



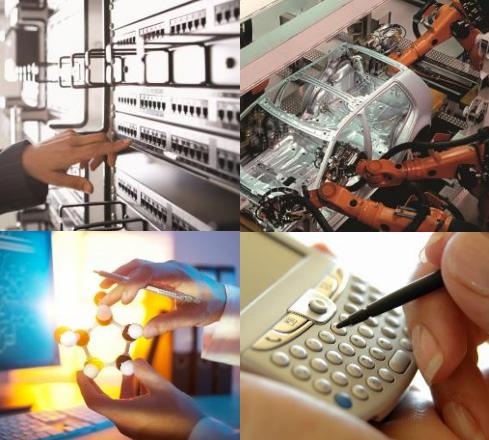
我一直在问我自己
到底有没有一种简便的方法
能够让更多 **Vivado** 的用户
从我们的新技术新产品中受益，
从而帮助他们更好更快地完成自己的设计？

VIVADO

ADVANCED DESIGN TIPS

使用误区与进阶

作者：Ally Zhou, Xilinx工具与方法学应用专家



让更多的用户受益于强大的 Vivado 与 UltraFAST

想到要写这一系列关于工具和方法学的小文章是在半年多前，那时候 Vivado® 已经推出两年，陆续也接触了不少客户和他们的设计。我所在的部门叫做“Tools & Methodology Applications”，其实也是专为 Vivado 而设的一个部门，从 Vivado 的早期计划开始，我和我的同事们就投入到了 Xilinx® 和 Vivado 的客户们的推广和支持中，我们给客户做培训，在市场活动上做报告，培训和考核代理商，也去现场支持客户的设计。两年的时间，Vivado 不断成熟，客户们也从最初的焦虑抗拒到全面接受，但随着与用户更深层次的技术交流。接触了一些客户的设计后，我渐渐发现其实很多 Vivado 的用户并没有真正了解它的好处，或者说，要么不够深入，要么就是有很多认识的偏差。也许是语言的限制，也许是各种各样的动辄上百页的 PDF 文档的无所适从，我能感觉到他们需要一些更直接，更有针对性的指引。

我一直在问我自己，到底有没有一种简便的方法，能够让更多 Vivado 的用户从我们的新技术新产品中受益，从而帮助他们更好更快地完成自己的设计？

下定决心后，我便开始从日常工作和大家的反馈中寻找普遍感兴趣的话题，分门别类、挑选实例、尽量用简洁明快的文字和一目了然的插图叙事，把一篇文章控制在十页以内。阅读这样一篇文章也许只需要你一顿午饭的时间，当你碰到一个技术问题，可以快速用关键字索引到对应的文章。作为工程师，应该比较欢迎这样的小文章吧。在此要特别感谢我的很多同事，这些文章中的不少实例和配图就是从他们创建的 PPT 中挑选的，我能做的就是把更多 Xilinx 技术专家们的经验之谈用大家熟悉的语言传播出去，传递下去。同时也感谢 Xilinx 市场部同事们的一路支持，使得以下九篇文章在广大客户群中广为流传，并以此电子书的形式分享给更多的网友们。



周丽娜 (Ally Zhou)

赛灵思公司工具与方法学应用专家

Ally Zhou 拥有十多年 FPGA 设计、EDA 工具和多年客户支持的经验。2012 年加入公司以来，专注于 Vivado 设计套件以及 UltraFast 设计方法学的推广和支持。

Ally 曾先后在同济大学，芬兰米凯利理工学院和复旦大学求学，获得工学硕士学位。加入赛灵思公司之前，曾在 Synopsys 工作，主要负责 FPGA 综合和 ASIC 原型验证方案的支持。

第一章：十分钟教会你 UltraFast

第二章：XDC 约束技巧之时钟篇

第三章：XDC 约束技巧之 CDC 篇

第四章：XDC 约束技巧之 I/O 篇 (上)

第五章：XDC 约束技巧之 I/O 篇 (下)

第六章：Tcl 在 Vivado 中的应用

第七章：用 Tcl 定制 Vivado 设计实现流程

第八章：在 Vivado 中实现 ECO 功能

第九章：读懂用好 Timing Report





十分钟教会你 UltraFast

UltraFAST™是Xilinx®在2013年底推出的一套设计方法学指导，旨在指引用户最大限度地利用现有资源，提升系统性能，降低风险，实现更快速且可预期的设计。面向Vivado®的UltraFAST方法学的主体是UG949文档，配合相应的Checklist，随Vivado版本同时更新，用户可以在Xilinx的主页上免费下载。目前，针对Vivado设计套件的UltraFAST中文版也已经上市，另外一套全新的针对嵌入式可编程设计的UltraFAST嵌入式设计方法指南UG1046也已经在Xilinx官网上开放下载。

尽管UltraFAST这个字眼经常在网上看到，不论官方还是其他媒体上说起Vivado设计套件时也常常提到，但很多用户仍然对这个概念十分模糊，有不少人下载文档后看到300页的PDF顿时也失去了深入学习和了解的兴趣。

套用在Xilinx内部被誉为“Vivado之父”的Vivado产品营销总监Greg Daughtry在去年第一届Club Vivado中所提出的“时序收敛十大准则”的概念，试着用十分钟的篇幅来概括一下什么是UltraFAST，以及怎样利用UltraFast真正帮助我们的FPGA设计。

准则一：合适的代码风格

准则六：少而精的物理约束

准则二：精准的时序约束

准则七：选择实现策略

准则三：管理高扇出网络

准则八：共享控制信号

准则四：层次化设计结构

准则九：读懂日志和报告

准则五：处理跨时钟域设计

准则十：发挥Tcl的作用

Greg Daughtry, Xilinx Vivado产品营销总监

时序设计的十大准则，基本上也涵盖了UltraFAST设计方法指南的基本要点。UG949中将FPGA设计分为设计创建、设计实现和设计收敛几大部分来讨论，除了介绍所有可用的设计方法和资源，更多的是一些高级方法学技巧，这些技巧基本上都跟时序收敛有关或是以时序收敛为目标，有些通用的方法和技巧甚至脱离了具体选用的FPGA器件的限制，适用于更广泛意义上的时序收敛。

最宝贵的是，所有这些UltraFAST设计方法学技巧都来自一线技术支持人员的经验以及客户的反馈，是业界第一本真正意义上完全面向用户的指南。这一点只要你试着读过一两节UG949就会有明显感觉，所有其中提到的技巧和方法都具有很高的可操作性，可以带来立竿见影的效果。

接下来我们就由这十大准则展开，带领各位读者在十分钟内理清UltraFAST方法学的脉络，一探其究竟。

UltraFAST™
Design Methodology

准则一：合适的代码风格

理想环境下，源代码可以独立于最终用于实现的器件，带来最佳的可移植性和可复用性。但是，底层器件各自独特的结构，决定了通用代码的效率不佳，要最大化发挥硬件的性能，必然需要为实现工具和器件量身定制代码。

关于 Xilinx 器件和 Vivado 适用的代码风格，我们有以下建议：

1. 多使用 Vivado 自带的代码模板
2. 尽量避免使用异步复位
3. 在模块边界上使用寄存器而非组合逻辑
4. 使用流水结构来降低逻辑层数
5. 采用适当的 RAM 和 DSP 的实现方式（是否选用硬核）
6. 在综合后或是逻辑优化（opt_design）后的时序报告上分析代码优化的方向

准则二：精准的时序约束

精准的时序约束是设计实现的基础，对时序驱动工具 Vivado 来说，约束就是最高指示，是其努力实现的目标。很多时候我们发现，约是有经验的工程师约是喜欢用一些旧有经验套用在 Vivado 上，例如很多人偏爱用过约束的方式来追求更高的性能，但实际上对 Vivado 来说，大部分的过约束只会阻碍时序收敛。

简要概括而言，精简而准确的约束是时序收敛的必要条件，而 UltraFast 中提出的 Baseline 基线方法则是充分条件。

具体的约束方法我们在 [《XDC 约束技巧》](#) 中有详细讨论，除了保证语法正确，还要注意设置 XDC 约束的顺序，通常第一次运行时只需要约束所有时钟，然后在内部路径基本满足时序约束的情况下加入关键 I/O 的约束，其次再考虑必要的时序例外约束。

所有这些约束都必须遵循精简而准确的原则，且可以借助 Vivado 中的 XDC Templates 以及 Timing Constraint Wizard 的帮助来进行。

Baseline 基线方法可以说是 UltraFast 的灵魂部分，强烈建议所有 Vivado 的用户都能精读 UG949 中的这部分内容，并将之应用在具体的设计中。有机会我会深入展开一篇专门介绍 Baseline 方法的短文，这里先将其核心的概念做一个总结。

Critical Path could be a Moving Target

Example from a Real Design (250 MHz)

► Post-synthesis

– Worst path: 13 levels of logic



► Post-place

– Worst path: 7 levels

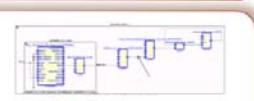
– Paths with 7-13 levels got placed locally



► Post-route

– Worst Path: 4 levels of logic

– Paths with 5-13 levels got preferred routing



Analyze & Fix timing issues at early stages for faster timing convergence

上图展示了同一个设计在三个不同阶段用同样的命令报告时序所得到的最差路径，可以清晰的看出，即使不做任何源代码上的改动，设计中真正最差的路径已经不会作为最差路径出现在布局布线后的报告中。这正是因为 Vivado 时序驱动的天性决定了其在设计实现的每一步都是以开始时读到的设计输入和约束为依据，尽量将最好的资源用在最差的路径上，从而尽最大可能实现时序收敛。

这便是 Baseline 理论的基础，除了按顺序设置精准的时序约束，在设计实现的每一步，用户都需要关注时序报告，并以其为依据来调整设计源代码或是应用其他必要的约束和选项来优化设计。保证每一阶段之后的时序报告都满足约束或是仅余 300ps 以内的时序违例，再进入下一阶段的设计实现过程，否则，应该继续在当前阶段或是退回到上一阶段调整后重跑设计，直到满足要求再继续。

越早发现和定位问题，越是可以通过少量的努力来达到更大范围的改进。

准则三：管理高扇出网络

高扇出网络几乎是限制 FPGA 设计实现更高性能的第一大障碍，所以我们需要很严肃地对待设计中的高扇出网络。

很多人会陷入一个误区，反复纠结到底多大的扇出值算是大？其实这一点不是绝对的，在资源充裕时序要求不高的情况下几千甚至上万都不算大，反之在局部关键路径上仅有几十的扇出也可能需要进一步降低。

在 Vivado 中，我们除了关注时序报告，尤其是布局后布线前的报告来定位关键路径上影响时序的高扇出网络外，还有一个专门的命令 `report_high_fanout_nets`，在给其加上 `-timing` 的选项后，可以在报告高扇出路径的同时报告出这条路径的 Slack，帮助用户直观了解当前路径的时序裕量。此外，这个命令在报告中还会指出高扇出网络的驱动类型，是 FF 或是 LUT 等。

找到目标后，可以利用 `max_fanout` 来限定其扇出值，让工具在实现过程中复制驱动端寄存器来优化。如果高扇出网络并不是由同步逻辑来驱动，则可能需要修改代码。还有一些工具层面上的降扇出方法，比如选择更强更有针对性的策略，或是允许多次物理优化 `phys_opt_design`，甚至是通过我们在[《用 Tcl 定制 Vivado 设计实现流程》](#)中提到的“钩子”脚本等方式来进行局部降扇出的物理优化等等。

但有一点需要注意，Vivado 综合选项中的全局扇出限定要慎用，不要将其设置的过低以免综合出的网表过于庞大，带来资源上的浪费，并可能导致局部拥塞。

准则四：层次化设计结构

随着设计规模的不断扩大，以及 SoC 设计的兴起，越来越多的 IP 被整合到大设计中，曾经为高性能设计而生，便于统一管理和控制的自顶向下的设计流程变得不再适用，FPGA 设计也跟大规模 SoC 设计一样，需要采用层次化的设计流程，即自底向上的流程。这也要求设计者在源代码阶段就考虑到最终的实现，处理好模块的层次边界。

Vivado 中的 IP 设计是原生的自底向上流程，用户可以将 IP 生成独立的 DCP 再加入到顶层设计中去。我们也鼓励用户将某些相对固定或独立的模块综合成 DCP 后加入顶层设计，这么做除了加快设计迭代外，也更利于设计开始阶段的调试和问题的定位。

Vivado 中的 OOC 模式甚至还支持完全层次化的设计，即将底层模块的布局布线结果也进行复用，这么做虽然流程复杂，却带来了更全面的控制性，也是部分可重配置技术的实现基础。

准则五：处理跨时钟域设计

FPGA 设计中通常都带有跨时钟域的路径，如何处理这些 CDC 路径非常重要。由于 Vivado 支持的约束标准 XDC 在处理 CDC 路径上与上一代 ISE 中支持的 UCF 约束有本质区别，如何约束以及怎样从设计上保证 CDC 路径的可靠性就成了重中之重。

[《XDC 约束技巧之 CDC 篇》](#)中对 Vivado 中的跨时钟域设计有详细描述，UG949 中也有不少篇幅用来讨论 CDC 路径的各种设计技巧和约束方法。建议用户深入学习和了解这部分的内容，其中有不少概念并不仅仅局限于 FPGA 设计中的跨时钟域设计，放在其他 IC 设计上也一样有效。

需要提醒大家的是，一定要利用好 Vivado 中的各种报告功能，例如 report_cdc 和 DRC 报告中的 methodology_checks 来检测设计中的 CDC 结构问题，并作出具体的设计调整或是补全 CDC 约束。另外要注意各种不同的 CDC 路径处理方法之间的优劣，选择最适合自己的设计方式，配合相应的约束来保证跨时钟域路径的安全。

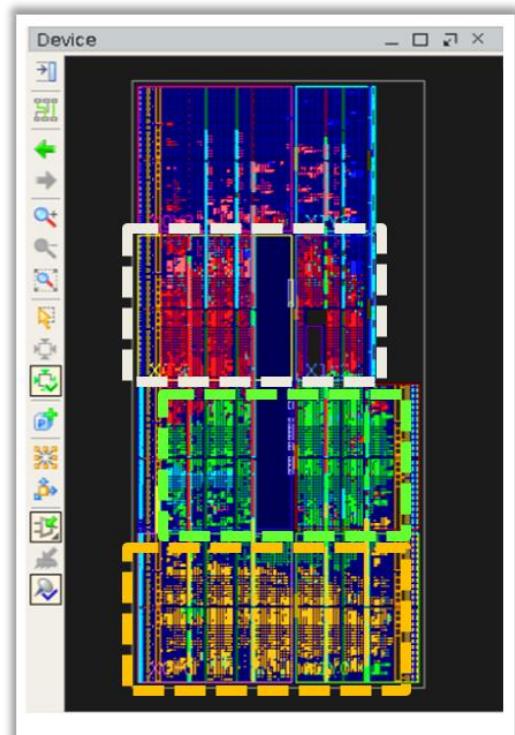
准则六：少而精的物理约束

不同于对时序约束尤其是时钟约束之全面而精准的要求，Vivado 对物理约束的要求只有一个字：少。这里的物理约束更多强调的是除了 I/O 引脚位置这些必要项之外的约束，例如对 RAMB 和 DSP48 的位置约束，还有局部的 floorplan 计划。

很多资深工程师非常喜欢画 floorplan，因为其对设计的数据流和资源使用情况了如指掌，根据自己理解画出的 floorplan 通常也算合理。但是，floorplan 在 Vivado 中的重要性远低于以往在 ISE 上的作用。根据客户实际经验反馈，绝大多数的设计中都无需任何 floorplan（某些时序要求较高的 SSI 芯片设计上可能需要），因为算法的改进，Vivado 在布局上比上一代 ISE 更聪明，没有任何物理约束（除了 IO 引脚位置约束）的设计反而能在更短的时间内更好地满足时序要求。

在确实需要锁定某些宏单元以及进行 floorplan 的设计中，一般我们会推荐先不加任何物理约束来跑设计，在其他诸如改进源代码，设置约束和选项，改变策略等办法都试过后，再尝试物理约束。而且，最好只在少量关键的设计区域进行 floorplan，切忌过度约束，不要创建资源利用率过高的 pblocks，同时避免重叠的 pblocks 区域。

顺便提一下，Vivado IDE 中的 Device 视图可以通过设置不同颜色来高亮显示不同模块，用户可以根据当前设计的布局结果配合时序报告和关键和调整 floorplan，操作非常便捷。



准则七：选择实现策略

从 ISE 升级到 Vivado 后，很多用户发现 SmartXplorer 功能不见了，当设计进行到后期，假如不能遍历种子，常让人感到无所适从，甚至怀疑到了这一步 Vivado 便无计可施。那么事实到底如何呢？

严格来讲，Cost Table 其实是一种无奈之举，说明工具只能通过随机种子的改变来“撞大运”般筛选出一个最佳结果，这也解释了为何改变 Cost Table 的结果是随机的，一次满足时序，并不代表一直可以满足。

因为更高级算法的引入，Vivado 中的设计实现变得更加可靠，而且是真正意义上的可预计的结果。但这并不代表在 Vivado 中对同一个设计进行布局布线只能有一种结果。我们可以通过“策略”来控制实现过程中的算法侧重，从而可以产生更优化的结果。

策略（Strategy）是一组工具选项和各个阶段指示（Directive）的组合，Vivado IDE 中内置了几十种可供用户直接选用，但如果穷尽各种组合，整个实现过程大约有上千种策略。当然，我们没必要遍历每种策略。而且因为策略是一种可预计可重现的实现方法，所以对同一个设计，可以在选择几种有侧重点的策略后挑选出效果最好的那个，只要设计后期没有大的改动，便可一直延用同样的策略。

具体策略的特性，请参考 UG949 和 UG904 等文档，也可以在 Vivado 中通过 help 菜单了解。更多时候，选择怎样的策略是一种经验的体现，另外，即使找到了最佳实现策略，也仍旧有可能不满足时序要求，这时候我们还可以参考 [《用 Tcl 定制 Vivado 设计实现流程》](#) 中所述，对设计实现的流程进行进一步的个性化定制。

另外要强调一点，修改策略来提升性能必须放在调整代码、约束和选项等更直接高效的优化方法之后进行，其能带来的性能提升比起前述优化方法来说也更加局限。

准则八：共享控制信号

共享控制信号这一点充分体现了设计必须考虑到用于底层实现的芯片结构的重要性，在 Xilinx 的芯片上，时钟、置位/复位和时钟使能等信号通称为 Control Set，进入同一个 SLICE 的 Control Set 必须统一。换句话说，不同 Control Set 控制下的 FFs 不能被 Vivado 放进同一个 SLICE。

为了提升 SLICE 的利用率，获得更高效的布局方案，提升时序性能，我们必须控制一个设计中 Control Set 的总数，尽量共享控制信号。具体做法包括：

1. 尽量整合频率相同的时钟和时钟使能信号；
2. 在生成 IP 时选择“共享逻辑”功能，则可以在不同 IP 间尽可能的共享时钟资源；
3. 遵循 Xilinx 建议的复位准则：
 - a) 尽量少使用复位
 - b) 必须复位时采用同步复位
 - c) 确保使用高电平有效的复位
 - d) 避免异步复位（RAMB 和 DSP48 模块中不支持异步复位）

Xilinx 的复位准则必须严格遵守，根据现场支持的经验来看，很多设计性能的瓶颈就在于设计源代码时没有考虑底层实现器件的硬件结构特点，尤其以复位信号的实现问题最为突出。

准则九：读懂日志和报告

任何一个工具的日志和报告都是衡量其性能最重要的一环，正因为有了完备的日志与报告，用户才可以通过其中显示的信息，定位设计中可能的问题，决定优化方向。

Vivado 日志中将信息显示为三大类，分别为 Error、Critical Warning 和一般 Warning/Note 等。Error 会导致工具直接中断，其他警告不会中断工具运行，但所有的 Critical Warning 都需要用户逐一检查并通过修改设计、增加约束或设置选项之类的方法来修复。

Vivado 的报告功能很强大，除了《读懂用好 Timing Report》中描述的时序分析报告，还有很多重要的报告，小到检查设计中的特定时序元件和链路，大到各种预置和自定义的 DRC 检查，不仅提供给了用户多样的选择，也进一步保证了设计的可靠性。

Vivado 也一直在增强和更新报告的种类，比如 2014.3 之后还增加了一个设计分析报告 report_design_analysis，用来报告关键路径上的潜在问题以及设计的拥塞程度。完整的 report 命令和功能可以在 UG835 中查询。

准则十：发挥 Tcl 的作用

Tcl 在 Vivado 中的作用不容小觑，不仅设计流程和报告全面支持 Tcl 脚本，就连 XDC 约束根本上也来自于 Tcl，用户甚至可以直接把包含有循环等功能的高级约束以 Tcl 的形式读入 Vivado 中用来指引整个实现流程。

《Vivado 使用误区与进阶》系列中有三篇关于 Tcl 在 Vivado 中的应用文章，详细描述了如何使用 Tcl 创建和应用约束，查找目标和定位问题；如何用 Tcl 来定制 Vivado 的设计实现流程，为图形化界面提供更多扩展支持；以及如何用 Tcl 实现 ECO 流程。Tcl 所带来的强大的可扩展性决定了其在版本控制、设计自动化流程等方面具有图形化界面不能比拟的优势，也解释了为何高端 FPGA 用户和熟练的 Vivado 用户都更偏爱 Tcl 脚本。

另外，随着 Xilinx Tcl Store 的推出，用户可以像在 App Store 中下载使用 app 一样下载使用 Tcl 脚本，简化了 Tcl 在 Vivado 上应用的同时，进一步扩展了 Tcl 的深入、精细化使用。最重要的是，Tcl Store 是一个基于 GitHub 的完全开源的环境，当然也欢迎大家上传自己手中有用的 Tcl 脚本，对其进行补充。

小结

关于 UltraFast 的要点总结基本可以概括在上述十点，这也可以说是对《Vivado 使用误区与进阶》系列短文的一个串烧。说实话，八九页的篇幅要将整个 UltraFast 讲透基本没有可能，对于正在使用 Vivado 做设计或是有兴趣试用的读者们，强烈建议各位在 Xilinx 官网下载完整的 UltraFast 指南并通读。



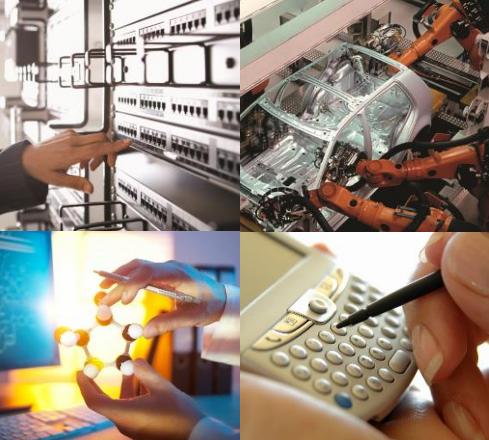
[点击这里，下载中文 UltraFAST 设计指南](#)

Download

这篇短文和这本电子书旨在帮助大家尽快上手 Vivado 和 XDC，宝剑在手，再加上盖世神功傍身，行走江湖岂不快哉。衷心祝福大家在 FPGA 设计之路上收获更多喜悦，让 Xilinx 和 Vivado 为您的成功助力。

— Ally Zhou 2015-4-27 于 Xilinx 上海 Office

[返回目录页](#)

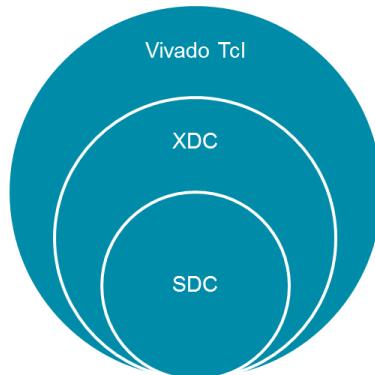


XDC 约束技巧之时钟篇

Xilinx®的新一代设计套件 Vivado®中引入了全新的约束文件 XDC，在很多规则和技巧上都跟上一代产品 ISE 中支持的 UCF 大不相同，给使用者带来许多额外挑战。Xilinx 工具专家告诉你，其实用好 XDC 很容易，只需掌握几点核心技巧，并且时刻牢记：XDC 的语法其实就是 Tcl 语言。

XDC 的优势

XDC 是 Xilinx Design Constraints 的简写，但其基础语法来源于业界统一的约束规范 SDC（最早由 Synopsys 公司提出，故名 Synopsys Design Constraints）。所以 SDC、XDC 跟 Vivado Tcl 的关系如下图所示。



XDC 的主要优势包括：

1. 统一了前后端约束格式，便于管理；
2. 可以像命令一样实时录入并执行；
3. 允许增量设置约束，加速调试效率；
4. 覆盖率高，可扩展性好，效率高；
5. 业界统一，兼容性好，可移植性强；

XDC 在本质上就是 Tcl 语言，但其仅支持基本的 Tcl 语法如变量、列表和运算符等等，对其它复杂的循环以及文件 I/O 等语法可以通过在 Vivado 中 source 一个 Tcl 文件的方式来补充。（对 Tcl 话题感兴趣的读者可以参考作者的另一篇文章 [《Tcl 在 Vivado 中的应用》](#)）XDC 与 UCF 的最主要区别有两点：

1. XDC 可以像 UCF 一样作为一个整体文件被工具读入，也可以在实现过程中被当作一个个单独的命令直接执行。这就决定了 XDC 也具有 Tcl 命令的特点，即后面输入的约束在有冲突的情况下会覆盖之前输入的约束（时序例外的优先级会在下节详述）。另外，不同于 UCF 是全部读入再处理的方式，在 XDC 中，约束是读一条执行一条，所以先后顺序很重要，例如要设置 IO 约束之前，相对应的 clock 一定要先创建好。
2. UCF 是完全以 FPGA 的视角看问题，所以缺省认为所有的时钟之间除非预先声明是同步的，否则就视作异步而不做跨时钟域时序分析；XDC 则恰恰相反，ASIC 世界的血缘背景决定了在其中，所有的时钟缺省视作全同步，在没有时序例外的情况下，工具会主动分析每一条跨时钟域的路径。

VIVADO®

XDC 的基本语法

XDC 的基本语法可以分为时钟约束、I/O 约束以及时序例外约束三大类。根据 Xilinx 的 UltraFast 设计方法学中 Baseline 部分的建议 (UG949 中有详细介绍)，对一个设计进行约束的先后顺序也可以依照这三类约束依次进行。本文对可以在帮助文档中查到的基本 XDC 语法不做详细解释，会将重点放在使用方法和技巧上。

时钟约束

时钟约束必须最早创建。对 7 系列 FPGA 来说，端口进来的时钟以及 GT 的输出 RXCLK/TXCLK 都必须由用户使用 `create_clock` 自主创建为主时钟。如果是差分输入的时钟，可以仅仅在差分对的 P 侧用 `get_ports` 获取端口，并使用 `create_clock` 创建。例如，

```
create_clock -name clk_200 -period 5 [get_ports clk200_p]
```

- Vivado 自动推导的衍生时钟

MMCM/PLL/BUFR 的输出作为衍生时钟，可以由 Vivado 自动推导，无需用户创建。自动推导的好处在于当 MMCM/PLL/BUFR 的配置改变而影响到输出时钟的频率和相位时，用户无需改写约束，Vivado 仍然可以自动推导出正确的频率/相位信息。劣势在于，用户并不清楚自动推导出的衍生钟的名字，当设计层次改变时，衍生钟的名字也有可能改变。这样就会带来一个问题：用户需要使用这些衍生钟的名字来创建 I/O 约束、时钟关系或是时序例外等约束时，要么不知道时钟名字，要么时钟名字是错的。

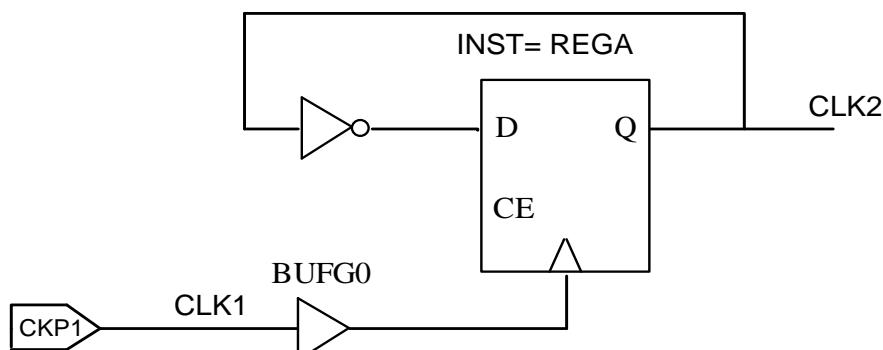
```
create_generated_clock -name my_clk_name [get_pins mmcm0/CLKOUT] \
    -source [get_pins mmcm0/CLKIN] \
    -master_clock main_clk
```

推荐的做法是，由用户来指定这类衍生时钟的名字，其余频率等都由 Vivado 自动推导。这样就只需写明 `create_generated_clock` 的三个 option，其余不写即可。如上所示。

当然，此类情况下用户也可以选择完全由自己定义衍生时钟，只需补上其余表示频率/相位关系的 option，包括 `-multiply_by` 、`-divide_by` 等等。需要注意的是，一旦 Vivado 在 MMCM/PLL/BUFR 的输出检测到用户自定义的衍生时钟，就会报告一个 Warning，提醒用户这个约束会覆盖工具自动推导出的衍生时钟（例外的情况见文章下半段重叠时钟部分的描述），用户须保证自己创建的衍生钟的频率等属性正确。

