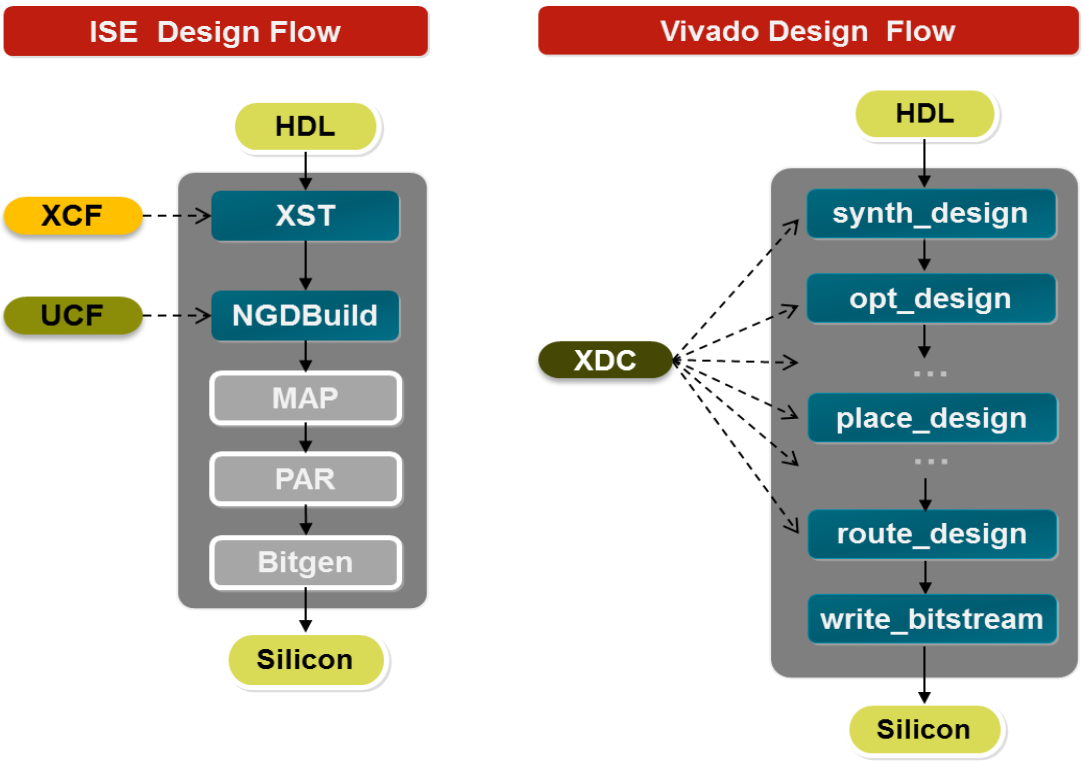
用 Tcl 定制 Vivado 设计实现流程

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。

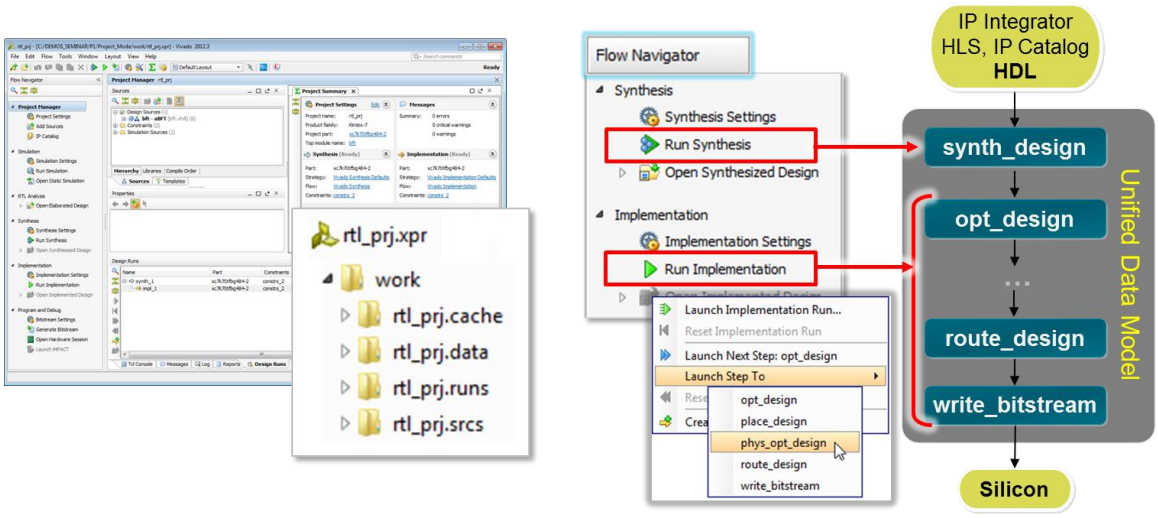
从使用方式上来讲，Vivado 支持工程模式（**Project Based Mode**）和非工程模式（**None Project Mode**）两种，且都能通过 Tcl 脚本批处理运行，或是在 Vivado 图形化界面 IDE 中交互运行和调试。

