查看所有版本

Versal 自适应 SoC 硬件、IP 和平台开发方法指南

UG1387 (v2023.2) 2023 年 11 月 15 日

目录

第 1 章

简介

关于 Versal 自适应 SoC 设计方法论

AMD Versal™ 自适应 SoC 设计方法论是一整套旨在帮助简化当今 Versal 器件设计进程的最佳实践。鉴于这些设计的规模与复杂性，因此必须通过执行特定步骤与设计任务才能确保设计每个阶段都能成功完成。建议您遵循这些步骤和最佳实践进行操作，这将有助于您以尽可能最快且最高效的方式实现期望的设计目标。

按设计进程浏览内容

AMD 自适应计算文档按一组标准设计进程进行组织，以便帮助您查找当前开发任务相关的内容。您可以在设计中心页面上访问 AMD Versal™ 自适应 SoC 设计流程。您还可以使用设计流程助手来更深入了解设计流程，并找到特定于预期设计需求的内容。

- 硬件、IP 和平台开发：为硬件平台创建 PL IP 块、创建 PL 内核、功能仿真以及评估 AMD Vivado™ 时序收敛、资源使用情况和功耗收敛。还涉及为系统集成开发硬件平台。

如需获取更多方法论相关信息，请参阅以下文档：

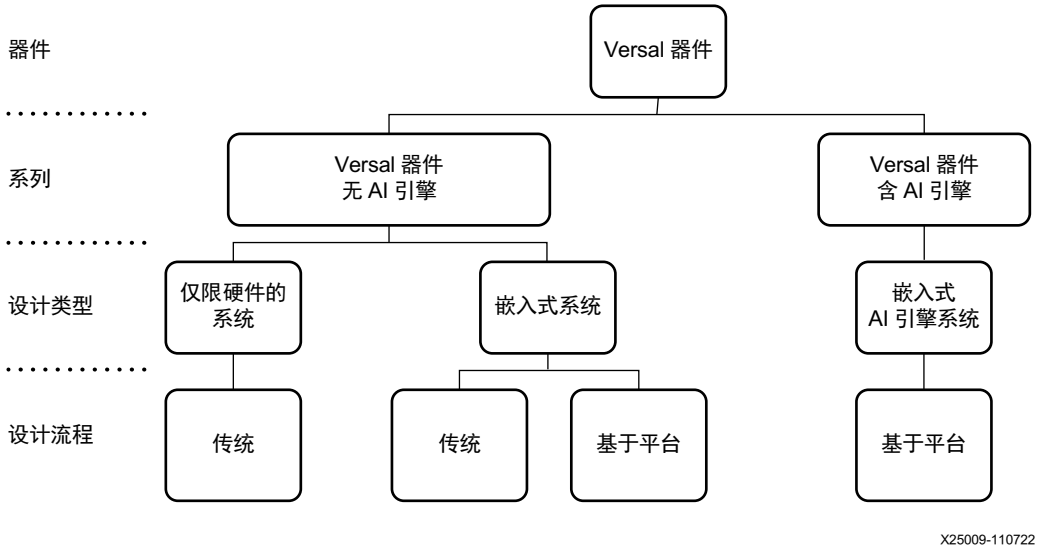
- 系统和解决方案规划：确认系统级别的组件、性能、I/O 和数据传输要求。包括解决方案到 PS、PL 和 AI 引擎的应用映射。请参阅《Versal 自适应 SoC 设计指南》(UG1273) 和《Versal 自适应 SoC 系统和解决方案规划方法指南》(UG1504)。
- 嵌入式软件开发：基于硬件平台来创建软件平台，并使用嵌入式 CPU 开发应用代码。还涵盖 XRT 和计算图 API。请访问此链接以参阅《AI 引擎工具和流程用户指南》(UG1076) 中的相应内容。
- AI 引擎开发：创建 AI 引擎计算图及内核、库用法、仿真调试与剖析以及算法开发。还包含 PL 与 AI 引擎内核的集成。请参阅《AI 引擎工具和流程用户指南》(UG1076) 和《AI 引擎内核与计算图编程指南》(UG1079)。
- 系统集成与确认：集成和确认系统功能性能，包括时序收敛、资源使用情况和功耗收敛。请参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388)。
- 开发板系统设计：通过板级原理图和开发板布局来设计 PCB。还包含功耗、散热以及信号完整性注意事项。请参阅《Versal 自适应 SoC 开发板系统设计方法指南》(UG1506)。

系统设计类型

AMD Versal™ 自适应 SoC 属于异构计算平台，具有多个计算引擎。在 Versal 自适应 SoC 上可映射各种应用，包括对无线系统、机器学习推断和视频处理算法进行信号处理。除了多个计算引擎外，Versal 自适应 SoC 还可使用高速串行 I/O、片上网络 (NoC)、DDR4/LPDDR4 存储器控制器、HBM 控制器和多重速率以太网媒体访问控制器 (MRMAC) 来提供超高系统带宽。Versal 器件分类为以下几个系列：Versal Prime、Premium、HBM、AI Core 和 AI Edge。下图显示了每种 Versal 器件系列所支持的不同系统设计类型和设计流程。

注释： Versal Prime 系列、Premium 系列和 HBM 系列的设计流程与 AMD FPGA 所使用的流程类似。Versal AI Core 系列、AI Edge 系列以及 Versal Premium VP2502 和 VP2802 器件的设计流程要求您面向异构计算平台进行设计，此平台具有特殊的硬件配置和软件支持要求。

图 1：**系统设计类型**



下表显示了每种 Versal 器件系列所支持的系统设计类型和设计流程。如该表中所示，大部分设计流程都以构建平台为基础。

表 1：**系统设计类型**

设计类型	器件系列	设计流程	平台源文件	GitHub 示例
仅限硬件的系统	Versal Prime 系列 Versal Premium 系列 Versal HBM 系列	传统	不适用	Versal 器件架构教程
嵌入式系统	Versal Prime 系列 Versal Premium 系列 Versal HBM 系列	传统	不适用	Versal 自适应 SoC 嵌入式设计教程
		基于平台	定制	Versal Prime 系列 VMK180 目标参考设计
嵌入式 AI 引擎系统	Versal AI Core 系列 Versal AI Edge 系列 Versal Premium VP2502 器件和 VP2802 器件	基于平台	定制	AI 引擎开发设计教程 VCK190 基本 TRD AI 引擎机器学习教程

提示：请访问 GitHub 以获取更多示例，这些示例会定期更新。

以下提供了每种系统设计类型的汇总信息：

- 仅限硬件的系统：可编程逻辑设计。使用传统设计流程创建此系统。
- 嵌入式系统：嵌入式处理器系统，软件在 Arm® Cortex®-A72 或 Cortex-R5F 处理器上运行，硬件内容则位于 PL 内。使用传统设计流程或基于平台的设计流程创建此系统。
- 嵌入式 AI 引擎系统：嵌入式处理器系统，软件在 Arm Cortex-A72 或 Cortex-R5F 处理器上运行，硬件内容位于 PL 内，算法内容则位于 AI 引擎内。使用基于平台的设计流程创建此系统。

Versal 自适应 SoC 的设计流程如下所示：

- 传统设计流程：在传统设计流程中，系统的整个 PL 部分都是在单个 AMD Vivado™ 工程中定义的。该工程必须包括 Versal 基础硬件 IP 块（例如，Control, Interface, and Processing System (CIPS)、NoC、I/O 控制器）以及工程所需的任何其他定制 RTL 和 IP 块。设计源文件将添加到 Vivado 工具中，并通过 Vivado 实现流程进行编译。如果系统仅包含 PL 组件，那么可使用 Vivado 工具来生成可编程器件镜像 (PDI)，以便对 Versal 器件进行编程。如果系统还包含嵌入式软件内容，那么将在从 Vivado 工具导出的固定硬件设计上的 AMD Vitis™ 环境中开发软件应用。此流程类似于用于 AMD Zynq™ UltraScale+™ MPSoC 的传统流程。
- 基于平台的设计流程：在基于平台的设计流程中，硬件系统分为下列不同元素：可复用的基础平台以及基本硬件扩展，此平台是在 Vivado 中开发的，而扩展则是在 Vitis 中通过基础平台的可扩展区域内精确定义的一组连接接口来开发的。大部分硬件设计是在 Vivado 中开发的，但设计中以 C++ 而非硬件描述语言 (HDL) 指定的部分大多是在 Vitis 中自然开发并集成的。后者示例包括 AI 引擎计算图与内核以及以通过高层次综合 (HLS) 编译的 PL 作为目标的内核函数。

  您可根据自身工作效率来选择任一设计分区方式：基础平台或可扩展区域。在整个设计周期过程中，基本硬件和可扩展区域均可进化，精心设计的基础平台能为多种应用奠定基础，以便 Vitis 工具在其中对可扩展区域进行扩展。相应开发团队可通过合理的松散耦合与紧密耦合将设计内容从 Vivado 导出到 Vitis，反之亦然，这有助于促进组成异构系统的不同元素的并发开发和集成。

设计流程

AMD Versal™ 自适应 SoC 支持 2 种设计流程：传统设计流程和基于平台的设计流程。要充分利用 Versal 自适应 SoC 资源，重要的是选择正确的设计流程。下表显示了根据设计类型和目标器件系列所使用的设计流程。

表 2：**设计流程**

设计类型	器件系列	设计流程
仅限硬件的系统	Versal Prime 系列 Versal Premium 系列 Versal HBM 系列	传统
嵌入式系统	Versal Prime 系列 Versal Premium 系列 Versal HBM 系列	传统
		基于平台
嵌入式 AI 引擎系统	Versal AI Core 系列	基于平台

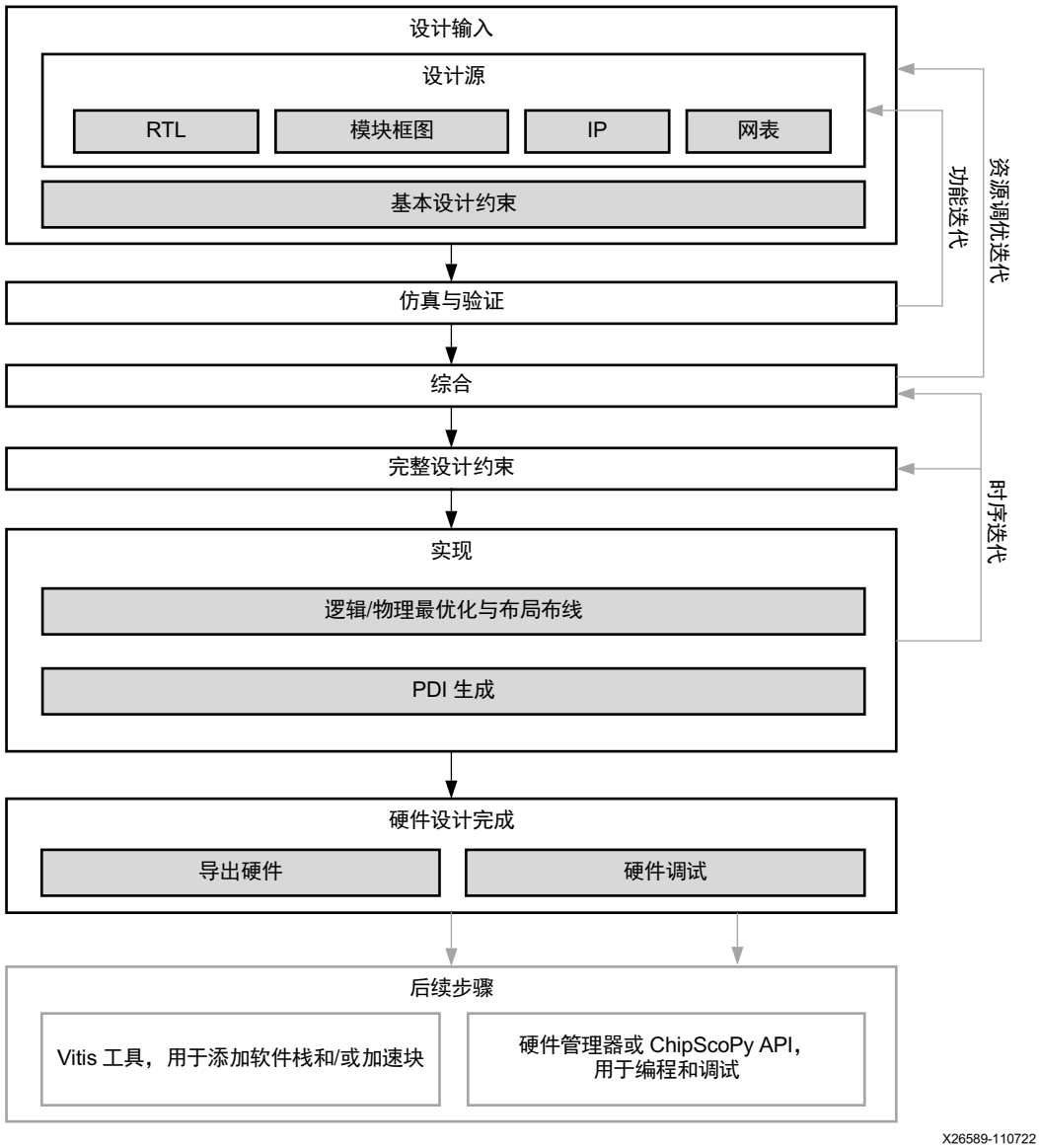
注释：如需了解有关设计类型的更多信息，请参阅《Versal 自适应 SoC 系统和解决方案规划方法指南》(UG1504)。

设计流程图示

下图展示了 Versal 器件设计流程的高层次汇总信息。各设计步骤因设计流程和设计类型而异，如下所示：

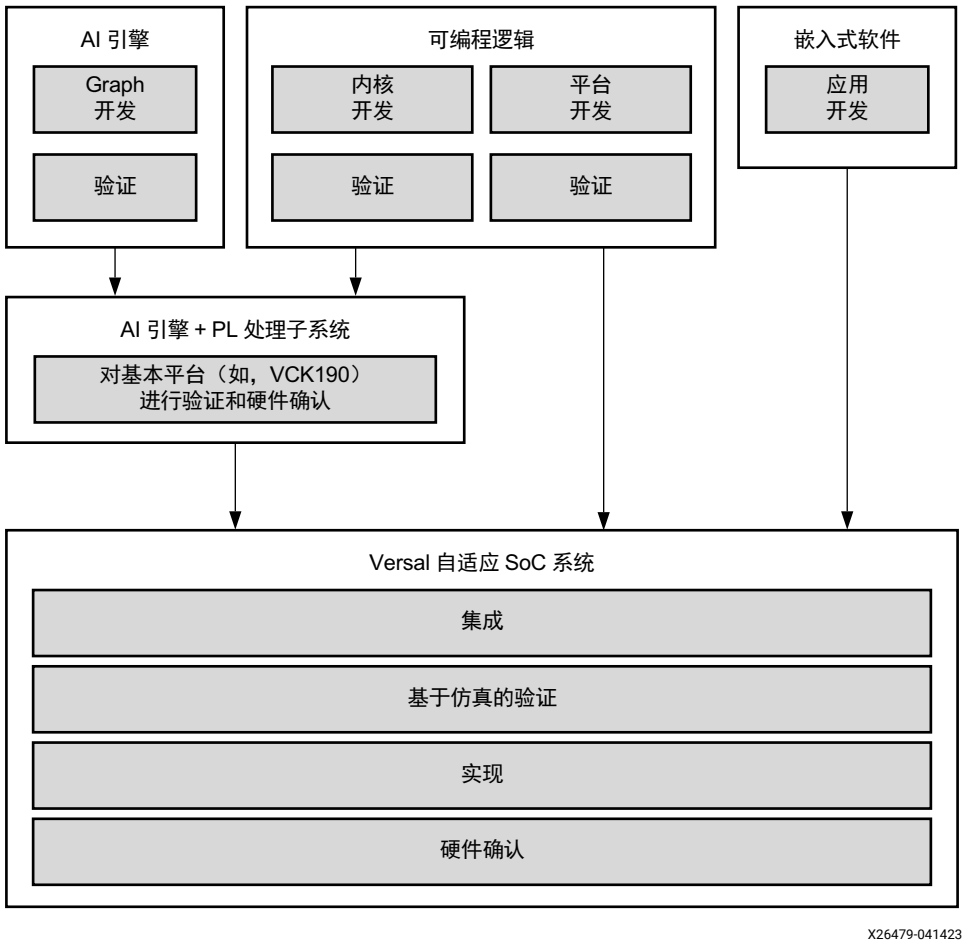
- 面向仅限硬件的系统的传统设计流程：使用传统设计流程。实现后，继续执行“硬件调试”步骤。
- 面向嵌入式系统的传统设计流程：使用传统设计流程。实现后，继续执行“导出硬件”步骤，以在 Vitis 环境中添加软件栈。
- 面向嵌入式系统的基于平台的设计流程：从传统设计流程开始操作。实现后，继续执行“导出硬件”步骤，以将平台导出至 Vitis 环境。继续执行基于平台的设计流程。在 Vitis 环境中，添加 PL 加速器和软件栈以完成设计。
- 面向嵌入式 AI 引擎系统的基于平台的设计流程：从传统设计流程开始操作。实现后，继续执行“导出硬件”步骤，以将平台导出至 Vitis 环境。继续执行基于平台的设计流程。在 Vitis 环境中，添加 PL 加速器、AI 引擎加速器和软件栈以完成设计。

图 2：**Versal 器件的传统设计流程**



下图显示了如何通过 Vitis 环境为 Versal 器件完成基于平台的设计流程。

图 3：**Versal 器件的基于平台的设计流程**



传统设计流程

面向仅限硬件系统的传统设计流程

如果您的设计仅包含 PL 组件（仅含 RTL 和 IP），那么您可使用 AMD Vivado™ 工具来生成可编程器件镜像 (PDI)，以便用于对 Versal 器件进行编程。与先前架构类似，设计源被添加到 Vivado 工具中，并通过 Vivado 实现流程来进行编译。

重要提示！平台管理控制器 (PMC) 整合到 CIPS IP 中，必须对其加以配置才能使 Versal 器件正确启动。因此，所有 Versal 器件设计必须包含 CIPS IP。

以下另提供了其他重要注意事项：

- 硬化的 DDR 存储器控制器和 HBM 控制器只能通过 NoC IP 来访问。要使用 DDR 存储器控制器或 HBM 控制器，您的设计必须包含 NoC IP。
- 硬件调试核默认情况下通过 CIPS IP 来连接。JTAG 仍可用，但不再作为首选流程。您必须熟悉硬件调试连接和流程方面的更改。

您必须使用 Vivado IP integrator 来例化、配置和连接 CIPS IP、NoC/DDR 存储器控制器 IP 以及硬件调试 IP，才能在设计变更迭代过程中充分利用块设计自动化。Vivado IP integrator 还可为 GT IP 和连接 IP（如 MRMAC IP）提供特殊支持，从而简化基于 GT 的设计创建和 I/O 管脚分配。

您可使用定制封装 IP、RTL 模块参考的块以及 IP 目录提供的其他 IP，将完整设计与 Vivado IP integrator 集成。或者，您可使用 Vivado IP integrator 来配置并连接关键 Versal 自适应 SoC IP（例如，CIPS IP 和 NoC/DDR IP），然后在 RTL 设计中例化生成的块设计。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994) 中的相应内容。

注释：仅在工程模式下支持 Vivado IP integrator。

重要提示！此设计流程不支持对 AI 引擎核进行编程，因此仅适用于 Versal Prime 器件、Versal Premium 器件和 Versal HBM 器件。

面向嵌入式系统的传统设计流程

您也可以使用传统设计流程来创建同时包含 PL 组件和嵌入式软件组件的设计。在此情况下，流程与用于 AMD Zynq™ UltraScale+™ MPSoC 的嵌入式软件设计流程相似。硬件团队负责创建、验证和实现硬件设计，以供软件团队用于开发嵌入式软件应用。

注释：适用于面向仅限硬件系统的传统设计流程的所有建议都同样适用于面向嵌入式系统的传统设计流程。

以下是此流程中的主要步骤：

1. 使用 Vivado IP integrator 创建并验证硬件设计。
2. 使用 Vivado 实现工具来实现硬件设计。
3. 将硬件设计导出至 Vitis 嵌入式软件开发流程。
4. 在固定硬件设计上使用 Vitis 嵌入式软件开发流程开发软件应用。

注释：仅在工程模式下支持 Vivado IP integrator。

重要提示！此设计流程不支持对 AI 引擎核进行编程，因此仅适用于不含 AI 引擎的 Versal Prime 器件、Versal Premium 器件和 Versal HBM 器件。

基于平台的设计流程

在基于平台的设计流程中，硬件设计按概念分为 2 个不同要素：平台和处理器系统。平台包含基本 Versal IP 块（包括 CIPS、NoC、AI 引擎和 Clocking Wizard）和开发板接口 IP 块（包括高速 I/O 和存储器控制器）。处理器系统包含特定于应用的系统部分，这部分由可编程逻辑与 AI 引擎块组成。此平台为可扩展平台，因为它不含可编程逻辑的全部内容。而是改为通过添加处理器系统来扩展此平台。

下面是此流程中的主要步骤。前 3 个步骤可以并行完成。您可在最终完成固定硬件平台后单独更新 AI 引擎程序。

1. 使用 Vivado IP integrator 和 RTL 代码开发硬件平台。
2. 使用 Vitis 工具开发 AI 引擎计算图与内核。

   注释：仅当使用 Versal AI Core 系列和 AI Edge 系列或带有 AI 引擎的 Versal Premium 时，AI 引擎才可用。
3. 使用 Vitis 工具（C++ 内核）或 Vivado 工具（RTL 内核）开发 PL 内核。
4. 汇编 AI 引擎程序和 PL 内核以构成处理器系统，并使用 Vitis 连接器将处理器系统与平台相集成以创建固定硬件设计。
5. 使用 Vivado 工具在固定硬件设计上实现和执行设计收敛。

6. 在固定硬件设计上使用 Vitis 嵌入式软件开发流程开发软件应用。

注释：仅在工程模式下支持 Vivado IP integrator。

重要提示！这是支持对 AI 引擎核进行编程的唯一流程，因此对于 Versal AI Core 器件、AI Edge 器件或 Premium 器件而言，此流程是必需的。

提示：AMD 为 Versal 自适应 SoC 评估套件（如 VCK190）提供现成的平台。

基于平台的设计流程最佳实践

AMD 建议使平台部分在设计中所占比重保持尽可能小。例如，将平台中的 RTL 限制为仅限 I/O，并将封装功能 RTL 限制为内核。最大程度减少平台中的逻辑即可减少完成设计所需的平台迭代总数。

总之，AMD 建议将计算逻辑块或算法逻辑块作为内核来处理，并将以下块保留在平台内：

- AI 引擎
- NoC
- CIPS
- I/O 块（外部管脚、MIPI、PHY 等）和相关 IP（DMA for PCIe®、MAC for Ethernet 等）

下表显示了每一种逻辑类型（在平台中或在内核中）的建议布局。

表 3：**平台分区最佳实践**

逻辑	平台	内核
AI 引擎	仅限在平台内使用	不支持
NoC	仅限在平台内使用	不支持
硬核处理器（PS8 和 CIPS）	仅限在平台内使用	不支持
软核处理器（MicroBlaze™ 处理器）	首选在平台内使用	可接受作为内核
I/O 块（外部管脚、MIPI、PHY 等）	仅限在平台内使用	不支持
需要 Linux 驱动程序和软件栈的 IP（VPSS、Ethernet MAC、DMA for PCIe 等）	仅限在平台内使用	不支持
含 AXI 接口的 HLS IP	可接受在平台内使用	首选作为内核使用
含 AXI 接口的 RTL IP	可接受在平台内使用	首选作为内核使用
含非 AXI 接口的 IP	首选在平台内使用	可接受作为内核 **注释：**欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。
Vitis 库	可接受在平台内使用	首选作为内核使用

了解 Versal 自适应 SoC 设计方法论的概念

从设计之初即采用正确方法，从早期阶段开始对设计目标（包括 IP 选择和配置、块连接、RTL、时钟、I/O 接口和 PCB 管脚分配）给予足够的重视，这些对于确保设计成功都至关重要。在每个设计阶段中务必正确定义和确认设计，这有助于缓解在子系统和完全集成的系统的实现阶段中出现的时序收敛、性能收敛和功耗使用问题。

创建和实现硬件设计

完成器件 I/O 管脚分配、PCB 布局规划并决定使用模型后，即可开始创建设计。设计创建包括：

- 规划设计的层级
- 识别要在设计中使用和定制的 IP 核
- 例化 IP 目录中不可用的特殊互连或功能所需的 RTL 模块
- 创建时序约束、功耗约束和物理约束
- 指定综合与实现阶段所使用的其他约束、属性及其他元件

创建设计时，主要的考虑要素包括：

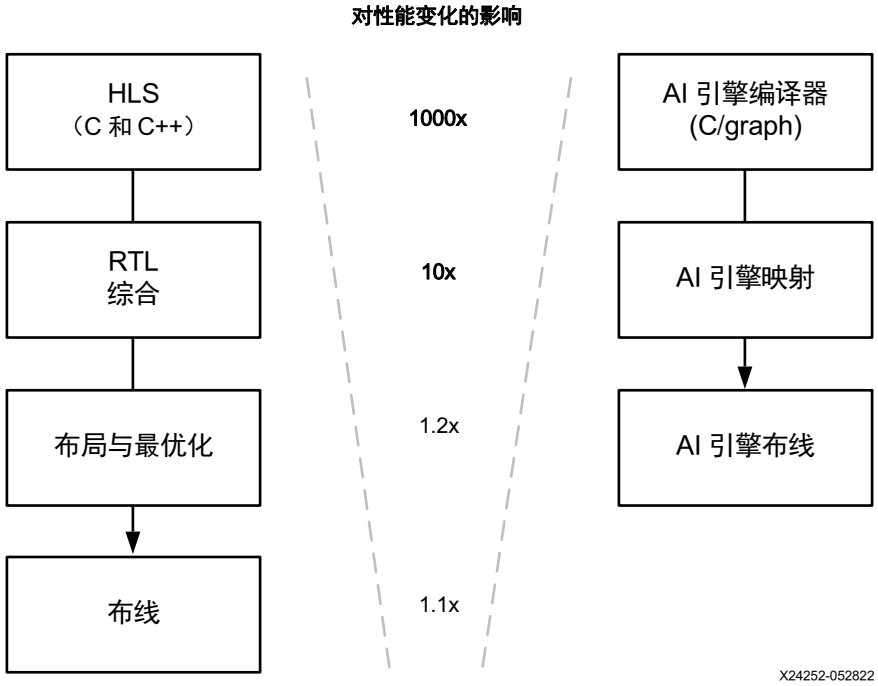
- 实现所需的功能
- 按期望的频率运行
- 按期望的可靠程度运行
- 符合硅片资源和功耗预算要求

在此阶段做出的决策将影响最终产品。在这一阶段的错误决策会导致后续阶段问题层出不穷，进而造成整个设计周期中不断返工。在此过程中尽早花时间详细规划设计有助于达成设计目标并最大限度缩短实验室中的调试时间。

在开发周期早期最大限度扩大影响

如下图所示，设计流程的早期阶段（C、C++ 和 RTL 综合）对于设计性能、密度和功耗的影响远超后期实现阶段的影响。因此，如果设计不满足时序、时延或功耗目标，AMD 建议您重新评估综合阶段（包括 C、C++、HDL 和约束），而不是仅在实现阶段通过迭代来寻找解决方案。

图 4：**整个流程中设计变更的影响**



在每个设计阶段进行确认

Versal 自适应 SoC 设计方法强调对设计预算（例如，面积、功耗、时延和时序）进行监控以及尽早采取如下措施更正设计的重要性：

- 尽可能多加利用 Versal 自适应 SoC 集成块，使用片上网络 (NoC) 实现高带宽连接并在模块框图级别确认设计性能。

  由于围绕块之间的器件进行高效数据移动至关重要，因此必须通过 NoC 或可编程逻辑 (PL) 来探索各种块连接选项的作用。尽可能利用 NoC 即可释放 PL 资源，并降低后续布局规划或实现难度。

- 利用 AMD 模板创建最佳 RTL 结构，并在执行细化后进行综合前采用方法 DRC 来确认 RTL。

  由于 Vivado 工具从始至终使用时序驱动的算法，设计必须从设计流程开始就加以正确约束。

- 在综合后开展时序分析。

  要指定正确的时序，您必须分析设计中每个主时钟与相关的生成时钟之间的关系。在 Vivado 工具中，每次时钟交互都必须满足时序要求，除非显式声明为异步时钟交互或伪路径 (false path)。

- 通过运行非关联综合和实现来确认每个主要 PL IP 或模块框图的时序收敛可行性。

  如果待到分析完整设计时才尝试通过更改设计或流程选项来解决时序、性能或功耗问题，则问题复杂性会明显提升。通过确认设计的每个小部分，可以降低设计周期后续的收敛风险。AMD 建议在非关联实现期间对设计时钟进行过约束（不超过 10%），如果可能，还可添加 Pblock 以建立高使用率场景模型。

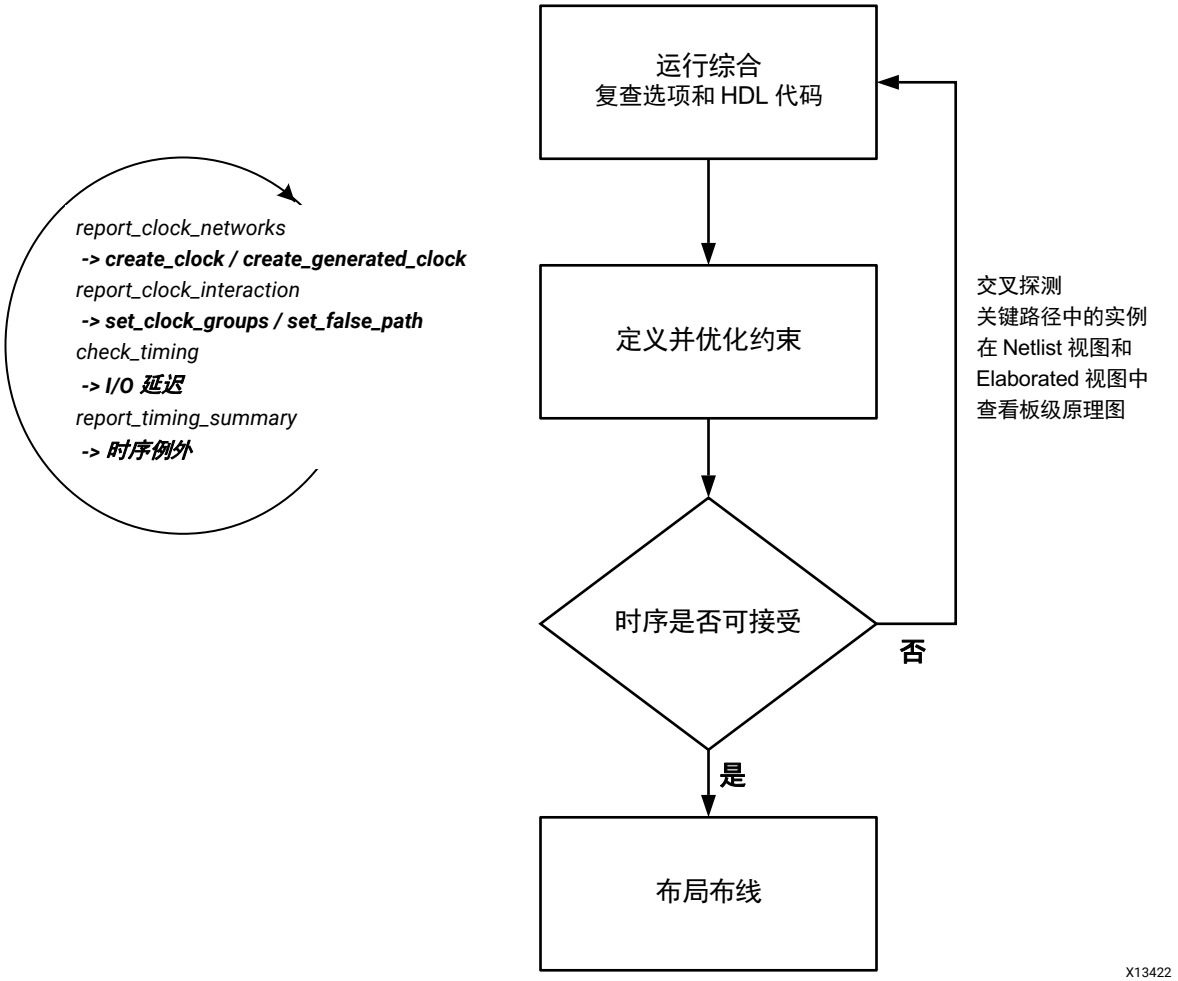
- 在继续执行下一个设计阶段前采用正确的约束满足时序要求。

  您可遵循如下建议并使用 Vivado Design Suite 的交互式分析环境来加速总体时序与实现收敛。

> **提示：**您还可通过结合上述方法以及本指南中的 HDL 设计指南进一步加速收敛过程。

下图展示了这一推荐的设计方法。

图 5：**实现快速收敛的 RTL 设计方法**



如能够通过正时序裕度 (positive margin) 或相对较小的负时序裕度 (negative timing margin) 满足设计目标，那么综合即可视为完成。例如，如果综合后未能满足时序要求，那么布局布线结果也不太可能满足时序要求。然而，即便时序得不到满足，您仍然可以继续开展流程其余部分。如果实现工具能为失效的路径分配最佳资源，则可能能够收敛时序。此外，继续执行此流程可以更准确理解负时序裕量的量级，这有助于您确定综合后最差负时序裕量 (WNS) 所需的提升程度。改进 HDL 和约束后返回综合阶段时即可利用此信息。

使用 Versal 自适应 SoC 设计方法论 DRC

Vivado Design Suite 包含一组方法论相关 DRC，可供您使用 `report_methodology` Tcl 命令来运行。此命令针对以下每个设计阶段都具有相应的规则：

- 在综合前，在细化 RTL 设计中用于确认 RTL 结构
- 在综合后，用于确认网表和约束
- 在实现后，用于确认约束和时序相关问题。

建议：为了最大限度发挥作用，请在每个设计阶段运行方法论 DRC，并解决其中的严重警告 (Critical Warnings) 和警告 (Warnings)，然后再继续执行下一个阶段。

重要提示！Vivado IP integrator 当前不提供方法论检查。您必须改为使用 `validate_bd_design` 命令来提前识别连接和 IP 配置问题。请谨记，在后续综合与实现期间发现的方法违例（如不建议采用的时钟约束或时钟拓扑）可能需要在 Vivado IP integrator 块设计定义中方可解决。

如需了解有关设计方法论 DRC 的更多信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835) 中的 report_methodology Tcl 命令。

使用 Vivado Design Suite

Vivado Design Suite 具有灵活的使用模型，可适应各种开发流程和不同类型的设计。如需了解有关如何使用 Vivado Design Suite 中的各项功能的详细信息，请参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892) 和其他 Vivado Design Suite 文档。

采用版本控制系统管理 Vivado Design Suite 源

大部分设计团队都采用商用的版本控制系统来管理自己的设计源与设计结果。Vivado Design Suite 支持通过各种使用模型来管理设计和 IP 数据。如需了解有关将 Vivado 工具与版本控制系统配合使用的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892) 中的相应内容。

升级到新发行版的 Vivado Design Suite

新发行版的 Vivado Design Suite 通常包含 AMD IP 更新。请仔细考量您是否要升级自己的 IP，因为升级可能导致设计变更。此外，升级后如需使用通过先前发行版配置的 IP，必须遵循具体规则进行操作。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896) 中的相应内容。

使用 Vitis 环境

AMD Vitis™ 环境将 AMD 软件开发的方方面面都整合到统一的单一环境内。Vitis 环境支持嵌入式软件开发流程和加速开发流程。此处内容主要与应用加速流程以及 Vitis 核开发套件和 Xilinx Runtime (XRT) 的使用有关。如需了解有关 Vitis 环境的更多信息，请参阅《Vitis 统一软件平台文档》。

关于本指南

本指南包含对应如下主题的高层次信息、设计指南和设计决策利弊取舍：

- 第 2 章：设计规划：提供有关设计规划的信息，以帮助您最大程度利用 Versal 架构，包括有关关键 IP 块、DFX 设计以及不同设计流程的规划信息。
- 第 3 章：使用块设计来创建设计：提供有关创建块设计的指南，涉及 GT、CIPS、NoC 和其他 IP 以及定制封装 IP 和 RTL。

- 第 4 章：使用 RTL 创建设计：提供用于创建 RTL 模块的最佳实践，满足 IP 目录中无法实现的更高性能或特殊功能的需求。
- 第 5 章：使用 Vitis HLS 创建设计：提供有关利用 Vitis HLS 创建设计的概述以及方法论建议，包括定义接口和使用 Vitis 内核流程。
- 第 6 章：I/O 管脚分配设计流程：提供有关不同 I/O 管脚分配流程的信息，并提供相关建议，用于为特定管脚分配信号以精简流经器件的数据流，并满足高性能需求。
- 第 9 章：设计约束：提供相关建议，用于创建适当的时序、功耗和物理约束，以及用于指定综合与实现阶段所使用的其他约束、属性及其他元件。
- 第 10 章：设计实现：提供设计综合与实现相关的最佳实践。
- 第 11 章：Vitis 环境嵌入式平台创建方法：提供有关创建嵌入式平台（包括将功能映射至平台和子系统）的高层次信息。

第 2 章

设计规划

正确规划所有 AMD Versal™ 自适应系统级芯片 (SoC) 设计至关重要。Versal 器件具有多个独特 IP，这些 IP 可能对您将自己的设计映射到 Versal 架构以及您构建自己的设计时所选的设计流程具有重大影响。创建传统设计或 AMD Vitis™ 平台时，Control, Interface, and Processing System (CIPS) IP 必须整合到您的设计中才能启动 SoC。此外，创建 Vitis 平台时，您必须决定设计哪部分驻留在平台中、规划如何在设计中移动数据，并选择哪些平台接口可从设计的用户部分进行访问。最后，如果您计划在设计中使用 Dynamic Function eXchange (DFX)，则必须留意边界信号线和布局规划。在设计初提前投入时间用于开展规划有助于您充分利用 Versal 架构。

面向关键 IP 块的设计规划

Versal 自适应 SoC 包含多个不可或缺的硬核 IP。在审慎规划设计的过程中，您必须根据自己的设计对这些 IP 进行相应的分析和配置。

- CIPS IP：CIPS IP 包含 Versal 架构的多个关键组件，包括平台管理控制器 (PMC)、处理器子系统 (PS) 以及加速器缓存一致性互连 (CCIX) PCIe® 模块 (CPM)。PMC 负责管理 Versal 器件的编程和启动、监控系统并保护器件抵御有害攻击。由于器件编程和启动都需要 PMC，因此在每个 Versal 自适应 SoC 设计中都必须包含 CIPS IP。此外，CIPS IP 只能从 AMD Vivado™ IP integrator 来访问。因此，所有 Versal 自适应 SoC 设计都有至少一部分设计是使用 IP integrator 创建的，而 CIPS IP 就包含在这部分设计中。
- NoC IP：NoC 属于高带宽硬化互连，用于为 Versal 架构中的所有数据移动提供主干。您可使用标准 AXI 存储器映射接口或 AXI 串流接口来与 NoC IP 进行交互。NoC 编译器可汇总请求的带宽以及所有流量的相关优先级，并相应分配物理布线。NoC 是访问 Versal 自适应 SoC 硬化的存储器控制器的唯一途径。此外，NoC 端口在 CIPS 上、在整个 AI 引擎阵列中以及在可编程逻辑 (PL) 互连结构中均可用。
- GT IP：在 Versal 器件中，千兆位收发器 (GT) 按四通道 (quad) 来进行分组。这样可以支持 GT 共享时钟和复位，从而减少开销。通过分组，即可将 GT 与其父 IP（例如，MRMAC Ethernet IP）分离。在 IP integrator 画布上布局父 IP 时，块自动化设置能够处理将父 IP 连接到 GT 四通道的操作。GT 的管脚分配被整合到 Vivado Hard Block Planner（硬核块分配器）中，不包含在 IP 生成内。对于第三方 IP，可使用 Bridge IP 来简化与 GT 四通道的连接。

传统设计流程的设计规划注意事项

使用 Versal 自适应 SoC 传统设计流程时，您必须在进行设计规划时考量下列要点：

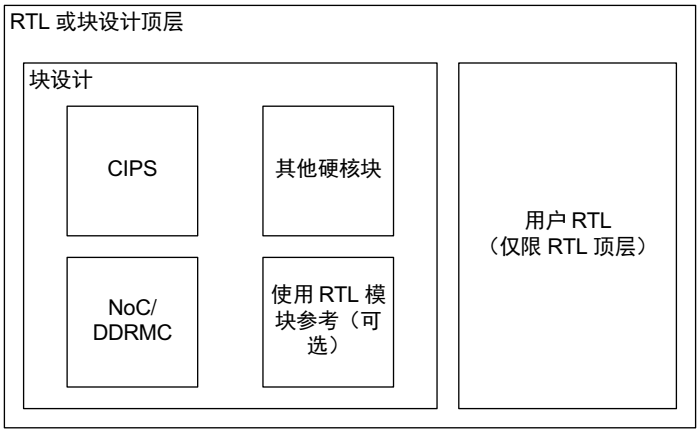
- CIPS IP：Vivado 工具会生成最终可编程器件镜像 (PDI)。为确保正确生成 PDI，必须在设计中布局 CIPS IP，即使不使用处理器系统也是如此。在启动 Versal 器件时需使用 PMC，它布局在 CIPS 内。
- 设计层级和 NoC 编译器：在 Versal 自适应 SoC 设计中可以例化一个或多个 NoC IP，前提是这些 IP 均位于单一 BD 层级下。这样可确保在确认最顶层的 BD 时，可自动调用 NoC 编译器并且可完整查看设计中的所有 NoC 主单元和从单元，包括其连接、带宽要求和相关优先级。

  注释：如果设计使用 BD 顶层，那么 NoC 编译器始终可查看该设计中例化的所有 NoC IP 核。

- 仿真：仿真需要创建到 NoC 的特殊连接才能对其连接进行正确建模。导出设计用于仿真时，会在设计层级中添加额外的层次，以表示 NoC 连接。您并不知晓此进程存在，但它是保证 NoC 正确执行仿真所必需的进程。

下图展示了传统设计流程示例。

图 6：**传统设计流程**



基于平台的设计流程的设计规划注意事项

使用基于 Versal 自适应 SoC 平台的设计流程时，您必须在硬件平台与子系统之间进行设计分区，此处的硬件平台是在 Vivado 工具中创建的，而子系统则是由 AMD Vitis™ 环境创建并整合到该平台中的。要成功使用此设计流程，您必须正确识别要包含在平台中的各设计部分，以及要通过 Vitis 环境来整合的其他各部分。总之，公用共享资源应包含在平台内，设计的专用组件则可能使用 Vitis 环境来加以整合。

平台内包含的 IP 常见示例如下：

- CIPS
- NoC 互连
- 时钟设置
- 复位
- 中断控制器

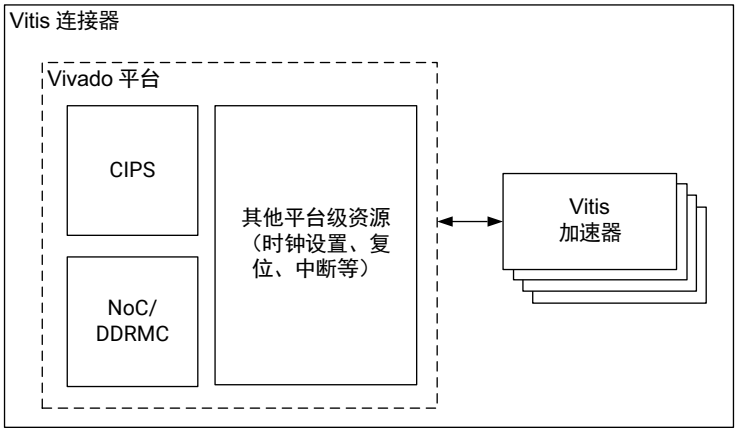
Vitis 环境可接受多种类型的设计源文件，包括：

- 封装的 BD
- 封装的 RTL
- HLS C 语言代码
- AI 引擎内核代码

注释： AI 引擎内核之间的连接以及 AI 引擎内核与平台之间的连接通过 graph 图文件来定义。

下图展示了基于平台的设计流程示例。

图 7：**基于平台的设计流程**



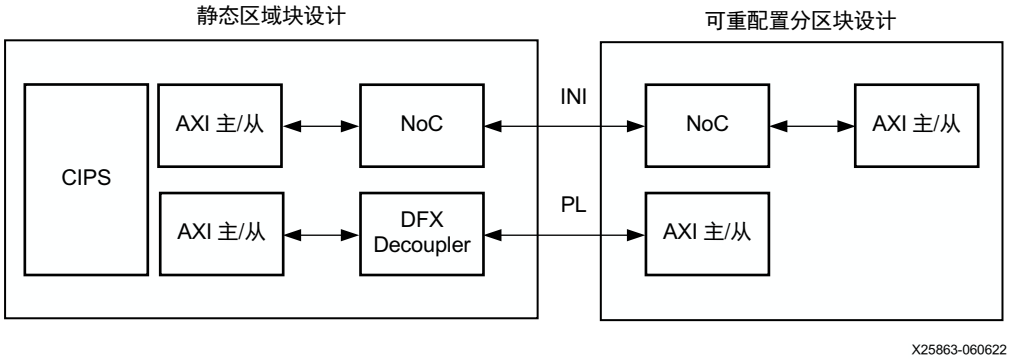
Dynamic Function eXchange 的设计规划注意事项

对于传统流程和基于平台的流程，Dynamic Function eXchange (DFX) 有特殊的规划注意事项要求。在 Versal 自适应 SoC 中，访问 DFX 的建议方法是使用 IP integrator 画布上的块设计容器 (BDC)。BDC 具有一个属性用于指示该 BDC 是否适用于 DFX。如果设置该属性，那么此 BDC 会变为可重配置分区 (RP)。可重配置模块 (RM) 可添加到 RP 中，方法是将额外的块设计 (BD) 与 BDC 相关联。每个 BD 都表示一个 RM。就像所有 DFX 设计一样，判定逻辑分区边界和正确的设计层级至关重要。正确完成 BDC 配置后，Vivado 工具会创建父实现运行和子实现运行以完成编译，并生成编程文件。

当 NoC 互连跨接任一分区时，Versal 自适应 SoC 设计具有如下特殊的注意事项要求。为处理此类情况，NoC IP 必须布局在设计的静态部分，另一个 NoC IP 实例则必须布局在该分区内。必须使用名为 NoC 间互连 (INI) 的虚拟互连来连接 2 个 NoC IP。这样可以确保所有 RM 都包含一组公用的物理接口，但允许 RM 采用不同的地址。或者，您可在静态区域内将 DFX Decoupler IP 用于 RM 与该静态区域之间的基于 PL 的交汇。但 AMD 建议在静态到 RM 边界处使用更省资源的基于 INI 的交汇，因为 NoC 在内部处理 NoC 路径的静止和关闭状态，无需手动干预去耦。但部分重配置有一处关键细节需要您注意，如果静态域中的传输事务向正在重配置的动态区域发出信息请求，那么仍由您负责暂停这些传输事务。

下图显示了 DFX 设计的设计层级，其中静态 RM 接口以基于 PL 的去耦器或基于 NoC 的 INI 为基础。

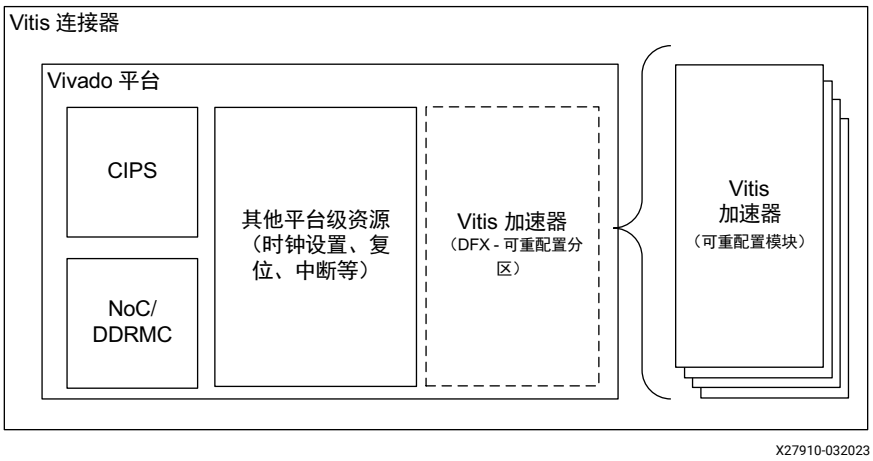
图 8：**DFX 设计层级**



DFX 布局规划必须根据 Versal 自适应 SoC 的独特架构来设计，包括 NoC 资源、硬化的 IP 位置以及时钟设置资源。如需获取更多信息，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909) 和 GitHub 仓库中提供的 DFX 教程。如需获取创建基于 BDC 的 DFX 设计示例，请访问此链接以参阅《Vivado Design Suite 教程：Dynamic Function eXchange》(UG947) 中的相应内容。

下图展示了使用 DFX 的基于平台的设计流程示例。

图 9：**使用 DFX 的基于平台的设计流程**



相关信息

面向平台和 Dynamic Function eXchange 的时钟设置建议
Dynamic Function eXchange 的布局规划约束

基于 DFX 的 Vitis 加速平台开发的设计规划注意事项

您可在 Vivado 工具中使用基于 Dynamic Function eXchange (DFX) 的设计流程来创建可扩展赛灵思支持存档 (XSA) 文件，以便在 Vitis 软件平台内用于创建加速应用。在加速平台内的硬件平台开发进程中，请使用以下步骤。软件开发者将 XSA 文件导入 Vivado 工具以创建可扩展 XSA 文件。

1. Vivado IP integrator 块设计容器流程

   使用 Vivado IP integrator 块设计容器 (BDC) 流程创建 DFX 分区。BDC 功能特性可将 IP integrator 画布中的分层块转化为块设计 (BD)。您可在此新 BD 中启用 DFX，随后即可在 Vitis 环境内将其用于链接加速逻辑。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994) 中的相应内容。

2. 平台设置

   IP integrator 的“Platform Setup”（平台设置）窗口可用于选择平台块设计中的不同接口类型，包括 AXI 端口、AXI Stream 端口、时钟、中断和存储器。您还可在“Platform Setup”窗口中分配平台名称、版本、供应商和开发板信息。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994) 中的相应内容。

   如需了解有关 DFX 和非 DFX 嵌入式平台创建的更多信息，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

3. DFX Wizard

为设计源生成目标后，就会在 Flow Navigator 窗口中显示 DFX Wizard。您可定义 DFX 配置，并将 DFX 配置与实现运行相关联。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)。

4. 布局规划

   Pblock 约束可用于将 DFX 分区分配到器件中的物理区域，此区域可供 Vitis 环境用于加速应用实现。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)。

5. 用于创建启动镜像的最小实现

   在可重配置分区 BDC 中请保留最少量的逻辑，以便您用于在 Vivado 工具中创建平台期间执行实现。这样有助于减小可用作为平台的启动镜像的器件镜像大小。虽然此最小配置可能是上电时交付至器件的首个镜像，但在 Vivado 中编译设计时，不应将其作为父配置。而应采用给定可重配置分区的所有可重配置模块的最大、最复杂和/或要求最严苛的接口的配置作为父配置。

6. 硬件导出

   完成初始实现后，可使用 `write_hw_platform` 命令导出可扩展 XSA 文件。XSA 文件包含以下内容：

   - 用于平台静态镜像的 DCP
   - 动态区域块设计，供 Vitis 环境用于链接至加速软件应用
   - 来自初始实现的器件镜像
   - 成功完成从 Vivado 到 Vitis 工具的硬件设计交接所需的其他元数据

要在现有平台内添加新的 PFM 属性而不更改初始实现和器件镜像（以避免更新启动镜像更新），请使用以下步骤。

1. 打开先前用于创建原始平台的已存档的工程。
2. 修改与可重配置分区 (RP) 关联的 BD，并保存 BD。
3. 确保初始实现结果仍更新至最新状态。

   注释：仅限已修改 RP 的 BD 以非关联 (OOC) 模式运行时才会因修改而过期。

4. 使用 `write_hw_platform` 命令导出修改后的平台。

修改平台时，请注意：

- 执行修改进程后，与静态区域关联的任何综合与实现都必须保持更新至最新状态。重新实现静态区域会导致用于实现静态区域的器件镜像发生更改。
- 切勿更改静态 RP 接口。更改静态 RP 接口会导致初始实现过期，从而需要重新实现静态区域（启动镜像）。

串联配置的设计规划注意事项

串联配置属于分阶段配置，在 Versal 自适应 SoC 供电稳定后的 100 ms 内完成 PCIe 协议的初始化。这是使用阶段 1 可编程器件镜像 (PDI) 来达成的。器件其余部分（包括 PL）可由用户应用作为阶段 2 PDI 来完成下载。在 Versal 自适应 SoC 中，串联配置是使用 CPM 集成块来完成的。串联配置支持 Tandem PCIe（串联 PCIe）模式和 Tandem PROM（串联 PROM）模式。如需了解更多信息，请参阅《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347)。

不同于先前架构，阶段 1 配置元件是 CPM 中的硬化的块。CIPS IP 的 CPM 部分包含启用串联配置的选项。您可选择“Tandem PROM”或“Tandem PCIe”。可编程器件镜像 (PDI) 的构造中包含两个分区：stage1 和 stage2。设计的 stage1 部分包含 PMC 和 CPM 的配置，设计的 stage2 部分则包含所有其他配置。如果选择“Tandem PROM”（串联 PROM）模式，那么 stage1 和 stage2 部分在单个 PDI 文件内生成，并通过配置闪存接口自动加载。如果选择“Tandem PCIe”（串联 PCIe）模式，那么 stage1 PDI 文件通过配置闪存来加载，stage2 PDI 则通过 PCIe 链接来加载。“Tandem PCIe”解决方案和“DFX over PCIe”解决方案都需要额外的设计连接和主机交互才能完成器件配置。

适用于 AI 引擎核可编程逻辑集成的设计规划注意事项

可通过使用 Vivado 工具开发硬件与使用 Vitis 工具开发软件相结合的方式来完成 Versal 自适应 SoC 设计的开发。有多种设计方法论可用于设计开发，应根据设计流程的具体需求来选择方法论。这些方法论能显著提升设计开发流程的效率。AMD 建议在 Vivado 中开发硬件设计。如有硬件平台可用，即可在 Vitis 中使用它来开发软件组件。您可使用 Vitis 集成流程或使用 Vitis 导出到 Vivado 流程来开发 Versal 自适应 SoC 设计。

如果您已完成 AI 引擎和 HLS 软件组件开发，并且想仅在 Vivado 内继续进行硬件开发，那么 AMD 建议使用 Vitis 导出到 Vivado 流程。此流程支持您在 Vivado 内完成硬件设计开发与验证。

1. Vitis 导出到 Vivado 流程的步骤如下：
   a. 使用 Vivado 创建硬件设计。
   b. 从 Vivado 导出到可扩展的 `xsa`。
   c. 在 Vitis 内编译内核（AI 引擎/HLS），以生成 `libadf.a` 和 `.xo`。
   d. 通过将编译输出、`extensible.xsa` 和 `config` 文件与 `--export_archive` 开关相链接来生成 `.vma` 文件。`v++ --link` 并不会运行综合与实现。
   e. 将 `.vma` 导入 Vivado。
   f. 在 Vivado 中执行设计修改。
   g. 运行综合与实现。
   h. 使用 `write_hw_platform -fixed` 生成 `fixed.xsa`。
   i. 使用 `v++ --package`，从 Vivado 生成的 `fixed.xsa` 生成 `xclbin`。
   j. 将设计导入硬件。

   如需了解有关 Vitis 导出到 Vivado 流程的更多信息，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。
2. 建议对 Vivado 内完成构建的硬件设计平台采用 Vitis 集成流程。此流程支持您使用平台和接入应用的 AI 引擎/HLS 软件组件。通常，硬件开发完成并且需要添加或修改软件组件时，适用此流程。此流程步骤如下：
   a. 使用 Vivado 创建硬件设计。
   b. 从 Vivado 导出到可扩展的 `xsa`。
   c. 在 Vitis 内编译内核（AI 引擎/HLS），以生成 `libadf.a` 和 `.xo`。
   d. 通过将编译输出、`extensible.xsa` 和 `config` 文件相链接来生成 `fixed.xsa`。综合与实现在此阶段中运行。固定 `xsa` 是由 Vitis 在结束所有设计运行后生成的。
   e. 使用 `v++ --package`，从 Vitis 生成的 `fixed.xsa` 生成 `xclbin`。
   f. 将设计导入硬件。

如需了解有关报告生成的更多信息，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

3. 对于不含平台的 Vitis 设计开发，建议使用此流程来处理包含 AMD 尚未发布的评估板或平台的设计。您可通过使用此流程来开始为应用的 AI 引擎或 HLS 组件进行软件开发。此流程不需要使用 Vivado 生成的 `xsa` 来进行软件开发。此流程步骤如下：

   a. 利用 `--part` 编译 AI 引擎。

   b. 使用 `v++ --link --part ...` 生成 XSA。

   c. 利用生成的 XSA 编译 HLS。

   d. 通过将编译输出、`extensible.xsa` 和 `config` 文件相链接来生成 `fixed.xsa`。综合与实现在此阶段中运行。固定 `xsa` 是由 Vitis 在结束所有设计运行后生成的。

   e. 使用 `v++ --package`，从 Vitis 生成的 `fixed.xsa` 生成 `xclbin`。

   f. 将设计导入硬件。

   如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393)。

第 3 章

使用块设计来创建设计

AMD Vivado™ IP integrator 支持您通过例化并连接各种来源的 IP 块来创建复杂的系统设计。您可在画布 GUI 上以交互方式创建设计，或者也可以通过 Tcl 编程界面以编程方式创建设计。在 IP integrator 中创建设计的主要优势如下所述：

- 在接口级别完成设计构造（效率更高），但也可在端口级别进行操作（确保精准的设计操作）。
- 设计以自动建构校正方式来创建，这表示工具中的自动设计规则检查 (DRC) 能够在设计周期中尽早检测出问题。
- 块自动化配置和连接功能可以节省开发时间。
- 协作功能（例如，块设计容器）支持团队协作设计和保障可复用性。

此外，IP integrator 还可为 AMD Versal™ 自适应 SoC 设计提供下列优势：

- 自动配置和连接 CIPS 和 NoC Versal 器件专用的块。

  注释：您无需在 IP integrator 中创建自己的整个设计。但您必须至少在块设计中创建这部分设计。随后，生成的块设计即可搭配其他 RTL 源一起例化和使用。对于 Dynamic Function eXchange (DFX) 等高级功能，则必须使用 IP integrator。
- 在基于 GT 的 IP 中进行 Versal 器件收发器的配置、共享和集成。
- 为各 Versal 器件域（PL、PS 和 AI 引擎）简化完整的设计集成。
- 与 AMD Vitis™ 工具无缝交互，支持导出定制硬件平台。

以下各部分的最佳实践和信息可帮助您在 Versal 自适应 SoC 设计过程中使用 IP integrator 得到更好的结果。

注释：作为设计创建过程中的一部分，运行综合后，必须在 Vivado Design Suite 中复查并完成设计约束。

重要提示！本节描述了如何使用 AMD Vivado™ IP integrator 以最佳方式创建硬件设计。这是适用于非 AI 引擎工程的主要设计类型。平台设计的方法与此类似，但允许 AMD Vitis™ 连接器将额外的 PL 和 AI 引擎块添加到设计中。平台设计对于基于 AI 引擎的工程而言是必需的，但对于并非基于 AI 引擎的工程而言，则为可选。平台设计的创建还有其他要求，如《Vitis 统一软件平台文档：嵌入式软件开发》(UG1400) 中所述。

相关信息

设计约束

定义理想的块设计层级

Versal 器件的架构与先前器件存在显著差异，因此定义设计层级时有些特别的注意事项需要考量。在设计进程中尽早规划层级有助于尽可能减少后续问题。在 Versal 器件中，软件必须与硬件搭配协同工作。为了确保从 Vivado Design Suite 到 Vitis 环境的无缝硬件交接，定义设计层级时请遵循如下建议：

- 将设计的可寻址部分包含在单个 BD 层级内。

设计的可寻址部分包括 CIPS、NoC、收发器、MicroBlaze™ 处理器以及任何其他可寻址元件。BD 可位于设计层级顶层（搭配 AMD 管理的顶层 RTL 封装文件），或者 BD 也可在定制 RTL 顶层中例化。如果使用本章中包含的任何方法将其他 IP 整合到块设计内，请使用支持硬件交接的方法。例如，使用块设计容器对设计进行区隔化和分区的方法即可搭配硬件交接来使用。但如果使用 RTL 模块参考，则寻址信息将不予保留。在设计进程中尽早考量这些限制至关重要。

- 如果您计划首先启动处理子系统，稍后再加载 PL 编程，那么请在设计顶层中以及块设计容器 (BDC) 的 PL 部分中包含 CIPS 和 NoC。

将块设计与 IP integrator 搭配使用的方法

本节假定您已熟悉 IP integrator 的以下基本功能特性。如需了解有关这些功能的更多信息，请参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994)。

- 使用 IP integrator 画布创建块设计
- 了解块设计的寻址方法
- 在块设计中传输参数
- 调试块设计
- 对块设计使用 Tcl 脚本编制

例化块设计

创建块设计时，BD 文件（.bd 扩展名）会保存到工程源文件中。IP integrator BD 源文件无法在 Vivado 工具中进行直接综合。例化 BD 文件的方法有两种：

- BD 顶层

  BD 文件在顶层 RTL 文件内例化。顶层 RTL 是可自动生成的封装文件，仅包含 BD 的单个实例。此方法为建议采用的方法，但 Versal 自适应 SoC 设计无需使用。

  此方法可确保硬件到系统软件交接能无缝完成，并使用建议的 Versal 自适应 SoC 系统结构。软件能识别设计中的所有可寻址组件，并确保正确创建设备树和驱动程序。

- RTL 顶层

  BD 文件以及定制逻辑和 IP 都在顶层 RTL 文件内例化。在某些情况下，一个或多个 BD 文件也可在子层级中进行例化。

  此方法能够为传统设计移植到 Versal 自适应 SoC 提供最大的灵活性。仅当在顶层 RTL 文件下的层级内例化的单个 BD 中包含完整寻址范围时，才支持在 RTL 顶层中成功完成硬件交接的方法。在任何其他情况下，均由设计师负责创建设备树和加载必要的软件驱动程序。

以下章节将聚焦如何将设计源码整合到模块框图中。

在 IP integrator 中使用不同源文件

IP integrator 使用各种源文件，包括 AMD IP 目录、其他块设计、定制封装 IP 和 RTL 参考模块。源文件必须正确使用，以简化设计结构并提升个人和团队工作效率。下表汇总了与不同源文件相关联的优缺点。

表 4：**IP integrator 中使用的不同源文件对比**

	IP		块设计		RTL
	目录 IP	**定制封装 IP**	**封装的块设计**	**块设计容器**	**RTL 模块参考**
描述	现成可配置 IP	将 HDL 模块转换为可复用的 IP 块	将 BD 转换为可复用的定制封装 IP	允许在 BD 内例化另一个 BD	允许将 HDL 或另一个 BD 直接添加到 BD
优点	IP 块经过测试和验证	· 在多个工程之间共享 IP · 可控制封装自定义，以保障易用性	传统方法，支持复用 BD **注释：**对于新设计，则首选 BDC。	· 例化的 BD 是独立的工程源文件 · 允许从顶层 BD 进行寻址和参数传输	无需 IP 封装即可快速添加 RTL 或嵌套单个 BD
限制	仅限对可用 IP 设置进行自定义，无修改 IP 源文件的选项	无法从顶层 BD 进行查看或修改	封装的 BD 包含固定的寻址信息	尚未支持嵌套 BD **注释：**如需了解更多信息，请参阅答复记录 75853。	RTL 文件中不允许设计检查点或其他嵌套的模块参考
可复用性	高	高	低	高	中

以下章节所述方法论侧重于讲解如何充分利用这些源文件和 IP integrator 功能来改善复杂的 Versal 自适应 SoC 设计的结构和集成。

定制封装 IP

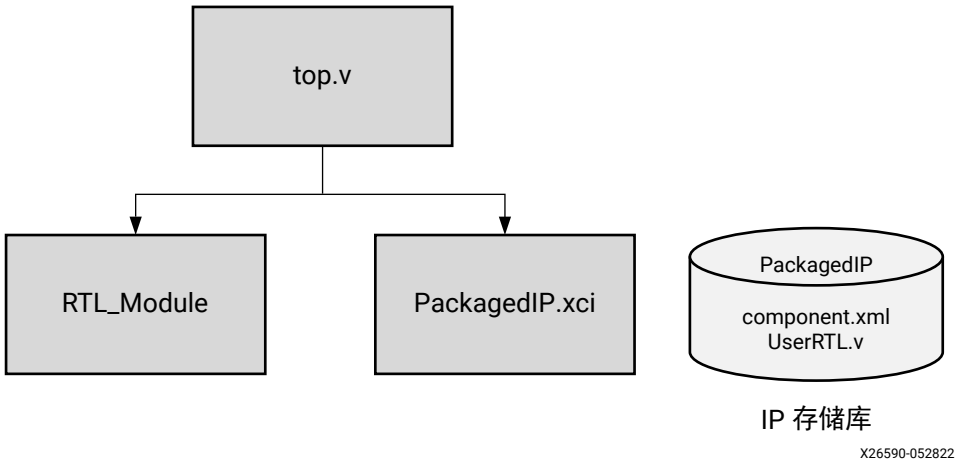
您可使用 Vivado IP 封装器将 HDL 源文件封装到 IP 内，以供块设计使用。封装 IP 可包含 RTL 源文件、块设计或两者混合。封装 IP 支持您控制供应商、库、名称和版本 (VLNV)。但您不能为封装 IP 编辑 RTL 源文件，并且在用于例化封装 IP 的块设计内看不到封装 IP 的内容。为确保定制 IP 的功能正常，AMD 建议您遵循《Vivado Design Suite 用户指南：创建和封装定制 IP》(UG1118) 中所述要求进行操作。