- 用户自定义的衍生时钟

工具不能自动推导出衍生钟的情况，包括使用寄存器和组合逻辑搭建的分频器等，必须由用户使用 `create_generated_clock` 来创建。举例如下，



```
create_clock -name clk1 -period 4 [get_ports CKP1]
create_generated_clock -name clk2 [get_pins REGA/Q] \
    -source [get_ports CKP1] -divide_by 2
```

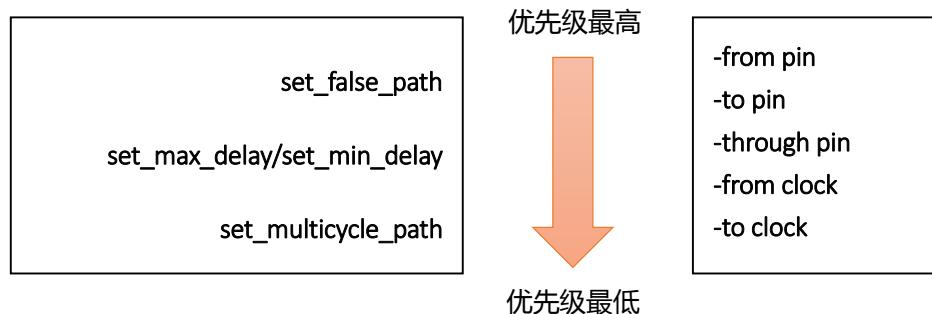
I/O 约束

在设计的初级阶段，可以不加 I/O 约束，让工具专注于满足 FPGA 内部的时序要求。当时序要求基本满足后，再加上 I/O 约束跑实现。XDC 中的 I/O 约束有以下几点需要注意：

1. 不加任何 I/O 约束的端口时序要求被视作无穷大。
2. XDC 中的 set_input_delay / set_output_delay 对应于 UCF 中 OFFSET IN / OFFSET OUT，但视角相反。OFFSET IN / OFFSET OUT 是从 FPGA 内部延时的角度来约束端口时序，set_input_delay / set_output_delay 则是从系统角度来约束。
3. 典型的 I/O 时序，包括系统同步、源同步、SDR 和 DDR 等等，在 Vivado 图形界面的 XDC templates 中都有示例。2014.1 版后还有一个 Timing Constraints Wizard 可供使用。

时序例外约束

时序例外约束包括 set_max_delay/set_min_delay，set_multicycle_path，set_false_path 等，这类约束除了要满足 XDC 的先后顺序优先级外，还受到自身优先级的限制。一个总的原则就是针对同一条路径，对约束目标描述越具体的优先级越高。不同的时序例外约束以及同一约束中不同条件的优先级如下所示：

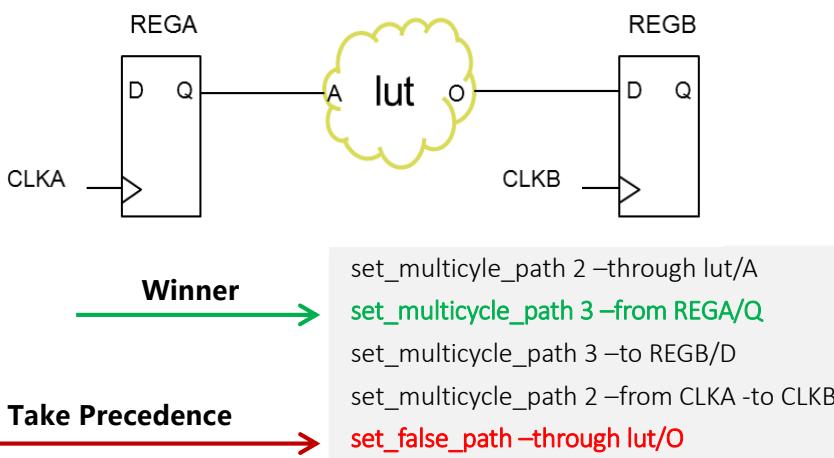


举例来说，依次执行如下两条 XDC，尽管第二条较晚执行，但工具仍然认定第一条约束设定的 15 为 clk1 到 clk2 之间路径的 max delay 值。

```
set_max_delay 15 -from [get_clocks clk1] -to [get_clocks clk2]
set_max_delay 12 -from [get_clocks clk1]
```

Winner

再比如，对图示路径依次进行如下四条时序例外约束，优胜者将是第二条。但如果再加入最后一条约束，false path 的优先级最高，会取代之前所有的时序例外约束。

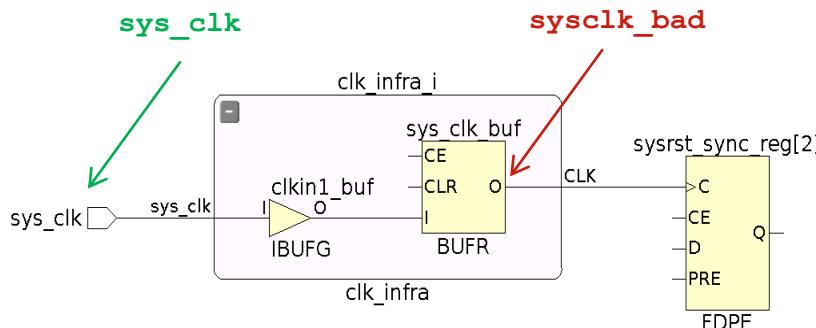


高级时钟约束

约束最终是为了设计服务，所以要用好 XDC 就需要深入理解电路结构和设计需求。接下来我们就以常见 FPGA 设计中的时钟结构来举例，详细阐述 XDC 的约束技巧。

● 时序的零起点

用 create_clock 定义的主时钟的起点即时序的“零起点”，在这之前的上游路径延时都被工具自动忽略。所以主时钟创建在哪个“点”很重要，以下图所示结构来举例，分别于 FPGA 输入端口和 BUFG 输出端口创建一个主时钟，在时序报告中体现出的路径延时完全不同，很明显 sysclk_bad 的报告中缺少了之前一段的延时，时序报告不可信。



```
create_clock -name sysclk -period 10 [get_ports sys_clk]
```

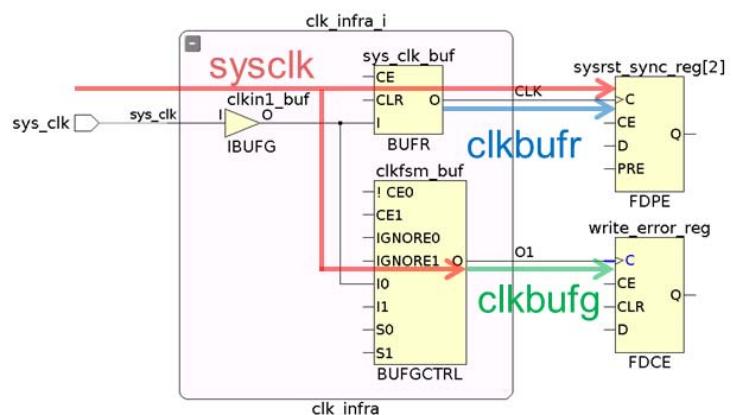
Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
(clock sysclk rise edge)	0.000	0.000 r	
	0.000	0.000 r	sys_clk
net (fo=0)	0.000	0.000	clk_infra_i/sys_clk
IBUFG (Prop_ibufg_I_O)	0.766	0.766 r	clk_infra_i/clkin1_buf/O
net (fo=3, unplaced)	1.109	1.875	clk_infra_i/clkin1
BUFR (Prop_bufr_I_O)	0.314	2.189 r	clk_infra_i/sys_clk_buf/O
net (fo=10, unplaced)	1.109	3.298	sys_clk_int
		r	sysrst_sync_reg[2]/C

```
create_clock -name sysclk_bad -period 10 [get_pins clk_infra_i/sys_clk_buf/O]
```

Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
(clock sysclk_bad rise edge)	0.000	0.000 r	
	0.000	0.000 r	clk_infra_i/sys_clk_buf/O
BUFR			
net (fo=10, unplaced)	1.109	1.109	sys_clk_int
		r	sysrst_sync_reg[2]/C

● 时钟定义的先后顺序

时钟的定义也遵从 XDC/Tcl 的一般优先级，即：在同一个点上，由用户定义的时钟会覆盖工具自动推导的时钟，且后定义的时钟会覆盖先定义的时钟。若要二者并存，必须使用 -add 选项。



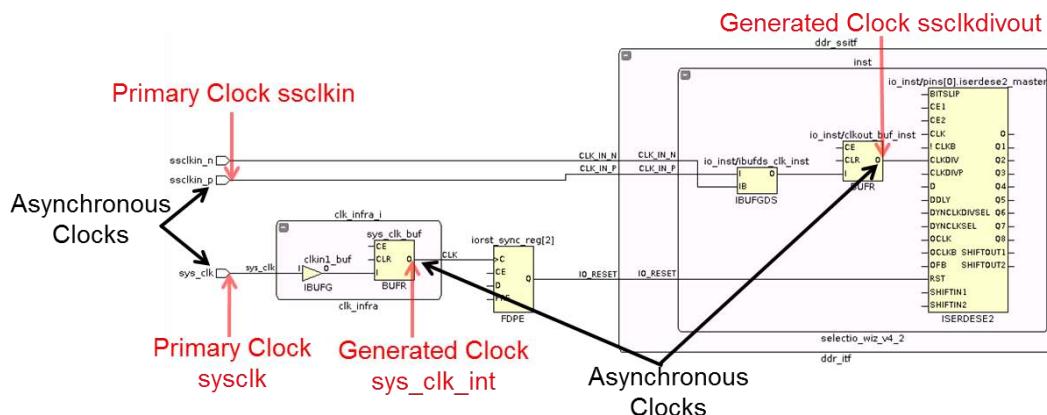
```
create_clock -name sysclk -period 10 [get_ports sys_clk]
create_generated_clock -name clkbufg -source [get_ports sys_clk] -divide_by 1 \
    [get_pins clk_infra_i/clkfsm_buf/O]
create_generated_clock -name clkbufr -source [get_ports sys_clk] -divide_by 1 \
    [get_pins clk_infra_i/sys_clk_buf/O] -add -master_clock sysclk
```

上述例子中 BUFG 的输出端由用户自定义了一个衍生钟 clkbufg，这个衍生钟便会覆盖此处原有的 sysclk。此外，图示 BUFR 工作在 bypass 模式，其输出不会自动创建衍生钟，但在 BUFR 的输出端定义一个衍生钟 clkbufr，并使用 -add 和 -master_clock 选项后，这一点上会存在 sysclk 和 clkbufg 两个重叠的时钟。如下的 Tcl 命令验证了我们的推论。

```
% get_clocks -of [get_pins sysrst_sync_reg[2]/C]
sysclk clkbufr
% get_clocks -of [get_pins write_error_reg/C]
clkbufg
```

● 同步时钟和异步时钟

不同于 UCF 约束，在 XDC 中，所有的时钟都会被缺省认为是相关的，也就是说，网表中所有存在的时序路径都会被 Vivado 分析。这也意味着 FPGA 设计人员必须通过约束告诉工具，哪些路径是无需分析的，哪些时钟域之间是异步的。



如上图所示，两个主时钟 ssclkin 和 sysclk 由不同的端口进入 FPGA，再经由不同的时钟网络传递，要将它们设成异步时钟，可以使用如下约束：

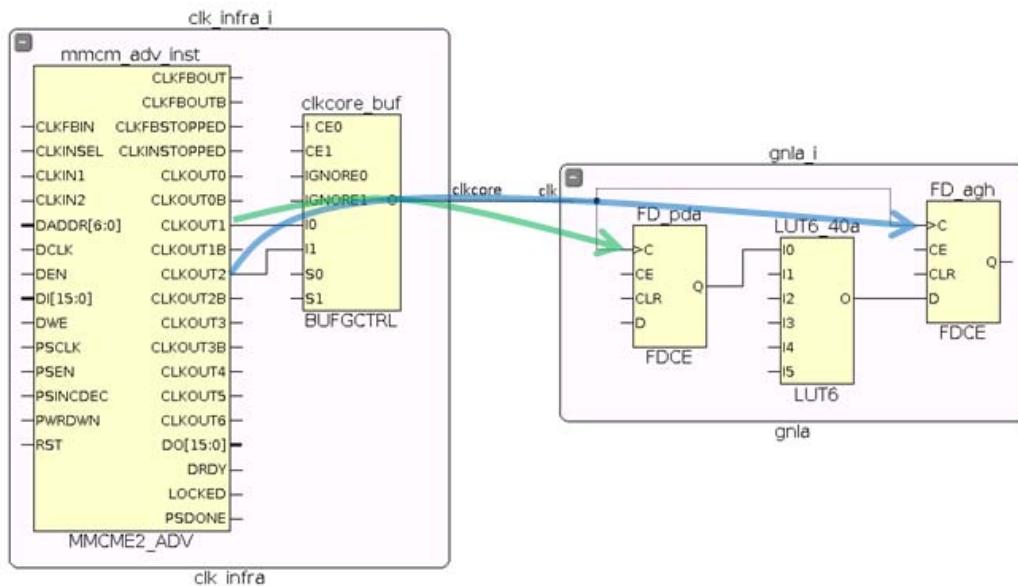
```
set_clock_groups -name sys_ss_async -asynchronous \
    -group [get_clocks -include_generated_clocks sysclk] \
    -group [get_clocks -include_generated_clocks ssclkin]
```

其中，**-include_generated_clocks** 表示所有衍生钟自动跟其主时钟一组，从而与其它组的时钟之间为异步关系。不加这个选项则仅仅将时钟关系的约束应用在主时钟层面。

● 重叠（单点多个）时钟

重叠时钟是指多个时钟共享完全相同的时钟传输网络，例如两个时钟经过一个 MUX 选择后输出的时钟，在有多种运行模式的设计中很常见。

如下图所示，clk125 和 clk250 是 clkcore_buf 的两个输入时钟，不约束时钟关系的情况下，Vivado 会对图示路径做跨时钟域（重叠时钟之间）分析。这样的时序报告即便没有违例，也是不可信的，因为 clk125 和 clk250 不可能同时驱动这条路径上的时序元件。这么做也会增加运行时间，并影响最终的实现效果。



Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
(clock clk125_rise edge)	0.000	0.000 r	
	0.000	0.000 r sys_clk	
...			
BUFGCTRL (Prop_bufgct...)	0.093	-3.307 r clk_infra_i/clkcore_buf/O	
net (fo=4900, unplaced)	1.109	-2.198 r gnla_i/clk	
		r gnla_i/FD_pda/C	
FDCE (Prop_fdce_C_Q)	0.249	-1.949 r gnla_i/FD_pda/Q	
...			
FDCE (Setup_fdce_C_D)	-0.003	-1.092 gnla_i/FD_agh/D	
(clock clk250_rise edge)	4.000	4.000 r	
	0.000	4.000 r sys_clk	
...			
BUFGCTRL (Prop_bufgct...)	0.083	1.416 r clk_infra_i/clkcore_buf/O	
net (fo=4900, unplaced)	1.109	2.525 r gnla_i/clk	
		r gnla_i/FD_agh/C	
clock pessimism	-0.733	1.792	
clock uncertainty	-0.191	1.601	
required time		1.601	
arrival time		1.092	
slack		2.693	

Different clocks

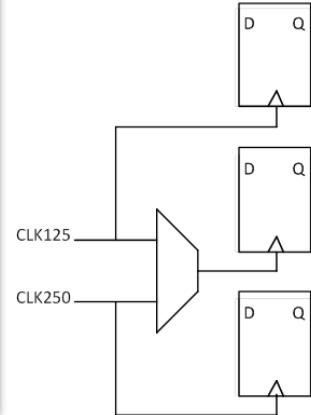
Same clock tree

如果 clk125 和 clk250 除了通过 clkcore_buf 后一模一样的扇出外没有驱动其它时序元件，我们要做的仅仅是补齐时钟关系的约束。

```
set_clock_groups -physically_exclusive \
    -group [get_clocks clk125] \
    -group [get_clocks clk250]
```

在很多情况下，除了共同的扇出，其中一个时钟或两个都还驱动其它的时序元件，此时建议的做法是在 clkcore_buf 的输出端上创建两个重叠的衍生钟，并将其时钟关系约束为 **-physically_exclusive** 表示不可能同时通过。这样做可以最大化约束覆盖率，也是 ISE 和 UCF 中无法做到的。

```
create_generated_clock -name clk125_bufgctrl \
    -divide_by 1 [get_pins bufctrl_i/O] \
    -source [get_ports bufctrl_i/I0]
create_generated_clock -name clk250_bufgctrl \
    -divide_by 1 [get_pins bufctrl_i/O] \
    -source [get_ports bufctrl_i/I1] \
    -add -master_clock clk250
set_clock_groups -physically_exclusive \
    -group clk125_bufgctrl \
    -group clk250_bufgctrl
```



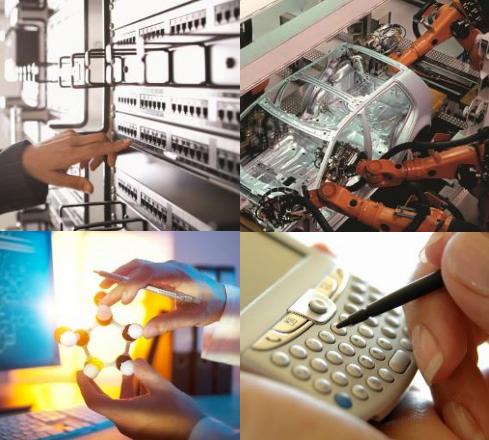
其它高级约束

时钟的约束是 XDC 的基础，熟练掌握时钟约束，也是 XDC 约束技巧的基础。其它高级约束技巧，包括复杂的 CDC (Clock Domain Crossing) 约束和接口时序 (SDR、DDR、系统同步接口和源同步接口) 约束等方面还有很多值得注意的地方。

这一系列《XDC 约束技巧》文章还会继续就上述所列方向分篇详述，敬请关注作者的后续更新，以及 Xilinx 官方网站和中文论坛上的更多技术文章。

—— Ally Zhou , 2014-9-25 于 Xilinx 上海 Office

[返回目录页](#)



XDC 约束技巧之 CDC 篇

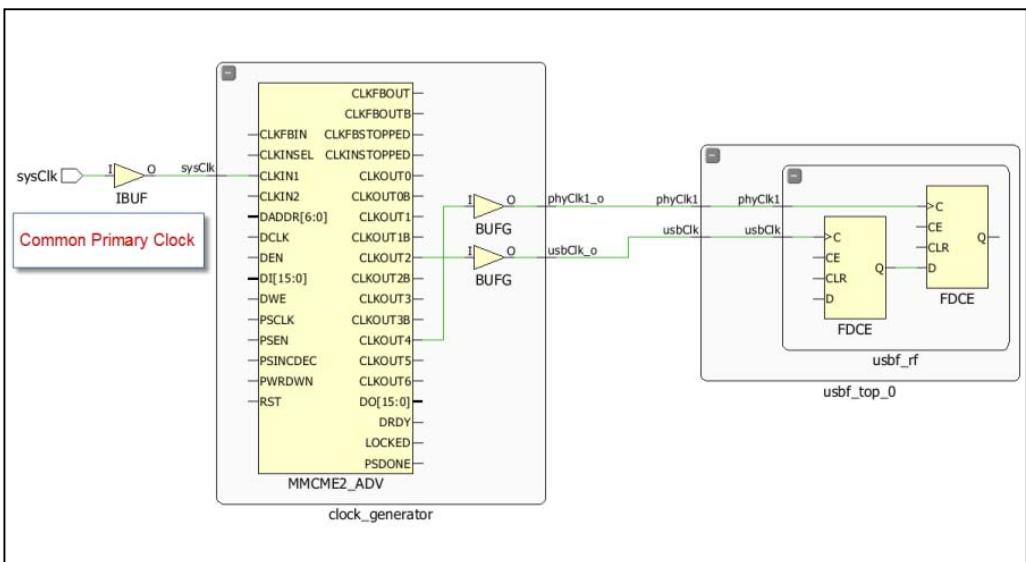
上一篇《XDC 约束技巧之时钟篇》介绍了 XDC 的优势以及基本语法，详细说明了如何根据时钟结构和设计要求来创建合适的时钟约束。我们知道 XDC 与 UCF 的根本区别之一就是对跨时钟域路径（CDC）的缺省认识不同，那么碰到 FPGA 设计中常见的 CDC 路径，到底应该怎么约束，在设计上又要注意些什么才能保证时序报告的准确性？

CDC 的定义与分类

CDC 是 Clock Domain Crossing 的简称，CDC 时序路径指的是起点和终点由不同时钟驱动的路径。在电路设计中对这些跨时钟域路径往往需要进行特别的处理来避免亚稳态的产生，例如使用简单同步器、握手电路或是 FIFO 来隔离。

安全的 CDC 路径

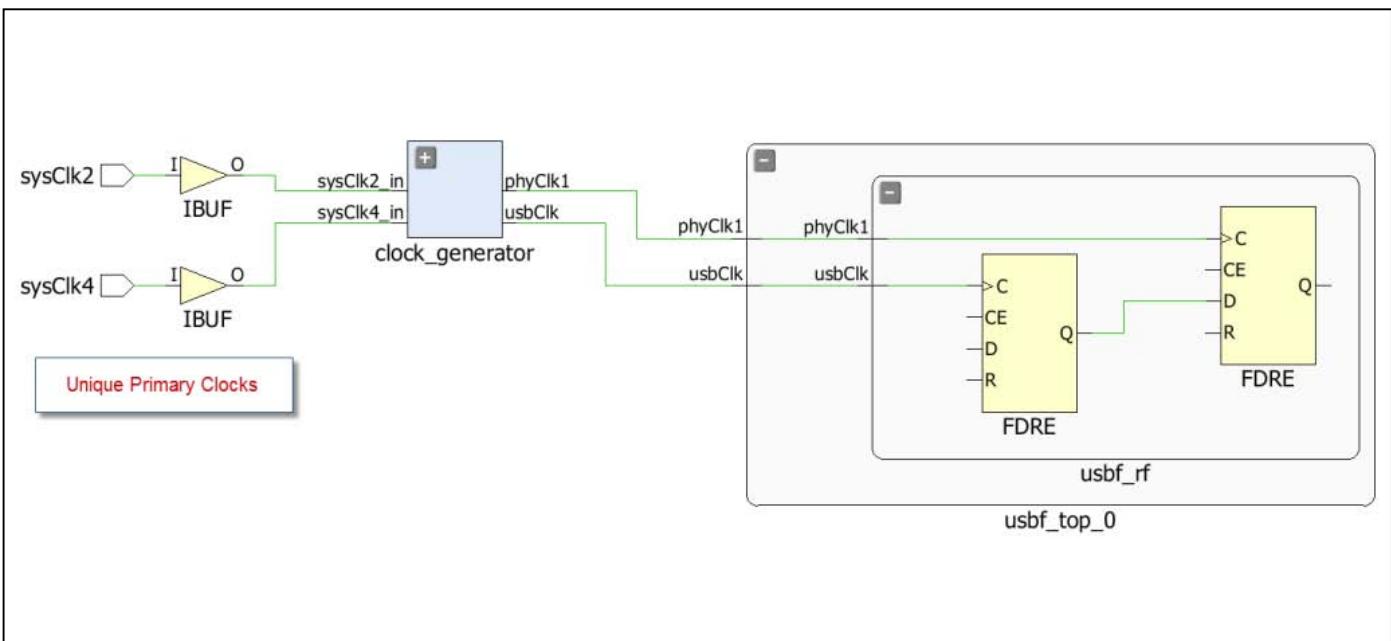
所谓安全的 CDC 路径是指那些源时钟和目标时钟拥有相同的来源，在 FPGA 内部共享部分时钟网络的时序路径。这里的安全指的是时钟之间的关系对 Vivado®来说是全透明可分析的。



不安全的 CDC 路径

不安全的 CDC 路径则表示源时钟和目标时钟不同，且由不同的端口进入 FPGA，在芯片内部不共享时钟网络。这种情况下，Vivado 的报告也只是基于端口处创建的主时钟在约束文件中所描述的相位和频率关系来分析，并不能代表时钟之间真实的关系。

VIVADO®

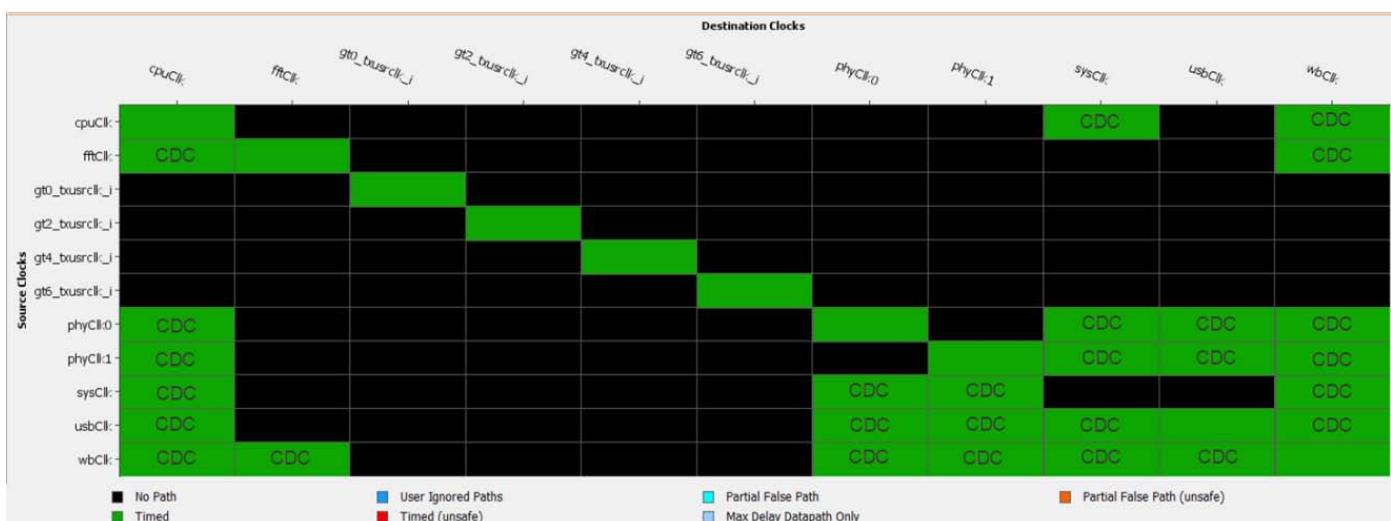


在 Vivado 中分析 CDC

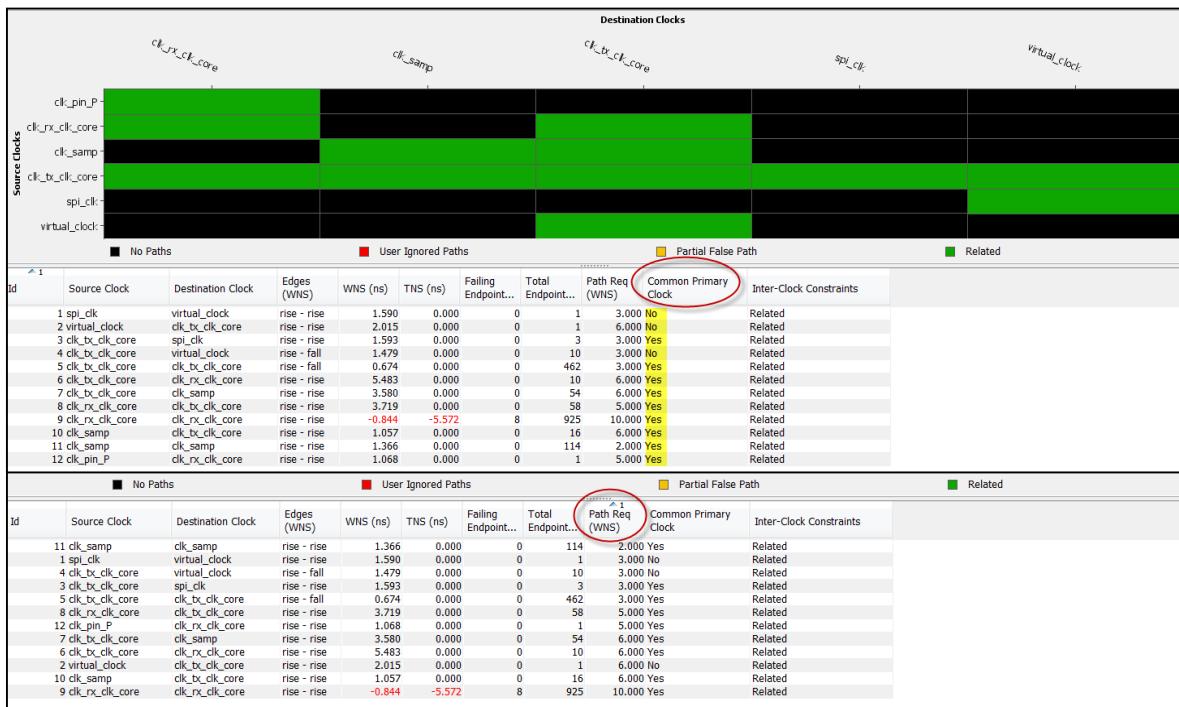
在 ISE 中想要快速定位那些需要关注的 CDC 路径并不容易，特别是要找到不安全的 CDC 时，因为 ISE 缺省认为所有来源不同的时钟都不相关且不做分析，要报告出这类路径，需要使用 ISE Timing Analyzer (TRCE)，并加上 “-u” (表示 unconstrained) 这个选项。

