钟定义一个时钟名（TNM）来分组，这个时钟组包含了所有被这个时钟触发的输出寄存器**

**2. 定义接口的全局 OFFSET OUT 约束**

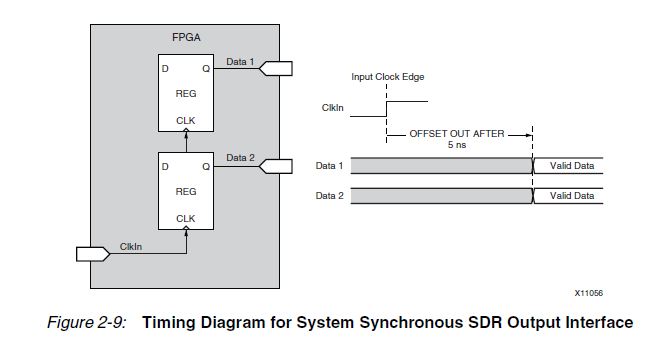
**语法如下：**

OFFSET = OUT value AFTER clock;

其中 OFFSET = OUT <value> 规定了 “接收器件的输入时钟（=发送端 FPGA 的时钟）上升沿 ---> 发送端 FPGA 输出数据变有效” 直接的最大时延。

举例：

例9. 如下图所示的 System Synchronous SDR output interface



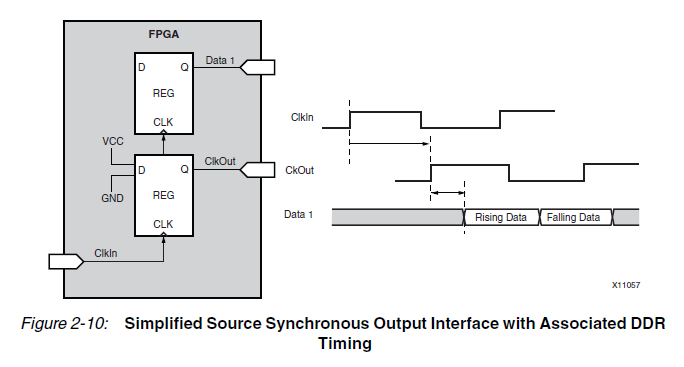
假设发送端的输出数据必须在时钟上升沿之后的 5 ns 内变为有效，则约束如下：

NET "ClkIn" TNM\_NET = "ClkIn";

OFFSET = OUT 5 ns AFTER "ClkIn";

### Source Synchronous Output

在 source synchronous output interface，发送端的 FPGA 会重新生成一个时钟信号，并且将时钟信号和数据一起发送出去，如下图所示：



**对于 source synchronous output 接口，使用全局 OFFSET OUT 是最有效的方法。每个 OFFSET OUT 都约束了一个对应的输出时钟，并且所有使用这个时钟来触发的输出数据路径都被这个约束所覆盖。**

**约束方法：**

**1. 为输出时钟定义一个时钟名（TNM）来分组、，这个时钟组包含了所有被这个时钟触发的输出寄存器**

**2. 为时钟上升沿添加 global Offset Out 约束**

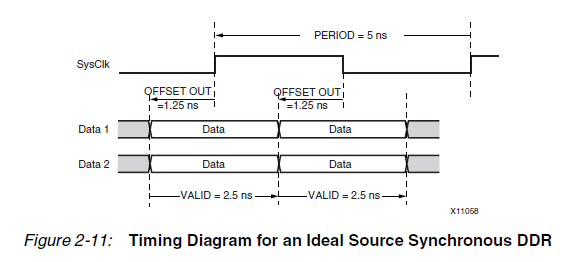
**3. 为时钟下降沿添加 global Offset Out 约束**

**语法如下：**

* OFFSET = OUT <value> 约束了从 “输入时钟的上升沿 ---> 发送端 FPGA 输出端口数据变为有效” 的最大时延。
* 关键词 REFERENCE\_PIN 约定以重新生成的时钟作为参考，输出数据的 skew 报告就是以这个时钟作为参考生成的。

举例：

例10. 下图是一个理想的 Source Synchronous DDR interface 时序图



时钟周期为 5 ns，并且占空比为 50%，数据保持有效的时间为 1/2 时钟周期。所以，这个示例的约束如下：

NET "ClkIn" TNM\_NET = "ClkIn";

OFFSET = OUT AFTER "ClkIn" REFERENCE\_PIN "ClkOut" RISING;