以下是使用定制封装 IP 时值得注意的重要信息：

- 封装输出会损失寻址信息。但您可以在 IP 封装器内自定义封装 IP 的寻址。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：创建和封装定制 IP》(UG1118)。
- SmartConnect 地址信息是固定的，仅封装生成的 HDL。如要利用 SmartConnect 的动态寻址功能，请将此 IP 直接布局在 IP integrator 画布上，或者在块设计容器内例化此 IP。
- IP 封装器输出不提供对参数传输的访问权。但可通过编译指示来为 IP 封装器提供指引。
- IP 封装器不支持对关联的 ELF 文件进行仿真，但可支持对关联的 ELF 文件进行综合。

下图显示了用户封装的 IP 的设计层级，此 IP 包含用户 RTL 源文件。

图 10：**定制封装的 IP 设计层级**



封装的块设计

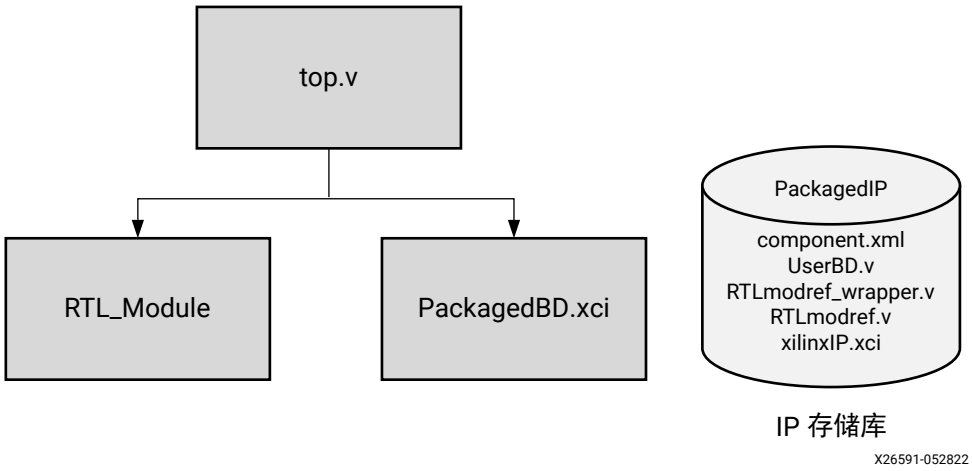
您可在当前 Vivado 工程内封装块设计源文件，并将其作为封装 IP 包含在用户 IP 存储库内。

以下是使用封装 BD 时值得注意的重要信息：

- 顶层 BD 不提供查看或修改功能。
- 不支持封装含 BDC 的 BD。
- 封装 BD 会丢失 BD 边界属性和元数据（例如，FREQ_HZ、X_INTERFACE_* 属性等）。要保留此信息，此信息必须存在于 BD 封装文件内或者复制到 IP 封装的顶层源文件中。
- 封装 BD 是块设计的快照。封装后，BD 是静态 IP，并非可编辑块设计，也不是动态块设计。
- 支持封装含 RTL 模块参考的 BD。但不支持封装含块设计容器的 BD。
- 不允许封装含 CIPS 或 NoC IP 的 BD。
- SmartConnect 和 AXI Interconnect 地址信息在封装 IP 内均为静态信息。

下图显示了用户封装的块设计的设计层级，此块设计包含 RTL 模块参考和 AMD IP。

图 11：**封装的块设计层级**



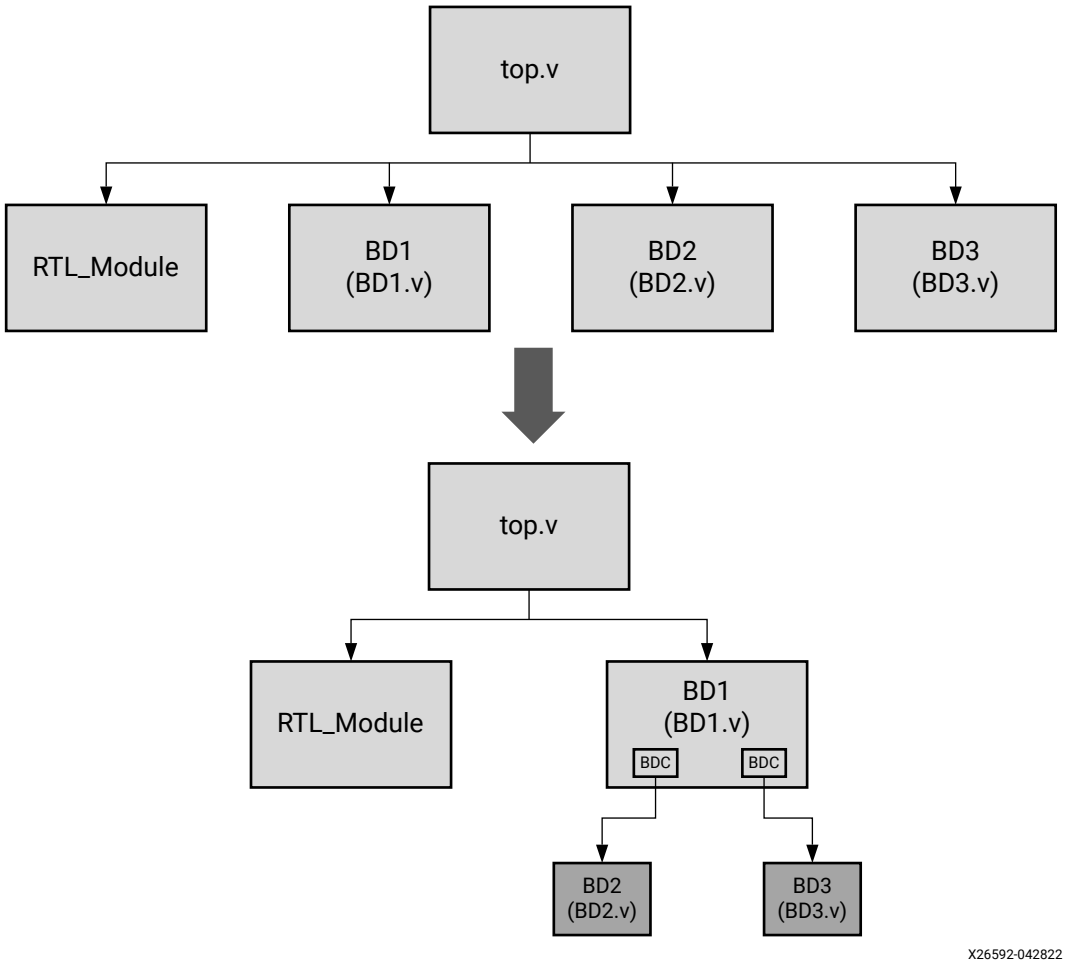
块设计容器

您可使用 RTL 设计方法来例化位于层级内各层的各 RTL 模块中的独立 IP integrator 块设计源。但这些块设计可能是独立的实例，且不能使用模块化来控制块设计内的寻址或参数传输。

块设计容器 (BDC) 可供您用于在任一块设计内例化另外的一个或多个块设计。该功能本质上就是将分层块及其内容转换为块设计。生成的块设计定义为 .bd 文件，并且也可在其他块设计工程内使用。

在下图中，某一层级上例化的 3 个独立块设计转换至该层级内的 2 个级别中，其中块设计 1 使用块设计容器来对块设计 2 和块设计 3 进行例化。

图 12：**块设计容器层级示例**



BDC 具有如下优势：

- 将大型块设计分区为 BDC 子块，每个块设计均可在子块中进行独立开发
- 在画布上复制 BD 内容
- 跨不同 IP integrator 工程复用 BD 源文件

在顶层块设计图示中可查看子块设计层级的内容。但您必须在该 BDC 的源块设计中对 BDC 的内容进行更改。随后，所作的更改将自动与顶层块设计同步。

来自顶层块设计中的 IP 的所有参数都将自动传输至子块设计 BDC 内已连接的 IP。这样即可对同一源块设计进行多次例化，并可向这些实例传输不同参数组或属性组。BDC 子块的所有寻址都必须从顶层块设计中执行。

RTL 模块参考

您可将来自 Verilog、VHDL 或 BD RTL 封装源文件的模块或实体定义直接添加到自己的块设计中。此方法支持快速添加 RTL 模块，无需将 RTL 封装为 IP 以通过 Vivado IP 目录来添加。

源 RTL 中包含的任何泛型和参数都会在向块设计添加模块时进行推断，并且可在选定模块的“Re-customize Module Reference”（重新自定义模块参考）对话框中进行配置。您必须将属性插入 HDL 代码，以便正确推断接口、时钟、复位、中断、地址和时钟使能。对于 BD 模块参考，则会在 BD 封装文件中自动创建这些属性，并在创建 BD 模块参考时耗用。

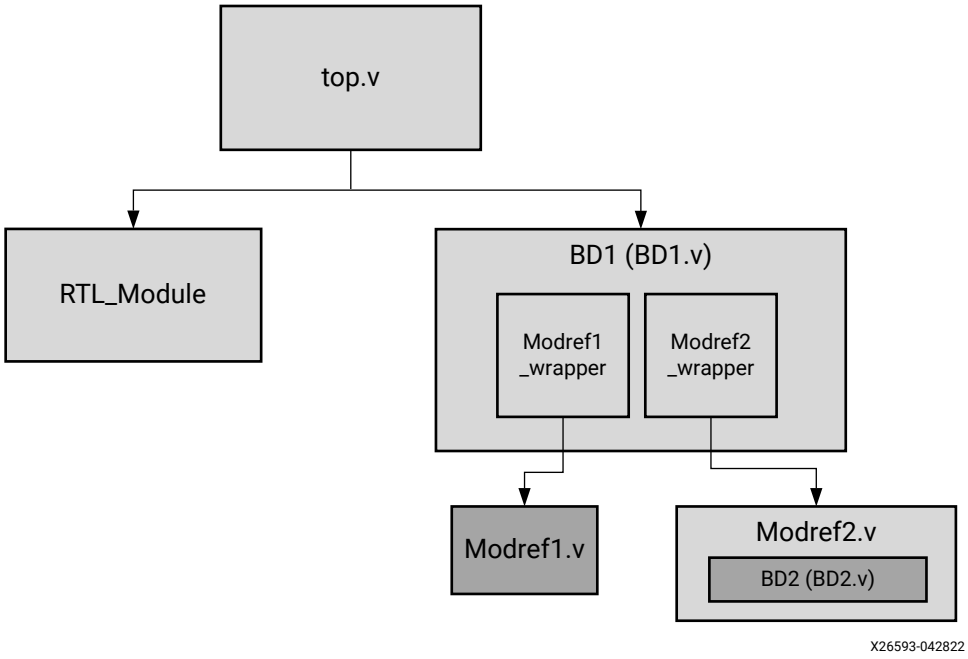
此外，不存在跨边界的参数传输。您必须在确认设计时使用属性来支持通过 IP integrator 运行 DRC。例如，IP integrator 可提供 DRC 用于确认源时钟和目标时钟之间的时钟频率。通过在 RTL 代码中指定正确的频率，您即可确保自己的设计连接正确。

以下是使用 RTL 模块参考时值得注意的重要信息：

- IP integrator 中的“Module Reference”（模块参考）功能允许为 RTL 代码中嵌套的 IP 推断 XCI（.xci 扩展名）文件。虽然支持推断大部分 IP，但有小部分 IP 不受支持。如需了解更多信息（包括不受支持的 IP 列表），请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994) 中的相应内容。
- RTL 模块定义不得包含网表（EDIF 或 DCP），也不得包含在 RTL 模块内设置为非关联 (OOC) 的其他模块。
- 模块定义支持的语言仅限于 VHDL 和 Verilog。对于位于 RTL 模块顶层的模块或实体定义，不支持 SystemVerilog 和 VHDL 2008。
- 包含模块参考的块设计不能封装为 IP。您必须改为将该模块单独封装为 IP，然后封装包含该 IP 的 BD。

下图显示了包含 2 个 RTL 模块参考的 BD 设计层级：其中一个模块参考包含 RTL，另一个模块参考则包含另一个 BD。

图 13：**RTL 模块参考设计层级**



块设计比较

在版本控制系统中，您可以使用 IP integrator 来比较 2 个块设计并判定所发生的改变。例如，您可以将受源代码控制的块设计与已检出的块设计进行比较。该功能可在 GUI 上使用，也可作为独立命令 (`diffbd`) 来使用。

块设计更新

将 Vivado 工具升级至最新版本后，AMD 建议您将块设计升级至最新 IP 版本、执行必要的设计更改、确认设计并生成目标输出文件。

您也可以选择仅升级块设计内的部分 IP。要使用此功能，必须在前版本的 Vivado 工具中完整生成块设计。AMD 强烈建议在升级前保存工程的备份副本。当出现 IP 升级状态时，请取消选中无需升级的任意 IP。对于不升级的 IP，将应用 `LOCK_UPGRADE` 属性，在 Tcl 控制台消息中，该属性显示 IP 旁。设计中其余 IP 将以正常流程进行升级。

有关采用 Versal 器件 IP 进行设计的建议

Versal 器件包含独特 IP，需遵循如下所述特殊注意事项：

- 在设计中包含 Control, Interface, and Processing System (CIPS) IP。

  CIPS IP 必须存在于每个 Versal 自适应 SoC 设计中，因为此 IP 包含启动和配置器件所必需的平台管理控制器 (PMC)。如果设计不包含 CIPS IP，则会为您的设计添加链接后/布局前 DRC 标志。如需获取 PMC 和 PS 的描述，请参阅《Versal 自适应 SoC 技术参考手册》(AM011)。如需了解有关 CIPS IP 的信息，请参阅《Control, Interface and Processing System LogiCORE IP 产品指南》(PG352)。如需了解有关 CPM 的信息，请参阅《Versal 自适应 SoC CPM CCIX 架构手册》(AM016)、《Versal Adaptive SoC CPM Mode for PCI Express 产品指南》(PG346) 和《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347)。

- 通过 CIPS IP 配置 CPM 控制器，包括 GT 选择。

  如需了解更多信息，请参阅《Versal Adaptive SoC CPM Mode for PCI Express 产品指南》(PG346)。通过 IP 目录使用 PCI Express® IP 即可配置对 PCIe® 接口的 PL 访问权。如需了解有关 PCIe 的更多信息，请参阅下列文档：

  - 《Versal Adaptive SoC Integrated Block for PCI Express LogiCORE IP 产品指南》(PG343)
  - 《Versal Adaptive SoC PCIe PHY LogiCORE IP 产品指南》(PG345)
  - 《Versal Adaptive SoC CPM Mode for PCI Express 产品指南》(PG346)
  - 《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347)

- 对 NoC 资源设计要求开展提前分析和尽早确认。

  NoC IP 充当物理 NoC 的逻辑表示法。Vivado IP integrator 汇总了连接和服务质量 (QoS) 信息，此信息可组成统一的流量规格。如需了解有关 NoC 和集成存储器控制器 IP 以及性能调优的更多信息，请参阅《Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP 产品指南》(PG313)。

- 使用 Advanced I/O Planner 来分配 DDR 存储器控制器的物理位置和管脚。如需了解更多信息，请参阅《Advanced I/O Wizard LogiCORE IP 产品指南》(PG320)。

  硬化的 DDR 存储器控制器集成到 NoC IP 内。NoC 编译器会在汇总设计要求时选择 DDR 存储器控制器的位置。DDR 存储器控制器的物理分配会在实现期间进行相应的调整。

- 将 SmartConnect 限制为下列连接：需 AXI4-Lite 的连接、当设计其余部分已耗尽 NoC 带宽时用于补充 NoC 的连接或者无足够 NoC 端口时用于补充 NoC 的连接。

  NoC 是在整个 Versal 器件中移动数据的首选方法。如需了解更多信息，请参阅《SmartConnect LogiCORE IP 产品指南》(PG247)。

- 使用 IP integrator 中的块自动化设置来辅助 IP 与 GT 之间的连接。

您必须使用此方法，因为使用 GT 资源的 Versal IP 内不再集成 GT 组件。或者也可以通过在 RTL 内直接配置、例化和连接此 IP 来手动拼接这些连接。如需了解使用 GT 父 IP 创建设计的概述，请访问此链接以参阅《Versal Adaptive SoC Transceivers Wizard LogiCORE IP 产品指南》(PG331) 中的相关内容。

- 使用 Versal Adaptive SoC Transceivers Wizard IP 配置 Versal 自适应 SoC 收发器。

  使用 Hard Block Planner 在 Versal 自适应 SoC 设计内分配 GT 四通道的物理位置。如需了解有关 Hard Block Planner 的信息，请参阅《Vivado Design Suite 用户指南：I/O 管脚分配和时钟规划》(UG899)。如需了解有关 GT 四通道完整布局以及受支持的配置选项的信息，请参阅《Versal 自适应 SoC GTY 和 GTYP 收发器架构手册》(AM002) 和《Versal 自适应 SoC GTM 收发器架构手册》(AM017)。

- Bridge IP 可用于将定制 IP 连接到 Versal 自适应 SoC GT 四通道。

  欲知详情，请访问此链接以参阅《Versal Adaptive SoC Transceivers Wizard LogiCORE IP 产品指南》(PG331) 中的相应内容。

针对不同 Versal 器件设计拓扑结构的建议

以下是 Versal 器件的主设计拓扑结构：

- RTL 作为顶层
- BD 作为顶层

所有 Versal 设计都会将设计的某些部分包含在模块框图中，因为 CIPS 和 NoC IP 都必须使用块设计 (BD) 来配置。RTL 作为顶层和 BD 作为顶层是指将块设计整合到更大的设计中的方式。在 RTL 设计作为顶层的设计中，设计的模块框图部分是作为子模块来生成并拼接到 RTL 层级其余部分中的。此设计并不局限于 RTL 层级内的单个 BD。例如，CIPS 和 NoC 可驻留在单个 BD 上，收发器可驻留在另一个 BD 上。这些设计均可独立编译，并且均可作为子模块拼接到 RTL 层级其余部分中的。这样即可为您的设计分区方式提供更多灵活选择。

在 BD 作为顶层的设计中，模块框图与所有设计源码拼接在一起。这些源码可以作为 IP 封装并添加到块设计中，或者也可以使用 RTL 模块参考来引用。使用块设计容器 (BDC) 即可将更多块设计整合到顶层设计中。

设计拓扑结构对于整体工程的最重大的影响是硬件与软件之间的交互。使用 BD 作为顶层的设计时，软硬件之间必需的交互方式相关的所有信息都会通过硬件交接文件进行无缝传递。RTL 暂无法提供类似程度的透明度。以下提供了有关每种设计拓扑结构的使用时机的建议：

- 如果您愿意手动创建设备树并为设计外设安装驱动程序，则建议使用 RTL 作为顶层。
- 如果您希望硬件无缝交接，则建议使用 BD 作为顶层。

相关信息

例化块设计

为基于平台的设计流程创建硬件平台

可扩展平台是 Vitis 环境内基于平台的设计流程的基础。此平台使应用开发者能够免于应付低层次基础架构的细枝末节，转而将注意力集中于开发处理器系统内的特定功能，例如，软件、AI 引擎计算图或 PL 内核逻辑。开发者可以通过使用 Vitis 环境将 AI 引擎和 PL 内核添加到可扩展平台中。使用 Vitis 环境添加的内核将自动接入平台中包含的存储器、中断控制器、复位和时钟设置资源。

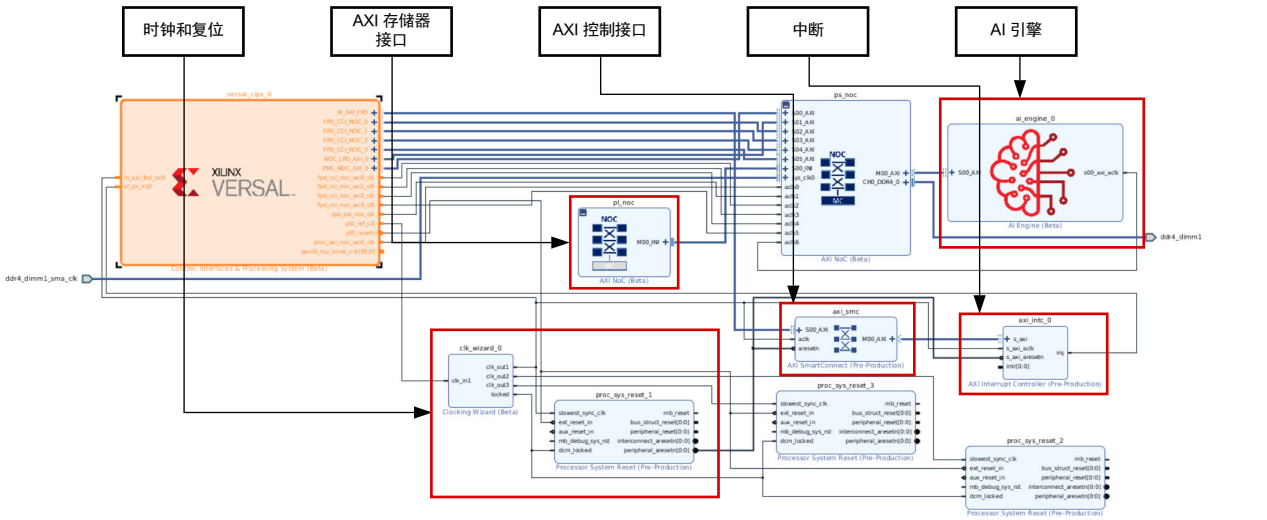
Vitis 可扩展平台由硬件组件和软件组件构成。平台的硬件组件是使用 Vivado IP integrator 设计的。平台的软件组件则是使用 Vitis 或 PetaLinux 工具链创建的。

本节旨在描述使用 IP integrator 创建和配置平台硬件组件的流程。如需了解有关创建软件平台和封装整个平台的信息，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

使用 IP integrator 创建的设计必须捕获基本 Versal 自适应 SoC 硬件 IP 块，包括 CIPS、NoC、I/O 控制器和 AI 引擎阵列。设计还必须公开相应的逻辑接口，以供内核稍后使用 Vitis v++ 连接器连接到这些接口。借助 AMD IP、定制 IP 与 RTL 的组合，来完成处理器、存储器和所有外部开发板接口的配置。

下图显示了硬件平台的模块框图示例。

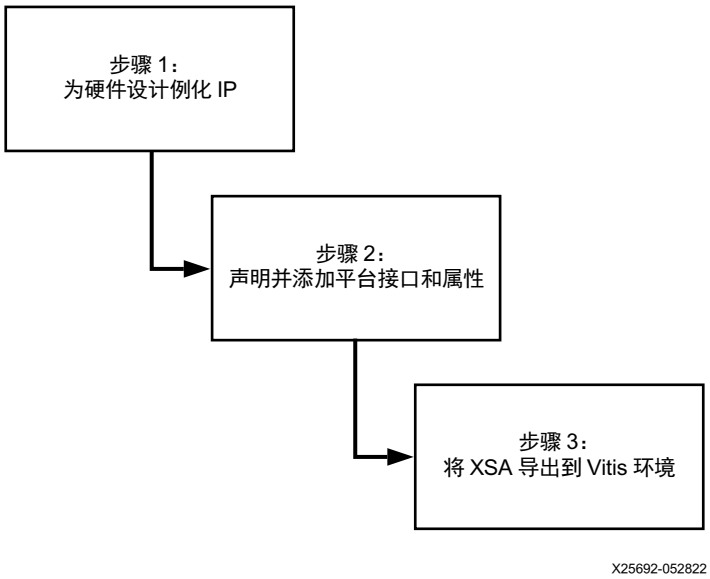
图 14：**硬件平台模块框图示例**



X25691-052822

在 IP integrator 中构建硬件平台的步骤如下所述。

图 15：**创建硬件平台**



1. 例化必要的 IP 以创建平台的硬件部分。

   这包括正确配置 CIPS、NoC、Processor System Reset Module 以及 Clocking Wizard IP 以满足目标平台的需求。这些块的输入和输出管脚可供硬件函数使用。硬件函数由 Vitis 工具在后续步骤中构建。

2. 在 IP integrator 中构建块设计后，先在 IP 块上声明并添加平台 (PFM) 接口和属性，然后再将设计作为硬件平台导出至 Vitis 环境。

   这些平台设置包括 Vitis 环境中的硬件函数所需的时钟设置、中断、复位、存储器和处理器 AXI 接口。IP integrator GUI 可提供“Platform Setup”（平台设置）窗口，用于声明这些接口及其属性。

   PFM 步骤的要求如下所述：

   - 平台内至少有 1 个已启用的 AXI 端口主接口。
   - 平台可包含一个或多个时钟。平台内至少有 1 个已启用的时钟接口。如果硬件函数使用特定时钟，那么它使用该时钟的已同步的复位输出。
   - 在平台中，通常通过 Concat 块来连接中断。

3. 生成设计后，将硬件定义 (XSA) 导出至 Vitis 环境。

   这样即可导出必要 XML 文件，供 Vitis 工具用于解读设计中使用的 IP，并且从处理器角度来看，这样还可导出存储器映射。

创建硬件平台后，必须将导出的 XSA 与软件组件封装在一起，以便创建完整 Vitis 平台，搭配 v++ 编译器/连接器一起使用。

如需了解有关 IP integrator 的更多信息，请参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994)。

第 4 章

使用 RTL 创建设计

AMD 需要使用 AMD Vivado™ IP integrator 来为 AMD Versal™ 器件例化并配置 CIPS 和 NoC IP。在启动和配置 Versal 器件时，需使用平台管理控制器 (PMC)，它包含在 CIPS IP 中。CIPS IP 仅在 IP integrator 中可用。

AMD 建议（但不强制要求）使用 IP integrator 来为设计创建顶层 RTL 封装文件。由于 Vivado 工具可以为 NoC IP 自动连接正确的仿真模型，启用 Vivado 工具的 RTL 封装文件保留功能可以简化设计仿真。您可在 IP integrator 中配置 CIPS 和 NoC、指定到设计其余部分的所有外部接口并在顶层 RTL 中例化生成的块设计。此方法会将 CIPS 和 NoC 包含在更大的系统内，但要求您手动确保在其 RTL 封装文件内正确连接仿真模型。

注释：如需了解有关 I/O 管脚分配进程、在 RTL 前的设计中由 PCB 设计师执行端口对齐以及使用时钟资源的更多信息，请参阅《Vivado Design Suite 用户指南：I/O 管脚分配和时钟规划》(UG899)。

相关信息

使用块设计来创建设计

定义理想的 RTL 设计层级

设计创建的第一步是决定如何对设计进行逻辑分区。定义层级时主要考虑的是如何对含特定功能的设计部分进行分区。这样便于特定设计人员单独设计 IP，以及隔离一段代码以供复用。

但仅根据功能来确定层级会导致对时序收敛、运行时间和调试的最优化方法考虑不周。在层级规划过程中考虑如下因素也有助于时序收敛。

在顶层附近添加 I/O 组件

尽可能在顶层附近添加 I/O 组件，以保障设计可读性。推断组件时，请提供要完成功能的描述。然后，综合工具会对 HDL 代码进行解释，以确定使用哪些硬件组件来执行该功能。可推断的组件为简单的单端 I/O（IBUF、OBUF、OBUFT 和 IOBUF）以及 I/O 中的单倍数据速率寄存器。

使用工具推断 IOBUF 或 OBUFT 组件时，请确保使能逻辑和输入/输出逻辑都位于相同层级内。如果逻辑位于不同层级内并且在各层级之间存在 KEEP_HIERARCHY 或 DONT_TOUCH 属性，那么该工具将无法推断这些缓冲器。

如差分 I/O（IBUFDS 和 OBUFDS）和双倍数据速率寄存器（IDDR 和 ODDR）等需例化的 I/O 组件也应在顶层附近进行例化。例化组件时，会将组件的实例添加到 HDL 文件中。例化可以让您完全控制组件的使用方式。因此，您将准确掌握逻辑的使用方式。

在顶层附近插入时钟元件

朝顶层方向插入时钟元件便于模块间的时钟共享。时钟共享可以减少时钟资源占用，从而提高资源使用率、提升最高时钟频率，并降低功耗。

除了在其中创建时钟的模块之外，时钟路径只能向下驱动进入模块。任何先自上而下而后又自下而上贯穿的路径都会在 VHDL 仿真中造成增量周期 (delta cycle) 问题，此类问题的调试既艰难又费时。

在逻辑边界处寄存数据路径

对层级边界输出进行寄存可将关键路径包含在单一模块或边界内。输入同样可以寄存在层级边界处。相比于遍布多个模块的路径，模块内部的时序路径始终更便于分析和修复。未在层级边界处寄存的任何路径都应采用层级重构来加以综合或者扁平化以便实现跨层级最优化。在逻辑边界处寄存数据路径有助于保留整个设计进程中的可追溯性（用于调试），因为这样可以最大限度避免跨层级最优化，并且逻辑不会跨模块迁移。

针对功能和时序调试最优化层级

如本章前文所述，把关键路径限定在同一层级边界内有助于时序的调试和满足时序要求。同样，出于功能调试（及修改）目的，相关信号应限定在同一层级上。这样即可使相关信号的探测和修改变得相对简单，因为当信号限定在单个层级中时，更易于跟踪通过综合对信号名称进行的最优化。

在模块层次应用属性

在模块层次应用属性可以使代码更整洁且更便于缩放。请勿在信号层次应用属性，而应改为在模块层次应用属性并将属性传输至当前层级中声明的所有信号。在模块层次应用属性还可支持您覆盖全局综合选项。

注意！不同于其他属性，DONT_TOUCH 属性不会从模块传输至该模块中的所有信号。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

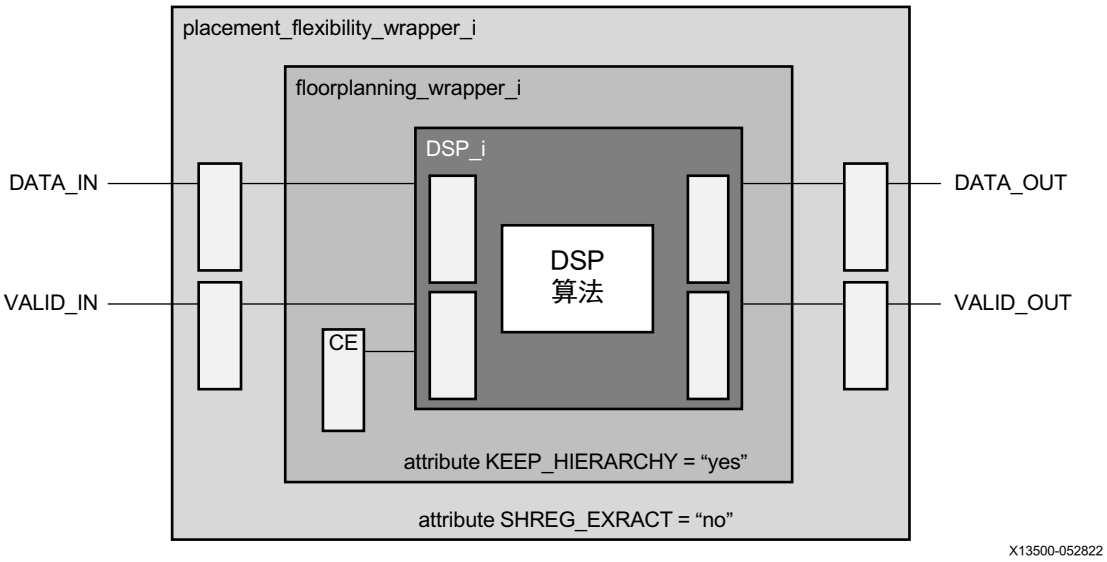
针对高级设计技巧实现层级最优化

高级设计技巧（如自下而上综合、Dynamic Function eXchange (DFX) 和非关联设计）需按层级进行规划。设计师必须根据使用的设计技巧选择合适的层级。本文不对这些技巧做详细介绍。

高速 DSP 设计的提前层级规划示例

以下示例并不适用于所有设计，但可演示层级的作用。DSP 设计一般允许对设计增加时延。这样可以在设计中添加寄存器，实现时钟频率更高的设计。此外，寄存器还可用于提升布局灵活性。这非常重要，因为时钟频率较高时，信号无法在 1 个时钟周期内遍历裸片。添加寄存器可使难以到达的区域变为可供使用。下图显示了有效的层级规划对于加速时序收敛的作用。

图 16：**有效的层级规划示例**



本设计部分分 3 个层级：

- `DSP_i`

  在 DSP_i 算法块中，输入和输出将同时寄存。由于器件拥有大量寄存器，因此，这是提升时序预算的首选方法。

- `floorplanning_wrapper_i`

  在 `floorplanning_wrapper_i` 中有 1 个 CE 信号。CE 信号一般为重负载信号，会导致时序问题，应在布局规划时包含这些信号。通过创建布局规划封装，后续即可根据需要手动对此模块进行布局规划。

  此外，在模块层次已添加 KEEP_HIERARCHY 以确保保留该层级用于布局规划，与任何其他全局综合选项无关。

- `placement_flexibility_wrapper_i`

  在 `placement_flexibility_wrapper_i` 中，寄存 DATA_IN、VALID_IN、DATA_OUT 和 VALID_OUT 信号。由于并未计划在布局规划中包含这些信号，因此这些信号不包含在 `floorplanning_wrapper_i` 中。如果这些信号包含在布局规划中，将无法满足布局灵活性的要求。

  此外，只要将 DATA_IN + VALID_IN 或 DATA_OUT 与 VALID_OUT 配对处理，稍后即可添加更多寄存器。如果添加更多寄存器，综合工具可能推断移位寄存器 LUT (SRL)，这将强制所有寄存器集中到 1 个组件中，这对于布局灵活性无益。为避免出现此情况，在模块层次已添加 SHREG_EXTRACT 并将其设置为 NO。

IP 的使用

使用预先确认的 IP 核能够大幅减少设计和确认工作量，从而加速产品上市进程。如需了解有关使用 IP 的更多信息，请参阅以下资源：

- 《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896)
- 《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994)
- Vivado Design Suite QuickTake 视频：在 Vivado 中配置和管理可复用 IP

规划 IP 要求

对任何新工程而言，规划 IP 要求都是最重要的环节之一：

- 请根据所需功能以及其他设计目标，评估 AMD 或其他第三方合作伙伴提供的 IP 选项。例如：
  - 与可用 IP 核相比，定制逻辑是否更好？
  - 以业界标准格式封装定制设计以便在多个工程中复用是否有意义？
- 考虑需要使用的接口，例如，存储器接口、网络接口和外设接口。

重要提示！为确保这些工具正确处理 IP 专用约束，请为工程添加 XCI 或 XCIX IP 源文件。处理 IP 时，请勿将 IP 生成的输出 DCP 文件作为工程源。

AMBA AXI

AMD 已对符合开放式 AMBA® 4 AXI4 互连协议的 IP 接口进行了标准化。这种标准化能够简化 AMD 和第三方提供商的 IP 的集成工作并最大程度提升系统性能。AMD 已与 Arm® 协作定义 AXI4、AXI4-Lite 和 AXI4-Stream 规格，以便将这些规格高效映射到其器件架构中。

AXI4 专为高性能、高时钟频率系统设计制定，适用于高速互连。AXI4-Lite 是 AXI4 的精简版，主要用于访问控制寄存器和状态寄存器。

AXI4-Stream 用于从主接口到从接口的单向数据串流传输。典型应用包括 DSP、视频和通信。

Vivado Design Suite IP 目录

IP 目录是集中包含 AMD 提供的 IP 的场所。在“IP catalog”中，您可查找嵌入式系统、DSP、通信、接口等的 IP 核。

在“IP catalog”中，您可浏览可用 IP 核，并查看任意 IP 的“Product Guide”（产品指南）、“Change Log”（更改日志）、“Product Web page”（产品网页）和“Answer Records”（答复记录）。

您可在“IP catalog”中通过 GUI 或 Tcl shell 来访问核并对其进行自定义。您还可使用 Tcl 脚本来自动执行 IP 核的自定义。

定制 IP

AMD 使用业界标准 IP-XACT 格式来交付 IP，并提供工具（IP 封装器 (IP packager)）来封装定制 IP。因此，您还可将自己的自定义 IP 添加到目录中，并创建 IP 存储库以便在团队内部或整个公司内部共享。来自第三方提供商的 IP 同样可添加到此目录中，前提是此类 IP 是在 IP 封装器中封装的，即使它已采用 IP-XACT 格式也是如此。

从 IP 目录选择 IP

所有 AMD 和第三方厂商的 IP 都按“通信和网络”、“视频和图像处理”、“汽车”以及“工业”等不同应用进行分类。您可根据该编目方法浏览 IP 目录，查看自己感兴趣领域的 IP 核。

IP 目录中的大部分 IP 都是免费提供的。但部分高价值 IP 要收取相应的成本并需要许可证。IP 目录会告知您，此 IP 是否需要购买以及许可证的状态。在从 IP 目录中选择 IP 的时候，应根据设计要求以及特定 IP 的功能考虑下列关键特性：

- 该 IP 所需的芯片资源（见相应的 IP 产品指南）

- 器件是否支持该 IP？是否考虑了速度等级（IP 选择往往决定速度等级选择）？如果支持，最大可实现的吞吐量以及最高频率 (Fmax) 是多少？
- 设计中所需的与开发板上辅助芯片通信的外部接口标准：
  - 以太网、PCIe® 接口等业界标准接口。
  - 存储器接口：存储器接口的数量、尺寸和性能。
  - AMD 专有接口（如 Aurora）。

    注释：也可选择设计自己的定制接口。
- IP 支持的片上总线协议（应用接口）
- 与设计其余部分互动所需的片上总线协议。示例：
  - AXI4
  - AXI4-Lite
  - AXI4-Stream
- 如果涉及多重协议，可能必须使用 IP 目录中的基础架构 IP 来选择桥接 IP 核。例如：
  - AXI-AHB bridge
  - AXI SmartConnect
  - DMA/bridge subsystem for PCIe

自定义 IP

可通过 GUI 或 Tcl 脚本来自定义 IP。

相关信息

使用自定义 GUI
使用 Tcl 脚本

使用自定义 GUI

使用图形界面是查找、研究和自定义 IP 的最简单的途径。每个 IP 都有一组专为其自定义的选项卡或页面。相关配置选项均分组归于一处。定制窗口示例如下图所示。可创建以 XCI 文件表示的独特 IP 定制方案。随后即可以此为基础为任一 IP 创建多个输出文件。

使用 Tcl 脚本

几乎每项 GUI 操作都会导致发出一条 Tcl 命令。IP 创建过程包括 Tcl 脚本中可执行且无需用户交互的所有定制选项的设置。

您需掌握配置选项的名称以及这些选项可设置为哪些值。通常首先通过 GUI 执行定制，然后在此基础上创建脚本。当您看到生成的 Tcl 脚本后，即可轻松按需修改脚本，例如，更改数据大小。

采用 Tcl 脚本创建 IP 可便于在诸如处理版本控制系统等情况下实现自动化。如需了解有关源代码管理和版本控制的信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892) 中的相应内容。

IP 版本和版本控制

自定义 IP 后，该工具会创建 1 个 XCI 文件，其中包含所有选定的参数值。每个 Vivado IDE 版本都仅支持 1 个版本的 IP。AMD 建议您使用此最新 IP 版本。如果您使用较低版本的 IP，那么必须保存所有输出文件以供低版本使用。如需了解有关源代码管理和版本控制的信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892) 中的相应内容。

RTL 编码准则

您可创建定制 RTL 来实现胶合逻辑功能以及不含适合 IP 的功能。为了获得最佳结果，请遵循本节中的编码准则进行操作。如需获取其他指南，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

使用 Vivado Design Suite HDL 模板

创建 RTL 或例化 AMD 原语时使用 Vivado Design Suite 语言模板。这些语言模板包含建议的编码结构，用于正确推断 AMD 器件架构。使用语言模板可以简化设计进程，并改进结果。要从 Vivado IDE 打开“Language Templates”（语言模板），请在 Flow Navigator 中选中“Language Templates”选项，然后选择所需模板。

控制信号和控制集

控制集是控制信号（置位/复位信号、时钟使能信号和时钟信号）的组合，用于驱动任意给定 SRL、LUTRAM、CLB 或 IMUX 寄存器。对于控制信号的任意独特组合，都会组成 1 个独立控制集。由于 Versal 自适应 SoC slice 中的寄存器共享公用控制信号，从而管辖将含有不同控制集的寄存器封装到同一个 slice 中的过程，因此该功能十分重要。例如，如果具有给定控制集的寄存器仅具有 1 个寄存器作为负载，那么其占据的 slice 中的其他寄存器对于含不同时钟信号或置位/复位信号的任何寄存器都将不可用。欲知详情，请访问此链接以参阅《Versal 自适应 SoC 可配置逻辑块架构手册》(AM005) 中的相应内容。

如果设计所含独立控制集过多，可能导致资源浪费过多并且布局选项减少，从而导致功耗上升且可实现的时钟频率降低。设计所含控制集越少，则布局选项更多且灵活性更高，并且通常可以得到更好的结果。

复位

复位是需要在设计中考虑并加以限制的较为常见且重要的控制信号之一。复位会对设计的最高时钟频率、面积和功耗产生显著影响。

经推断的同步代码可能产生如下资源：

- LUT
- 寄存器
- SRL
- 块存储器或 LUT 存储器
- DSP58 寄存器

复位的选择和使用会影响这些组件的选择，导致针对给定设计选择的资源欠佳。任何阵列上复位布局错误都可能导致截然不同的结果，比如推断 1 个块 RAM 或推断数千个寄存器。

复位的使用时机和位置

AMD 器件具有专用的全局置位/复位信号 (GSR)。在器件配置结束时，此信号会设置硬件中所有时序单元的初始值。

如果未指定初始状态，将为时序原语分配默认值。在大多数情况下，默认值为 0。FDSE 和 FDPE 原语是例外，其默认为逻辑 1。每个寄存器在配置结束时都将处于已知状态。因此无需编写仅用于在上电时初始化器件的全局复位代码。

AMD 强烈建议您谨慎判断何时设计需要复位以及何时不需要复位。大多数情况下，在控制路径逻辑上可能需要复位以确保正常运行。然而在数据路径逻辑上通常不需要复位。复位的使用限制如下：

- 限制复位信号线的总体扇出。
- 减少复位布线所需的互连数量。
- 简化复位路径的时序。
- 在大多数情况下，这样即可整体改善时钟频率、面积和功耗。

建议：评估每个同步块，尝试判断是否需要复位以确保正常运行。默认情况下，除非确定确实有需要，否则请勿编写复位代码。

功能仿真可轻松判断是否需要复位。

对于不含复位编码的逻辑，可以灵活选择用于映射逻辑的器件资源。

随后，综合工具即可选择最适合代码的资源，并通过考量如下因素来尽可能实现最佳结果：

- 请求的功能
- 时钟周期要求
- 可用器件资源
- 功耗

同步复位对比异步复位

如需复位，AMD 建议使用同步复位。同步复位相比于异步复位具有如下优势：

- 同步复位可以直接映射至器件架构中的更多资源元件。
- 异步复位会影响通用逻辑结构的最大时钟频率。由于所有 AMD 器件的通用寄存器均可将置位/复位编程为异步或同步，可能看似使用异步复位不会受到任何惩罚。使用全局异步复位并不会增加控制集。但由于需要将此复位信号布局到所有寄存器元件，因此会增加布线复杂性。
- 复位断言有效期间，异步复位导致块 RAM、LUTRAM、以及 SRL 的存储器内容损坏的可能性更高。对于含异步复位（用于驱动块 RAM、LUTRAM 和 SRL 的输入管脚）的寄存器尤其如此。
- 需要更高密度或者微调布局时，同步复位会为控制集重新映射提供更多的灵活性。如果在布局更优化的 slice 中发现不兼容的复位，那么可将同步复位重新映射到寄存器的数据路径。这样即可根据需要减少布线资源使用率并增加布局密度，从而实现正确的适配并改善可实现的时钟频率。

以下另提供了其他注意事项：

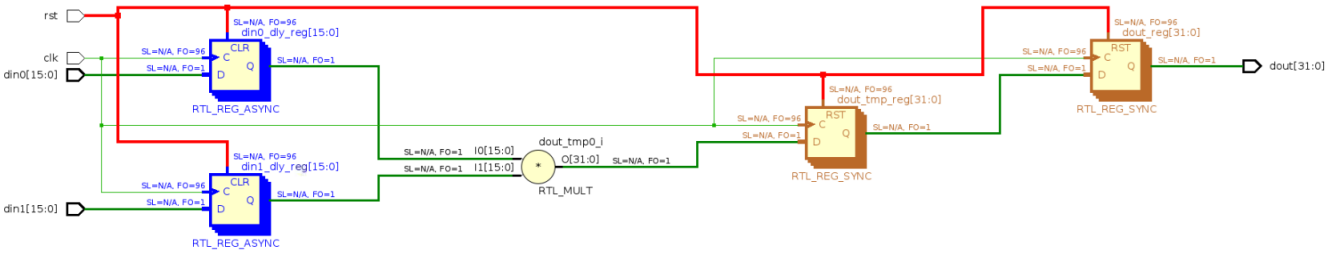
- 对于同步复位的复位小毛刺，时钟会充当滤波器的角色。但如果这些毛刺出现在有效时钟沿附近，那么触发器可能会变为亚稳态。
- 同步复位可能需要扩大脉冲宽度，以确保复位信号脉冲宽度足够容纳时钟沿有效期间存在的复位。

- 使用异步复位时，请务必对异步复位断言无效操作进行同步。虽然在复位断言有效期间可以忽略时钟与复位之间的相对时序，但复位释放必须同步到时钟。不执行复位释放时钟沿同步可能导致亚稳态。复位释放期间，与寄存器的时钟管脚相关的复位管脚必须满足建立时间和保持时间的时序条件。异步复位的建立时间和保持时间条件违例（例如，复位的恢复和移除时序）可能导致触发器变为亚稳态，从而因切换至未知状态而导致设计失败。请注意，此状况与触发器数据管脚的建立时间和保持时间条件违例相似。

复位编码示例：含同步和异步复位的乘法器流水线寄存器

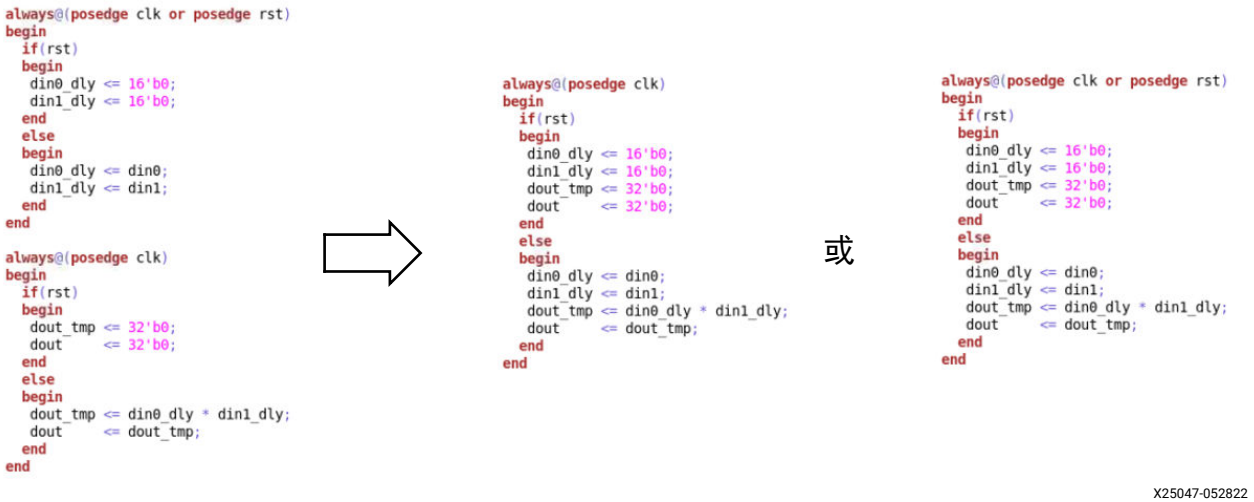
在此示例中，流水线寄存器的逻辑内混用了以专用 DSP 资源为目标的同步和异步复位，导致将乘积累加映射到 DSP 原语的效果欠佳。下图显示了基于 16x16 位 DSP58 的乘法器，该乘法器使用含异步复位的输入流水线寄存器和含同步复位的乘法器输出寄存器。综合的输入阶段必须使用常规互连结构寄存器。含同步复位的乘法器寄存器打包到 DSP 输出（DSP58 M 寄存器和 P 寄存器）内。这样总共就能得到 32 个额外的互连结构寄存器，且生成的 DSP58 配置为：AREG/BREG=0，MREG=1，PREG=1。

图 17：**含流水线寄存器的乘法器（同步和异步复位）**



为了充分利用现有 DSP 原语功能，您可以重写前例，将流水线寄存器上混用的异步复位和同步复位更改为所有流水线寄存器采用单一类型的复位。下图显示了如何在 RTL 中更改复位定义，使乘法器逻辑周围的所有流水线寄存器都使用单一类型的复位（同步复位或异步复位）。完成此更改后，综合即可充分利用 DSP58 内部寄存器（AREG/BREG=1，MREG=1，PREG=1）。

图 18：**针对围绕乘法器逻辑的所有流水线阶段将混用同步和异步复位更改为采用单一类型的复位**



```
always@(posedge clk or posedge rst)
begin
  if(rst)
  begin
   din0_dly <= 16'b0;
   din1_dly <= 16'b0;
  end
  else
  begin
   din0_dly <= din0;
   din1_dly <= din1;
  end
end

always@(posedge clk)
begin
  if(rst)
  begin
   dout_tmp <= 32'b0;
   dout     <= 32'b0;
  end
  else
  begin
   dout_tmp <= din0_dly * din1_dly;
   dout     <= dout_tmp;
  end
end
```

```
always@(posedge clk)
begin
  if(rst)
  begin
   din0_dly <= 16'b0;
   din1_dly <= 16'b0;
   dout_tmp <= 32'b0;
   dout     <= 32'b0;
  end
  else
  begin
   din0_dly <= din0;
   din1_dly <= din1;
   dout_tmp <= din0_dly * din1_dly;
   dout     <= dout_tmp;
  end
end
```

或

```
always@(posedge clk or posedge rst)
begin
  if(rst)
  begin
   din0_dly <= 16'b0;
   din1_dly <= 16'b0;
   dout_tmp <= 32'b0;
   dout     <= 32'b0;
  end
  else
  begin
   din0_dly <= din0;
   din1_dly <= din1;
   dout_tmp <= din0_dly * din1_dly;
   dout     <= dout_tmp;
  end
end
```

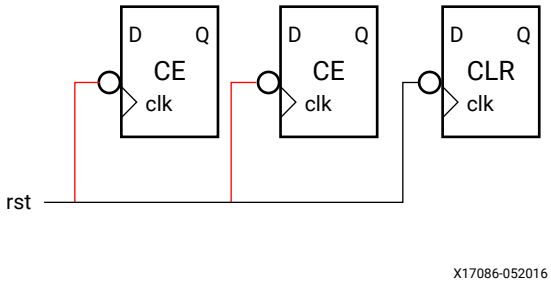
X25047-052822

尝试在 HDL 代码中消除复位时出现问题

当最优化代码以消除复位时，注释掉复位声明中的条件无法生成期望的结构，反而导致出现问题。例如，下图显示了 3 个分别使用异步复位的流水线阶段。如果尝试通过注释掉含复位条件的代码来消除其中 2 个流水线阶段的复位条件，则将启用异步复位（`rst` 的逻辑取反）。

图 19：**注释掉含复位条件的代码**

```
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        //din_dly1 <= 16'b0;
        //din_dly2 <= 16'b0;
        dout     <= 16'b0;
    end
    else
    begin
      din_dly1 <= din;
      din_dly2 <= din_dly1;
      dout     <= din_dly2;
    end
end
```



X17086-052016

移除复位的最佳途径是创建独立的顺序逻辑过程，其中一个过程用于复位条件，另一个过程用于非复位条件，如下图所示。

图 20：**为含复位和无复位的寄存器创建独立的过程声明**

```
always@(posedge clk)
begin
    din_dly1 <= din;
    din_dly2 <= din_dly1;
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        dout <= 16'd0;
    else
        dout <= din_dly2;
end
```

提示：使用复位时，请确保过程语句中的所有寄存器均已复位。

时钟使能

如果正确使用，时钟使能可显著降低设计功耗，同时对面积或最高时钟频率的影响极小。但错误使用时钟使能时，可能会导致：

- 资源使用率增加
- 布局密度降低
- 功耗上升
- 可实现的时钟频率降低

大多数情况下，低扇出时钟使能是造成控制集数量过高的主要因素。

创建时钟使能

在同步块中编写的条件语句不完整时，就会创建时钟使能。通过推断时钟使能即可在先前条件未得到满足时，保留最后一个值。需要使用该功能时，采用此方式进行编码即为有效。在某些情况下，虽然先前条件值未得到满足，但并不影响输出。在此情况下，AMD 建议采用定义的常量（即为信号赋值 1 或 0）来关闭该条件（即，使用 `else` 子句）。

在大多数实现方案中，这不会导致额外增加逻辑，同时可避免使用时钟使能。但对于大型总线而言，推断时钟使能时，如果其中保留的值有助于降低功耗，则属例外情况，不适用此规则。此规则的基本前提是推断少量寄存器数时，由于时钟使能会增加控制集的数量，因此会产生不利影响。但是对较大型的群组而言，其利大于弊，所以建议使用。

复位和时钟使能优先顺序

在 AMD 器件中，所有寄存器构建时均将置位/复位设置为优先于时钟使能，不论所述对象为异步或同步置位/复位都是如此。为了取得最佳结果，AMD 建议您始终在同步时钟内通过 `if/else` 结构进行编码，以将置位/复位置于时钟使能（如果需要）之前。通过编码将时钟使能置于优先位置会导致强制复位进入数据路径，造成逻辑额外增加。

相关信息

时钟设置准则

使用综合属性控制使能/复位提取

您可以通过根据需要应用 DIRECT_RESET/DIRECT_ENABLE/EXTRACT_RESET/EXTRACT_ENABLE 属性来强制控制集映射，以处理给定结构的控制集的映射。

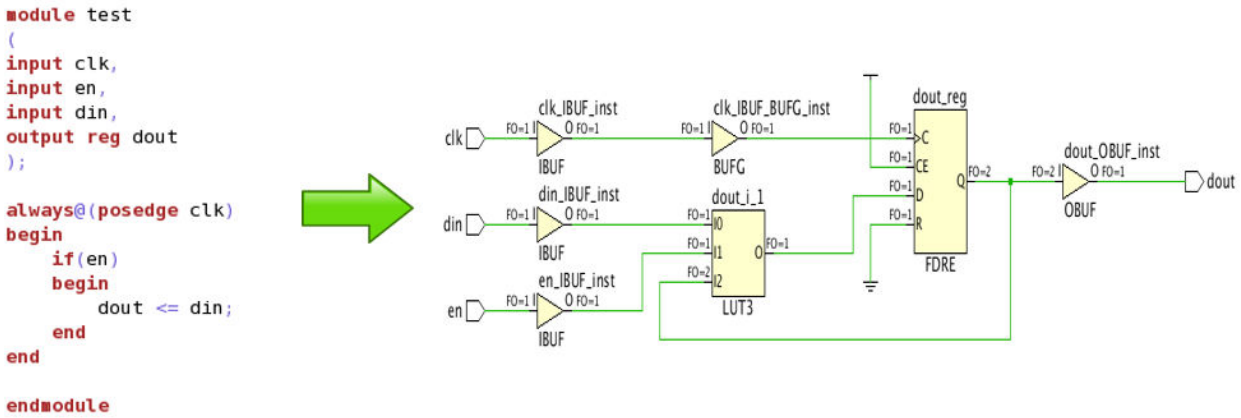
当设计包含同步复位/使能时，如果负载等于或高于 `-control_set_opt_threshold` 综合开关所设置的阈值，综合会创建通过 CE/R/S 管脚映射的逻辑椎，或者如果负载低于该阈值，那么综合会创建通过 D 管脚映射的逻辑椎。Versal 自适应 SoC 的默认阈值为 2。

使用 DIRECT_ENABLE 和 DIRECT_RESET

要使用控制集映射，可向已连接到使能信号/复位信号的信号线应用属性，这将强制综合使用 CE/R 管脚。

在下图中，使能信号 (`en`) 只能连接到 1 个触发器。因此，综合引擎已将 en 信号连接到逻辑的 FDRE/D 管脚椎。请注意，CE 管脚已绑定到逻辑 1。

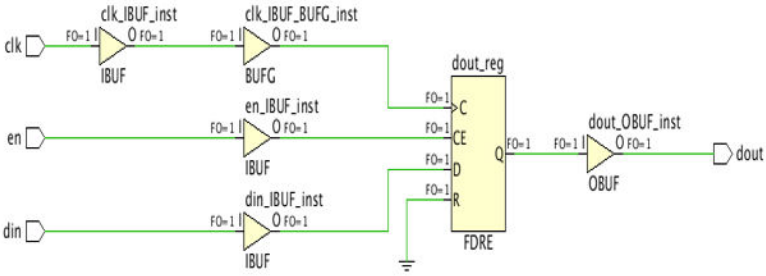
图 21：**使用数据路径逻辑完成时钟使能实现**



要覆盖此默认行为，可使用 DIRECT_ENABLE 属性。例如，下图显示了如何通过将 DIRECT_ENABLE 属性添加到端口/信号来将使能信号 (en) 连接到寄存器的 CE 管脚。

图 22：**使用 direct_enable 完成专用时钟使能实现**

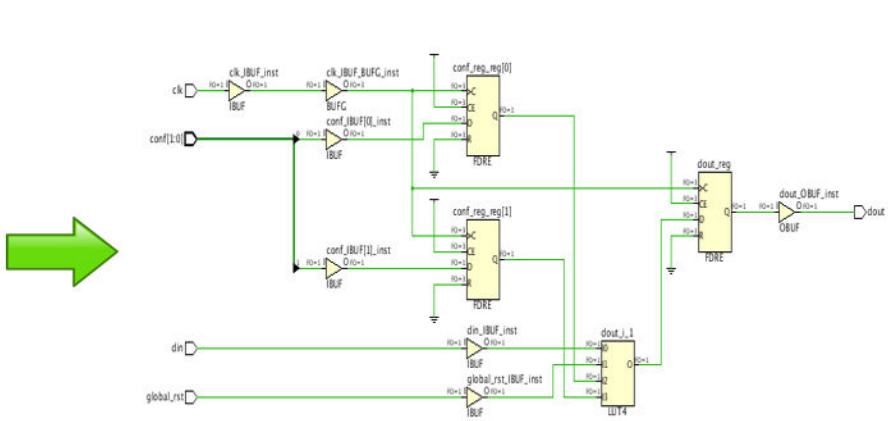
```
module test
(
input clk,
(* direct_enable = "true" *) input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```



下图显示了 RTL 代码，其中 global_rst 或 int_rst 可将寄存器复位。默认情况下，两者都映射到逻辑的复位管脚椎。

图 23：**通过数据路径逻辑映射的多个复位条件**

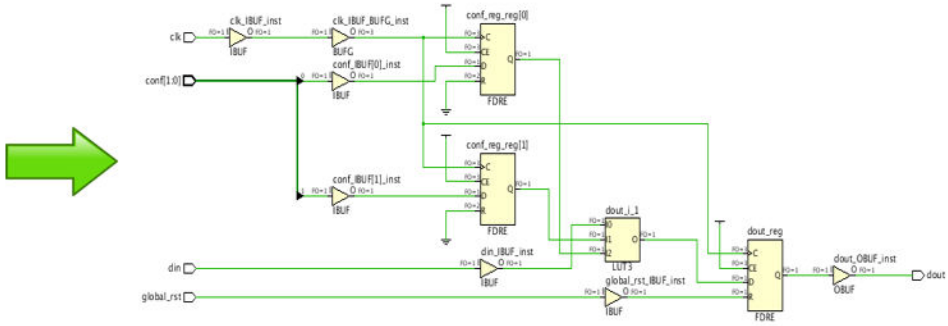
```
module test (
             input clk,
             input global_rst,
             input [1:0] conf,
             input din,
             output reg dout
            );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```



可使用 DIRECT_RESET 属性来指定要连接到寄存器复位管脚的复位信号。例如，下图显示了如何使用 DIRECT_RESET 属性来仅将 global_rst 信号连接到寄存器 FDRE/R 管脚，并将 int_rst 信号连接到逻辑的 FDRE/D 椎。