在 Vivado 中则容易许多，我们可以使用 `report_clock_interaction` 命令 (GUI 支持) 来鉴别和报告设计中所有的时钟关系。执行命令后会生成一个矩阵图，其中对角线上的路径表示源时钟与目标时钟相同的时钟内部路径，其余都是 CDC 路径。

Vivado 还会根据网表和已读入的约束分析出 CDC 路径的约束情况，并分颜色表示。例如绿色代表有时序约束，红色代表不安全的 CDC 路径但是没有约束时序例外，橙色表示有部分路径已约束为 false path 的不安全 CDC 路径。



矩阵下方是时钟关系表格，可以就各种条件进行筛选和排序，方便定位 CDC 路径。建议的做法是：首先，对“Common Primary Clock”排序 (显示为 Yes 或 No)，这么做可以快速鉴别出那些安全和不安全的 CDC 路径，接着观察对应的“Inter-Clock Constraints”栏内的内容，判断已读入的 XDC 中是否对这类路径进行了合理的约束。



第二步，可以对“Path Req (WNS)”由小到大进行排序，找到那些数值特别小（例如小于 100ps）或是显示为“Unexpanded”的 CDC 路径，结合是否共享“Common Primary Clock”来鉴别此类路径，作出合理的约束。

过小的 Path Req (WNS)一般都表示此类跨时钟域路径缺少异步时钟关系或其它时序例外的约束，如果两个时钟连“Common Primary Clock”也不共享，则 100%可以确认为异步时钟，应该加上相应的时钟关系约束。

显示为“Unexpanded”的时钟关系，表示 Vivado 在一定长度（缺省为 1000）的周期内都没有为两个时钟的频率和相位找到固定的关系，则无法推导出相应的 Path Req 约束值。此类 CDC 需要特别留意，也要加上异步时钟关系约束。

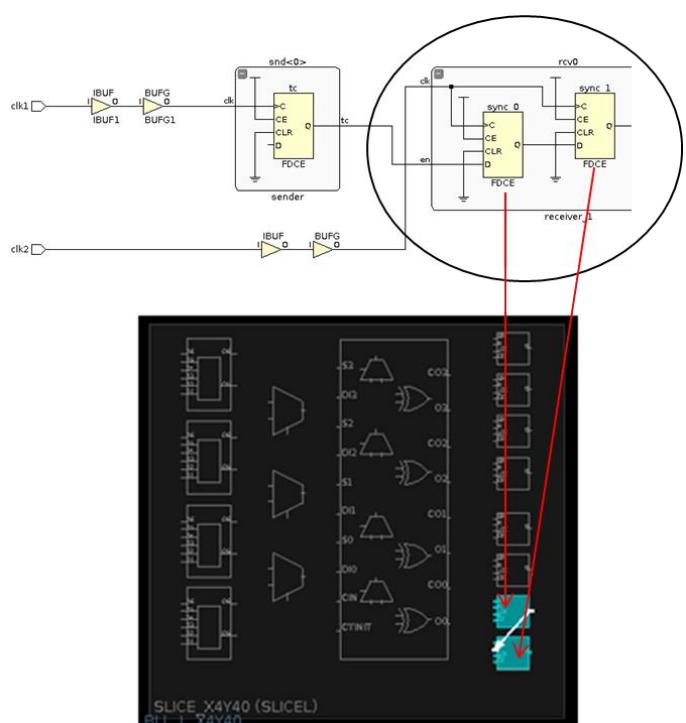
这个矩阵还支持交互式的时序分析，选中任意一个方框，右键显示下拉菜单：选择 Report Timing，会报告出这一格代表的时钟域（本时钟域或是跨时钟域）内最差的时序路径；选择 Set Clock Groups 则可以设置时钟关系约束并添加到 XDC 文件中。

CDC 的设计与约束

CDC 路径在 FPGA 设计中普遍存在，在设置相应的约束前，必须了解设计中采取了怎样的方法来处理跨时钟域路径。

简单同步器

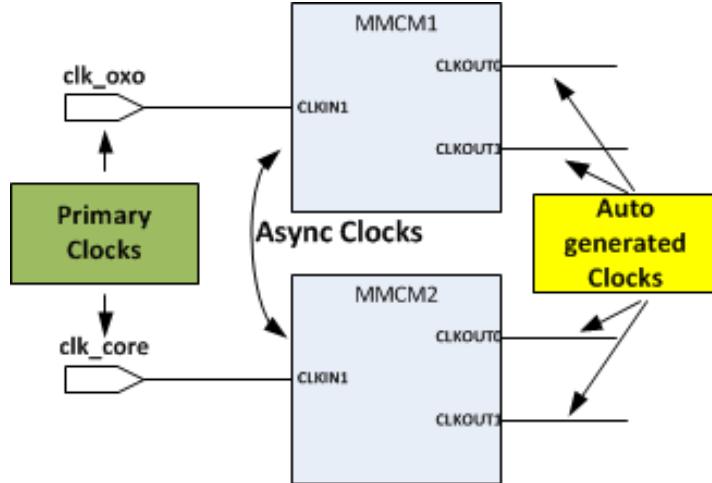
对于单根跨时钟域路径，一般采用简单同步器（Simple Synchronizer），就是由至少两级 CE 端和 Reset/Clear 端接死的寄存器序列来处理。



这种情况下，为了更长的平均无故障时间 MTBF (Mean Time Between Failures)，需要配合一个 ASYNC_REG 的约束，把用作简单同步器的多个寄存器放入同一个 SLICE，以降低走线延时的不一致和不确定性。

```
set_property ASYNC_REG TRUE [get_cells [list sync0_reg sync1_reg]]
```

在 XDC 中，对于此类设计的 CDC 路径，可以采用 set_clock_groups 来约束。



```
set_clock_groups -asynchronous -group [get_clocks -include_generated_clocks clk_oxo] \
                    -group [get_clocks -include_generated_clocks clk_core]
```

用 FIFO 隔离 CDC

在总线跨时钟域的设计中，通常会使用异步 FIFO 来隔离。根据 FIFO 的实现方式不同，需要加入不同的 XDC 约束。

- Build-in 硬核 FIFO

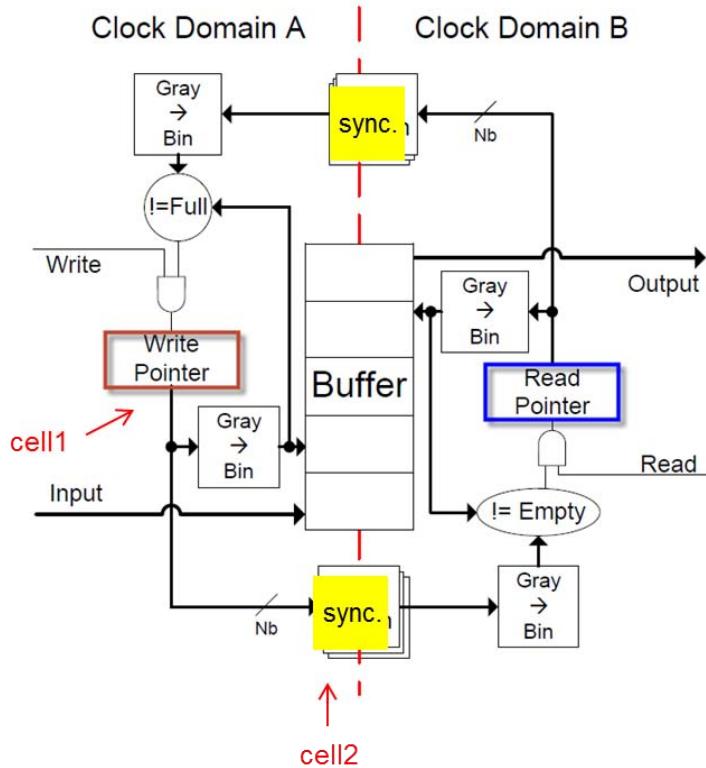
这种 FIFO 实际上就是用 FPGA 内部的 BRAM 来搭建，所有控制逻辑都在 BRAM 内部，是推荐的 FIFO 实现方式。其所需的 XDC 也相对简单，只要像上述简单同步器的时钟关系约束一样用 set_clock_groups 将读写时钟约束为异步即可。

- 带有格雷码控制的 FIFO

为了在亚稳态下做读写指针抽样也能正确判断空满状态，设计中也常用一种带有格雷码控制的 FIFO 来实现异步时钟域的隔离。计数器和读写指针等需要用 BRAM 外部的逻辑搭建，这样的结构就不能简单约束 set_clock_groups，还要考虑这些外部逻辑如何约束。

如下图所示 FIFO，在存储器外部有一些用 FPGA 逻辑搭建的写指针和读指针控制，分属不同的时钟域，存在跨时钟域的时序路径。

此时如果仅将读写时钟用 set_clock_groups 约束为异步时钟，相当于设置从 A 到 B 和从 B 到 A 的路径全部为 false path。根据 [《XDC 约束技巧之时钟篇》](#) 所列，false path 的优先级最高，很显然这么做会导致所有跨读写时钟域的路径全部不做时序分析，读写指针和相关控制逻辑也就失去了存在的意义。



所以建议的做法是不设 `set_clock_groups` 约束，转而采用 `set_max_delay` 来约束这些跨时钟域路径。以写入侧举例，一个基本的原则就是约束从 cell1 到 cell2 的路径之间的延时等于或略小于 cell2 的驱动时钟一个周期的值。读出侧的约束同理。

```
set_max_delay $delay -from [get_cells
cell1] -to [get_cells cell2] -datapath_only
```

如果用户使用 Vivado 的 IP Catalog 来产生此类 FIFO，这样的 XDC 会随 IP 的源代码一起输出（如下所示），使用者仅需注意确保这个 FIFO 的读写时钟域没有被用户自己的 XDC 约束为 false path 或是异步 clock groups。

```
set_max_delay -from [get_cells ...../rd_pntr_gc_reg[*]] -to [get_cells ...../Q_reg_reg[*]] \
-datapath_only [get_property PERIOD $rd_clock]
set_max_delay -from [get_cells ...../wr_pntr_gc_reg[*]] -to [get_cells ...../Q_reg_reg[*]] \
-datapath_only [get_property PERIOD $wr_clock]
```

自 2013.4 开始，Vivado 中还提供一个称作 `methodology_checks` 的 DRC 检查，其中包含有对此类异步 FIFO 的 max delay 约束与时钟域 clock groups 约束的冲突检查。

CDC 约束方案的对比

CDC 路径在 FPGA 设计中普遍存在，不少公司和研发人员都有自己偏爱的约束方式，这些方式通常有各自适用的环境，当然也各有利弊。

● 全部忽略的约束

最大化全部忽略 CDC 路径的约束，即采用 `set_clock_groups` 或是 `set_false_path` 对时钟关系进行约束，从而对跨时钟域的路径全部忽略。

- 示例：`set_clock_groups -asynchronous -group clkA -group clkB`
- 优势：简单、快速、执行效率高。
- 劣势：会掩盖时序报告中所有的跨时钟域路径，容易误伤，不利于时序分析。

● 使用 `datapath_only` 约束

`datapath_only` 是从 ISE 时代的 UCF 中继承过来的约束，在 XDC 中必须作为一个选项跟 `set_max_delay` 配合使用，可以约束在时钟之间，也可以对具体路径进行约束。

- 示例：`set_max_delay 10 -datapath_only -from clkA -to clkB`
- 优势：简便、执行效率较高。
- 劣势：1) 需要特别留意是否设置了过于严格的约束，因为使用者经常会使用较快的时钟周期来约束跨时钟域路径。2) 注意约束优先级的关系，是否跟设计中其它的约束有冲突。3) `set_max_delay` 而没有配套设置 `set_min_delay` 的情况下，同一路径只做 setup 分析而不做 hold 分析。

● 逐条进行时序例外约束

对设计中的 CDC 路径分组或逐条分析，采用不同的时序例外约束，如 set_false_path，set_max_delay 和 set_multicycle_path 等来约束。

- **示例**：set_false_path -from [get_cells a/b/c/*_meta*] -to [get_cells a/b/c/*_sync*]
- **优势**：灵活、针对性好、便于时序分析和调试。
- **劣势**：1) 逐条约束会占用大量时间来调试和分析，效率低下。2) 时序例外的优先级比较复杂，多种时序例外约束共存的情况下，很容易产生意想不到的冲突，进一步增加调试时间，降低效率。3) 这么做极容易产生臃肿的 XDC 约束文件，而且时序例外的执行更耗内存，直接导致工具运行时间变长。

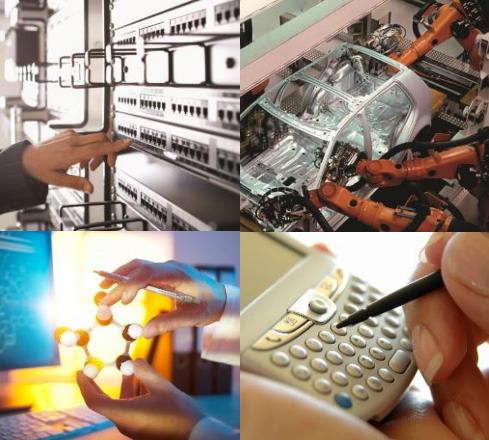
小结

CDC 路径的分析和约束不仅在 FPGA 设计中至关重要，也是数字电路设计领域一个非常重要的命题。IP 提供商、EDA 公司都有不少关于 CDC 的技术文档。Vivado 相比于 Xilinx 上一代产品 ISE，已经在 CDC 的鉴别和分析方面有了很大改进，XDC 相比于 UCF，在 CDC 路径的约束上也更为高效，覆盖率更高。

希望本篇短文可以帮助 Vivado 的用户快速掌握对 FPGA 设计中 CDC 路径的鉴别、分析和约束方法，提高设计效率。

— Ally Zhou, 2014-9-28 于 Xilinx 上海 Office

[返回目录页](#)



XDC 约束技巧之 I/O 篇 (上)

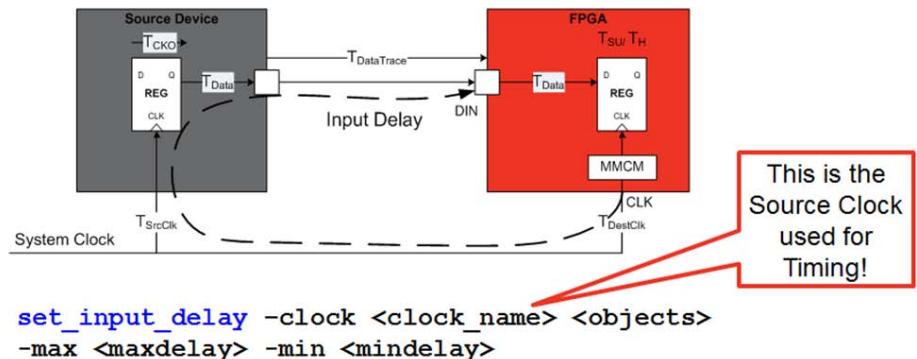
《XDC 约束技巧之时钟篇》中曾对 I/O 约束做过简要概括，相比较而言，XDC 中的 I/O 约束虽然形式简单，但整体思路和约束方法却与 UCF 大相径庭。加之 FPGA 的应用特性决定了其在接口上有多种构建和实现方式，所以从 UCF 到 XDC 的转换过程中，最具挑战的可以说便是本文将要讨论的 I/O 约束了。

I/O 约束的语法

XDC 中可以用于 I/O 约束的命令包括 `set_input_delay` / `set_output_delay` 和 `set_max_delay` / `set_min_delay`。其中，只有那些从 FPGA 管脚进入和/或输出都不经过任何时序元件的纯组合逻辑路径可以用 `set_max_delay` / `set_min_delay` 来约束，其余 I/O 时序路径都必须由 `set_input_delay` / `set_output_delay` 来约束。如果对 FPGA 的 I/O 不加任何约束，Vivado 会默认认为时序要求为无穷大，不仅综合和实现时不会考虑 I/O 时序，而且在时序分析时也不会报出这些未约束的路径。

本文以下章节将会着重讨论 XDC 接口约束和时序分析的基础，介绍如何使用 `set_input_delay` / `set_output_delay` 对 FPGA 的 I/O 时序进行约束。

Input 约束



上图所示 `set_input_delay` 的基本语法中，`<objects>` 是想要设定 input 约束的端口名，可以是一个或数个 port；`-clock` 之后的 `<clock_name>` 指明了对 `<objects>` 时序进行分析所用的时钟，可以是一个 FPGA 中真实存在的时钟也可以是预先定义好的虚拟时钟；`-max` 之后的 `<maxdelay>` 描述了用于 setup 分析的包含有板级走线和外部器件的延时；`-min` 之后的 `<mindelay>` 描述了用于 hold 分析的包含有板级走线和外部器件的延时。

上述这些选项是定义 Input 约束时必须写明的，还有少数几个可选项，如 `-add_delay` 和 `-clock_fall` 用于 DDR 接口的约束。

主要文档

- 即插即用 IP 背景资料
- 【中文】UG949 - UltraFast 设计方法指南
- UG1046 - UltraFast 嵌入式设计方法指南
- Vivado Design Suite 加速设计生产力的九大理由
- Vivado IP Integrator 背景资料

快速链接

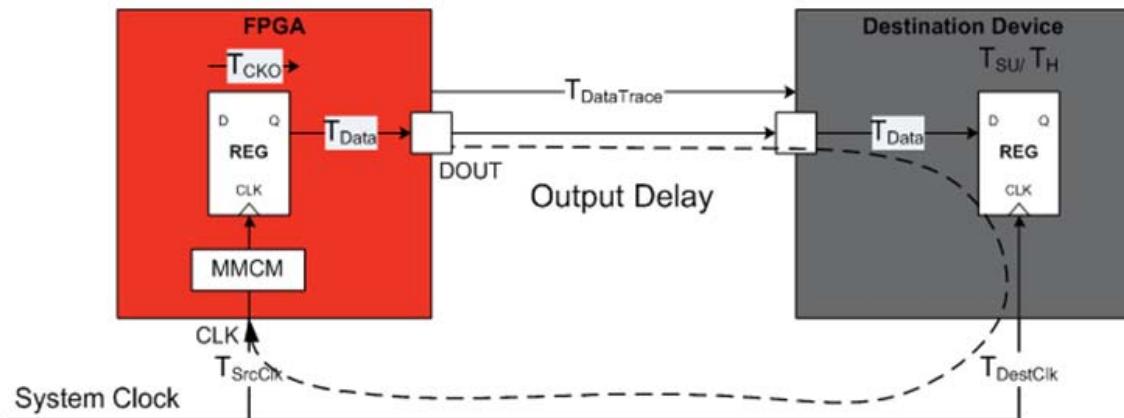
- 免费下载：Vivado Design Suite 评估和 WebPACK 版本
- 下载
- 支持和技术文档
- Vivado 视频辅导资料
- IP 中心
- 支持的目标参考设计
- 存储器推荐

培训与活动

- Vivado 课程
- 高层次综合课程

VIVADO

Output 约束



```
set_output_delay -clock <clock_name> <objects>
-max <maxdelay> -min <mindelay>
```

上图所示 set_output_delay 的基本语法中，<objects> 是想要设定 output 约束的端口名，可以是一个或数个 port；-clock 之后的 <clock_name> 指明了对 <objects> 时序进行分析所用的时钟，可以是一个 FPGA 中真实存在的时钟也可以是预先定义好的虚拟时钟；-max 之后的 <maxdelay> 描述了用于 setup 分析的包含有板级走线和外部器件的延时；-min 之后的 <mindelay> 描述了用于 hold 分析的包含有板级走线和外部器件的延时。

上述这些选项是定义 Output 约束时必须写明的，还有少数几个可选项如 -add_delay 和 -clock_fall 用于 DDR 接口的约束。

Setup/Hold 时序分析

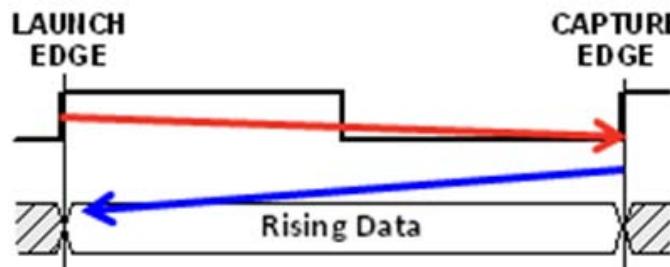
我们知道约束是为了设计服务，而设置好的约束必须在时序报告中加以验证。所以，怎样理解时序分析中的检查规则就成了重中之重，这一点对 I/O 约束来说尤为重要。理解时序分析工具如何选取路径分析的发送端（Launch）和接收端（Capture）时钟沿（Clock Edges），在 Setup 和 Hold 分析时又有怎样的具体区别，以及这些数字在时序报告中如何体现等等是设置正确 I/O 约束的基础。

更具体的时序分析方法以及如何深入解读时序报告等内容将会在后续另开主题文章详述，这里仅就 Setup/Hold 分析时对时钟边沿的选择加以描述，便于以下章节的展开。

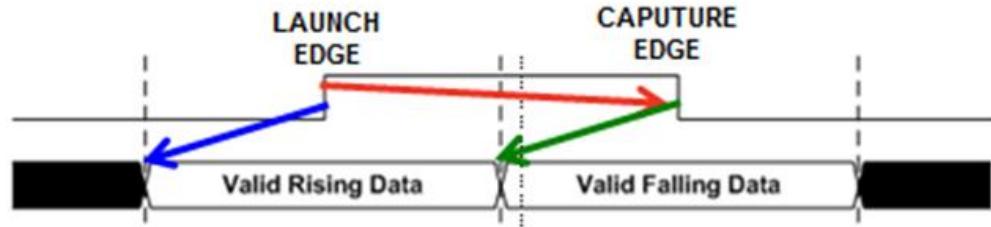
Setup 时序分析

同步电路设计中，一般情况下，数据在时钟上升沿发送，在下一个时钟上升沿接收，发送的时钟沿称作 Launch Edge，接收沿称作 Capture Edge。时序分析中的 Setup Check 跟 Capture Edge 的选择息息相关。

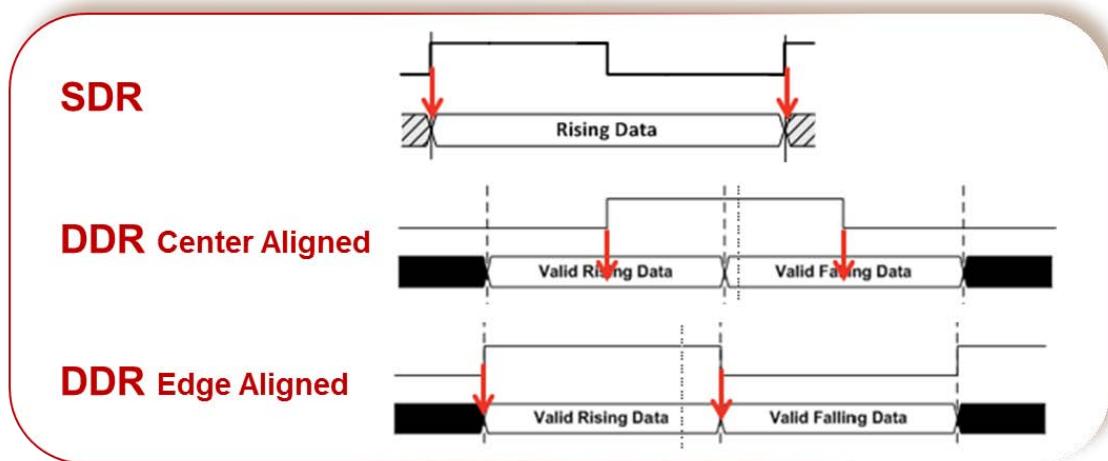
在 SDR 接口的 setup 分析中，工具如下图这样识别发送和接收时钟沿。



而在 DDR 接口的 setup 分析中，因为数据是双沿采样，所以发送和接收时钟沿变成上升（下降）沿发送，下降（上升）沿接收。



Hold 时序分析



Hold Check 主要是为了保证数据在接收（采样）端时钟沿之后还能稳定保持一段时间，对 Hold 分析而言，同一个时钟沿既是 Launch Edge 也是 Capture Edge，这一点对 SDR 和 DDR（不论是中心对齐还是边沿对齐）都一样。

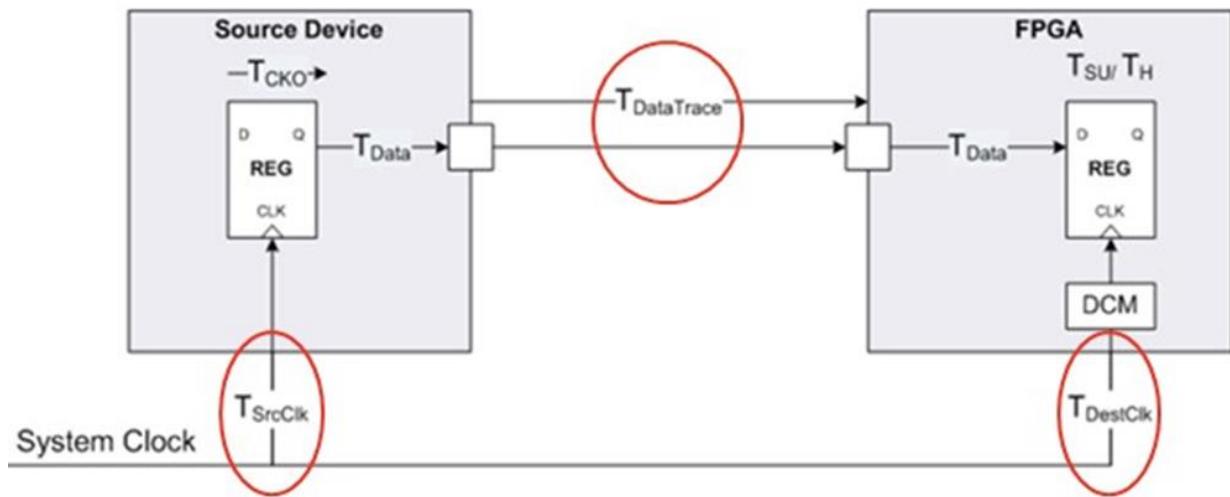
Input 接口类型和约束

由于历史的原因，相当一部分 FPGA 设计仍然在系统中起到胶合逻辑（Glue Logic）的作用，当然，如今的 FPGA 中嵌入了高速串行收发器和嵌入式处理器等，早就不仅仅局限于系统设计的配角，反而成为了其中的主角甚至是明星。但数据接口的同步一直是 FPGA 设计中的常见问题，也是一个重点和难点，很多设计不稳定都是因为数据接口的同步有问题。

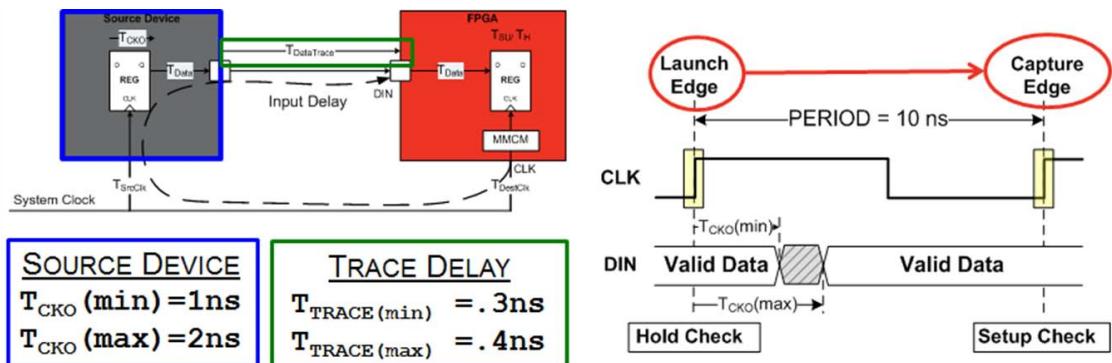
FPGA 的数据接口同步根据系统级设计方式来讲可以分为系统同步和源同步两种。

系统同步接口

系统同步接口（System Synchronous Interface）的构建相对容易，以 FPGA 做接收侧来举例，上游器件仅仅传递数据信号到 FPGA 中，时钟信号则完全依靠系统板级来同步。时钟信号在系统级上同源，板级走线的延时也要对齐。正因为这样的设计，决定了数据传递的性能受到时钟在系统级的走线延时和 skew 以及数据路径延时的双重限制，无法达到更高速的设计要求，所以大部分情况也仅仅应用 SDR 方式。



对系统同步接口做 Input 约束相对容易，只需要考虑上游器件的 T_{CKO} 和数据在板级的延时即可。下图是一个 SDR 上升沿采样系统同步接口的 Input 约束示例。



```
create_clock -name sysclk -period 10      [get_ports CLK];
set_input_delay -clock sysclk -max 2.4 [get_ports DIN];
set_input_delay -clock sysclk -min 1.3 [get_ports DIN];
```

设置和分析 I/O 约束一定要有个系统级思考的视角，如上右图所示，Launch Edge 对应的是上游器件的时钟，而 Capture Edge 则对应 FPGA 的输入时钟，正因为是系统同步时钟，所以可以将其视作完全同步而放在一张图上分析，这样一来，就可以用一般时序分析方法来看待问题。