OFFSET = OUT AFTER "ClkIn" REFERENCE\_PIN "ClkOut" FALLING

## Path specific exceptions

通过前面的 3 节的讨论，对输入、寄存器-寄存器、输出路径进行约束，大部分时序路径都得到了正确约束，然后在一些情况中，存在少数不适应于全局约束的少数路径，这些例外最常见的就是：

* False Paths (Paths Between Registers That Do Not Affect Timing)
* Multi-Cycle Paths

下面分别讨论。

### False Paths (Paths Between Registers That Do Not Affect Timing)

如果有些路径不影响时序性能，那么我们就可以将这些路径从时序分析中移除。**最常用的方法就是使用带 time ignore (TIG) 关键词的FROM\_TO 约束。** 使用这种约束，可以

* 从源时钟域指定一组寄存器
* 从目标时钟域指定一组寄存器
* 将源到目标域的路径从时序分析中移除

**约束方法：**

**1. 在源时钟域指定一组寄存器**

**2. 在目标时钟域指定一组寄存器**

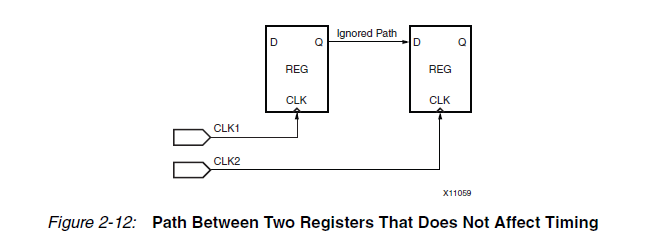
**3. 使用带 TIG 关键词的 FROM-TO 来移除这两个域之间的路径**

**语法如下：**

TIMESPEC "TSid" = FROM "SRC\_GRP" TO "DST\_GRP" TIG;

举例：

例11. 假设下图中的两个寄存器之间的路径并不影响设计的时序，希望将这条路径从时序约束中移除



则本例的约束如下：

NET "CLK1" TNM\_NET = FFS "GRP\_1";

NET "CLK2" TNM\_NET = FFS "GRP\_2";

TIMESPEC TS\_Example = FROM "GRP\_1" TO "GRP\_2" TIG;

### Multi-Cycle Paths

在多周期路径中，从发送端到接收端的同步单元，数据以低于周期约束中的 PERIOD 速率传输。这种情况最常见的场景是同步单元使用一个共同的 clock enable 来门控。

通过定义一个多周期路径（Multi-Cycle path），这些同步单元的约束条件会比默认的周期约束宽松很多。方法就是先给周期约束定义一个标识，然后再声明 Multi-Cycle path 包含多少个时钟周期。然后工具就可以合理分配这些路径的优先级。

定义一个多周期路径的 **最常用的方法就是用 clock enable 信号定义一个时钟组，这样我们就可以用这个 clock enable 来定义一个包含了源、目的寄存器的时钟组，然后将多周期约束应用到这些寄存器之间的路径上。**

**约束方法：**

**1. 对公用时钟域进行周期约束**

**2. 定义所有基于同一个 clock enable 信号的寄存器**

**3. 对新的时序要求进行 From:To(Multi-Cycle) 约束**

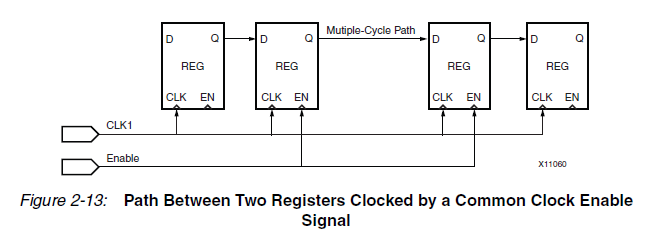
**语法如下：**

TIMESPEC "TSid" = FROM "MC\_GRP" TO "MC\_GRP" <value>;

* MC\_GRP 定义了一组公用时钟的寄存器
* 所有从 MC\_GRP 开始，到 MC\_GRP 结束的路径就是需要进行多周期约束的路径

举例：

例12. 如下图所示的假设情形中



两个寄存器直接的路径被一个共同点 clock enable 控制，并且 clock enable 的变化速率是时钟频率的一半。则本例的时序约束如下：

NET "CLK1" TNM\_NET = "CLK1";

TIMESPEC "TS\_CLK1" = PERIOD "CLK1" 5 ns HIGH 50%;

NET "Enable" TNM\_NET = FFS "MC\_GRP";

TIMESPEC TS\_Example = FROM "MC\_GRP" TO "MC\_GRP" TS\_CLK1\*2;

## Summary

通过对这四种类型的 timing path 进行约束，基本上系统内所有路径都得到了合理约束。