图 24：**使用 DIRECT_RESET 属性的专用复位管脚的用法**

```
module test (
            input clk,
            (* direct_reset = "true" *) input global_rst,
            input [1:0] conf,
            input din,
            output reg dout
           );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```



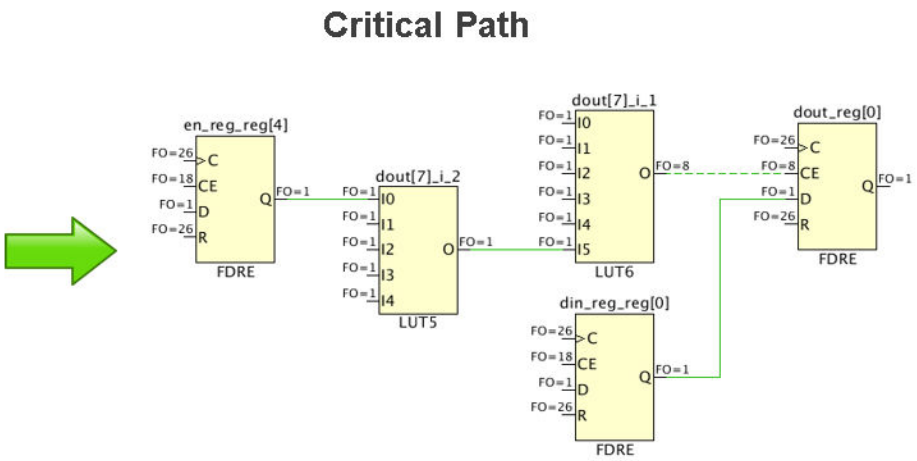
将逻辑从控制管脚推送到数据管脚

分析关键路径期间，您可能会发现有多条路径止于控制管脚。您必须对这些路径进行分析，以判定是否能够将逻辑推送到数据路径中，同时避免发生惩罚（例如，额外的逻辑层次）。相同逻辑层次的前提下，相比于指向 CE/R/S 管脚的路径，指向 D 管脚的路径的延迟较小，因为从最后一个 LUT 的输出到 FF 的 D 输入之间存在直接连接。以下编码示例显示了如何将逻辑从控制管脚推送到寄存器的数据管脚。

在以下示例中，dout_reg[0] 的使能管脚的逻辑层次数为 2，而数据管脚的逻辑层次数则为 0。在此情况下，您可通过将使能逻辑移至 D 管脚来改进时序，方法是在 RTL 文件中的 dout 寄存器定义上将 EXTRACT_ENABLE 属性设置为“no”。

图 25：**止于寄存器控制管脚（使能）的关键路径**

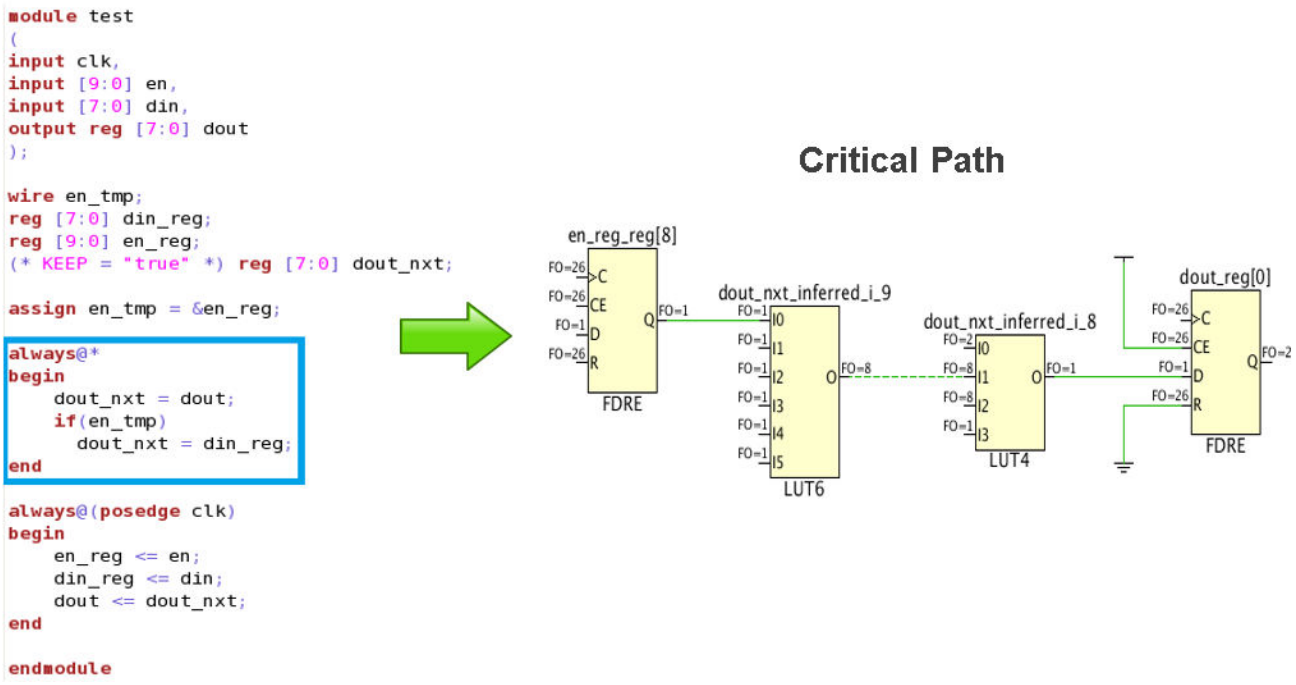
```
module test
(
input clk,
input [9:0] en,
input [7:0] din,
output reg [7:0] dout
);

wire en_tmp;
reg [7:0] din_reg;
reg [9:0] en_reg;

assign en_tmp = &en_reg;

always@(posedge clk)
begin
    en_reg <= en;
    din_reg <= din;
    if(en_tmp)
      dout <= din_reg;
end

endmodule
```



以下示例显示了如何拆分组合逻辑和顺序逻辑以及如何将完整逻辑映射到数据路径中。这将把逻辑推送到仍含 2 个逻辑层次的 D 管脚中。

您可通过将 EXTRACT_ENABLE 属性设置为“no”来实现相同的结构。如需了解有关 EXTRACT_ENABLE 属性的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

图 26：**止于寄存器的数据管脚（禁用使能提取）的关键路径**



信号控制技巧

- 检查是否真正需要全局复位。
- 避免异步控制信号。
- 保持时钟、使能和复位信号极性一致。
- 勿将置位和复位编码到同一寄存器元件中。
- 如果确实需要异步复位，请务必对其断言无效执行同步。

掌握推断的结果

代码最终必须映射到器件中存在的资源上。请尽力理清所处理的关键架构中的关键算术、存储和逻辑元件。这样在对设计功能进行编码时，即可预测代码将映射到的硬件资源。理解此映射方式将便于您尽早洞悉所有潜在的问题。

以下示例演示了了解硬件资源和映射方式给某些设计决策所带来的帮助：

- 对于超出 8 位的加法、减法和加减法，通常会使用进位链，并且每 2 位加法使用 1 个 LUT（即，8 位加 8 位的加法器使用 8 个 LUT 和关联的进位链）。对于三进制加法或者如果将任一加法器的结果与另一个值相加但中间不使用寄存器，那么，每 3 位加法使用 2 个 LUT（即，8 位加 8 位加 8 位的加法使用 16 个 LUT 和关联的进位链）。
- 一般乘法针对的是 DSP 块。宽度不超过 27x24 的有符号位和宽度不超过 26x23 的无符号位将映射到单个 DSP 块。乘积更大的乘法可能会映射到多个 DSP 块。DSP 块内部具有流水打拍资源。

针对推断到 DSP 块内的逻辑正确执行流水打拍操作可显著改善最大时钟频率并降低功耗。描述乘法时，围绕乘法进行三级流水打拍操作可生成最佳的建立时间、时钟输出 (clock-to-out) 和功耗特性。流水打拍操作层级过浅（一级或无）可能导致时序问题并导致这些块的功耗增加，而 DSP 内部的流水打拍寄存器却被闲置。

- 深度不超过 16 位的 2 个 SRL 可映射到单个 LUT，而高达 32 位的单个 SRL 同样可映射到单个 LUT。
- 对于可生成标准 MUX 组件的条件代码：
  - 4 选 1 MUX 可实现 1 个 LUT，从而生成 1 个逻辑层。
  - 8 选 1 MUX 可实现 3 个 LUT，从而生成 2 个逻辑 (LUT) 层。
  - 16 选 1 MUX 可实现 5 个 LUT，从而有效生成 2 个逻辑 (LUT) 层。

对于通用逻辑，请考虑给定寄存器的唯一输入的数量。根据此数量，可估算 LUT 数和逻辑级数。一般情况下，输入不超过 6 个时，始终生成 1 个逻辑层。理论上，2 个逻辑层可管理最多 36 个输入。但实际上，应假定 2 个逻辑层最多可管理约 20 个输入。总而言之，随着输入数量以及逻辑公式复杂性的提升，所需 LUT 和逻辑级数也会增加。

重要提示！在设计周期中尽早检查硬件资源的可用性以及有效利用这些资源的方式即可简化修改工作。此方法较之设计周期后期时序收敛期间再行检查的效果更好。

推断 RAM 和 ROM

可通过多种方式指定 RAM 和 ROM。每种方法都有各自的优缺点。

- 推断

  优点：
  - 便于移植
  - 便于读取和理解
  - 自我文档化
  - 快速仿真

  不足：
  - 不能访问所有可用的 RAM 配置
  - 结果可能并非最佳

  由于推断结果一般较为理想，因此建议采用推断，除非给定用例不受支持，或者无法可实现的时钟频率、面积或功耗方面产生足够的结果。在此类情况下，请尝试其他方法。

  推断 RAM 时，AMD 建议您使用 Vivado 工具中提供的 HDL 模板。如前文所述，使用异步复位会给 RAM 推断造成不利影响，应避免使用。

- 赛灵思可参数化宏 (XPM)

  优点：
  - 可在各 AMD 器件系列之间进行移植
  - 快速仿真
  - 支持非对称宽度
  - 可预测的结果质量 (QoR)

  不足：

- 仅支持 XPM 选项

XPM 是基于使用固定模板的推断构建的，此类模板无法修改。因此，其 QoR 有保证，并且可以支持标准推断所不具备的功能。当标准推断不支持所需功能时，AMD 建议您改为使用 XPM。

注释：使用 `compile_simlib` 编译仿真库时，会自动编译 XPM。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

- 直接例化 RAM 原语

优点：

- 对实现方案有最高控制权限
- 能访问块的各项功能

不足：

- 代码可移植性差
- 功能和用途冗长繁琐，难以理解

- IP 目录提供的 IP 核

优点：

- 在使用多个组件时一般能提供更优化的结果
- 易于指定和配置

不足：

- 代码可移植性差
- 需要管理核

相关信息

使用 Vivado Design Suite HDL 模板

实现 RAM 时的性能注意事项

为了有效推断存储元件，请考虑下列影响性能的因素：

- 使用专用块还是分布式 RAM

RAM 可在专用块 RAM 内实现，或者也可在使用分布式 RAM 的 LUT 内实现。此选择不仅影响资源选择，还会对可实现的时钟频率的和功耗产生巨大影响。

一般情况下，RAM 所需深度即第一项标准。深度高达 64 位的存储器阵列通常在 LUTRAM 内实现，其中，深度不超过 32 位时，每个 LUT 映射 2 位，深度高达 64 位时每个 LUT 映射 1 位。根据可用资源和综合工具分配，更深的 RAM 也可在 LUTRAM 内实现。

深度超过 256 位的存储器阵列一般在块存储器中实现。AMD 器件能够按不同宽度和深度组合灵活映射此类结构。您应熟练掌握这些配置以便了解用于代码中较大型的存储器阵列声明的块 RAM 数量和结构。

- 使用输出流水线寄存器

对于以更高时钟频率工作的设计，输出寄存器是必需的，对于所有设计也建议使用输出寄存器来简化时序收敛。这样可改进块 RAM 的时钟输出时序。此外，第二个输出寄存器是很实用的，因为 slice 输出寄存器的时钟输出时序比块 RAM 寄存器更快。同时使用 2 个寄存器的总读取时延为 3。在推断这些寄存器时，这些寄存器与 RAM 阵列应处于相同层级。这样便于工具将块 RAM 输出寄存器合并到原语中。

- 使用输入流水线寄存器

  当 RAM 阵列较大并跨多个原语映射时，可覆盖裸片上的大片面积。这可能导致地址线路和控制线路上出现时序收敛问题。请考虑在生成这些信号后，于 RAM 之前添加额外寄存器。为了进一步改进时序，请在流程后期使用 `phys_opt_design` 来复制此寄存器。输入上不含逻辑的寄存器将更便于复制。

阻碍块 RAM 输出寄存器推断的场景

AMD 建议在单一层级上对所有存储器和输出寄存器都进行推断，因为这是确保按期望方式进行推断的最简单的方法。在两种情况下会对块 RAM 输出寄存器进行推断。首先是在输出上存在额外的寄存器，其次是在存储器阵列中对读取地址寄存器重新进行时序约束。只有在使用单端口 RAM 时才会出现此类情况。如下图所示：

图 27：**RAM 具有额外的读取寄存器用于进行块 RAM 输出寄存器推断**

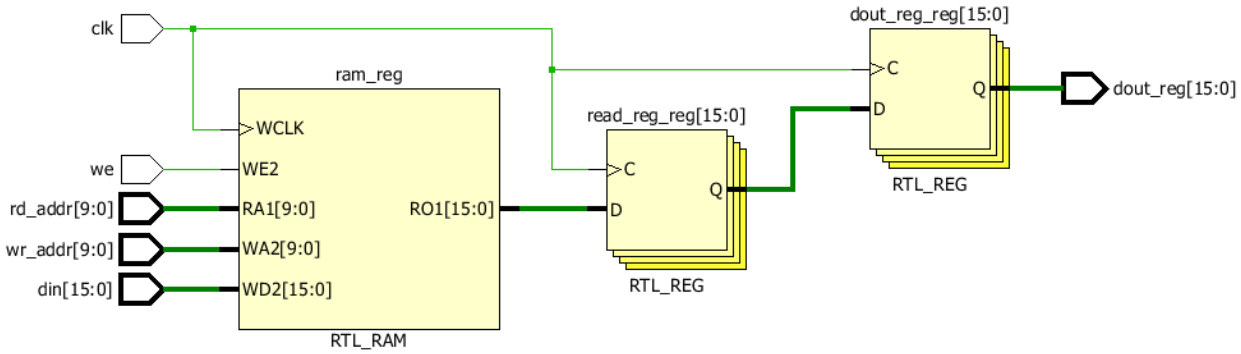
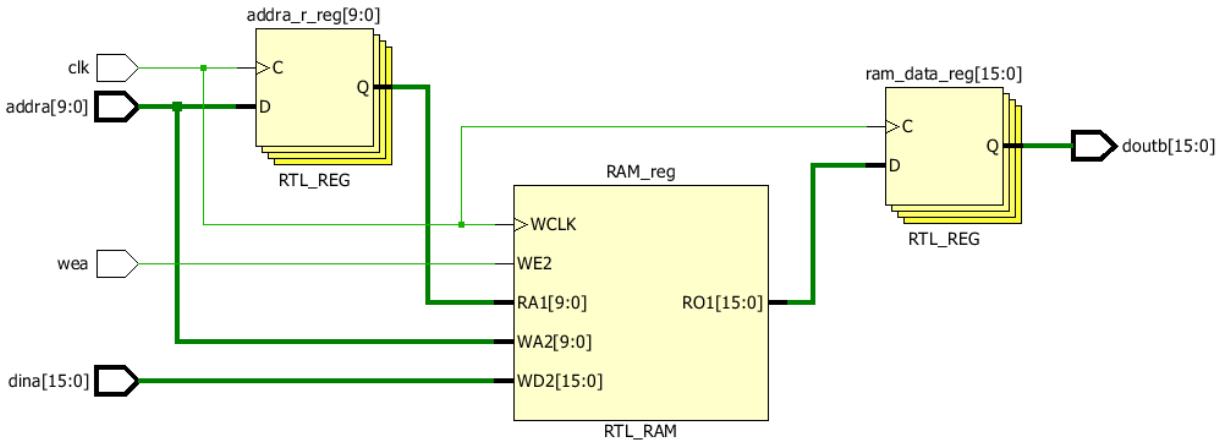


图 28：**对地址寄存器重定时前的 RAM 视图**

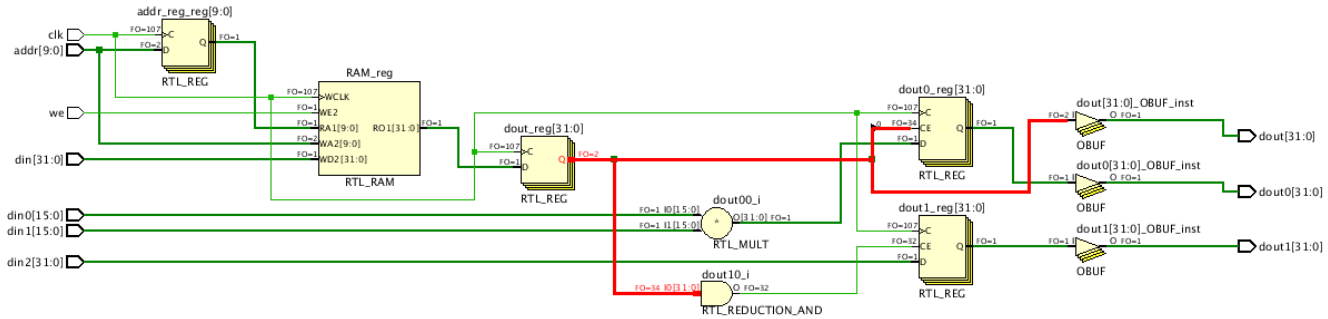


只要与这些示例有些许偏差，就会导致无法推断输出寄存器。

检查读取数据寄存器输出上的多重扇出

为使第二寄存器被 RAM 原语吸收，来自存储器阵列的数据输出位的扇出必须为 1。如下图中所示。

图 29：**多重扇出阻止块 RAM 输出寄存器执行推断**

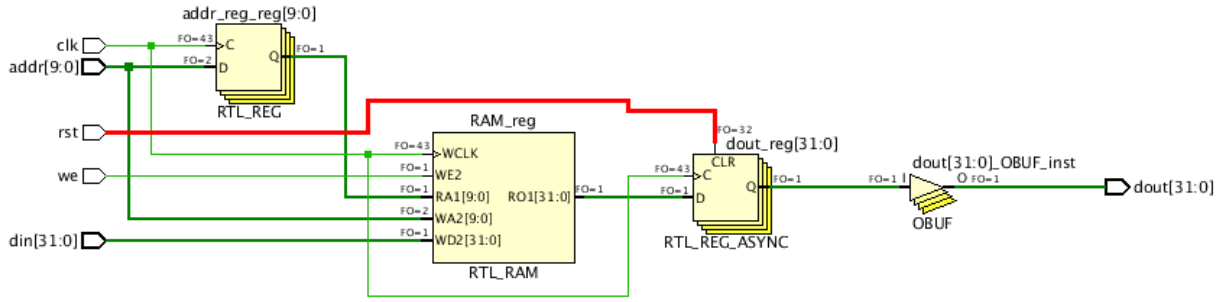


检查地址/读取数据寄存器上的复位信号

存储器阵列不应复位。仅限 RAM 输出才能容许复位。Versal 架构可处理 RAM 输出上的异步复位或同步复位。但是，输出寄存器和可选输出寄存器必须具有相同的异步复位。

下图突出显示了为确保正确推断 RAM 和输出寄存器而需要避免的行为示例。读取地址上的寄存器将用于创建块 RAM，但在块 RAM 中将不会使用输出上的额外寄存器，因为其异步复位与 RAM 输出复位不匹配。

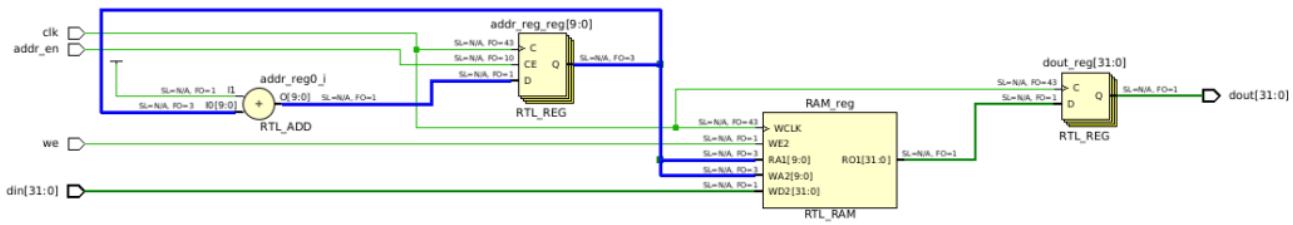
图 30：**检查地址/读取数据寄存器上的复位**



检查寄存器的反馈结构

请确保寄存器不含反馈逻辑，因为此逻辑会阻碍寄存器最优化。在以下示例中，地址寄存器会驱动 RAM 和加法器，即，无法将此寄存器打包到块 RAM 中。由此生成的电路是块 RAM，其中 dout 寄存器打包到 RAM 内以使 RAM 完全同步。但 RAM 不使用输出寄存器（DOA_REG 和 DOB_REG 将设为“0”），这样效率低下。

图 31：**检查 RAM 块周围的寄存器上是否存在反馈**



将存储器映射到 UltraRAM 块

UltraRAM 是 1 个大型存储器块，具有 2 个使用同一时钟的端口。除块 RAM 资源外，还包含 UltraRAM。UltraRAM 可配置为 4Kx72、8Kx36、16Kx18 和 32Kx9。

可采用以下任一方法在设计中运用 UltraRAM：

- 依靠综合在 HDL 中的存储器声明上设置 `ram_style = "ultra"` 属性，以便推断 UltraRAM。
- 例化 AMD XPM_MEMORY 原语。
- 例化 UltraRAM UNISIM 原语。

以下代码示例显示了 XPM 存储器的例化，可在 HDL 语言模型中使用。突出显示的 `MEMORY_PRIMITIVE` 和 `READ_LATENCY` 参数均为关键参数，用于推断存储器作为 UltraRAM 用以实现最优资源映射效率。

- `MEMORY_PRIMITIVE = "ultra"` 指定存储器将推断为 UltraRAM。
- `READ_LATENCY` 用于定义存储器输出上存在的流水线寄存器数量。

较大的存储器将映射到 UltraRAM 矩阵，其中包含多个配置为行 x 列结构的 UltraRAM 单元。

可基于深度创建单列或多列矩阵。UltraRAM 列高度的当前默认阈值为 8，可通过 CASCADE_HEIGHT 属性来加以控制。

单列和多列 UltraRAM 矩阵的差异如下所述：

- 单列 UltraRAM 矩阵使用内置硬件级联，无互连结构逻辑。
- 多列 UltraRAM 矩阵的每个列中均使用内置硬件级联，并附加一些互连结构逻辑用于连接各列。可能需要增加流水打拍以保持可达成的时钟频率。这是通过增加读取时延来控制的。Vivado 工具会根据需要自动将这些寄存器封装到 UltraRAM 中。

图 32：**在 RTL 代码中通过 XPM 指定 UltraRAM**

```
xpm_memory_spram # (

  // Common module parameters
  .MEMORY_SIZE        (8*(4096*72)),     //positive integer
  .MEMORY_PRIMITIVE   ("ultra"),         //string; "auto", "distributed", "block" or "ultra";
  .MEMORY_INIT_FILE   ("none"),          //string; "none" or "<filename>.mem"
  .MEMORY_INIT_PARAM  ("" ),             //string;
  .USE_MEM_INIT       (0),               //integer; 0,1
  .WAKEUP_TIME        ("disable_sleep"),//string; "disable_sleep" or "use_sleep_pin"
  .MESSAGE_CONTROL    (0),               //integer; 0,1

  // Port A module parameters
  .WRITE_DATA_WIDTH_A (72),              //positive integer
  .READ_DATA_WIDTH_A  (72),              //positive integer
  .BYTE_WRITE_WIDTH_A (72),              //integer; 8, 9, or WRITE_DATA_WIDTH_A value
  .ADDR_WIDTH_A       (16),               //positive integer
  .READ_RESET_VALUE_A ("0"),             //string
  .READ_LATENCY_A     (9),               //non-negative integer
  .WRITE_MODE_A       ("read_first")     //string; "write_first", "read_first", "no_change"

) xpm_memory_spram_inst (

  // Common module ports
  .sleep          (1'b0),

  // Port A module ports
  .clka           (clka),
  .rsta           (rsta),
  .ena            (ena),
  .regcea         (regcea),
  .wea            (wea),
  .addra          (addra),
  .dina           (dina),
  .injectsbiterra (1'b0),  //do not change
  .injectdbiterra (1'b0),  //do not change
  .douta          (douta),
  .sbiterra       (),      //do not change
  .dbiterra       ()       //do not change

);
```

上述示例使用 32 K x 72 存储器配置，此配置使用 8 个 UltraRAM。为了增大 UltraRAM 的最高时钟频率，应向级联链添加更多流水打拍寄存器。这是通过增加读取时延参数值来实现的。

如需了解有关 Vivado 综合中推断 UltraRAM 的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相关内容。

通过编码实现最优化的 DSP 和算术推断

AMD 器件中的 DSP 块可以执行许多不同的功能，包括：

- 乘法
- 加法和减法
- 比较器
- 计数器
- 一般逻辑

DSP 块属于高度流水打拍的块，具有多个寄存器阶段，能在降低资源总体功耗的情况下实现高速运行。AMD 建议将准备映射到 DSP58 中的代码完全流水打拍，这样即可利用所有的流水线阶段。为了灵活运用这一额外资源，功能中应避免使用置位条件，这样才能正确映射到该资源。

AMD 器件中的 DSP58 slice 寄存器只包含复位功能，没有置位功能。因此除非必要，应避免围绕乘法器、加法器、计数器或其他可在 DSP58 slice 中实现的逻辑进行置位编码（在施加信号的情况下，逻辑值等于 1）。

许多 DSP 设计都非常适合采用 AMD 架构。要充分利用该架构，必须熟悉底层特性和功能，以便设计输入代码能够充分利用这些资源。

DSP58 块使用符号化的算术实现方案。AMD 建议在 HDL 源代码中使用有符号的值进行编码，以便与资源功能实现最佳匹配并实现总体上最有效的映射。如果在代码中使用无符号的总线值，综合工具也许仍然能够使用该资源，但由于需要进行无符号到有符号转换，该组件可能无法实现满位精度。

如果预计目标设计将包含大量加法器，AMD 建议对设计进行评估，以便更好地利用 DSP58 slice 的预加法器和后加法器。例如在使用 FIR 滤波器的情况下，可使用加法器级联来构建脉动型滤波器，而不必使用多重连续加法功能（加法器树）。如果采用对称滤波器，那么可以使用专用预加法器进行评估，以便进一步将该功能整合到数量更少的 LUT、触发器和 DSP slice 中（大多数情况下可减少一半的资源占用）。

在了解这些功能的基础上，就可以提前进行适当的权衡取舍，并体现在 RTL 代码当中，从而从一开始就实现更加顺畅和更加高效的实现方案。在大多数情况下，AMD 建议推断 DSP 资源。

对移位寄存器和延迟线进行编码

一般而言，移位寄存器应具备以下部分或全部控制和数据信号特征：

- 时钟
- 串行输入
- 异步置位/复位
- 同步置位/复位
- 同步/异步并行负载
- 时钟使能
- 串行或并行输出

AMD 器件包含专用 SRL16 和 SRL32 资源（集成在 LUT 中）。这样无需使用触发器资源即可高效实现移位寄存器。但是这些元件仅支持左 (LEFT) 移位操作，且 I/O 信号数量有限：

- 时钟
- 时钟使能
- 串行数据输入
- 串行数据输出

此外，SRL 提供用于确定移位寄存器长度的地址输入（针对 SRL16 为 A3、A2、A1、A0 输入）。移位寄存器可采用固定的静态长度，也可以动态调节。在动态模式下，每次向地址管脚应用新地址时，LUT 访问延迟时间过后，就会在 Q 输出上显示新的比特位置值。

在 SRL 原语中不提供同步和异步置位/复位控制信号。但是，如果 RTL 代码包含复位，那么 AMD 综合工具就会围绕 SRL 推断其他逻辑以提供复位功能。

为了在使用 SRL 时得到更高的时钟频率，AMD 建议在专用分片 (slice) 寄存器中实现移位寄存器的最终阶段。slice 寄存器时钟输出 (clock-to-out) 时间比 SRL 更短。这样即可为源自移位寄存器逻辑的路径提供更大的时序裕量。除非因属性或跨层级边界最优化限制而导致例化此资源，或者已阻止综合工具推断此寄存器，否则综合工具会自动推断此寄存器。要推断其他寄存器，请将动态延迟的信号单独寄存在 RTL 内。

AMD 建议您使用 Vivado Design Suite HDL 模板中演示的 HDL 编码样式。

使用寄存器实现芯片内的布局灵活性时，请使用以下属性关闭 SRL 推断：

```
SHREG_EXTRACT = "no"
```

如需了解有关综合属性以及如何在 HDL 代码中指定这些属性的更多信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901)。

所有已推断的寄存器、SRL 和存储器的初始化

GSR 信号线会将所有寄存器初始化为 HDL 代码中指定的初始值。如果未提供初始值，综合工具可自行判断将初始状态赋值为 0 或 1。Vivado 综合通常默认设置为 0，但少数例外除外，如独热 (one-hot) 状态机编码。

对于任何已推断的 SRL、存储器或其他同步元件，同样可为其定义初始状态，并在配置时将其初始状态编程到关联的元件中。

AMD 强烈建议您对所有同步元件进行相应的初始化。寄存器初始化完全可由所有主要的器件综合工具进行推断。这样可消除单纯为初始化而添加复位的需求，并且使 RTL 代码与功能仿真中实现的设计更紧密匹配，因为所有同步元件配置后都会从器件中的已知值启动。

寄存器和锁存器初始状态 VHDL 编码示例 1：

```
signal reg1 : std_logic := '0'; -- specifying register1 to start as a zero
signal reg2 : std_logic :=  '1' ; -- specifying register2 to start as a one
signal reg3 : std_logic_vector(3 downto 0):="1011"; -- specifying INIT
value for
4-bit register
```

寄存器和锁存器初始状态 Verilog 编码示例 2：

```
reg register1 = 1'b0; // specifying regsiter1 to start as a zero
reg register2 = 1'b1; // specifying register2 to start as a one
reg [3:0] register3 = 4'b1011; //specifying INIT value for 4-bit register
```

另外，在 Verilog 中还可使用 `initial` 语句：

```
reg [3:0] register3;
initial begin
  register3= 4'b1011;
end
```

判断例化或推断的时机

AMD 建议您为自己的设计添加 RTL 描述，并使用综合工具将代码映射到器件中的可用资源。除了增强代码的可移植性外，所有推断所得逻辑均可供综合工具查看，从而使其能够执行各项功能之间的最优化。最优化包括逻辑复制、重构与合并，以及通过重定时来平衡寄存器间的逻辑延迟。

综合工具最优化

默认情况下，完成器件库单元例化后，综合工具不会对其进行最优化。即便收到对器件库单元进行最优化的指令，综合工具一般也无法执行与 RTL 相同等级的最优化操作。因此，综合工具通常仅在出入这些单元的路径上执行最优化，而不会对穿过这些单元的路径执行最优化。

例如，如果对某个 SRL 进行例化，并且将其包含在长路径中，那么此路径可能成为瓶颈。此 SRL 相比于常规寄存器的时钟输出 (clock-to-out) 延迟更长。为了保留此 SRL 所达成的面积缩小结果，同时改善其时钟输出时序特性，将创建 1 个比实际期望的延迟少 1 个延迟的 SRL，并在常规触发器中实现最后一个阶段。

判断例化的时机

当综合工具映射无法满足时序、功耗或面积约束时，或者当无法推断器件内的特定功能时，可能需要例化。

借助例化，即可全权掌控综合工具。例如，为了提高时钟频率，可单独使用 LUT 来实现比较器，而不是采用综合工具通常所选择的 LUT 与进位链组合方法，后者通常适用于节省面积。

有时，例化可能是利用器件中可用的复杂资源的唯一途径。原因可能是：

- HDL 语言限制

  例如，无法描述 VHDL 中的双倍数据速率 (DDR) 输出，因为它需要 2 个单独的进程驱动同一个信号。

- 综合工具推断限制

  例如，综合工具当前无法根据 RTL 描述推断时钟修改块 (CMB)。因此，您必须将其例化。

  如果您决定例化 AMD 原语，请参阅目标架构的相应《用户指南》和《库指南》，以便充分了解组件功能、配置和连接功能。

  对于推断和例化，AMD 建议您使用来自 Vivado Design Suite 语言模板的例化和语言模板。

以下提供了一些实用技巧：

- 尽可能推断功能。
- 当综合的 RTL 代码不满足要求时，请先复查要求，然后再将代码替换为器件库组件例化。
- 编写公用 Verilog 和 VHDL 行为构造时或者需要例化所需原语时，请考虑使用 Vivado Design Suite 语言模板。

改善最高频率的编码样式

对于高性能设计而言，本章节中所讨论的编码方法可以减少潜在的时序危机。

关键路径上的高扇出

高扇出信号线在设计进程早期阶段更便于处理。目标时钟频率要求和路径的结构往往会导致扇出过高。您可以使用以下技巧来解决高扇出信号线的问题。

建议：在综合后使用 `report_high_fanout_nets` Tcl 命令识别高扇出信号线。在实现流程期间，监控这些信号线对设计时序收敛的影响。

在设计各部分中减少不必要的负载

对于高扇出控制信号，请评估是否设计的所有编码部分都需要该信号线。减少负载数量可以大幅度减少时序问题。

复制高扇出信号线驱动

寄存器复制可通过复制寄存器来减少给定信号的扇出，从而提升关键路径的速度。这便于实现工具更加灵活地对各类不同负载和相关逻辑进行布局布线。综合工具广泛采用了这种方法。

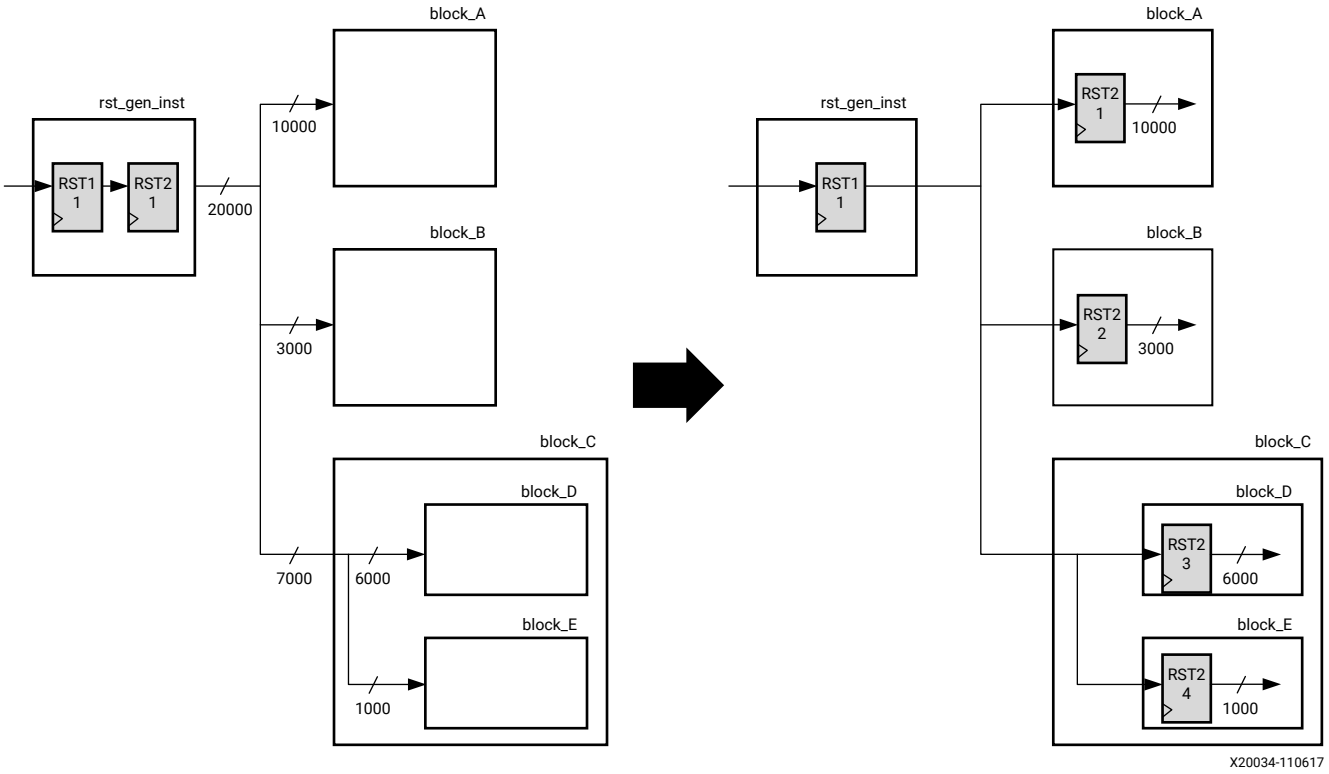
大多数综合工具使用扇出阈值限值来自动判定是否需要复制寄存器。降低此全局阈值即可自动复制高扇出信号线。但这样就无法控制需复制的寄存器范围以及这些寄存器的负载分组方式。此外，全局复制机制无法准确评估时序裕量，导致不必要的复制单元、逻辑占用率增加以及潜在功耗增加。

对于高频率设计，要减少扇出，最好对高扇出信号使用平衡树。可考虑根据设计层级手动复制寄存器，因为层级中包含的单元通常布局在一起。例如，在下图所示的平衡复位树中，在 RTL 中复制高扇出复位 FF RST2 以平衡不同模块之间的扇出。如果需要，物理综合可以基于布局信息执行进一步的复制以改进 WNS。

提示：要在综合中保留重复寄存器，请使用 KEEP 属性，而不是 DONT_TOUCH。在实现流程后期进行物理最优化期间，DONT_TOUCH 属性会阻止进行进一步最优化。

注释：如果复制的是 LUT1 而不是寄存器，表明应用的属性或约束错误。

图 33：**高扇出复位变换为平衡复位树**



建议：在全局高扇出信号上使用 MAX_FANOUT 属性会导致复制结果欠佳，与在综合中降低全局扇出限制时类似。因此，AMD 建议仅在支持中低扇出的局部信号上的层级内使用 MAX_FANOUT。

切勿复制用于同步跨时钟域的信号的寄存器。如果这些寄存器上存在 ASYNC_REG 属性，将导致工具无法复制这些寄存器。如果同步链扇出极高且复制必须满足时序要求，那么请在不含 ASYNC_REG 约束的同步链之后添加额外的寄存器。

流水打拍注意事项

另一种提升性能的方法是对具有多个逻辑层次的长数据路径进行重构并将其分布在多个时钟周期中。此方法可加速时钟周期并增加数据吞吐量，但代价是时延和流水线开销逻辑管理工作增加。

由于器件包含许多寄存器，因此通常额外的寄存器和开销逻辑不足为虑。但是，数据路径将跨多个周期，需特别留意设计其余部分，并考量路径时延增加的问题。

提前考量流水打拍

提前而非滞后考量流水线有助于改进时序收敛。在后期对某些路径添加流水线通常会导致在电路中产生传输时延差异。这可能导致看似微小的更改需要对部分代码进行大幅重新设计。

在设计中提前识别是否有机会添加流水线可以显著改进时序收敛、实现运行时间（因为时序问题变得更易于解决）和器件功耗（因为相关逻辑切换次数减少）。

检查推断的逻辑

对设计进行编码时，请注意正在推断的逻辑。请注意如下条件，以了解其他流水线设置注意事项：

- 带有大扇入的逻辑椎

  例如，需要大量总线或多个组合信号来计算输出的代码。

- 布局受限、时钟输出缓慢或者具有较高的建立时间要求的块

  例如，块 RAM 内不含输出寄存器或者未适当流水打拍的算术代码。

- 导致布线过长的强制布局

  例如，如果管脚分配强制跨芯片布线，则可能需要流水打拍才能执行高速操作。

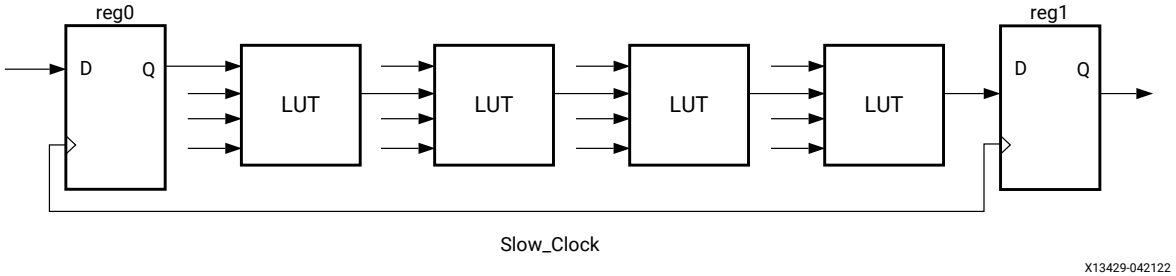
- 包含大量 XOR 函数的逻辑

  大量 XOR 函数通常导致转换率高，由此造成大量动态功耗损耗。对这些函数进行流水打拍可降低转换率，从而对所述电路的功耗产生积极影响。

在下图中，时钟速度受下列因素限制：

- 源触发器的时钟输出时间
- 贯穿四个逻辑层次的逻辑延迟
- 四个函数发生器的相关布线
- 目标寄存器的建立时间

图 34：**流水打拍前**

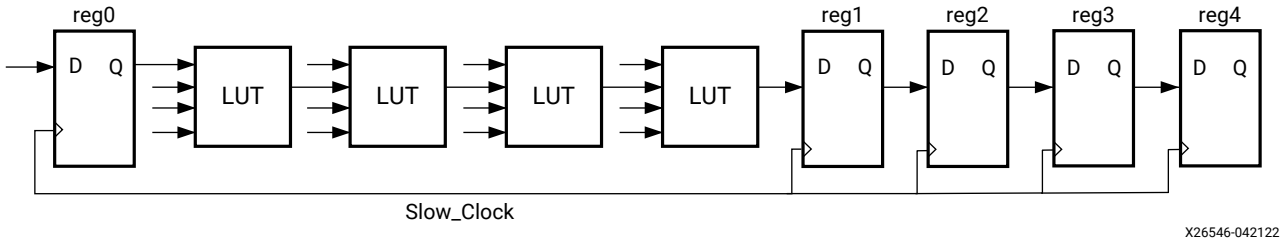


使用下列任一方法确保设计正确使用流水线寄存器：

- 在 RTL 代码中，在要重定时的逻辑之前或之后（最好在层级内）添加寄存器。
- 使用 Vivado 综合全局重定时或 BLOCK_SYNTH.RETIMING 选项，分析路径时序并移动寄存器以改善时序（如可能）。
- 或者为了进一步控制，可使用 `retiming_forward` 和 `retiming_backward` 综合属性。您可在特定寄存器上添加这些属性，强制工具穿过组合逻辑进行定时，忽略逻辑的时序得分。如需了解有关这些属性的更多信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901)。

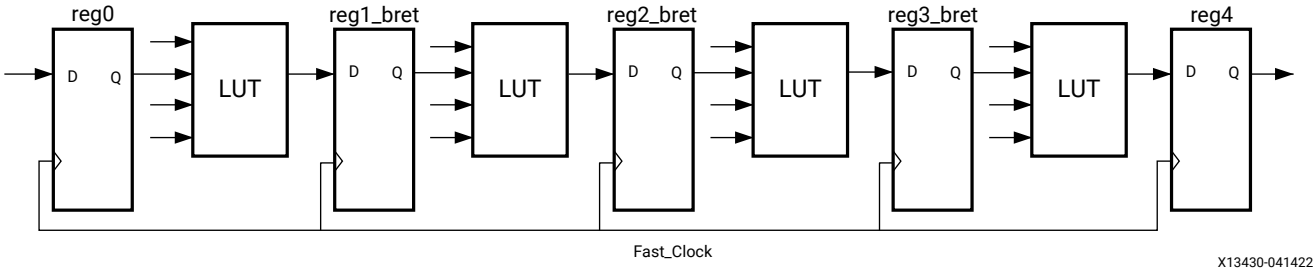
下图显示了添加额外寄存器之后的流水打拍。

图 35：**添加额外寄存器后的流水打拍**



下图是“流水打拍前的图示”中所示的数据路径示例。由于触发器与函数发生器包含在相同 slice 内，时钟速度受到下列要素的限制：源触发器的时钟输出时间、贯穿 1 个逻辑层次的逻辑延迟、布线延迟，以及目标寄存器的建立时间。在此示例中，完成流水打拍和重定时之后，系统时钟运行速度比原始设计中更快。

图 36：**重定时后流水打拍**



以下代码示例演示了如何使用重定时属性来强制执行“重定时后流水打拍”图例中所示的特定重定时。

```
(* retiming_backward = 3 *) reg reg1;
(* retiming_backward = 2 *) reg reg2;
(* retiming_backward = 1 *) reg reg3;
```

确定是否需要流水打拍

常用的流水线方法可以识别大型组合逻辑路径、将其细分为较小的路径，并在这些路径之间引入寄存器阶段，从而实现每个流水线阶段的理想平衡状态。

要确定设计是否需要采用流水线方法，请确定时序的频率和分布在每个时钟组中的逻辑数量。您可以使用含 `-logic_level_distribution` 选项的 `report_design_analysis` Tcl 命令来确定每个时钟组的逻辑级分布。

提示：设计分析报告还可突出显示不含任何逻辑层的路径数，您可以使用这些路径确定代码中需要修改的位置。

平衡时延

要通过添加流水线阶段来平衡时延，请将此阶段添加到控制路径中，而不是数据路径中。数据路径包含更宽的总线，这可增加所使用的触发器和寄存器资源的数量。

例如，如果有一条 128 位数据路径、2 个寄存器阶段并且需要 5 个时延周期，插入 3 个寄存器阶段会导致额外产生 3 x 128 = 384 个触发器。或者，您可以使用寄存器来控制启用数据路径的逻辑。使用 5 个阶段的单比特寄存器可分别控制数据路径触发器的使能信号和多周期路径时序例外。

注释：此示例仅适用于某些设计。例如，如果在中间数据路径触发器中存在扇出，那么仅采用 2 个阶段是无效的。

建议：器件中的最优 LUT:FF 比率为 1:1。如果设计所含 FF 数量显著增加，就会将更多不相关的逻辑封装成 slice，这将造成布线复杂性增加并且可能导致 QoR 劣化。

平衡流水线深度和 SRL 的使用

如果寄存器流水线较深，应尽可能将更多寄存器映射在 SRL 中，以避免寄存器使用率显著增加。例如，宽度为 32 的数据的流水线深度为 9，这会导致每个比特 9 个寄存器，总计使用 32 x 9 = 288 个寄存器。要将同样的结构映射到 SRL，需使用 32 个 SRL。每个 SRL 的地址管脚 A4 到 A0 都连接到 5'b01000，从而实现 9 个阶段的深度。

在综合期间可通过多种方法来推断 SRL，包括：

- SRL
- REG -> SRL
- SRL -> REG
- REG -> SRL -> REG

您可在 RTL 代码中使用 `srl_style` 属性创建这些结构，如下所示：

- `(* srl_style = "srl" *)`
- `(* srl_style = "reg_srl" *)`
- `(* srl_style = "srl_reg" *)`
- `(* srl_style = "reg_srl_reg" *)`

常见的错误是在较深的流水线阶段中使用不同的使能/复位控制信号。以下示例显示的是深度为 9 的流水线阶段中的复位，其中复位连接到第 3、第 5 和第 8 个流水线阶段。对于这种结构，工具只能将流水线阶段映射到寄存器，因为在 SRL 原语中有 1 个复位管脚。

```
FF->FF->FF(reset) -> FF->FF(reset)->FF->FF->FF(reset)->FF
```

要充分利用 SRL 推断，请：

- 确保流水线阶段不含任何复位。
- 分析是否确实需要复位。
- 在 1 个触发器上使用复位（例如，在流水线的第一个阶段或最后一个阶段）。

避免不必要的流水打拍

对于使用率较高的设计，流水打拍过多可能导致次优结果。例如，不必要的流水线级增加了触发器和布线资源的数量，如果使用率高，这可能限制布局和布线算法。

注释：如果有许多具有 0/1 逻辑级别的路径，请检查以确保这是有意为之。

考虑流水打拍宏原语

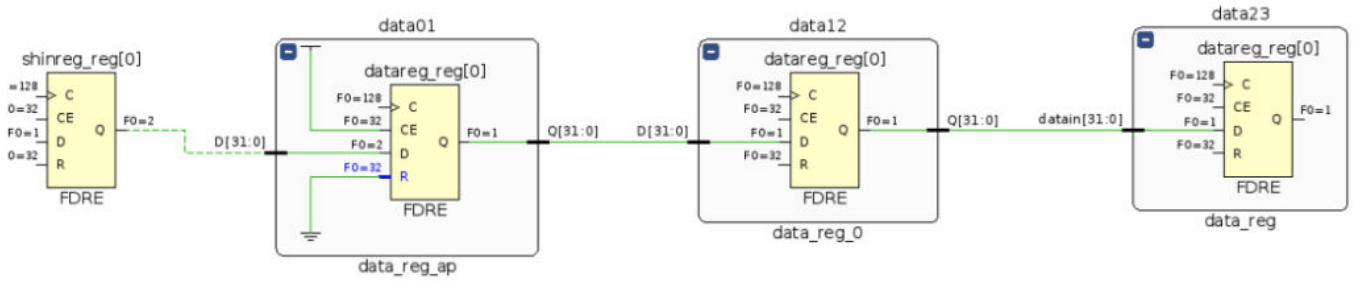
基于目标架构，如果使用足够的流水线，专用原语（如块 RAM 和 DSP）可在 500 MHz 以上的频率范围工作。对于高频设计，AMD 建议在这些块中使用所有流水线。

自动流水打拍注意事项

自动流水打拍功能支持布局器判断所需流水打拍阶段数量及其最佳位置，从而帮助跨接口边界实现时序收敛。您可通过设置 AXI Register Slice 核的自动流水打拍功能或者通过为数据总线应用自动流水打拍 HDL 属性或 XDC 约束来启用该功能。由于插入受时序驱动，因此请务必始终确保在目标路径上应用正确的时序约束。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

以下示例显示了模块 data01 与 data12 之间的接口上应用的自动流水打拍。data01 的输出由不含控制集的寄存器组成。

图 37：**模块间的简单数据流连接**



此示例的 RTL 代码如下所示。`autopipeline_module` 属性应用于分层模块 data01 上，而 `autopipeline_group`/`autopipeline_limit`/`autopipeline_include` 属性则应用于由寄存器的 Q 管脚直接驱动的信号线上。

```
data_reg_ap #( .C_DATA_WIDTH(C_DATA_WIDTH)) data01 (
.clk (clk),
.datain (shinreg),
.datareg (d1)
);

data_reg #( .C_DATA_WIDTH(C_DATA_WIDTH)) data12 (
.clk (clk),
.datain (d1),
.datareg (d2)
);

(* autopipeline_module="yes" *)
module data_reg_ap # (
parameter integer C_DATA_WIDTH = 32
)
( input wire clk,
input wire [C_DATA_WIDTH-1:0] datain,
(* autopipeline_group="fwd",autopipeline_limit=24 *)
output reg [C_DATA_WIDTH-1:0] datareg
);
```

```
always @(posedge clk) begin
datareg <= datain;
end
endmodule
```

此示例提供了在 RTL 代码中使用上述属性的另一种替代方式，其 XDC 约束如下所示。

```
# It's suggested to add the USER_SLR_ASSIGNMENT property at the module
#level to ensure better logic clustering with its driver and load, see
UG912
#for more details on this property
set_property USER_SLR_ASSIGNMENT APSRC [get_cells data01]
set_property USER_SLR_ASSIGNMENT APDST [get_cells data12]

set_property AUTOPIPELINE_MODULE TRUE [get_cells data01]
set_property AUTOPIPELINE_GROUP WBUS [get_nets -of [get_pins -filter
REF_PIN_NAME==Q -of [get_cells data01/*]]]
set_property AUTOPIPELINE_LIMIT 10 [get_nets -of [get_pins -filter
REF_PIN_NAME==Q -of [get_cells data01/*]]]
```

改善功耗的编码样式

时钟或数据路径门控

不使用时钟路径或数据路径的结果时，常用的停止转换的方法是对这些路径进行门控。对时钟进行门控可停止所有受驱动的同步负载，并阻止数据路径信号开关和毛刺继续进行传输。

功耗最优化 (`power_opt_design`) 可自动生成信号门控逻辑，以减少开关活动。但您可获得有关应用、数据流和依赖关系的信息，这些信息对于工具不可用并且只有您才能指定。

最大限度增加门控元件数量

最大限度增加受门控信号影响的元件数量。例如，在驱动源位置对时钟域进行门控相比使用时钟使能信号对每个负载进行门控更节省功耗。

使用专用时钟缓冲器的时钟使能管脚

在通过对时钟进行门控或多路复用以最大限度减少活动或降低时钟树使用率时，请使用专用时钟缓冲器的时钟使能端口。无论是插入 LUT 还是使用其他方法通过门控关闭时钟信号，对于满足功耗和时序要求的效果都不理想。

无需优先级编码器时使用 Case 块

不需要进行优先级编码时，请使用 case 块代替 if-then-else 块或三元运算符。

低效率编码示例：

```
if (reg1)
  val = reg_in1;
else if (reg2)
   val = reg_in2;
else if (reg3)
   val = reg_in3;
else val = reg_in4;
```

正确编码示例：

```
(* parallel_case *) casex ({reg1, reg2, reg3})
1xx: val = reg_in1 ;
01x: val = reg_in2 ;
001: val = reg_in3 ;
default: val = reg_in4 ;
endcase
```

块 RAM 的性能/功耗利弊取舍

有多种方法可用于细分存储器配置以满足具体要求。特定器件的要求可包括时钟频率、功耗或两者混合。

以下示例着重演示了为满足您的要求而可生成的各种不同结构。综合可使用 CASCADE_HEIGHT 属性来限制块 RAM 级联以实现时钟频率与功耗之间的利弊取舍。如需了解该属性的用法和实参，请参阅《Vivado Design Suite 用户指南：综合》(UG901)。

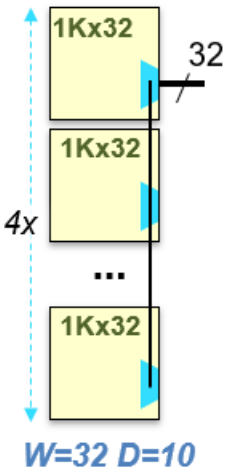
下图显示了为实现更高时钟频率（时序）的 4Kx32 存储器配置的示例。

图 38：**使用 4Kx8 和 CASCADE_HEIGHT=1 实现 4Kx32 的 RTL 表达形式**

在此实现中，所有块 RAM 都始终处于启用状态（针对每次读取或写入）并耗用更多功耗。

下图显示了级联所有块 RAM 以降低功耗的示例。

图 39：**使用 1Kx32 和 CASCADE_HEIGHT=4 实现 4Kx32 的 RTL 表达形式**

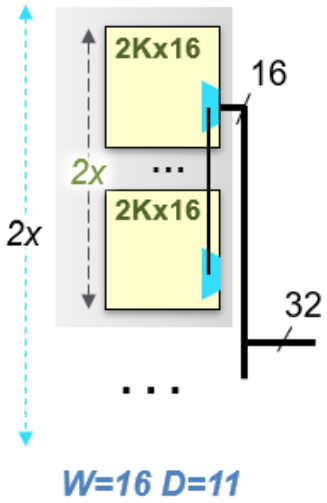



```
module test(
input clk,
input we,
input [31:0] din,
input [11:0 ] addr,
output reg [31:0] dout
);

(* ram_style = "block",cascade_height = 4 *)
reg [31:0] mem [(2**12)-1:0];
reg [11:0] addr_reg;

always @(posedge clk)
begin
   addr_reg <= addr;
   dout <= mem[addr_reg];
   if (we)
      mem[addr_reg] <= din;
end

endmodule
```

在此实现中，由于每次仅选中 1 个块 RAM（从每个单元中），因此动态功耗贡献几乎减半。块 RAM 具有专用级联 MUX 和布线结构，支持构建宽而深的存储器，此类存储器需要在功耗效率极高的配置中构建多个块 RAM 原语。

下图显示了如何限制级联并同时保障功耗和时钟频率（通常无需牺牲性能）的示例。

图 40：**使用 2Kx16 和 CASCADE_HEIGHT=2 实现 4Kx32 的 RTL 表达形式**




```
module test(
input clk,
input we,
input [31:0] din,
input [11:0 ] addr,
output reg [31:0] dout
);

(* ram_style = "block",cascade_height = 2 *)
reg [31:0] mem [(2**12)-1:0];
reg [11:0] addr_reg;

always @(posedge clk)
begin
   addr_reg <= addr;
   dout <= mem[addr_reg];
   if (we)
      mem[addr_reg] <= din;
end

endmodule
```

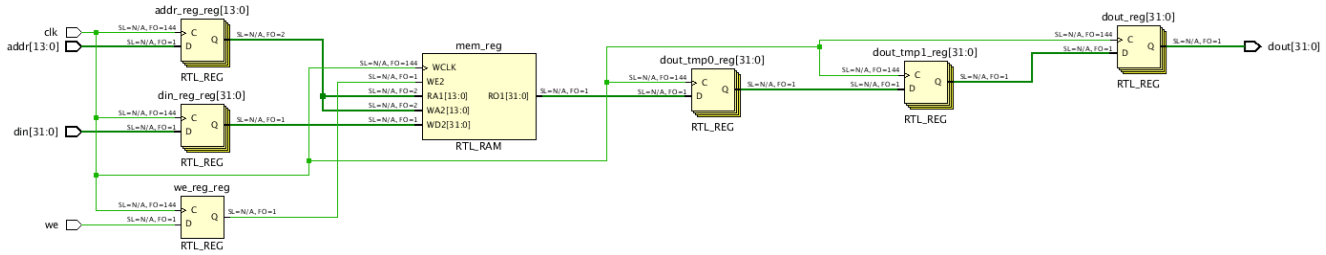
在此实现中，由于每次仅选中 2 个块 RAM，因此动态功耗贡献比高时钟频率结构更好，但不及低功耗结构。此结构相比于低功耗结构的优势在于它在级联路径中仅使用 2 个块 RAM，而相比于低功耗结构的关键路径中的 4 个块 RAM，这将影响目标频率。

分解更深的存储器配置，实现功耗与时钟频率平衡

在使用更深的存储器配置时，可在 RTL 中使用 RAM_DECOMP 综合属性，通过改进存储器组合来降低功耗。当 RAM_DECOMP 属性被应用到存储器阵列上时，存储器逻辑将映射到更宽的块 RAM 原语阵列上。为平衡功耗与时钟频率，您可以使用 CASCADE_HEIGHT 属性和 RAM_DECOMP 属性来控制级联。这种方法需要更多的地址解码逻辑，但有助于减少为每个读取操作启用的块 RAM 数量，从而帮助降低功耗。

例如，下图显示的是 1 个 32x16K 存储器配置。

图 41：**32x16K 存储器配置**



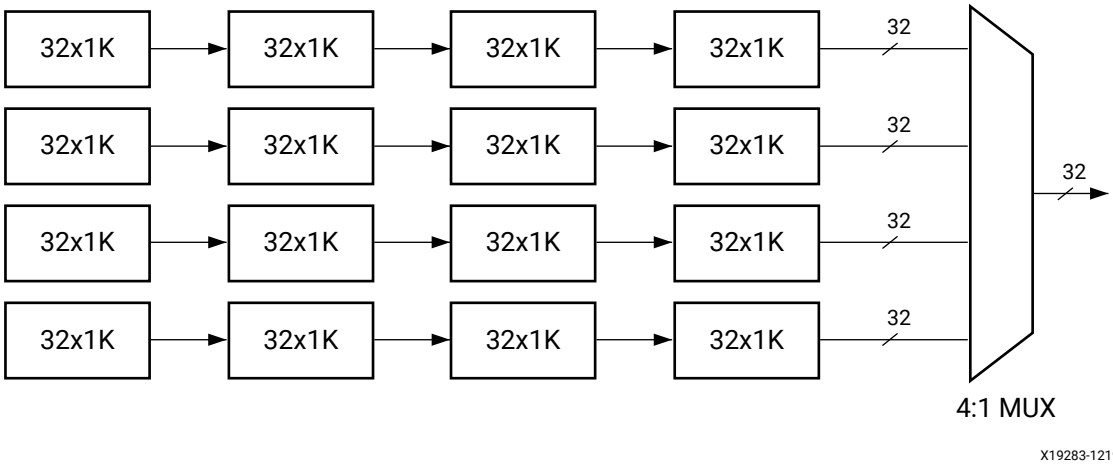
如果您应用下列属性：

```
ram_decomp = "power"
cascade_height = 4
```

将推断得出 16 个 RAMB36E2 且存储器分解方式如下：

- 基本原语为 32x1K。
- 4 个块 RAM 将通过级联创建 32x4K 配置。
- 4 个并行结构可创建 1 个 16K 深的存储器。
- 输出通过多路复用来生成输出数据。

图 42：**使用 CASCADE_HEIGHT 和 RAM_DECOMP 属性生成的 32x16K 存储器配置结构示例**



以下 RTL 代码示例显示了 CASCADE_HEIGHT 和 RAM_DECOMP 属性的用例。

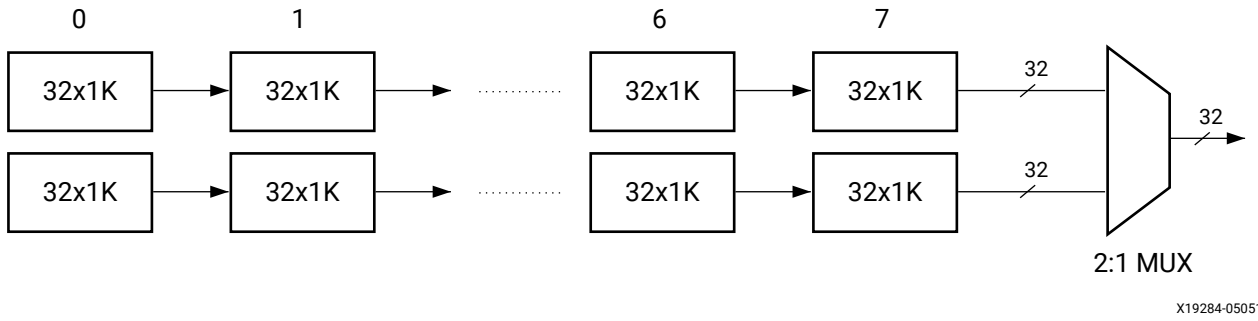
图 43：**使用 CASCADE_HEIGHT 和 RAM_DECOMP 属性的 32x16K 存储器配置的 RTL 代码**

```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power", cascade_height = 4 *) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    din_reg  <= din;
    we_reg   <= we;
    dout_tmp0 <= mem[addr_reg];
    dout_tmp1 <= dout_tmp0;
    dout <= dout_tmp1;
    if (we_reg)
        mem[addr_reg] <= din_reg;
end

endmodule
```

如果仅应用 `ram_decomp = "power"` 属性，那么将推断得出 16 个 RAMB36E2 并且存储器分解方式如下：

- 基本原语为 32x1K。
- 8 个块 RAM 将通过级联创建 32x8K 配置。
- 2 个并行结构可创建 1 个 16K 深的存储器。
- 输出通过多路复用生成 2:1 MUX，以生成输出数据。

图 44：**使用 RAM_DECOMP 属性生成的 32x16K 存储器配置的结构**



以下 RTL 代码示例显示了 RAM_DECOMP 属性的用例。

图 45：**使用 RAM_DECOMP 属性的 32x16K 存储器配置的 RTL 代码**