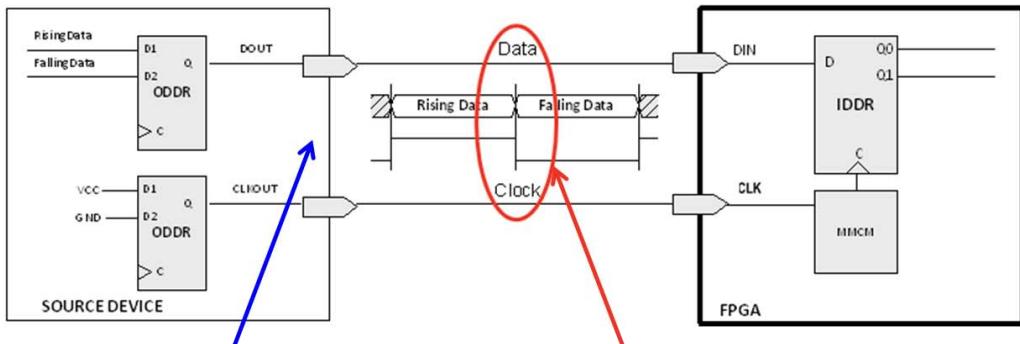
一条完整的时序路径，从源触发器的 C 端开始，经过 T_{CKO} 和路径传输延时再到目的触发器的 D 端结束。放在系统同步的接口时序上，传输延时则变成板级传输延时（还要考虑 skew），所以上述 -max 后的数值是 T_{CKO} 的最大值加上板级延时的最大值而来，而-min 后的数值则是由两个最小值相加而来。

源同步接口

为了改进系统同步接口中时钟频率受限的弊端，一种针对高速 I/O 的同步时序接口应运而生，在发送端将数据和时钟同步传输，在接收端用时钟沿脉冲来对数据进行锁存，重新使数据与时钟同步，这种电路就是源同步接口电路（Source Synchronous Interface）。

源同步接口最大的优点就是大大提升了总线的速度，在理论上信号的传送可以不受传输延迟的影响，所以源同步接口也经常应用 DDR 方式，在相同时钟频率下提供双倍于 SDR 接口的数据带宽。

源同步接口的约束设置相对复杂，一则是因为有 SDR、DDR、中心对齐（Center Aligned）和边沿对齐（Edge Aligned）等多种方式，二则可以根据客观已知条件，选用与系统同步接口类似系统级视角的方式来设置约束。



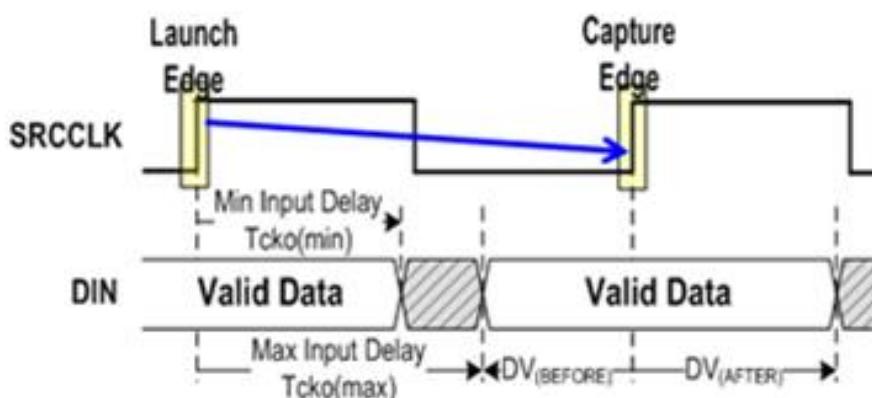
方法 1: System Method

- 已知上游器件的Tcko值
- 无需接口的数据有效窗口

方法 2: Source Sync Method

- 已知接口的数据有效窗口
- 源同步接口约束的典型方式

如上图所示，对源同步接口进行 Input 约束可以根据不同的已知条件，选用不同的约束方式。一般而言，FPGA 作为输入接口时，数据有效窗口是已知条件，所以方法 2 更常见，Vivado IDE 的 Language Templates 中关于源同步输入接口 XDC 模板也是基于这种方法。但不论以何种方式来设置 Input 约束，作用是一样的，时序报告的结果也应该是一致的。



针对上图所示中心对齐源同步 SDR 接口时序，分别按照两种方式来约束，需要的已知条件和计算方式虽然不同，但却可以得到完全一样的结果。

System Method Parameters

Tcko Max	3ns
Tcko Min	2ns
Period	5ns

```
create_clock -name sysclk -period 5
set_input_delay -clock sysclk -max 3
set_input_delay -clock sysclk -min 2
```

```
[get_ports CLK];
[get_ports DIN];
[get_ports DIN];
```

Src Sync Method Parameters

DV _(BEFORE)	2ns
DV _(AFTER)	2ns
Period	5ns

$$\text{Max} = (\text{Period} - \text{DV}_{\text{BEFORE}})$$

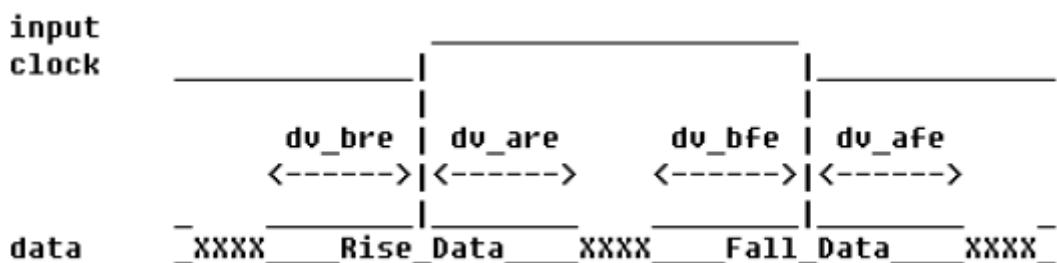
```
create_clock -name sysclk -period 5
set_input_delay -clock sysclk -max 3
set_input_delay -clock sysclk -min 2
```

[get_ports CLK];
[get_ports DIN];
[get_ports DIN];

DDR 接口的约束设置

DDR 源同步接口的约束稍许复杂，需要将上升沿和下降沿分别考虑和约束，以下以源同步接口为例，分别就输入接口数据为中心对齐或边沿对齐的方式来举例。

- DDR 源同步中心对齐输入接口



已知条件如下：

- ✓ 时钟信号 src_sync_ddr_clk 的频率： 100 MHz
- ✓ 数据总线： src_sync_ddr_din[3:0]
- ✓ 上升沿之前的数据有效窗口 (dv_bre) : 0.4 ns
- ✓ 上升沿之后的数据有效窗口 (dv_are) : 0.6 ns
- ✓ 下降沿之前的数据有效窗口 (dv_bfe) : 0.7 ns
- ✓ 下降沿之后的数据有效窗口 (dv_afe) : 0.2 ns

可以这样计算输入接口约束：DDR 方式下数据实际的采样周期是时钟周期的一半；上升沿采样的数据 (Rise Data) 的 -max 应该是采样周期减去这个数据的发送沿 (下降沿) 之前的数据有效窗口值 dv_bfe，而对应的-min 就应该是上升沿之后的数据有效窗口值 dv_are；同理，下降沿采样的数据 (Fall Data) 的 -max 应该是采样周期减去这个数据的发送沿 (上升沿) 之前的数据有效窗口值 dv_bre，而对应的-min 就应该是下降沿之后的数据有效窗口值 dv_afe。

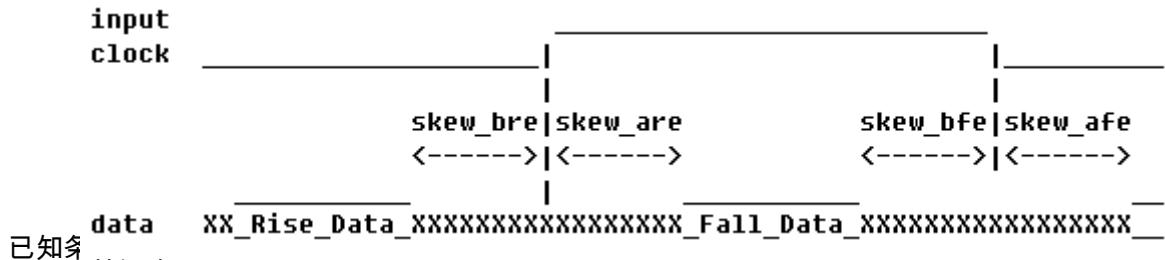
所以最终写入 XDC 的 Input 约束应该如下所示：

```

set period 10.0;
create_clock -period $period -name clk [get_ports src_sync_ddr_clk];
set_input_delay -clock clk -max [expr $period/2 - 0.7] [get_ports src_sync_ddr_din[*]];
set_input_delay -clock clk -min 0.6 [get_ports src_sync_ddr_din[*]];
set_input_delay -clock clk -max [expr $period/2 - 0.4] \
    [get_ports src_sync_ddr_din[*]] -clock_fall -add_delay;
set_input_delay -clock clk -min 0.2 [get_ports src_sync_ddr_din[*]] -clock_fall -add_delay;

```

- DDR 源同步边沿对齐输入接口



- ✓ 时钟信号 `src_sync_ddr_clk` 的频率： 100 MHz
- ✓ 数据总线： `src_sync_ddr_din[3:0]`
- ✓ 上升沿之前的数据 skew (`skew_bre`) : 0.6 ns
- ✓ 上升沿之后的数据 skew (`skew_are`) : 0.4 ns
- ✓ 下降沿之前的数据 skew (`skew_bfe`) : 0.3 ns
- ✓ 下降沿之后的数据 skew (`skew_afe`) : 0.7 ns

可以这样计算输入接口约束：因为已知条件是数据相对于时钟上升沿和下降沿的 skew，所以可以分别独立计算；上升沿的 `-max` 是上升沿之后的数据 skew (`skew_are`)，对应的 `-min` 就应该是负的上升沿之前的数据 skew (`skew_bre`)；下降沿的 `-max` 是下降沿之后的数据 skew (`skew_afe`)，对应的 `-min` 就应该是负的下降沿之前的数据 skew (`skew_bfe`)。

所以最终写入 XDC 的 Input 约束应该如下所示：

```

create_clock -period 10.0 -name clk [get_ports src_sync_ddr_clk];
set_input_delay -clock clk -max 0.4 [get_ports src_sync_ddr_din[*]];
set_input_delay -clock clk -min -0.6 [get_ports src_sync_ddr_din[*]];
set_input_delay -clock clk -max 0.7 [get_ports src_sync_ddr_din[*]] -clock_fall -add_delay;
set_input_delay -clock clk -min -0.3 [get_ports src_sync_ddr_din[*]] -clock_fall -add_delay;

```

出现负值并不代表延时真的为负，而是跟数据相对于时钟沿的方向有关。请一定牢记 `set_input_delay` 中 `-max/-min` 的定义，即时钟采样沿到达之后最大与最小的数据有效窗口（`set_output_delay` 中 `-max/-min` 的定义与之正好相反，详见后续章节举例说明）。

在这个例子中，数据是边沿对齐，只要有 jitter 跟 skew 的存在，最差情况下，数据有效窗口在到达时钟采样沿之前就已经结束，所以会有负数出现在 `-min` 之后。因此，在实际应用中，FPGA 用作输入的边沿对齐 DDR 源同步接口的情况下，真正用来采样数据的时钟会经过一个 MMCM/PLL 做一定的相移，从而把边沿对齐

变成中心对齐。

另外，在经过 MMCM/PLL 相移后的采样时钟跟同步接口输入的时钟之间需要做 set_false_path 的约束（如下述例子）而把那些伪路径从时序报告中剔除，这里不再详述。

```
set_false_path -setup -rise_from [get_clocks adc_dclk_p] -rise_to [get_clocks clk_outp0_adc_pll_1]
set_false_path -setup -fall_from [get_clocks adc_dclk_p] -fall_to [get_clocks clk_outp0_adc_pll_1]
set_false_path -hold -fall_from [get_clocks adc_dclk_p] -rise_to [get_clocks clk_outp0_adc_pll_1]
set_false_path -hold -rise_from [get_clocks adc_dclk_p] -fall_to [get_clocks clk_outp0_adc_pll_1]
```

虚拟时钟

在 FPGA 做系统同步输入接口的情况下，很多时候上游器件送入的数据并不是跟某个 FPGA 中已经存在的真实的时钟相关，而是来自于一个不同的时钟，这时就要用到虚拟时钟（Virtual Clock）。

举例来说，上游器件用一个 100MHz 的时钟送出数据到 FPGA，实际上这个数据每两个时钟周期才变化一次，所以可以用 50MHz 的时钟来采样。FPGA 有个 100MHz 的输入时钟，经过 MMCM 产生一个 50MHz 的衍生时钟，并用其来采样上游器件送来的同步数据。当然，系统级的设计上，必须有一定的机制来保证上游器件中的发送时钟和 FPGA 中的接收时钟的时钟沿对齐。

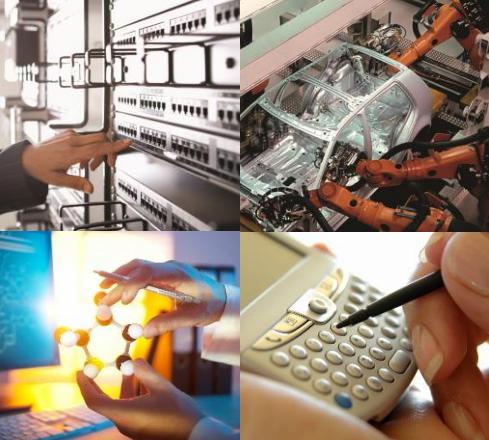
此时，我们可以借助虚拟时钟的帮助来完成相应的 Input 接口约束。

```
create_clock -period 10 -name clk_100 [get_ports i_clk_100MHz];
create_clock -period 20 -name clk_50_virtual;
set_input_delay -max 5.2 -clock clk_50_virtual [get_ports i_data_50];
set_input_delay -min 2.0 -clock clk_50_virtual [get_ports i_data_50];
```

篇幅所限，对 XDC 中 I/O 约束的设计思路、分析方法以及如何具体设置 Input 接口约束的讨论就到这里，下一篇我们接着分析如何设置 Output 接口约束，看看 Vivado 中有什么方式可以帮助用户准确便捷地定义接口时序约束。

—— Ally Zhou , 2015-2-28 于 Xilinx 上海 Office

[返回目录页](#)



XDC 约束技巧之 I/O 篇 (下)

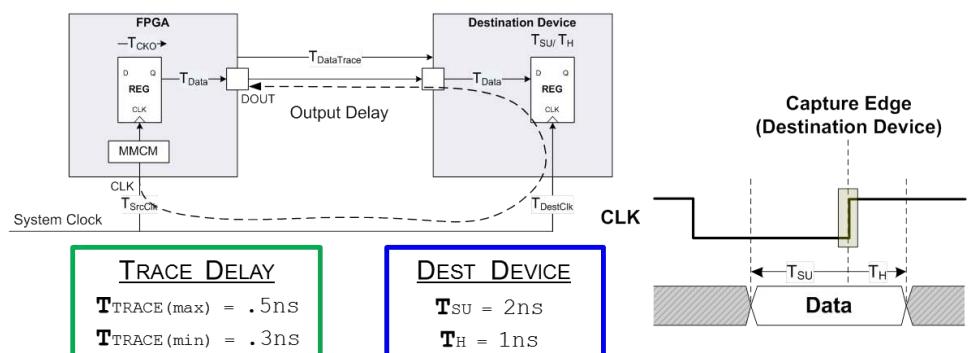
继《XDC 约束技巧之 I/O 篇 (上)》详细描述了如何设置 Input 接口约束后，我们接着来聊聊怎样设置 Output 接口约束，并分析 UCF 与 XDC 在接口约束上的区别。

Output 接口类型和约束

FPGA 做 Output 的接口时序同样也可以分为系统同步与源同步。在设置 XDC 约束时，总体思路与 Input 类似，只是换成要考虑下游器件的时序模型。另外，在源同步接口中，定义接口约束之前，需要用 `create_generated_clock` 先定义送出的随路时钟。

系统同步接口

与 Input 的系统同步接口一样，FPGA 做 Output 接口的系统同步设计，芯片间只传递数据信号，时钟信号的同步完全依靠板级设计来对齐。所以设置约束时候要考虑的仅仅是下游器件的 T_{SU}/T_H 和数据在板级的延时。



```
create_clock -name sysclk -period 10 [get_ports CLK];
set_output_delay -clock sysclk -max 2.5 [get_ports DOUT];
set_output_delay -clock sysclk -min -0.7 [get_ports DOUT];
```

上图是一个 SDR 上升沿采样系统同步接口的 Output 约束示例。其中，`-max` 后的数值是板级延时的最大值与下游器件的 T_{SU} 相加而得出，`-min` 后的数值则是板级延时的最小值减去下游器件的 T_H 而来。

VIVADO®

源同步接口

与源同步接口的 Input 约束设置类似，FPGA 做源同步接口的 Output 也有两种方法可以设置约束。

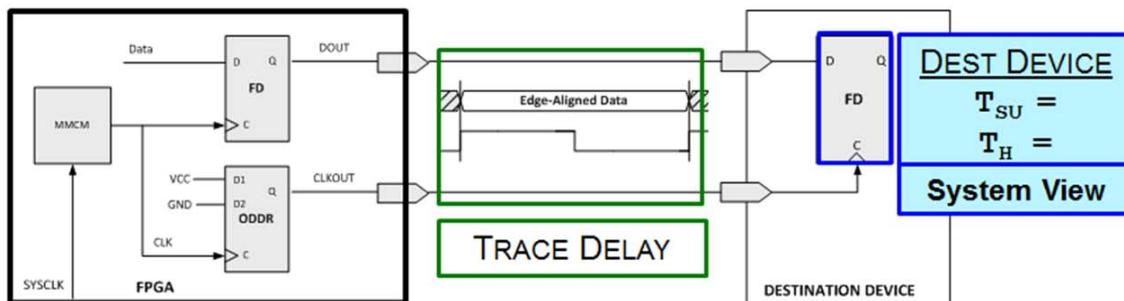
方法一我们称作 Setup/Hold Based Method，与上述系统同步接口的设置思路基本一致，仅需要了解下游器件用来锁存数据的触发器的 T_{SU} 与 T_H 值与系统板级的延时便可以设置。方法二称作 Skew Based Method，此时需要了解 FPGA 送出的数据相对于时钟沿的关系，根据 Skew 的大小和时钟频率来计算如何设置 Output 约束。

具体约束时可以根据不同的已知条件，选用不同的约束方式。一般而言，FPGA 作为输出接口时，数据相对时钟的 Skew 关系是已知条件（或者说，把同步数据相对于时钟沿的 Skew 限定在一定范围内是设计源同步接口的目标），所以方法二更常见。

Vivado® IDE 的 Language Templates 中关于源同步输出接口的 XDC 约束模板包含了以上两种方式的设置方法。

- **方法一 Setup/Hold Based Method**

Setup/Hold Based Method 的计算公式如下，可以看出其跟系统同步输出接口的设置方法完全一样。如果换成 DDR 方式，则可参考上一篇 I/O 约束方法中关于 Input 源同步 DDR 接口的约束，用两个可选项 `-clock_fall` 与 `-add_delay` 来添加针对时钟下降沿的约束值。

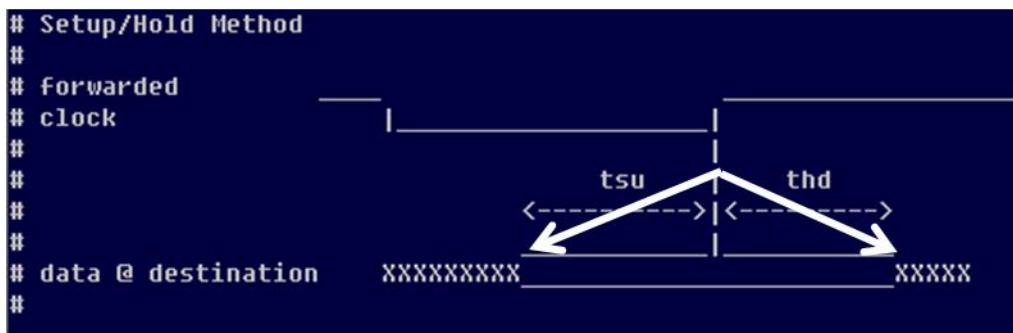


```
set_output_delay -max: TTRACE(max) + Dest Register TSU  
set_output_delay -min: TTRACE(min) - Dest Register TH
```

如果板级延时的最小值（在源同步接口中，因为时钟与信号同步传递，所以板级延时常常可以视作为 0）小于接收端寄存器的 T_H ，这样计算出的结果就会在 $-min$ 后出现负数值，很多时候会让人误以为设置错误。其实这里的负数并不表示负的延迟，而代表最小的延迟情况下，数据是在时钟采样沿之后才有效。同样的， $-max$ 后的正数，表示最大的延迟情况下，数据是在时钟采样沿之前就有效了。

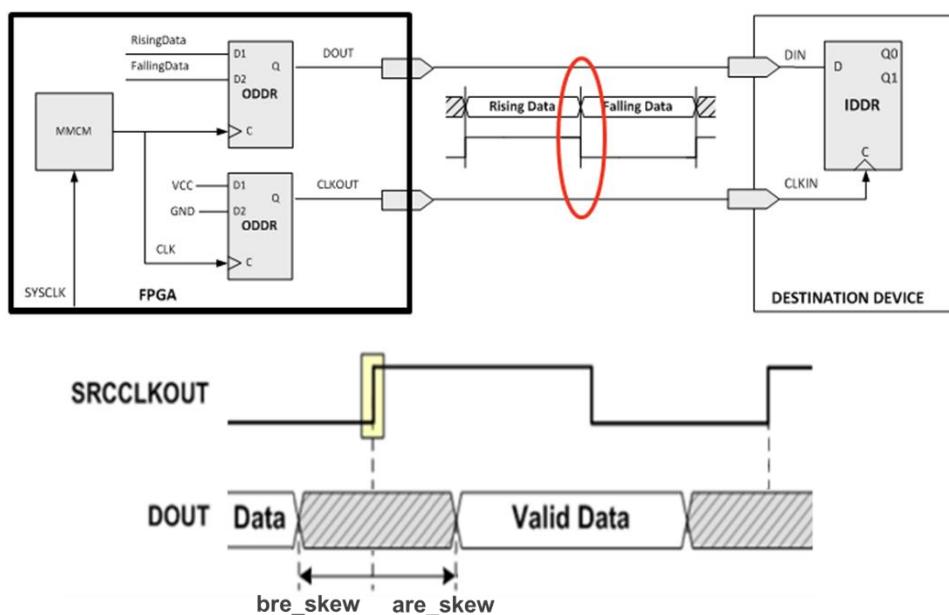
这便是接口约束中最容易混淆的地方，请一定牢记 `set_output_delay` 中 `-max/-min` 的定义，即时钟采样沿到达之前最大与最小的数据有效窗口。

如果我们在纸上画一下接收端的波形图，就会很容易理解：用于 setup 分析的 -max 之后跟着正数，表示数据在时钟采样沿之前就到达，而用于 hold 分析的 -min 之后跟着负数，表示数据在时钟采样沿之后还保持了一段时间。只有这样才能满足接收端用于锁存接口数据的触发器的 Tsu 和 Th 要求。



● 方法二 Skew Based Method

为了把同步数据相对于时钟沿的 Skew 限定在一定范围内，我们可以基于 Skew 的大小来设置源同步输出接口的约束。此时可以不考虑下游采样器件的 Tsu 与 Th 值。



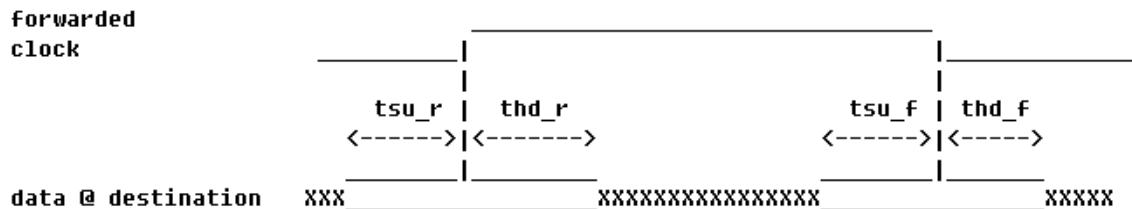
我们可以通过波形图来再次验证 set_output_delay 中 -max/-min 的定义，即时钟采样沿到达之前最大与最小的数据有效窗口。



DDR 接口的约束设置

DDR 接口的约束稍许复杂，需要将上升沿和下降沿分别考虑和约束，以下以源同步接口为例，分别就 Setup/Hold Based 方法和 Skew Based 方法举例。

● 方法一 Setup/Hold Based Method



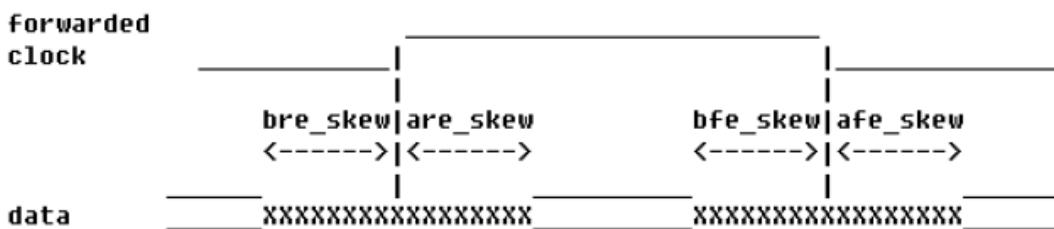
已知条件如下：

- ✓ 时钟信号 `src_sync_ddr_clk` 的频率： 100 MHz
- ✓ 随路送出的时钟 `src_sync_ddr_clk_out` 的频率： 100 MHz
- ✓ 数据总线： `src_sync_ddr_dout[3:0]`
- ✓ 接收端的上升沿建立时间要求 (`tsu_r`) : 0.7 ns
- ✓ 接收端的上升沿保持时间要求 (`thd_r`) : 0.3 ns
- ✓ 接收端的下降沿建立时间要求 (`tsu_f`) : 0.6 ns
- ✓ 接收端的下降沿保持时间要求 (`thd_f`) : 0.4 ns
- ✓ 板级走线延时 : 0 ns

可以这样计算输出接口约束：已知条件包含接收端上升沿和下降沿的建立与保持时间要求，所以可以分别独立计算。上升沿采样数据的 `-max` 是板级延时的最大值加上接收端的上升沿建立时间要求 (`tsu_r`)，对应的 `-min` 就应该是板级延时的最小值减去接收端的上升沿保持时间要求 (`thd_r`)；下降沿采样数据的 `-max` 是板级延时的最大值加上接收端的下降沿建立时间要求 (`tsu_f`)，对应的 `-min` 就应该是板级延时的最小值减去接收端的下降沿保持时间要求 (`thd_f`)。所以最终写入 XDC 的 Output 约束应该如下所示：

```
create_clock -period 10.0 -name clk [get_ports src_sync_ddr_clk];
create_generated_clock -name clk_out [get_ports ddr_src_sync_clk_out] \
    -source [get_ports src_sync_ddr_clk] -divide_by 1;
set_output_delay -clock clk_out -max 0.7 [get_ports src_sync_ddr_dout[*]];
set_output_delay -clock clk_out -min -0.3 [get_ports src_sync_ddr_dout[*]];
set_output_delay -clock clk_out -max 0.6 [get_ports src_sync_ddr_dout[*]]-clock_fall -add_delay;
set_output_delay -clock clk_out -min -0.4 [get_ports src_sync_ddr_dout[*]]-clock_fall -add_delay;
```

● 方法二 Skew Based Method



已知条件如下：

- ✓ 时钟信号 `src_sync_ddr_clk` 的频率： 100 MHz
- ✓ 随路送出的时钟 `src_sync_ddr_clk_out` 的频率： 100 MHz
- ✓ 数据总线：`src_sync_ddr_dout[3:0]`
- ✓ 上升沿之前的数据 skew (`bre_skew`) : 0.4 ns
- ✓ 上升沿之后的数据 skew (`are_skew`) : 0.6 ns
- ✓ 下降沿之前的数据 skew (`bfe_skew`) : 0.7 ns
- ✓ 下降沿之后的数据 skew (`afe_skew`) : 0.2 ns

可以这样计算输出接口约束：时钟的周期是 10ns，因为是 DDR 方式，所以数据实际的采样周期是时钟周期的一半；上升沿采样的数据的 `-max` 应该是采样周期减去这个数据的发送沿（下降沿）之后的数据 skew 即 `afe_skew`，而对应的 `-min` 就应该是上升沿之前的数据 skew 值 `bre_skew`；同理，下降沿采样数据的 `-max` 应该是采样周期减去这个数据的发送沿（上升沿）之后的数据 skew 值 `are_skew`，而对应的 `-min` 就应该是下降沿之前的数据 skew 值 `bfe_skew`。