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。

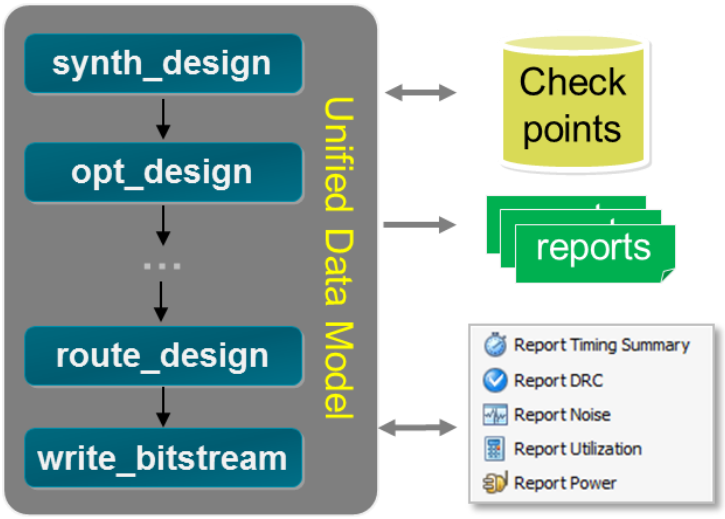


特别需要指出的是 Flow Navigator 只有在 Vivado IDE 中打开.xpr 工程文件才会显示，如果打开的是设计检查点.dcp 文件（不论是工程模式或是非工程模式产生的 dcp）都不会显示这个侧栏。

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 Vivdao 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。

下图所示是同一个设计（Vivado 自带的 Example Design）采用两种模式实现所需使用的不同脚本，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

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.vhdl
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
```

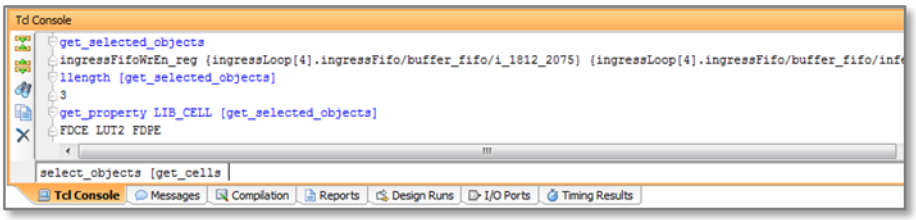
Project Mode:

```
create_project project_bft ./project_bft -part xc7k70tfbg484-2
add_files {./Sources/hdl/FifoBuffer.v ./Sources/hdl/async_fifo.v ./Sources/hdl/bft.vhdl}
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。

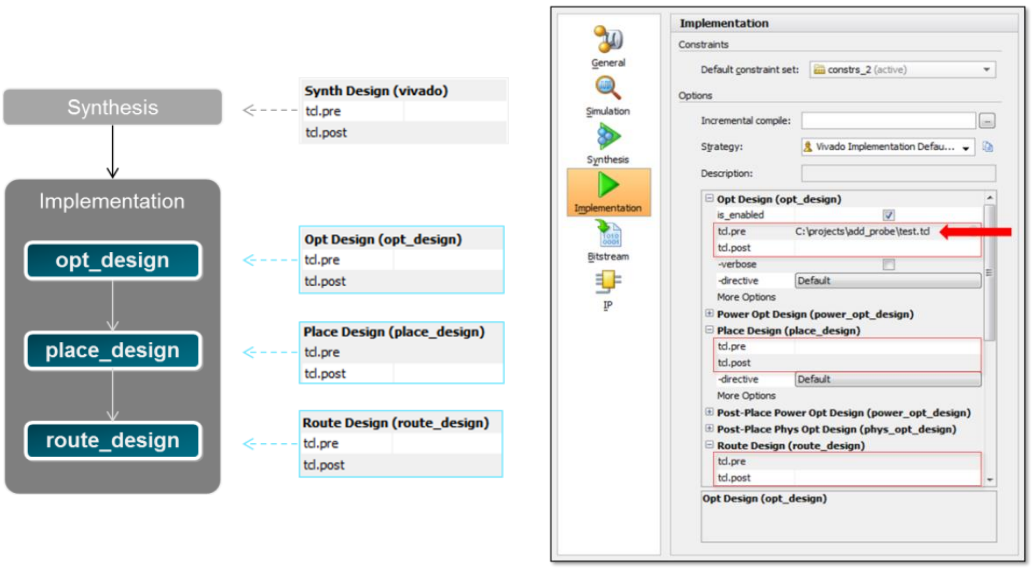
举例来说，设计调试过程中，需要将一些约束应用在某些网表目标上（具体可参照《Tcl 在 Vivado 中的应用》所示），推荐的做法就是在 IDE 中打开.dcp 然后在 Tcl Console 中输入相应的 Tcl/XDC 命令，验证返回值，碰到问题可以直接修改，直到找到正确合适的命令。然后可以记录这些命令，并存入 XDC 文件中以备下次实现时使用。