```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power"*) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    din_reg  <= din;
    we_reg   <= we;
    dout_tmp0 <= mem[addr_reg];
    dout_tmp1 <= dout_tmp0;
    dout <= dout_tmp1;
    if (we_reg)
        mem[addr_reg] <= din_reg;
end

endmodule
```

如果仅使用 RAM_DECOMP 属性，那么总体功耗节省与同时使用 RAM_DECOMP 和 CASCADE_HEIGHT 属性相似，因为每次只有 1 个块 RAM 处于活动状态。为了实现最高时钟频率，创建深度为 4 的级联块 RAM 链比深度为 8 的级联块 RAM 链效果更好。

欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

运行 RTL DRC

有一组 RTL DRC 规则用于识别 HDL 潜在的编码问题。您可单击 Flow Navigator 下的“Open Elaborated Design”便可打开细化视图并进行检查。您可在 Flow Navigator 中选中“RTL Analysis” → “Report Methodology”（RTL 分析 > 方法论报告）或者在 Tcl 命令提示符处执行 `report_methodology`，以运行这些 DRC 检查。

FIFO 编码

Versal 架构不包含硬化的 FIFO。在 Versal 器件中处理 FIFO 时，请使用以下任一方法：

- RTL：使用 XPM_FIFO。Vivado IDE 语言模板提供了 XPM_FIFO 例化。
- IP integrator：例化相应的 AMD FIFO IP。

为基于平台的设计流程创建和封装 RTL 内核的编码建议

适用于 Versal 自适应 SoC 的基于平台的设计流程能够方便用户使用现有 RTL 代码或 Vivado IP 作为 Vitis 内核。Vitis 内核均为规范化的设计块，可使用 Vitis v++ 连接器以自动建构校正方式自动集成到现有平台内。

内核类型

以下提供了不同类型的 Vitis 内核，欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容：

- 软件可控内核：软件可控内核会公开一个可编程的寄存器接口，以允许主机软件应用通过寄存器读取和写入来与内核进行交互。这些都属于最常见且适用最广泛的内核类型。有 2 种类型的软件可控内核：用户管理的内核和 XRT 管理的内核。用户管理的内核的接口要求比 XRT 管理的内核的接口要求更少，并且是使用 RTL 代码或 Vivado IP 块创建 PL 内核的设计师最常用的选择。
- 非软件控制的内核：这些内核存在于器件中，但对于软件应用不可见或者不可供软件应用直接访问。这些内核没有可编程寄存器接口。这些内核必须具有至少一个 AXI4-Stream 接口。内核通过这些串流传输接口与系统其余部分进行同步。

内核接口

AMD 强烈建议使用以下接口创建 PL 内核：

- 至少一个时钟和一个复位
- 标准 AXI 接口：
  - AXI4 存储器映射接口，用于通过 NoC 执行存储器映射传输
  - AXI4-Stream，用于连接到其他 PL 内核、AI 引擎计算图或串流平台端口
  - AXI4-Lite，供 PS 控制（仅限软件可控内核）

欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

处理标准 AXI 接口支持通过 Vitis 工具流程来实现设计自动化，减少人工错误以及易于出错的任务。但 PL 内核也可能包含非 AXI 接口。在此情况下，您需要在 Vitis 链接阶段中使用 `connectivity.connect` 选项手动显式连接每个非 AXI 信号，欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的描述。

RTL 内核的设计建议

第 4 章：使用 RTL 创建设计 中所述的所有建议都同样适用于 RTL 内核。

封装 RTL 内核

您可使用 Vitis IP 封装器从 RTL 代码或 Vivado IP 块来创建 Vitis PL 内核，欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

内核调试和验证注意事项

以下是调试和验证内核时的建议：

- 使用先进的验证技术（包括验证组件、随机化和协议检查器）在各 RTL 内核自己的测试激励文件中对其执行验证。Vivado IP 目录提供 AXI Verification IP (VIP)，可帮助验证 AXI 接口。RTL 内核设计示例包含基于 AXI VIP 的测试激励文件，其中带有样本激励文件。
- 使用硬件仿真来测试主机代码软件集成或查看多个内核之间的交互。

使用逻辑仿真来验证 RTL 内核属于块级任务。使用硬件仿真来验证 RTL 内核则属于系统集成任务。如需了解有关 Versal 自适应 SoC 的逻辑仿真和硬件仿真流程的更多信息，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相关内容。

时钟设置准则

每个器件架构都有用于时钟设置的专用资源。掌握器件架构中的时钟设置资源将使您能够规划好自己的时钟设置，从而实现时钟资源的最佳利用。大多数设计无需您了解这些细节。但如果您能够控制布局，同时熟悉每个时钟域上的扇出，就可以根据以下时钟设置详情，研究出多种备选方案。如果您决定使用任何时钟设置资源，就需要显式例化相应的时钟元件。

Versal 器件时钟设置

Versal 器件的时钟结构与 AMD UltraScale™ 器件相似，在整个器件中使用全局时钟设置，但负载可采用局部或全局布局。Versal 器件中的多时钟缓冲器 (MBUFG) 原语支持叶级时钟分频，以降低时钟轨道使用率，并改进同步时钟域交汇上的时序约束。此架构有助于提升时钟资源使用率的效率、支持更多数量的设计时钟，并改进时钟的性能和功耗特性。

在 Versal 器件中，时钟通常源自列式 HDIO bank 或水平高性能 XPIO bank。每个 Versal 器件时钟区域都包含如下时钟网络布线：可分频至 24 个垂直布线轨道、12 个水平布线轨道、24 个垂直分布轨道和 24 个水平分布轨道。底部时钟区域行是唯一的，并且包含 24 个水平布线轨道（相比之下，其他时钟区域行中包含 12 个水平布线轨道）。

图 46：**Versal 器件时钟布线架构**

X24352-081220

以下是时钟类型的主要类别及其关联的时钟结构（按其驱动和用途分组）：

- 高速 I/O 时钟

  这些时钟与 SelectIO™ XPHY 逻辑关联，并由 XPLL 生成。这些时钟通过专用低抖动资源从 XPLL 布线到 XPHY 逻辑，以供高性能 I/O 接口使用。通常，此时钟结构由 AMD IP（例如，NoC IP DDR4 存储器控制器、软核存储器控制器 IP 或 Advanced IO Wizard IP）来控制。

- 通用时钟

  这些时钟可在大部分时钟树结构中使用，可作为 GCIO 封装管脚或时钟修改块（例如，MMCM、XPLL 或 DPLL）的来源。通用时钟网络必须由支持叶级时钟分频的典型 BUFGCE/BUFGCE_DIV/BUFGCTRL 缓冲器或者新 MBUFGCE/MBUFGCE_DIV/MBUFGCTRL 原语驱动。在 HDIO bank 中，通过仅提供 DPLL 和 BUFGCE，对时钟设置资源加以限制。任意给定时钟区域均可支持最多 24 个独立时钟，Versal 器件根据其拓扑结构、扇出和负载布局，可支持 100 余个时钟树。

- 千兆位收发器时钟

  千兆位收发器 (GT*_QUAD) 的发射、接收和参考时钟均使用包含 GT 的时钟区域内的专用时钟。在 Versal 器件中，GT 时钟列包含 DPLL，并且还支持通过新 MBUFG_GT 原语进行叶级时钟分频。您可以使用 GT 时钟来实现以下功能：

  - 驱动 DPLL 进行频率综合、抖动过滤或时钟纠偏
  - 使用 BUFG_GT 或 MBUFG_GT 缓冲器连接到互连结构中的任何负载，以驱动通用时钟网络
  - 在相同或不同四通道 (Quad) 中的多个收发器上共享时钟

  注释：只有 12 条偶数时钟布线和时钟分布轨道跨过 GT 列中的 SLR 边界。

  注释：没有任何时钟布线和时钟分布轨道会跨越 GT 列中的 SLR 边界以往返 AMD Versal™ HBM 器件顶层 SLR。如果负载分布在顶层 SLR 与其他 SLR 之间，那么布局器会忽略 GT 列中的 USER_CLOCK_ROOT。

时钟原语

大部分时钟通过支持全局时钟的 I/O (GCIO) 管脚进入器件。在水平 XPIO bank 中，这些时钟通过全局时钟缓冲器直接驱动时钟网络，或者由位于 XPIO bank 的时钟管理模块 (CMT) 中的 MMCM、XPLL 或 DPLL 进行变换。对于含列式 HDIO bank 的器件，时钟通过全局时钟缓冲器来驱动时钟网络，或者由位于 HDIO bank 的 CMT 中的 DPLL 进行变换（前提是器件具备该功能）。

每个水平 XPIO bank 包含以下时钟资源：

- 时钟生成块
  - 1 个 MCMM
  - 2 个 XPLL
  - 1 个 DPLL
- 全局时钟缓冲器
  - 24 个 BUFGCE/MBUFGCE
  - 8 个 BUFGCTRL/MBUFGCTRL
  - 4 个 BUFGCE_DIV/MBUFGCE_DIV

注释：位于 Versal 器件角点的 XPIO bank 中的时钟资源受到限制，并且具有不可访问的资源，例如，BUFGCTRL 和 BUFGCE_DIV。要将 BUFGCTRL 和 BUFGCE_DIV 用于源自角点的 bank 的时钟，可能需要使用级联时钟拓扑结构，如级联时钟缓冲器 中所述。您可使用具有未绑定的 I/O 的 XPIO bank 中的时钟资源。如需了解有关 XPIO 角点 bank 限制的更多信息，请访问此链接以参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003) 中的相关内容。

对于含列式 HDIO bank 的器件，每个 bank 都包含以下时钟资源：

- 时钟生成块
  - 1 个 DPLL
- 全局时钟缓冲器
  - 4 个 BUFGCE/MBUFGCE

注释：在 VC1902、VC1802 和 VM1802 器件的 HDIO bank 中，DPLL 不可用。

每个千兆位收发器 (GT*_QUAD) 时钟区域列都包含以下时钟资源：

- 时钟生成块
  - 1 个 DPLL
- 全局时钟缓冲器
  - 24 个 BUFG_GT/MBUFG_GT

下表提供了 Versal 器件时钟缓冲器的汇总信息。

表 5：**Versal 器件时钟缓冲器**

Versal 器件时钟缓冲器	支持叶级时钟分频	位置	描述
BUFGCE	支持，需使用 MBUFGCE	XPIO bank 和 HDIO bank	最常用的缓冲器是 BUFGCE，它是通用时钟缓冲器，具有启用/禁用时钟的功能。
BUFGCE_DIV	支持，需使用 MBUFGCE_DIV	XPIO bank	可在以下情况下使用 BUFGCE_DIV：需要对时钟进行简单分频时。相比于使用 MMCM 或 PLL 进行简单时钟分频，此时钟缓冲器更便于使用并且能效更高。
BUFGCTRL	支持，需使用 MBUFGCTRL	XPIO bank	BUFGCTRL 可例化为 BUFGMUX，通常可在以下情况下使用：对 2 个或更多个时钟源进行多路复用以构成单个时钟网络时。就像 BUFGCE 和 BUFGCE_DIV 一样，该时钟缓冲器可驱动时钟网络用于区域时钟设置或全局时钟设置。
BUFG_GT	支持，需使用 MBUFG_GT	GT*_QUAD 列	使用由 GT 生成的时钟时，BUFG_GT 时钟缓冲器允许连接至全局时钟网络。大多数情况下，BUFG_GT 用作为区域缓冲器，其负载布局在 1 或 2 个相邻时钟区域内。BUFG_GT 内置动态时钟分频功能，可替代 MMCM 执行时钟速率更改。
BUFG_PS	支持，需使用 MBUFG_PS	与 PS 相邻的垂直时钟列	BUFG_PS 是简单的时钟缓冲器，包含 1 个时钟输入 (I) 和 1 个时钟输出 (O)。此时钟缓冲器作为 PS 的资源，可为时钟提供 PL 时钟布线资源，用于从处理器布线到 PL 内。可用 BUFG_PS 缓冲器的最大数量为 12。
BUFG_FABRIC	不支持	NoC 列	BUFG_FABRIC 由 PL 驱动，用于高扇出非时钟信号线的布线，支持将信号从 PL 布线资源引入时钟网络。但该时钟缓冲器不可用于全局时钟设置。

多时钟缓冲器 (MBUFG)

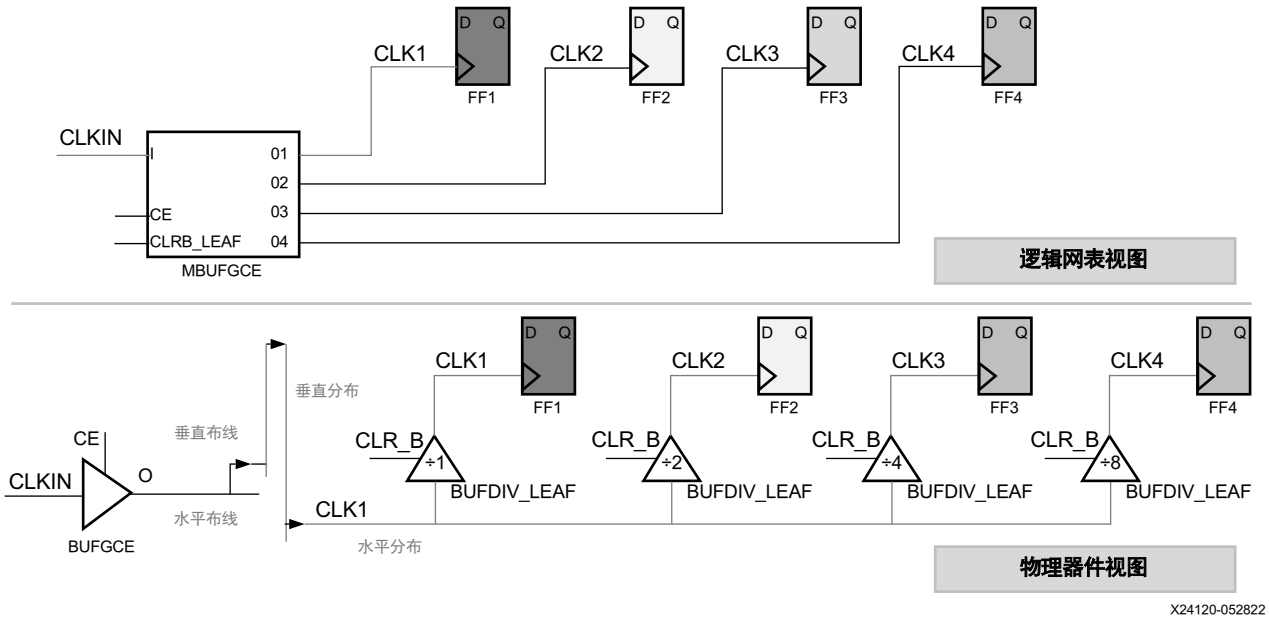
MBUFG 属于时钟原语，支持您利用叶级时钟分频器，这些分频器由 Versal 器件中的局部水平时钟分布轨道来驱动。叶级分频仅使用单一全局时钟布线资源，因而会导致降低时钟轨道资源使用率、提升能效并降低同步时钟域之间的偏差。MBUFG 原语（MBUFGCE、MBUFGCE_DIV、MBUFG_GT、MBUFGCTRL 和 MBUFG_PS）包含 4 项输出（O1、O2、O3 和 O4），分别用于配置时钟分频器设置 1、2、4 和 8，对于用于驱动连接到 MBUFG 原语的时钟负载的任意叶时钟分频器，这些设置都适用。如需了解有关如何使用 MBUFG 原语来帮助降低同步时钟域之间过高的偏差的更多信息，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

要在设计中使用 MBUFG 原语来代替标准 BUFG 原语，请在运行 Versal 自适应 SoC Clocking Wizard 时选中 MBUFG 原语。仅当输出频率为彼此间呈倍数关系（2、4 或 8 倍）时，MBUFG 原语才可用。欲知详情，请访问此链接以参阅《适用于 Versal 自适应 SoC 的 Clocking Wizard LogiCORE IP 产品指南》(PG321) 中的相应内容。

注释：当 BUFG 原语在设计中已例化时，您可使用逻辑最优化 (`opt_design`) 来将部分 BUFG 原语转换为 MBUFG 原语。使用 `opt_design` 进行转换仅限在某些情况下才可行且存在一些限制。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：实现》(UG904)。

下图显示了 MBUFG 驱动的时钟网络的逻辑和物理实现视图。BUFDIV_LEAF 原语不显示在逻辑信号线中，仅用作为 Vivado 布线器所配置的直通式布线原语。MBUFG 上的 CLRB_LEAF 输入可用于异步复位 BUFDIV_LEAF 分频器。连接到 MBUFG CLRB_LEAF 管脚的信号将自动布线到其连接的 BUFDIV_LEAF CLR_B 管脚。由于在全局时钟布线和分布轨道上仅布线 1 个时钟，因此 MBUFG 驱动的时钟可保留时钟资源。此外，由同一 MBUFG 的 2 个输出时钟负责时钟设置的路径的公用节点通常距离驱动程序和负载更近，从而降低时钟偏差并简化时序收敛。

图 47：**MBUFGCE 逻辑和物理视图**



在器件启动时，BUFDIV_LEAF 时钟分频器会复位，并且 MBUFG 输出时钟以“High”（高电平）状态启动。在下列情况下，需要特殊处理以确保先将 BUFDIV_LEAF 分频器复位到其启动状态，然后 MBUFG 才会接收到输入时钟或重新启用：

- MBUFG 时钟缓冲器或驱动 MBUFG 的时钟修改块在器件操作期间复位
- 器件启动后，驱动 MBUFG 的时钟违反最小脉冲宽度规格（即，时钟修改块未锁定）
- MBUFG 驱动的时钟网络属于可重配置分区的一部分

要复位 BUFDIV_LEAF 缓冲器，MBUFG 的 CLRB_LEAF 管脚必须断言为低电平有效。为确保器件正常操作，用户逻辑必须在断言 CLRB_LEAF 管脚为低电平有效之前先停止 MBUFG 时钟，并在 CLRB_LEAF 信号断言为高电平无效之后，使该时钟在预定义的时间范围内保持无效状态。当 CLRB_LEAF 信号断言无效之后，该时钟保持无效状态的时间必须比布线器所报告的下列时间更长：用于将已连接到 MBUFG CLRB_LEAF 管脚的信号布线到 BUFDIV_LEAF CLR_B 管脚的最大管脚延迟时间。布线器会在 INFO 消息中报告此时间，如以下示例所示。大多数情况下，10 ns 延迟足以满足 CLRB_LEAF 信号线布线延迟。

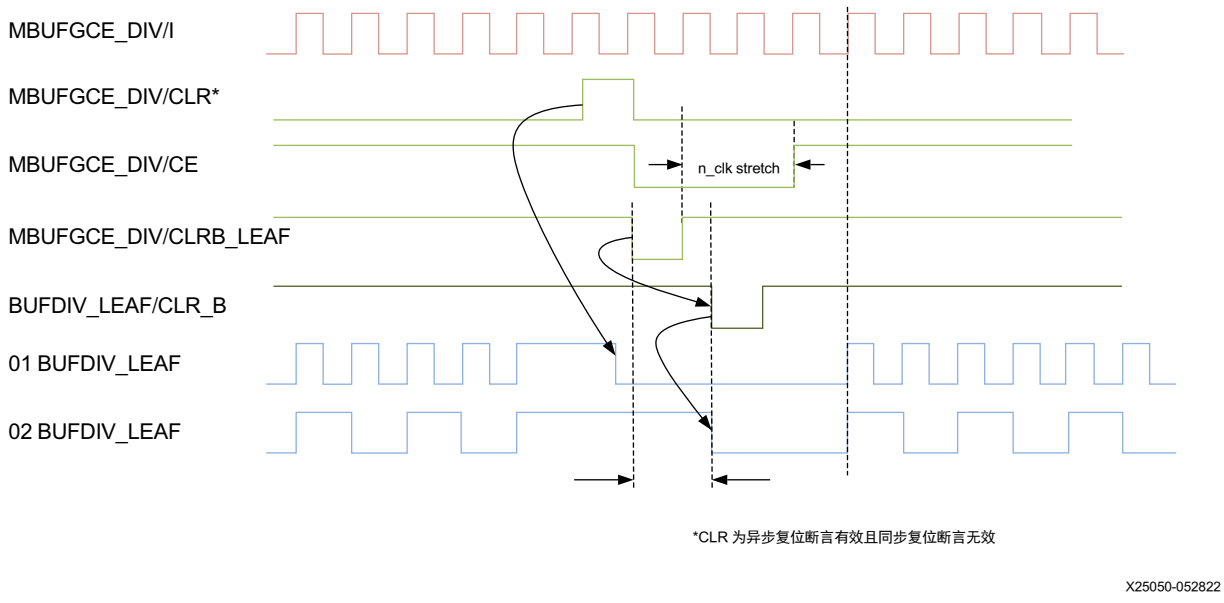
```
INFO: [Route 35-3345] MBUFG*/CLRB_LEAF net route delay summary. Please
ensure that the wait time between de-asserting the CLRB_LEAF signal to
each MBUFG and enabling the MBUFG output clocks is greater than the
delay listed in the table below.
+------------------+-------------+---------------------+---------------------+
| MBUFG Cell       | Site        | CLRB_LEAF Net Name  | Max Pin Delay (ns)  |
+------------------+-------------+---------------------+---------------------+
| U_engine/bufg_fx | BUFGCE_X3Y0 | U_engine/p_1_out    | 2.252               |
+------------------+-------------+---------------------+---------------------+
```

使用 Clocking Wizard IP 时，会自动添加满足此复位要求的电路。如需了解更多信息，请参阅《适用于 Versal 自适应 SoC 的 Clocking Wizard LogiCORE IP 产品指南》(PG321)。

下图显示了 CLR 断言有效后，MBUFGCE_DIV CLRB_LEAF 与 CE 信号断言有效之间所需的时序关系。CE 信号保持低电平 (Low) 以停止该时钟，直至 CLRB_LEAF 信号被传输到所有 BUFDIV_LEAF CLR_B 管脚为止。在此示例中，MBUFGCE_DIV 的 CE_TYPE 属性设置为 SYNC。

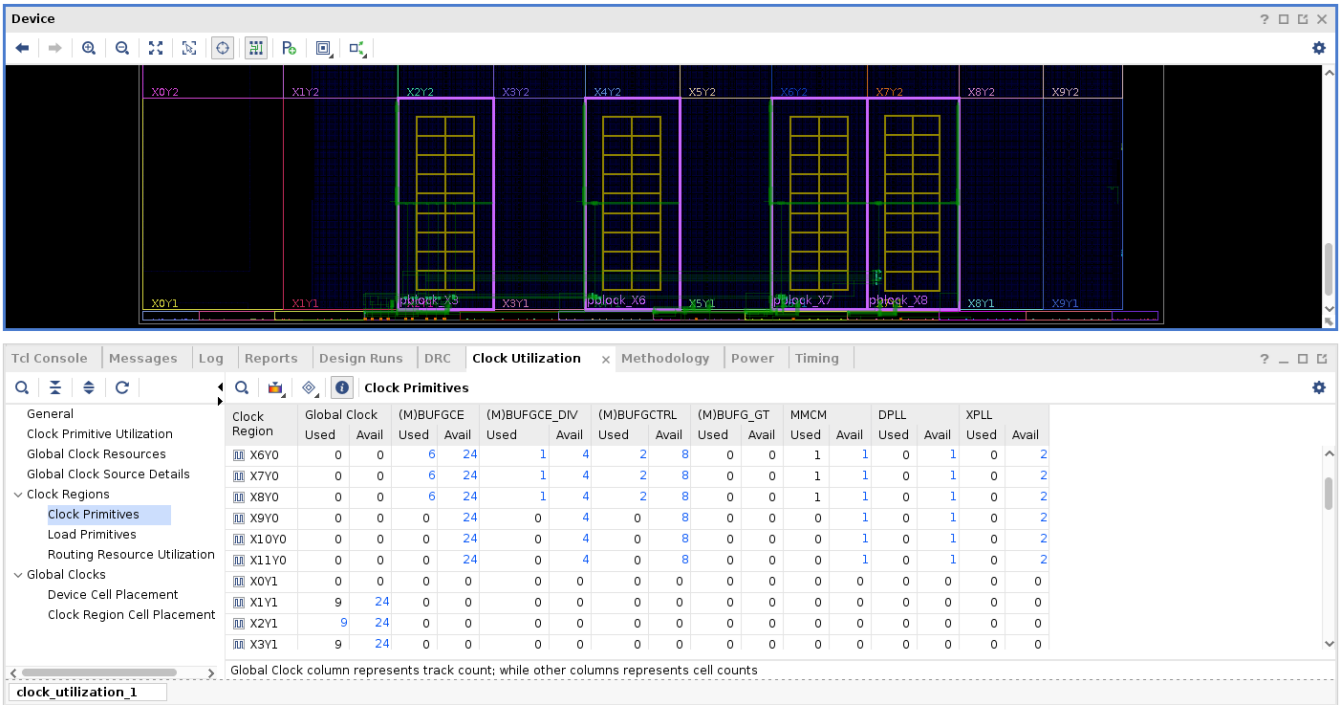
注释：在下图中，MBUFGCE_DIV/CLRB_LEAF 到 BUFDIV_LEAF/CLR_B 的传输时间是因布线而产生的。

图 48：**在 MBUFGCE_DIV 上断言 CLR 有效时，断言 CLRB_LEAF 信号有效与断言 CE 信号有效之间的时序关系**



您可以使用 Vivado IDE 中的“Clock Utilization”（时钟使用率）报告来以可视化方式分析时钟资源使用率和时钟布线。下图显示了在“Device”（器件）窗口中叠加的时钟资源使用率（按时钟区域）。如需了解有关此报告的更多信息，请参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906)。

图 49：**“Clock Utilization”报告**



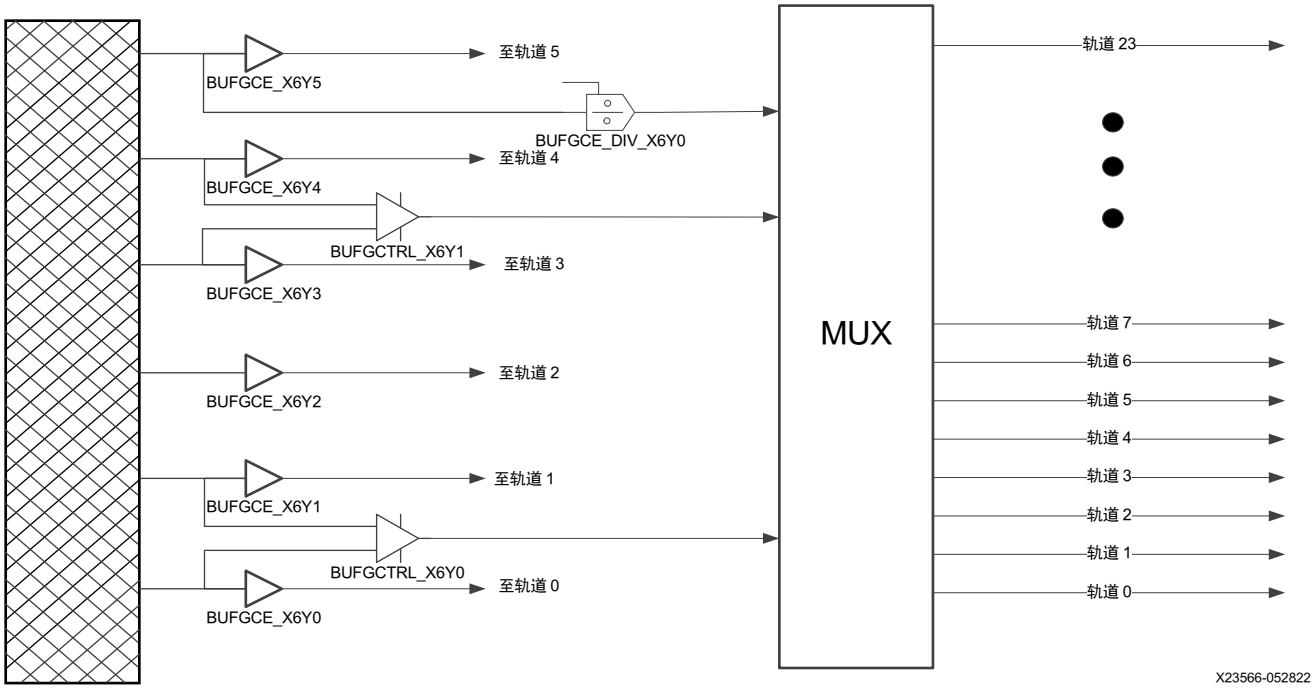
如需了解有关 BUFGCE、MBUFGCE、BUFGCE_DIV、BUFGCTRL 和 MBUFGCTRL 缓冲器的更多信息，请参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003)。如需了解有关 BUFG_GT 和 MBUFG_GT 缓冲器的连接和使用的详细信息，请参阅《Versal 自适应 SoC GTY 和 GTYP 收发器架构手册》(AM002) 和《Versal 自适应 SoC GTM 收发器架构手册》(AM017)。

全局时钟缓冲器连接和布线轨道

XPIO 时钟区域中的全部 24 个 BUFGCE/MBUFGCE 缓冲器各自都只能驱动特定的时钟布线轨道。然而，BUFGCTRL/MBUFGCTRL 和 BUFGCE_DIV/MBUFGCE_DIV 输出可以通过 MUX 结构使用 24 条轨道中的任何一条。每个 BUFGCE_DIV/MBUFGCE_DIV 与特定的 BUFGCE 站点 (site) 共享输入连接，每个 BUFGCTRL/MBUFGCTRL 与 2 个特定的 BUFGCE 站点 (site) 共享输入连接。因此，当时钟区域中使用 BUFGCE_DIV/MBUFGCE_DIV 或 BUFGCTRL/MBUFGCTRL 缓冲器时，BUFGCE/MBUFGCE 缓冲器的使用会受到限制。下图显示了在时钟区域内的底部 6 个 BUFGCE 站点，这些 BUFGCE 在 XPIO 时钟区域内复制 4 次。

注释：针对器件中每个特定轨道 ID 都会分配 1 个全局时钟信号线，以供时钟所使用的所有垂直布线、水平布线和分布资源使用。除非时钟穿过另一个时钟缓冲器，否则无法更改轨道 ID。

图 50：**BUFGCE、BUFGCE_DIV 和 BUFGCTRL 共享输入和输出多路复用**



注释：含 X6Y# 的 BUFG* 站点 (site) 表示 XPIO CLOCK_REGION X6Y0。

时钟布线、时钟根和时钟分布

要正确理解 Versal 器件的时钟容量以及设计的时钟使用率，重要的是了解时钟布线使用专用布线资源的方式：

- 从时钟缓冲器到时钟根，时钟信号经过 1 个或多个垂直和水平布线阶段。每个分段都必须使用相同的轨道 ID（介于 0 和 23 之间）。
- 在时钟根处，时钟信号从布线轨道转变为具有相同轨道 ID 的分布轨道。为了减少偏差，时钟根通常在位于时钟扩展窗口中心的时钟区域中。时钟扩展窗口 (clock expansion window) 是包含布局时钟信号线负载的所有时钟区域的矩形区域。通过读取只读 CLK_EXPANSION_WINDOW 属性即可判定时钟信号线的时钟扩展窗口。下面给出 1 个示例：

```
get_property CLK_EXPANSION_WINDOW [get_nets -of [get_pins BUFGCE_inst/O]]
=> CLOCKREGION_X5Y2:CLOCKREGION_X6Y3
```

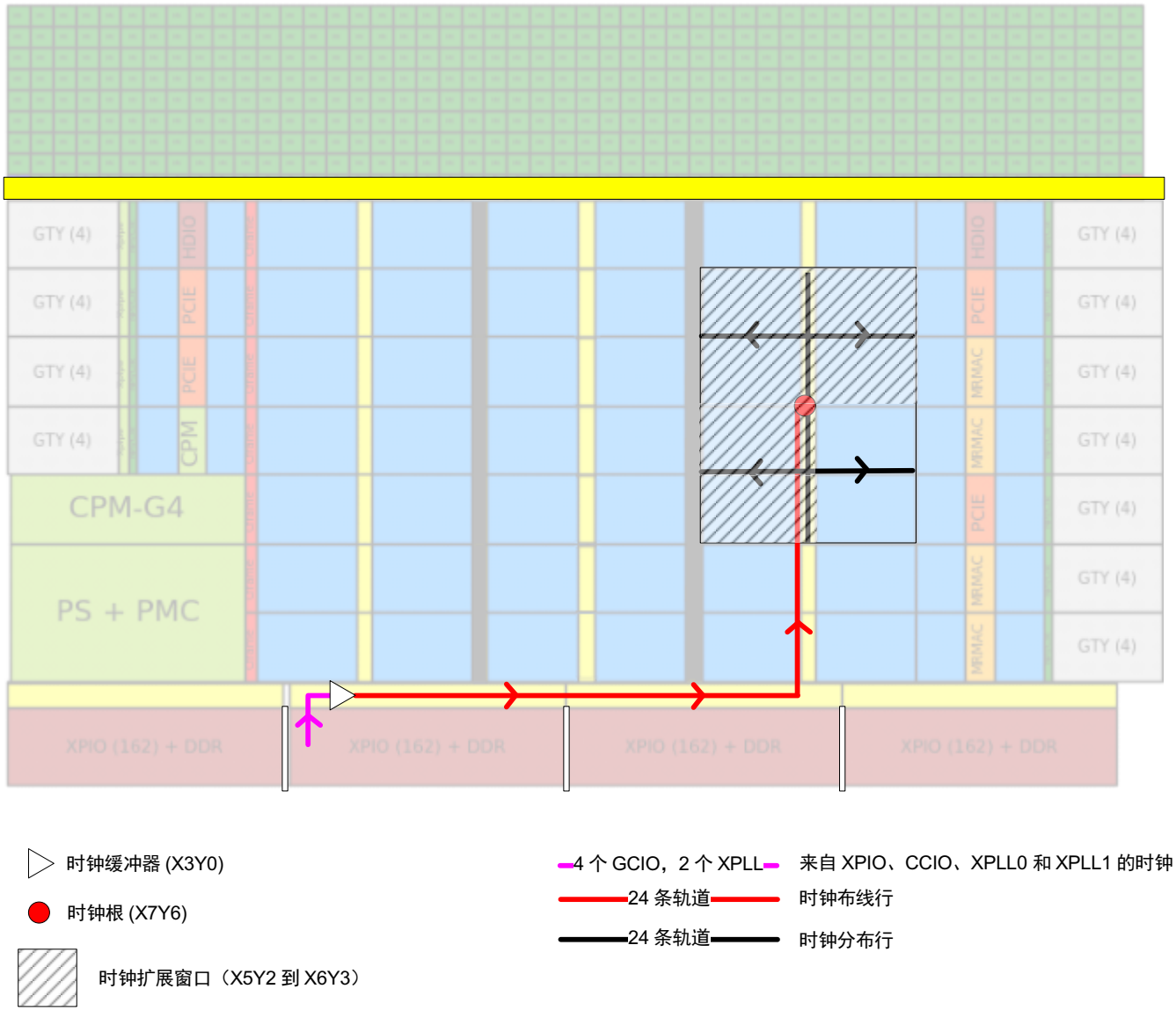
- 从时钟根到负载所在的 CLB 列，时钟信号垂直分布（根据需要沿器件向上和向下）方向移动，然后沿水平分布（根据需要向左和向右）移动。
- CLB 列分成两半，分别位于水平分布资源的上方和下方。每一半 CLB 列都包含几个叶时钟布线资源，可以通过任何水平分布轨道来获取。Versal 器件中的叶时钟布线资源允许 MBUFG* 原语执行叶级时钟分频，以降低时钟轨道资源使用率、提升功耗效率并降低同步时钟域之间的偏差。

在某些情况下，时钟缓冲器可以直接驱动到时钟分布轨道上。当时钟根与时钟缓冲器位于相同时钟区域中或者当时钟缓冲器仅驱动非时钟管脚（例如，高扇出信号线）时，通常会发生此情况。

由于时钟布线资源已分段，因此所耗用的资源仅限于用于在整个时钟扩展窗口内遍历时钟区域和执行时钟分布的布线段和分布段。

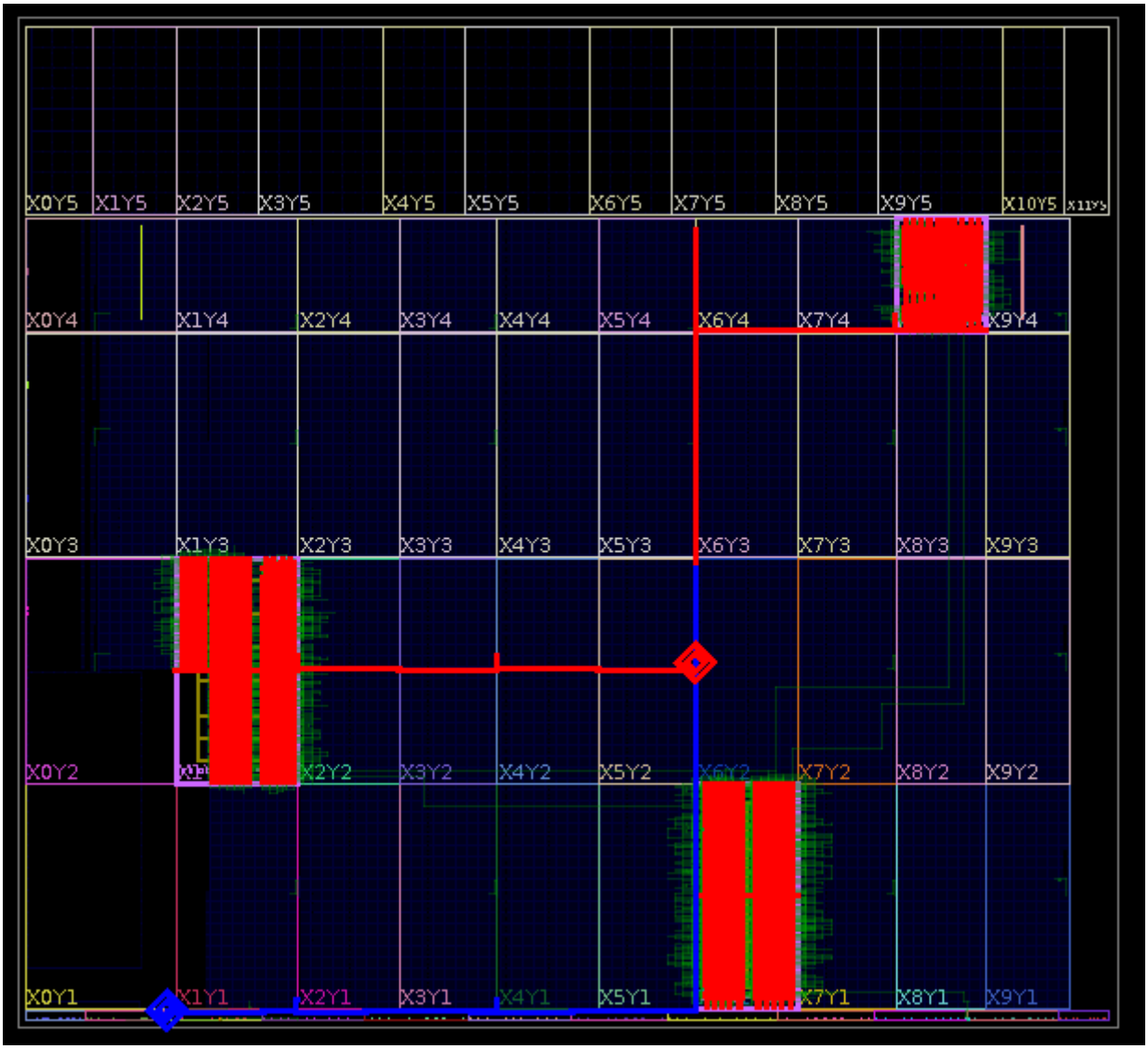
下图显示了位于时钟区域 X3Y0 内的时钟缓冲器访问其布局在时钟扩展窗口内的负载的方式，该时钟扩展窗口是由从 X5Y2 到 X6Y3 的矩形时钟区域构成的。

图 51：**Versal 自适应 SoC 时钟布线（从驱动到负载）**



在下图中，已布线的器件视图显示了遍布器件上大部分区域的全局时钟示例。驱动网络的时钟缓冲器在时钟区域 X2Y0 中以蓝色高亮显示，并通过 XPIO bank 驱动到该时钟区域中的水平布线，直至 X7Y0 为止。然后，信号线从水平布线转换到时钟区域 X5Y1 中的垂直布线，并到达时钟区域 X5Y2 内的时钟根。所有时钟布线均标记为蓝色。时钟根在时钟区域 X5Y2 中标记为红色。信号线从 X5Y2 中的时钟根转换到垂直分布，然后通过水平分布到达时钟叶管脚。CLB 列中的分布层和叶时钟布线资源标记为红色。

图 52：**Versal 自适应 SoC 已布线的时钟网络的布线器件视图**



时钟树布局布线

在以下阶段中，Vivado 布局器会在遵循物理 XDC 约束的同时，判定 MMCM/XPLL/DPLL、全局时钟缓冲器和时钟根的布局：

1. I/O 和时钟布局

   布局器根据连接规则和用户约束对 I/O 缓冲器和 MMCM/XPLL/DPLL 进行布局。布局器将时钟缓冲器分配给时钟区域，但不分配给单个站点 (site)，除非使用 LOC 属性进行约束。仅限驱动非时钟负载的时钟缓冲器才能在流程后期根据其驱动和负载的布局移至其他时钟区域。

   在此阶段的任何布局器错误都是由于连接规则和/或用户约束存在冲突所导致的。log 日志文件会显示有关错误的可能根本原因的详细信息，您必须仔细查看以便执行适当的设计或约束更改。

2. 时钟树预布线

布局器会指导后续实现步骤，并为布局后时序分析提供准确的延迟估算。

布局后，Vivado 工具可按如下所示修改时钟树实现：

- Vivado 物理最优化器可以将单元复制并移动到时钟区域中，此类时钟区域位于时钟信号线的时钟扩展窗口内且没有关联时钟。
- Vivado 布线器可通过调整来改进时序 QoR 并使时钟布线合规。

含布局规则和不含布局规则的拓扑结构

约束源	未约束的目标	行为
GCIO	BUFGCE/MBUFGCE、BUFGCTRL/MBUFGCTRL、BUFGCE_DIV/MBUFGCE_DIV 和 MMCM/XPLL/DPLL	自动布局在相同时钟区域内
MMCM/XPLL/DPLL	BUFGCE/MBUFGCE、BUFGCTRL/MBUFGCTRL 和 BUFGCE_DIV/MBUFGCE_DIV	自动布局在相同时钟区域内
GT*_QUAD	BUFG_GT/MBUFG_GT	自动布局在相同时钟区域内
BUFGCTRL	BUFGCTRL	自动布局在相同时钟区域内 **注释：**您可使用 CLOCK_REGION 约束覆盖相同时钟区域内的布局。
BUFG*/MBUFG*	BUFG*/MBUFG*	未约束的目标 BUFG*/MBUFG* 的布局不可预测 建议使用 CLOCK_REGION 约束来约束目标 BUFG*/MBUFG* **注释：**这不包括 BUFGCTRL -> BUFGCTRL。
BUFG*/MBUFG*	MMCM/XPLL/DPLL	未约束的目标 MMCM/XPLL/DPLL 的布局不可预测 建议使用 LOC 约束来约束 MMCM/XPLL/DPLL 当布线跨相邻时钟区域或多个时钟区域时，建议采用 CLOCK_DEDICATED_ROUTE 约束

相关信息

使用 CLOCK_DEDICATED_ROUTE 约束

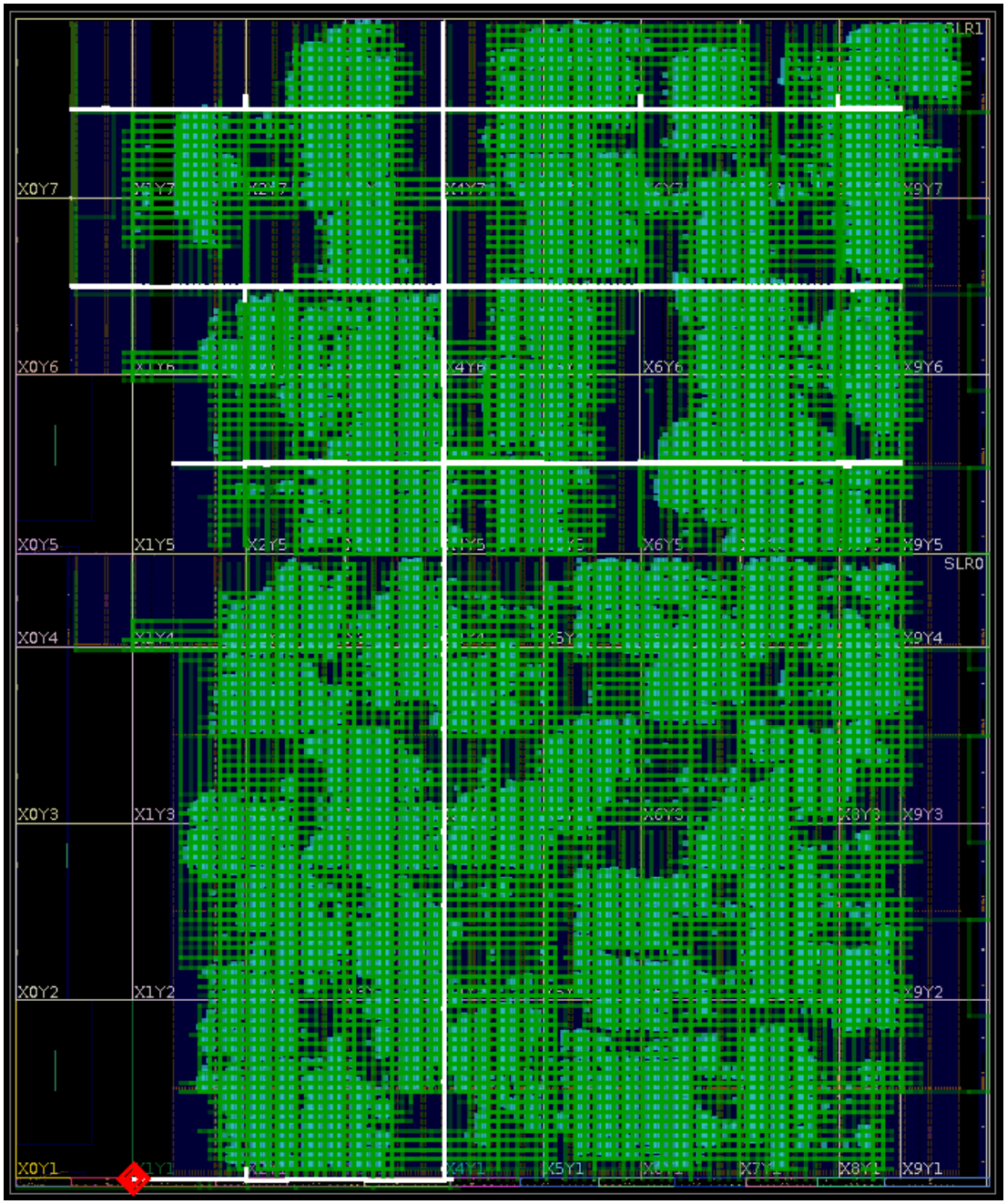
时钟功能

时钟规划必须根据目标器件中的高扇出时钟的总数来执行。

高扇出时钟

高扇出时钟几乎遍布 Versal 器件的所有时钟区域，或者几乎遍布 Versal 自适应 SoC 堆叠硅片互联 (SSI) 技术器件的整个 SLR。下图显示的高扇出时钟几乎遍布 Versal 自适应 SoC SSI 技术器件的整个 SLR，其中 XPIO bank BUFGCE 驱动程序以红色显示。时钟信号线占用 SLR0 和 SLR1 中的垂直布线资源，但仅扇出到 SLR1 中的分散资源。在任一设计中使用超过 24 个高扇出时钟，可能导致诸多需提前规划的问题，例如，使用 Pblock 作为 SLR 内的逻辑层级或者将 LOC 约束分配到 XPIO bank 中的时钟源。

图 53：**遍布 SLR1（源于 SLR0 内的 XPIO bank）的高扇出时钟**



低扇出时钟

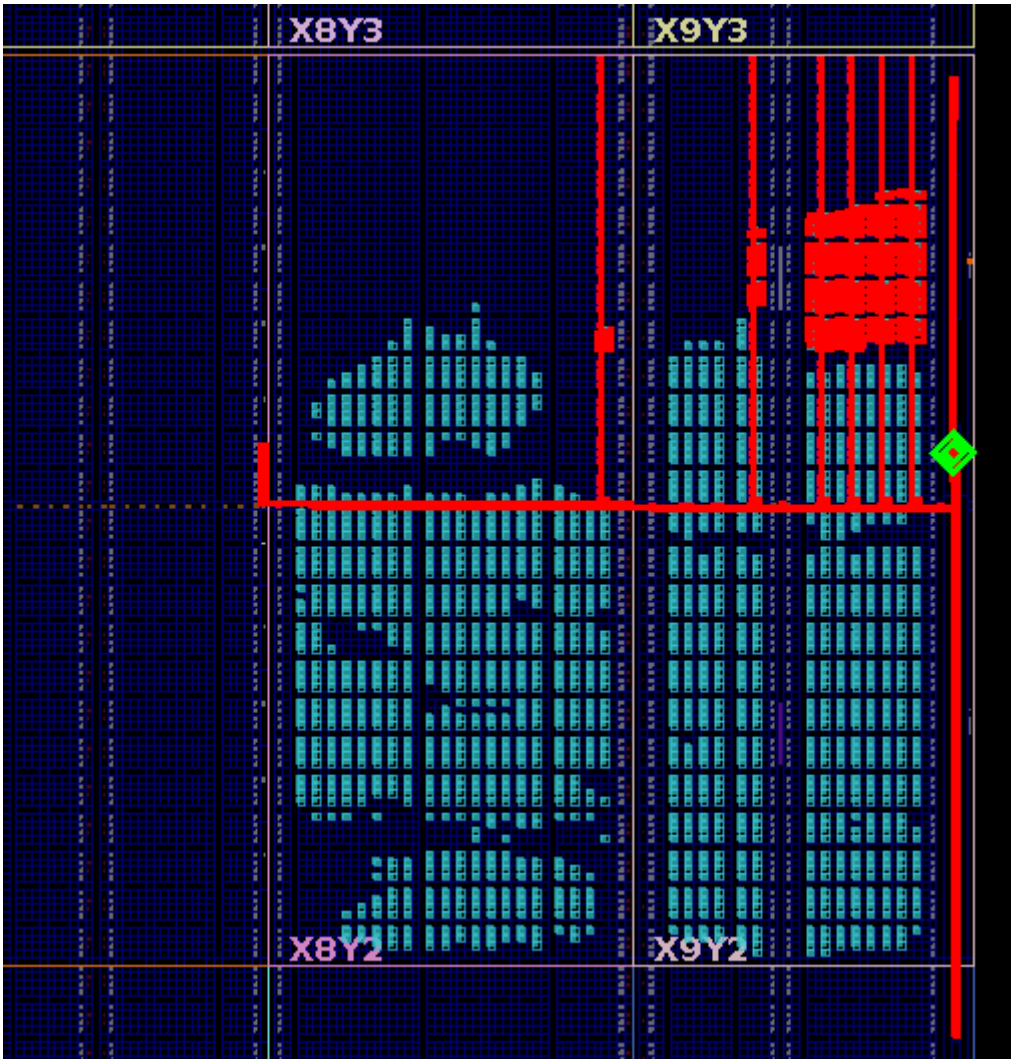
大多数情况下，低扇出时钟是连接到少于 1000 个时钟管脚的时钟信号线，这些时钟管脚布局在不超过 3 个水平相邻的时钟区域内。时钟布线、时钟根和时钟分布全部包含在局部区域内。

布局器应可识别低扇出时钟，但在某些情况下可能无法识别。这可能是由于设计大小、器件大小或物理 XDC 约束（如，LOC 约束或 Pblock）所导致的，这些因素可能导致布局器无法将负载布局到局部区域内。要解决此问题，您可能需要手动创建 Pblock 或修改现有物理约束以指导该工具完成操作。

由 BUFG_GT 驱动的时钟即为低扇出时钟的示例。下图显示了含 BUFG_GT 驱动（以绿色显示）的 2 个时钟区域内包含的低扇出时钟。

提示：您可使用 CLOCK_LOW_FANOUT XDC 约束将低扇出时钟包含在单个时钟区域内。

图 54：**2 个时钟区域包含的低扇出时钟**



时钟约束

物理 XDC 约束驱动时钟树的实现并控制使用高扇出时钟资源。由于 Versal 器件时钟比具有先前架构的时钟更灵活，并且包含其他架构约束，因此，了解如何正确约束时钟以完成实现至关重要。

针对 IO/MMCM/XPLL/DPLL/GT 使用 LOC 约束

要约束时钟，可按如下所示分配布局约束：

- 在 I/O 端口的时钟输入处

为 GCIO 上的时钟分配 PACKAGE_PIN 约束或者将 LOC 分配给 IOB 会影响时钟网络。直接连接到输入端口的 MMCM/XPLL/DPLL 和时钟缓冲器必须布局在相同时钟区域内。

- 对 MMCM/XPLL/DPLL 分配约束

  直接连接到 MMCM/XPLL/DPLL 输出的时钟缓冲器和连接到 MMCM/XPLL/DPLL 输入的输入时钟端口自动布局在相同时钟区域内。如果输入时钟端口和 MMCM/XPLL/DPLL 直接连接并约束到不同时钟区域，那么必须手动插入时钟缓冲器并对连接到 MMCM/XPLL/DPLL 的信号线设置 CLOCK_DEDICATED_ROUTE 约束。

- 在 GT*_QUAD 或 IBUFDS_GT* 单元上

  由单元驱动的 BUFG_GT 布局在相同时钟区域内。

注意！AMD 不推荐在时钟缓冲器单元上使用 LOC 约束。此方法将为时钟强制分配特定轨道 ID，这可能导致布局无法进行规范布线。仅当您已明确了解设计的整个时钟树并且设计中的布局保持一致时，才能使用 LOC 约束在 Versal 器件中布局高扇出时钟缓冲器。即使在采取这些预防措施后，由于设计或约束变化，在实现期间仍可能发生冲突。

在全局时钟缓冲器上使用 CLOCK_REGION 属性

您可以使用 CLOCK_REGION 约束为时钟区域分配时钟缓冲器，而不指定站点 (site)。这样即可提升布局器的灵活性，从而对所有时钟树进行最优化，并判定相应的缓冲器站点以便成功完成所有时钟的布线。

您还可使用 CLOCK_REGION 约束来提供有关级联时钟缓冲器或由非时钟原语（例如，互连结构逻辑）所驱动的时钟缓冲器的布局指南。

在以下示例中，XDC 约束将 clkgen/clkout2_buf 时钟缓冲器分配至 XPIO bank CLOCK_REGION X3Y0。

```
set_property CLOCK_REGION X3Y0 [get_cells clkgen/clkout2_buf]
```

注释：大多数情况下，时钟缓冲器由已约束到时钟区域的输入时钟端口、MMCM、XPLL、DPLL 或 GT*_QUAD 直接驱动。在此情况下，时钟缓冲器将自动布局在相同时钟区域内，您无需使用 CLOCK_REGION 约束。

在时钟信号线上使用 GCLK_DESKEW 属性

您可以使用 GCLK_DESKEW 属性来为时钟信号线禁用校准去歪斜。对于支持校准去歪斜的器件，时钟网络的初始延迟抽头是在器件启动时进行校准的，这样在为时钟信号线启用校准去歪斜时即可进一步减小时钟偏差。如需了解有关不同 Versal 器件的默认校准去歪斜设置的更多信息，请参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003)。对于某些时钟拓扑结构，尽可能减小插入延迟更重要，对于时序收敛而言，时钟网络去歪斜的重要性略低。在这些情况下，您可通过将时钟信号线上的 GCLK_DESKEW 属性设置为 OFF 来实现无附加延迟的时钟网络以平衡时钟偏差。这样同样会为该时钟信号线禁用校准去歪斜。您必须在由时钟缓冲器直接驱动的信号线段上设置 GCLK_DESKEW 属性。下面给出 1 个示例：

```
set_property GCLK_DESKEW OFF [get_nets -of [get_pins clkgen/
BUFG_clkout2_inst/O]]
```

注释：GCLK_DESKEW 属性搭配 USER_CLOCK_ROOT 约束使用时最有效，这样可以强制时钟根更接近所需插入延迟最少的负载。

使用 CLOCK_LOW_FANOUT 约束

您可以使用 CLOCK_LOW_FANOUT 约束在单个时钟区域中包含时钟缓冲器负载。您可在全局时钟缓冲器直接驱动的时钟信号线段上或者在一系列触发器上设置 CLOCK_LOW_FANOUT 约束。

注释：配合其他时钟使用时，CLOCK_LOW_FANOUT 约束优先级较低。如果 CLOCK_LOW_FANOUT 与其他时钟约束（如 USER_CLOCK_ROOT、CLOCK_DELAY_GROUP 或 CLOCK_DEDICATED_ROUTE）存在冲突，那么将不会遵循 CLOCK_LOW_FANOUT。

触发器的约束示例

在全局时钟缓冲器驱动的一系列触发器上设置 CLOCK_LOW_FANOUT 约束将导致 `opt_design` 创建新的并行全局时钟缓冲器，以隔离这些触发器。在 `place_design` 期间，由新创建的并行全局时钟缓冲器驱动并已隔离的触发器将包含到单个时钟区域中。

以下示例显示了适用于一系列触发器的 CLOCK_LOW_FANOUT 约束，这些触发器在时钟门控同步电路中用于控制全局时钟缓冲器的时钟使能。

```
set_property CLOCK_LOW_FANOUT TRUE [get_cells safeClockStartup_reg[*]]
```

在设计中，不间断的时钟网络最初驱动 2000 余个负载，包括在时钟门控同步电路中用于对其他逻辑进行时钟门控的触发器。以下板级原理图显示了在 `opt_design` 创建新的并行全局时钟缓冲器用于隔离时钟门控同步电路前后，时钟门控同步电路的不同状态以及连接到不间断时钟网络的其他逻辑。

图 55：**执行 opt_design 变换并对触发器应用 CLOCK_LOW_FANOUT 之前的板级原理图**

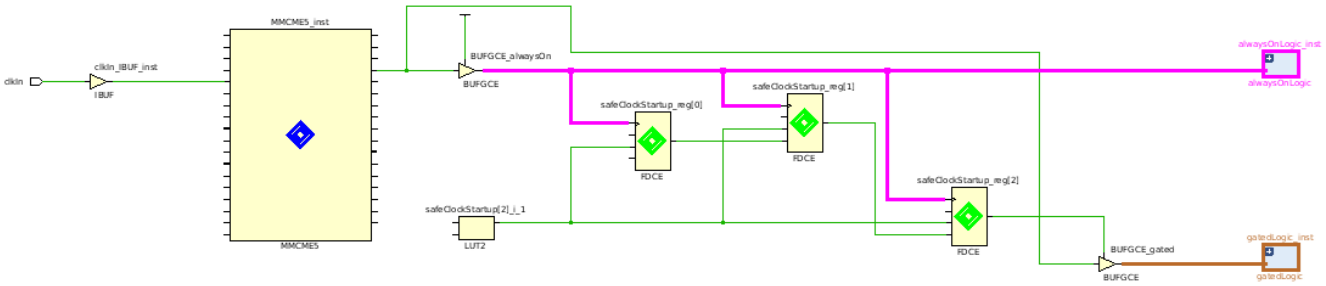
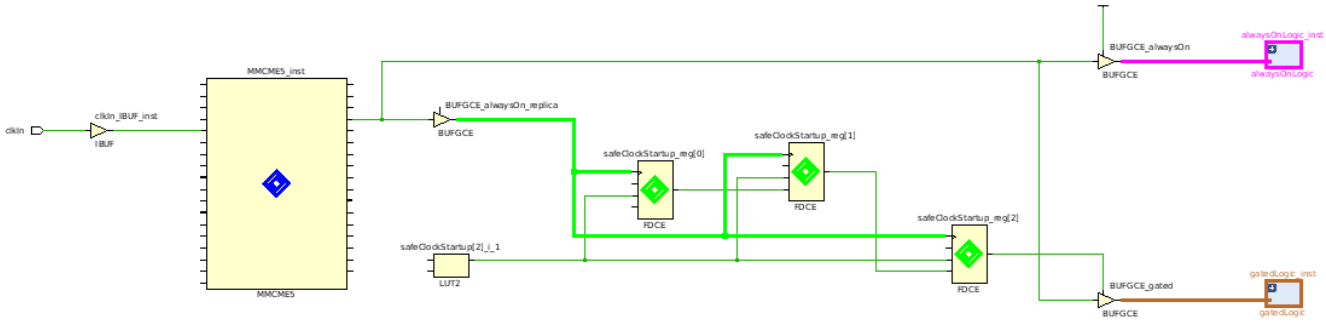
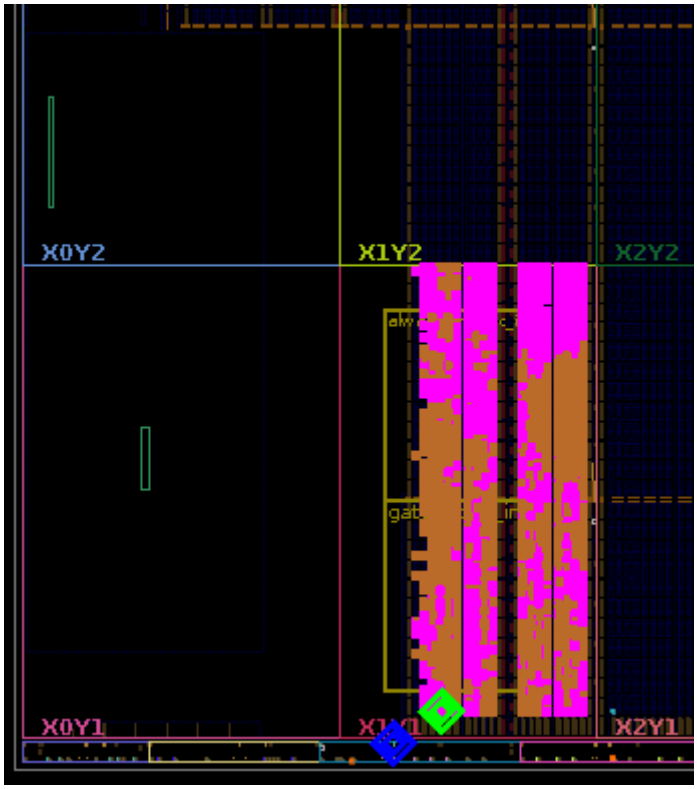


图 56：**执行 opt_design 变换并对触发器应用 CLOCK_LOW_FANOUT 之后的板级原理图**



完全实现的设计的“Device”（器件）窗口会显示时钟门控同步电路，其中不间断逻辑和时钟门控逻辑带有绿色标记。时钟门控同步电路布局在 CLOCK_REGION 中，导致 XPIO 中的 BUFG 存在最低插入延迟。

图 57：**完全实现的设计（含时钟门控同步电路布局）**



由 XPIO 全局时钟缓冲器驱动的时钟信号线的约束示例

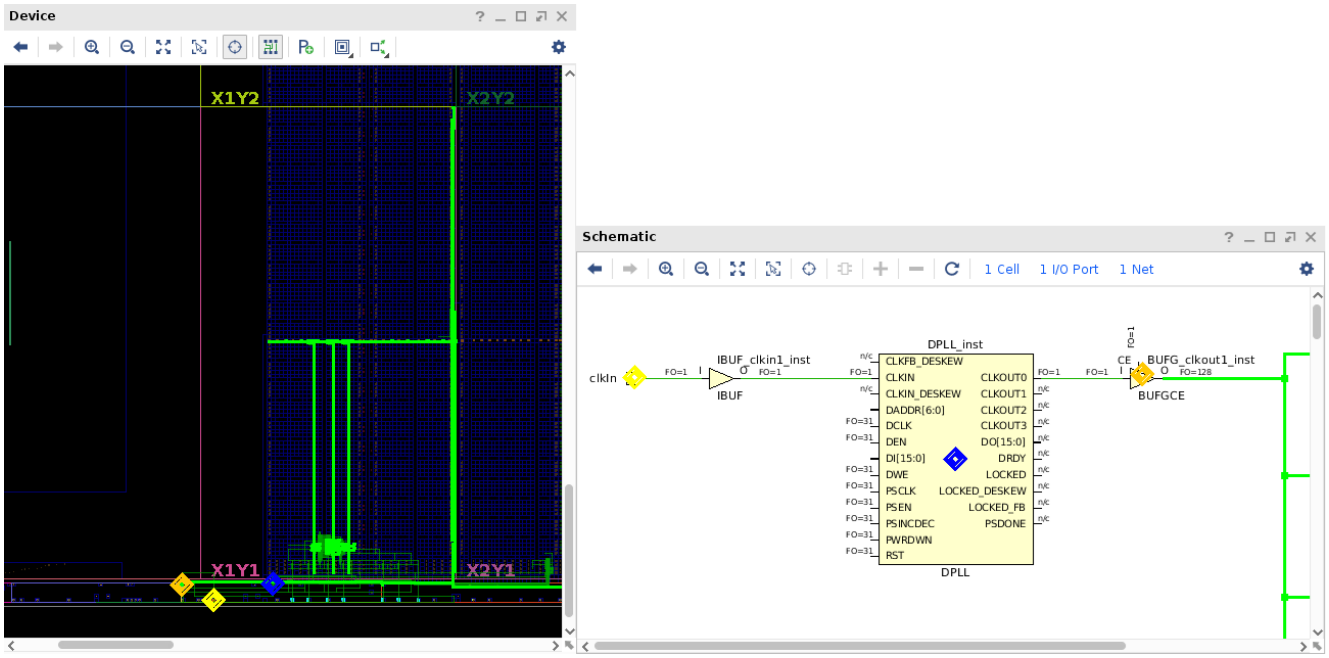
如果在由 XPIO CLOCK_REGION 中的全局时钟缓冲器直接驱动的时钟信号线段上设置 CLOCK_LOW_FANOUT 属性，并且全局时钟缓冲器的扇出的负载数不足 2000，那么负载布局将包含在单个时钟区域内。

注释：由于 XPIO CLOCK_REGION 与互连结构 CLOCK_REGION 对齐错误，因此使用 CLOCK_LOW_FANOUT 约束将导致这些负载被布局到单一时钟区域内，且源于 XPIO 全局时钟缓冲器的插入延迟最小。

以下示例所示的 CLOCK_LOW_FANOUT 应用于时钟信号线段，该时钟信号线段由布局在 XPIO 中的全局时钟缓冲器直接进行驱动。输入时钟端口 `clkIn` 包含分配到 GCIO（位于 XPIO CLOCK_REGION X2Y0 中）中的 PACKAGE_PIN，并驱动 DPLL。此 DPLL 可驱动全局时钟缓冲器，后者继而驱动含 128 个负载的时钟网络。全局时钟缓冲器的负载均布局在 CLOCK_REGION X1Y1 内。

```
# PACKAGE_PIN BB44 - High Performance XPIO IOBank 702 - CLOCK_REGION X2Y0
set_property PACKAGE_PIN BB44 [get_ports clkIn]
set_property IOSTANDARD SSTL12 [get_ports clkIn]
set_property CLOCK_LOW_FANOUT TRUE [get_nets -of [get_pins
BUFG_clkout1_inst/O]]
```

图 58： **“Device”窗口与“Schematic”窗口中的 XPIO CLOCK_LOW_FANOUT 示例**



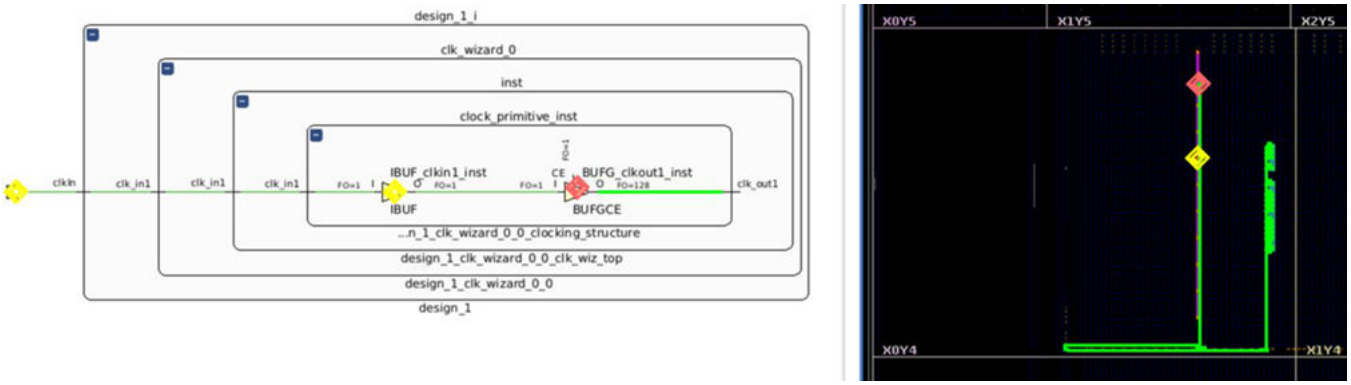
由 HDIO 全局时钟缓冲器驱动的时钟信号线的约束示例

如果在由 HDIO CLOCK_REGION 中的全局时钟缓冲器直接驱动的时钟信号线段上设置 CLOCK_LOW_FANOUT 属性，并且全局时钟缓冲器的扇出的负载数不足 2000，那么负载布局将包含在与全局时钟缓冲器相同的 CLOCK_REGION 内。

以下示例所示的 CLOCK_LOW_FANOUT 应用于时钟信号线段，该时钟信号线段由布局在 HDIO 中的全局时钟缓冲器直接进行驱动。输入时钟端口 `clkIn` 包含分配到 GCIO（位于 HDIO CLOCK_REGION X0Y4 中）中的 PACKAGE_PIN，并驱动 DPLL。此 DPLL 可驱动全局时钟缓冲器，后者继而驱动含 128 个负载的时钟网络。全局时钟缓冲器的负载均布局在 CLOCK_REGION X0Y4 内。