所以最终写入 XDC 的 Output 约束应该如下所示：

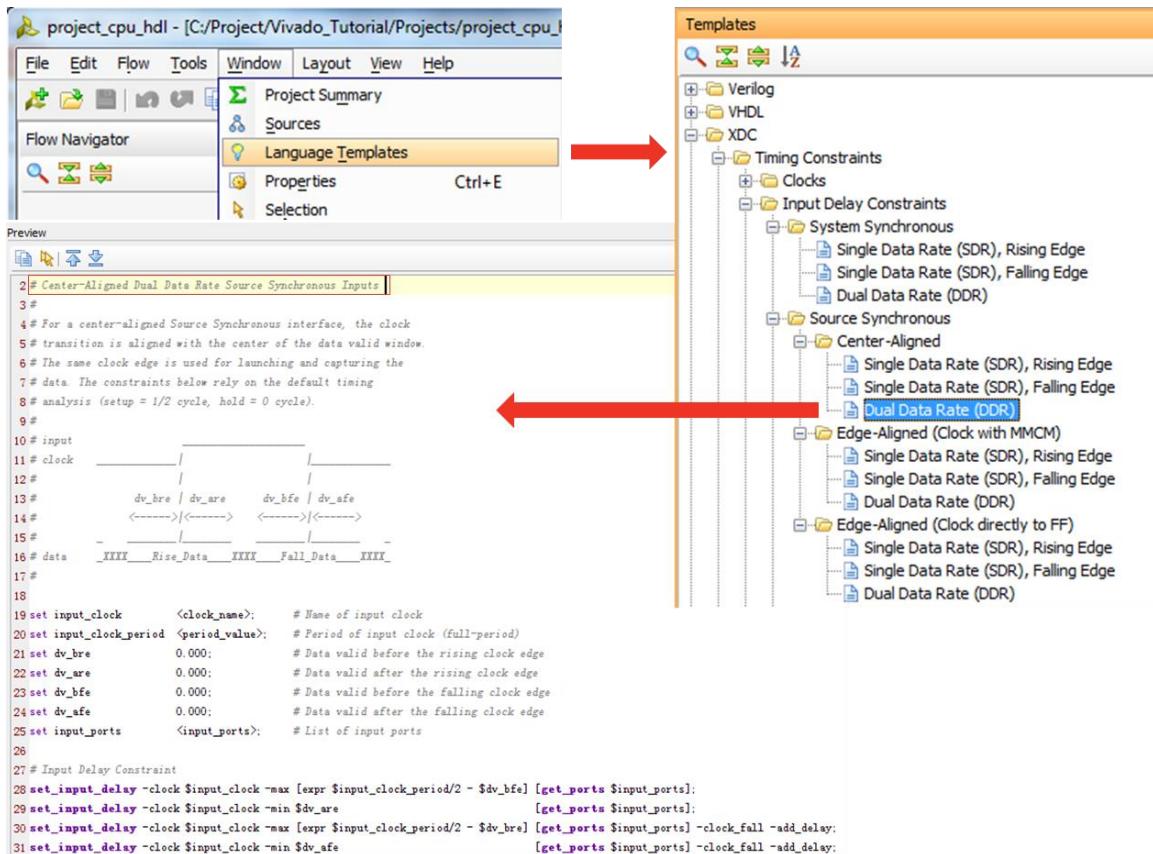
```
set period 10.0;
create_clock -period $period -name clk [get_ports src_sync_ddr_clk];
create_generated_clock -name clk_out [get_ports ddr_src_sync_clk_out] \
    -source [get_ports src_sync_ddr_clk] -divide_by 1;
set_output_delay -clock clk_out -max [expr $period/2 - 0.2] [get_ports src_sync_ddr_dout[*]];
set_output_delay -clock clk_out -min 0.4 [get_ports src_sync_ddr_dout[*]];
set_output_delay -clock clk_out -max [expr $period/2 - 0.6] [get_ports src_sync_ddr_dout[*]] \
    -clock_fall -add_delay;
set_output_delay -clock clk_out -min 0.7 [get_ports src_sync_ddr_dout[*]] -clock_fall -add_delay;
```

对以上两种方法稍作总结，就会发现在设置 DDR 源同步输出接口时，送出的数据是中心对齐的情况下，用 Setup/Hold Based 方法来写约束比较容易，而如果是边沿对齐的情况，则推荐使用 Skew Based 方法来写约束。

在 Vivado 中设置接口约束

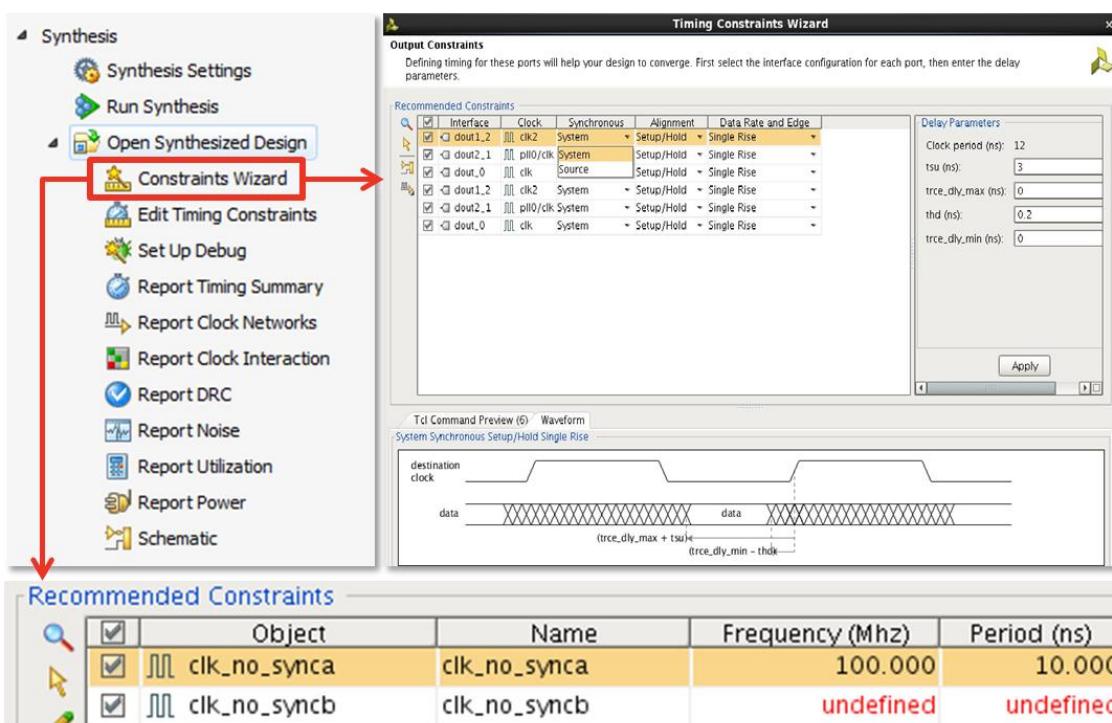
FPGA 的接口约束种类多变，远非一篇短文可以完全覆盖。在具体设计中，建议用户参照 Vivado IDE 的 Language Templates 。其中关于接口约束的例子有很多，而且也是按照本文所述的各种分类方法分别列出。

具体使用时，可以在列表中找到对应的接口类型，按照模板所示调整成自己设计中的数据，然后可以方便地计算出实际的约束值，并应用到 FPGA 工程中去。



自 2014.1 版开始，Vivado 还提供一个 Constraints Wizard 可供用户使用。只需打开综合后的设计，然后启动 Wizard，工具便可以根据读到的网表和设计中已有的 XDC 时序约束（也可以任何约束都不加而开始用 Wizard）一步步指引用户如何添加 Timing 约束，包括时钟、I/O 以及时序例外约束等等。

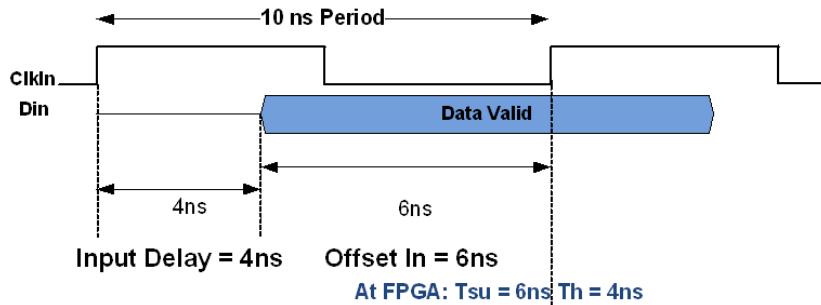
Constraints Wizard 的调出方法和界面如下图所示。



UCF 与 XDC 的区别

《XDC 约束技巧》开篇描述 XDC 基础语法时候曾经提到过设置接口约束时 UCF 与 XDC 的区别，简单来讲，UCF 是原生的 FPGA 约束，所以分析问题的视角是 FPGA 本身，而 XDC 则是从系统设计的全局角度来分析和设置接口约束。

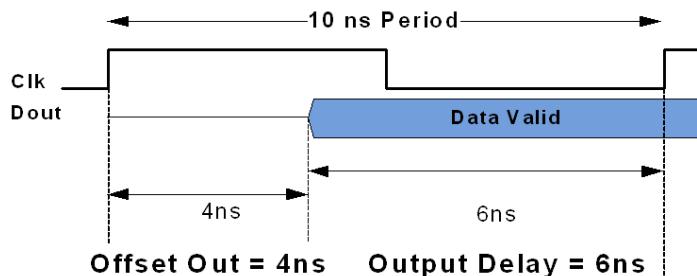
以最基础的 SDR 系统同步接口来举例。输入侧的设置，UCF 用的是 OFFSET = IN，而 XDC 则是 set_input_delay。



UCF: OFFSET = IN 6ns BEFORE ClkIn

XDC: set_input_delay 4 -clock ClkIn [get_ports Din]

输出侧的设置，UCF 用的是 OFFSET = OUT，而 XDC 则是 set_output_delay。



UCF: OFFSET = OUT 4ns AFTER Clk

XDC: set_output_delay 6 -clock Clk [get_ports Dout]

如果需要从旧设计的 UCF 约束转到 XDC 约束，可以参考上述例子。以一个采样周期来看，UCF 中与 XDC 中设置的接口约束值加起来正好等于一个周期的值。

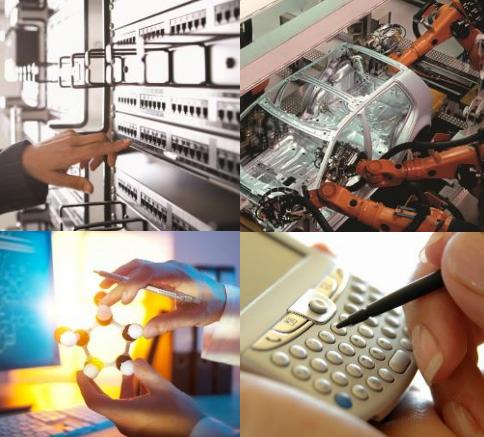
小结

这一系列《XDC 约束技巧》的文章至此暂时告一段落。其实读懂了这几篇涵盖了时钟、CDC 以及接口约束的短文，基本上已经足够应对绝大多数的 FPGA 设计约束问题。当然在这么短小的篇幅内，很多问题都无法更加深入地展开，所以也提醒读者，需要关注文中推荐的各类 Xilinx 官方文档，以及 Vivado 本身自带的帮助功能与模板。

希望各位能从本文中吸取经验，少走弯路，尽快地成为 Vivado 和 XDC 的资深用户，也希望本文能真正为您的设计添砖加瓦，达到事半功倍的效果。

—— Ally Zhou , 2015-3-13 于 Xilinx 上海 Office

返回目录页



Tcl 在 Vivado 中的应用

Xilinx®的新一代设计套件 Vivado®相比上一代产品 ISE，在运行速度、算法优化和功能整合等很多方面都有了显著地改进。但是对初学者来说，新的约束语言 XDC 以及脚本语言 Tcl 的引入则成为了快速掌握 Vivado 使用技巧的最大障碍，以至于两年多后的今天，仍有很多用户缺乏升级到 Vivado 的信心。

本文介绍了 Tcl 在 Vivado 中的基础应用，希望起到抛砖引玉的作用，指引使用者在短时间内快速掌握相关技巧，更好地发挥 Vivado 在 FPGA 设计中的优势。

Tcl 的背景介绍和基础语法

Tcl (读作 tickle) 诞生于 80 年代的加州大学伯克利分校，作为一种简单高效可移植性好的脚本语言，目前已经广泛应用在几乎所有的 EDA 工具中。Tcl 的最大特点就是其语法格式极其简单甚至可以说僵化，采用纯粹的 [命令 选项 参数] 形式，是名副其实的“工具命令语言”(即 Tcl 的全称 Tool Command Language)。

实际上 Tcl 的功能可以很强大，用其编写的程序也可以很复杂，但要在 Vivado 或大部分其它 EDA 工具中使用，则只需掌握其中最基本的几个部分。

注：在以下示例中，% 表示 Tcl 的命令提示符，执行回车后，Tcl 会在下一行输出命令执行结果。// 后是作者所加注释，并不是例子的一部分。

设置变量

```
% set myVar "Hello World!" // 设置一个名为 myVar 的变量，其值为 Hello World!
```

打印

```
% puts $myVar  
Hello World!
```

```
% puts "Hello World!"  
Hello World!
```

```
% puts myVar  
myVar
```

```
% puts {$myVar}  
$myVar
```

```
% puts "$myVar"  
$myVar
```

打印主要通过 puts 语句来执行，配合特殊符号，直接决定最终输出内容。

VIVADO

文件 I/O

写文件

```
%set wfp [open "my_file.txt" w]
file1073b243
%puts $wfp "Hello World! "
%puts $wfp $myVar
%close $wfp
```

读文件

```
%set rfp [open "my_file.txt" r]
file10
%set file_data [read $rfp]
Hello World!
Hello World!
%close $rfp
```

可以看到 Tcl 对文件的操作也是通过设置变量，改变属性以及打印命令来进行的。上述写文件的例子中通过 puts 命令在 my_file.txt 文件中写入两行文字，分别为“Hello World!” 和 myVar 变量的值，然后在读文件操作中读取同一文件的内容。

控制流和循环命令

Tcl 语言中用于控制流程和循环的命令与 C 语言及其它高级语言中相似，包括 if、while、for 和 foreach 等等。

具体使用可以参考如下示例，

```
% if {$myVar != 1} {puts "Sweet!"} //判断 myVar 变量的值，若不等于 1 就打印 Sweet!
Sweet!
% if {$myVar == 1} { puts "$myVar is = 1" } else { puts "$myVar is != 1" } //多条件判断
Hello World is != 1
% foreach x $myVar {puts $x} //逐一读取 myVar 变量的值并打印
Hello
World!
% set x 1 //设置变量 x
1
% while {$x < 5} { puts "x is $x"; set x [expr {$x + 1}] } //判断变量的值，打印，变量再赋值。
x is 1
```

子程序/过程

Tcl 中的子程序也叫做过程（Procedures），Tcl 正是通过创建新的过程来增强其内建命令的能力，提供更强的扩展性。具体到 Vivado 的使用中，用户经常可以通过对一个个子程序/过程的创建来扩展或个性化 Vivado 的使用流程。

```
% proc myProc {var} { //创建一个新的子程序（过程）myProc，用来打印输入变量的值。
    puts $var
}
% myProc "Yay!" //调用子程序（过程）
Yay!
```

一些特殊符号

符号	具体含义
\$	变量置换
[]	命令置换。即执行括号内的命令，且允许嵌套。
{}	1) 文字字符串，即不进行变量置换和命令置换，把其内的特殊符号当作符号本身。 2) 命令组合，即多个命令的组合。
" "	字符串，对其内的分隔符不作处理，但是允许进行变量置换和命令置换。
\	1) 反斜杠置换，允许其后紧跟着的特殊符号在字符串中以其本身符号的形式出现。 2) 出现在行尾，表示当前命令在下一行继续，从而允许一条命令出现在多行。
;	表示命令的结束。用来把多个命令隔开，从而允许多个命令出现在同一行。
#	注释符。必须出现在 Tcl 解析器期望命令的第一个字符出现的地方，其后直到所在行尾的所有字符都被看作注释。但出现在命令符后的#会被当作字符本身。

注： 反斜杠出现在行尾以允许命令在下一行继续时，必须是这一行的最后一个字符，其后不能有空格。

```
% set flop_name "flop1"      //字符串
flop1
% set myVar "flop_name"      //没有置换变量
flop_name
% set myVar "$flop_name"     //变量置换
flop1
% set a 1; # your comment   //允许在行尾注释
1
```

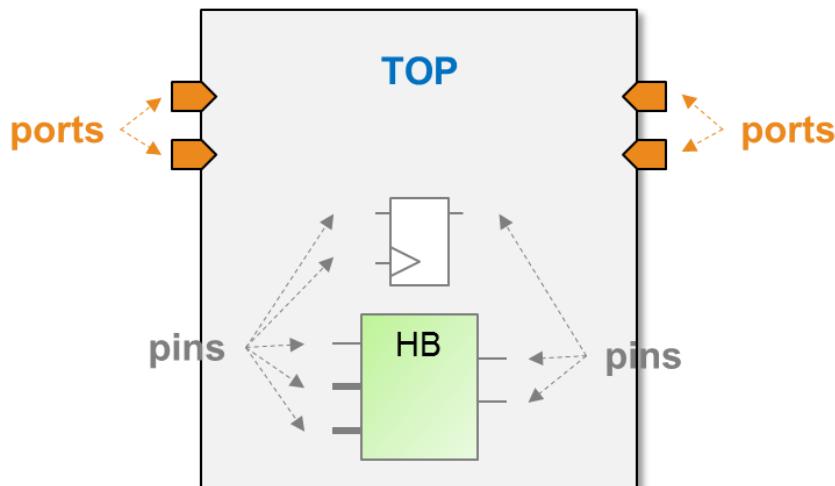
```
% set myVar ${flop_name}      //${代表其本身
$flop_name
% set myVar "\$flop_name"     //反斜杠置换
$flop_name
%set myVar \      //反斜杠允许命令出现在多行
"$flop_name"
flop1
```

Tcl 语言的基本语法相对简单，但要熟练掌握仍需日常不断练习。Xilinx 网站上有很多相关资料，这里推荐两个跟 Tcl 相关的文档 UG835 和 UG894，希望对大家学习 Vivado 和 Tcl 有所帮助。

在 Vivado 中使用 Tcl 定位目标

在 Vivado 中使用 Tcl 最基本的场景就是对网表上的目标进行遍历、查找和定位，这也是对网表上的目标进行约束的基础。要掌握这些则首先需要理解 Vivado 对目标的分类。

目标的定义和定位



如上图所示，设计顶层的 I/O 称作 ports，其余底层模块或是门级网表上的元件端口都称作 pins。而包括顶层在内的各级模块，blackbox 以及门级元件，都称作 cells。连线称作 nets，加上 XDC 中定义的 clocks，在 Vivado 中一共将网表文件中的目标定义为五类。要选取这五类目标，则需用相应的 get_*命令，例如 get_pins 等等。

● get_ports

ports 仅指顶层端口，所以 get_ports 的使用相对简单，可以配合通配符“*”以及 Tcl 语言中处理 list 的命令一起使用。如下所示，

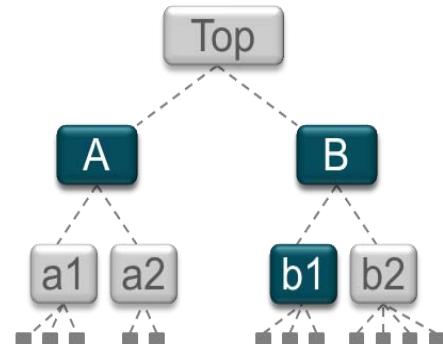
```
get_ports A           // 仅列出叫做 A 的顶层端口
get_ports *           // 列出所有顶层端口
get_ports *data*     // 列出所有名字中含有 “data”的顶层端口
llength [get_ports *] // 列出设计中所有顶层端口的数量
lindex [get_ports *data*] 0 // 列出名字中含有 “data”的顶层端口中的第一个
foreach x [get_ports *] {puts $x} // 逐一打印出所有顶层端口
```

● get_cells/get_nets

不同于 ports 仅指顶层端口，要定位 cells 和 nets 则相对复杂，首先需要面对层次的问题。这里有个大背景需要明确：Vivado 中 Tcl/XDC 对网表中目标的搜索是层次化的，也就是一次仅搜索一个指定的层次 current_instance，缺省值为顶层。

以右图所示设计来举例，若要搜索 A (不含 a1,a2) 层次内的所有 cells 和名字中含有 nt 的 nets，有两种方法：

```
current_instance A // 修改搜索层次到 A
get_cells *
get_nets *nt*
current_instance // 修改搜索层次到 Top
get_cells A/* // 搜索层次 A 内的所有 cells
```



若要将搜索层次改为 A+B+b1，则可以写一个循环，逐一用 current_instance 将搜索层次指向 A，B 和 b1，再将搜索到的 cells 或 nets 合成一个 list 输出即可。

若要将搜索层次改为当前层次以及其下所有子层次，可以使用 -hierarchical (在 Tcl 中可以简写为-hier)

```
get_cells -hierarchical * // 搜索 Top 层以及其下所有子层次内的所有 cells
get_nets -hier *nt* // 搜索 Top 层以及其下所有子层次内的名字中含有 nt 的 nets
```

```
current_instance A // 修改搜索层次到 A
get_cells -hier *
get_nets -hierarchical *nt* // 搜索 A 以及其下所有子层次内的所有 cells
// 搜索 A 以及其下所有子层次内的名字中含有 nt 的 nets
current_instance // 修改搜索层次到缺省值 Top
```

在使用-hierarchical 时有一点需要特别留意，即后面所跟的搜索条件仅指目标对象的名字，不能含有代表层次的“/”。下面列出的写法便是一种常见的使用误区，并不能以此搜索到 A 及其下子层次内所有的 cells。



```
get_cells -hier A/* // 搜索 Top 层以及其下所有子层次内名字以 “A/” 开头的 cells
```

- `get_pins`

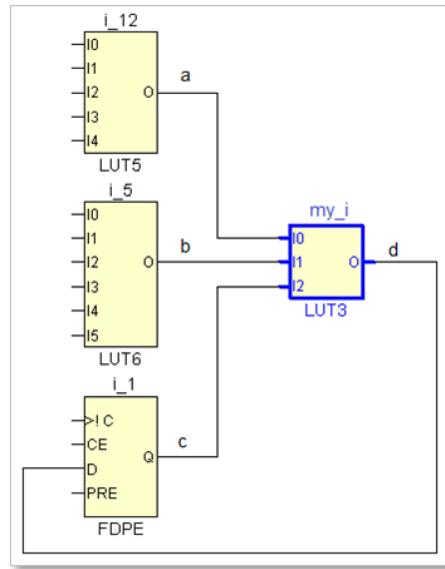
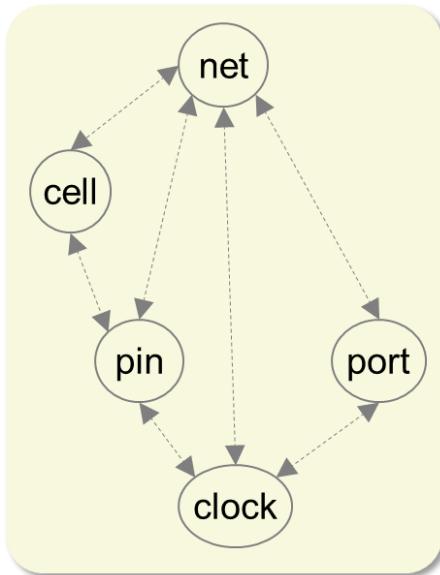
`pins` 在 Vivado 数据库中有个独特的存在形式，即 `<instance>/<pin>`。这里的“ / ”不表示层次，而是其名字的一部分，表示这个 pin 所属的实体。也就是说，在使用 `get_pins` 配合-hier 来查找 pins 时，“ / ”可以作为名字的一部分，出现在搜索条件内（注意与上述 `get_cells` 和 `get_nets` 的使用区别）

```
get_pins      i_b1/p1      //返回空的 list
get_pins      x/i_b1/p1    //找到 i_x/i_b1/p1
get_pins      */i_b1/p1    //找到 i_x/i_b1/p1
get_pins      *p1          //返回空的 list
get_pins -hier  *p1        //找到 i_x/i_b1/p1
get_pins -hier  */p1       //找到 i_x/i_b1/p1
```

<code>current_instance</code>	<code>i_x</code>
<code>get_pins i_b1/p1</code>	//找到 i_x/i_b1/p1
<code>get_pins */p1</code>	//找到 i_x/i_b1/p1
<code>get_pins */*</code>	//找到 i_x/i_b1/p1 i_x/i_b1/p2 i_x/i_b1/x1 i_x/i_b1/x2
<code>get_pins *</code>	//同上，找到四个 pins

目标之间的关系

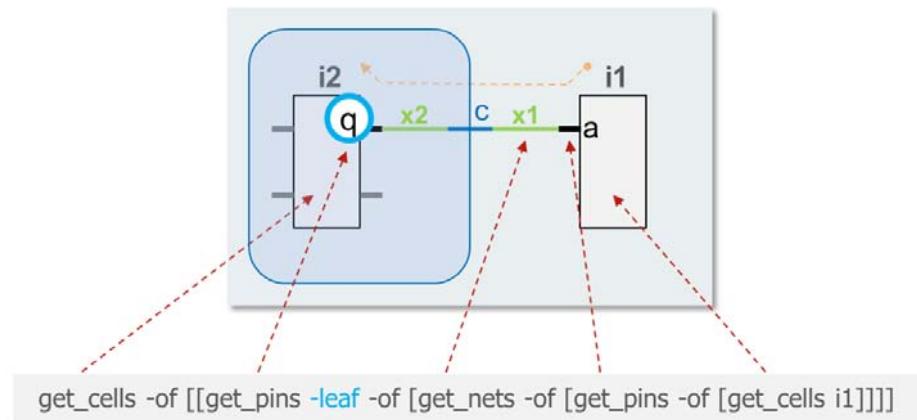
Tcl 在搜索网表中的目标时，除了上述根据名字条件直接搜索的方式，还可以利用目标间的关系，使用 `-of_objects` (在 Tcl 中可以简写为 `-of`) 来间接搜索特定目标。Vivado 中定义的五类目标间的关系如下页左图所示。



以上示例右图的设计来举例，

```
get_pins -of [get_cells my_i]           //返回 my_i/I0  my_i/I1  my_i/I2  my_i/O
get_nets -of [get_cells my_i]           //返回 a  b  c  d
get_cells -of [get_nets -of [get_cells my_i]] //返回 i_12  i_5  i_1 my_i
```

下图是一个更复杂的示例，涉及跨层次搜索。可以看到在 `get_pins` 时，要加上 `-leaf` 才能准确定位到门级元件（或 blackbox ）的端口 `q`。另外，在实际操作中，使用 `get_nets` 和 `get_pins` 时，需要视情况而加上其它条件（`-filter` ）才能准确找到下述例子中的 cells （ `i2` ）。



高级查找功能

在使用 `get_*` 命令查找网表中的目标时，除了名字这一直接条件，往往还需要辅以其它更复杂的条件判断，这就需要用到高级查找功能：`-filter` 结合 Tcl 支持的各种关系和逻辑运算符（`==, !=, =~, !~, <=, >=, >, <, &&, ||`）甚至是正则表达式来操作。

```
get_cells -filter {REF_NAME == FDCE} *          //找出所有的 FDCE  
get_cells -filter {REF_NAME == FDCE && INIT == 1'b0} *    //找出所有初值为 0 的 FDCE  
get_cells -filter {IS_PRIMITIVE == 1} *        //找出所有 primitive  
get_ports -filter {DIRECTION == in && LOC =~ H*} *      //找出锁定在 H 区的输入管脚  
get_ports -filter {DIRECTION == in && SLEW == FAST} *data* //找出名字中含有 data 且 slew 定义为 fast 的输入管脚
```

在创建子程序时也常常需要用到-filter，例如下述 get_p 的子程序/过程就可以用来返回指定管脚的方向属性，告诉用户这是一个输入管脚还是一个输出管脚。需要特别指出的是，通常在-filter 后会使用{}，但此时需要对 \$direction 做变量替换，必须如下所示改用" "。

```
proc get_p { direction }{ // 定义一个过程，读取 ports 的 direction 属性
    return [get_ports -filter "DIRECTION == $direction " ]
}
get_p  in // 调用上述过程，返回管脚 “in” 的 direction 属性
```

Tcl 在 Vivado 中的延伸应用

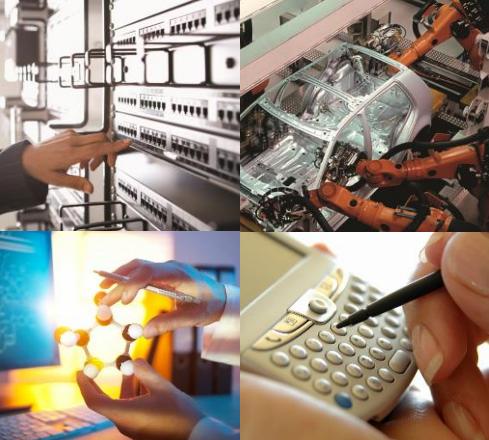
Tcl 在 Vivado 中的应用还远不止上述所列，其它常用的功能包括使用预先写好的 Tcl 脚本来跑设计实现流程，创建高级约束（ XDC 不支持循环等高级 Tcl 语法）以及实现复杂的个性化设计流程等等。Tcl 所带来的强大的可扩展性决定了其在版本控制、设计自动化流程等方面具有图形化界面不能比拟的优势。

Vivado 在不断发展更新的过程中，还有很多新的功能，包括 ECO、PR、HD Flow 等等都是从 Tcl 脚本方式开始支持，然后再逐步放入图形化界面中实现。这也解释了为何高端 FPGA 用户和熟练的 Vivado 用户都更偏爱 Tcl 脚本。

篇幅所限，不能一一展开。关于以上 Tcl 在 Vivado 中的延伸应用，敬请关注 Xilinx 官方网站和中文论坛上的更多技术文章。

— Ally Zhou , 2014-9-12 于 Xilinx 上海 Office

[返回目录页](#)