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。



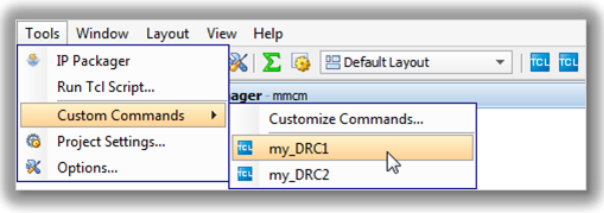
tcl.pre 表示当前这步之前 Vivado 会主动 source 的 Tcl 脚本，tcl.post 表示这步之后会 source 的脚本。Tcl 脚本必须事先写好，然后在上图所示的设置界面由用户使用弹出窗口指定到脚本所在位置。

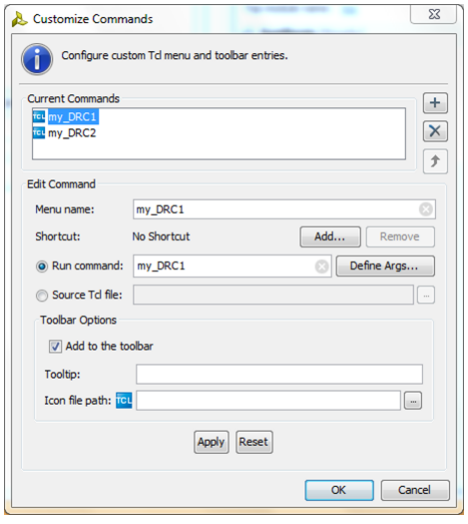
这些就是所谓的“钩子”脚本，正是有了这样的脚本，我们才得以在图形化界面上既享有一键式执行的便利，又充分利用 Tcl 带来的扩展性。比较常见的使用场景是，在某个步骤后多产生几个特别的报告，或是在布线前再跑几次物理优化等。

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。

```
# my_DRC.tcl
proc my_DRC1 {…} {…}
proc my_DRC2 {…} {…}
```



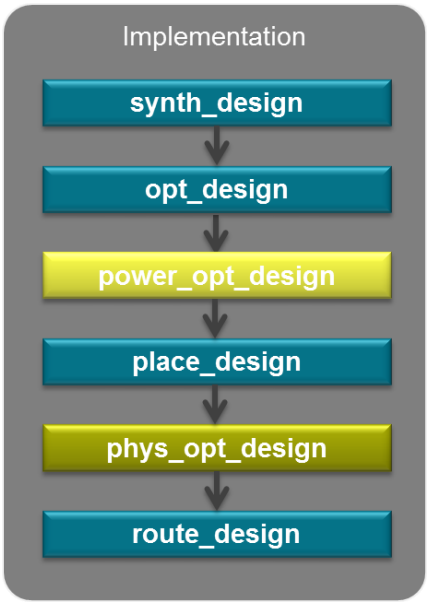


用 Tcl 定制实现流程

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

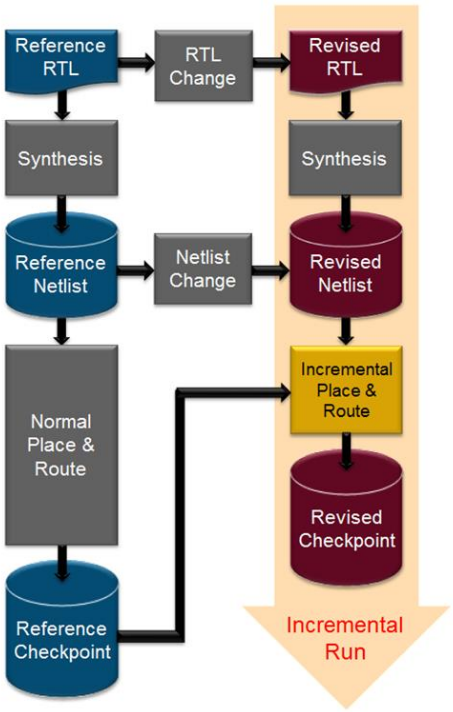
通常的 FPGA 设计流程是一个开环系统，从前到后依次执行。但 Vivado 中提供了一种可能，用户可以通过 place_design -post_place_opt 在已经完成布局布线的设计上再做一次布局布线，从而形成一个有了反馈信息的闭环系统。这次因为有了前一次布线后的真实连线延迟信息，布局的针对性更好，并且只会基于时序不满足的路径进行重布局而不会改变大部分已经存在的布局信息，之后的布线过程也是增量流程。

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
}
```

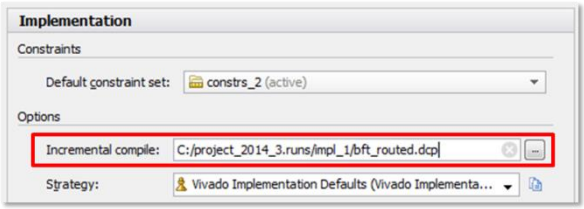
增量设计流程

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，关于更细节的 Tcl 使用场景，包括 ECO 流程等，会另外展开，敬请关注 Xilinx 官方网站和中文论坛上的更多技术文章。

Ally Zhou 2014-11-11 于 Xilinx 上海 Office