```
# PACKAGE_PIN L35 - High Density HDIO IOBank 306 - CLOCK_REGION X0Y4
set_property PACKAGE_PIN L35 [get_ports clkIn]
set_property IOSTANDARD LVCMOS33 [get_ports clkIn]
set_property CLOCK_LOW_FANOUT TRUE [get_nets -of [get_pins
BUFG_clkout1_inst/O]]
```

图 59：**“Device”窗口与“Schematic”窗口中的 HDIO CLOCK_LOW_FANOUT 示例**

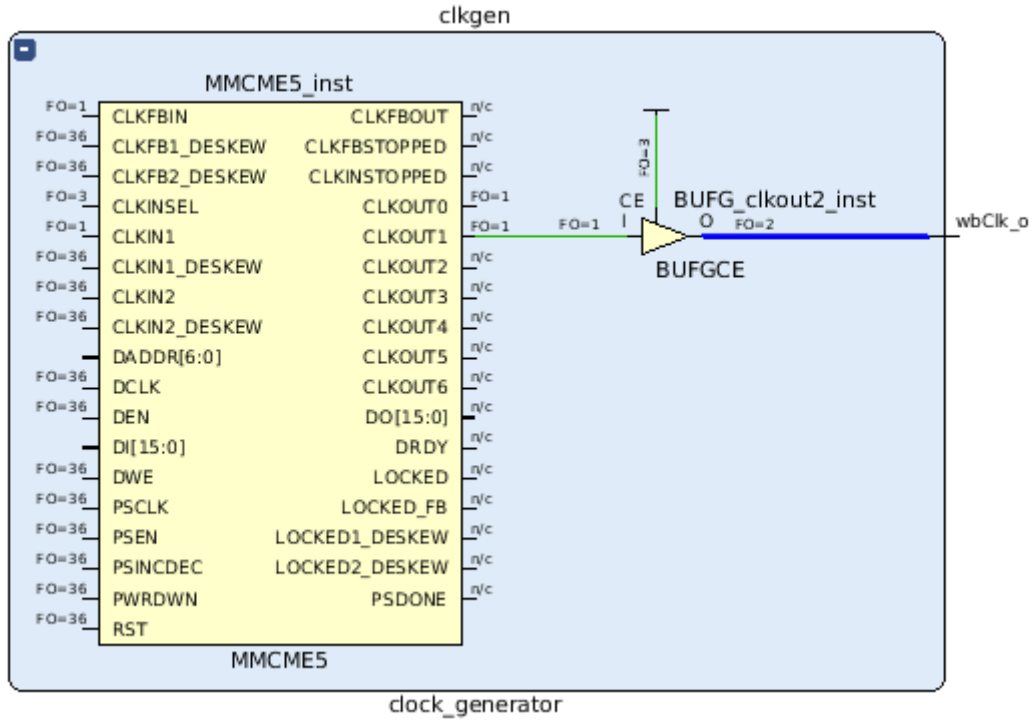


在时钟信号线上使用 USER_CLOCK_ROOT 属性

USER_CLOCK_ROOT 属性可用于强制限制时钟缓冲器所驱动的时钟的时钟根位置。指定 USER_CLOCK_ROOT 属性会影响设计布局，因为它会修改时钟布线，从而影响插入延迟和偏差。USER_CLOCK_ROOT 值对应时钟区域，并且必须在高扇出时钟缓冲器所驱动的信号线段上直接设置该属性。下面给出 1 个示例：

```
set_property USER_CLOCK_ROOT X3Y2 [get_nets -of [get_pins clkgen/
BUFG_clkout2_inst/O]]
```

图 60：**应用于由时钟缓冲器驱动的信号线段上的 USER_CLOCK_ROOT**



完成布局后即可使用 CLOCK_ROOT 属性来查询实际时钟根，如以下示例所示。CLOCK_ROOT 会报告由用户分配的根和由 Vivado 工具自动分配的根。

```
get_property CLOCK_ROOT [get_nets -of [get_pins clkgen/BUFG_clkout2_inst/O]]
=> X3Y2
```

复查已实现的设计的时钟根分配的另一种方法是使用 `report_clock_utilization` Tcl 命令。例如：

```
report_clock_utilization -clock_roots_only
```

下图显示了此报告。

图 61：**report_clock_utilization 时钟根分配**

```
| Index | Clock Net                                          | Root Clock Region
+-------+----------------------------------------------------+-------------------
|     1 | clkgen/clkfbout_buf                                | X4Y1
|     2 | clkgen/cpuClk_o                                    | X4Y1
|     3 | clkgen/fftClk_o                                    | X3Y2
|     4 | clkgen/phyClk0_o                                   | X3Y3
|     5 | clkgen/phyClk1_o                                   | X3Y2
|     6 | clkgen/usbClk_o                                    | X3Y3
```

在多个时钟信号线上使用 CLOCK_DELAY_GROUP 约束

您可使用 CLOCK_DELAY_GROUP 约束来匹配由不同时钟缓冲器驱动的多个相关时钟网络的插入延迟。此约束常用于最大限度减少源自相同 MMCM、XPLL、DPLL 或 GT 来源的时钟之间的同步 CDC 时序路径的偏差。以下示例显示了由时钟缓冲器直接驱动的 `clk1_net` 和 `clk2_net` 时钟信号线：

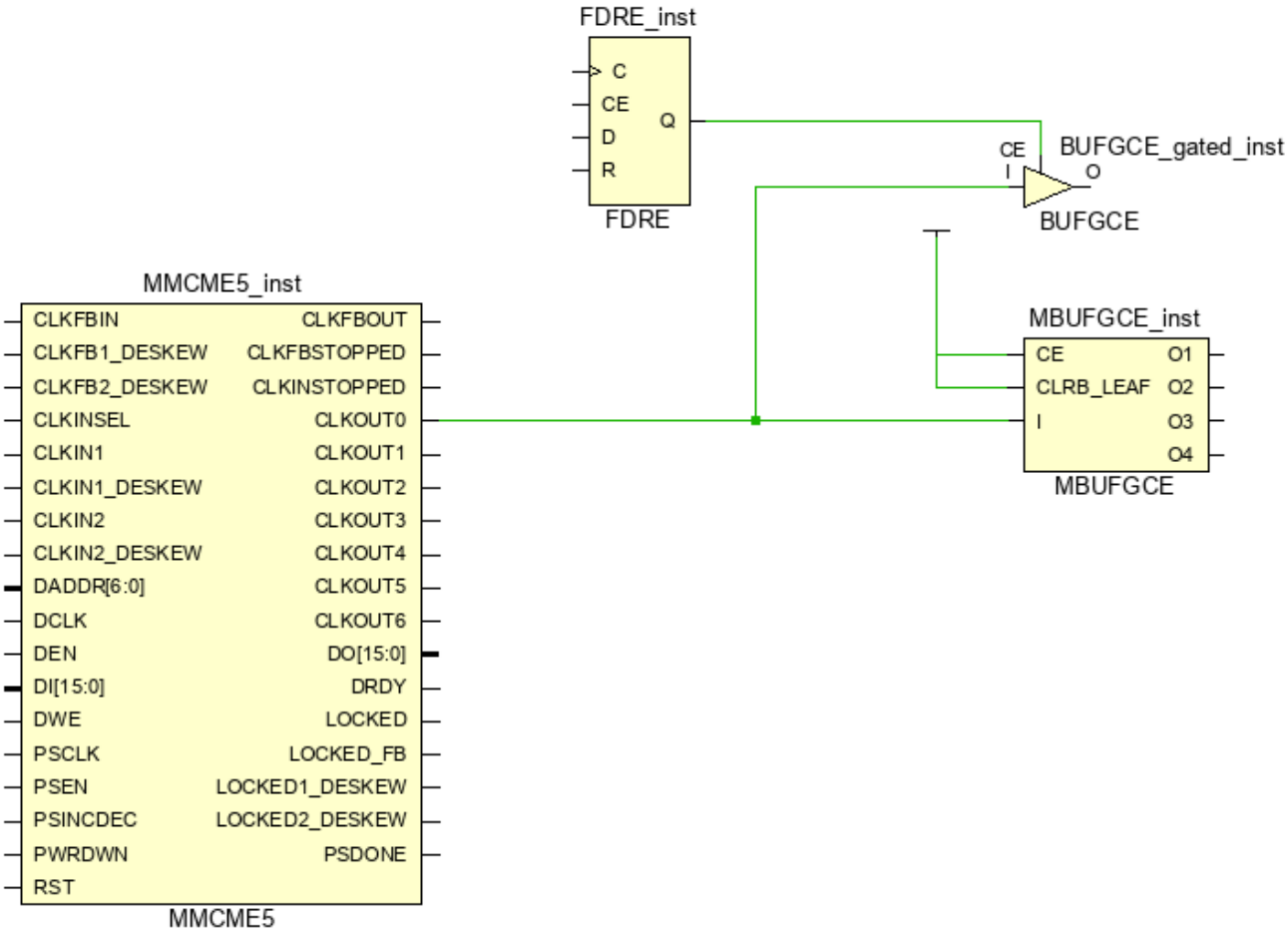
```
set_property CLOCK_DELAY_GROUP grp12 [get_nets {clk1_net clk2_net}]
```

重要提示！您必须在直接连接到时钟缓冲器的信号线段上设置 CLOCK_DELAY_GROUP 约束。

如果可能，请使用 MBUFG* 单元来最大限度减小同步 CDC 时序路径上的偏差，或者在匹配多个相关时钟网络的插入延迟时，请使用此单元。使用单个 MBUFG* 单元时，请勿使用 CLOCK_DELAY_GROUP 约束。将 MBUFG* 单元所驱动的时钟网络的插入延迟与另一个全局时钟缓冲器的时钟网络相匹配时，可使用 CLOCK_DELAY_GROUP 约束，如以下示例所示：

```
set_property CLOCK_DELAY_GROUP mbufGrp [get_nets -of [get_pins MBUFGCE_inst/
O*]]
set_property CLOCK_DELAY_GROUP mbufGrp [get_nets -of [get_pins
BUFGCE_gated_inst/O]]
```

图 62：**MBUFGCE 输出与门控 BUFGCE 的插入延迟匹配**



使用 CLOCK_DEDICATED_ROUTE 约束

从某 1 个时钟区域中的时钟缓冲器驱动到另一个时钟区域中的 MMCM/XPLL/DPLL 时，通常会使用 CLOCK_DEDICATED_ROUTE 约束。默认情况下，CLOCK_DEDICATED_ROUTE 约束设置为 TRUE，并且缓冲器与 MMCM/XPLL/DPLL 配对必须布局在相同时钟区域内。

重要提示！不同于其他架构，在 Versal 器件中不支持 CLOCK_DEDICATED_ROUTE 的值为 BACKBONE 和 ANY_CMT_COLUMN。在 Versal 器件中，您可改用以下 CLOCK_DEDICATED_ROUTE 值：SAME_CMT_ROW 和 ANY_CMT_REGION。更新约束前，AMD 建议查看时钟拓扑布局，因为先前架构中的时钟设置采用的是严格的列式时钟设置。

下表汇总了不同 CLOCK_DEDICATED_ROUTE 约束值、使用方式和行为。

注释：仅在全局时钟缓冲器的信号线上应用 ANY_CMT_COLUMN 约束和 SAME_CMT_COLUMN 约束。

表 6：**Versal 器件 CLOCK_DEDICATED_ROUTE 约束汇总**

值	用途	行为
TRUE	时钟信号线上的默认值	全局时钟缓冲器和 MMCM/XPLL/DPLL 必须布局在相同时钟区域内。 该值用于确保仅使用全局时钟资源对信号线进行布线。
SAME_CMT_COLUMN	由 BUFG_GT 全局时钟缓冲器或 HDIO bank 中的全局时钟缓冲器驱动的信号线 示例： `set_property CLOCK_DEDICATED_ROUTE SAME_CMT_COLUMN \ [get_nets -of [get_pins BUFG_GT_inst/O]]`	DPLL 必须布局在时钟区域内的同一个垂直列中。 该值用于确保仅使用全局时钟资源对信号线进行布线。 为实现最佳结果，AMD 建议在 DPLL 上使用 LOC 约束来将 DPLL 布局控制在同一垂直列中。
SAME_CMT_ROW	全局时钟缓冲器驱动的信号线 示例： `set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW \ [get_nets -of [get_pins BUFGCE_inst/O]] set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW \ [get_nets -of [get_pins BUFGCE_DIV_inst/O]] set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW \ [get_nets -of [get_pins BUFGCTRL_inst/O]]`	MMCM/XPLL/DPLL 可布局在含可用资源的任一水平时钟区域行内。 该值用于确保仅使用全局时钟资源对信号线进行布线。 为实现最佳结果，AMD 建议在 MMCM/XPLL/DPLL 上使用 LOC 约束来将 MMCM/XPLL/DPLL 布局控制在器件的水平时钟区域行内。
ANY_CMT_REGION	全局时钟缓冲器驱动的信号线 示例： `set_property CLOCK_DEDICATED_ROUTE ANY_CMT_REGION \ [get_nets -of [get_pins BUFGCE_inst/O]] set_property CLOCK_DEDICATED_ROUTE ANY_CMT_REGION \ [get_nets -of [get_pins BUFGCE_DIV_inst/O]] set_property CLOCK_DEDICATED_ROUTE ANY_CMT_REGION \ [get_nets -of [get_pins BUFGCTRL_inst/O]]`	MMCM/XPLL/DPLL 可布局在含可用资源的任一时钟区域内。 该值用于确保仅使用全局时钟资源对信号线进行布线。 为实现最佳结果，AMD 建议在 MMCM/XPLL/DPLL 上使用 LOC 约束来控制器件内的 MMCM/XPLL/DPLL 布局。

表 6：**Versal 器件 CLOCK_DEDICATED_ROUTE 约束汇总** (续)

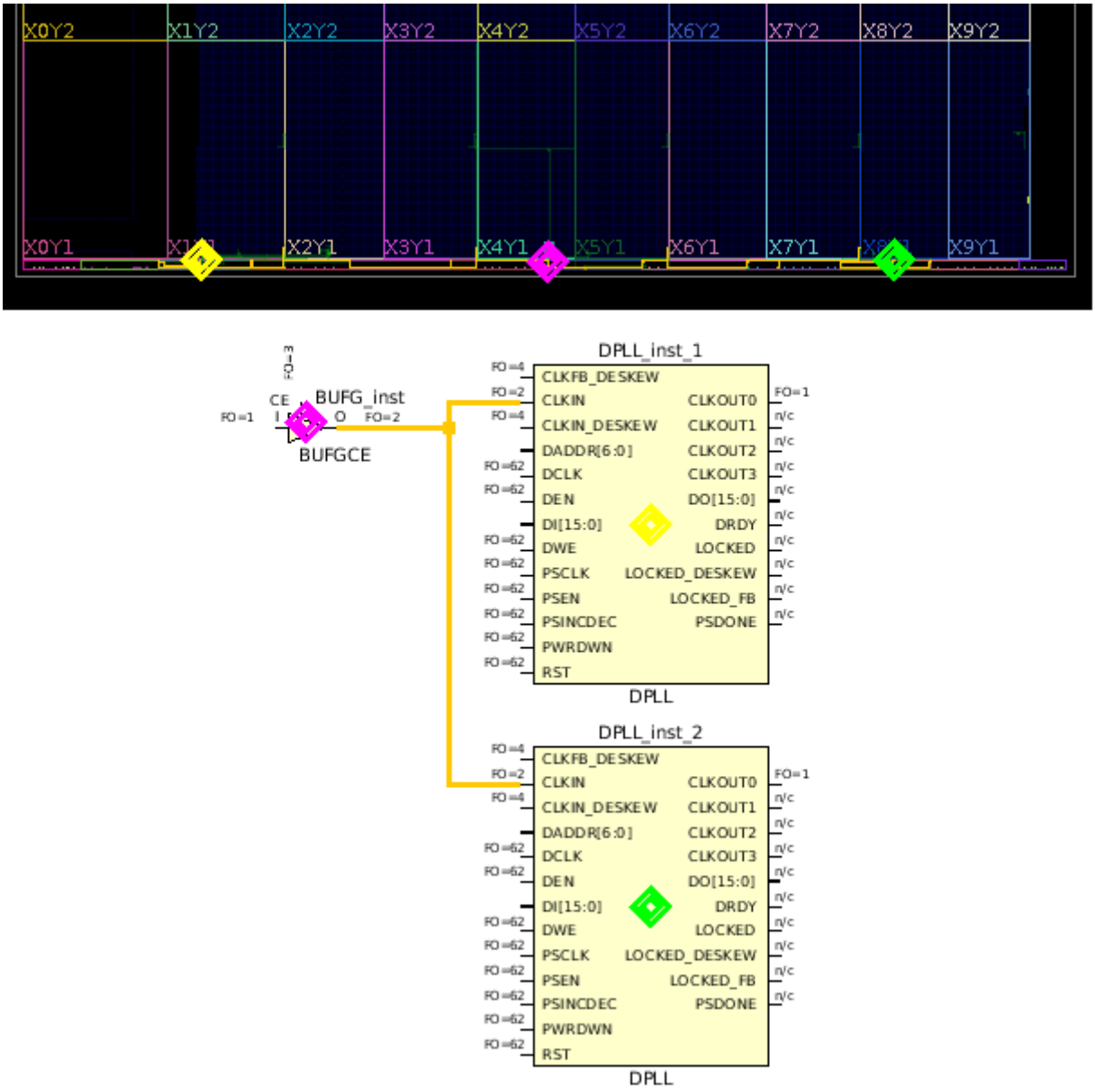
值	用途	行为
FALSE	时钟信号线不受全局时钟缓冲器驱动，但受时钟信号线的一部分驱动（例如，受 IBUF 输出驱动的信号线，或直接连接到 MMCM 的输出时钟管脚的信号线） 示例： `set_property CLOCK_DEDICATED_ROUTE FALSE \ [get_nets -of [get_pins MMCME5_inst/ CLKOUT0]] set_property CLOCK_DEDICATED_ROUTE FALSE \ [get_nets -of [get_pins IBUF_inst/O]]`	信号线是使用互连结构和全局时钟资源完成布线的。 这会对时钟网络的时序特性产生负面影响，例如抖动和偏差。 **重要提示！**对于 Versal 器件，仅当正常情况下使用全局时钟资源布线的时钟因特殊设计原因而需采用互连结构资源进行布线时，才能使用 FALSE。

驱动行对齐的 XPIO 时钟区域之间的 MMCM/XPLL/DPLL

从某个 XPIO 时钟区域内的时钟缓冲器驱动到另一个不相邻但行对齐的 XPIO 时钟区域内的 MMCM/XPLL/DPLL 时，必须将 CLOCK_DEDICATED_ROUTE 约束设置为 SAME_CMT_ROW。这样可防止出现实现错误，并确保在整个 XPIO 时钟区域内使用全局时钟资源对时钟进行布线。以下示例和图示显示了驱动非相邻的 XPIO 时钟区域内的 2 个 DPLL 的时钟缓冲器。

```
set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW [get_nets -of [get_pins
BUFG_inst/O]]
set_property LOC DPLL_X4Y0 [get_cells DPLL_inst_1]
set_property LOC DPLL_X11Y0 [get_cells DPLL_inst_2]
```

图 63：**CLOCK_DEDICATED_ROUTE 约束设置为 SAME_CMT_ROW**



在未实现行对齐的时钟区域之间驱动 DPLL

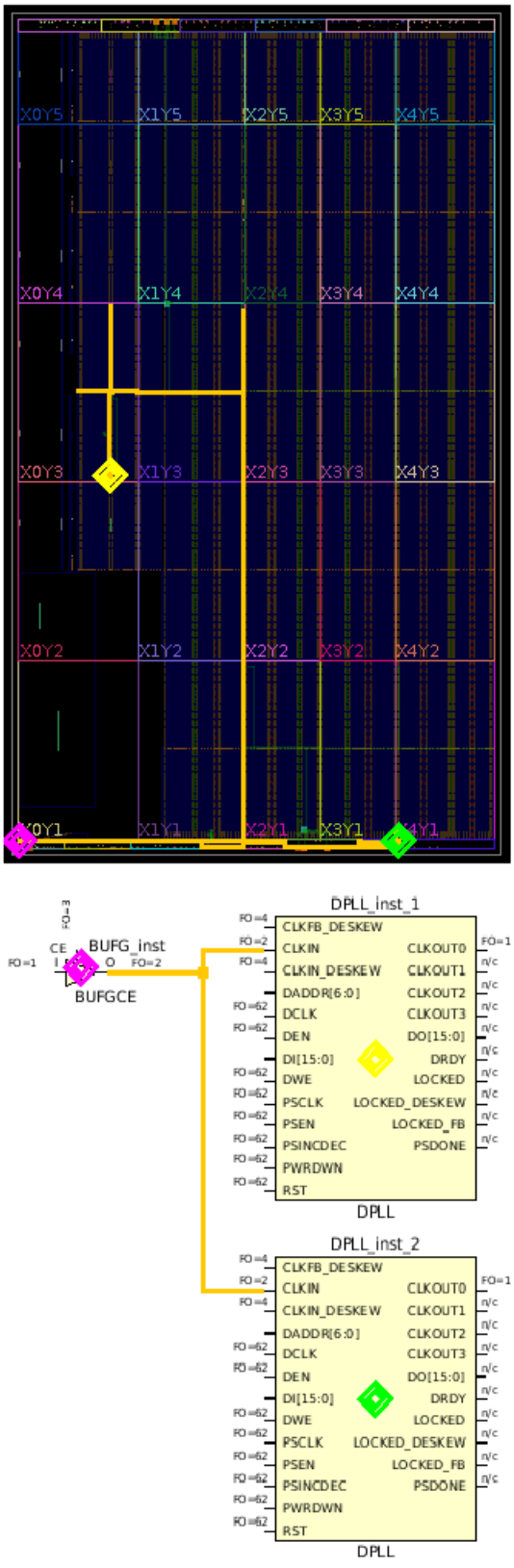
如果器件的 HDIO 内有 DPLL 可用，那么在下列情况下您必须将 CLOCK_DEDICATED_ROUTE 约束设置为 ANY_CMT_REGION：

- 从 XPIO 时钟区域内的时钟缓冲器驱动到 HDIO 时钟区域内的 DPLL 时
- 从 HDIO 时钟区域内的时钟缓冲器驱动到 XPIO 时钟区域内的 MMCM/XPLL/DPLL 时

这样可防止出现实现错误，并确保在整个器件中使用全局时钟资源对时钟进行布线。以下示例显示了来自 XPIO 时钟区域的时钟缓冲器，该时钟缓冲区驱动 2 个 DPLL，其中 1 个 DPLL 位于 XPIO 时钟区域内，另 1 个 DPLL 位于 HDIO 时钟区域内。

```
set_property CLOCK_DEDICATED_ROUTE ANY_CMT_REGION [get_nets -of [get_pins
BUFG_inst/O]]
set_property LOC DPLL_X4Y2 [get_cells DPLL_inst_1]
set_property LOC DPLL_X12Y0 [get_cells DPLL_inst_2]
```

图 64：**CLOCK_DEDICATED_ROUTE 约束设置为 ANY_CMT_REGION**



使用 CLOCK_ROUTE_GUIDE 约束

CLOCK_ROUTE_GUIDE 是 STRING 属性，可应用于时钟管脚对象。该属性允许您定义指定时钟信号线在 XPIO 中加载管脚时采用的布线。CLOCK_ROUTE_GUIDE 属性可用于与驱动 XPIO 负载管脚（例如，MMCM/CLKFB1_DESKEW 管脚）的时钟信号线的时钟布线和延迟紧密匹配。有效值包括：CMT_ROW、BUFDIV_LEAF 和 ANY。

注释：该属性仅适用于 XPIO 时钟负载。

下表汇总了不同 CLOCK_ROUTE_GUIDE 约束值、使用方式和行为。

表 7：**Versal 器件 CLOCK_ROUTE_GUIDE 约束汇总**

值	用途	行为
CMT_ROW	取该值时，表示使用仅包含在 XPIO 时钟区域行中的水平布线 示例： `set_property CLOCK_ROUTE_GUIDE CMT_ROW [get_pins myHier/myBUFG/I]`	当 CLOCK_ROUTE_GUIDE == CMT_ROW 时，时钟布线应包含在 XPIO 时钟区域行中。
BUFDIV_LEAF	取该值时，表示布线使用 BUFDIV_LEAF 穿过 BLI 到达 XPIO。 示例： `set_property CLOCK_ROUTE_GUIDE BUFDIV_LEAF [get_pins myHier/myBUFG/I]`	当 CLOCK_ROUTE_GUIDE == BUFDIV_LEAF 时，时钟布线应经过 BUFDIV_LEAF 和 BLI。
ANY	取该值时，表示使用 ANY 任意可能的布线到达 XPIO。 示例： `set_property CLOCK_ROUTE_GUIDE ANY [get_pins myHier/myBUFG/I]`	当 CLOCK_ROUTE_GUIDE == ANY，时钟布线可穿过 ANY 任意允许的节点。

当 CLOCK_ROUTE_GUIDE == BUFDIV_LEAF 时，时钟布线应经过 BUFDIV_LEAF 和 BLI。

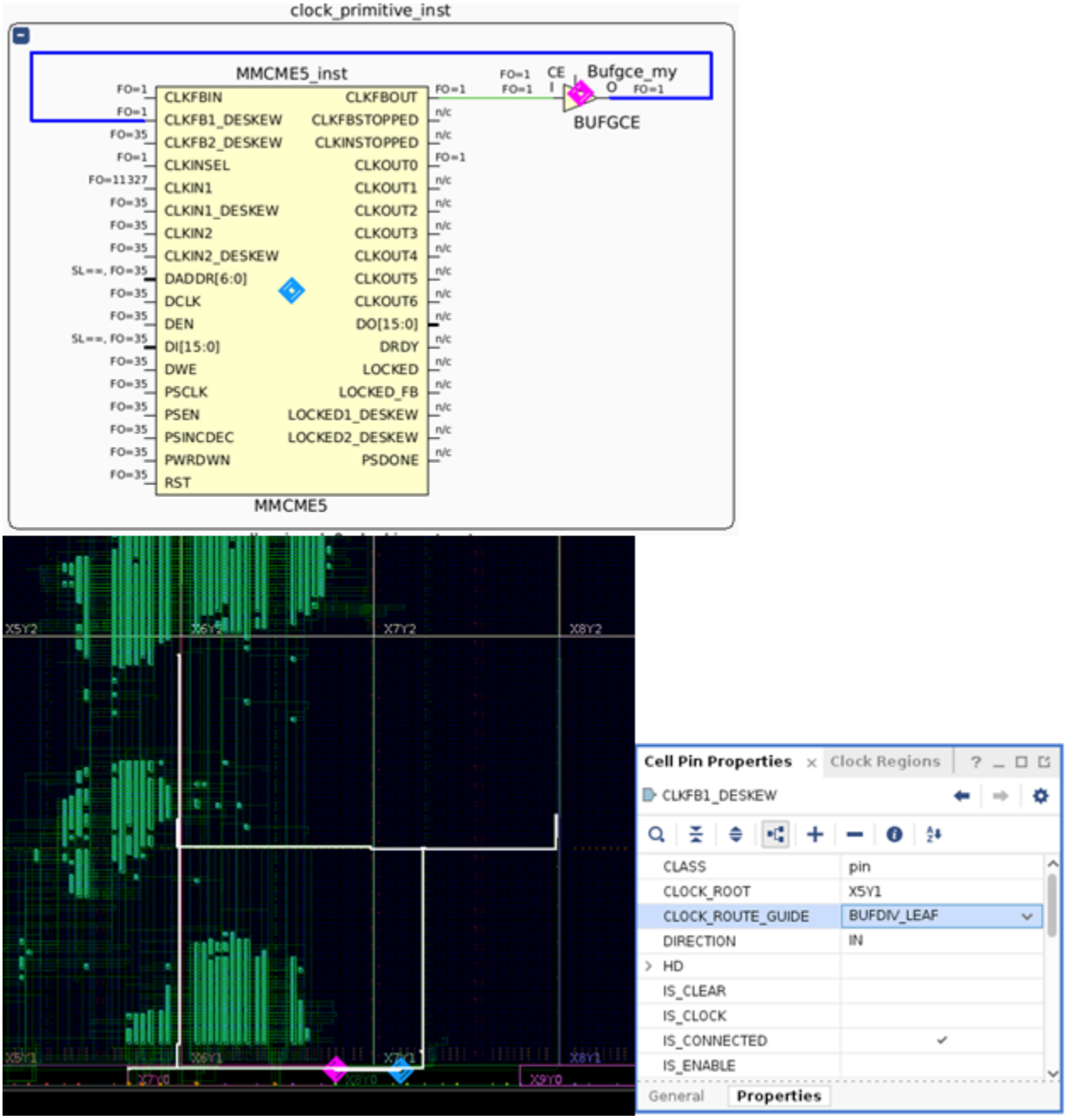
当 CLOCK_ROUTE_GUIDE == CMT_ROW 时，时钟布线应包含在 HSR 行中。

以下是 BUFDIV_LEAF 的代码示例：

```
set_property CLOCK_ROUTE_GUIDE BUFDIV_LEAF [get_pins sample_clk_mmcm/inst/clock_primitive_inst/MMCME5_inst/CLKFB1_DESKEW]
```

下图显示了穿越 BUFDIV_LEAF 和 BLI 的时钟布线，适用于 XPLL 反馈路径。

图 65：**CLOCK_ROUTE_GUIDE 约束设为 BUFDIV_LEAF**

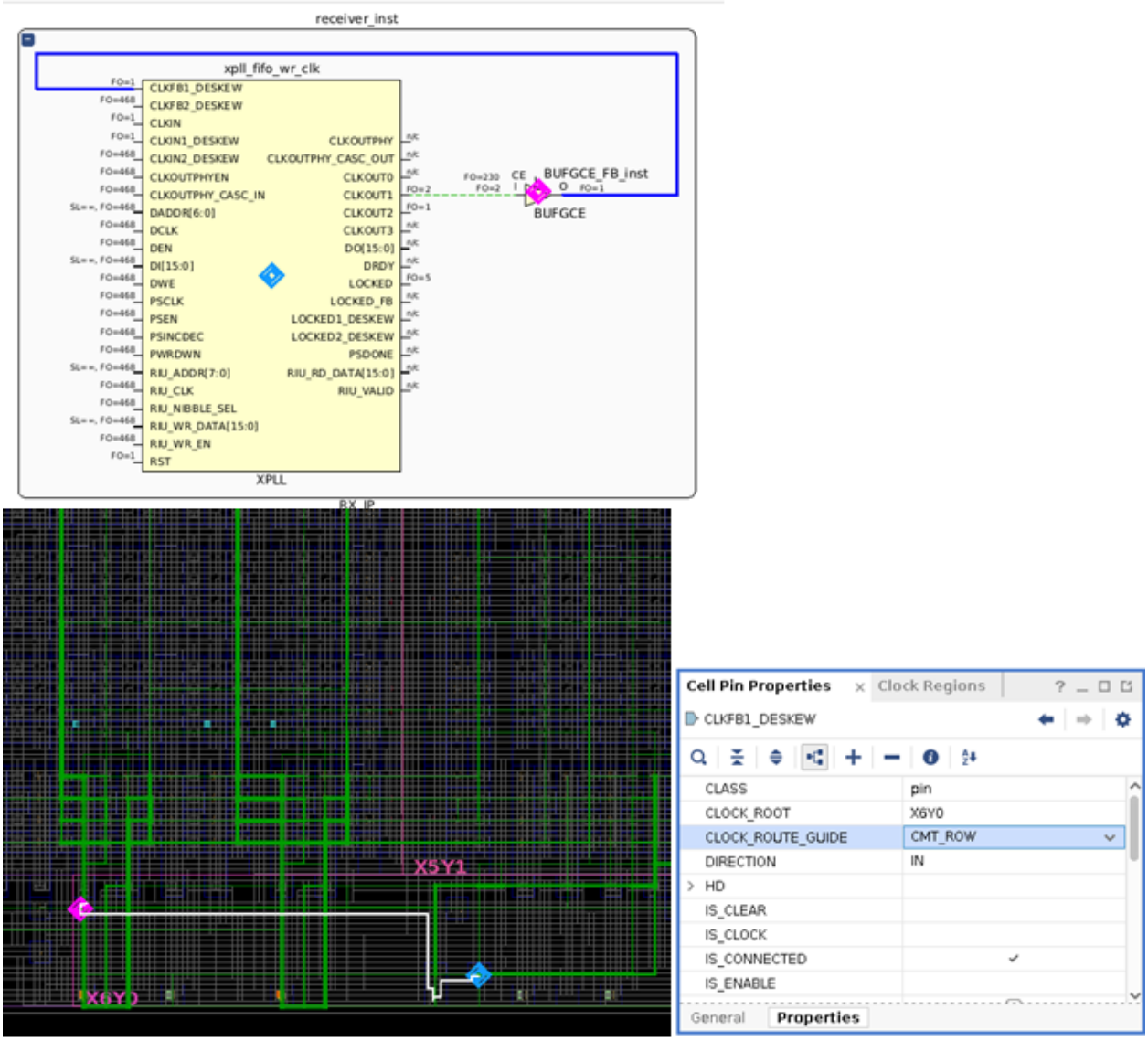


以下是 CMT_ROW 的代码示例：

```
set_property CLOCK_ROUTE_GUIDE CMT_ROW [get_pins receiver_inst/
xpll_fifo_wr_clk/CLKFB1_DESKEW]
```

下图显示了仅包含在 XPIO bank 中的时钟布线，适用于 MMCM 反馈路径。

图 66：**CLOCK_ROUTE_GUIDE 约束设为 CMT_ROW**



要使用 CLOCK_ROUTE_GUIDE，请根据所需行为输入约束并赋值。当前受支持的值包括：CMT_ROW、BUFDIV_LEAF 和 ANY。

例如，使用仅包含在 XPIO bank（其中时钟管理拼块按行对齐）中的水平布线时：

```
set_property CLOCK_ROUTE_GUIDE CMT_ROW [get_pins myHier/myBUFG/I]
```

例如，布线使用 BUFDIV_LEAF 穿过 BLI 到达 XPIO 时：

```
set_property CLOCK_ROUTE_GUIDE BUFDIV_LEAF [get_pins myHier/myBUFG/I]
```

时钟拓扑建议

AMD 建议使用简单的时钟树拓扑结构，因其设计所需的时钟缓冲器数量最少。使用额外的时钟缓冲器需要更多的布线轨道，这可能导致时钟区域内因时钟布线要求过高且接近最大容量而出现布局错误或者布线冲突。

以下是适用于 MBUFG 原语和 BUFGCE/BUFGCTRL/BUFGCE_DIV 连接的时钟拓扑建议。

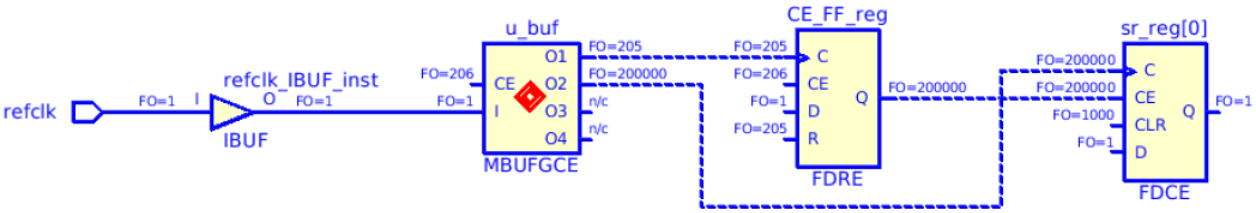
MBUFG 原语

Versal 架构引入了全新的时钟原语，称为 MBUFG。MBUFG 可接收输入时钟，并可提供输入时钟通过分频或倍频所生成的多个输出时钟。以下是 MBUFG 单元类型：

- MBUFGCE：用于通用全局时钟设置
- MBUFGCE_DIV：用于通用全局时钟设置，含与 BUFGCE_DIV 相似的分频功能
- MBUFGCTRL：用于通用全局时钟设置，含与 BUFGCTRL 相似的控制功能
- MBUFG_GT：用于全局时钟设置，含千兆位收发器 (GT)
- MBUFG_PS：用于由 Control, Interface, and Processing System (CIPS) IP 生成的全局时钟

MBUFG 单元指令实现工具使用单一全局时钟布线轨道以及叶时钟分频器和倍频器来重新创建输出时钟。这样即可减少偏差，对于同步 CDC 实现时序收敛大有益处。下图显示了 1 个简单的示例，其中有 1 个时钟输入连接到 MBUFGCE 输入以生成时钟和另一个 2 分频（除以 2）时钟。

图 67：**Versal 自适应 SoC 同步 CDC 的建议拓扑结构**



并行时钟缓冲器

使用并行时钟缓冲器来实现以下目的：

- 确保跨实现运行的布局可预测。

  当由相同输入时钟端口（MMCM、XPLL、DPLL 或 GT*_QUAD）直接驱动并行时钟缓冲器时，这些缓冲器始终与其驱动布局在相同时钟区域内，而与网表更改或逻辑布局变动无关。

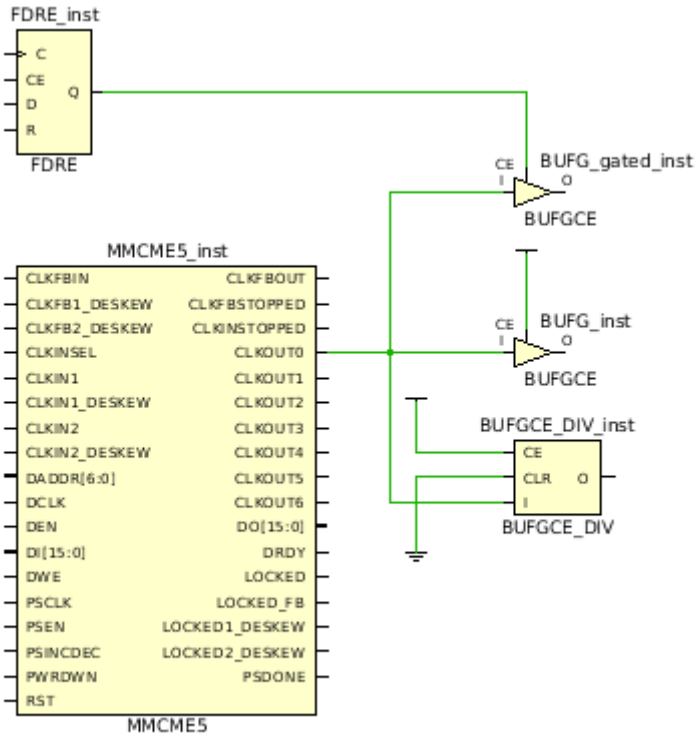
- 匹配时钟树的并行分支之间的插入延迟。

  AMD 推荐使用并行缓冲器替代级联时钟缓冲器，在分支之间存在同步路径时尤其如此。当使用级联缓冲器时，时钟树的分支之间的时钟插入延迟将不匹配，即使使用 CLOCK_DELAY_GROUP 或 USER_CLOCK_ROOT 约束也是如此。这可能导致高时钟偏差，从而导致时序收敛难度增加，甚至无法实现。

注释：如果可能，请使用 MBUFG* 单元来最大限度减小同步 CDC 时序路径上的偏差，或者在匹配多个相关时钟网络的插入延迟时，请使用此单元。使用 MBUFG* 单元时，不应使用 CLOCK_DELAY_GROUP 约束。

下图显示了由 MMCM CLKOUT0 端口驱动的 3 个并行全局时钟缓冲器。

图 68：**MMCM 输出上的并行 BUFGCE**



级联时钟缓冲器

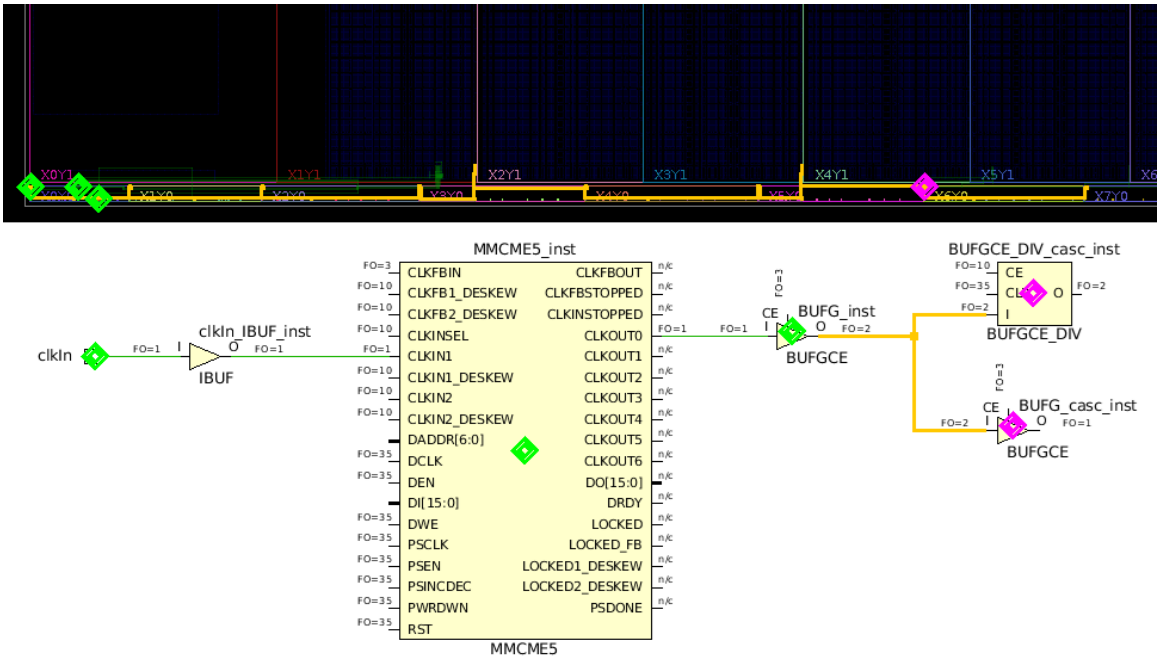
一般 AMD 不建议使用级联缓冲器在不相关的时钟树分支之间人为增加延迟并减少偏差。不同于 BUFGCTRL 之间的连接，其他时钟缓冲器连接在架构中不具有专用路径。因此，时钟缓冲器的相对布局不可预测，并且所有布局规则都优先于未约束的级联缓冲器的布局。

但是，使用级联时钟缓冲器可以实现以下功能：

- 从 XPIO 角点 (corner) bank 布线到位于不同时钟区域中的时钟资源。

  在 Versal 器件中，XPIO 角点 bank 无法连接到一般互连结构，并且全局时钟功能受限。例如，XPIO 角点 bank 能够访问通用 BUFGCE，但无法访问 BUFGCTRL 和 BUFGCE_DIV 资源。下图显示了 IOB-MMCM-BUFG 路径的输出，此路径连接到其他 XPIO 时钟区域内的并行 BUFGCE 和 BUFGCE_DIV 以插入延迟匹配。

图 69：**XPIO 角点 bank 级联时钟缓冲器**



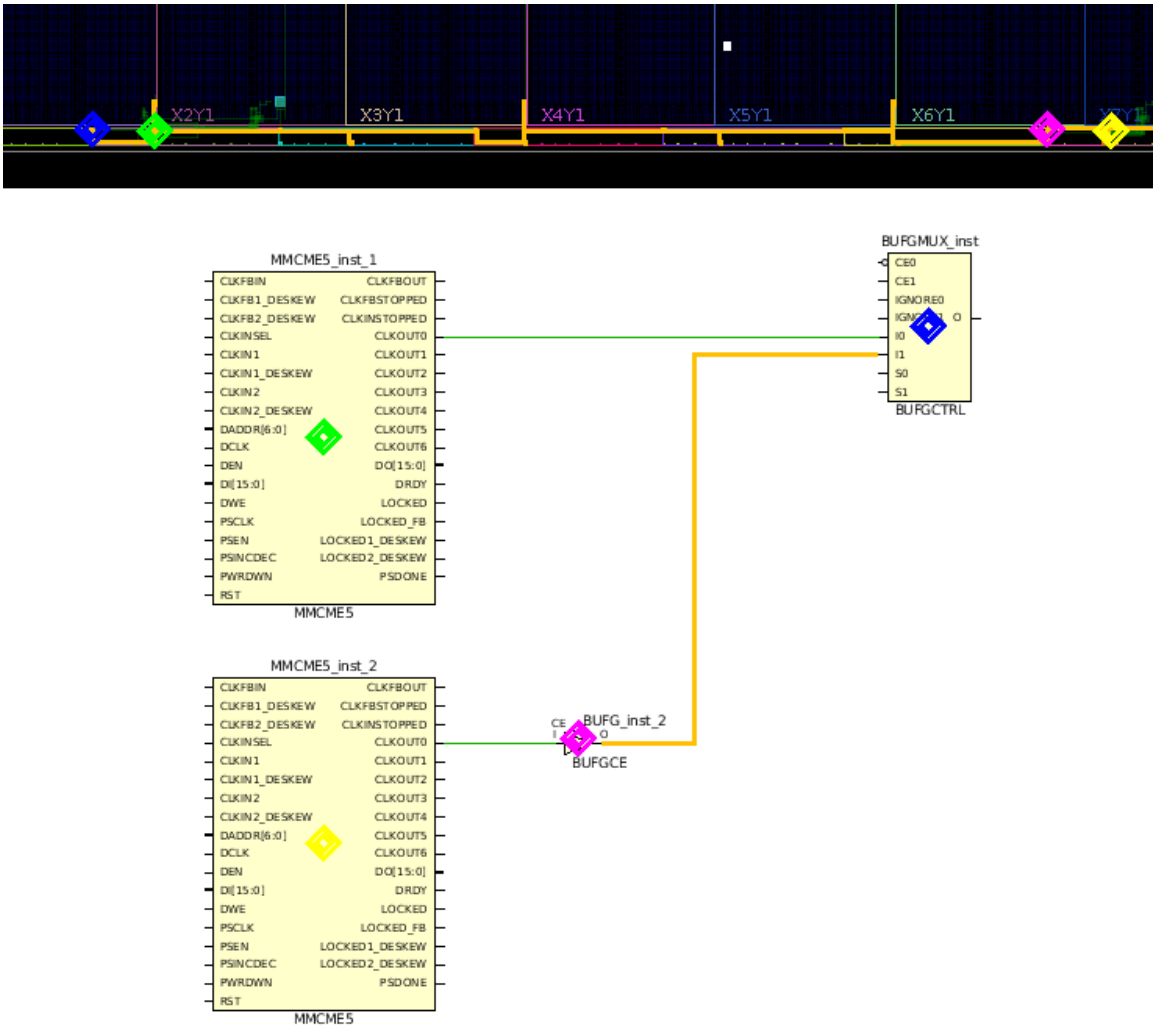
以下是用于为角点 bank 实现时钟拓扑结构的约束：

```
#PORT clkIn is in Corner Bank GCIO
set_property PACKAGE_PIN AF35 [get_ports clkIn]

#prevent opt_design removal of the BUFG_inst to BUFG_casc_inst cascade
set_property DONT_TOUCH TRUE [get_cells {BUFG_inst BUFG_casc_inst}]

#design of the clock structures
set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW [get_nets -of [get_pins
BUFG_inst/O]]
set_property CLOCK_REGION X6Y0 [get_cells BUFGCE_DIV_casc_inst]
set_property CLOCK_REGION X6Y0 [get_cells BUFG_casc_inst]
set_property CLOCK_DELAY_GROUP myDelayGrp [get_nets -of [get_pins
{BUFGCE_DIV_casc_inst/O BUFG_casc_inst/O}]]
```

- 将时钟布线到位于不同时钟区域中的另一个时钟缓冲器。

  对由其他时钟区域内的 MMCM 生成的时钟使用时钟多路复用器时，通常采用此方法。虽然其中 1 个 MMCM 可以直接驱动 BUFGCTRL (BUFGMUX)，但是另一个 MMCM 需要中间时钟缓冲器来将时钟信号布线到其他区域。下图显示了 1 个示例。

图 70：**将时钟布线到另一个时钟区域**



以下是用于为时钟多路复用器实现时钟拓扑结构的约束：

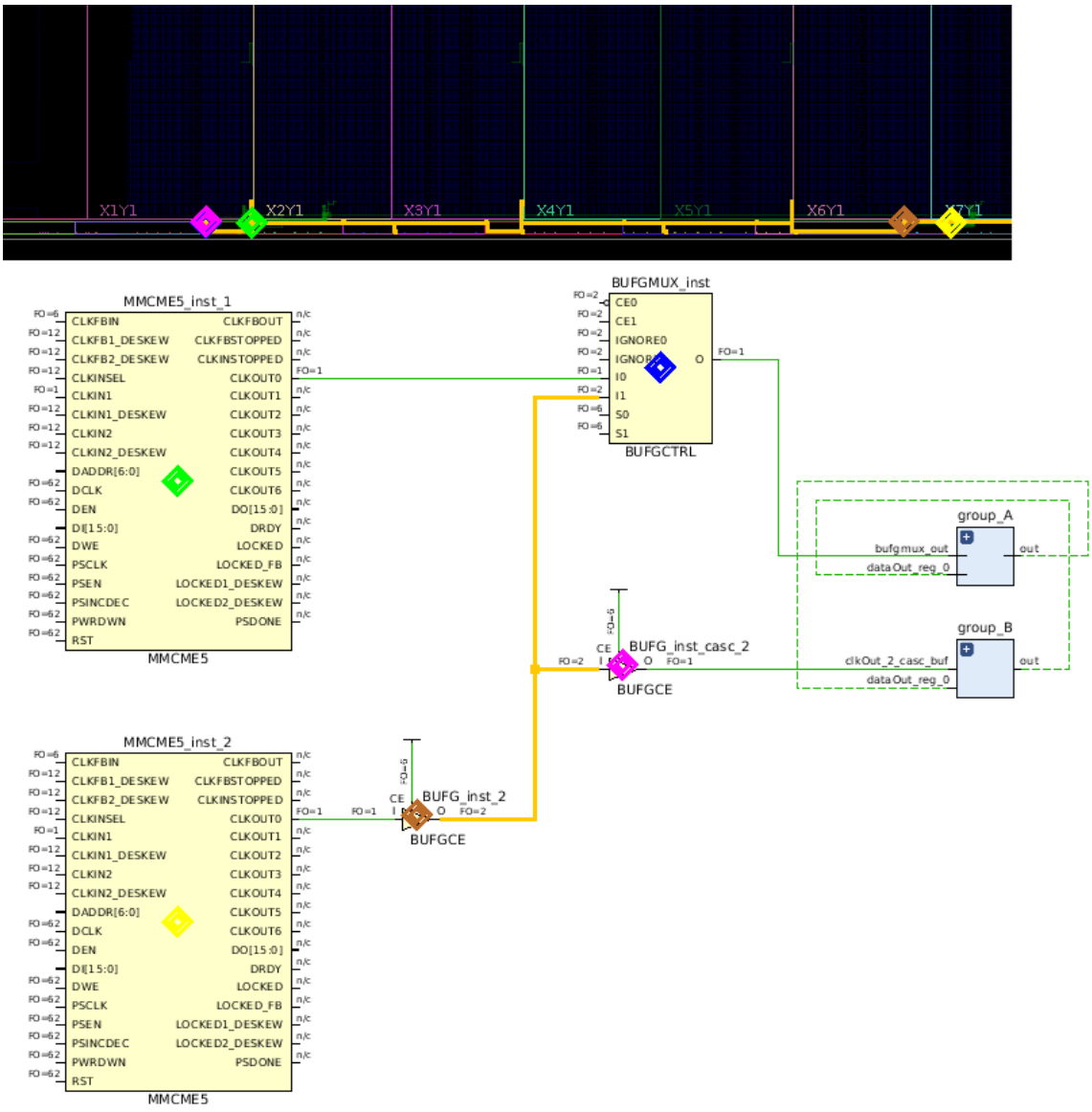
```
#PORT clkIn_1 is in XPIO ClockRegion X3Y0 GCIO
set_property PACKAGE_PIN AU27 [get_ports clkIn_1]

#PORT clkIn_2 is in XPIO ClockRegion X8Y0 GCIO
set_property PACKAGE_PIN AJ5 [get_ports clkIn_2]

#Guide placement of BUFGMUX
set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW [get_nets -of [get_pins
BUFG_inst_2/O]]
```

- 当时钟树分支之间存在同步路径时，平衡这些分支之间的时钟缓冲器等级数。

  以驱动 group_A（通过位于其他时钟区域内的 BUFGCTRL 驱动的时序单元）和 group_B（时序单元）的 MMCME5_inst_2 输出上的时钟为例。为了更好匹配分支之间的延迟，请为 group_B 插入 BUFGCE 并将其布局在 BUFGCTRL 所在的时钟区域内。这样可确保 group_A 与 group_B 之间的同步路径的偏差量可控。下图显示了 1 个示例。

注释：Vivado 工具逻辑最优化命令 `opt_design` 无法确认时序时钟与时钟网络分支之间的时序关系。因此，`opt_design` 会尽可能删除更多级联时钟缓冲器或冗余时钟缓冲器。在此示例中，`opt_design` 会移除 BUFG_inst_casc_2，除非您在其中设置 `DONT_TOUCH="TRUE"` 属性。如果在时钟树分支之间只有异步路径，那么只要接收端时钟域上存在正常的同步电路，就不需要平衡这些分支。

图 71：**为时钟区域之间的同步路径平衡时钟树**



以下是用于实现时钟树平衡电路的约束：

```
#PORT clkIn_1 is in XPIO ClockRegion X3Y0 GCIO
set_property PACKAGE_PIN AU27 [get_ports clkIn_1]

#PORT clkIn_2 is in XPIO ClockRegion X8Y0 GCIO
set_property PACKAGE_PIN AJ5 [get_ports clkIn_2]

#allow for routing from BUFG_inst_2 (X8Y0) to BUFG_inst_casc_2 (X3Y0) and
prevent optimization
set_property CLOCK_DEDICATED_ROUTE SAME_CMT_ROW [get_nets -of [get_pins
```

```
BUFG_inst_2/O]]
set_property CLOCK_REGION X3Y0 [get_cells BUFG_inst_casc_2]
set_property DONT_TOUCH TRUE [get_cells BUFG_inst_casc_2]

#balance output of BUFGMUX and BUFG_inst_casc_2, both placed in X3Y0
set_property CLOCK_DELAY_GROUP myDelayGrp [get_nets -of [get_pins
{BUFG_inst_casc_2/O BUFGMUX_inst/O}]]
```

- 构建时钟多路复用器。

  为减少插入延迟和偏差变动，AMD 建议使用级联时钟缓冲器时遵循以下准则：

  - 将级联缓冲器保持在相同或相邻时钟区域内。
  - 平衡时钟树分支时，将相同等级的所有时钟缓冲器都分配到相同时钟区域内。

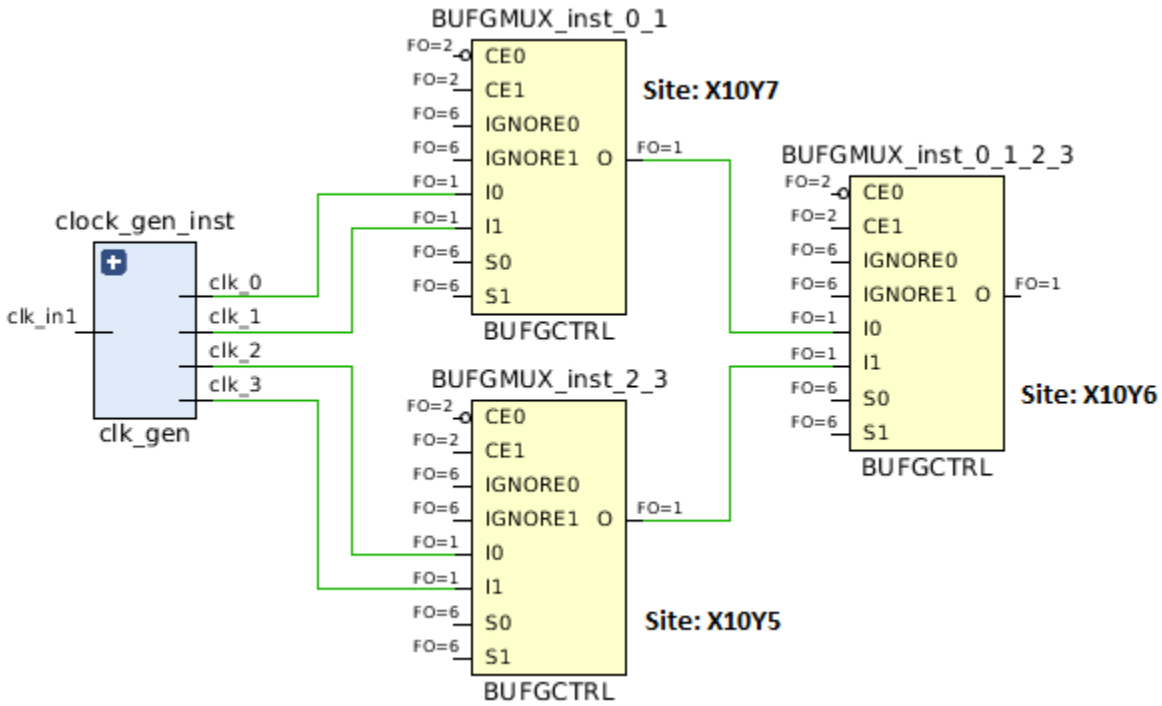
    注释：如果确实有必要，AMD 建议使用 2 个级联 BUFGCTRL 代替级联 BUFGCE。使用专用布线即可将 2 个相邻 BUFGCTRL 加以级联，当这 2 个 BUFGCTRL 都布局在相同时钟区域内时可最大程度降低延迟。

时钟多路复用

您可以使用并行和级联 BUFGCTRL 的组合构建时钟多路复用器。布局器基于时钟缓冲器站点 (site) 可用性查找最佳布局。如果可能，布局器将 BUFGCTRL 布局在相邻站点 (site) 中以便充分利用专用级联路径。如无法实现，则布局器将尝试在相邻时钟区域的相同层级中布局 BUFGCTRL。

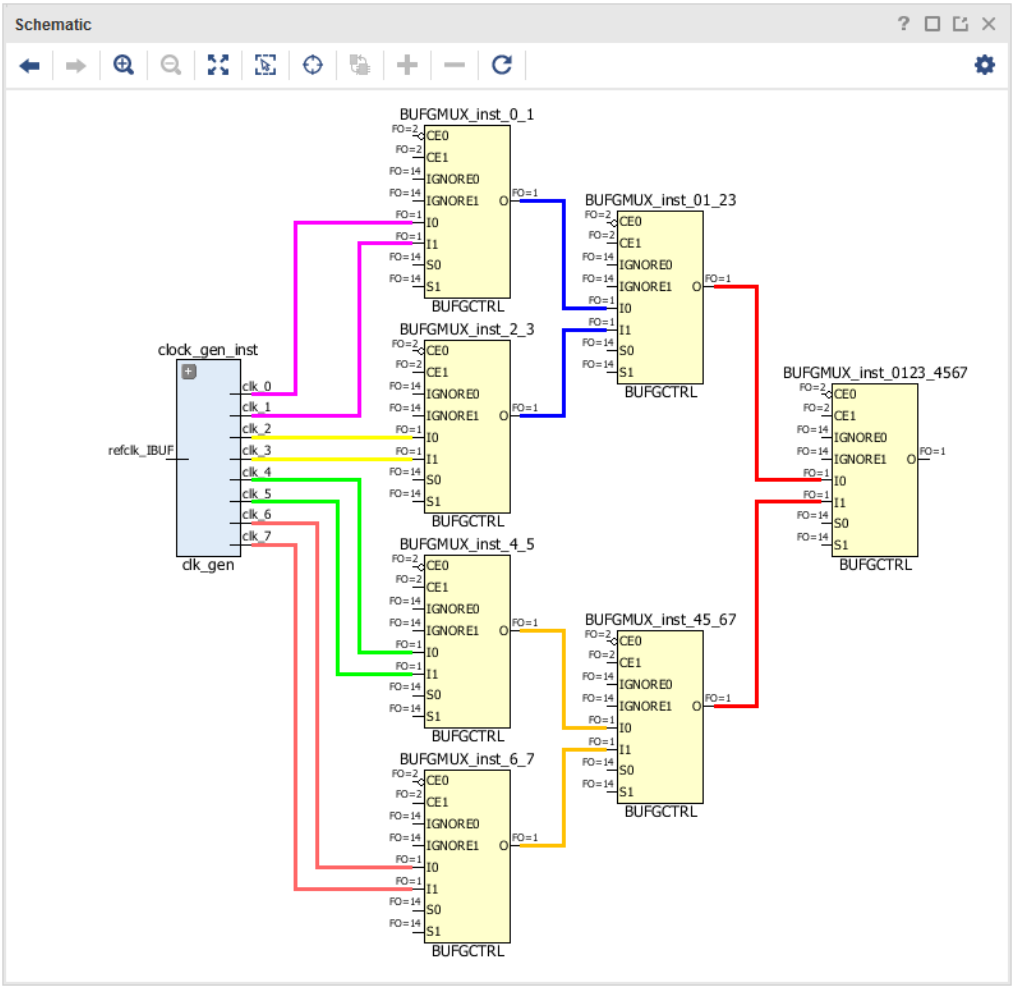
下图显示了具有平衡级联的 4:1 MUX。第 1 级 BUFGCTRL 缓冲器都布局在最后一个 BUFGCTRL (X10Y6) 的直接相邻站点 (site) (X10Y7，X10Y5)。此配置确保到达最后一个 BUFGCTRL 的所有时钟的插入延迟相近。对于 3:1 MUX，可以使用类似结构。

图 72：**使用并行 BUFGCTRL 的 4:1 MUX**



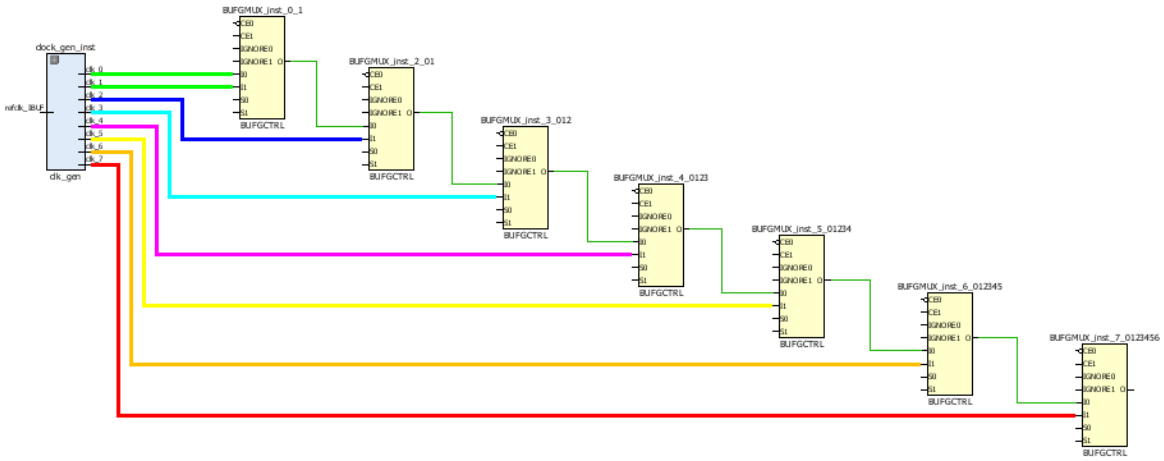
如下图所示，当创建 5:1 或更大的时钟 MUX 结构时，通常会创建 1 个对称的时钟结构。然而这只是次优解决方案，因为每个 BUFGCTRL 只有 1 条到 2 个相邻 BUFGCTRL 的级联路径，这无法为 BUFGCTRL 之间的所有连接提供最小延迟。

图 73：**非推荐的 8:1 平衡时钟 MUX 结构**



为支持更大的时钟多路复用器（从 5:1 到 8:1 MUX），AMD 建议使用级联 BUFGCTRL 缓冲器，如下图所示。此图显示了使用 7 个 BUFGCTRL 缓冲器的最优化 8:1 MUX。

图 74：**使用级联 BUFGCTRL 的 8:1 MUX**



使用基于较宽的 BUFGCTRL 的时钟多路复用器时，无法平衡时钟插入延迟，因为硬件中部分路径比其他路径更长。因此，建议仅对异步时钟多路复用采用此方法。

MMCM 反馈路径和补偿模式

下表根据 MMCM 的补偿模式来识别 Vivado 工具的反馈路径行为。其中记录的 COMPENSATION 属性值仅作参考，Vivado 工具会根据电路拓扑结构自动选择相应的补偿。

表 8：**MMCM 补偿模式**

COMPENSATION	Vivado 工具行为
BUF_IN	针对由反馈路径全局时钟缓冲器和直接连接到 CLKOUT0 管脚的所有全局时钟缓冲器所驱动的信号线分配相同的时钟根。Vivado 工具不会自动将插入延迟与其他 MMCM/XPLL 输出匹配。 要执行与其他 MMCM/XPLL 输出管脚的延迟匹配和时钟根匹配，需要使用 CLOCK_DELAY_GROUP 和 USER_CLOCK_ROOT 约束。
INTERNAL	INTERNAL 补偿不需要反馈路径。没有必要使用时钟设置资源来获取含全局时钟缓冲器的反馈路径。此行为对时钟设置使用率较高的设计中的 QoR 可能产生不利影响。 Vivado 工具会尝试删除不必要的反馈路径全局时钟缓冲器。在某些情况下，Vivado 工具无法删除不必要的反馈路径全局时钟缓冲器，您必须手动删除反馈路径全局时钟缓冲器。
EXTERNAL	如果反馈板载走线与外部组件的走线相匹配，那么可配置 MMCM 以进行外部纠偏。外部延迟值是使用以下 XDC 约束设置的： `set_external_delay -from <output_port> -to <input_port> <external_delay_value>` Vivado 工具在计算 MMCM 补偿延迟时会使用外部延迟值。

注释：XPLL 和 DPLL 不含 COMPENSATION 属性。使用检相器纠偏电路时，DPLL 能够通过 ZHOLD 属性为整个 HDIO bank 的所有 I/O 寄存器提供负的保持时间值。XPIO bank 不支持 DPLL 的 ZHOLD 功能。

源自 GT 的互连结构时钟

BUFG_GT/MBUFG_GT 缓冲器可驱动结构中的任意负载，并包含可选分频器以供您用于对来自 GT*_QUAD 的时钟进行分频。这样就不再需要使用额外的 MMCM/XPLL/DPLL 或 BUFGCE_DIV 来对时钟进行分频。

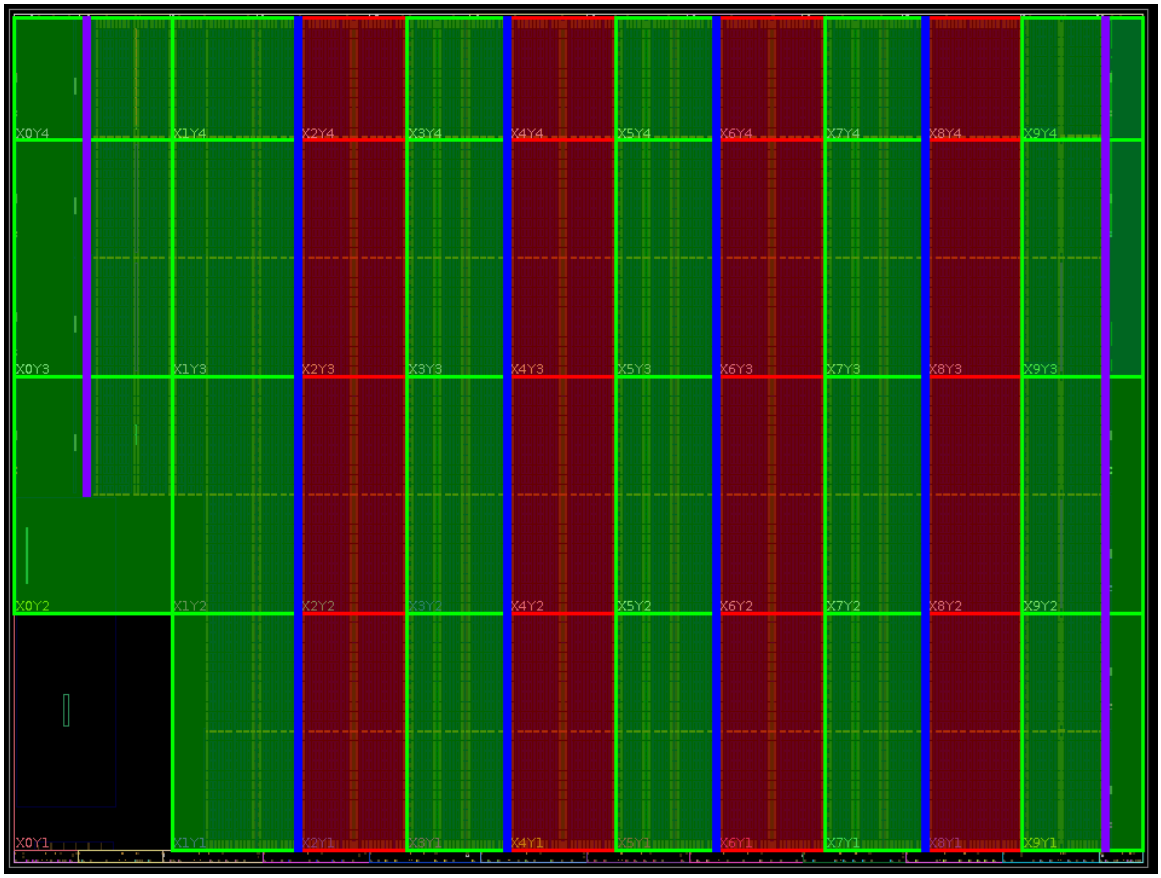
含 GT 资源的时钟区域还可直接获取 DPLL 资源，以供您用于频率综合和时钟网络纠偏。

USER_CLOCK_ROOT 分配

Versal 器件中的 USER_CLOCK_ROOT 分配与 UltraScale 器件有细微差异，因为时钟架构发生了变化。在 Versal 器件中，并非所有时钟区域都具有垂直时钟轴，这些时钟区域无法用于 USER_CLOCK_ROOT 分配。

下图显示了垂直时钟轴的位置。以蓝色高亮显示的时钟轴与位于 2 个时钟区域边界处的垂直 NoC 列相邻。Vivado 工具仅支持将 USER_CLOCK_ROOT 置位于这些时钟轴的左侧时钟区域内。器件左侧和右侧的收发器四通道列中的垂直时钟轴以紫色高亮显示。您可将 USER_CLOCK_ROOT 分配至收发器四通道列中的任意时钟区域。但如果将 USER_CLOCK_ROOT 分配至最左侧收发器四通道列中的时钟区域（PS 上方）则会导致该时钟无法触及 PS 同一行上的时钟区域内的任意负载。此外，源自 XPIO 的时钟无法将其 USER_CLOCK_ROOT 分配至 PS 上方最左侧的收发器四通道列，因为该时钟无法越过 PS 布线，必须沿 VNoC 列进行必须。绿色高亮的时钟区域表示 USER_CLOCK_ROOT 分配合规，红色时钟区域则表示时钟根分配违规。在下图中，如果 USER_CLOCK_ROOT 设为时钟区域 X0Y2、X0Y3 或 X0Y4，那么该时钟就无法到达时钟区域 X*Y1 中的负载。

图 75：**合规和违规 USER_CLOCK_ROOT 分配的垂直时钟轴和时钟区域**



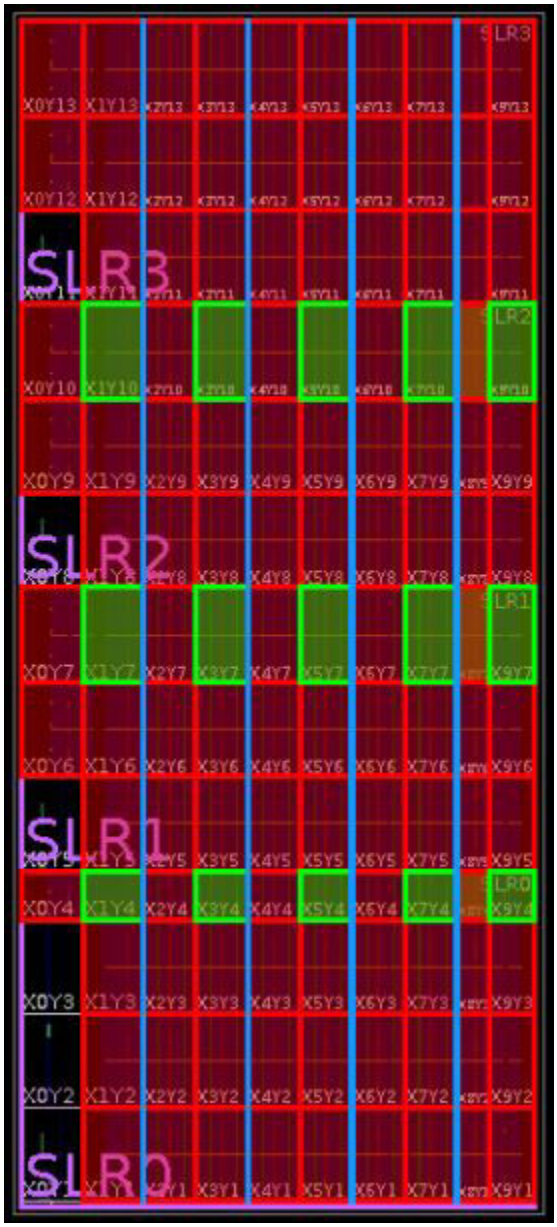
在以下示例中，USER_CLOCK_ROOT 约束违规，因为此约束设置为垂直时钟轴右侧的时钟区域：