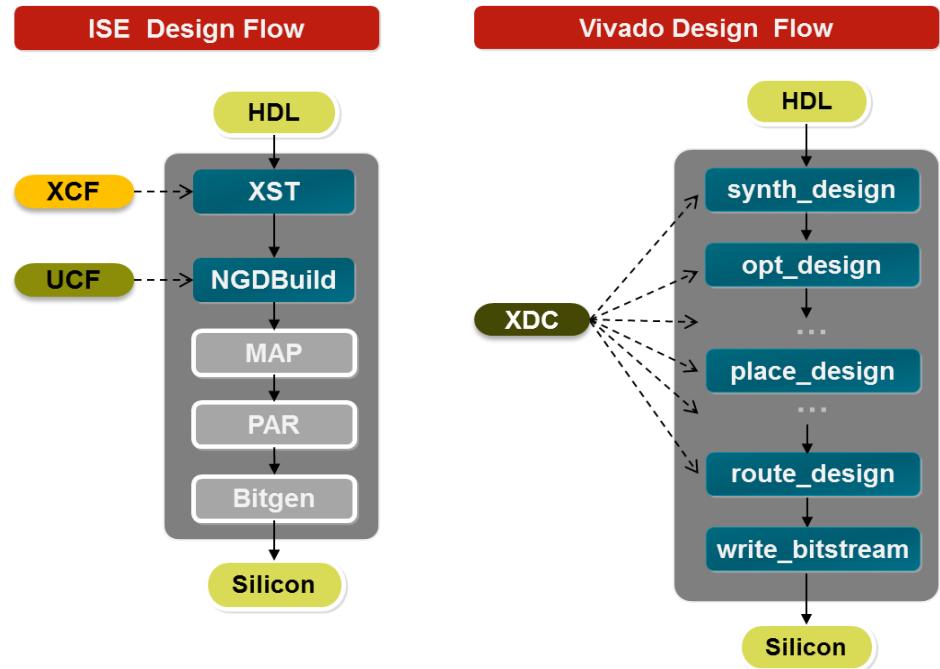
用 Tcl 定制 Vivado 设计实现流程

上一篇 [《Tcl 在 Vivado 中的应用》](#) 介绍了 Tcl 的基本语法以及如何利用 Tcl 在 Vivado 中定位目标。其实 Tcl 在 Vivado 中还有很多延展应用，接下来我们就来讨论如何利用 Tcl 语言的灵活性和可扩展性，在 Vivado 中实现定制化的 FPGA 设计流程。

基本的 FPGA 设计实现流程

FPGA 的设计流程简单来讲，就是从源代码到比特流文件的实现过程。大体上跟 IC 设计流程类似，可以分为前端设计和后端设计。其中前端设计是把源代码综合为对应的门级网表的过程，而后端设计则是把门级网表布局布线到芯片上最终实现的过程。

以下两图分别表示 ISE 和 Vivado 的基本设计流程：



ISE 中设计实现的每一步都是相对独立的过程，数据模型各不相同，用户需要维护不同的输入文件，例如约束等，输出文件也不是标准网表格式，并且形式各异，导致整体运行时间过长，冗余文件较多。

Vivado 中则统一了约束格式和数据模型，在设计实现的任何一个阶段都支持 XDC 约束，可以生成时序报告，在每一步都能输出包含有网表、约束以及布局布线信息（如果有）的设计检查点（DCP）文件，大大缩短了运行时间。

主要文档

- 即插即用 IP 背景资料
- [【中文】UG949 - UltraFast 设计方法指南](#)
- [UG1046 - UltraFast 嵌入式设计方法指南](#)
- [Vivado Design Suite 加速设计生产力的九大理由](#)
- [Vivado IP Integrator 背景资料](#)

快速链接

- 免费下载：[Vivado Design Suite 评估和 WebPACK 版本](#)
- 下载
- 支持和技术文档
- [Vivado 视频辅导资料](#)
- [IP 中心](#)
- [支持的目标参考设计](#)
- [存储器推荐](#)

培训与活动

- [Vivado 课程](#)
- [高层次综合课程](#)

VIVADO 

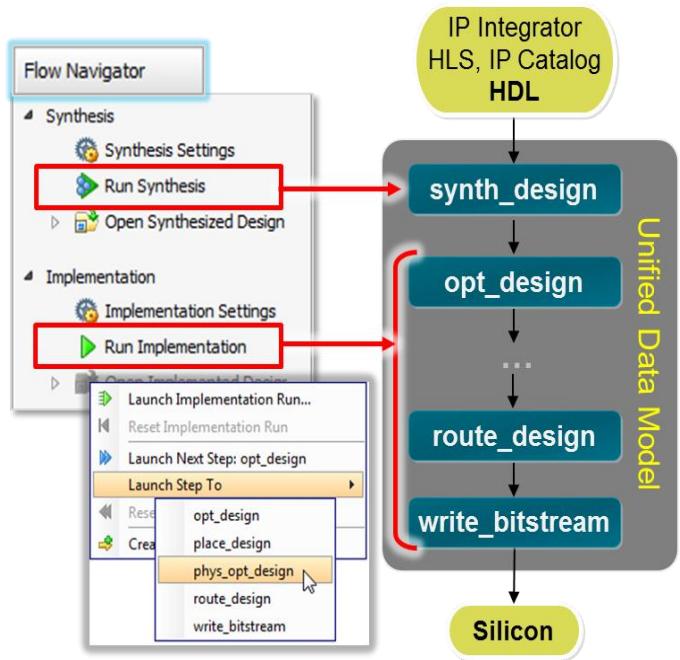
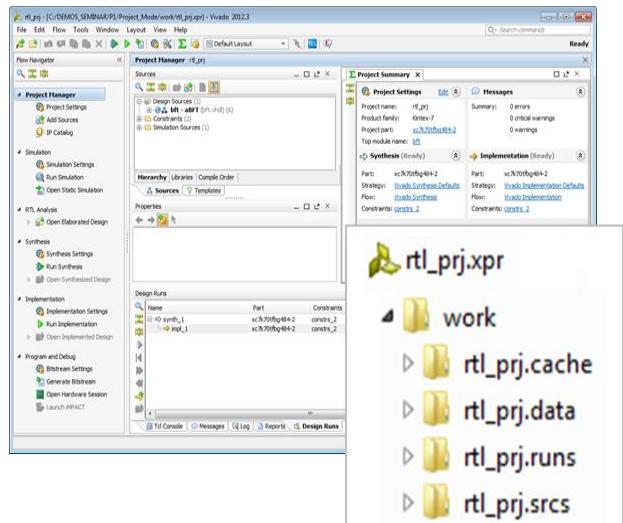
从使用方式上来讲，Vivado 支持工程模式（Project Based Mode）和非工程模式（None Project Mode）两种，且都能通过 Tcl 脚本批处理运行，或是在 Vivado 图形化界面 IDE 中交互运行和调试。

工程模式

工程模式的关键优势在于可以通过在 Vivado 中创建工程的方式管理整个设计流程，包括工程文件的位置、阶段性关键报告的生成、重要数据的输出和存储等。

如下左图所示，用户建立了一个 Vivado 工程后，工具会自动创建相应的.xpr 工程文件，并在工程文件所在的位置同层创建相应的几个目录，包括<prj_name>.cache、<prj_name>.data、<prj_name>.runs 和 <prj_name>.srcs 等等（不同版本可能有稍许差异），分别用于存储运行工程过程中产生的数据、输出的文件和报告以及工程的输入源文件（包含约束文件）等。

如下右图所示，在 Vivado IDE 中还可以一键式运行整个设计流程。这些预置的命令按钮就放置在工具最左边的侧栏：Flow Navigator。不同按钮对应不同的实现过程，其中在后端实现阶段，还可以用右键调出详细分步命令，指引工具具体执行实现的哪一步。

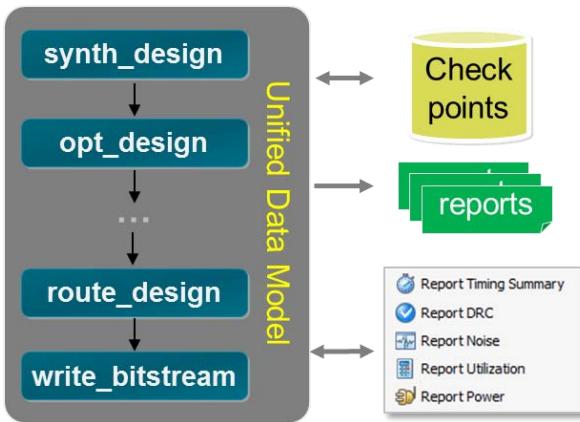


特别需要指出的是 Flow Navigator 只有在 Vivado IDE 中打开.xpr 工程文件才会显示，如果打开的是设计检查点.dcp 文件（不论是工程模式或是非工程模式产生的 dcp）都不会显示这个侧栏。

非工程模式

非工程模式下，由于不会创建工程，用户就需要自己管理设计源文件和设计过程。源文件只能从当前位置访问，在设计实现过程中的每一步，数据和运行结果都存在于 Vivado 分配到的机器内存中，在用户不主动输出的情况下，不会存储到硬盘中。

简单来讲，非工程模式提供了一种类似 ASIC 设计的流程，用户拥有绝对的自由，可以完全掌控设计实现流程，但也需要用户对设计实现的过程和数据，尤其对文件输出和管理全权负责，包括何时、何地、输出怎样的文件等等。



使用非工程模式管理输入输出文件、进行设计实现都需要使用 Tcl 脚本，但这并不代表非工程模式不支持图形化界面。非工程模式下产生的.dcp 文件一样可以在 Vivado IDE 中打开，继而产生各种报告，进行交互式调试等各种在图形化下更便捷直观的操作。这是一个常见误区，就像很多人误认为工程模式下不支持 Tcl 脚本运行是一个道理。但两种模式支持的 Tcl 命令确实是完全不同的，使用起来也不能混淆。

右图所示是同一个设计 (Vivado 自带的 Example Design) 采用两种模式实现所使用的不同脚本，更详细的内容可以在 UG975 和 UG835 中找到。需要注意的是，工程模式下的 Tcl 脚本更简洁，但并不是最底层的 Tcl 命令，实际执行一条相当于执行非工程模式下的数条 Tcl 命令。

Vivado 支持的两种 Tcl 脚本 – 右图

Batch Mode Script Examples

Example scripts for both Project Mode and Non-Project Mode, using the BFT example design.

Non-Project Mode:

```

set outputDir ./Tutorial_Created_Data/bft_output
file mkdir $outputDir
# STEP#1: setup design sources and constraints
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhd
read_verilog [ glob ./Sources/hdl/*.v ]
read_xdc ./Sources/bft_full.xdc
# STEP#2: run synthesis, report utilization and timing estimates, write checkpoint design
synth_design -top bft -part xc7k70tfbg484-2
write_checkpoint -force $outputDir/post_synth
report_utilization -file $outputDir/post_synth_util.rpt
report_timing -sort_by group -max_paths 5 -path_type summary \
    -file $outputDir/post_synth_timing.rpt
# STEP#3: run placement and logic optimization, report utilization and timing estimates
opt_design
power_opt_design
place_design
phys_opt_design
write_checkpoint -force $outputDir/post_place
report_clock_utilization -file $outputDir/clock_util.rpt
report_utilization -file $outputDir/post_place_util.rpt
report_timing -sort_by group -max_paths 5 -path_type summary \
    -file $outputDir/post_place_timing.rpt
# STEP#4: run router, report actual utilization and timing, write checkpoint design, run DRCs
route_design
write_checkpoint -force $outputDir/post_route
report_timing_summary -file $outputDir/post_route_timing_summary.rpt
report_utilization -file $outputDir/post_route_util.rpt
report_power -file $outputDir/post_route_power.rpt
report_drc -file $outputDir/post_imp_drc.rpt
write_verilog -force $outputDir/bft_impl_netlist.v
write_xdc -no_fixed_only -force $outputDir/bft_impl.xdc
# STEP#5: generate a bitstream
write_bitstream $outputDir/design.bit

```

Project Mode:

```

create_project project_bft ./project_bft -part xc7k70tfbg484-2
add_files {./Sources/hdl/FifoBuffer.v ./Sources/hdl/async_fifo.v ./Sources/hdl/bft.vhd}
add_files [ glob ./Sources/hdl/bftLib/*.vhdl]
set_property library bftLib [get_files [ glob ./Sources/hdl/bftLib/*.vhdl]]
import_files -force -norecurse
import_files -fileset constrs_1 ./Sources/bft_full.xdc
set_property steps.synth_design.args.flatten_hierarchy full [get_runs synth_1]
launch_runs synth_1
wait_on_run synth_1
launch_runs impl_1
wait_on_run impl_1
launch_runs impl_1 -to_step write_bitstream

```

Tcl 对图形化的补充

相信对大部分 FPGA 工程设计人员来说，图形化界面仍旧是最熟悉的操作环境，也是设计实现的首选。在 Xilinx 推出全面支持 Tcl 的 Vivado 后，这一点依然没有改变，但我们要指出的是，即使是在图形化界面上跑设计，仍然可以充分利用 Tcl 的优势。在 Vivado IDE 上运行 Tcl 脚本主要有以下几个渠道。

Tcl Console



Vivado IDE 的最下方有一个 Tcl Console，在运行过程中允许用户输入 Tcl/XDC 命令或是 source 预先写好的 Tcl 脚本，返回值会即时显示在这个对话框。

举例来说，设计调试过程中，需要将一些约束应用在某些网表目标上（具体可参照[《Tcl 在 Vivado 中的应用》](#)所示），推荐的做法就是在 IDE 中打开.dcp 然后在 Tcl Console 中输入相应的 Tcl/XDC 命令，验证返回值，碰到问题可以直接修改，直到找到正确合适的命令。然后可以记录这些命令，并存入 XDC 文件中以备下次实现时使用。

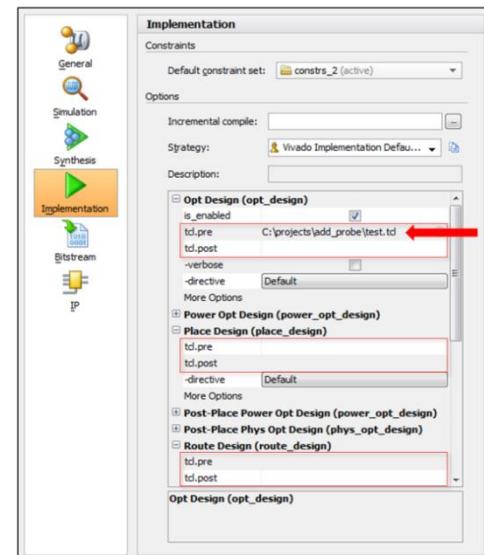
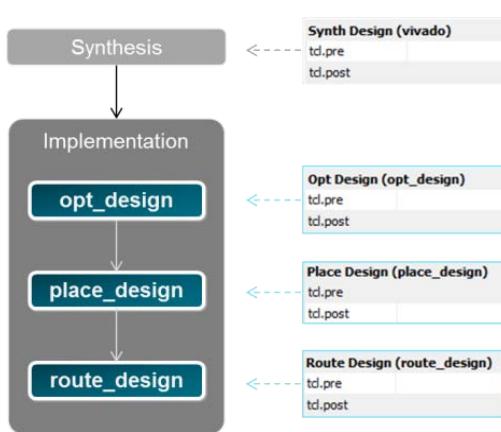
还有一种情况是，预先读入的 XDC 中有些约束需要修改，或是缺失了某些重要约束。不同于 ISE 中必须修改 UCF 重跑设计的做法，在 Vivado 中，我们可以充分利用 Tcl/XDC 的优势，在 Tcl Console 中输入新的 Tcl/XDC，无需重跑设计，只要运行时序报告来验证。当然，如果能重跑设计，效果会更好，但是这种方法在早期设计阶段提供了一种快速进行交互式验证的可能，保证了更快地设计迭代，大大提升了效率。

另外，通过某些 Tcl 命令（例如 show_objects、select_objects 等等）的帮助，我们还可以利用 Tcl Console 与时序报告、RTL 和门级网表以及布局布线后的网表之间进行交互调试，极大发挥 Vivado IDE 的优势。

Hook Scripts

Vivado IDE 中内置了 tcl.pre 和 tcl.post，用户可以在 Synthesis 和 Implementation 的设置窗口中找到。设计实现的每一步都有这样两个位置可供用户加入自己的 Tcl 脚本。

tcl.pre 表示当前这步之前 Vivado 会主动 source 的 Tcl 脚本，tcl.post 表示这步之后会 source 的脚本。

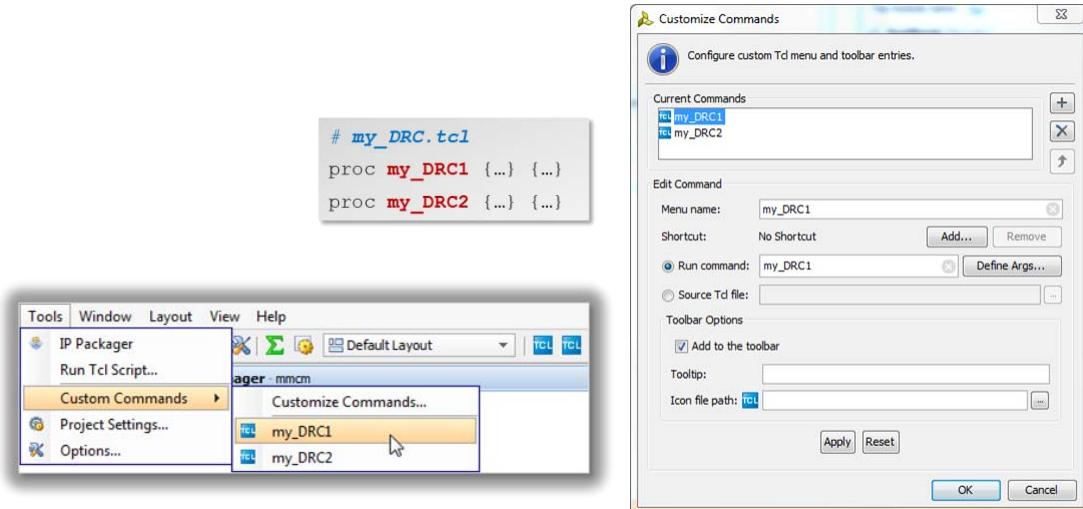


Tcl 脚本必须事先写好，然后在上图所示的设置界面由用户使用弹出窗口指定到脚本所在位置。

这些就是所谓的“钩子”脚本，正是有了这样的脚本，我们才得以在图形化界面上既享有一键式执行的便利，又充分利用 Tcl 带来的扩展性。比较常见的使用场景是，在某个步骤后多产生几个特别的报告，或是在布线前再跑几次物理优化等。

Customer Commands

Vivado IDE 中还有一个扩展功能，允许用户把事先创建好的 Tcl 脚本以定制化命令的方式加入图形化界面，成为一个按钮，方便快速执行。这个功能常常用来报告特定的时序信息、修改网表内容、实现 ECO 等等。



用 Tcl 定制实现流程

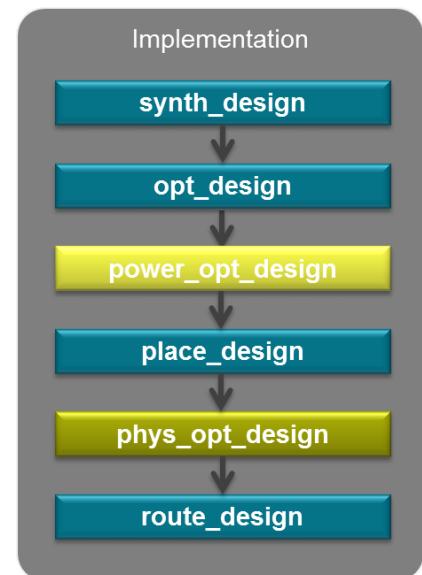
综上所述，标准的 FPGA 设计实现流程完全可以通过 Vivado IDE 一键式执行，如果仅需要少量扩展，通过前述钩子脚本等几种方法也完全可以做到。若是这些方法都不能满足需求，还可以使用 Tcl 脚本来跑设计，从而实现设计流程的全定制。

注：以下讨论的几种实现方案中仅包含后端实现具体步骤的区别，而且只列出非工程模式下对应的 Tcl 命令。

右图所示是 Vivado 中设计实现的基本流程，蓝色部分表示实现的基本步骤（尽管 opt_design 这一步理论上不是必选项，但仍强烈建议用户执行），对应 Implementation 的 Default 策略。黄色部分表示可选择执行的部分，不同的实现策略中配置不同。

这里不会讨论那些图形化界面中可选的策略，不同策略有何侧重，具体如何配置我们将在另外一篇关于 Vivado 策略选择的文章中详细描述。

我们要展示的是如何对设计流程进行改动来更好的满足设计需求，这些动作往往只能通过 Tcl 脚本来实现。



充分利用物理优化

物理优化即 phys_opt_design 是在后端通过复制、移动寄存器来降扇出和 retiming，从而进行时序优化的重要手段，一般在布局和布线之间运行，从 Vivado 2014.1 开始，还支持布局后的物理优化。

很多用户会在 Vivado 中选中 phys_opt_design，但往往不知道这一步其实可以运行多次，并且可以选择不同的 directive 来有侧重的优化时序。

比如，我们可以写这样一个 Tcl 脚本，在布局后，使用不同的 directive 或选项来跑多次物理优化，甚至可以再多运行一次物理优化，专门针对那些事先通过 get_nets 命令找到并定义为 highfanout_nets 的高扇出网络

具体 directive 的含义可以通过 UG835、UG904 或 phys_opt_design -help 命令查询。

布局布线之间的多次物理优化不会恶化时序，但会增加额外的运行时间，也有可能出现时序完全没有得到优化的结果。布线后的物理优化有时候会恶化 THS，所以请一定记得每一步后都运行 report_timing_summary，并且通过 write_checkpoint 写出一个.dcp 文件来保留阶段性结果。这一步的结果不理想就可以及时退回到上一步的.dcp 继续进行。

```

synth_design
opt_design
place_design
phys_opt_design -directive AggressiveExplore
phys_opt_design -directive AggressiveFanoutOpt
phys_opt_design -force_replication_on_nets $highfanout_nets
phys_opt_design -retime
route_design
phys_opt_design

```

闭环设计流程

通常的 FPGA 设计流程是一个开环系统，从前到后依次执行。但 Vivado 中提供了一种可能，用户可以通过 place_design -post_place_opt 在已经完成布局布线的设计上再做一次布局布线，从而形成一个有了反馈信息的闭环系统。这次因为有了前一次布线后的真实连线延迟信息，布局的针对性更好，并且只会基于时序不满足的路径进行重布局而不会改变大部分已经存在的布局信息，之后的布线过程也是增量流程。

这一过程所需的运行时间较短，是一种很有针对性的时序优化方案。可以通过 Tcl 写一个循环多次迭代运行，但需留意每次的时序报告，若出现时序恶化就应及时停止。

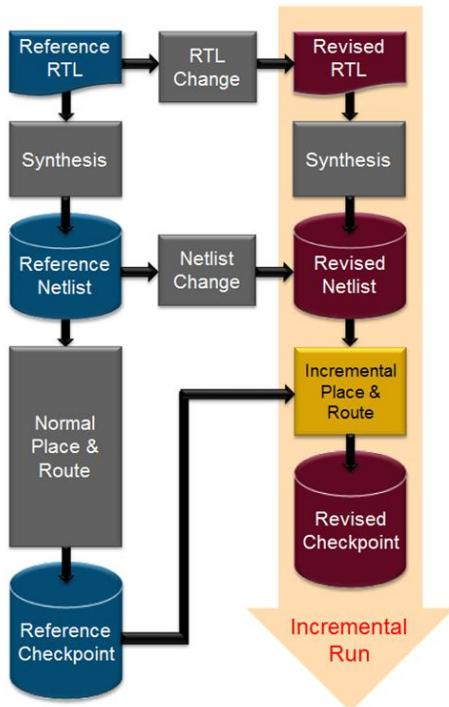
```

synth_design
opt_design
place_design
phys_opt_design
route_design
for {set i 0} {$i<=3} {incr i} {
    place_design -post_place_opt
    route_design
    report_timing_summary -file $i.rpt
    write_checkpoint -force post_place_opted.dcp

```

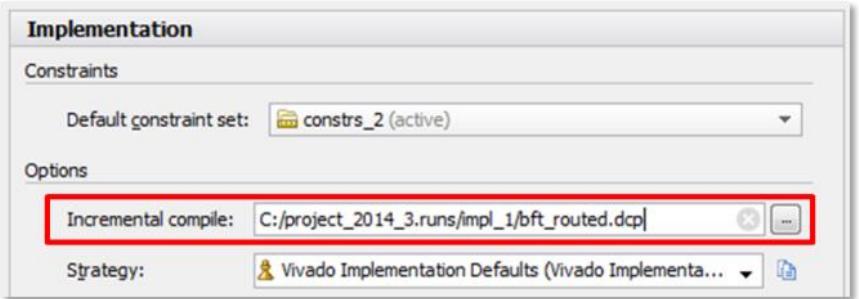
增量设计流程

Vivado 中的增量设计也是一个不得不提的功能。当设计进行到后期，每次运行改动很小，在开始后端实现前读入的设计网表具有较高相似度的情况下，推荐使用 Vivado 的增量布局布线功能。



如左图所示，运行增量流程的前提是有一个已经完成布局布线的.dcp 文件，并以此用来作为新的布局布线的参考。

运行过程中，Vivado 会重新利用已有的布局布线数据来缩短运行时间，并生成可预测的结果。当设计有 95% 以上的相似度时，增量布局布线的运行时间会比一般布局布线平均缩短 2 倍。若相似度低于 80%，则使用增量布局布线只有很小的优势或者基本没有优势。



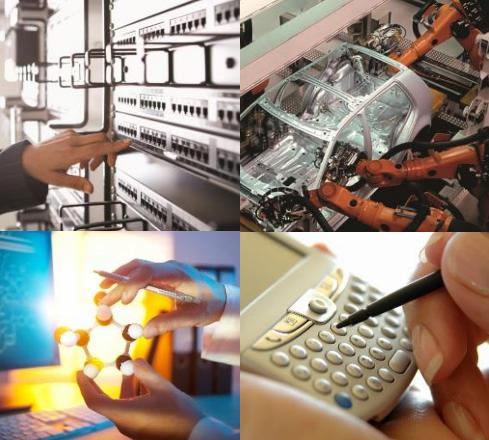
除了缩短运行时间外，增量布局布线对没有发生变化的设计部分造成的破坏也很小，因此能减少时序变化，最大限度保留时序结果，所以一般要求用做参考的.dcp 文件必须是一个完全时序收敛的设计。

参考点.dcp 文件可以在 Vivado IDE 的 Implementation 设置中指定，也可以在 Tcl 脚本中用 `read_checkpoint -incremental` 读入。特别需要指出的是，在工程模式中，如要在不新建一个 impl 实现的情况下使用上一次运行的结果作为参考点，必须将其另存到这次运行目录之外的位置，否则会因冲突而报错。

以上用 Tcl 定制 Vivado 设计实现流程的讨论就到这里，后面我将就 Tcl 使用场景，包括 ECO 流程等的更多细节进行展开。

—— Ally Zhou , 2014-11-11 于 Xilinx 上海 Office

[返回目录页](#)



在 Vivado 中实现 ECO 功能

关于 Tcl 在 Vivado®中的应用文章从 Tcl 的基本语法和在 Vivado 中的应用展开，继上篇《用 Tcl 定制 Vivado 设计实现流程》介绍了如何扩展甚至是定制 FPGA 设计实现流程后，引出了一个更细节的应用场景：如何利用 Tcl 在已完成布局布线的设计上对网表或是布局布线进行局部编辑，从而在最短时间内，以最小的代价完成个别的设计改动需求。

什么是 ECO

ECO 指的是 Engineering Change Order，即工程变更指令。目的是为了在设计的后期，快速灵活地做小范围修改，从而尽可能的保持已经验证的功能和时序。ECO 的叫法算是从 IC 设计领域继承而来，其应用在 FPGA 设计上尚属首次，但这种做法其实在以往的 FPGA 设计上已被广泛采用。简单来说，ECO 便相当于 ISE 上的 FPGA Editor。

但与 FPGA Editor 不同，Vivado 中的 ECO 并不是一个独立的界面或是一些特定的命令，要实现不同的 ECO 功能需要使用不同的方式。

ECO 的应用场景和实现流程

ECO 的应用场景主要包含：修改 cell 属性、增减或移动 cell、手动局部布线。还有一些需要多种操作配合的复杂场景，例如把 RAM（或 DSP）的输出寄存器放入/拉出 RAMB（或 DSP48）内部，或是把设计内部信号接到 I/O 上作调试 probe 用等等。

针对不同的应用场景，Vivado 中支持的 ECO 实现方式也略有区别。有些可以用图形界面实现，有些则只能使用 Tcl 命令。但通常可以在图形化界面上实现的操作，都可以改用一条或数条 Tcl 命令来实现。

ECO 的实现流程如下图所示：



主要文档

- 即插即用 IP 背景资料
- 【中文】UG949 - UltraFast 设计方法指南
- UG1046 - UltraFast 嵌入式设计方法指南
- Vivado Design Suite 加速设计生产力的九大理由
- Vivado IP Integrator 背景资料

快速链接

- 免费下载 : Vivado Design Suite 评估和 WebPACK 版本
- 下载
- 支持和技术文档
- Vivado 视频辅导资料
- IP 中心
- 支持的目标参考设计
- 存储器推荐