```
set_property USER_CLOCK_ROOT X4Y3 [get_nets -of [get_pins
BUFGCE_DIV_inst/O]]
```

在此情况下，布局器会发出一条消息，并将合规的 CLOCK_ROOT 分配给 CLOCK_REGION X3Y3 中的信号线。

除了以上针对 Versal SSI 技术器件所罗列的通用 USER_CLOCK_ROOT 规则外，如有任意时钟跨多个 SLR，那么其时钟根都必须位于 SLR 边界之下的时钟区域行上。如果 USER_CLOCK_ROOT 约束设置为任何其他时钟区域，那么布局器将忽略此约束，以创建平衡的时钟树并最大程度减小时钟偏差。

图 76：**Versal SSI 技术 XCVP1802 器件中多 SLR 时钟的有效的 USER_CLOCK_ROOT 时钟区域**



最优化 CLOCK_ROOT 布局

重要提示！要全权控制布局和 USER_CLOCK_ROOT 选择，AMD 建议您创建包含时钟网络的所有负载的 Pblock。然后，将 USER_CLOCK_ROOT 分配给包含垂直时钟轴的时钟区域。大多数情况下，Vivado 布局器会为设计选择最优化 CLOCK_ROOT，无需手动进行 USER_CLOCK_ROOT 分配。

在某些情况下，USER_CLOCK_ROOT 分配可能只是次优解决方案，因为并非所有负载均以它为中心，并且由此造成时钟树不平衡。在此情况下，布局器会忽略 USER_CLOCK_ROOT 约束，并对时钟信号线分配最优化的 CLOCK_ROOT。

例如，USER_CLOCK_ROOT 置位于 X3Y2，但负载布局在 X3Y3 中。在这 2 个时钟区域内都存在垂直时钟轴。但最优化解决方案是针对 CLOCK_ROOT 使用时钟区域 X3Y3。布局器会发出消息以指示最优化解决方案。

增大 Fmax 的准则

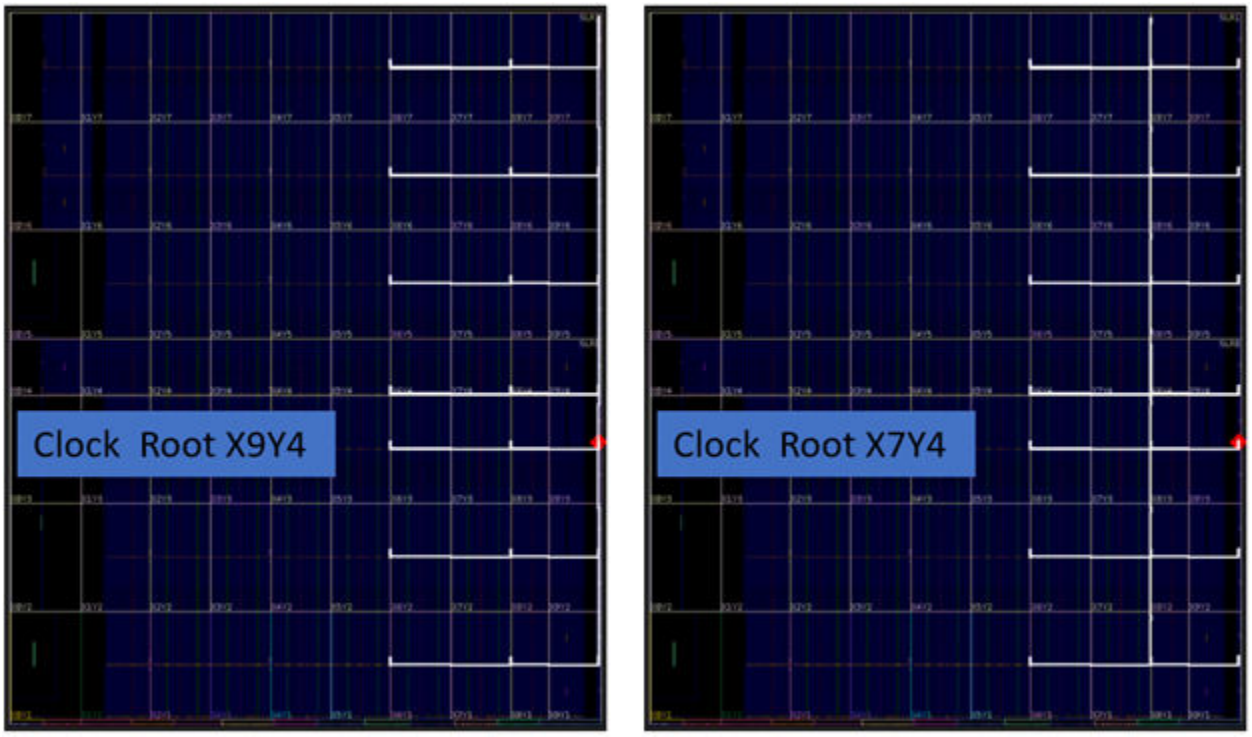
Versal 器件中全局时钟网络可达到的最大频率 (Fmax) 取决于时钟布线和时钟扩展范围。Vivado 定时器会根据时序汇总报告中最差脉冲宽度时序裕量 (WPWS) 下的时钟布线和报告裕量来计算每条时钟信号线的 Fmax。如果满足“Min Period”（最小周期）约束，则无需采取进一步操作。否则，遵循如下建议即可尽可能增大全局时钟网络可支持的频率。

时钟跨 SLR 时避免 GT 列

如果时钟布线穿过 GT 列，并且该列跨 SLR 边界，那么将导致受支持的 Fmax 比相同时钟布线延伸穿越垂直 NoC 列时的 Fmax 更低。对于横跨 SLR 边界的时钟网络，选中任一垂直 NoC 列而不是 GT 列中的时钟根即可改善性能。在以下示例中，时钟区域 X9Y4 中的 BUFG_GT 用于驱动时钟列 X6、X7、X8 和 X9 中的时钟负载。左侧时钟根 X9Y4 所得 Fmax 更低，因为其垂直分布跨越 GT 时钟列中的 SLR 边界。左侧时钟根 X7Y4 可改善 Fmax，因为它跨越垂直 NoC 列中的 SLR 边界。USER_CLOCK_ROOT 属性可用于在垂直 NoC 列中分配时钟根：

```
set_property USER_CLOCK_ROOT X7Y4 [get_nets -of [get_pins BUFG_GT_inst/O]]
```

图 77：**垂直 NoC 列对比 GT 列中的时钟根**

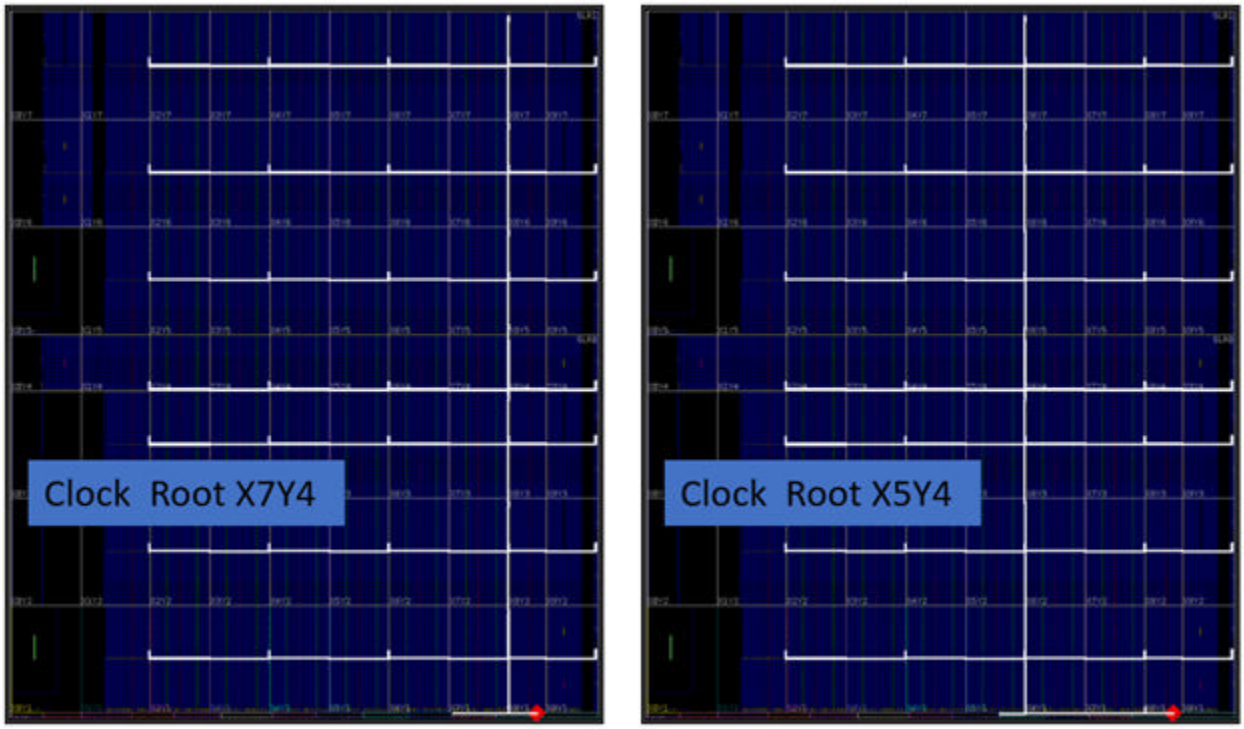


水平方向的中间时钟根

影响 Fmax 的最大时钟扩展范围是时钟布线与分布网络所遍历的所有架构时钟区域的总和。水平布线穿过 XPIO 时钟区域时，对于 Fmax 下降没有任何影响。因此，通过为给定时钟网络分配位于最中间的垂直 NoC 列中的时钟根，即可改善 Fmax。在以下示例中，时钟区域 X12Y0 中的 BUFGCE 用于驱动时钟列 X2、X3、X4、X5、X6、X7、X8 和 X9 中的时钟负载。相比于左图中更优化的时钟根 X5Y4，左侧的时钟根 X7Y4 所得 Fmax 更低，因为在水平方向上遍历了 2 个额外的架构时钟区域，由此可得最大的时钟扩展范围。USER_CLOCK_ROOT 属性可用于在最中间的垂直 NoC 列中分配时钟根：

```
set_property USER_CLOCK_ROOT X5Y4 [get_nets -of [get_pins BUFGCE_inst/O]]
```

图 78：**中间时钟根所得比较结果**



时钟相移建模

时钟相移对应于因时钟路径内的特殊硬件所导致的参考时钟相关的延迟时钟波形。在 AMD 器件中，时钟相移通常是由 MMCM、XPLL 或 DPLL 原语引入的，前提是这些原语的输出时钟属性 CLKOUT*_PHASE 为非零值。

通过模拟纠偏，时钟相移即可作为时钟波形中的更改 (PHASESHIFT_MODE=WAVEFORM) 来建模，或者也可以通过 MMCM 或 XPLL 作为延迟 (PHASESHIFT_MODE=LATENCY) 来建模。在 Versal 器件中，时钟相移的默认建模方式是作为延迟通过时钟设置原语来建模。如需了解有关时钟相移建模和 PHASESHIFT_MODE 属性的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

其他注释如下：

- 对于 DPLL 或者配置的 MMCM/XPLL 带有数字纠偏时，仅支持 PHASESHIFT_MODE=LATENCY 配置。

- 如果配置的 MMCM/XPLL/DPLL 带有数字纠偏，并且包含属性 PHASESHIFT_MODE=WAVEFORM，那么 Report Methodology 会报告如下警告 (Warning)：

```
TIMING-54: The clock modifying block <MMCM/XPLL/DPLL> is configured for
digital deskew and has PHASESHIFT_MODE=WAVEFORM. This combination is
unsupported, and timing analysis will proceed by treating it as if
PHASESHIFT_MODE=LATENCY. Change the specified cell configuration to
PHASESHIFT_MODE=LATENCY and ensure that no timing constraints are written
against the expectation of PHASESHIFT_MODE=WAVEFORM.
```

PLL/MMCM 反相时钟

在 Versal 器件中，MMCM 不具有 CLKOUTxB 端口可用于生成反相时钟。要生成反相时钟，可将 CLKOUTx 输出相移 180 度。

将该时钟相移 180 度后，定时器就会对相移进行建模，将其作为时钟波形沿中的变更或者作为时延（无需更改时钟波形）来处理。您可使用 MMCM 上的 PHASESHIFT_MODE 属性来控制定时器对相移进行建模的方式。PHASESHIFT_MODE 属性的默认值为 LATENCY。

欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

注释：配置有数字化纠偏的 DPLL 和 MMCM/XPLL 仅支持 PHASESHIFT_MODE=LATENCY。

如果在 I/O 时序内不使用相移，而改为在互连结构内部的时序路径中使用相移 (PHASESHIFT_MODE=LATENCY)，则可能导致非相移时钟与相移时钟之间出现意外的建立/保持时间要求。当时钟发生 180 度相移以生成互连结构反相时钟并替换 Versal 器件中缺失的 CLKOUTxB 时钟后，就会发生这种状况。

以下选项可用于在非相移时钟与相移时钟之间生成正确的建立/保持时间要求：

- 添加多周期路径以调整定时器所使用的时钟沿：

```
set_multicycle_path -from [get_clocks clk] -to [get_clocks
clk_phase_shitfed] -setup -end 0
```

- 将定时器使用的相移模型从 LATENCY 更改为 WAVEFORM。在如下情况下，无需多周期路径：

```
set_property PHASESHIFT_MODE WAVEFORM [get_cells <MMCM>]
```

SelectIO I/O 逻辑时钟设置

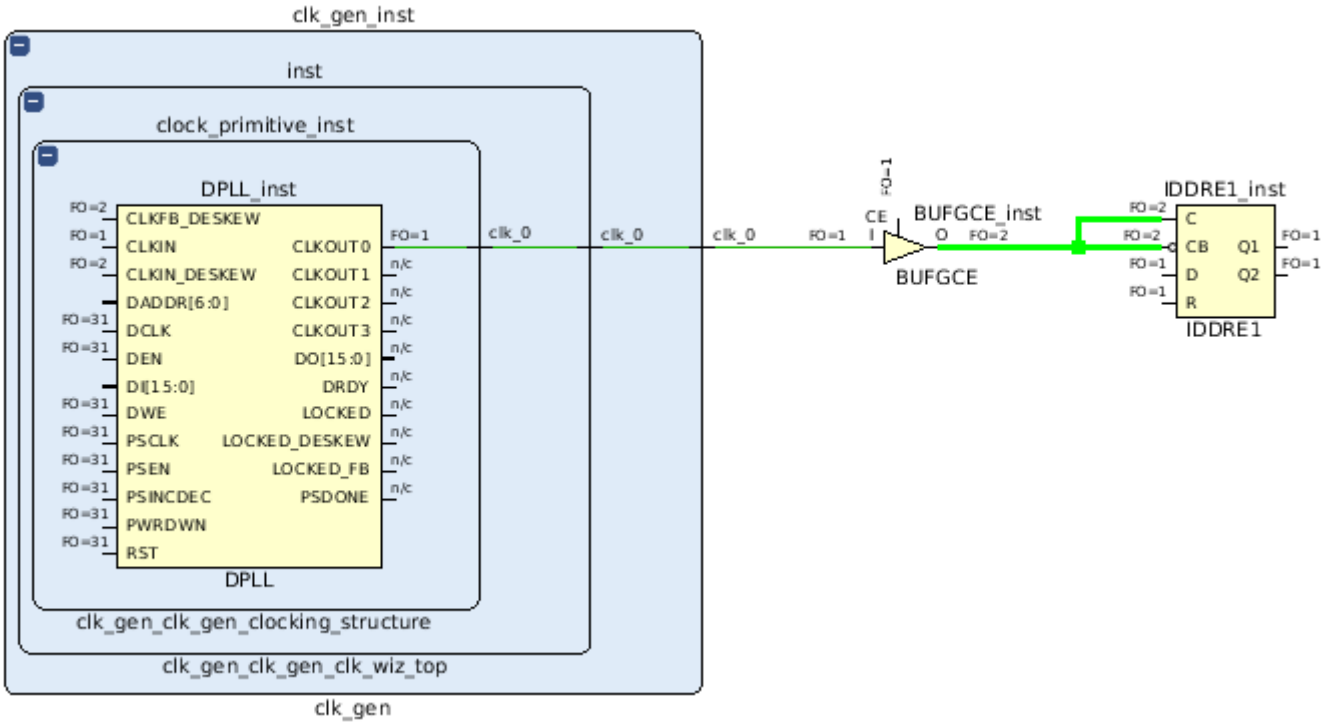
Versal 器件 SelectIO I/O 逻辑原语对于时钟管脚之间的最大偏差存在要求。对 SelectIO I/O 逻辑原语使用最优化时钟设置拓扑可防止出现最大偏差违例、改进 Versal 器件与互连结构逻辑之间的接口时序，并减少使用的时钟资源。

IDDRE1 时钟设置

对于 Versal 器件中的 IDDRE1 时钟，在时钟与反相时钟管脚之间存在最大偏差要求。为满足最大偏差要求，AMD 建议在使用局部反转时，对时钟和反相时钟管脚使用单一信号线。

下图显示在 IDDRE1 的 CB 管脚上使用局部反转的最优化配置。使用最优化配置可以保证使用较少的全局时钟资源时能够满足最大偏差要求。

图 79：**IDDRE1 的最优化时钟拓扑**



同步 CDC

当设计在源自相同 MMCM/XPLL/DPLL 的时钟之间包含同步 CDC 路径时，可使用以下技巧来更好地控制这些路径上的时钟插入延迟和偏差，从而控制时序裕量。

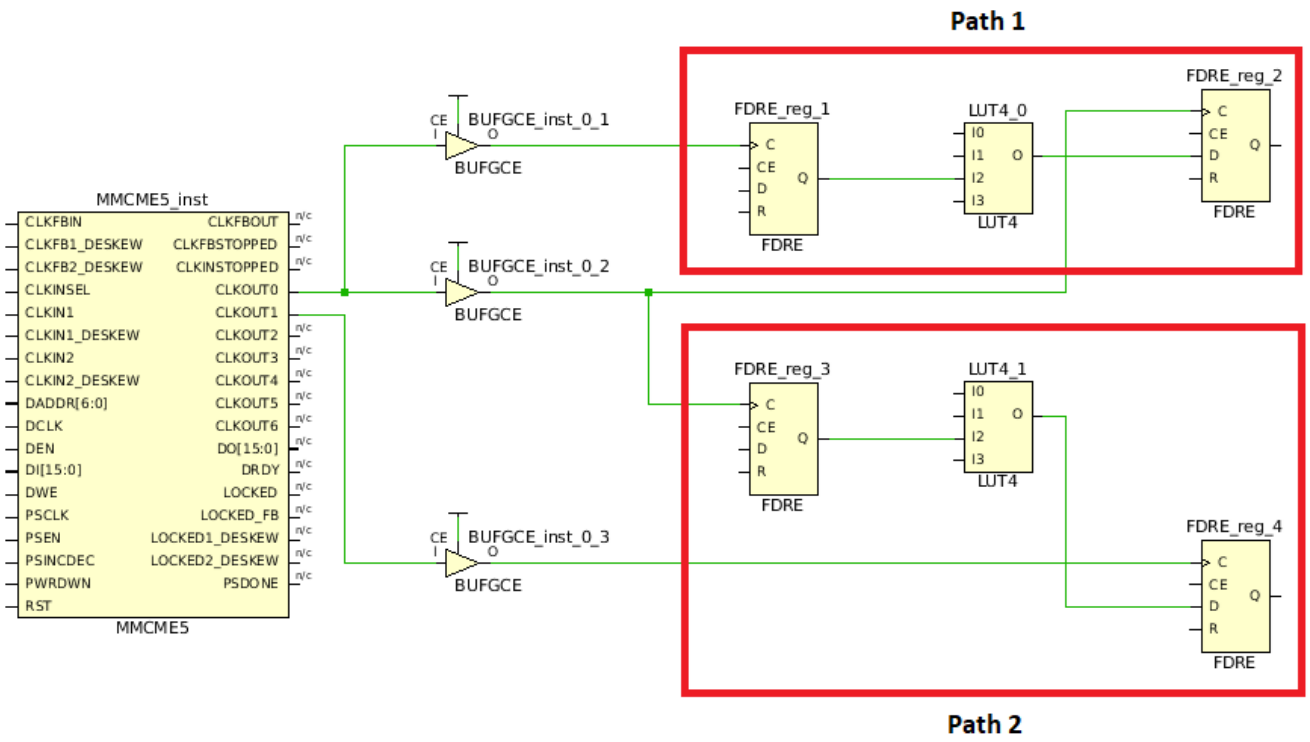
重要提示！对于位于源自不同 MMCM/XPLL/DPLL 的时钟之间的 CDC 路径，MMCM/XPLL/DPLL 之间的时钟插入延迟更难以控制。在此情况下，AMD 建议您将这些时钟域交汇作为异步时钟来处理并执行相应的设计更改。

重要提示！如果 CDC 路径位于输入时钟和输出时钟之间，或者位于两个或两个以上 MMCM 或 XPLL 原语的输出时钟之间，请访问此链接以参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003) 中的相应内容，确保原语配置允许时钟间的安全定时。如需了解 DPLL 的信息，请访问此链接以参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003) 中的相应内容。

如果 2 个时钟源自相同 MMCM/XPLL/DPLL 的不同输出管脚，那么对这 2 个时钟之间的路径进行时序约束时，MMCM/XPLL/DPLL 相位误差会导致路径的时钟不确定性增加。对于使用高时钟频率的设计，相位误差可能导致建立时间和保持时间的时序收敛出现问题。

下图显示了含相位误差和不含相位误差的路径示例。路径 1 为 CDC 路径，此路径由连接到相同 MMCM 输出的 2 个缓冲器进行时钟设置，并且不含相位误差。路径 2 由源自 2 个不同 MMCM 输出的 2 个时钟进行时钟设置，并且包含相位误差。

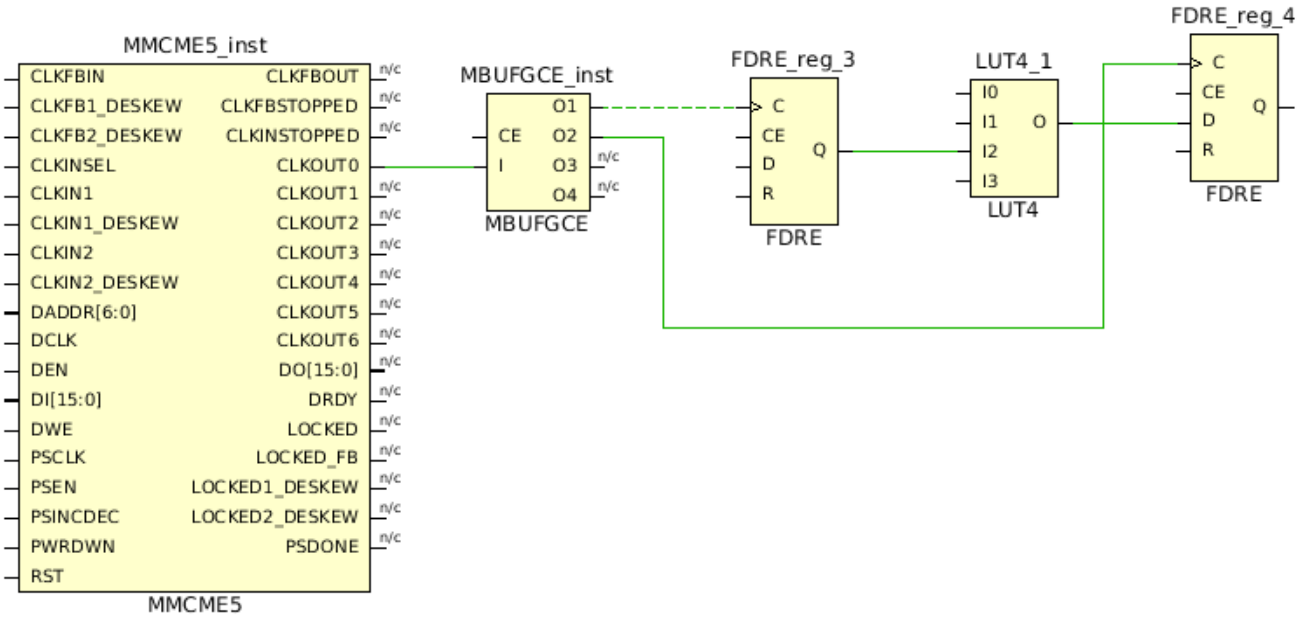
图 80：**MMCM 和相位误差**



当源自相同 MMCM/XPLL/DPLL 的 2 个同步时钟具有简单的周期比 (/2 /4 /8) 时，可以使用单个 MMCM/XPLL/DPLL 输出（连接到单个 MBUFGCE 或者连接到 2 个 BUFGCE_DIV 缓冲器）来防止 2 个时钟域之间出现相位误差。MBUFGCE 单元可执行简单的时钟分频 (/1 /2 /4 /8) 和简单的时钟倍频 (*2)。BUFGCE_DIV 缓冲器可执行简单的时钟分频 (/1 /2 /4 /8)。BUFGCE_DIV 还可提供其他分频比率 (/3 /5 /6 /7)，但这需要修改时钟占空比，导致混合时钟沿时序路径变得更为困难。

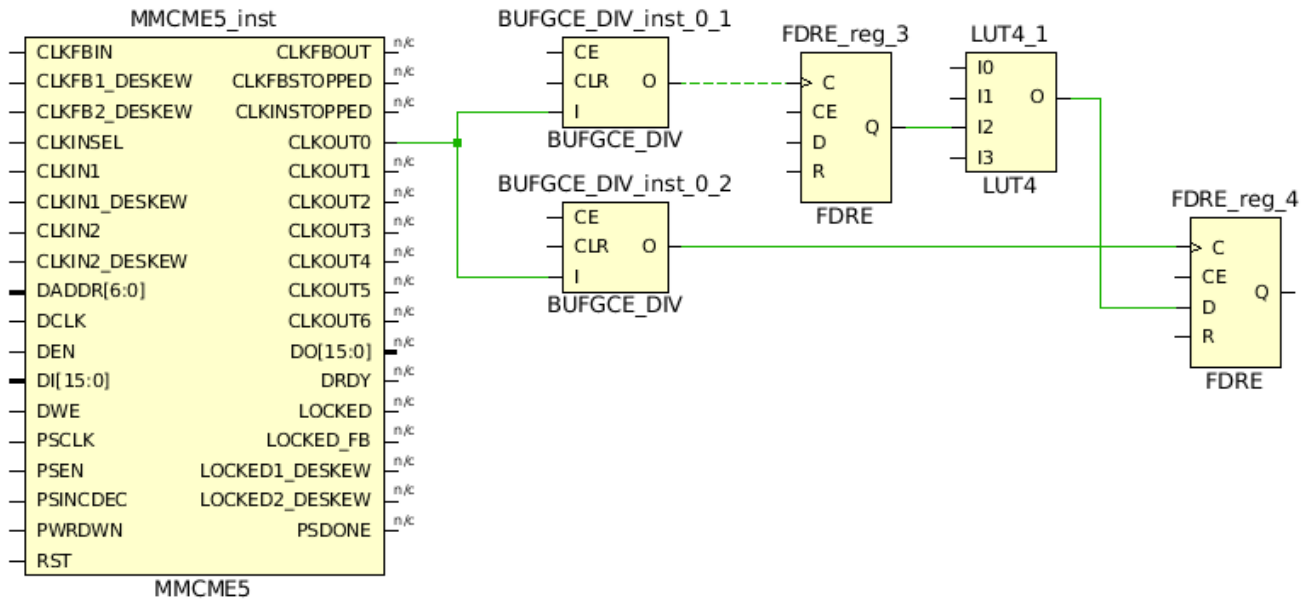
下图显示了单个 MBUFGCE 单元，此单元在 O1 管脚上将 CLKOUT0 时钟除以 1，在 O2 管脚上将此时钟除以 2。MBUFGCE 单元无需在逻辑输出信号线上增加时钟约束，因为此信号线在单一时钟轨道上布线，直至达到叶级分频器为止。

图 81：**同步 CDC，其中 MBUFGCE 连接到 1 个 MMCM 输出**



下图显示了 2 个 BUFGCE_DIV，这 2 个缓存分别将 CLKOUT0 时钟除以 1 和 2。

图 82：**同步 CDC，其中 BUFGCE_DIV 连接到 1 个 MMCM 输出**



注释：由于 BUFGCE 和 BUFGCE_DIV 的单元延迟不同，AMD 建议针对 2 个同步时钟（例如，2 个 BUFGCE 缓冲器或 2 个 BUFGCE_DIV 缓冲器）使用相同的时钟缓冲器。

重要提示！如果并行 BUFGCE_DIV 单元中设置的 BUFGCE_DIVIDE 属性值大于 1，那么为确保此类单元之间的时序约束安全，这 2 个缓冲器必须使用相同使能信号 (CE) 和相同复位信号 (RST)。否则，在硬件中分频后的时钟之间可能出现相移，而 Vivado 工具不会报告此现象。

要在源自相同 MMCM 或 PLL 的多个时钟之间自动实现平衡，请在需平衡的时钟缓冲器所驱动的信号线上设置相同的 CLOCK_DELAY_GROUP 属性值。以下是提供的附加建议：

- 避免在过多时钟上设置 CLOCK_DELAY_GROUP 约束，因为这会给时钟布局器施压，导致出现次优解决方案或误差。
- 将 GCLK_DESKEW 约束（值为 OFF）与 CLOCK_DELAY_GROUP 约束搭配使用，以最大程度降低并匹配时钟信号线上的插入延迟。
- 复查“Timing Summary Report”（时序汇总报告）中的关键同步 CDC，以判定哪些时钟必须匹配延迟以满足时序约束要求。
- 对于具有完全相同的时序拓扑结构并且时序要求较为严格的同步时钟组，请限制使用 CLOCK_DELAY_GROUP。

重要提示！AMD 建议使用 Clocking Wizard 来创建最优化时钟结构，此结构将配合相关时钟分组约束混用 BUFGCE 和 BUFGCE_DIV。

GT 接口时钟

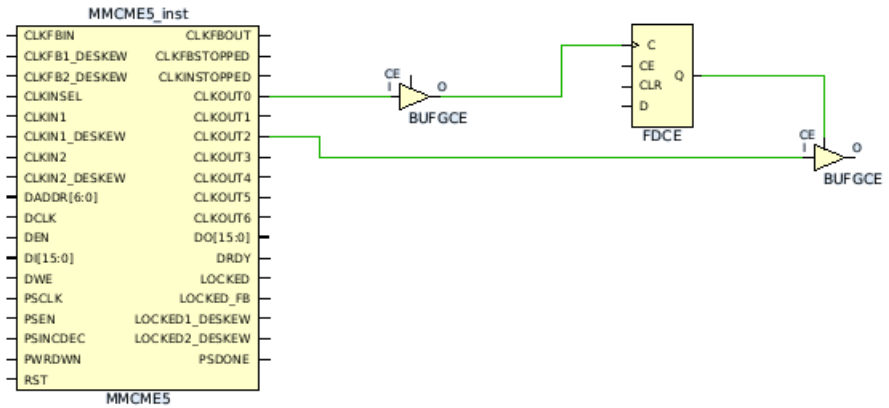
每个 GT 接口都需要数个时钟，包括位于 1 个或多个 GT 四通道内的绑定 GT*_QUAD 单元之间共享的部分时钟。GT 四通道位于 Versal 器件中左侧时钟区域和最右侧时钟区域内。每个时钟区域都包含 2 个 GT 四通道、24 个 BUFG_GT/MBUFG_GT 时钟缓冲器和 1 个 DPLL。大部分 GT 时钟扇出较低，负载采用局部布局方式布局在时钟区域内关联 GT*_QUAD 旁。部分 GT 时钟可驱动整个器件上的负载，并且需要在许多时钟区域内使用时钟布线资源。

XPIO 全局时钟缓冲器时钟使能时序

即使时钟频率较低，也同样可能难以满足全局时钟缓冲器使能管脚上的建立时间要求。下列因素相结合可能导致建立时间路径出现困难：

- 使能时钟沿太晚，其中发送时钟使用全局时钟布线，随后布线从触发器开始，跨边界逻辑接口 (BLI) 到达使能管脚。
- 捕获时钟沿太早直接到达门控的全局时钟缓冲器输入管脚，布线未穿过全局时钟网络。

图 83：**全局时钟缓冲器时钟使能电路**



您可使用以下技巧来改善全局时钟缓冲器使能管脚的时序：

- 在使用三阶段内部同步器的全局时钟缓冲器上使用 HARDSYNC 功能特性。这样即可消除时序要求，但是会在时钟输出处引发三个或四个时钟周期的时延。
- 使用负相移时钟来驱动使能控制逻辑，并拉入发送时钟沿。

- 在用于驱动使能控制逻辑的时钟上使用 CLOCK_LOW_FANOUT 约束。这样即可将源时钟路径保持在相邻时钟区域本地，从而降低此路径上的时钟插入延迟。时钟信号线必须包含有限数量的负载，这样此约束才能正常生效。
- 在直接驱动全局时钟缓冲器的触发器上使用 BLI 约束。BUFGCE 时钟使能管脚没有关联的 BLI 触发器资源。因此，您必须使用 BUFGCE_DIV 除以 1，或者如果使用 BLI 触发器时则使用 BUFGCTRL。
- 使用级联缓冲器来驱动门控时钟缓冲器，并确保：
  - 级联缓冲器不会被优化掉
  - 级联缓冲器与门控时钟缓冲器布局在同一个 CLOCK REGION 内
  - 级联缓冲器和驱动使能控制逻辑的缓冲器达到平衡状态

注释：使用 HARDSYNC 时钟缓冲器模式时，必须确保门控的缓冲器时钟与其他设计时钟之间的相位关系不受影响，对于含整数周期比（例如，2、4、8 等）的时钟尤其如此。如果此关系可能发生更改，就必须将此时钟视作为与设计内其他时钟存在异步关系，方法是添加相应的时序约束和电路。

时钟结构设计

鉴于您已了解时钟设置判定的主要考量因素，下文将介绍如何实现设计所期望的时钟设置。

推断

Vivado 综合无需用户干预即可为时钟结构自动指定全局缓冲器 (BUFG)，适用的时钟结构数量不超过架构中允许的最大数量（除非另行指定或者由综合工具加以控制）。如前文所述，BUFG 能够提供高可控性低偏差网络，适合满足大部分时钟需求。除非设计时钟超出目标器件中 BUFG 的数量或功能，否则无需额外操作。

但是，对时钟结构应用额外控制可能有助于改善抖动、偏差、布局、功耗、最高时钟频率等方面的特性。

综合约束和属性

控制时钟资源的简单方法是使用 CLOCK_BUFFER_TYPE 综合约束或属性。综合约束可以用于：

- 防止 BUFG 推断。
- 用替代性时钟结构取代 BUFG。
- 指定其他情况下不存在的时钟缓冲器。

通过使用综合约束，无需对代码进行任何修改即可实现此类控制。

属性可置于下列任一位置：

- 直接置于 HDL 代码中，以便使其永久保存于代码中
- 作为 XDC 文件中的约束，这样无需修改源 HDL 代码就能实现此类控制

IP 的使用

某些 IP 对创建时钟结构有帮助。Clocking Wizard 和 Advanced IO Wizard 专用于帮助选择和创建时钟资源和结构，包括：

- BUFG
- BUFGCE
- BUFGCE_DIV

- BUFGCTRL
- 时钟修改块，如：
  - 混合模式时钟管理器 (MMCM)
  - XPLL
  - DPLL

较复杂的 IP（如 PCIe 或 Transceivers Wizard IP）还可能在总体 IP 中包含时钟结构。如能妥善考量其作用，即可提供额外的时钟资源。但如忽视其存在，则可能会限制设计其余部分的某些时钟选项。

AMD 强烈建议尽可能准确掌握并妥善利用设计其他部分中的所有例化的 IP 的时钟要求、功能和资源。

相关信息

IP 的使用

例化

控制时钟结构的最低级且最直接的方法是将所需时钟资源例化到 HDL 设计中。这样即可使用器件的所有可用功能，并全权掌控这些功能。使用 BUFGCE、BUFGMUX、BUFGCE_DIV 或需要额外逻辑和控制的其他时钟结构时，例化通常是唯一的选择。但即便是对简单的缓冲器而言，有时候实现期望的结果的最快方法还是直接将其例化到设计中。

将时钟资源包含在独立实体或模块内并在代码顶层或顶层附近将此实体或模块例化是一种有效时钟资源管理方式，在例化时尤其如此。通过将时钟资源置于代码顶层，就可以更方便地将其分配给设计中的多个模块。

请注意可在哪些场合以及应在哪些场合共享时钟资源。创建冗余时钟资源不仅是资源浪费，而且通常会增加功耗，造成更多潜在冲突和布局决策，导致总体实现工具的编译时间延长，使时序约束状况变得更加复杂。这也是把时钟资源置于顶层模块附近的又一重要原因。

提示：您可使用 Vivado HDL 模板来例化特定时钟原语。

相关信息

使用 Vivado Design Suite HDL 模板

控制时钟的相位、频率、占空比和抖动

本节提供了对时钟特性进行微调的技巧。

使用时钟修改块（MMCM、XPLL 和 DPLL）

您可使用 MMCM、XPLL 或 DPLL 来更改传入时钟的总体特性。MMCM 最常用于制约和控制时钟特性，例如：

- 创建更严格的相位控制
- 筛选时钟中的抖动
- 更改时钟频率
- 更正或更改时钟占空比

要使用 MMCM、XPLL 或 DPLL，必须协调多个属性以确保 MMCM 按规格范围内运行，并在其输出端提供所期望的时钟特性。因此，AMD 强烈建议您使用 Clocking Wizard 来正确配置此资源。

您还可直接例化 MMCM、XPLL 或 DPLL，从而进一步强化可控性。但是，请务必使用正确的设置来避免导致以下问题：

- 因抖动增加而导致时钟不确定性增加
- 构建的相位关系不正确
- 时序收敛实现难度增大

重要提示！使用 Clocking Wizard 配置 MMCM 或 PLL 时，默认情况下，Clocking Wizard 会尝试配置 MMCM，以使用合理的功耗特性降低输出抖动。

根据目的，您可更改 Clocking Wizard 中的设置以进一步最大程度降低抖动，从而以增加功耗为代价来改进时序。或者，您可降低功耗，但这会增加输出抖动。

使用 MMCM、XPLL 或 DPLL 时，请务必：

- 切勿使任何输入保持浮动。不建议依靠综合或其他最优化工具来锁定浮动值，因为值可能与期望值不同。
- 将 RST 连接到用户逻辑，以便通过由可靠的时钟源所控制的逻辑来对其进行断言。如果时钟中断，RST 接地就会导致问题出现。
- 在实现复位时使用 LOCKED 输出。例如，使 PLL 中完成时钟设置的同步逻辑保持处于复位状态中，直至断言 LOCKED 为止。LOCKED 信号必须达成同步后方可在设计的同步部分中使用。AMD 建议将 LOCKED 添加到处理器映射中，这样在调试时 LOCKED 即可见。
- 确认 CLKFBIN 与 CLKFBOUT 之间的连接。仅当 MMCM 输出时钟需与输入时钟保持相位对齐时，才需要在反馈路径中包含 BUFG，例如，使用 BUF_IN 补偿模式时。
- 为避免 Versal 器件中的同步时钟域交汇路径上遇到 MMCM、XPLL 或 DPLL 相位误差时序惩罚，请使用采用叶等级分频的 MBUFG* 原语。如果无法使用 MBUFG* 原语，请考虑使用来自单一 CLKOUT 端口的并行 BUFGCE_DIV 代替来自多个 CLKOUT 端口的各 BUFGCE。

建议：浏览 Clocking Wizard 中的不同设置，以确保根据总体设计目标来创建最符合期望的配置。

使用门控时钟

AMD 器件内置专用时钟网络，可提供高扇出、低偏差的时钟资源。HDL 代码中包含的高精度时钟门控技巧可能会干扰此功能并阻碍专用时钟资源的有效使用。因此，在将 HDL 写入器件时，AMD 不建议将时钟门控结构编码到时钟路径中。而应改为使用编码技巧来控制时钟，通过推断时钟使能来停止设计中的相应部分，以便满足不同功能或功耗需求。

将时钟门控转换为时钟使能

如果代码已包含时钟门控结构，或者仅供需要此类编码样式的其他技术使用，那么 AMD 建议您使用综合工具，此类工具可将布局在时钟路径中的门重新映射到数据路径中的时钟使能。这样即可更有效地映射到时钟资源，并简化进出门控域的数据电路的时序分析。例如，对 Vivado 综合使用 `-gated_clock_conversion auto` 选项来尝试自动转换为寄存器时钟使能逻辑。对于复杂的门控时钟接口，请在 RTL 代码中使用 GATED_CLOCK 属性来指导 Vivado 综合。

对时钟缓冲器进行门控

当大部分时钟网络都可分时间段关闭时，您可以使用 BUFGCE 或 BUFGCTRL 启用或禁用时钟网络。

当不同时间段内时钟可减慢时，您也可以使用这些缓冲器和附加逻辑来周期性启用时钟信号线。或者，您也可以使用 BUFGMUX 或 BUFGCTRL 将时钟源从速度较快的时钟信号切换为速度较慢的时钟信号。

这些技巧都可以有效降低动态功耗。但是根据要求和时钟拓扑，某一种技巧可能比其他技巧更为行之有效。

重要提示！以 Versal 器件为目标时，请勿使用 MMCME5 LOCKED 信号对 MMCME5 反馈路径（位于 CLKFBOUT 和 CLKFBIN 管脚间）中的全局时钟缓冲器进行门控。否则将导致 MMCME5 无法达成锁定。

控制和同步器件启动

器件完成配置后，器件将按顺序经历一系列事件并完成配置，随后进入正常工作状态。大部分配置序列中，最终步骤之一是断言全局置位复位 (GSR) 无效，随后断言全局使能 (GWE) 信号无效。在此过程中，设计处于已知的初始状态，随后经释放即可正常工作。

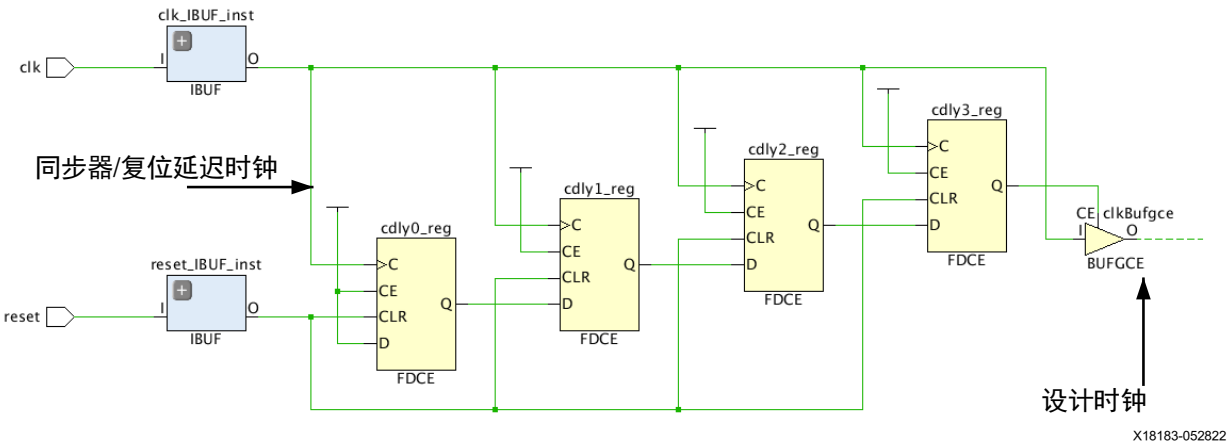
如果此释放点尚未同步到给定时钟域，或者如果时钟运行速度超过安全释放 GWE 的速度，那么部分设计可能会进入未知状态。对于某些设计，这无关紧要。而对于其他设计，这可能导致设计不稳定或者以错误方式处理初始数据集。

如果设计必须以已知状态启动，AMD 建议您采取行动，采用如下任一方法启动同步流程：

- 在设计的关键部分（例如，状态机）上使用时钟使能和/或局部复位（已同步）来确保这些设计部分的启动可控且已知。
- 使用含时钟使能功能的例化时钟缓冲器组件。

  启用设计时钟前，延迟执行复位释放操作，且延迟的周期数按需增加。以下示例演示了如何在释放复位后延迟首个设计时钟沿。通过在同步器寄存器上设置 ASYNC_REG=TRUE，所有寄存器都将布局在单个 slice 中，因此无需受全局时钟资源驱动。为防止同步器时钟上发生时钟缓冲器插入，请在输入时钟端口上使用 CLOCK_BUFFER_TYPE=NONE 属性。

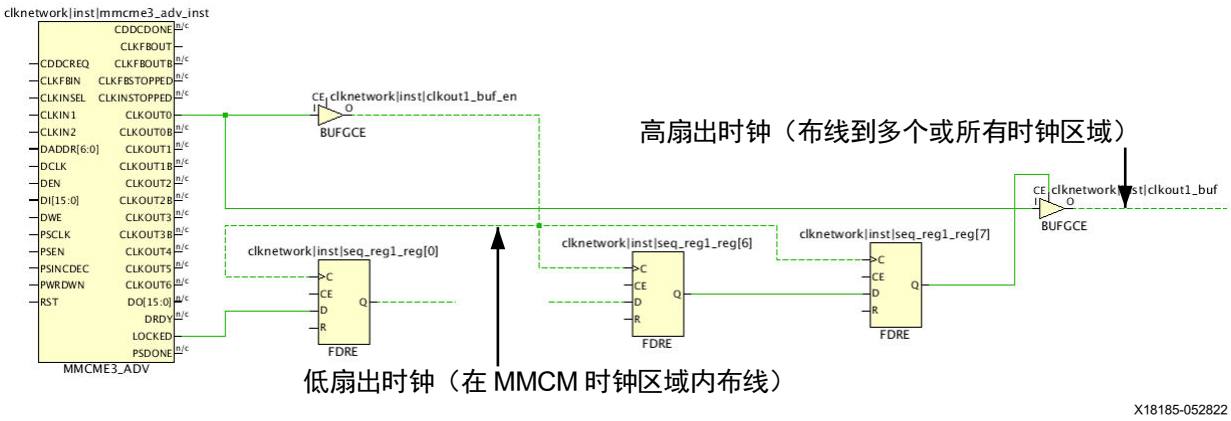
图 84：**复位安全时钟启动的同步和延迟示例**



- 使用 MMCM 时，可以从 Clocking Wizard 中选择“Safe Clock Startup”（安全时钟启动）选项，以确保只有在设计时钟稳定可靠之后才能启用设计时钟。

  以下示例显示了 MMCM LOCKED 信号的同步阶段，该信号已连接到 BUFGCE 的 CE 管脚，此管脚用于驱动用户逻辑。第二个 BUFGCE 已并行连接到高扇出 BUFGCE（用户时钟），专用于控制 BUFGCE/CE 管脚的逻辑。此拓扑结构有助于在 BUFGCE/CE 上实现时序收敛，因为它可最大限度减小同步器与 BUFGCE 管脚之间的时钟偏差。

图 85：**MMCM 安全时钟启动示例**



提示：如果 MMCM 补偿模式设置为 BUF_IN，那么来自 CLKOUT0 的所有时钟都将与反馈时钟组合，并使用相同的 CLOCK_ROOT。如果由此导致在 BUFGCE/CE 上出现时序违例，请仅在高扇出时钟与反馈时钟之间创建 CLOCK_DELAY_GROUP 约束。或者，您还可在低扇出时钟信号线上设置 CLOCK_LOW_FANOUT 约束，这将导致负载布局到与 BUFG 和 MMCM 相同的时钟区域内。

避免使用局部时钟

局部时钟是使用常规互连结构资源而不是专用全局时钟资源进行布线的时钟信号线。大多数情况下，Vivado 综合和 Vivado 逻辑最优化工具会插入时钟缓冲器以满足架构要求，或者用于具有超过 30 个时钟负载的时钟信号线。通常在如下情况下会出现局部时钟：

- 全局时钟由用互连结构逻辑实现的计数器进行分频
- 时钟门控转换不能从时钟路径中删除所有 LUT

一般情况下请避免使用局部时钟。局部时钟给实现工具带来了几个难题：

- 时钟偏差不可预测，导致难以执行时序收敛
- 增加由布线器谨慎处理的中低扇出信号线会导致潜在的可布线性问题

提示：如果局部时钟引发时序约束 QoR 问题，请尝试使用 Pblock 在一小块面积上对时钟驱动和负载进行布局规划。使用 `report_clock_utilization` 来识别局部时钟的位置，查看时钟布局，并决定如何降低其数量或影响。

Versal 器件提供的 BUFG_FABRIC 单元可用于将信号线从常规互连结构资源布线到专用全局时钟资源上。在整个器件上的各 NoC 列中存在多个 BUFG_FABRIC site，用于对高扇出信号线（例如，全局时钟资源上的复位和时钟使能）进行布线。

注意！BUFG_FABRIC 单元并非供时钟信号线用于访问全局时钟资源。如有局部时钟需要使用全局时钟网络，但无法直接访问全局时钟资源，则可使用 BUFG_FABRIC 来将此局部时钟布线到全局时钟网络上。相比于可直接访问全局时钟资源的时钟，由此生成的时钟将具备次优时钟特性。

创建输出时钟

使用 ODDR 组件即可从器件转发出时钟，以便对该器件外部的器件进行时钟设置。通过使其中一项输入保持高电平，使另一项输入保持低电平，即可轻松创建时钟并妥善控制其相位关系和占空比（例如，使 D1 保持为 0，使 D2 管脚保持为 1，即可实现 180 度相移）。通过使用置位/复位和时钟使能，还能控制时钟停止以及使时钟极性在一段时间内保持不变。

如果需要针对外部时钟进行进一步相位控制，可将 MMCM 与外部反馈补偿和/或低精度或高精度的固定或可变相位补偿配合使用。这样即可更有效地控制相对其他器件的时钟相位和传输时间，简化来自该器件的外部时序要求。

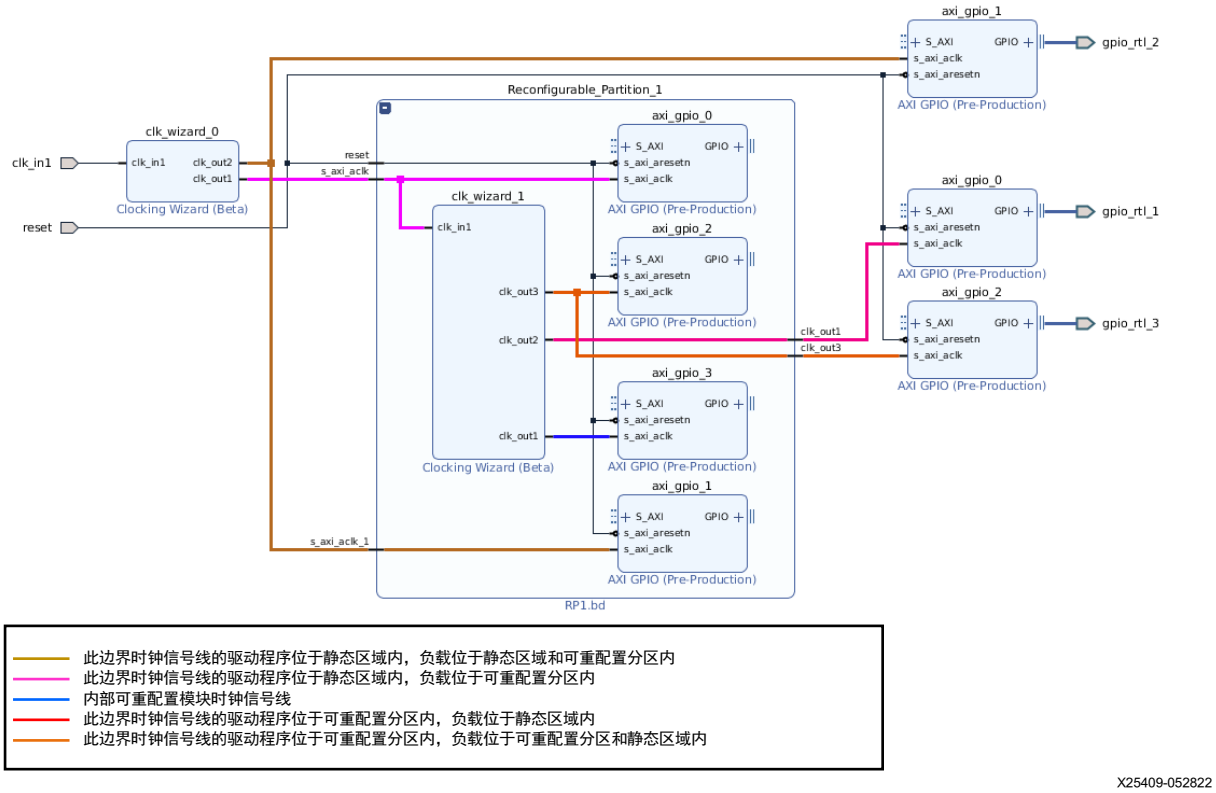
面向平台和 Dynamic Function eXchange 的时钟设置建议

本节涵盖了 Dynamic Function eXchange (DFX) 设计的时钟设置准则。一般，DFX 设计中的时钟分类为内部时钟和边界时钟：

- 可重配置模块内部时钟：含有驱动程序和所有负载的时钟位于可重配置模块 (RM) 内。
- 边界时钟：时钟所含的信号线跨可重配置模块的单元边界，如下所示：
  - 驱动程序位于静态区域内，负载位于 RM 内
  - 驱动程序位于 RM 内，负载位于静态区域内
  - 驱动程序位于静态区域内，负载分布于 RM 与静态区域之间
  - 驱动程序位于 RM 区域内，负载分布于 RM 与静态区域之间

下图显示了不同边界时钟的示例。

图 86：**DFX 时钟拼块共享**



如需了解有关 DFX 的更多信息，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)。

时钟信号线的 DFX 行为

可重配置模块内部时钟信号线

在可重配置模块 (RM) 内部时钟信号线中，时钟根布局在可重配置分区 (RP) Pblock 内的负载中心位置。此时钟根布局可以在后续实现中为 RM 内部时钟的布局布线提供更多灵活性。AMD 建议尽可能采用此方法以实现更好的偏差和最优时钟根布局。

边界时钟信号线

首次实现后，边界时钟信号线轨道将锁定。边界时钟信号线上的分区管脚位置 (PPLOC) 将分配到可重配置分区 (RP) Pblock 所涵盖的所有时钟区域内。

由于边界时钟信号线可同时驱动静态负载和 RP 负载，因此边界时钟信号线的时钟根可布局在器件中的任意位置。AMD 建议在边界时钟信号线上使用 USER_CLOCK_ROOT 约束来手动约束 CLOCK_ROOT 位置，原因如下：

- 如果边界时钟的负载主要位于静态区域中，那么时钟根可能布局在静态区域内。
- 如果首次实现在 RP Pblock 中使用训练逻辑，那么首次实现后，边界时钟信号线可能锁定，并且时钟根位置偏离中心。
- 由于边界时钟信号线分配到 RP Pblock 所涵盖的所有时钟区域，因此相比于内部 RM 时钟信号线，边界时钟的时钟插入延迟相对较高。

注释：具有 SSI 技术的 Versal 器件对于时钟根具有更严格的要求。对于跨多个超级逻辑区域 (SLR) 的时钟信号线，时钟根通常布局在器件中间的 SLR 顶部以平衡时钟树。因此，AMD 建议，针对设计中需要低时钟插入延迟的时序关键路径，应避免使用边界时钟。

适用于 Versal 自适应 SoC 的 DFX 时钟设置增强功能

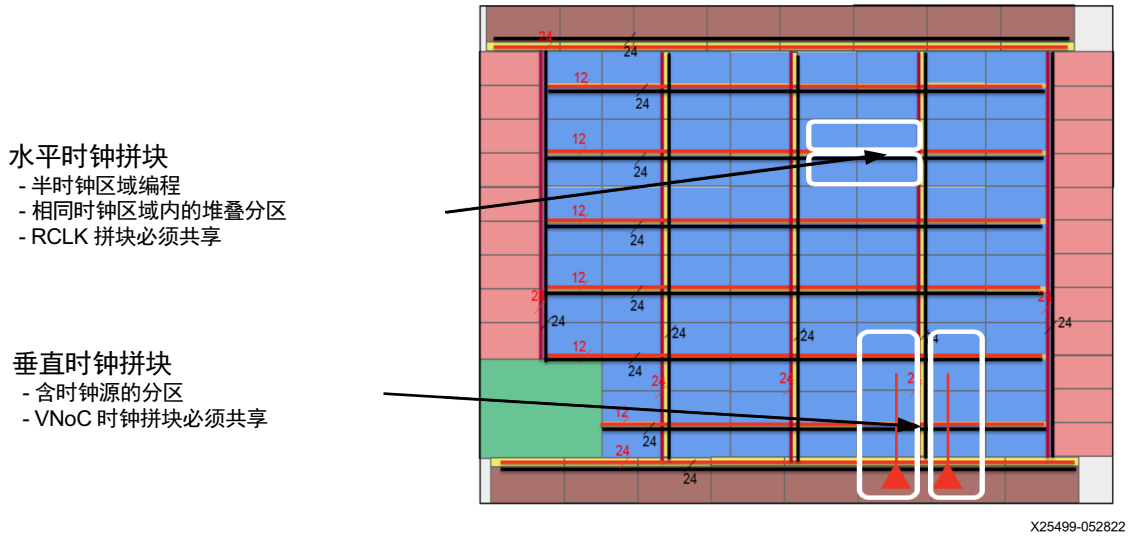
DFX 中可重配置分区之间的时钟布线拼块共享

Versal 器件允许在多个 RP 之间拆分时钟拼块。指定时钟源（例如 MMCM 和 BUFG）到 Pblock 的范围后，DFX 流程会自动使用所需的时钟设置拼块来进行布线。

下图显示了共享多个 RP 的 RCLK 行和垂直 NoC (VNoC) 列。在 RCLK 行中，2 个 RP 共享同一个时钟区域；一个位于 RCLK 行上方，另一个则位于该行下方。VNoC 拼块同样在多个 RP 间共享。

建议：如果您的设计包含 2 个以上 RP，AMD 建议在这 2 个 RP 之间保留时钟区域间隔，以免 RP 内部时钟尝试在同一 VNoC 拼块内使用时钟布线。如果尝试在单个 VNoC 列中使用时钟布线的 RP 超过 2 个，则可能出现无法布线的状况。

图 87：**RCLK 行与垂直 NoC 列共享多个 RP**



DFX 设计中的 MBUFG 原语使用规则

Versal 器件中的 MBUFG 原语支持叶级时钟分频，以降低时钟轨道使用率并改进同步 CDC 上的时序收敛。对于 DFX 设计，仅限在下列情况下才允许执行 MBUFG 最优化：静态时钟信号线、内部 RM 时钟信号线，或者在 RP 边界处仅使用未分频的 O1 输出。边界时钟信号线可以继续使用 BUFGCE_DIV/MMCM/PLL 时钟原语进行时钟分频。但相比于使用 MBUFG 原语，这可能导致 QoR 优势降低，因为 MBUFG 原语可以提供更接近叶级负载的公用时钟节点。因此，AMD 建议在 DFX 设计中的分区内使用 MMCM/PLL 时钟设置原语，将边界时钟信号线转换为内部时钟信号线，以便利用由 Vivado 工具所提供的一整套 MBUFG 最优化措施。在此情况下需要特殊处理方法以确保将 BUFDIV_LEAF 分频器复位至部分 PDI 下载期间的初始状态。如需了解有关复位 BUFDIV_LEAF 缓冲器的信息，请参阅 时钟原语。

时钟域交汇

设计中存在的时钟域交汇 (CDC) 电路会直接影响设计可靠性。您可自行设计电路，但 Vivado Design Suite 必须能够识别该电路，并且您必须正确应用 `ASYNC_REG` 属性。AMD 提供了 XPM 以确保电路设计正确，包括：

- 在 `place_design` 中驱动特定功能，以便缩短同步电路上的平均故障间隔时间 (MTBF)。
- 避免 `report_cdc` 错误和警告，通常如果迭代较长，那么在设计周期后期会出现此类错误和警告。

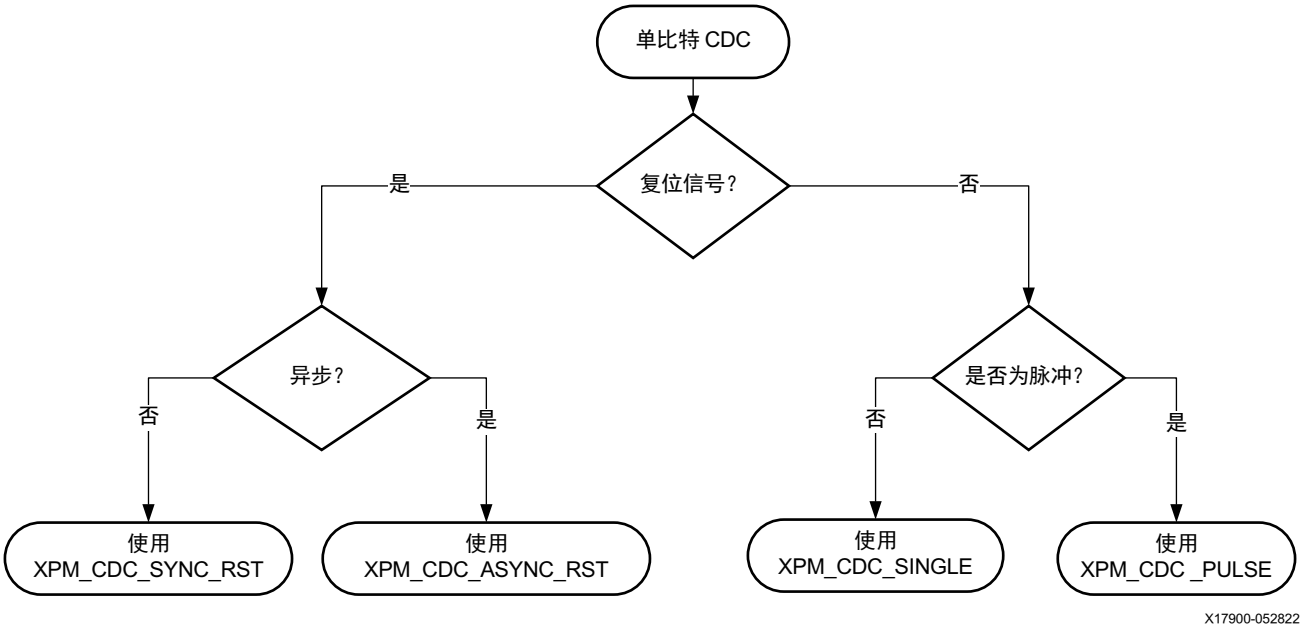
提示：对于可安全忽略的 CDC 违例，您可以使用豁免机制来豁免此类违例。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

跨 2 个异步时钟时或者尝试通过添加伪路径约束来放宽 2 个同步时钟之间的时序约束时，需要使用 CDC 电路。使用 XPM 时，可以选择单比特总线或多比特总线来跨 2 个域。

单比特 CDC

下图显示了使用单比特交汇时所需的决策。

图 88：**单比特 CDC 决策树**

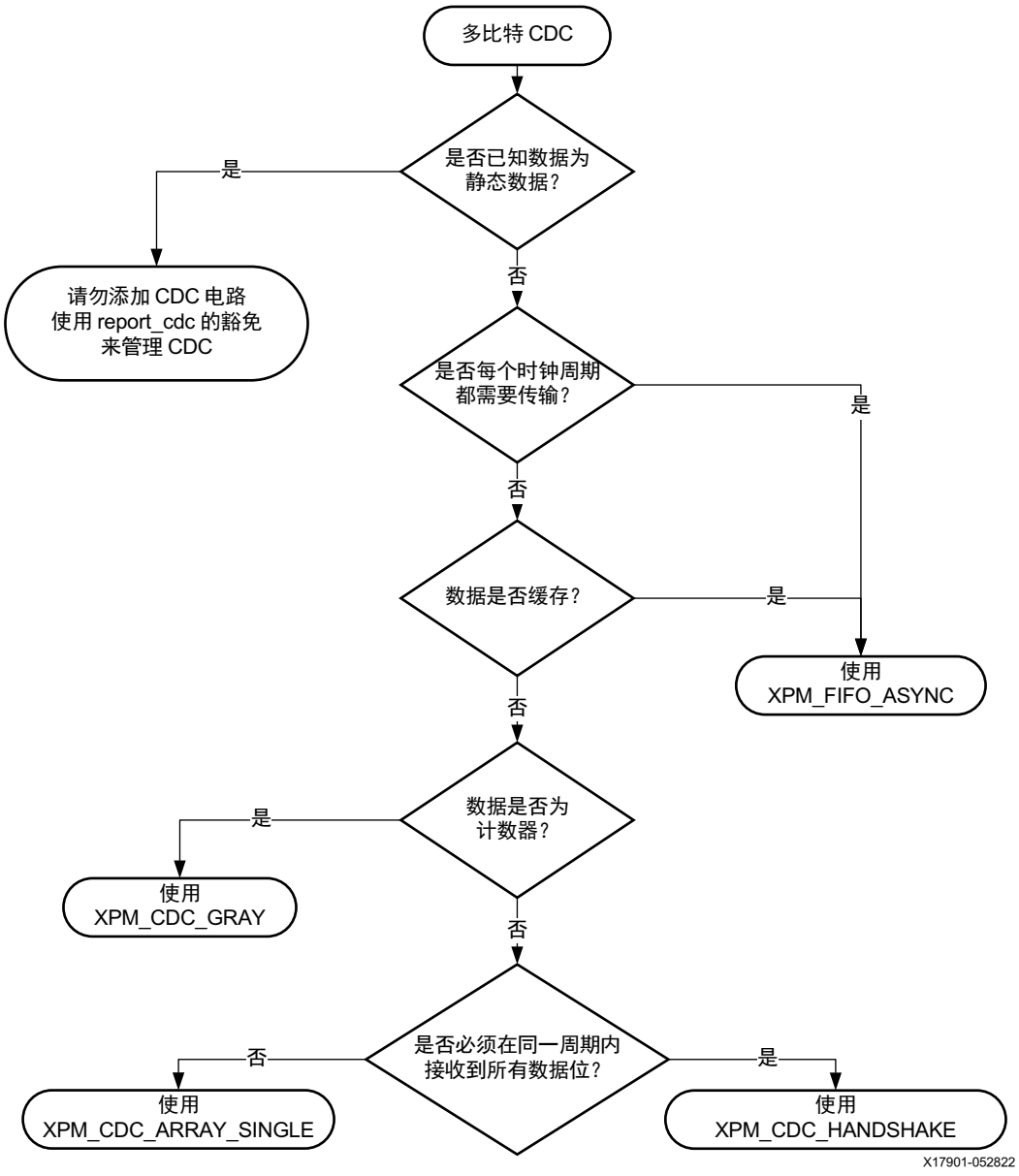


注释：如需了解有关不同单比特同步器的更多信息，请参阅对应您的器件的《库指南》。

多比特 CDC

下图显示了使用多比特交汇时所需的决策。

图 89：**多比特 CDC 决策树**



注释：如需了解有关不同多比特同步器的更多信息，请参阅对应您的器件的《库指南》。

正确执行设计约束

XPM CDC 提供了自带的 `set_max_delay -datapath_only` 约束。XPM CDC 与 `set_clock_groups` 约束不兼容，后者优先级更高并且将覆盖 XPM 中的约束。

相关信息

定义时钟组和 CDC 约束

第 5 章

使用 Vitis HLS 创建设计

AMD Vitis™ HLS 是一种高层次综合工具，支持将 C、C++ 和 OpenCL™ 函数硬连线到器件逻辑互连结构和 RAM/DSP 块上。Vitis HLS 可在应用加速开发流程中实现硬件内核，并使用 C/C++ 语言代码在 AMD Vivado™ Design Suite 中为 AMD 器件设计开发 RTL IP。

建议：如需了解有关 Vitis HLS 已知限制的信息，请参阅答复记录 75342。如果要从 Vivado HLS 工具移植到 Vitis HLS 工具，请参阅《Vitis 高层次综合用户指南》(UG1399)。

在 Vitis 应用加速流程中，在可编程逻辑中实现和最优化 C/C++ 语言代码以及实现低时延和高吞吐量所需的大部分代码修改操作均可通过 Vitis HLS 工具来自动执行。在应用加速流程中，Vitis HLS 的基本作用是通过推断所需的编译指示来为函数实参生成正确的接口，并对代码内的循环和函数执行流水打拍。Vitis HLS 还支持自定义代码以实现不同接口标准或者实现特定最优化以达成设计目标。

Vitis HLS 设计流程如下所述：

1. 编译、仿真和调试 C/C++ 语言算法。
2. 查看报告以分析和最优化设计。
3. 将 C 语言算法综合到 RTL 设计中。
4. 使用 RTL 协同仿真来验证 RTL 实现。
5. 将 RTL 实现封装到已编译的对象文件 (`.xo`) 扩展中，或者导出到 RTL IP。

Vitis HLS 方法论

Vitis HLS 支持以下流程目标：

- Vivado IP 流程：支持各种接口和数据传输协议，提供多种设计选项，灵活性较高。但您必须处理 IP 的集成和管理。如需了解有关如何在 Vitis HLS 中启用此流程的信息，请访问此链接以参阅《Vitis 高层次综合用户指南》(UG1399) 中的相应内容。
- Vitis 内核流程：支持一组特定接口，局限性较高。这种更为结构化的流程允许 HLS 块与 Vitis 可扩展平台之间的自动建构校正集成，并支持与 Xilinx Runtime (XRT) 软件栈无缝集成，显著简化硬件/软件集成流程。如需了解有关如何在 Vitis HLS 中启用此流程的信息，请访问此链接以参阅《Vitis 高层次综合用户指南》(UG1399) 中的相应内容。

以 AMD Versal™ 器件为目标时，您必须根据设计流程（传统设计流程或基于平台的设计流程）并根据整体工程中生成的块的使用方式来配置 Vitis HLS 工程，如下表所示。

表 9：**Versal 器件 Vitis HLS 工程类型**

设计流程	Vitis HLS 输出	流程目标	目标用户
传统设计流程	与其他 RTL 和 IP 块集成	Vivado IP 流程	硬件设计师
基于平台的设计流程	在可扩展平台内集成	Vivado IP 流程	硬件设计师

表 9：**Versal 器件 Vitis HLS 工程类型** (续)

设计流程	Vitis HLS 输出	流程目标	目标用户
基于平台的设计流程	作为内核链接到可扩展平台	Vitis 内核流程	硬件设计师或软件开发者

建议：在基于平台的设计流程中使用 Vitis HLS 时，AMD 强烈建议在 Vitis 内核流程中使用 Vitis HLS 来生成块，此块可使用 Vitis v++ 连接器以自动建构校正方式自动链接至可扩展平台。

定义接口

Vivado IP 流程与 Vitis HLS 中的 Vitis 内核流程之间的主要差异在于设计接口的处理方式。在 Vivado IP 流程中，接口必须由设计师进行显式管理，而 Vitis HLS 则可在 Vitis 内核流程中自动推断 AXI 接口（AXI4 存储器映射接口、AXI4-Lite 接口或 AXI4-Stream 接口）。在以任一流程启动工程前，开发者需熟悉 Vitis HLS 处理接口的方式，欲知详情，请访问此链接以参阅 Vitis HLS 用户指南 (UG1399) 中的相应内容。

在 Vitis 内核流程中使用 Vitis HLS

以 Versal 器件为目标时，基于平台的设计流程允许使用 C++ 语言对大部分器件进行编程：

- PS 上运行的软件应用可采用 C/C++ 语言来编写
- AI 引擎 graph 是以 C/C++ 编写的
- PL 内核可采用 C/C++ 来编写并使用 Vitis HLS 工具编译到硬件内

建议：AMD 建议对 PL 块使用 Vitis HLS 和 Vitis 内核流程，此流程与 AI 引擎 graph 紧密交互。使用相同的高层次语言来尽可能对大部分系统进行建模，这样可以简化团队协作、简化信息交换、加速迭代并加速仿真。

面向软件程序员的 Vitis HLS 原则

由于 Vitis HLS 允许从 C/C++ 源代码生成硬件设计，因此对于需要在 Versal 自适应 SoC 工程内创建 PL 块的软件开发者，建议使用 Vitis HLS。但为硬件编写 C/C++ 不同于为软件编写 C/C++，因为 Versal 器件可能包含来自 CPU 甚至 GPU 的底层架构。因此，使用针对 Versal 器件最优化的编程模式至关重要。“producer-consumer”（生产者 - 使用者）模式最适合 Versal 器件编程。如果您不熟悉此概念或者“data streaming”（数据串流）或“pipelining”（流水打拍）的概念，请访问此链接以参阅 Vitis HLS 用户指南 (UG1399) 中的相应内容。

设计高效内核

使用 C/C++ 来执行硬件建模和创建可以提供编码灵活性。为了最大程度提升可预测性并加速实现结果，AMD 建议采用“load-compute-store”（加载 - 计算 - 存储）编码样式，即在数据移动块与计算块之间对内核进行显式分解。这种高度结构化的编码样式最适合“producer-consumer”（生产者 - 使用者）模式，有助于实现高吞吐量设计。欲知详情，请访问此链接以参阅 Vitis HLS 用户指南 (UG1399) 中的相应内容。

将 HLS 内核连接到 AI 引擎计算图

PL 内核可使用串流接口与 AI 引擎计算图相连并进行交互。这些串流接口是使用 AXI4-Stream 协议来实现的。

Vitis HLS 提供了 `hls::stream<>` C++ 类库，用于简化通过串流接口对设计进行建模与综合的过程。在 Vitis 内核流程中，Vitis HLS 使用 `hls::stream<>` 类从 C++ 代码自动推断 AXI4-Stream 接口。如需了解有关此库的更多信息，请访问此链接以参阅《Vitis 高层次综合用户指南》(UG1399) 中的相应内容。

除了简单的数据传输外，AXI4-Stream 协议还提供了可选的边带信号，用于传递其他控制信息。AXI4-Stream 边带信号可使用 `hls::axis<>` 数据类型来建模。欲知详情，请访问此链接以参阅《Vitis 高层次综合用户指南》(UG1399) 中的相应内容。

将 HLS 内核连接到 NoC

PL 内核可使用存储器映射接口来与 Versal NoC 相连并进行交互。这些存储器映射接口都是使用 AXI4 存储器映射协议来实现的。

在 Vitis 内核流程中，Vitis HLS 会根据顶层函数的指针实参自动推断 AXI4 存储器映射接口。但对于高性能设计，开发者必须确保能够高效访问这些接口。举例而言，在随机存储器位置执行单次访问的效率尤其低。在可推断突发传输事务的连续存储器位置中，最重要的是内核能够按顺序连续执行访问。欲知详情，请访问此链接和此链接以参阅 Vitis HLS 用户指南 (UG1399) 中的相应内容。

Vitis HLS 中的最优化技巧

请访问此链接以参阅《Vitis 高层次综合用户指南》(UG1399) 中的最优化技巧完整列表，包括：

- 用于将高层次综合写入 RTL 代码的 C/C++ 编程技巧
- 提升代码性能的最优化方法

Vitis HLS 调试和验证注意事项

使用 Vitis HLS 执行调试和验证时，请考虑以下事项：

- 确保 C++ 源代码正确完成验证并能完全正常工作，然后再使用 Vitis HLS 对其进行编译。在大多数情况下，您可使用标准 C++ 验证方法，以及您偏好的 C++ 编译器和 IDE。
- 为 C++ 代码创建测试激励文件后，AMD 建议将测试激励文件导入 Vitis HLS 工程以便利用此流程内置的验证功能。
- 使用 Vitis HLS 进行设计综合前，您可使用综合前 C 语言仿真流程来验证自己的设计能否在 Vitis HLS 环境内正常工作。欲知详情，请访问此链接以参阅 Vitis HLS 用户指南 (UG1399) 中的相应内容。
- 完成设计综合后，可使用综合后 C/RTL 协同仿真流程来验证生成的 RTL 行为是否符合期望。运行 C/RTL 协同仿真流程还会生成剖析信息，此信息可用于对设计性能进行分析。
- 使用 Vitis 硬件仿真来测试内核与软件应用的集成，或者用于测试多个内核之间的交互。

通过 C 语言仿真和 C/RTL 协同仿真来验证 HLS 内核属于块级任务。使用硬件仿真来验证 HLS 内核则属于系统集成任务。如需了解有关 Versal 自适应 SoC 的 HLS 仿真和硬件仿真流程的更多信息，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相关内容。

第 6 章

I/O 管脚分配设计流程

AMD Vivado™ IDE 支持您以交互方式浏览、直观显示、分配并确认设计中的 I/O 端口和时钟逻辑。此环境不仅可确保实现自动建构校正 (correct-by-construction) 式 I/O 分配。它还支持直观显示与内部裸片焊盘相关的外部封装管脚。

您可通过直观方式查看流经器件的数据流并从外部和内部双视角来正确规划 I/O。通过 Vivado IDE 完成 I/O 分配和配置后，即可为实现工具自动创建约束。

AMD 建议按以下顺序对高速接口执行 I/O 管脚分配，以便最大程度利用可用的 XPHY 逻辑资源：

1. 通过 NoC 集成的 DDR 存储器控制器
2. 软核存储器控制器
3. Advanced I/O Wizard
4. I/O 逻辑

同样，AMD 建议在 Vivado IP integrator 中同时规划所有 GT 块的使用，这样即可跨多个软核 IP（例如，Aurora、Ethernet、JESD 等）以最优方式共享 GT 四通道。硬核 IP（如 MRMAC、DCMAC 或 PCIe）则不共享 GT 四通道。

如需了解有关 Vivado Design Suite I/O 管脚分配和时钟规划功能的更多信息，请参阅《Vivado Design Suite 用户指南：I/O 管脚分配和时钟规划》(UG899)。

建议：AMD IP 现已支持 AMD Versal™ 器件，包括存储器和其他遵循具体时钟设置规则的高速 I/O 接口。因此，AMD 建议使用基于网表的 I/O 管脚分配流程，包括 AMD IP 以及用于实践 DRC 的基本逻辑。

适合 I/O 管脚分配的 Vivado Design Suite 工程类型

您可对以下类型的工程执行 I/O 管脚分配：

- I/O 管脚分配工程：I/O 管脚分配工程作为简单的起点，支持您指定选定 I/O 约束，并从已定义的管脚生成顶层 RTL 文件。

  建议：对于 Versal 器件，在 I/O 管脚分配工程中仅支持低性能 I/O 逻辑接口。因此，AMD 建议使用 RTL 工程和支持高性能 XPHY 逻辑的 AMD IP 核。

- RTL 工程：RTL 工程允许综合和实现，支持更全面的设计规则检查 (DRC)。RTL 工程还允许生成 IP 核，这些 IP 核对于存储器接口管脚分配、高性能 XPHY 逻辑和使用 GT 的所有核都至关重要。

  建议：AMD 建议使用 Vivado IP integrator 来生成复杂 IP（如 MRMAC 或 DCMAC），因为此类 IP 需块自动化设置才能将硬核块正确连接到 GT 四通道。

您可以在综合后网表上运行更全面的 DRC。设计实现与 PDI 生成后也同样如此。因此，AMD 建议使用包含时钟组件和部分基本逻辑的骨架设计来实践 DRC。这有助于确保开发板的管脚定义后续不会引发任何问题。

推荐的验收流程为：运行 RTL 工程直至 PDI 生成环节，以实践全部 DRC。但是，并非所有设计周期都有足够时间用于完成此流程。通常必须先定义 I/O 配置，然后再实现可综合的 RTL。虽然 Vivado 工具支持 RTL 前 I/O 管脚分配，但只能执行基本级别的 DRC。或者，您可使用含 I/O 标准和管脚分配的虚拟顶层设计来帮助执行 bank 分配规则相关的 DRC。

RTL 前 I/O 管脚分配

针对 Versal 器件不建议执行 RTL 前 I/O 管脚分配，因为 AMD IP 对于存储器和遵循特定时钟规则的其他高速 I/O 接口存在极高的依赖性。

基于网表的 I/O 管脚分配

AMD 建议在设计完成综合后分配 I/O 和时钟逻辑约束。对于 Versal 器件，AMD 建议在 1 个 RTL 工程中例化所有 IP 和 I/O 以及基本逻辑。随后即可对此工程进行综合。对于使用 GT 块（如 MRMAC 或 DCMAC）的设计，Hard Block Planner 可提供可视图，以帮助向有效站点 (site) 和封装管脚分配 GT 四通道和 GT 参考时钟。对于 AMD IP（如存储器接口和高速 I/O 接口），Advanced IO Wizard 支持自动建构校正 (correct-by-construction) 式管脚分配。对于使用 I/O 逻辑的传统低性能接口，可将管脚拖放到“Package”（封装）窗口上来执行管脚分配。

管脚分配的必需信息

为了使工具能够有效工作，您必须尽可能提供有关 I/O 特性和拓扑结构的详细信息。必须指定电气参数，包括 I/O 标准、驱动、斜率和 I/O 方向。

还必须考虑包括连接、时钟拓扑和时序约束在内的所有其他相关信息。尤其是时钟选择，其可能对管脚选择产生重大影响，反之亦然。

对于具有 I/O 要求的 IP（例如，收发器、PCIe® 核、存储器接口和高速 I/O 接口），必须在完成 I/O 管脚分配前配置 IP。如需了解有关指定 I/O 电气参数的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：I/O 管脚分配和时钟规划》(UG899) 中的相应内容。

重要提示！并非所有 I/O 都能访问 Versal 器件中的 PL 区域，例如，角点 bank 中的部分 I/O。欲知详情，请参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003)、《Versal 自适应 SoC SelectIO 资源架构手册》(AM010) 和《Versal 自适应 SoC 封装和管脚分配架构手册》(AM013)。

管脚分配选择

AMD 建议对于某些信号应审慎选择管脚分配，详情请参阅以下章节。

适用于管脚分配选择的通用准则

通用准则如下所述：

- 将相同的接口数据、地址和控制管脚组合到同一个 bank 上。如果无法将这些组件组合到同一个 bank 上，请将其组合到相邻 bank 中。
- 将以下接口控制信号布局在所控制的数据总线中间：时钟、使能、复位和选通。
- 将扇出极高且覆盖整个设计的控制信号布局在器件中心。

平台管理控制器管脚

要想设计一套高效的系统，必须选择充分满足系统要求的器件启动模式。每种启动模式都使用一组不同的平台管理控制器 (PMC) 管脚。需要考虑的因素包括：

- 请复查 PMC 专用管脚、PMC 多路复用 I/O (MIO) 管脚和 PL 扩展 MIO (EMIO) 管脚的要求。

  注释：每种启动模式都使用一组专用 I/O 管脚或多路复用 I/O 管脚。启动完成后，这些多路复用 I/O 管脚将被释放以便用作通用管脚。
- 请确保所选启动模式不会给共享 MIO bank 的外设施加不必要的电压限制。
- 为不同的 PMC 管脚选择合适的终端。
- 对 PMC 管脚，使用建议的上拉或者下拉电阻值。

建议：在开发板上执行信号完整性分析，以确保 PMC 管脚上的信号无干扰。

有多种启动模式选项可供使用。虽然选项灵活多样，但是每个系统一般都有 1 个最佳的解决方案。在选择最佳启动选项时，请考虑以下方面：

- 设置（包括管脚计数和管脚位置）
- 速度
- 成本
- 复杂性

如需了解有关器件启动模式选项的更多信息，请参阅：

- 请访问此链接以参阅《Versal 自适应 SoC 技术参考手册》(AM011) 中的相应内容
- 请访问此链接以参阅《Versal 自适应 SoC PCB 设计用户指南》(UG863) 中的相应内容
- 《Vivado Design Suite 用户指南：编程和调试》(UG908)

存储器和高级 I/O 接口

使用 AMD 存储器 IP 和 Advanced IO Wizard IP 时，需要执行额外 I/O 管脚分配步骤。自定义 IP 后，即可将顶层 IP 端口分配到 Vivado IDE 中经过细化或综合的设计中的物理封装管脚。与每个存储器 IP 或 Advanced IO Wizard IP 关联的所有端口都组合在一起并连接到同一个 I/O 端口接口，以便于识别和分配。所提供的 Advanced I/O Planner 可帮助您将 I/O 管脚组分配到物理器件管脚上的 XPHY NIBBLESLICE 中。欲知详情，请访问此链接以参阅《Versal 自适应 SoC SelectIO 资源架构手册》(AM010) 中的相应内容、访问此链接以参阅《Advanced I/O Wizard LogiCORE IP 产品指南》(PG320) 中的相应内容，或者请参阅对应存储器 IP 的产品指南。

重要提示！如需了解有关设计和管脚分配准则的信息，请访问此链接以参阅《Versal 自适应 SoC PCB 设计用户指南》(UG863) 中的相应内容。请遵循本指南中的走线长度匹配建议进行操作，验证使用的终端是否准确，并在 Advanced I/O Planner 中完成 I/O 分配后通过运行 DRC 来确认管脚分配。

千兆位收发器 (GT)

千兆位收发器 (GT) 具有特定的管脚分配 (pinout) 要求，您必须考量如下注意事项：

- 共享参考时钟
- 在四通道中共享 PLL

- GT 硬核块（如 PCIe 或 MRMAC）的布局及其与收发器的距离

注释：如需获取有关 CPM5 的 GT 选择和管脚分配指导信息，请访问此链接以参阅《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347) 中的相关信息。

AMD 建议您使用“GT Wizard”（GT 向导）来生成核。或者，您也可以使用 AMD IP 核来获取协议。使用 AMD 收发器 IP 时，需要执行额外的 I/O 管脚分配步骤。自定义 IP 后，即可使用 Vivado IDE 中的“Hard Block Planner”（硬核块分配器）或“Pin Planner”（管脚分配器）将顶层收发器 I/O 和 REFCLK 端口分配到经过细化或综合的设计中的物理封装管脚。如需获取管脚分配建议，请参阅《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347)。

注释：为实现时钟资源平衡，Vivado 布局器会尝试对由 GT 输出时钟进行时钟设置的负载进行约束，这些输出时钟位于为时钟供电的 GT 旁。

角点 bank

当 XPIO bank 分配资源位于处理器系统 (PS) 或高速收发器列等资源的相邻位置时，此 XPIO bank 的功能可能受限。由于这些受限的 XPIO bank 通常位于器件角落处，因此被称为“corner banks”（角点 bank）。角点 bank 位置因器件而异，在《Versal 自适应 SoC 封装和管脚分配架构手册》(AM013) 中以 `DDRMC` 标志来显式调用。以下是处理角点 bank 时的注意事项：

- 虽然角点 bank 中的时钟设置管脚 (GC) 有权访问全部钟设置资源，但非时钟设置管脚则无权访问 DDR 存储器控制器的所有功能。半字节边界上的部分 bank 虽然通常是随 bank 边界一起定义的，但在某些情况下，可能仅限用于 DDR 存储器控制器用途。
- 在对 Versal 自适应 SoC 设计进行管脚分配时，对于 DDR 存储器控制器接口，仅限使用角点 bank。对于 Advanced I/O Wizard 设计使用 Advanced I/O Planner 时，禁止使用角点 bank。
- 如果使用 GC 输入来驱动 XPIO 角点 bank，那么时钟缓冲器 BUFGCTRL 和 BUFGCE_DIV 无法布局在 MMCM 之后，因为这些均为保留的 site 位置。受影响的角点 bank 位于 PS 下（左下角），无权访问可编程逻辑的全部资源。右下角的 GT 下的角点 bank 则并无任何限制。
- 如需在受限的角点 bank 中使用 MMCM，那么 MMCM 输出时钟必须穿过 BUFGCE 才能到达任何其他负载，包括 BUFGCTRL 或 BUFGCE_DIV。

接口带宽确认

创建小型连接设计以确认器件上的每个接口。这些小型设计仅运行特定硬件接口以支持：

- 针对管脚、时钟与时序执行完整 DRC 检查
- 在返还开发板时执行硬件测试设计
- 通过 Vivado 工具进行快速实现，以提供最快速的接口调试途径

有多种选项可用于辅助生成这些接口的测试数据。对于某些接口 IP 核，Vivado 工具可为测试设计提供：

- 针对 SerDes 的 IBERT
- IP 核内的设计示例

提示：如不存在测试设计，请考虑使用 AXI Traffic Generator。

您可能需要创建独立的设计以供在量产环境内进行系统级测试。通常这一独立设计包含经过测试的接口，也可选择在其中包含处理器。您可以使用小型连接设计来构建此设计，以便充分实现设计复用。虽然流程早期无需此设计，但它可支持执行更高效准确的 DRC 检查和早期软件开发，并且您可使用 Vivado IP integrator 快速创建此设计。

适用于 I/O 管脚分配的 SSI 技术注意事项

为 SSI 技术 Versal 器件规划管脚分配时，重要的是确认 XPIO bank 位于底部 SLR (SLR0) 内，而不是像先前列式架构一样遍布整个 SLR。大多数情况下，与 XPIO 外部接口关联的逻辑必须与 I/O 和 XPHY 逻辑位于相同 SLR 内。如果使用硬化的 DDR 存储器控制器，则关联的数据移动可利用专用 NoC 布线越过 NoC 延续至另一个 SLR，而不会造成额外的 PL 设计实现复杂性。决定外部接口的布局时，应考量的事项包括：

- 对于较小的接口，请将所有管脚组合到单个 XPIO bank 内
- 对于较大的接口，请将所有管脚组合到多个相邻的 XPIO bank 内
- 将硬化的 DDR 存储器控制器布局在无权访问 PL 的角点 bank 内
- 跨 XPIO bank 均衡分配 CCIO 或 CMT 组件
- 对于 GT 接口，请将所有 GT 管脚组合到最少量的四通道内
- 对于连接到其他硬核 IP（例如，PCIe、MRMAC 或 DCMAC）的 GT 接口，请将所有 GT 管脚与硬核 IP 保留在同一个 SLR 内，并使这些 GT 管脚与硬核 IP 位于 SLR 的同一侧（左侧或右侧），理想情况下最好位于相同或相邻的时钟区域内
- 对于需要访问 MMCM 或时钟多路复用器资源的 GT 接口，请将 GT 管脚与 XPIO 布局在相同 SLR 内，以降低时钟布线使用率

第 7 章

采用 SSI 器件进行设计

SSI 管脚分配注意事项

为 Versal 自适应 SoC 堆叠硅片互联 (SSI) 技术器件规划管脚分配时，重要的是确认 XPIO bank 位于底部 SLR (SLR0) 内，而不是像先前列式架构一样遍布整个 SLR。与 XPIO 外部接口关联的逻辑大多数位于 SLR0 内，此逻辑不得跨 SLR。决定外部接口的布局时，请考量以下注意事项：

- 对于较小的接口，请将所有管脚组合到单个 XPIO bank 内。
- 对于较大的接口，请将所有管脚组合到多个相邻的 XPIO bank 内。
- 将硬化的 DDR 存储器控制器布局在无权访问 PL 的角点 bank 内。
- 在各 XPIO bank 之间平衡布局支持时钟功能的 I/O (CCIO) 或时钟管理模块 (CMT) 组件。

超级逻辑区域 (SLR)

“超级逻辑区域 (SLR)”是 SSI 技术器件中包含的单个器件裸片 slice。每个 SLR 都包含少量器件资源（例如，CLB、块 RAM、DSP 拼块 (tile) 和 GT），其结构与非 SSI 器件结构相似。

多个 SLR 组件采用纵向堆叠并通过中介层连接以构成 SSI 技术器件。底部 SLR 为 SLR0，后续 SLR 组件按纵向升序递增方式来命名。

例如，XCVP1802 器件包含 4 个 SLR 组件。底部 SLR 为 SLR0，SLR0 正上方的 SLR 为 SLR1，SLR1 正上方的 SLR 为 SLR2，顶端 SLR 为 SLR3。

注释：AMD 工具会在图形用户界面 (GUI) 和报告中标明 SLR 组件。

SLR 命名法

了解目标器件的 SLR 命名法对以下方面非常重要：

- 管脚选择
- 布局规划
- 分析时序及其他报告
- 确认逻辑所在的位置以及逻辑的源端和目的端

您可使用 Vivado Tcl 命令 `get_slrs` 来获取有关特定器件的 SLR 的具体信息。例如，使用如下命令：

- `llength [get_slrs]` 可获取器件中的 SLR 数量
- `get_slrs -of_objects [get_cells my_cell]` 可获取 `my_cell` 所在的 SLR

硅中介层

硅中介层是 SSI 技术器件中的无源层，通过在 SLR 组件间布线来实现：

- 配置
- 全局时钟设置
- 常规互连
- 片上网络 (NoC)

超长线路 (SLL) 布线

超长线路 (SLL) 布线将器件内各 SLR 间的信号联通。

提示：为确定 SLR 间的可用 SLL 数量，请使用 SLR 属性。例如：

```
get_property NUM_TOP_SLLS [get_slrs SLR0]
get_property NUM_BOT_SLLS [get_slrs SLR1]
```

传输限制

提示：要实现跨 SLR 高速传输，请务必寄存跨 SLR 边界的信号。

SLL 信号是 SLR 组件之间的唯一数据连接。

下列信号不在 SLR 组件间传输：

- 进位链
- DSP 级联
- 块 RAM 和 UltraRAM 级联

工具通常会考量上述传输限制。为确保设计布线正确，并且符合您的设计目标，在构建超长级联并在 SLR 边界附近手动布局此类逻辑时，您同样必须将此限制纳入考量范围。

SLR 使用率注意事项

Vivado 实现工具使用特殊算法将逻辑分区到多个 SLR 中。对于较困难的设计，可遵循如下准则改进对应 SSI 技术器件的时序收敛。

为改进时序收敛和编译时间，可使用 Pblock 来将逻辑分配到每个 SLR 并确认所有互连结构资源类型间的每个 SLR 都不存在使用率过高的问题。例如，对于块 RAM 使用率达 70% 的设计，如果在 SLR 间未平衡块 RAM 资源并且其中 1 个 SLR 的块 RAM 使用率超过 85%，那么可能导致时序收敛问题。

提示：您可通过指定完整 SLR（例如，`resize_pblock pblock_SLR0 -add SLR0`）来定义 SLR Pblock。

SLR 使用率示例

以下 VP1702 使用率报告示例显示总块 RAM 使用率为 56%，其中 SLR0 为 57%，SLR1 为 61%，SLR2 为 54%。块 RAM 使用率平均分布在各 SLR 间，每个 SLR 中均采用合理使用率，以便为 Vivado 实现命令提供更多的灵活性，从而满足时序要求。

图 90：**使用率报告中的块 RAM 部分**

3. BLOCKRAM

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	2096	0	0	3741	56.03
RAMB36E5	2034	0	0	3741	54.37
RAMB18E5*	124	0	0	7482	1.66
Block RAM Imux registers	0	0			
Pipelining	0				
URAM	0	0	0	1925	0.00
URAM Imux registers	0	0			
Pipelining	0				

图 91：**使用率报告中的 SLR 部分**

15. SLR CLB Logic and Dedicated Block Utilization

Site Type	SLR0	SLR1	SLR2	SLR0 %	SLR1 %	SLR2 %
SLICE	29765	34925	26527	26.45	34.06	25.87
SLICEL	15168	17764	13530	26.96	34.65	26.39
SLICEM	14597	17161	12997	25.94	33.48	25.35
CLB LUTs	136000	163188	122382	15.11	19.90	14.92
LUT as Logic	128631	154349	115751	14.29	18.82	14.11
single output	73062	87658	65722	8.12	10.69	8.01
dual output	55569	66691	50029	6.17	8.13	6.10
using O5 output only	10	27	18	<0.01	<0.01	<0.01
using O6 output only	73052	87631	65704	8.11	10.68	8.01
using O5 and O6	55569	66691	50029	6.17	8.13	6.10
LUT as Memory	7369	8839	6631	1.64	2.16	1.62
LUT as Distributed RAM	4160	4992	3744	0.92	1.22	0.91
single output	800	960	720	0.18	0.23	0.18
dual output	3360	4032	3024	0.75	0.98	0.74
using O5 output only	0	0	0	0.00	0.00	0.00
using O6 output only	800	960	720	0.18	0.23	0.18
using O5 and O6	3360	4032	3024	0.75	0.98	0.74
LUT as Shift Register	3209	3847	2887	0.71	0.94	0.70
single output	1558	1862	1400	0.35	0.45	0.34
dual output	1651	1985	1487	0.37	0.48	0.36
using O5 output only	0	0	0	0.00	0.00	0.00
using O6 output only	1558	1862	1400	0.35	0.45	0.34
using O5 and O6	1651	1985	1487	0.37	0.48	0.36
CLB Registers	97500	117000	87750	5.42	7.13	5.35
LOOKAHEAD8	3580	4296	3222	3.18	4.19	3.14
Block RAM Tile	770	736	652	57.42	61.33	54.33
RAMB36	750	712	634	55.93	59.33	52.83
RAMB36E5_INT only	750	712	634	55.93	59.33	52.83
RAMB18	40	48	36	1.49	2.00	1.50
RAMB18E5_INT only	40	48	36	1.49	2.00	1.50
URAM	0	0	0	0.00	0.00	0.00
XRAM	0	0	0	0.00	0.00	0.00
DSPs	0	0	0	0.00	0.00	0.00
Unique Control Sets	2841	3409	2557	1.26	1.66	1.25

AMD 建议将块 RAM 和 DSP 组分配到各 SLR Pblock，以最大限度减少共享信号跨 SLR 的现象。例如，如果地址总线扇出到遍布于多个 SLR 上的一组块 RAM，则可能导致更难以实现时序收敛，因为 SLR 交汇会导致时序关键信号出现额外延迟。

器件资源位置或用户 I/O 选择会将 IP 锚定到 SLR，例如 GT、ILKN、PCIe、MRMAC 以及 DCMAC 专用块或存储器接口控制器。AMD 建议：

- 应特别注意专用块位置与管脚选择，从而避免数据流多次出现跨 SLR 边界现象。
- 使相同 SLR 内的模块与 IP 保持紧密互连。如果无法实现，可以添加流水线寄存器来为布局器提供更大的灵活性，以便在逻辑组间存在 SLR 交汇情况下找到有效的解决方案。
- 确保关键逻辑处于同一 SLR 内。只要确保主模块在其接口处正确完成流水打拍，布局器找到含触发器到触发器 SLR 交汇的 SLR 分区的可能性就更高。

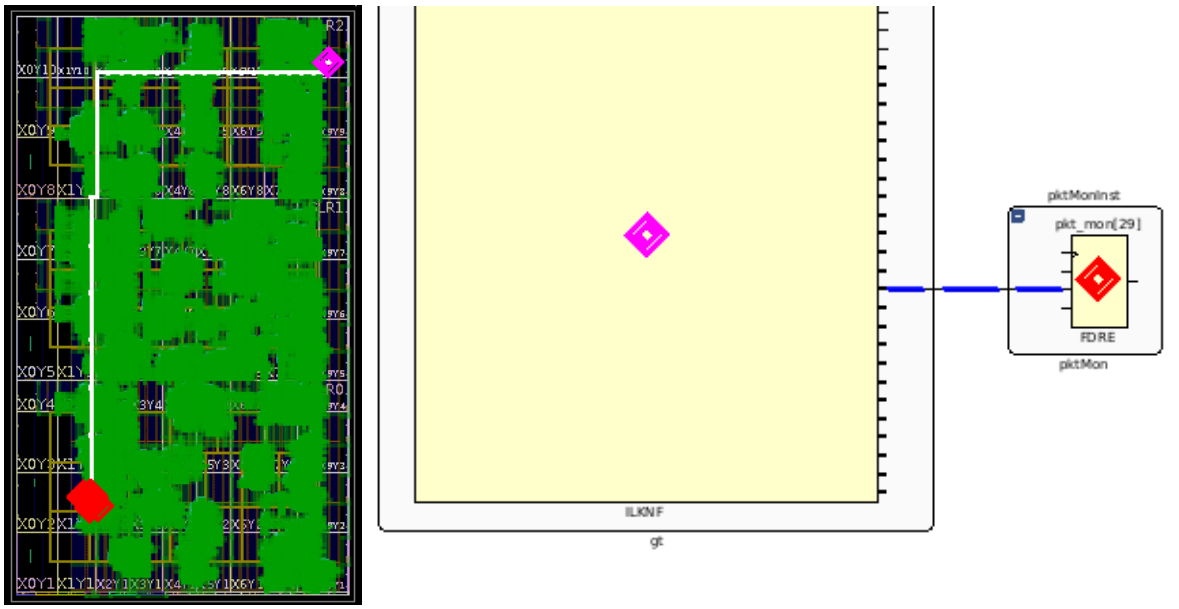
大宽度总线的 SLR 交汇

如果数据流要求中明确要求宽总线必须跨 SLR，请使用流水打拍策略来改进时序收敛并缓解长距离资源的布线拥塞。对于以高于 250 MHz 的频率运行的大宽度总线，AMD 建议使用至少 3 个流水线阶段来跨 1 个 SLR，这 3 个阶段分别位于 SLR 的顶部、底部和中间。对于超高时钟频率的总线，或者在遍历水平距离和垂直距离时，可能需要额外增加流水线阶段。

重要提示！使用 Versal 自适应 SoC SSI 技术器件时，对宽总线 SLR 交汇使用 NoC 的好处在于降低资源使用率和改善时序收敛等。对于宽总线 SLR 交汇请始终考量使用 NoC，使用 AXI 时尤其如此。

下图显示了 VP1702 器件的最差情况交汇。本示例从 SLR2 右上角的 Interlaken 专用块开始，直至分配至 SLR0 左下角的包监控块为止。数据总线与包监控之间无往来流水线寄存器的情况下，设计距离 300 MHz 时序要求相去甚远。

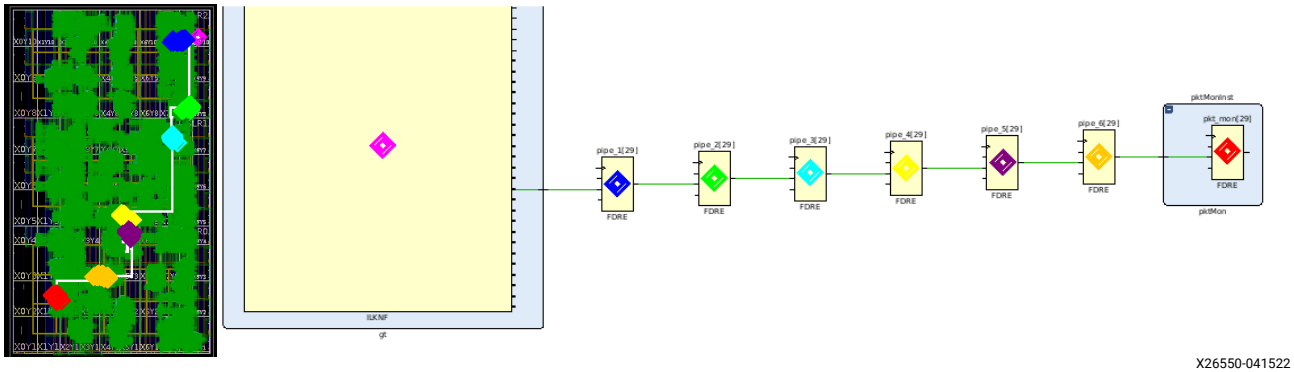
图 92：**跨 SLR（无流水线触发器）的数据路径**



X26549-041522

但是，添加 6 个流水线阶段有助于从 SLR2 遍历到 SLR0，从而帮助设计满足时序要求。这样还可减少垂直和水平长距离布线资源的使用，如下图所示。

图 93：**已添加跨 SLR（含流水线触发器）的数据路径**



提示：使用 AXI Register Slice IP 或定制自动流水打拍 IP 在跨 SLR 的大宽度总线上实现时序收敛。对于宽总线交汇 SLR，请始终考虑使用 NoC。

NoC 注意事项

SSI 技术在 NoC 管脚分配过程中需要特别注意。请自行熟练掌握 SSI 技术要求和建议。

采用 SSI 技术器件进行设计时，为最优化性能，使用布局规划方法来针对特定 SLR 进行设计分区，使每个 SLR 的使用率都保持在指导原则范围内。尝试减少跨 SLR 边界的关键信号数量。这也包括正确规划与布局规划相关的 I/O 接口、时钟和逻辑。

NoC 编译器在 Vivado IP integrator 中运行时，不考虑位置约束。在存储器中开启综合后处理之后，您可对 NoC 接口位置进行布局规划，得到最优 SLR，并在 Vivado IDE 中刷新 NoC 解决方案。由于 NMU 与 NSU 之间的距离会增大，时延可能会变得困难，必须在设计周期中尽早评估和解决。

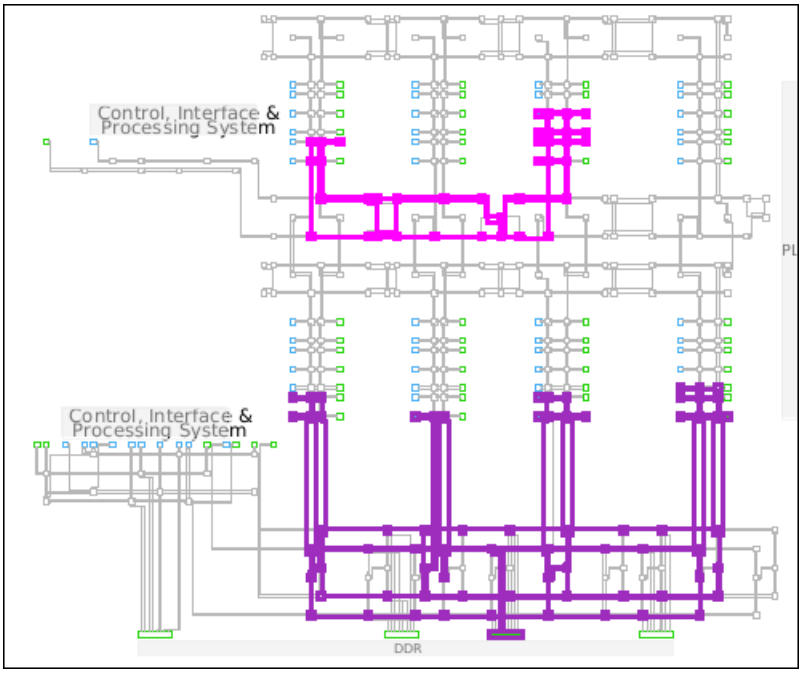
设计 SSI 技术的器件时，要最优化功耗，请减少 SLR 交汇并将布线约束在单一 VNoC 列内。

在 Versal 自适应 SoC SSI 技术的器件上为 Pblock 分配逻辑时，可能需要执行 NoC 设计规划，才能在整个设计进程中实现最优且一致的结果。执行 NoC 设计规划时，请考虑下列事项：

- 在实现工具中使用 SLR 级 Pblock 时，请考虑在 Vivado IP integrator 中约束 NoC NMU/NSU，以匹配实现结果。
- 对于开始或结束于 DDR 存储器控制器或 PS 的 NoC 路径，请考虑将这些路径布局在 SLR0 内以便尽可能缩短时延。
- 对于并非开始或结束于 DDR 存储器控制器或 PS 的 NoC 路径，请考虑将这些路径完全约束在 SLR 内（除 SLR0 外）。
- 建议使用 NoC 替代使用 AXI Register Slice 跨多个 SLR 流水打拍。

在以下示例中，紫色高亮的 PL-NoC 路径与 DDR 存储器控制器对接，并约束到 IP integrator 中的 SLR0。粉色高亮的 NoC 路径属于约束到 IP integrator 中的 SLR1 的 PL-NoC 路径。

图 94：**IP Integrator 中的 SLR 级设计规划**



第 8 章

采用 HBM 器件进行设计

AMD Versal™ HBM 器件按设计支持将快速存储器、自适应计算和安全连接聚合到单一平台内。Versal HBM 系列集成 HBM2e DRAM，可将提供的带宽提升至 DDR5 的 8 倍，功耗较 DDR5 降低 63%。该系列可提供最高 820 GB/s 的存储器带宽和 32 GB 容量，从而为计算密集型应用尽可能减少功耗、面积和时延。可编程 NoC 可从器件上任意位置全局访问集成的 HBM。通过集成存储器控制器和增强型硬化开关功能，即可从任意器件访问任意存储器位置。

根据 HBM 器件类型，Versal HBM 控制器支持访问一个或两个栈，针对 8 个堆叠器件提供最高 128 Gb (16 GB)。8 个独立 HBM 控制器连接到单个栈。HBM 控制器通过 NoC 对接到 PL 中的用户逻辑。NoC 允许从连接到 NoC 的任意主接口对整个 HBM 栈进行全局寻址。NoC IP 核可配置为包含一小部分或者所有集成 HBM 控制器。

在某些应用中，NoC 可能不足以满足 HBM 连接所需，部分数据须流经 NoC，部分数据则须流经互连结构。

使用 HBM 器件的布局注意事项

跨 SLR 流水打拍注意事项

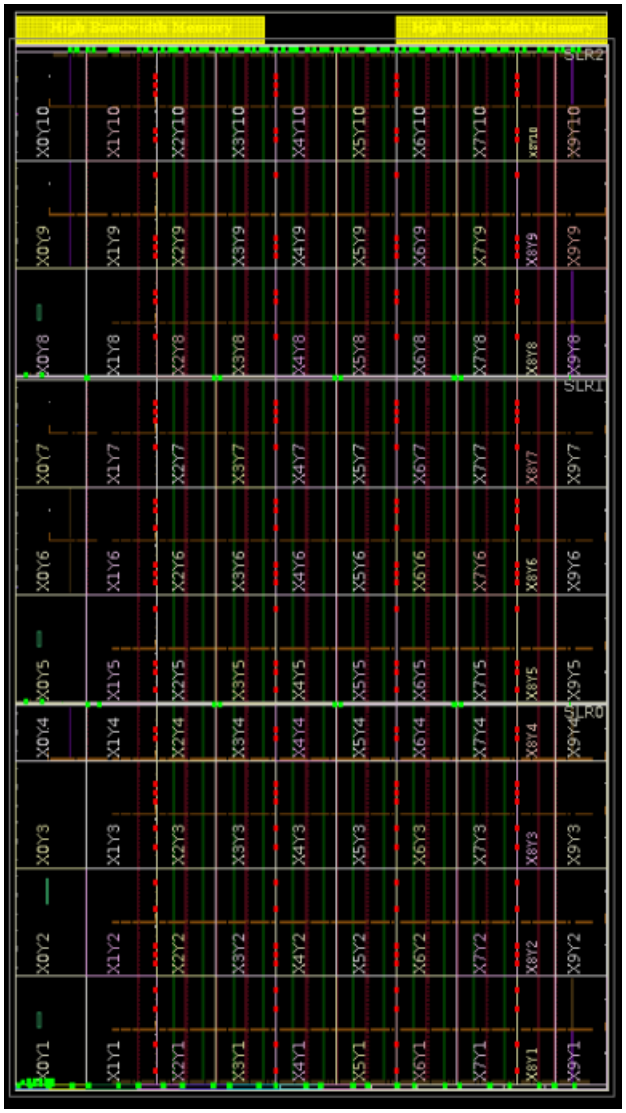
如 NoC 带宽不足，请考虑为距离 HBM 栈较远的模块使用流水打拍。有时设计会包含大量流水线，工具可能难以分析 SLR 分区，而改用 SLR 级别布局规划来最优化 HBM 栈。对于 SLR 级别的布局规划，请根据用于锚定 PS、GT、I/O、特殊 IP 或硬核 IP 等的连接来评估逻辑所在位置。考量 HBM 栈和连接的布局规划对齐（例如，左侧的栈应对齐左侧连接，右侧的栈对齐右侧连接）。

提示：请避免将相同逻辑/模块连接到多个栈，避免从左到右交叉连接。

顶层 SLR 内的资源规划

下图显示了 Versal HBM HBM 堆叠硅片互联器件。不同于 UltraScale+ HBM 器件，HBM 栈位于器件顶层并以黄色高亮、HNoC 以绿色高亮，VNoC 则以红色高亮。直接连接到 HBM 接口的模块应是作为在顶层 SLR（与 HBM 栈相邻）进行布局规划。这样即可避免 SLR 交叉，改善可布线性和时序收敛。并且，对于含 Pblock 统计数据的顶层 SLR 内的模块请执行功能检查。如有大量 HBM 接口布线越过后续 SLR，那么在顶层相邻 SLR 中需要大量长线布线。这可能导致 SLR 中没有足够布线资源可用于局部逻辑，因此请早做打算并尝试减少局部逻辑，以免出现任何布线拥塞。

图 95：**Versal HBM 堆叠硅片互联器件中的资源规划**



建议：AMD 建议将来自 SLR2 和 SLR1 的路径与其相应的 HBM AXI 接口保持垂直对齐，以避免对角穿过器件。

对距离各栈较远的关键连接使用 NoC 连接

对于距离 HBM 栈较远的任何已连接的模块，请考虑使用 NoC 连接。NoC 可能并不支持往来 HBM 器件的所有流量，因此请验证 NoC 解决方案是否正确，包括 QoR 和时延影响。

第 9 章

设计约束

设计约束用于定义各项要求，编译流程必须满足这些要求才能在硬件中正常运行设计。对于复杂的设计，约束通常还用于定义工具指南，以帮助实现收敛。并非所有约束都要在编译流程中的所有步骤中使用。例如，物理约束仅在执行实现步骤（最优化、布局和布线）期间使用。

由于综合与实现算法均由时序驱动，因此必须创建正确的时序约束。对设计进行过约束或欠约束都会导致难以实现时序收敛。您必须使用对应于自己的应用要求的合理约束。如需了解有关约束的更多信息，请参阅以下资源：

- 《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906)
- Vivado Design Suite 视频教程网页上提供了“应用设计约束”视频教程

注释：传统设计流程和基于平台的设计流程以相似方式来使用设计约束。但基于平台的设计需要额外关注从设计静态区域到动态区域的跨边界信号。正确约束这些信号可确保平台灵活性并最大程度减少平台修改。

对设计约束进行组织以便执行编译

通常约束按类别和/或按设计模块组织到 1 个或多个文件中。无论采用何种组织方式，您都必须了解其整体依赖关系，并在载入存储器后复查其最终时序。例如，由于时序时钟必须先定义后才可供其他约束使用，因此您必须确保其定义位于约束文件开头和/或位于载入存储器的第一组约束文件中。

建议的约束文件

根据工程大小和复杂性，有多种适用于约束组织的方法可供选择。下面给出了一些建议。

简单设计

对于小型设计团队开发的简单设计：

- 1 个文件存储所有约束
- 1 个文件存储物理约束 + 1 个文件存储时序约束
- 1 个文件存储物理约束 + 1 个文件存储时序（综合）+ 1 个文件存储时序（实现）

复杂设计

对于复杂设计（含多个 IP 核或多个设计团队）：

- 1 个文件存储顶层时序约束 + 1 个文件存储顶层物理约束 + 1 个文件对应 1 个 IP 或主块

确认读取顺序

完成工程约束文件的组织后，必须根据文件内容确认文件读取顺序。在“工程模式”下，可在 AMD Vivado™ IDE 中或者使用 `reorder_files` Tcl 命令来修改约束文件的顺序。在“非工程模式”下，顺序直接由编译流程 Tcl 脚本中的 `read_xdc` 命令（针对 XDC 文件）和 `source` 命令（针对由 Tcl 脚本生成的约束）来定义。

建议的约束顺序

约束语言 (XDC) 基于 Tcl 语法和解读规则。与 Tcl 一样，XDC 属于顺序语言：

- 必须先定义变量，然后才能加以使用。同样，必须先定义时序时钟，然后才能将其用于其他约束中。
- 对于覆盖相同路径并具有相同优先级的等效约束，适用最后一项约束。
- 当多个时序例外覆盖同一条路径时，适用具有更高优先级的约束。

当考虑以上优先规则时，时序约束总体上应遵循以下顺序：

```
## Timing Assertions Section
# Primary clocks
# Virtual clocks
# Generated clocks
# Delay for external MMCM/PLL feedback loop
# Clock Uncertainty and Jitter
# Input and output delay constraints
# Clock Groups and Clock False Paths
## Timing Exceptions Section
# False Paths
# Max Delay / Min Delay
# Multicycle Paths
# Case Analysis
# Disable Timing
```

当使用多个 XDC 文件时，必须特别留意时钟定义，并确认从属关系排序是否正确。

物理约束可能位于任意约束文件中的任意位置。

创建综合约束

综合将提取设计的 RTL 描述，并使用时序驱动的算法将其变换为经最优化的技术所映射的网表。结果质量受 RTL 代码质量和提供的约束的影响。在编译流程的这个阶段，信号线延迟建模采用近似法，无法反映布局约束或复杂影响（例如拥塞）。建模的主要目的是通过真实且简单的约束获取满足时序约束要求或者接近满足要求的网表。

综合引擎接受所有 XDC 命令，但其中只有部分命令真正有效：

- 与建立/恢复分析有关的时序约束会影响 QoR：
  - `create_clock` / `create_generated_clock`
  - `set_input_delay` / `set_output_delay`
  - `set_clock_groups` / `set_false_path` / `set_max_delay` / `set_multicycle_path`
- 与保持和移除分析有关的时序约束在综合步骤中被忽略：
  - `set_min_delay` / `set_false_path -hold` / `set_multicycle_path -hold`

- RTL 属性会强制采纳映射和最优化算法所制定出的决策。以下提供一些示例：
  - DONT_TOUCH / KEEP / KEEP_HIERARCHY / MARK_DEBUG
  - MAX_FANOUT
  - RAM_STYLE / ROM_STYLE / USE_DSP / SHREG_EXTRACT
  - FULL_CASE / PARALLEL_CASE（仅限 Verilog RTL）

  注释：在 XDC 文件中还可将同样的属性设置为特性。在不改变 RTL 的前提下，仅在某些情况下才能使用基于 XDC 的约束影响综合结果。
- 忽略物理约束（LOC、BEL、Pblock）

综合约束使用的名称必须来自细化的网表（最好是端口和时序单元）。某些 RTL 信号会在细化过程中消失，并且无法为其赋予 XDC 约束。此外，由于细化后执行的各种最优化，信号线或逻辑单元将合并到各种技术原语（例如，LUT 或 DSP 块）中。要了解详细设计对象的名称，单击 Flow Navigator 中的“Open Elaborated Design”，然后浏览您感兴趣的层级。

部分寄存器被吸收到 RAM 块中，部分层级可能消失，以便允许实现跨边界最优化。

所有经细化的网表对象或层级均可通过使用 DONT_TOUCH、KEEP、KEEP_HIERARCHY 或 MARK_DEBUG 约束来保留，但存在时序或面积 QoR 劣化的风险。

最后，某些约束可能存在冲突而不被综合所认可。例如，如果在跨多个层级的网表上设置 MAX_FANOUT 属性，并且使用 DONT_TOUCH 保留部分层级，那么将限制或完全阻止扇出最优化。

重要提示！与实现阶段不同，综合可能会将用于定义时序约束的 RTL 网表对象优化掉以便实现更好的面积 QoR。一般这不会导致问题，前提是对约束进行更新和确认以满足实现要求。但如果需要，仍可使用 KEEP 约束来保留任何对象以便在综合和实现期间应用约束。

完成综合后，AMD 建议您复查时序和使用报告，以确认网表质量满足应用要求并且可用于实现。

创建实现约束

实现约束必须准确反映最终应用的要求。物理约束（例如 I/O 位置和 I/O 标准）取决于开发板设计（包括开发板走线延迟）以及源自总体系统要求的设计内部要求。在进入实现步骤之前，AMD 强烈建议您对所有约束的正确性和准确性进行确认。错误约束可能会降低实现的 QoR 以及时序验收质量的可信度。

多数情况下，在综合与实现阶段可以使用相同的约束。但是，由于设计对象在综合阶段可能消失或发生名称变化，因此必须确认所有综合约束都可正确应用于实现网表。如果不是这样，那么您必须创建 1 个附加 XDC 文件，其中包含仅对实现有效的约束。

创建块级约束

开发多团队工程时，为方便起见，可为顶层设计的每个主要块创建独立的约束文件。通常每个主要块都会先独立开发并确认，最后再整合到 1 个或多个顶层设计中。

块级约束必须独立于顶层约束单独开发，并且必须尽可能采用通用设计以便应用于各种环境中。此外，这些约束不得影响块边界外的任何逻辑。

当实现子块时，最好在时序分析中包含全时钟网络，以确保偏差和时钟域交汇分析的准确性。这可能需要 1 个包含时钟组件的 HDL 封装文件和另一个约束文件以便复制顶层时钟约束。它仅用于子模块的时序确认。

如需了解有关约束范围以及将块级约束加载到顶层设计中的规则、准则和机制的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：使用约束》(UG903) 中的相应内容。

为 Vitis 环境指定约束

在 AMD Vitis™ 环境中，您可将硬件内核指定为 C/C++ 内核或 RTL 内核：

- 使用 C/C++ 内核时，必须使用 Vitis HLS 为综合或实现指定附加的用户约束。随后，必须在 IP 封装器内封装 Vitis HLS 输出，此封装 IP 包含用户约束和工具生成的约束。如需了解更多信息，请参阅 Vitis HLS 用户指南 (UG1399)。
- 使用 RTL 内核时，必须在 IP 封装期间指定附加的综合与实现约束。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：创建和封装定制 IP》(UG1118)。

在 Vitis 环境内，综合与实现的所有设计约束都必须随 IP 一并封装。如果封装 IP 还需其他约束，则必须重新封装 IP 以包含缺失的约束。

但封装 IP 后，可指定其他仅在实现期间使用的 XDC 约束。虽然 Vitis 环境会将底层 Vivado 工具流程抽象化用于实现可编程逻辑区域，但 Vitis 环境还会提供高级选项用于控制 Vivado 工具流程。借助这些高级控制选项，您即可指定要在每个实现阶段之前或之后执行的特定 Tcl 脚本，包括：`init_design`、`opt_design`、`place_design`、`phys_opt_design`、 `route_design` 或 `write_device_image`。如需了解有关 Tcl 脚本编制的更多信息，请参阅《Vivado Design Suite 用户指南：使用 Tcl 脚本》(UG894)。您可使用这些实现前 (Pre) 和实现后 (Post) Tcl 脚本来执行某些 Vivado 工具命令，例如，通过 `read_xdc` 或 `source` Tcl 命令来应用附加的 XDC 约束。

您可通过 Vitis 环境配置文件或者可直接在 v++ 编译器命令行上指定实现前和实现后 Tcl 脚本。

要在 Vitis 环境配置文件内指定实现前和实现后 Tcl 脚本，请在 `[vivado]` 节内使用 `prop=run.impl_1.STEP.<PHASE>.TCL.<PRE|POST>` 参数。

其中：

- `<PHASE>` 用于指定下列实现阶段：`INIT_DESIGN`、`OPT_DESIGN`、`PLACE_DESIGN`、`PHYS_OPT_DESIGN`、`ROUTE_DESIGN` 或 `WRITE_DEVICE_IMAGE`。
- `PRE` 用于在指定的实现阶段之前执行脚本。
- `POST` 用于在指定的实现阶段之后执行脚本。

例如：