培训与活动

- Vivado 课程
- 高层次综合课程

VIVADO®

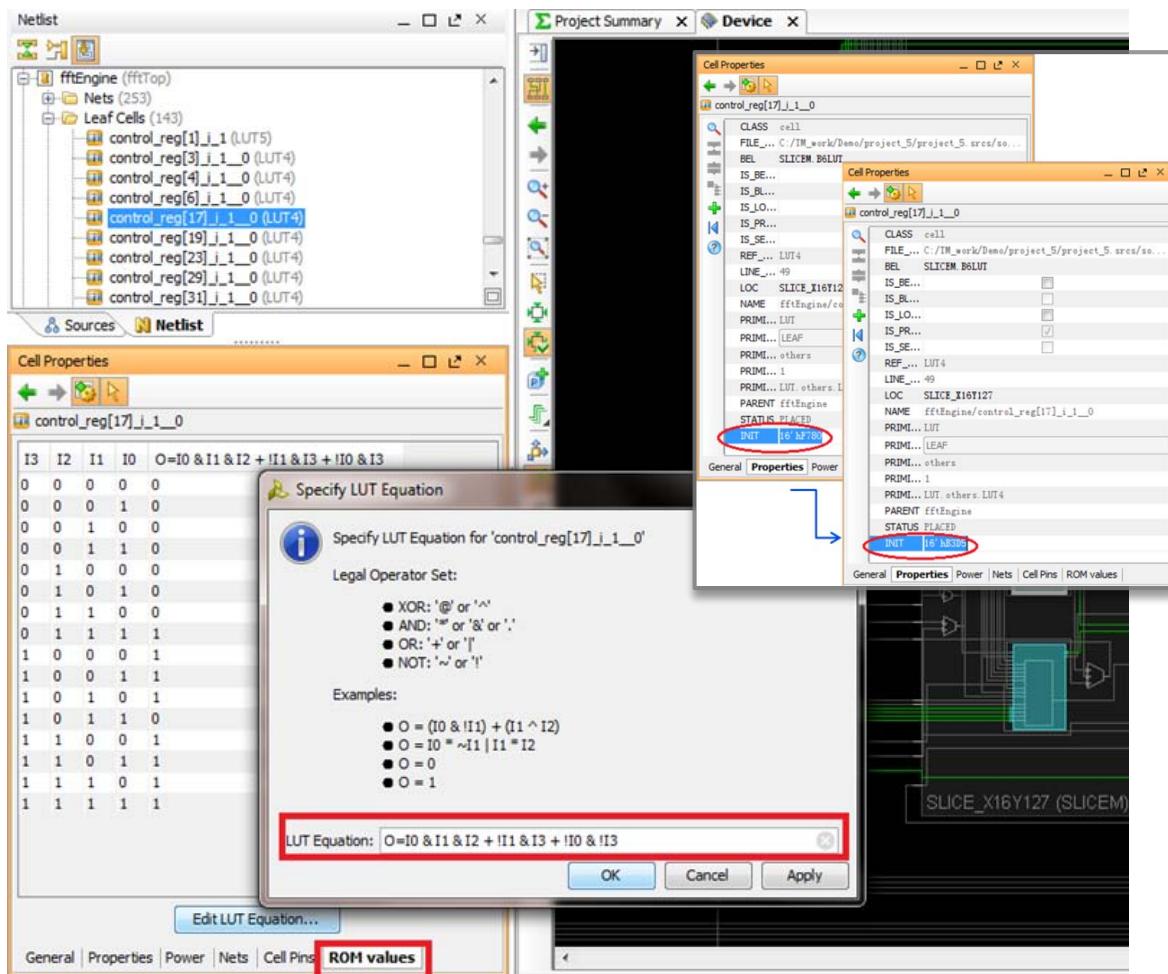
第一步所指的 Design 通常是完全布局布线后的设计，如果是在工程模式下，可以直接在 IDE 中打开实现后的设计，若是仅有 DCP 文件，不论是工程模式或是非工程模式产生的 DCP，都可以用 open_checkpoint 命令打开。

第二步就是 ECO 的意义所在，我们在布局布线后的设计上进行各种操作，然后仅对改动的部分进行局部布局/布线而无需整体重跑设计，节约大量时间的同时也不会破坏已经收敛的时序。

第三步就是产生可供下载的 bit 文件了，此时必须在 Tcl Console 中或是 Tcl 模式下直接输入命令产生 bit 文件，而不能使用 IDE 上的“Generate Bitstream”按钮。原因是后者读到的还是 ECO 前已经完成布局布线的原始设计，生成的 bit 文件自然也无法使用。

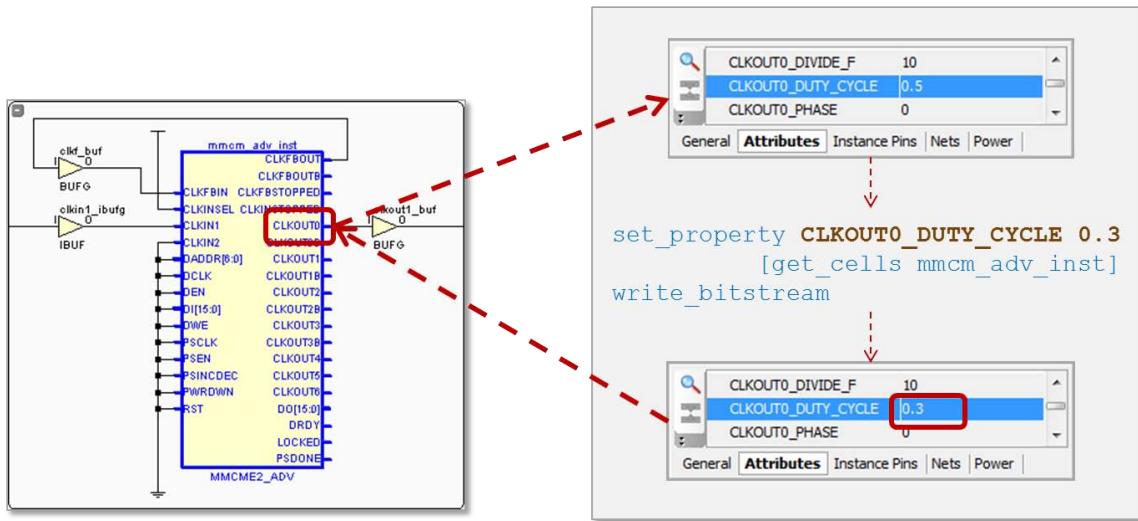
修改属性

绝大部分的属性修改都能通过 IDE 界面完成，如下图所示。



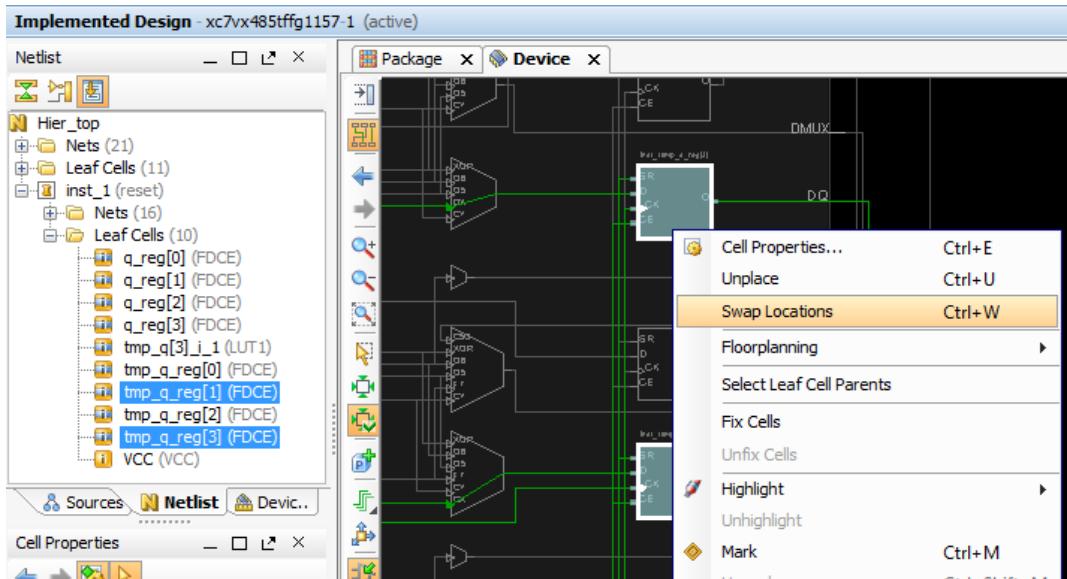
比如要修改寄存器的初值 INIT 或是 LUT 的真值表，用户只需在 Vivado IDE 中打开布局布线后的设计（Implemented Design），在 Device View 中找到并选中这个 FF/LUT，接着在其左侧的 Cell Properties 视图中选择需要修改的属性，直接修改即可。

除了对 FF/LUT 的操作外，很多时候我们需要对 MMCM/PLL 输出时钟的相移进行修改。对于这种应用，用户也无需重新产生 MMCM/PLL，与上述方法类似，可以在布局布线后的 Device View 上直接修改。



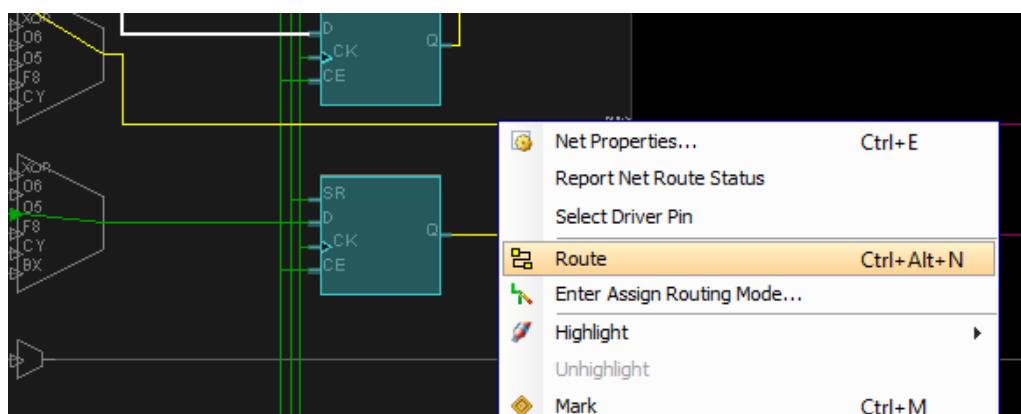
移动/交换 cells

移动/交换 cells 是对 FF/LUT 进行的 ECO 操作中最基本的一个场景，目前也只有这种情况可以通过图形化实现。如要删减 cells 等则只能通过 Tcl 命令来进行。

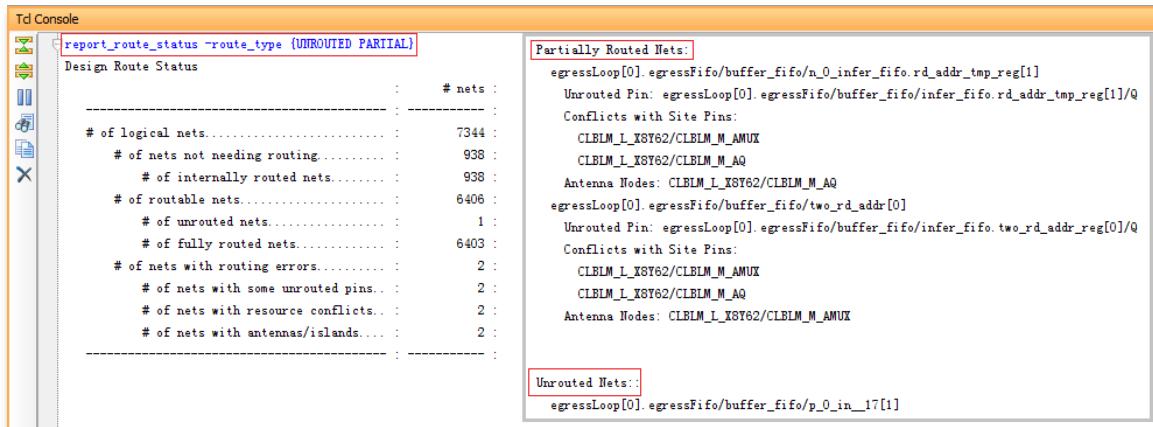


具体操作方法也相当简便，要互换 cells 位置的情况下，只要在 Device View 上选中需要的那两个 cells，如上图所示的两个 FFs，然后右键调出菜单，选择 Swap Locations 即可。若要移动 cells 则更简单，直接在图中选中 FF 拖移到新的位置即可。

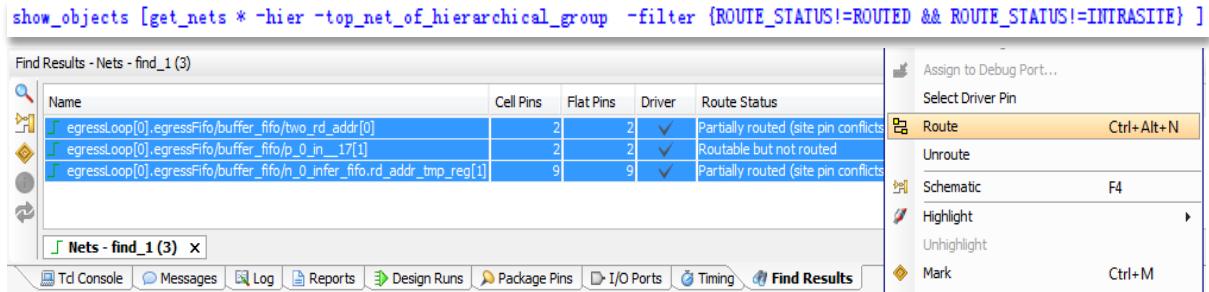
当用户移动或改变了 cells 的位置后会发现与其连接的 nets 变成了黄色高亮显示，表示这些 nets 需要重新布线。这时候需要做的就是在图中选中这些 nets 然后右键调出菜单，选择 Route 进行局部布线。



局部布线后一定要记得在 Tcl Console 中使用 report_route_status 命令检查布线情况，确保没有未完成布线 (unroute) 或是部分未完成布线 (partial routed) 的 nets 存在。给这个命令加上选项则可以报告出更细致的结果，如下图所示。



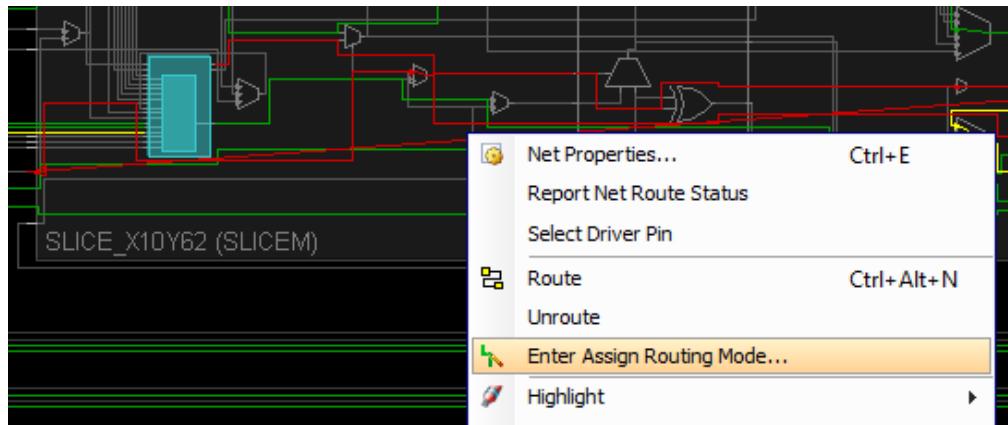
如果换个稍复杂些的 Tcl 命令配合图形化显示，更加直观的同时，也可以方便右键调出命令进行针对性的局部布线。



手动布线

手动布线是一种非常规的布线方式，一次只能针对一根 net 在图形化界面下进行。所谓手动布线，除了完全手动一个节点一个节点的选择外，也支持工具自动选择资源来布线。通常我们并不建议全手动的方式，Vivado 是时序驱动的工具，所以其自动选择的布线结果已经是遵循了时序约束下的最佳选择。

在 Device View 中选择一根没有布线或是预先 Unroute 过的 net (显示为红色高亮)，右键调出菜单并选择 Enter Assign Routing Mode... 便可进入手动布线模式。



复杂的 ECO 场景

篇幅过半，一直在铺垫，其实最有实践意义的 ECO 还没提到。相信大部分用户最怀念 FPGA Editor 中的一个功能就是 probe 了，如何快速地把一根内部信号连接到 FPGA 管脚上，无需重新布局布线，直接更新 bit 文件后下载调试。曾经数次被客户问及，很多人还为 Vivado 中不支持这样的做法而深表遗憾。

其实这样类似的功能在 Vivado 中一直支持，唯一的问题是暂时还没有图形化界面可以一键操作（相关开发工作已经在进行中）。但受益于 Tcl 的灵活多变，我们可以更有针对性地实现 probe 功能，效率也更高。

Tcl 操作命令

Netlist:

- `connect_net`
- `create_cell`
- `create_net`
- `create_pin`
- `disconnect_net`
- `get_net_delays`
- `remove_cell`
- `remove_net`
- `remove_pin`
- `rename_ref`
- `resize_net_bus`
- `resize_pin_bus`
- `tie_unused_pins`

在 UG835 中把 Vivado 支持的 Tcl 命令按照 Category 分类，这些列于 Netlist 目录下的命令就是实现 ECO 需要用到的那些。

通常涉及到增减 cells 的 ECO 基本分为三步实现：首先用 `create_cell` / `create_net` 等创建相关 cell 和/或 net，然后用 `disconnect_net` / `connect_net` 等命令修正因为 cell 和 net 的改动而影响到的连接关系，最后用 `route_design` 加选项完成局部布线。

不同的 Vivado 版本对此类 ECO 修改有稍许不同的限制，例如在 2014.1 之后的版本上，需要在改变 cell 的连接关系前先用 `unplace_cell` 将 cell 从当前的布局位置上释放，在完成新的连接关系后，再用 `place_cell` 放到新的布局位置上。

具体操作上可以根据 Vivado 的提示或报错信息来改动具体的 Tcl 命令，但操作思路和可用的命令相差无几。

Add Probe

这是一个在 Vivaod 上实现 probe 功能的 Tcl 脚本，已经写成了 proc 子程序，简单易懂。可以直接调用，也可以做成 Vivado 的嵌入式扩展命令。调用其生成 probe 只需先 source 这个脚本，然后按照如下所示在 Tcl Console 中输入命令即可。

```
Vivado% addProbe inst_1/tmp_q[3] D9 LVCMOS18 my_probe_1
```

该脚本已经在 Vivado2014.3 和 2014.4 上测试过，一次只能完成一个 probe 的添加，而且必须按照上述顺序输入信号名，管脚位置，电平标准和 probe 名。因为不具备预检功能，可能会碰到一些报错信息而导致无法继续。例如选择的信号是只存在于 SLICE 内部的 INTRASITE 时，则无法拉出到管脚。再比如输入命令时拼错了电平标准等，也会造成 Tcl 已经部分修改 Vivado 数据库而无法继续的问题。此时只能关闭已经打开的 DCP 并选择不保存而重新来过。

```

proc addProbe {signal pin IOStandard name} {
    # get placed cells already attached to net to be debugged
    set cells_to_unplace [get_cells -filter {IS_PRIMITIVE} -of [get_nets -segments $signal]]
    #save the LOCs of the placed cells
    foreach cell $cells_to_unplace {
        set cell_loc($cell) [get_property LOC $cell]
        set cell_bel($cell) [get_property BEL $cell]
    }
    # unplace the cells
    unplace_cell $cells_to_unplace
    # create the port, attach an OBUF
    create_port -direction OUT $name
    create_cell -reference OBUF $name\_obuf
    create_net $name\_obuf_net
    # connect OBUF output to the new port
    connect_net -net [get_nets $name\_obuf_net] -objects "[get_pins $name\_obuf/0] [get_ports $name]"
    # connect net to be debugged to the OBUF
    connect_net -hier -net $signal -objects [get_pins $name\_obuf/1]
    # set the debug probe I/O standards and package pin
    set_property IOSTANDARD $IOStandard [get_port $name]
    set_property PACKAGE_PIN $pin [get_port $name]
    # LOC the unplaced cells
    foreach cell $cells_to_unplace {
        place_cell $cell $cell_loc($cell)/$cell_bel($cell)
    }
    # re-route
    route_design -nets [get_nets $signal]
}

```

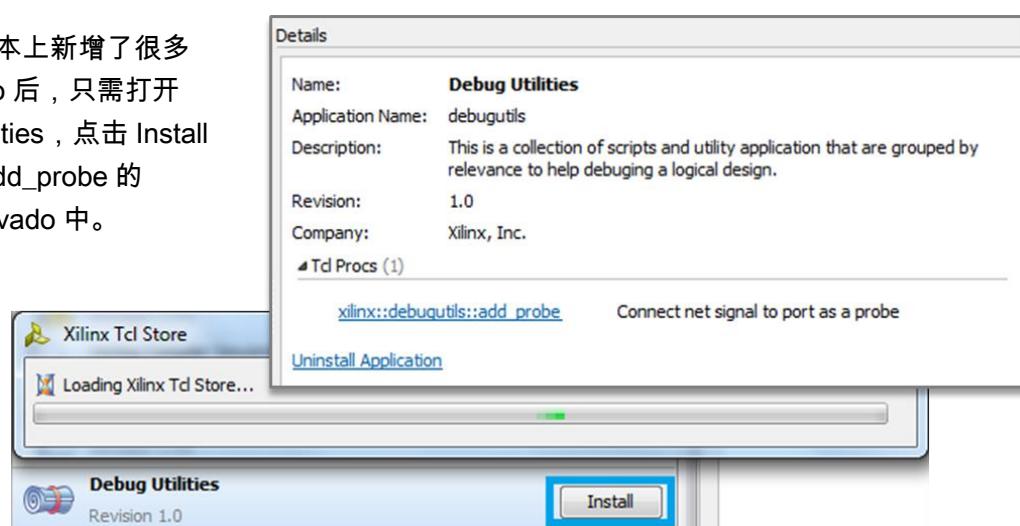
用户可以根据自己的需要扩展这个 Tcl 脚本，也可以仿照这个 Tcl 的写法来实现其它的 ECO 需求。例如文章开始举例时提到过一个将 RAMB 输出一级的 FF 拉出到 Fabric 上实现的场景，基本的实现方法和思路也类似：先将 RAMB 的输出口 REG 的属性改为 0，然后创建一个新的 FF，将其输入与 RAMB 的输出连接，再将 FF 的输出与原本 RAMB 输出驱动的 cell 连接，并完成 FF 的时钟和复位端的正确连接，然后选择合适的位置放置这个新的 FF，最后针对新增加的 nets 局部布线。

由此可见，用 Tcl 来实现的 ECO 虽然不及图形化界面来的简便直观，但是带给用户的却是最大化的自由。完全由用户来决定如何修改设计，那怕是在最后已经完成布局布线时序收敛的结果上，也能直接改变那些底层单元的连接关系，甚至是增减设计。

ECO 在 Vivado 上的发展

经过了两年多的发展，在 Vivado 上实现 ECO 已经有了多种方式，除了前面提到的图形化上那些可用的技巧，还有用户自定义的 Tcl 命令和脚本等。随着 Xilinx Tcl Store 的推出，用户可以像在 App Store 中下载使用 app 一样下载使用 Tcl 脚本，简化了 Tcl 在 Vivado 上应用的同时，进一步扩展了 Tcl 的深入、精细化使用，其中就包括 Tcl 在 ECO 上的应用。

目前 Vivado 2014.4 版本上新增了很多有用的脚本。安装好 Vivado 后，只需打开 Tcl Store，找到 Debug Utilities，点击 Install 稍等片刻，即可看到一个 add_probe 的 Tcl proc 被安装到了你的 Vivado 中。



这个 add_probe 是在上述 addProbe 例子的基础上扩展而来，不仅可以新增 probe，而且可以改变现有 probe 连接的信号。此外，这个脚本采用了 argument 写法，点击程序可以看 help，所以不一定要按照顺序输入信号、电平标准等选项，输错也没有问题。另外增加了预检和纠错功能，碰到问题会报错退出而不会改变 Vivado 数据库，效率更高。

此外，Tcl Store 上还有很多其它好用的脚本，欢迎大家试用并反馈给我们宝贵意见。虽然里面关于 ECO 的脚本还很少，但我们一直在补充。此外 Tcl Store 是一个基于 GitHub 的完全开源的环境，当然也欢迎大家上传自己手中有用的 Tcl 脚本，对其进行补充。

总体来说，ECO 是一个比较大的命题，因为牵扯到的改动需求太多，其实也很难限制在一个 GUI 界面中实现。这篇文章的目的就是为了让大家对在 FPGA 上实现 ECO 有个基本的认识，梳理看似复杂无序的流程，所谓观一叶而知秋，窥一斑而见全豹，希望能带给更多用户信心，用好 Vivado 其实一点都不难。

——Ally Zhou , 2015-2-9 于 Xilinx 上海 Office

[返回目录页](#)



读懂用好 Timing Report

《[XDC 约束技巧](#)》系列中讨论了 XDC 约束的设置方法、约束思路和一些容易混淆的地方。我们提到过约束是为了设计服务，写入 Vivado® 中的 XDC 实际上就是用户设定的目标，Vivado 对 FPGA 设计的实现过程必须以满足 XDC 中的约束为目标进行。那我们如何验证实现后的设计有没有满足时序要求？又如何在开始布局布线前判断某些约束有没有成功设置？或是验证约束的优先级？这些都要用到 Vivado 中的静态时序分析工具。

静态时序分析

静态时序分析 (Static Timing Analysis) 简称 STA，采用穷尽的分析方法来提取出整个电路存在的所有时序路径，计算信号在这些路径上的传播延时，检查信号的建立和保持时间是否满足时序要求，通过对最大路径延时和最小路径延时的分析，找出违背时序约束的错误并报告。

STA 不需要输入向量就能穷尽所有的路径，且运行速度很快、占用内存较少、覆盖率极高，不仅可以对芯片设计进行全面的时序功能检查，而且还可以利用时序分析的结果来优化设计。所以 STA 不仅是数字集成电路设计 Timing Sign-off 的必备手段，也越来越多地被用到设计的验证调试工作中。

STA 在 FPGA 设计中也一样重要，但不同于一般数字集成电路的设计，FPGA 设计中的静态时序分析工具一般都整合在芯片厂商提供的实现工具中。在 Vivado 中甚至没有一个独立的界面，而是通过几个特定的时序报告命令来实现。

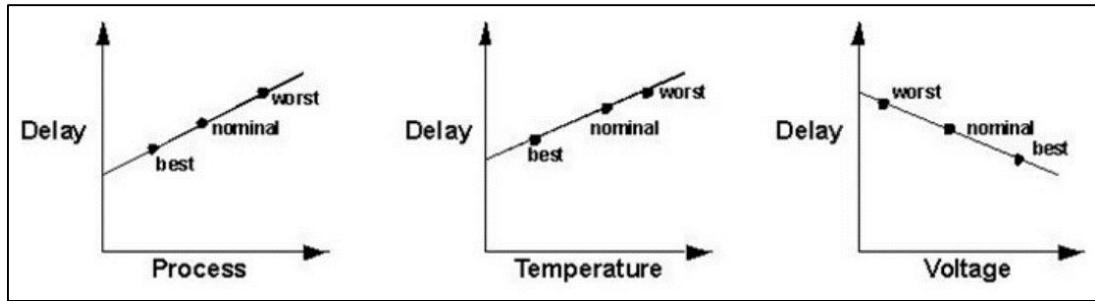
OCV 与 PVT

即便是同一种 FF，在同一个芯片上不同操作条件下的延时都不尽相同，我们称这种现象为 OCV (on-chip variation)。OCV 表示的是芯片内部的时序偏差，虽然很细小，但是也必须严格考虑到时序分析中去。

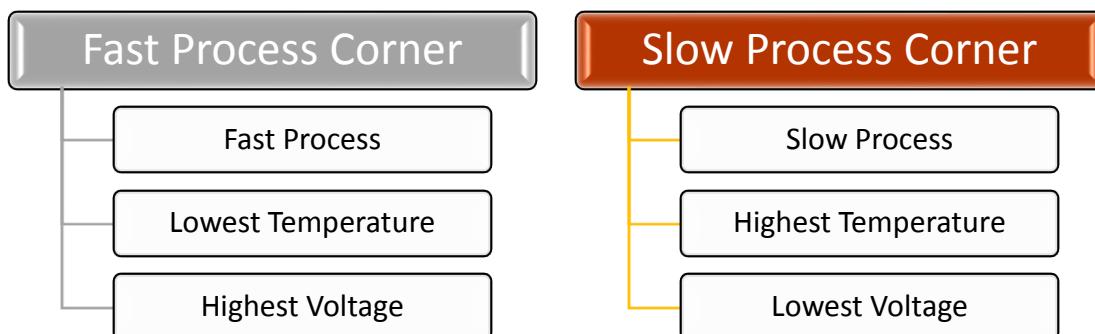
产生 OCV 的原因主要有 PVT (Process / Voltage / Temperature) 三个方面，而 STA 要做的就是针对不同工艺角(Process Corner)下特定的时序模型来分析时序路径，从而保证设计在任何条件下都能满足时序要求，可以正常工作。

通常 PVT 对芯片性能的影响如下图所示，

VIVADO®

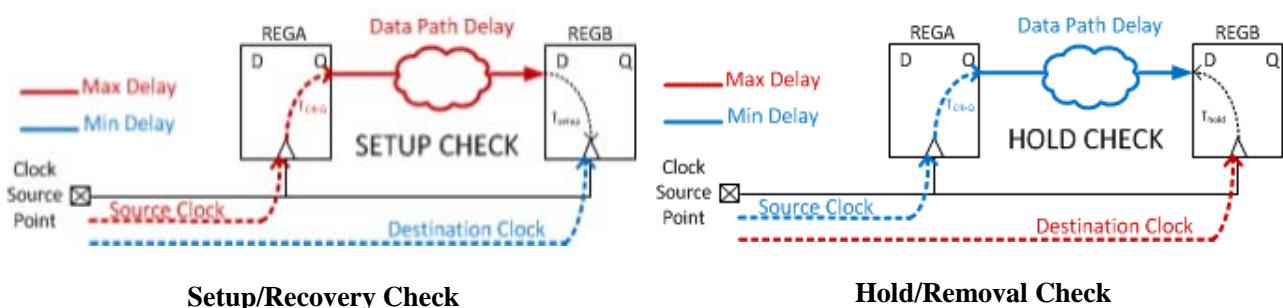


不同的 PVT 条件组成了不同的 corner，另外在数字电路设计中还要考虑 RC corner 的影响，排列组合后就可能有超过十种的 corner 要分析。但是在 FPGA 设计中的静态时序分析一般仅考虑 Best Case 和 Worst Case，也称作 Fast Process Corner 和 Slow Process Corner，分别对应极端的 PVT 条件。



Multi-Corner

Vivado 中的 STA 支持多角时序分析 (Multi-Corner Timing Analysis)，会对以上两种 corner 下的时序同时进行分析，然后报告最差的情况。因为每个 corner 下的延时也会有一定的变化范围，所以时序分析还会考虑每种 corner 下的最大延时和最小延时。



如果一个设计在 Best Case 和 Worst Case 下都能满足时序要求，则可以推算这个设计在其允许的任何操作条件下都能保持正常工作。

这里要提醒大家，不要被 corner 的名字误导，实际上，同样一条路径可能在 Slow Corner 中满足时序却在 Fast Corner 中有时序违例。但是你在 Vivado 中看到的时序报告只会显示其对两种 corner 并行分析后选出的最差情况。

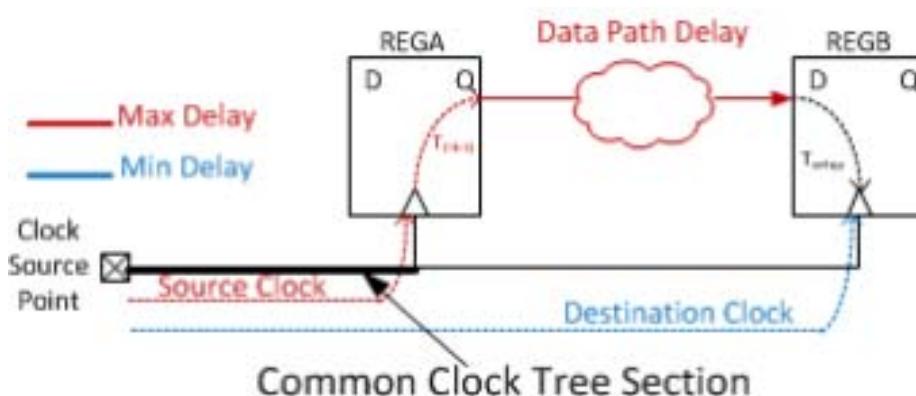
Slack (MET) : 0.156ns Source: dea_cub_reg/Q (rising edge-triggered cell FDCE clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Destination: dea_op (output port clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Path Group: sysclk Path Type: Max at Slow Process Corner Requirement: 10.000ns Data Path Delay: 5.335ns (logic 4.290ns (80.406%) route 1.045ns (19.595%))	Max = max delay analysis = setup/recovery check
Slack (MET) : 1.914ns Source: dca_ip (input port clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Destination: dca_inb_reg/D (rising edge-triggered cell FDCE clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Path Group: sysclk Path Type: Max at Fast Process Corner Requirement: 10.000ns Data Path Delay: 1.048ns (logic 0.542ns (51.716%) route 0.506ns (48.284%))	
Slack (MET) : 0.179ns Source: ssdca_fab_reg/Q (rising edge-triggered cell FDCE clocked by sscaclk {rise@0.000ns fall@2.500ns period=5.000ns}) Destination: ssdca_cub_reg/D (rising edge-triggered cell FDCE clocked by sscaclk {rise@0.000ns fall@2.500ns period=5.000ns}) Path Group: sscaclk Path Type: Min at Fast Process Corner Requirement: 0.000ns Data Path Delay: 0.179ns (logic 0.078ns (43.627%) route 0.101ns (56.373%))	Min = min delay analysis = hold/removal check
Slack (MET) : 0.940ns Source: dca_ip (input port clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Destination: dca_inb_reg/D (rising edge-triggered cell FDCE clocked by sysclk {rise@0.000ns fall@5.000ns period=10.000ns}) Path Group: sysclk Path Type: Min at Slow Process Corner Requirement: 0.000ns Data Path Delay: 1.447ns (logic 0.630ns (43.537%) route 0.817ns (56.463%))	

有特殊需要的情况下，可以在 Vivado 中通过 `config_timing_corners -corner <Slow|Fast> -delay_type <none|min|max|min_max>` 来选择将某种 corner 应用于 setup 和/或 hold 的分析。在 Report Timing Summary 和 Report Timing 的图形化界面也可以通过 Timer Setting 对 corner 做调整，具体界面详见稍后描述。

这样最大化考虑 OCV 的时序分析方法在处理同一条路径的共同时钟路径时也会应用不同的延时数据，从而会得出更为悲观的数据。为了真实反映路径延时情况，这部分延时必须被纠正，这就是 CRPR (Clock Reconvergence Pessimism Removal)。

仔细观察时序报告便可以发现在报告路径的 Slack 之前有一行显示 clock pessimism 已经被考虑在内，在进行 Setup Check 时会加上一定的 clock pessimism，而 Hold Check 时则会减去一定的 clock pessimism。

下图显示了 CRPR 的来源以及在 Vivado 时序报告中的具体体现。



```
Setup Analysis - Destination Clock path details
  (clock sysclk rise edge) 10.000 10.000 r
  V20                      0.000 10.000 r system_clk
  V20                      IBUF (Prop_ibuf_I_O) 0.741 10.741 r system_clk_IBUF_inst/O
  BUFR_X0Y1                BUFR (Prop_bufr_I_O) 0.917 11.658 r cb_sysclk/BUFR_i/O
  SLICE_X0Y41               net (fo=6) 0.621 12.279 r dca_oub_reg/C
  clock pessimism          0.193 12.472
  clock uncertainty        -0.035 12.437
```

```
Hold Analysis - Destination Clock path details
  (clock sysclk rise edge) 0.000 0.000 r
  V20                      0.000 0.000 r system_clk
  V20                      IBUF (Prop_ibuf_I_O) 0.829 0.829 r system_clk_IBUF_inst/O
  BUFR_X0Y1                BUFR (Prop_bufr_I_O) 0.982 1.811 r cb_sysclk/BUFR_i/O
  SLICE_X0Y41               net (fo=6) 0.661 2.472 r dca_inb_reg/C
  clock pessimism          -0.249 2.223
  clock uncertainty        0.035 2.258
```

时序命令与报告

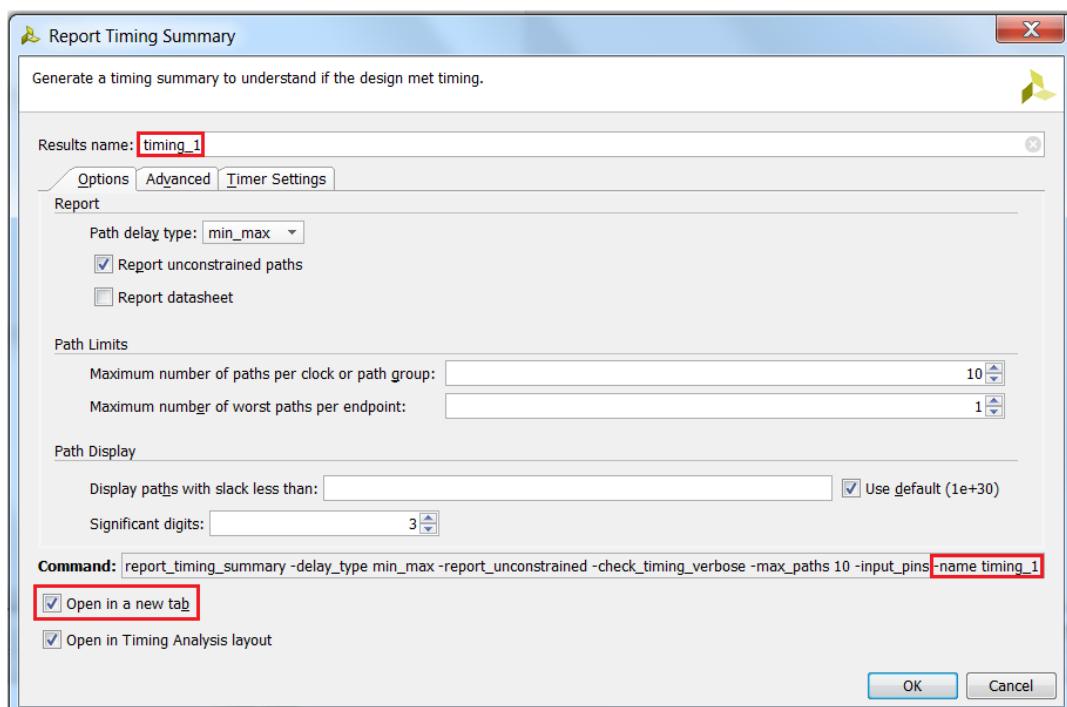
Vivado 中用于时序分析的命令主要有以下两条，且都有对应的图形化设置界面。

- ✓ `report_timing_summary` 主要用于实现后的 timing sign-off
- ✓ `report_timing` 主要用于交互式的约束验证以及更细致具体的时序报告与分析

report_timing_summary

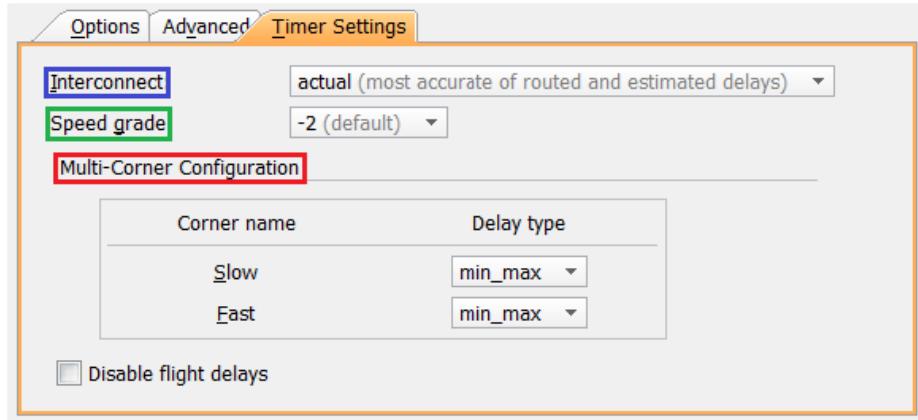
我们先看看 `report_timing_summary`，实际上，不仅在布局布线后，在综合后甚至是更具体的实现过程中的每一步之后都可以运行，从而得到一个全局的时序报告。

在 Vivado IDE 中点击 Report Timing Summary 后可以改变报告的内容，例如每个时钟域报告的路径条数，是否 setup 和 hold 全都报告等等。每改变一个选项都可以看到窗口下方的 Command 一栏显示出对应的 Tcl 命令。修改完设置后可以直接按 OK 键确认执行，也可以拷贝 Command 栏显示的命令到 Tcl 脚本中稍后执行。



这里有个小窍门，通过`-name`指定一个名字，就可以在 Vivado IDE 中新开一个窗口显示这条命令的执行结果，这个窗口还可以用来跟其他诸如 Device View 或是 Schematic View 等窗口之间 cross probing。这一点也同样适用于包括 report_timing 在内的绝大部分 Vivado 中的 report 命令。

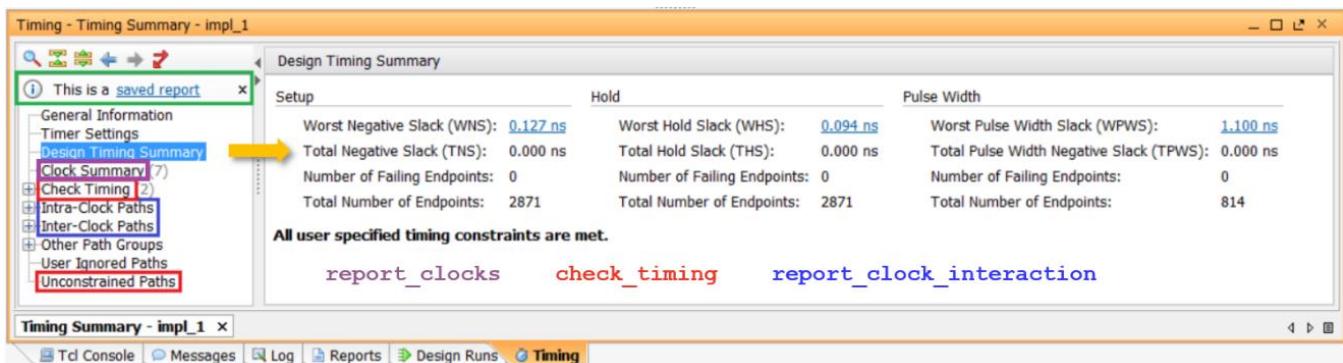
在设置窗口中还有 Timer Settings 一栏（report_timing 中也有），可以用来改变报告时采用的具体 corner、速度等级以及计算布线延时的方式。很多时候我们可以借助 Timer 的设置来快速验证和调试设计需求。



举例来说，在实现后的报告中显示时序违例比较严重，我们可以直接在 Timer 设置中改变速度等级后重新报告时序，来验证把当前这个已经布局布线完毕的设计切换到更快一档的芯片中是否可以满足时序要求。

另外，在布局布线后的设计上报告时序，往往不能更直观地发现那些扇出较大或是逻辑级数较高的路径。此时我们可以修改连线模型为 estimated，报告出布局后布线前的时序而无需另外打开对应阶段的 DCP 并重新运行时序报告命令来操作，这么做节约时间的同时，也更容易找到那些高扇出路径以及由于布局不佳而导致的时序违例。我们也可以修改连线模型为 none，这样可以快速报告出那些逻辑延时较大以及逻辑级数较高的路径。以上这些改变 Timer 设置的方法可以帮助我们快速定位设计中可能存在的问题和缺陷。

report_timing_summary 实际上隐含了 report_timing、report_clocks、check_timing 以及部分的 report_clock_interaction 命令，所以我们最终看到的报告中也包含了这几部分的内容。另外自 2014.3 版起，打开实现后的结果时会直接打开一个预先产生好的报告。



Timing Summary 报告把路径按照时钟域分类，每个组别下缺省会报告 Setup、Hold 以及 Pulse Width 检查最差的各 10 条路径，还可以看到每条路径的具体延时报告，并支持与 Device View、Schematic View 等窗口之间的交互。

每条路径具体的报告会分为 Summary、Source Clock Path、Data Path 和 Destination Clock Path 几部分，详细报告每部分的逻辑延时与连线延时。用户首先要关注的就是 Summary 中的几部分内容，发现问题后再根据具体情况来检查详细的延时数据。其中，Slack 显示路径是否有时序违例，Source 和 Destination 显示源驱动时钟和目的驱动时钟及其时钟频率，Requirement 显示这条路径的时序要求是多少，Data Path 显示数

据路径上的延时，Logic Level 显示这条路径的逻辑级数，而 Clock Path Skew 和 Clock Uncertainty 则显示时钟路径上的不确定性。

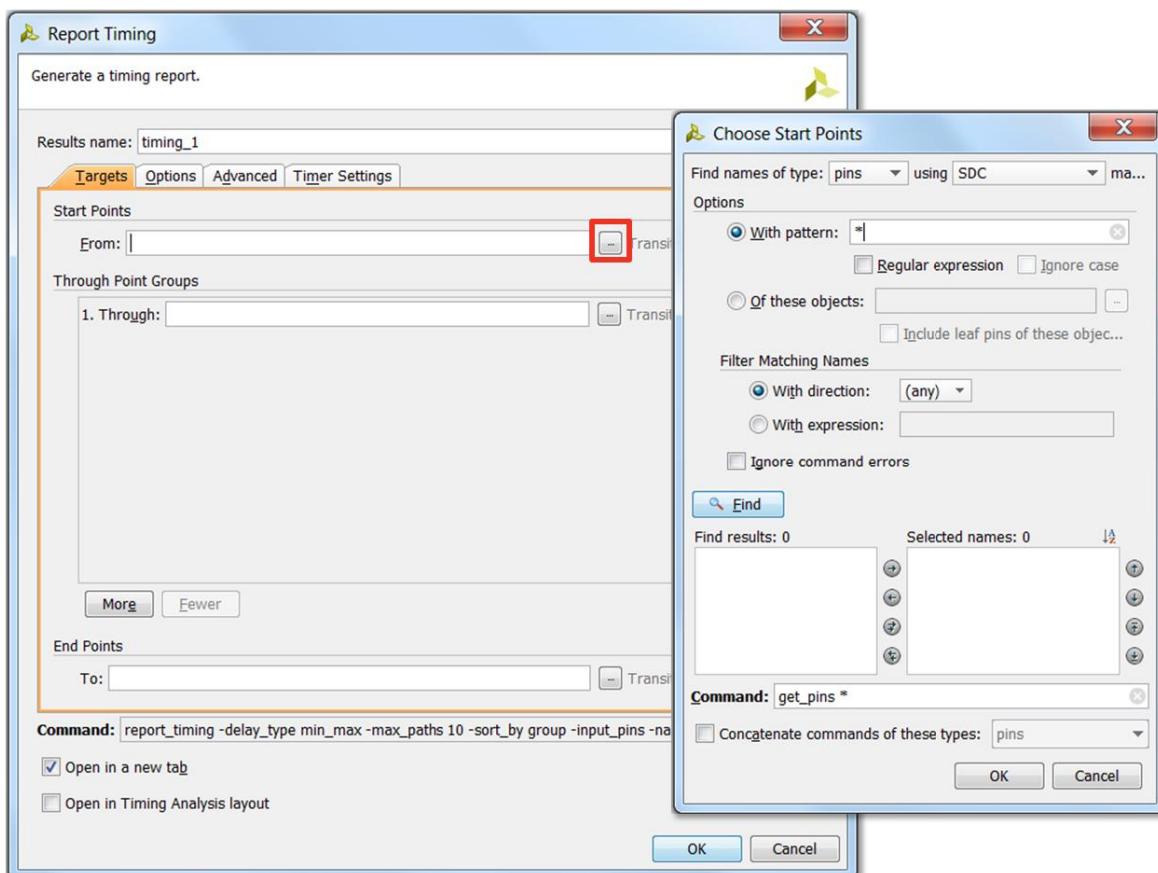
Summary	
Name	Path 1
Slack	-0.268ns
Source	i_CORES/i_TX/i_STRIPE/i_ADAPT/ready_to_dequeue/C (rising edge-triggered cell FDRE clocked by clk_ (rise@0.000ns fall@1.562ns period=3.125ns))
Destination	i_CORES/i_TX/i_STRIPE/i_ADAPT/burstshort_ptr_ok/D (rising edge-triggered cell FDSE clocked by clk_ (rise@0.000ns fall@1.562ns period=3.125ns))
Path Group	clk
Path Type	Setup (Max at Slow Process Corner)
Requirement	3.125ns
Data Path Delay	3.338ns (logic 0.552ns (16.536%) route 2.786ns (83.464%))
Logic Levels	6 (LUT2=1 LUT3=1 LUT5=1 LUT6=3)
Clock Path Skew	-0.020ns
Clock Uncertainty	0.035ns

以上图这条路径来举例，通过 Summary 我们可以得到这样的信息：这是一条 clk 时钟域内的路径，时钟周期为 3.125ns，这条路径有 0.268ns 的时序违例。违例的主要原因是逻辑级数较高导致的数据链路延时较大，但连线延时的比例也较高，所以可以仔细看看这条路径的数据路径上有没有可能改进布局、降低扇出或者是减少逻辑级数的优化方向。

report_timing

report_timing 是更具体的时序报告命令，经常用来报告某一条或是某些共享特定节点的路径。用户可以在设计的任何阶段使用 report_timing，甚至是一边设置 XDC，一边用其来验证约束的可行性与优先级。在 Vivado IDE 中可以由 Tools > Timing > Report Timing 调出其图形化设置窗口。

与 report_timing_summary 类似，调整选项后对应的 Tcl 命令也会在 Command 栏生成，在 Targets 一栏还可以设置需要报告路径的起始点/途经点/结束点，可以三个都设置或是仅设置其中任何一项，每一项都支持通配符匹配甚至是正则表达式查找。report_timing 报告出的路径延时与 report_timing_summary 中具体到每根路径上的报告一致，可以以此为依据帮助我们定位时序失败的原因。



用 report_timing 来报告时序其实还有一些更常见的应用场景，用来帮助我们设置和验证约束，尤其是那些时序例外约束。

举例来说，在设计过程中我们约束了一条或多周期约束，不同于 UCF 必须读入约束后重跑设计，我们可以直接在 Tcl Console 中输入这条 XDC，无需重跑设计，直接用 report_timing 来验证。在随之显示的时序报告 Summary 部分可以看到 Timing Exception 后列出这条路径被设置了怎样的时序例外约束（注意，不加额外 option 时，以下两条命令都仅针对 setup check）。

```
set_multicycle_path -from [get_cells rst_gen_i0/reset_bridge_clk_rx_i0/rst*] -to [get_cells clkx_spd_i0/bus_samp_src_reg* ] 2
report_timing -from [get_cells rst_gen_i0/reset_bridge_clk_rx_i0/rst*] -to [get_cells clkx_spd_i0/bus_samp_src_reg* ]
```

Timing Exception:	MultiCycle Path	Setup -end	2
-------------------	-----------------	------------	---

单纯的一条多周期约束没有什么特别，但是如果使用了通配符后的时序例外有重叠的情况下，Vivado 会根据优先级来决定对某条路径应用怎样的约束。当设计较大，XDC 较多时，一边设置 XDC 一边用 report_timing 来验证就变得尤其重要。

另外，仅仅输入 report_timing 而不加任何 option，Vivado 便会报告出时序违例最严重的那条路径，方便我们快速了解当前设计的 WNS，找到最差的那条路径。在验证 I/O 约束时也常常用到 report_timing，只要指定-from 某个输入或是-to 某个输出便可以快速验证当前设计在接口上的时序。

get_timing_paths

report_property -all [get_timing_paths]			
Property	Type	Read-only	Value
CLASS	string	true	timing_path
CLOCK_PESSIMISM	double	true	-0.656
CORNER	string	true	Slow
DAIAPATH_DELAY	double	true	1.116
DELAY_IYPE	string	true	max
ENDPOINT_CLOCK	clock	true	clk_tx_clk_core
ENDPOINT_CLOCK_DELAY	double	true	-2.197
ENDPOINT_CLOCK_EDGE	double	true	3.000
ENDPOINT_PIN	pin	true	dac_spi_i0/out_ddr_flop_spi_clk_i0/ODDR_inst/D2
EXCEPTION	string	true	
GROUP	string	true	clk_tx_clk_core
INPUT_DELAY	double	true	
INTER_SLR_COMPENSATION	double	true	
LOGIC_LEVELS	int	true	1
NAME	string	true	{dac_spi_i0/old_active_reg/C --> dac_spi_i0/out_ddr_flop_spi_clk_i0/ODDR_inst/D2}
OUTPUT_DELAY	double	true	
REQUIREMENT	double	true	3.000
SKEW	double	true	-0.023
SLACK	double	true	1.325
STARTPOINT_CLOCK	clock	true	clk_tx_clk_core
STARTPOINT_CLOCK_DELAY	double	true	-2.829
STARTPOINT_CLOCK_EDGE	double	true	0.000
STARTPOINT_PIN	pin	true	dac_spi_i0/old_active_reg/C
UNCERTAINTY	double	true	0.062
USER_UNCERTAINTY	double	true	

除了上述两个大家比较熟悉的时序报告命令，Vivado 中还提供一个 get_timing_paths 的命令，可以根据指定的条件找到一些特定的路径。我们可以利用其返回值中的一些属性来快速定位设计中的问题。

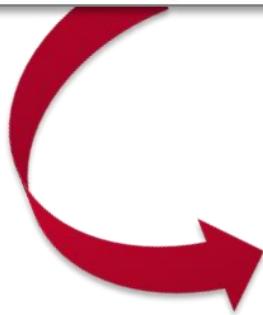
例如逻辑级数这个影响 FPGA 性能的一大因素，因为经常隐藏在时序报告后很难被发现。在 Vivado 中，除了借助综合后的报告来找到那些可能因为逻辑级数较高而导致的时序难满足的路径外，还有一个更直接的办法，可以一次性报告出设计中那些高逻辑级数的路径，方便我们有针对性的深入分析和优化。

下图这个例子报告了时序最差的 10 条路径的逻辑级数。需要注意的是，在综合后和在布局布线后用一样的脚本报告出的结果会稍有不同，对逻辑级数较为关注的情况，还是建议以综合后的结果为主要依据。

```

set numpaths 10
for {set path 0} {$path < $numpaths} {incr path} {
    puts -nouline "LOGIC LEVELS for worst path $path ="
    set logiclevels [get_property LOGIC_LEVELS [lindex [get_timing_paths -nworst 5 -max_paths $numpaths] $path]]
    puts "$logiclevels"
}

```



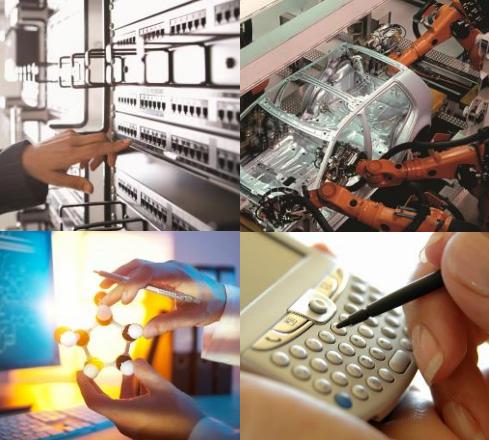
LOGIC LEVELS for worst path 0 =1
LOGIC LEVELS for worst path 1 =1
LOGIC LEVELS for worst path 2 =1
LOGIC LEVELS for worst path 3 =0
LOGIC LEVELS for worst path 4 =0
LOGIC LEVELS for worst path 5 =2
LOGIC LEVELS for worst path 6 =2
LOGIC LEVELS for worst path 7 =2
LOGIC LEVELS for worst path 8 =2
LOGIC LEVELS for worst path 9 =1

小结

本文可以视为对[《XDC 约束技巧》](#)系列文章的补充，希望可以帮助大家了解 FPGA 设计中的时序分析方法，学会使用 Vivado 中的静态时序分析工具来验证时序，定位问题，快速找到问题和解决方案。

—— Ally Zhou , 2015-3-30 于 Xilinx 上海 Office

[返回目录页](#)



编后记

做工程师很多年，一直有随手记笔记的习惯，有些问题被问的多了，或是自己觉得很有代表性，就会整理一下存档，所谓“好记性不如烂笔头”。大半年前开始试着把整理后的文章分享到我们的论坛，同事和客户都很喜欢，一直鼓励我继续做下去，就这样保持着一月一篇的节奏，陆陆续续居然也写出了近十篇。于是便有了这本电子书。

说实话，很感谢市场部的 Melissa 和 James，没有你们的鼓励和支持，就不会有这些文章和这本电子书。也要感谢我们的客户，还有在论坛和其他媒体上关注和留言的读者们，你们的反馈直接决定了哪些内容会出现在这本书里。更要感谢我的同事们，我们是一个团队，这里面提及的每一份经验都来自大家的努力，更多的时候，我只是一个忠实的记录者和传播者。

结集成册的《Vivado 使用误区与进阶》将要用更直接的方式面对更多的读者，衷心希望得到大家的喜爱，也欢迎大家继续提供意见和建议。我在 [Xilinx 中文官方论坛](#) 的更新还会继续，大家仍然可以在论坛上与我互动。论坛上除了能够看到最新版的文章更新，还有其他 Xilinx 专家的分享，你也可以发帖就你碰到的实际问题进行求助或探讨。

经过了三十多年的发展，Xilinx 在创新之路上依旧活力十足，在 All Programmable 时代，SDx 系列也在不断壮大。毫无疑问，能在这样一家公司做着自己感兴趣又有成就感的工作是幸运的，所以这本书不是一个结束而是一个开始。除了 Vivado，希望还有更多工具和方法学方面的经验能与大家分享，共同推进 Xilinx 领先技术的发展。

再次感谢大家的关注和支持，敬请期待更好的 Xilinx 和更好的明天。

紧急召集令

一年一度的 Club Vivado 2015 即将来临！

金秋十月，北京 | 上海

Club Vivado 再次将 Vivado 用户群和 Xilinx 开发团队召集在一起

交流最佳实践经验，聆听技术专家的意见。



CLUB
VIVADO
users group

话题调研



案例征集

 **XILINX**
ALL PROGRAMMABLE™

© Copyright 2015 Xilinx Inc.