```
[vivado]
prop=run.impl_1.STEPS.OPT_DESIGN.TCL.PRE=<pathToTclScript>
prop=run.impl_1.STEPS.OPT_DESIGN.TCL.POST=<pathToTclScript>
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=<pathToTclScript>
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.POST=<pathToTclScript>
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.TCL.PRE=<pathToTclScript>
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.TCL.POST=<pathToTclScript>
prop=run.impl_1.STEPS.ROUTE_DESIGN.TCL.PRE=<pathToTclScript>
prop=run.impl_1.STEPS.ROUTE_DESIGN.TCL.POST=<pathToTclScript>
```

要将 Pre 和 Post Tcl 脚本指定为 v++ 参数，请使用 `--vivado.prop run.impl_1.STEP.<PHASE>.TCL.<PRE|POST>=<pathToTclScript>` 命令行选项。例如，要指定在 `opt_design` 之前执行的 Tcl 脚本，请使用如下命令：

```
--vivado.prop run.impl_1.STEP.OPT_DESIGN.TCL.PRE=<pathToTclScript>
```

其中：

- `--vivado` 是 v++ 命令行选项，用于为 Vivado 工具指定指令。

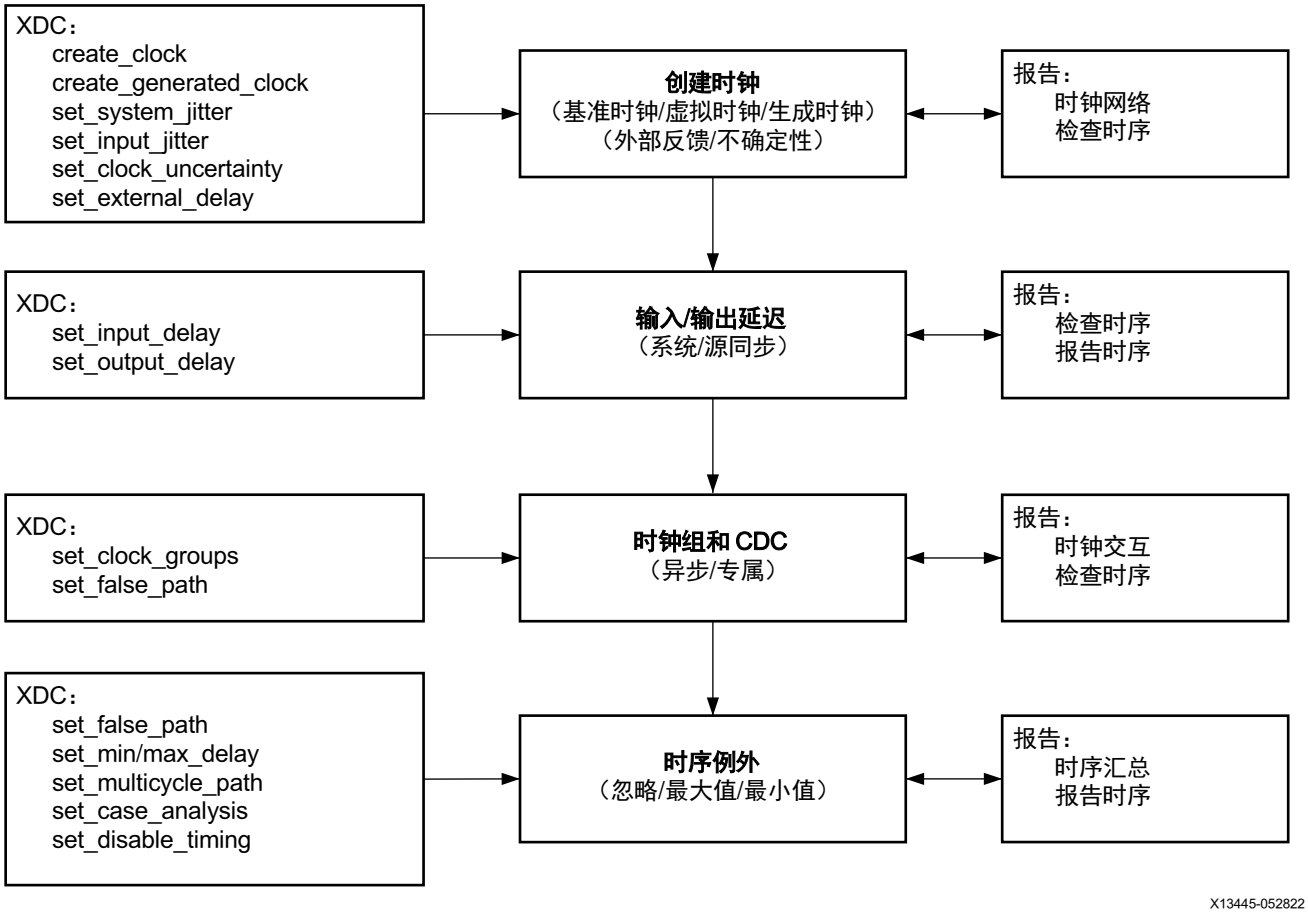
- `prop` 表示属性设置。
- `run.` 表示运行属性。
- `impl_1.` 表示运行轮次名称。
- `STEP.OPT_DESIGN.TCL.PRE` 表示当前指定的运行属性。
- `<pathToTclScript>` 表示属性值。

定义时序约束

定义时序约束的四个步骤

合格的约束的定义过程分为四个主要步骤，如下图所示。这些步骤遵循时序约束先后顺序和从属关系规则，并采用符合逻辑的方式来向时序引擎提供信息以执行分析。

图 96：**时序约束制定步骤**



- 前 2 个步骤与时序断言有效有关，期间将从时钟波形和 I/O 延迟约束中衍生出默认时序路径要求。

- 在第 3 个步骤中，将对至少共享 1 条逻辑路径的异步或专属时钟域之间的关系进行审核。根据关系的性质，可输入时钟组或伪路径约束以忽略这些路径上的时序分析。
- 最后一个步骤对应于时序例外，设计人员可在此判定如何更改默认时序路径要求，包括利用特定约束来忽略、放宽或收紧时序要求。

约束创建与约束识别和约束确认任务息息相关，这些任务必须通过时序引擎生成的各种报告才能实现。时序引擎只能配合经过完全映射的网表使用，例如综合之后的网表。尽管可以用细化的网表输入约束，但还是建议使用综合后网表创建第一组约束，以便约束的分析和报告可交互执行。

为新设计创建时序约束或者完成现有约束时，AMD 建议使用“Timing Constraints Wizard”（时序约束向导）来快速识别上图中的前 3 个步骤中缺失的约束。“Timing Constraints Wizard”遵循本节中所述方法论来确保设计约束的安全性和可靠性，从而实现正确的时序收敛。如需了解有关“Timing Constraints Wizard”的更多信息，请参阅《Vivado Design Suite 用户指南：使用约束》(UG903)。

下列章节将详细描述以上所述的四个步骤：

- 定义时钟约束
- 约束输入和输出端口
- 定义时钟组和 CDC 约束
- 指定时序例外

在约束创建流程中执行相应的步骤时，请参阅各对应章节以了解详细方法论和用例。

定义时钟约束

时钟必须首先完成定义，方可供其他约束使用。时序约束创建流程的第一步是明确必须定义哪些时钟，以及这些时钟必须定义为“primary clock”（基准时钟）还是“generated clock”（生成时钟）。

重要提示！使用特定名称定义时钟（`-name` 选项）时，必须验证该时钟名称未被任何其他时钟约束或现有自动生成时钟占用。如果已在多个时钟约束中使用某个时钟名称，Vivado Design Suite 时序引擎会发出消息，以提醒您第 1 个时钟定义被覆盖。如果同一时钟名称使用了两次，那么第 1 个时钟定义将会丢失，并且 2 个时钟定义之间输入的引用此名称的所有约束也都将丢失。AMD 建议您避免覆盖时钟定义，除非不影响任何其他约束，并且所有时序路径都保持受约束。

识别时钟源

在设计中可通过“Clock Networks”（时钟网络）报告和“Check Timing”（检查时序）报告来识别未约束的时钟源。

时钟网络报告

约束和未约束的时钟源点分别列在 2 个不同类别中。对于每个未约束的时钟源点，必须确定应定义基准时钟还是生成时钟。

```
% report_clock_networks
Unconstrained Clocks
Clock sysClk (endpoints: 15633 clock, 0 nonclock)
Port sysClk
Clock TXOUTCLK (endpoints: 148 clock, 0 nonclock)
GT_QUAD/TXOUTCLK
(mgtEngine/ROCKETIO_WRAPPER_TILE_i/gt0_ROCKETIO_WRAPPER_TILE_i/gtxe2_i)
Clock Q (endpoints: 8 clock, 0 nonclock)
FDRE/Q (usbClkDiv2_reg)
```

“检查时序”报告

`no_clock` 检查用于报告不含时钟定义的有源叶时钟管脚组。每个组均与 1 个时钟源点关联，在其中必须定义时钟方可解决问题。

```
% check_timing -override_defaults no_clock
1. checking no_clock
--------------------
 There are 15633 register/latch pins with no clock driven by root clock
pin: sysClk
(HIGH)
There are 148 register/latch pins with no clock driven by root clock pin:
mgtEngine/ROCKETIO_WRAPPER_TILE_i/gt0_ROCKETIO_WRAPPER_TILE_i/gtxe2_i/
TXOUTCLK
(HIGH)
There are 8 register/latch pins with no clock driven by root clock pin:
usbClkDiv2_reg/C (HIGH)
```

借助 `check_timing`，可使相同的时钟源管脚或端口出现在多个组中，具体取决于整个时钟树的拓扑结构。在此情况下，在建议的源管脚或端口上创建时钟即可解决所有关联组缺失时钟定义的问题。

注释：欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

创建基准时钟

基准时钟是指用于为设计定义时序参考的时钟，而时序引擎可利用基准时钟衍生出时序路径要求以及与其他时钟的相位关系。主时钟插入延迟的计算范围是从时钟源点（用于定义时钟的驱动管脚/端口）到时序单元（作为时钟扇出目标）的时钟管脚。

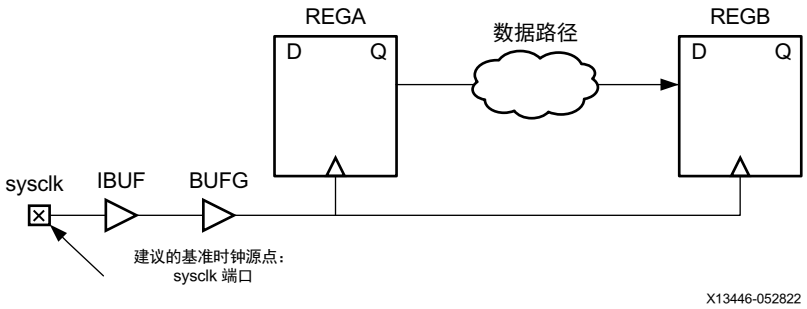
因此，重要的是在对应于设计边界的对象上定义基准时钟，以便准确计算其延迟并间接计算其偏差。

以下部分描述了典型的基准时钟根。

输入端口

您可使用输入端口作为基准时钟根，如下图所示。

图 97：**输入端口的 create_clock**



约束示例：

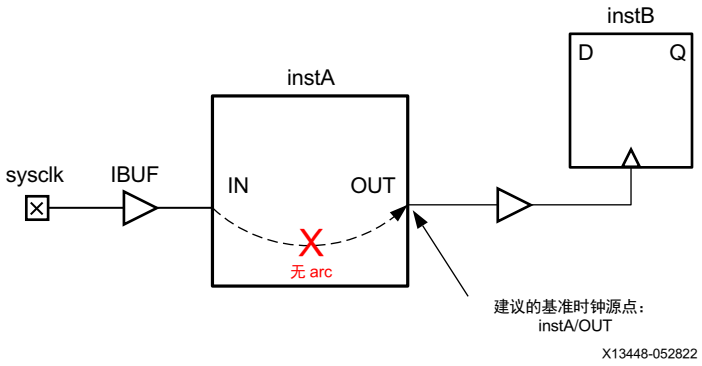
```
create_clock -name SysClk -period 10 -waveform {0 5} [get_ports sysclk]
```

该示例中，波形的占空比定义为 50%。以上显示的 `-waveform` 实参用于展示其使用率，只有在定义占空比非 50% 的时钟时才需要使用。欲知详情，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835) 中的 create_clock Tcl 命令。对于差分时钟输入缓冲器，只需在差分对的 P 侧对基准时钟进行定义即可。

某些硬件原语输出管脚

您可使用某些硬件原语的输出管脚作为基准时钟根（如下图中所示输出管脚），此类输出管脚不具有来自相同原语的输入管脚的时序 arc。

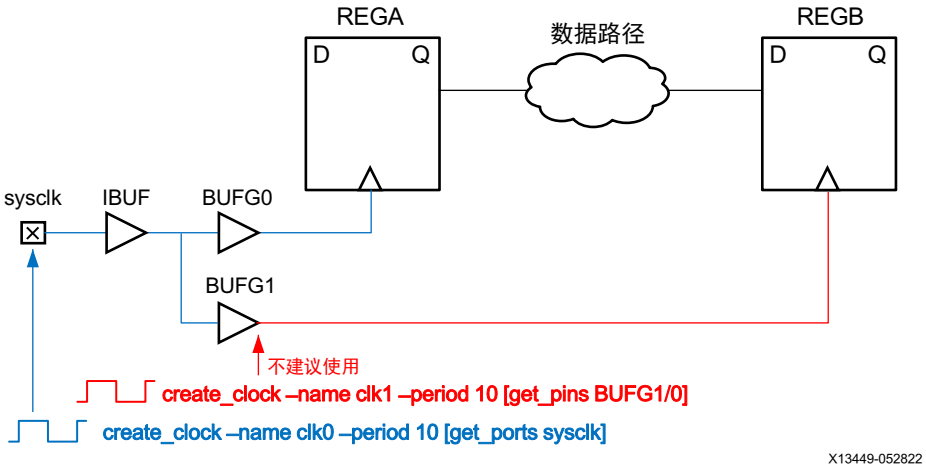
图 98：**时钟路径因缺失时序 arc 而断开**



重要提示！在基准时钟传递扇出中不应定义另外 1 个基准时钟，因为这种情况不但不符合任何硬件现实，还会妨碍完整的时钟插入延迟计算，从而阻碍正确的时序分析。如果发生任何这种情况，必须重新修改并修正约束。

下图显示的示例中，时钟 `clk1` 是在时钟 `clk0` 的传递扇出中定义的。时钟 `clk1` 会从 BUFG1 输出开始覆盖此输出处所定义的 `clk0`。因此，由于 `clk0` 与 `clk1` 之间歪斜突变无效导致 REGA 与 REGB 之间的时序分析并不准确。

图 99：**不建议在另一个时钟的扇出中使用 create_clock**



创建生成时钟

生成时钟 (generated clock) 是从称为主时钟 (master clock) 的另一个现有时钟衍生的。它通常用于描述逻辑块对主时钟执行的波形变换。由于生成时钟定义取决于主时钟特性，因此必须首先定义主时钟。为显式定义生成时钟，必须使用 `create_generated_clock` 命令。

自动衍生时钟

大部分生成时钟都是由 Vivado 时序引擎自动衍生的，该引擎可识别时钟修改块 (CMB) 及其对主时钟执行的变换。

在 AMD Versal™ 系列器件中，CMB 包括：

- MMCM*/XPLL*/DPLL*
- BUFG_GT/BUFGCE_DIV
- MBUFG_PS/MBUFG_GT/MBUFGCE/MBUFGCE_DIV/MBUFGCTRL
- GTYE5_QUAD/GTYP_QUAD/GTME5_QUAD
- IBUFDS_GTE5/IBUFDS_GTME5
- XPHY

对于时钟树上的任何其他组合单元而言，时序时钟可通过这些单元进行传输，且无需在输出端重新定义，除非此类单元已进行波形变换。通常应尽可能依靠自动衍生机制，因为就定义可对应于实际硬件行为的生成时钟来说，这是最安全的方法。

如果您认为 Vivado Design Suite 时序引擎所选的自动衍生时钟名称不合适，那么可以使用 `create_generated_clock` 命令（不指定波形变换）强制输入自己选择的名称。该约束应刚好位于约束文件中定义主时钟的约束之后。例如，如果由 MMCM 实例生成的时钟默认名称为 net0，那么您可添加以下约束来强制输入自己的名称（在给定示例中，此名称为 `fftClk`）：

```
create_generated_clock -name fftClk [get_pins mmcm_i/CLKOUT0]
```

为避免歧义，约束必须连接到时钟的源管脚。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：使用约束》(UG903)。

用户定义的生成时钟

定义所有基准时钟后，可使用“Clock Networks”（时钟网络）或“Check Timing”（检查时序）(`no_clock`) 报告来识别时钟树中不含时序时钟的部分，并定义相应的生成时钟。

有时要理解逻辑椎对主时钟所执行的变换并不容易。在此情况下，必须采用最保守的约束。例如，源管脚是时序单元输出。主时钟至少除以 2，因此，正确的约束应如下示例所示：

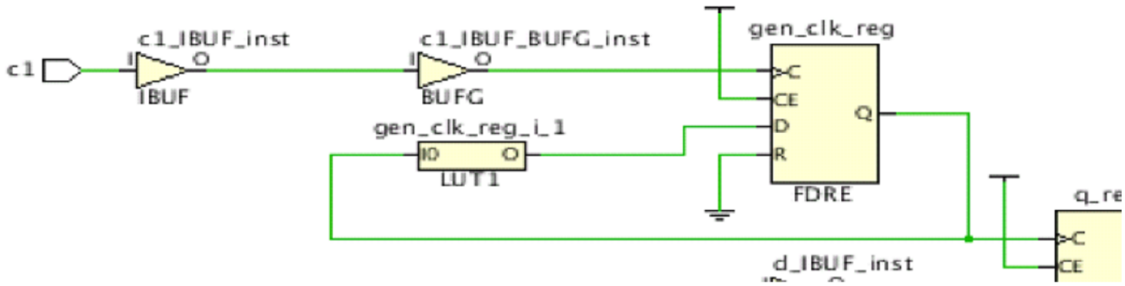
```
create_generated_clock -name clkDiv2 -divide_by 2 \
-source [get_pins fd/C] [get_pins fd/Q]
```

最后，如果设计包含锁存器，那么时序时钟还需要连接到锁存器门控管脚，并且如果缺少约束，则将由“Check Timing”（检查时序）(`no_clock`) 来报告锁存器门控管脚。您可遵循上述示例来定义这些时钟。

主时钟与生成时钟间的路径

与基准时钟不同，生成时钟必须在其主时钟的传递扇出中进行定义，这样时序引擎才能精确计算其插入延迟。不遵守此原则将导致时序分析错误，而且很有可能导致时序裕量计算无效。例如，在下图中，`gen_clk_reg/Q` 用作为下一个触发器 (`q_reg`) 的时钟，并且它还位于基准时钟 `c1` 的扇出椎中。因此，`gen_clk_reg/Q` 应包含 `create_generated_clock` 而不是 `create_clock`。

图 100：**主时钟扇出中的生成时钟**




```
create_generated_clock -name GC1 -source [get_pins gen_clk_reg/C] -
divide_by 2
[get_pins gen_clk_reg/Q]
```

验证时钟定义和覆盖范围

在存储器中定义并应用所有设计时钟后，即可使用 `report_clocks` 命令验证每个时钟的波形以及主时钟和生成时钟之间的关系：

```
Clock Period Waveform Attributes Sources
sysClk 10.00000 {0.00000 5.00000} P {sysClk}
clkfbout 10.00000 {0.00000 5.00000} P,G {clkgen/mmcm_adv_inst/CLKFBOUT}
cpuClk 20.00000 {0.00000 10.00000} P,G {clkgen/mmcm_adv_inst/CLKOUT0}
...
=====================================================
Generated Clocks
=====================================================
Generated Clock : cpuClk
Master Source : clkgen/mmcm_adv_inst/CLKIN1
Master Clock : sysClk
Edges : {1 2 3}
Edge Shifts : {0.000 5.000 10.000}
Generated Sources : {clkgen/mmcm_adv_inst/CLKOUT0}
```

此外，您还可验证所有内部时序路径都被至少 1 个时钟所覆盖。“Check Timing”（检查时序）报告为此提供了两项检查：

- no_clock：报告已定义的时钟无法连接到的任何活动时钟管脚。
- unconstrained_internal_endpoint：如果某些时序单元具有与时钟相关的时序检查但尚未定义时钟，则报告此类时序单元的所有数据输入管脚。

如果两项检查都返回 0，说明时序分析覆盖范围广。

或者，还可运行 XDC 和“Timing Methodology”（时序方法论）检查来验证在建议的网表对象上是否已定义所有时钟，同时避免造成任何约束冲突或不准确的时序分析情境。

请使用以下命令来运行这些检查：

```
report_methodology -checks [get_methodology_checks {TIMING-* XDC*}]
```

相关信息

运行 Report Methodology

调整时钟特性

定义时钟及其波形后，下一步是输入与噪声或不确定性建模相关的所有信息。XDC 语言用于将抖动和相位误差相关的不确定性与偏差和延迟建模相关的不确定性加以区分。

抖动

对于抖动，最好使用 Vivado Design Suite 所使用的默认值。您可按如下方式修改默认计算：

- 如果基准时钟进入器件时随机抖动大于 0，请使用 `set_input_jitter` 命令指定峰值间抖动值（以纳秒 (ns) 为单位）。
- 如果器件电源有噪声，请使用 `set_system_jitter` 调整全局抖动。AMD 不建议增大默认系统抖动值。

对于生成时钟，抖动是由主时钟和时钟修改块的特性衍生而来的。您无需调整这些数值。

其他不确定性问题

如果需要在某个时钟的时序路径上或 2 个时钟之间的时序路径上添加额外裕度，必须使用 `set_clock_uncertainty` 命令。这也是对部分设计进行过约束而不必修改实际时钟沿和总体时钟关系的最佳且最安全的途径。您定义的时钟不确定性是在 Vivado 工具计算所得抖动的基础上附加的，可为建立时间和保持时间分析单独指定此不确定性。

例如，设计时钟 `clk0` 的所有时钟间路径上的裕度需收紧，幅度为 500 ps，以使设计更稳健，承受建立时间和保持时间噪声的能力更强：

```
set_clock_uncertainty -from clk0 -to clk0 0.500
```

注释：收紧设计上的保持时间裕度可能导致专用站点内部路径和级联路径上出现保持时间违例，并且布线器无法通过绕行站点内部信号线来解决此类违例。

如果您在 2 个时钟之间指定额外的不确定性，那么必须应用双向约束（假定数据流为双向）。以下示例演示了如何在 `clk0` 和 `clk1` 之间仅针对建立时间将不确定性增加 250 ps：

```
set_clock_uncertainty -from clk0 -to clk1 0.250 -setup
set_clock_uncertainty -from clk1 -to clk0 0.250 -setup
```

时钟源位置的时钟时延

可使用含 `-source` 选项的 `set_clock_latency` 命令在时钟源处对时钟时延进行建模。该方法在两种情况下有用：

- 用于在器件外部指定与输入和输出延迟约束无关时钟延迟传输。
- 用于在非关联 (OOC) 编译期间，对块所使用的时钟内部传输时延进行建模。在此类编译流程中，不含完整时钟树的描述，因此块外部的最小和最大工作条件之间的差异无法自动进行计算，必须手动建模。

此约束仅限高级用户使用，因为它通常难以提供有效的时延值。

MMCM 外部反馈回路延迟

当连接 MMCM 反馈回路以便补偿开发板延迟（而非内部时钟插入延迟）时，必须使用 `set_external_delay` 命令指定最佳和最差延迟情况下器件外部的延迟。不指定此延迟将使与 MMCM 关联的 I/O 时序分析变得无关紧要，并可能导致时序收敛无法实现。此外，使用外部补偿时，必须相应调整输入和输出延迟约束，而不只是考量正常情况下开发板上的时钟走线延迟。

约束输入和输出端口

除了指定设计的每个端口的位置和 I/O 标准外，还必须指定输入和输出延迟约束以描述进出器件接口的外部路径的时序。这些延迟是根据通常同样在开发板上生成并进入器件的时钟来定义的。在某些情况下，如果与 I/O 路径相关的时钟所含波形不同于开发板时钟的波形，那么必须根据虚拟时钟来定义延迟。

重要提示！只能为使用 I/O 逻辑的接口（例如，IDDR/ODDR/IOB 寄存器或互连结构）约束 I/O 延迟。对于 Versal 自适应 SoC 中的高速 I/O 接口，AMD 提供了 Advanced IO Wizard 和 Advanced I/O Planner。Advanced IO Wizard 用于配置 XPHY，并且可用于估算接口的 I/O 时序。如需了解有关 Advanced IO Wizard 的更多信息，请参阅《Advanced I/O Wizard LogiCORE IP 产品指南》(PG320)。

系统级透视图

I/O 路径的建模方式与 Vivado Design Suite 时序引擎所执行的寄存器间路径建模方式较为相似，区别在于您必须定义约束才能对位于器件外部的路径延迟部分进行建模。分析内部路径时，建立和保持分析都会考虑最小和最大延迟。对于 I/O 路径而言同样如此。基于这个原因，对最小和最大延迟条件进行描述就显得尤为重要。默认情况下 I/O 时序路径可作为单周期路径进行分析，这意味着：

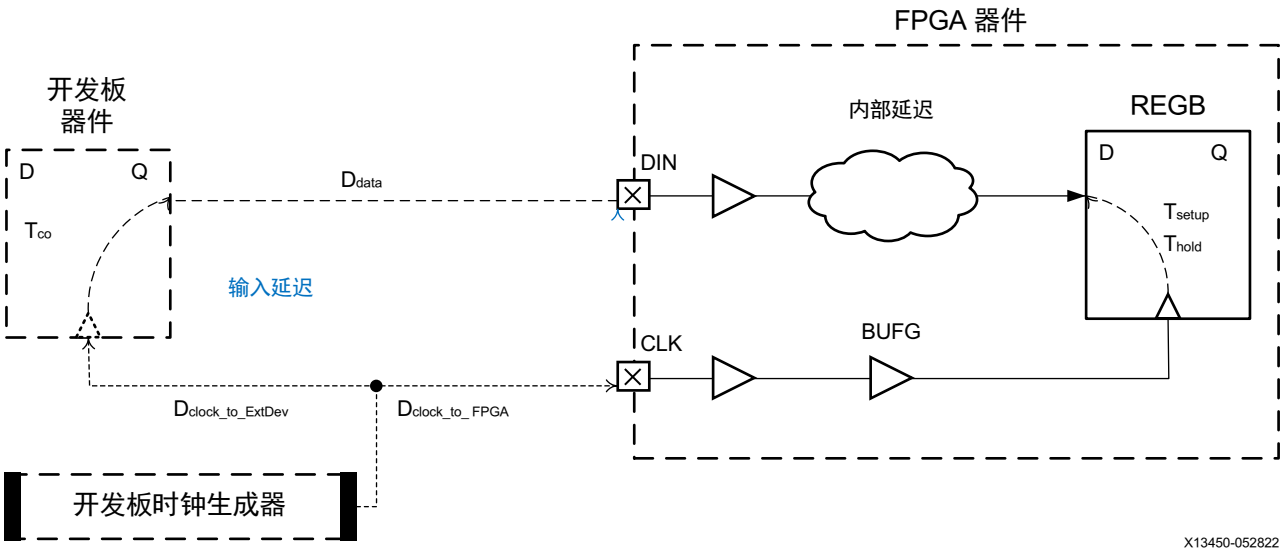
- 为实现最大延迟分析（建立），数据的捕获比单倍数据速率接口的发送沿晚 1 个时钟周期，比双倍数据速率接口的发送沿晚半个时钟周期。
- 为实现最小延迟分析（保持），请在相同时钟沿发送和捕获数据。

如果时钟和 I/O 数据之间的关系必须以不同方式进行时序约束（例如在时钟源同步接口中），那么必须指定不同的 I/O 延迟和附加时序例外。这对应于高级 I/O 时序约束方案。

定义输入延迟

输入延迟定义为与器件接口处的时钟相关的延迟。除非已在参考时钟的源管脚上指定 `set_clock_latency`，否则输入延迟对应于从发送沿到时钟走线、外部器件和数据走线的绝对时间。如果已单独指定时钟时延，即可忽略时钟走线延迟。

图 101：**输入延迟计算**



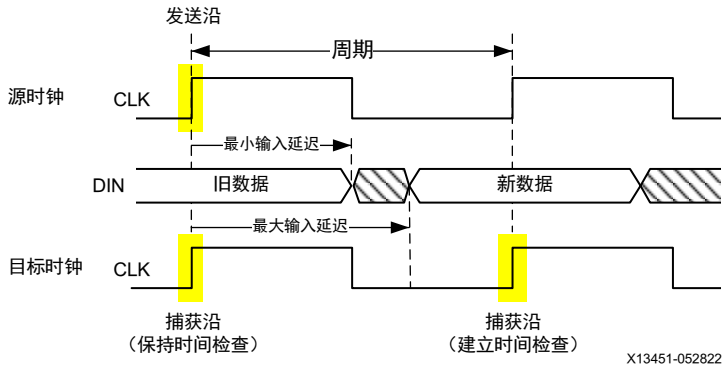
两类分析的输入延迟数值：

```
Input Delay(max) = Tco(max) + Ddata(max) + Dclock_to_ExtDev(max) -
Dclock_to_FPGA(min)
Input Delay(min) = Tco(min) + Ddata(min) + Dclock_to_ExtDev(min) -
Dclock_to_FPGA(max)
```

下图显示了建立时间（最大值）和保持时间（最小值）分析的输入延迟约束的简单示例，其中假定已在 `CLK` 端口上定义 `sysClk` 时钟：

```
set_input_delay -max -clock sysClk 5.4 [get_ports DIN]
set_input_delay -min -clock sysClk 2.1 [get_ports DIN]
```

图 102：**解读最小和最大输入延迟**

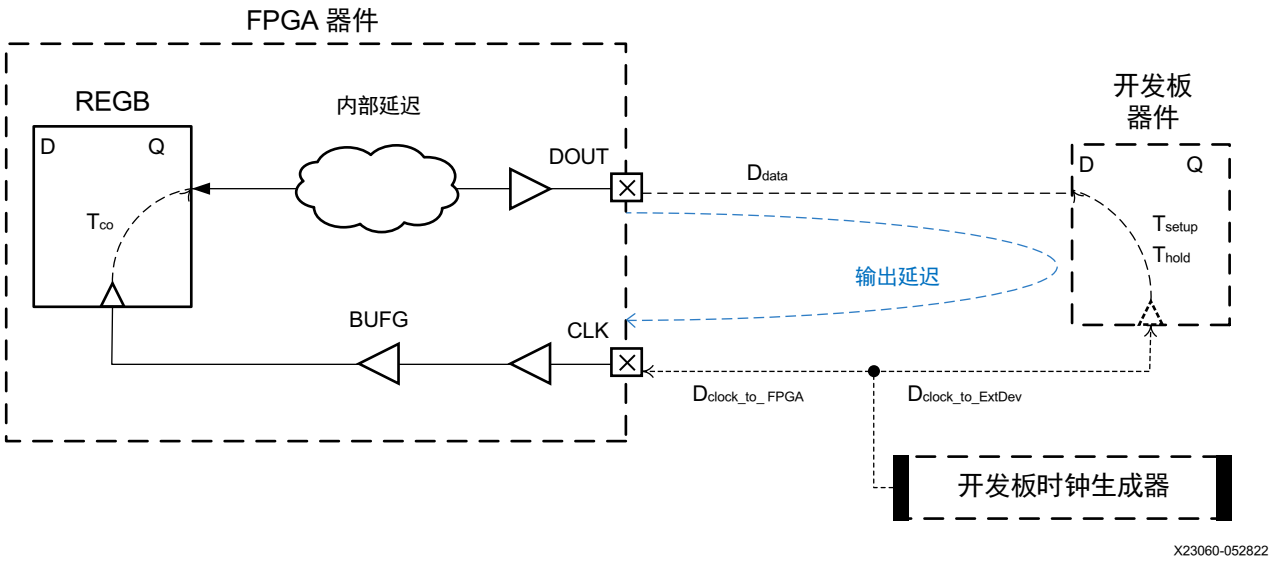


输入延迟为负表示数据到达器件接口的时间早于发送时钟沿。

定义输出延迟

输出延迟与输入延迟类似，区别在于输出延迟表示为了确保在所有情况下均可正常工作，输出路径在器件外部的最短和最长时间。

图 103：**输出延迟计算**



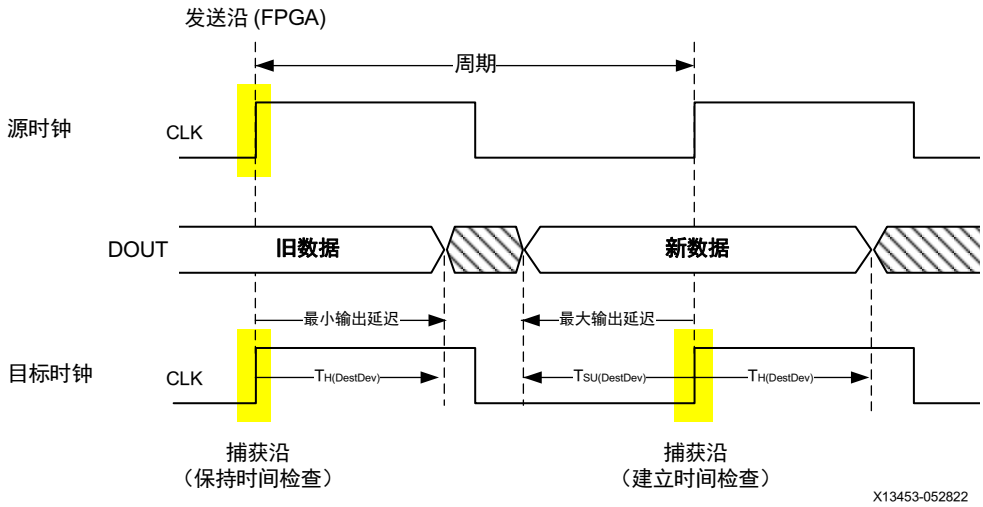
两类分析的输出延迟数值：

```
Output Delay(max) = Tsetup + Ddata(max) + Dclock_to_FPGA(max) -
Dclock_to_ExtDev(min)
Output Delay(min) = Ddata(min) - Thold + Dclock_to_FPGA(min) -
Dclock_to_ExtDev(max)
```

下图显示了建立时间（最大值）和保持时间（最小值）分析的输出延迟约束的简单示例，其中假定在 `CLK` 端口上已定义 `sysClk` 时钟：

```
set_output_delay -max -clock sysClk 2.4 [get_ports DOUT]
set_output_delay -min -clock sysClk -1.1 [get_ports DOUT]
```

图 104：**解读最小和最大输出延迟**



输出延迟对应于开发板上捕获沿之前的延迟。对于其中时钟和数据开发板走线已实现平衡的常规系统同步接口而言，目标器件的建立时间用于定义输出延迟最大值分析。目标器件保持时间用于定义输出延迟最小值分析。指定的输出延迟最小值表示从设计发出信号开始到在目标器件接口上使用信号进行保持时间分析之前，所发生的最小延迟。因此，块内部的延迟可能小得多。输出延迟最小值为正值表示信号在设计内部可能具有负延迟。因此，输出延迟最小值通常为负值。例如，以下代码示例表示设计内部截至 DOUT 为止的延迟必须至少为 +0.5 ns 才能满足保持时间要求。

```
set_output_delay -min -0.5 -clock CLK [get_ports DOUT]
```

选择参考时钟

根据控制输入或输出端口相关的时序单元的时钟树拓扑结构，必须选择最合适的时钟来定义输入或输出延迟约束。如果 I/O 路径寄存器的时钟是生成时钟，那么通常需要根据基准时钟来定义延迟约束，而基准时钟是在生成时钟上游定义的。这条规则存在部分例外，本节将做出解释。

识别与每个端口相关的时钟

在定义 I/O 延迟约束之前，必须首先识别哪些时钟与每个端口相关。您可使用以下章节中所述方法来识别时钟。

浏览开发板原理图

对于连接到开发板上的另一个器件接口的一组 I/O 端口，可使用同时连接到 AMD 器件和外部器件接口的开发板时钟作为输入或输出延迟约束的参考时钟。要控制相关端口组的时序收敛，必须在外部器件数据手册中验证开发板时钟是否已通过内部变换来实现 I/O 端口的时序收敛，从而确保此设计生成的时钟与 AMD 器件内的时钟相同。

浏览设计板级原理图

对于每个端口，可将路径板级原理图展开至时序单元的第一层，然后沿这些单元的时钟管脚走线回到时钟源。对于连接到高扇出信号线的端口，这种方法不可行。

报告进出端口的时序

无论端口是否已约束，均可使用 `report_timing` 命令识别设计中端口的相关时钟。定义完所有时序时钟后，即可报告进出 I/O 端口的最差路径、创建与报告的时钟相关的 I/O 延迟约束，并为进出设计的其他时钟重新运行相同的时序报告。如果发现端口与多个时钟相关，请创建对应的约束并重复此过程。

例如，din 输入端口与设计中的 clk1 时钟和 clk2 时钟相关：

```
report_timing -from [get_ports din] -sort_by group
```

此报告显示 din 端口与 clk1 相关。输入延迟约束为（同时适用于该示例中的最小和最大延迟）：

```
set_input_delay -clock clk1 5 [get_ports din]
```

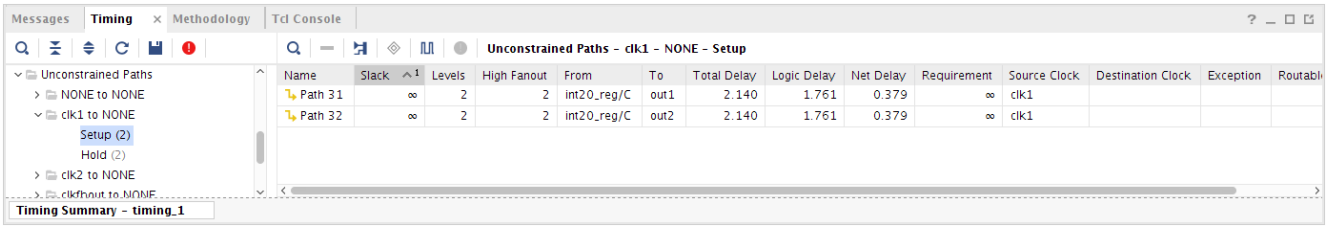
使用先前所用的命令重新运行时序分析，并观察发现 din 与 clk2 同样有关，原因在于 -sort_by group 选项，该选项针对每个端点时钟报告 N 条路径。您可添加对应的延迟约束并重新运行报告以确认 din 端口与其他时钟无关。

您还可使用“Timing Summary”（时序汇总）报告并选中 -report_unconstrained 选项来运行同样的分析。设计中仅有时钟约束的情况下，“Unconstrained Paths”（未约束的路径）部分显示结果如下：

```
------------------------------------------------
| Unconstrained Path Table
------------------------------------------------
Path Group    From Clock    To Clock
----------    ----------    --------
(none)
(none)      clk1
(none)      clk2
(none)                clk1
(none)                clk2
```

不含时钟名称（或者在 Vivado IDE 中显示 <NONE>）的字段表示一组路径，这组路径的起点（源时钟）或端点（目标时钟）与时钟无关联。未约束的 I/O 端口归为此类别。您可浏览报告其余部分以检索其名称。例如，在 Vivado IDE 中，通过选择“clk1 to NONE”类别的“Setup”路径，即可在“To”列中看到由 clk1 驱动的端口：

图 105：**获取未约束的输出端口的列表**



在存储器中添加并应用新约束后，必须重新运行报告以确定哪些端口仍处于未约束状态。对于大多数设计来说，必须增加报告路径的数量，以确保报告中已列出所有 I/O 路径。

使用自动识别的采样时钟

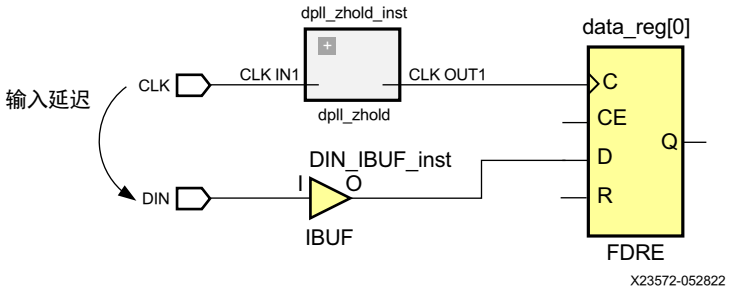
您无需指定相关时钟即可使用 set_input_delay 和 set_output_delay 约束。Vivado Design Suite 时序引擎将分析设计并自动将每个端口与所有采样时钟关联。随后，通过报告 I/O 路径上的时序，即可查看该工具约束每个 I/O 端口的方式。这样即可便于快速约束设计，但此类通用约束如果过于泛用且无法对硬件实际情况进行准确建模，则可能成为问题。

使用基准时钟

基准时钟（即传入开发板时钟）应在以下情况下使用：当它直接控制 I/O 路径时序单元，而不遍历任何时钟修改块时。I/O 延迟线不能视作为时钟修改块，因为这些延迟线仅影响时钟插入延迟，不影响波形。在 定义输入延迟 和 定义输出延迟 中提供的 2 个示例中演示了此情境。大多数情况下，外部器件的接口特性是根据相同开发板时钟定义的。

当以采用零保持时间违例 (ZHOLD) 模式的器件内的 HDIO DPLL 补偿基准时钟时，I/O 路径时序单元连接到基准时钟的内部副本（例如，生成时钟），并且该副本已采用检相器进行纠偏。由于两个时钟的波形完全相同，AMD 建议使用基准时钟作为输入/输出延迟约束的参考时钟。

图 106：**时钟路径中存在 ZHOLD DPLL 时的输入延迟**



这些约束与 定义输入延迟 中提供的示例完全相同，因为 HDIO ZHOLD DPLL 充当具有负插入延迟的时钟缓冲器，此插入延迟对应于补偿的量。

使用虚拟时钟

当板载时钟遍历某一时钟修改块以对波形进行变换并补偿整体插入延迟时，建议使用虚拟时钟代替开发板时钟作为输入和输出延迟的参考时钟。虚拟时钟的使用场景主要有以下 3 种：

- 内部时钟与板载时钟具有不同周期：虚拟时钟必须定义为：具有与内部时钟相同的周期和波形。其结果是要求 I/O 路径为常规的单周期路径。
- 对于输入路径来说，内部时钟相比于板载时钟拥有正向移位波形：虚拟时钟的定义类似板载时钟，针对建立时间定义一条从虚拟时钟到内部时钟的多周期路径（双周期）约束。上述约束导致强制执行建立时间时序分析，并且分析要求为 1 个时钟周期加相位移动量。
- 对于输出路径来说，内部时钟相比于板载时钟拥有负向移位波形：虚拟时钟的定义类似板载时钟，针对建立时间定义一条从内部时钟到虚拟时钟的多周期路径（双周期）约束。上述约束导致强制执行建立时间时序分析，并且分析要求为 1 个时钟周期加相位移动量。

综上，使用虚拟时钟需调整默认时序分析，以避免将 I/O 路径作为具有苛刻而又不切实际的要求的时钟域交汇路径来处理。

重要提示！当相移导致时钟波形发生修改时，针对具有相移时钟的 I/O 路径只需使用多周期路径即可。将相移添加到时钟修改块的插入延迟并保留时钟波形时，无需使用多周期路径。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

例如，假设 `sysClk` 板载时钟频率为 100 MHz，与 MMCM 相乘后生成运行频率为 266 MHz 的 `clk266`。由 `clk266` 生成的输出应使用 `clk266` 作为参考时钟。如果试图使用 `sysClk` 作为参考时钟（对应 `set_output_delay` 规格），那么它将显示为异步时钟，并且该路径无法再作为单周期路径进行时序约束。

使用生成时钟

对于输出源同步接口，设计会生成 1 个内部时钟的副本并将其随同数据转发给开发板。如需控制并报告转发时钟与数据之间的相位关系（偏差），即可使用此时钟作为输出数据延迟约束的参考时钟。转发时钟同样能用于系统同步接口的输入与输出延迟约束。

参考时钟上升沿和下降沿

I/O 约束使用的时钟沿必须与连接到器件的外部器件的数据手册一致。默认情况下，`set_input_delay` 和 `set_output_delay` 命令定义 1 个相对于参考时钟上升沿的延迟约束。您必须使用 `clock_fall` 选项来设定 1 个相对于时钟下降沿的延迟。此外，您还可以使用 `add_delay` 选项为相对于时钟上升沿和下降沿的延迟分别指定约束，且在端口上指定第二个约束。

大多数情况下，I/O 参考时钟沿对应于用来锁存或发送器件内 I/O 数据的时钟沿。通过分析 I/O 时序路径，可复查已使用的时钟沿，并核实这些时钟沿与其实际硬件行为是否对应。对于仅与时钟下降沿内部相关的 I/O 路径而言，如果误将时钟上升沿用作为此类路径的参考时钟，那么路径要求为 ½ 周期，这会导致时序收敛更加困难。

验证延迟约束

输入 I/O 时序约束后，复查 I/O 路径上的时序分析方式以及建立时间和保持时间检查的裕量违例数量就显得至关重要。通过使用进出所有端口的时序报告进行建立时间和保持时间分析（即，`delay type = min_max`）即可验证：

- 是否已将正确的时钟和时钟沿用作延迟约束的参考。
- 是否已使用期望时钟来发送和捕获器件内部的 I/O 数据。
- 是否能通过布局或设置适当的延迟界限配置以合理方式修复违例。如果无法修复，则必须复查约束中输入的 I/O 延迟值，并评估其合理性，以及是否需要修改设计以满足时序要求。

I/O 路径报告命令行示例

```
report_timing -from [all_inputs] -nworst 1000 -sort_by group \
-delay_type min_max
```

```
report_timing -to [all_outputs] -nworst 1000 -sort_by group \
-delay_type min_max
```

I/O 延迟约束错误会导致无法实现时序收敛。实现工具由时序驱动并且致力于对布局布线进行最优化以满足时序要求。如果 I/O 路径要求无法得到满足，并且 I/O 路径在设计中发生最严重的违例，那么总体设计 QoR 将受到影响。

输入至输出馈通路径

有多种等效的方法可用来约束从输入端口到输出端口间的组合路径。

示例 1

使用周期大于或等于馈通路径的目标最大延迟的虚拟时钟，并按如下方式应用最大输入和输出延迟约束：

```
create_clock -name vclk -period 10
set_input_delay -clock vclk <input_delay_val> [get_ports din] -max
set_output_delay -clock vclk <output_delay_val> [get_ports dout] -max
```

其中

```
input_delay_val(max) + feedthrough path delay (max) + output_delay_val(max)
<= vclk period.
```

本例中，仅约束最大延迟。

示例 2

在馈通端口之间使用最小延迟与最大延迟约束组合。示例：

```
set_max_delay -from [get_ports din] -to [get_ports dout] 10
set_min_delay -from [get_ports din] -to [get_ports dout] 2
```

这是同时约束路径上的最小延迟和最大延迟的简单方法。时序分析期间将同时使用相同端口上的所有现有输入和输出延迟约束。因此，这种方法并不常用。

最大延迟通常针对“Slow Timing Corner”（慢速时序角点）进行最优化和报告，而最小延迟则发生在“Fast Timing Corner”（快速时序角点）中。最好对馈通路径延迟约束运行几次迭代，以确认其合理性并确保实现工具可满足这些约束要求，当布局的端口间距离相去较远时尤其如此。

使用 XDC 模板 - 源同步接口

AMD 建议针对源同步接口使用 I/O 约束模板。可采用多种方法来编写源同步约束。Vivado Design Suite 所提供的模板基于默认时序分析路径要求。其语法更简单，但必须调整延迟值来解释执行建立时间分析时为何采用了不同发送沿和捕获沿（1 个周期或 1/2 个周期），而未采用相同的发送沿和捕获沿（0 个周期）。由于时钟沿不直接对应于硬件中的活动时钟沿，因此导致时序报告更难以读取。您可在 Vivado IDE 中通过如下操作导航到这些模板：“Tools” → “Language Templates” → “XDC” → “Timing Constraints” → “Input Delay Constraints” → “Source Synchronous”（工具 > 语言模板 > XDC > 时序约束 > 输入延迟约束 > 源同步）。

定义时钟组和 CDC 约束

默认情况下，Vivado IDE 用于对设计中所有时钟之间的路径进行时序约束。您可使用以下约束来修改此默认行为：

- `set_clock_groups`：禁用您识别的时钟组之间的时序分析，但不禁用同一个组中的时钟之间的时序分析。
- `set_false_path`：仅禁用由 `-from` 和 `-to` 选项所指定的方向上的时钟之间的时序分析。

在某些情况下，您可能想要对时钟域交汇 (CDC) 的一条或多条路径使用以下约束来限制时延或总线偏差：

- `set_max_delay -datapath_only`：对异步 CDC 路径设置最大延迟约束，以限制时延。

  注释：如果在时钟组之间或者相同 CDC 路径上已存在时钟组或伪路径约束，那么将忽略最大延迟约束。因此，重要的是完整复查所有时钟对之间的每条路径，然后再逐一选择 CDC 时序约束，以避免约束冲突。

  建议：AMD 还建议运行 `report_methodology` 以确认何时 `set_clock_groups` 或 `set_false_path` 约束将覆盖 `set_max_delay -datapath_only` 约束。

- `set_bus_skew`：使用总线偏差代替时延来约束异步 CDC 路径之间的一组信号。

  提示：您还可通过 Vivado IDE 来设置总线偏差约束。在“Timing Constraints”（时序约束）窗口中，展开“Assertions”（断言），然后双击“Set Bus Skew”（设置总线偏差）。

相关信息

运行 Report Methodology

检查时钟交互

相互间含逻辑路径的时钟需进行时序约束。可能的时钟关系包括：同步、异步和专属。

同步

当 2 个时钟具有固定相位关系时，时钟关系即为同步。共享以下对象的 2 个时钟即可视为具有固定相位关系：

- 公用电路（公共节点）
- 基准时钟（相同初始相位）

异步

当时钟不具有固定相位关系时，时钟关系即为异步。满足以下任一条件的时钟之间即为异步关系：

- 在设计中不共享任何公用电路，并且不具有公用基准时钟。
- 在 1000 个周期（不可扩展）内不具有公共周期，并且时序引擎无法通过正确的时序约束将其组合在一起。
- 具有公用时钟，但不共享公共节点。
- 其所属拓扑结构无法通过时钟自动衍生流程来确保已知的相位关系。

如果 2 个时钟同步，但其公共周期非常短，那么建立路径要求将过于苛刻，导致无法满足时序要求。AMD 建议您将 2 个时钟作为异步来处理，并实现安全的异步 CDC 电路。

专属

如果时钟关系在同一个时钟树上传输并到达相同的时序单元时钟管脚，但无法以物理方式同时激活，那么此时钟关系即为专属关系。

时钟对分类

可使用“Clock Interaction”（时钟交互）和“Check Timing”（检查时序）报告对时钟对进行分类。

“Clock Interaction”报告

“Clock Interaction”（时钟交互）报告可提供有关如何对 2 个时钟一起定时的高层次综述：

- 2 个时钟是否具有公用基准时钟？正确定义时钟时，设计中相同来源的所有时钟即共享相同的基准时钟。
- 2 个时钟是否具有公共周期？当时序引擎无法判定最消极的建立或保持关系时，这会显示在建立或保持路径要求列中，该列不可展开。
- 时钟组或时序例外约束是否部分或完全覆盖 2 个时钟之间的路径？
- 2 个时钟之间的建立路径要求是否极为苛刻？当 2 个时钟同步，但其周期未指定为精确倍数关系（例如，由于舍入）时，会出现此问题。经过多个时钟周期后，边沿可能出现偏离，导致最差情况时序要求变得极为苛刻。

“检查时序”报告

“Check Timing”（检查时序）报告 (`multiple_clock`) 可识别多个时钟连接到的时钟管脚，而在这些时钟之间尚未定义 `set_clock_groups` 或 `set_false_path` 约束。

约束专属时钟组

您可使用常规时序或时钟网络报告来检查时钟路径，并识别如下情况：在相同时钟树上传输 2 个时钟，以及在时序路径中同时使用 2 个时钟（在该时序路径中，起点和端点时钟管脚连接到相同时钟树）。此分析任务相当耗时。您可改为查看“检查时序”报告的 `multiple_clock` 部分。该部分会返回包含时钟管脚及其相关时序时钟的列表。

根据时钟树拓扑结构，您必须应用以下段落中所述的不同约束。

相同时钟源上定义的重叠时钟

在相同网表对象上使用 `create_clock -add` 命令定义两个时钟，并表示单一应用的多种模式时，会出现此情况。在此情况下，可在两个时钟之间安全应用时钟组约束。例如：

```
create_clock -name clk_mode0 -period 10 [get_ports clkin]
create_clock -name clk_mode1 -period 13.334 -add [get_ports clkin]
set_clock_groups -physically_exclusive -group clk_mode0 -group clk_mode1
```

如果 `clk_mode0` 和 `clk_mode1` 时钟生成其他时钟，那么还需对其生成时钟应用相同的约束，操作方式如下所述：

```
set_clock_groups -physically_exclusive \
-group [get_clocks -include_generated_clock clk_mode0] \
-group [get_clocks -include_generated_clock clk_mode1]
```

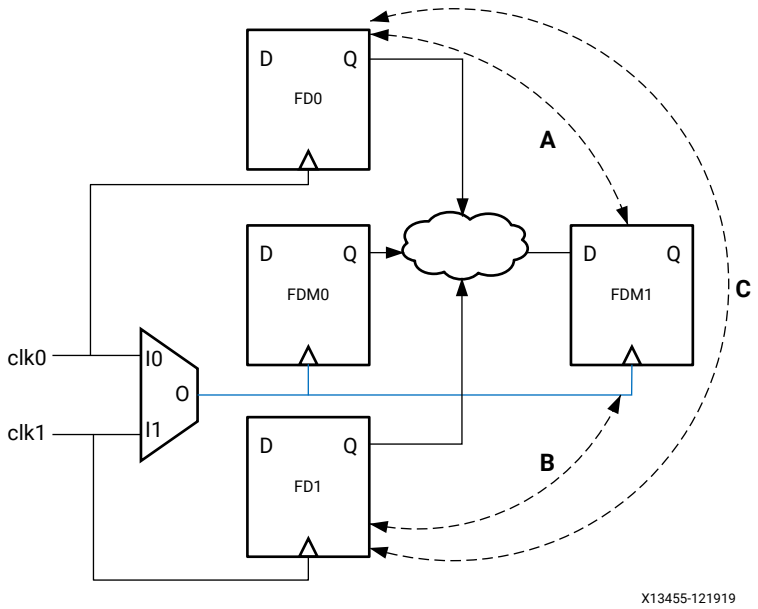
由时钟多路复用器驱动的重叠时钟

如果有 2 个或更多个时钟驱动到某 1 个多路复用器（或者更普遍的情况下，驱动到组合单元内）中，这些时钟全部都能顺利完成传输，并在单元扇出上重叠。实际上，每次只能传输 1 个时钟，但时序分析允许同时报告多种时序模式。

因此，您必须审查 CDC 路径，并添加新约束，以忽略部分时钟关系。正确的约束取决于设计中时钟交互的方式和位置。

下图演示了 2 个时钟驱动到同一个多路复用器中的示例，并演示了在此多路复用器前后这 2 个时钟之间可能发生的交互。

图 107：**多路复用时钟**



- 路径 A、B 和 C 都不存在的情况

clk0 和 clk1 仅在多路复用器（FDM0 和 FDM1）的扇出中交互。毋庸置疑，时钟组约束可直接应用到 clk0 和 clk1 中。

```
set_clock_groups -logically_exclusive -group clk0 -group clk1
```

- 仅存在路径 A、B 或 C 之一的情况

  clk0 和/或 clk1 与多路复用时钟直接交互。为了保留时序路径 A、B 和 C，无法直接向 clk0 和 clk1 直接应用约束。而是改为必须将其应用于多路复用器的扇出中需要额外的时钟定义的时钟部分。

```
create_generated_clock -name clk0mux -divide_by 1 \
-source [get_pins mux/I0] [get_pins mux/O]

create_generated_clock -name clk1mux -divide_by 1 \
  -add -master_clock clk1 \
-source [get_pins mux/I1] [get_pins mux/O]

set_clock_groups -physically_exclusive -group clk0mux -group clk1mux
```

对异步时钟组和时钟域交汇进行约束

在“Clock Interaction”（时钟交互）报告中可快速明确异步关系：无公用基准时钟的时钟对或者无公共周期（未扩展）的时钟对。即使时钟周期相同，从不同时钟源生成的时钟仍为异步关系。必须仔细审查异步“Clock Domain Crossing (CDC)”（时钟域交汇 (CDC)）路径以确保这些路径使用的同步电路正确，此类同步电路不依赖时序正确性，并且可以最大限度降低发生亚稳态的概率。异步 CDC 路径通常具有较高的偏差要求和/或不现实的路径要求。因此不应使用默认时序分析来对其进行时序约束，此分析无法证明其能否在硬件中正常工作。

Report CDC

Report CDC (report_cdc) 命令可执行设计中时钟域交汇的结构分析。您可使用此信息来识别潜在不安全的 CDC，此类 CDC 可能导致亚稳态或数据一致性问题。Report CDC 类似于“Clock Interaction”（时钟交互）报告，但 Report CDC 侧重于结构和相关的时序约束。Report CDC 不提供时序信息，因为时序裕量对于跨异步时钟域的路径没有意义。

Report CDC 可识别如下最常见的 CDC 拓扑结构：

- 单位同步装置
- 多位总线同步装置
- 异步复位同步装置
- 由 MUX 和 CE 控制的电路系统
- 同步装置前组合逻辑
- 多时钟扇入到同步装置
- 扇出到目标时钟域

如需了解有关 report_cdc 命令的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。另请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835) 中的 report_cdc。

应采用特定约束以避免异步时钟域交汇上执行默认时序分析。

相关信息

双向时钟间的全局约束
对应各 CDC 路径的约束

双向时钟间的全局约束

如果无需限制最大时延，即可使用时钟组。以下是忽略 clkA 与 clkB 之间的路径的示例：

```
set_clock_groups -asynchronous -group clkA -group clkB
```

当 2 个主时钟及其相应的生成时钟构成 2 个异步域，并且这 2 个异步域之间的所有路径均已正确完成同步时，即可立即对多个时钟应用时钟组约束：

```
set_clock_groups -asynchronous \
-group {clkA clkA_gen0 clkA_gen1 } \
-group {clkB clkB_gen0 clkB_gen1 }
```

或者直接执行：

```
set_clock_groups -asynchronous \
-group [get_clocks -include_generated_clock clkA] \
-group [get_clocks -include_generated_clock clkB]
```

对应各 CDC 路径的约束

如果 CDC 总线使用格雷编码（例如，FIFO）或者如果需要限制 1 个或多个信号上的 2 个异步时钟之间的时延，则必须使用 set_max_delay 约束及 -datapath_only 选项来忽略这些路径上的时钟偏差和抖动，并覆盖时延要求的默认路径要求。通常使用源时钟周期作为最大延迟值就足够了，这只是为了确保在任意给定时间，CDC 路径上最多仅存在一项数据。

当时钟周期之间的比率较高时，选择源时钟周期和目标时钟周期的最小值同样足以降低传输时延。简单标准的异步 CDC 路径的源时序单元与目标时序单元之间不应存在任何逻辑，因此实现工具很容易就可以满足“Max Delay Datapath Only”（仅最大延迟数据路径）约束。

某些异步 CDC 路径要求在总线的各个位之间施加偏移控制，而无需对总线时延施加约束。使用总线偏移约束可防止接收时钟域在同一时钟沿上锁存总线的多个状态。您可使用 set_bus_skew 命令来对总线设置总线偏移约束。例如，您可将 set_bus_skew 应用于使用格雷编码代替“Max Delay Datapath Only”（仅最大延迟数据路径）约束的 CDC 总线。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：使用约束》(UG903) 中的相应内容。

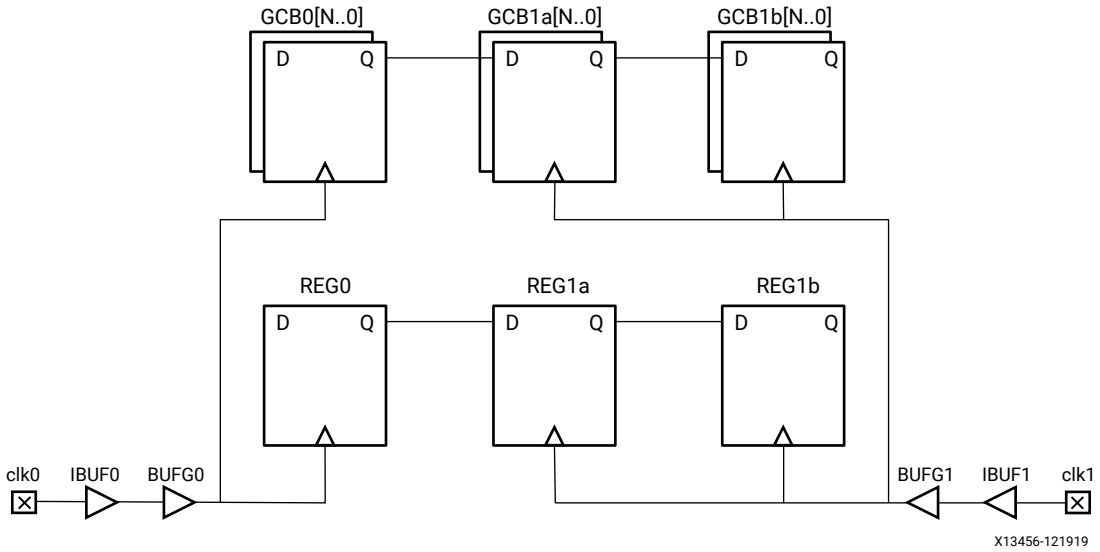
对于不需要时延控制的路径，您可定义 1 个点对点伪路径约束。

时钟例外相对于 set_max_delay 的优先顺序

编写 CDC 约束时，请验证是否遵循相应的优先顺序。如果在 2 个时钟之间至少 1 条路径上使用 set_max_delay -datapath_only，那么无法在相同时钟之间使用 set_clock_groups 约束，并且只能在 2 个时钟之间的其他路径上使用 set_false_path 约束。

在下图中，时钟 clk0 的周期为 5 ns，并且与 clk1 之间处于异步关系。从 clk0 域到 clk1 域存在两条路径。第 1 条路径为 1 位数据同步。第 2 条路径为多位格雷编码总线传输。

图 108：**2 个异步时钟之间的多次交互**



设计师判定格雷编码总线传输需要“Max Delay Datapath Only”（仅最大延迟数据路径）来限制比特间延迟变动，因此无法在时钟之间直接使用“Clock Group”（时钟组）或“False Path”（伪路径）约束。而改为必须定义以下 2 个约束：

```
set_max_delay -from [get_cells GCB0[*]] -to [get_cells [GCB1a[*]] \
-datapath_only 5
set_false_path -from [get_cells REG0] -to [get_cells REG1a]
```

无需设置从 `clk1` 到 `clk0` 的伪路径，因为在此示例中不含任何路径。

指定时序例外

时序例外用于修改对特定路径执行时序分析的方式。默认情况下，时序引擎假定所有路径都应通过单一建立时间分析周期要求来完成时序约束，以便覆盖大部分消极的时钟设置场景。对于某些路径，并非如此。以下提供一些示例：

- 由于时钟间缺乏固定的相位关系，导致无法安全完成异步 CDC 路径的时序约束。此类状况应予以忽略（时钟组，伪路径），或者只需设置数据路径延迟约束（仅最大延迟数据路径）即可
- 时序单元发送沿和捕获沿并非在每个时钟周期内都处于活动状态，因此可相应降低路径要求（多周期路径）
- 路径延迟要求需收紧，以增加硬件中的设计裕度（最大延迟）
- 通过组合单元的路径为静态路径，无需时序约束（伪路径，案例分析）
- 应仅限对多路复用器驱动的特定时钟执行分析（案例分析）。

无论在任何情况下，都必须谨慎使用时序例外，并且不得为了隐藏实际时序问题而添加例外。

时序例外准则

请尽量限制使用的时序例外的数量，并使时序例外尽可能保持简单。否则，您将面临下面两大挑战：

- 如果过多使用例外，实现编译时间将显著增加，当这些例外与大量网表对象相关联时尤其如此。
- 当多个例外覆盖相同路径时，约束调试会变得极为复杂。

- 对信号施加约束会阻碍该信号的最优化。因此无论是包含非必要的例外还是在例外命令中包含非必要的点，都会妨碍信号最优化。

以下是可能会对运行时间产生不利影响的时序例外示例：

```
set_false_path -from [get_ports din] -to [all_registers]
```

- 如果 `din` 端口没有输入延迟，那么它将不受约束。因此无需添加伪路径。
- 如果 `din` 端口仅供给时序元件，那么无需对时序单元显式指定伪路径。按如下方法编写此约束更有效：

  ```
  set_false_path -from [get_ports din]
  ```

- 如果需要伪路径，但从 `din` 端口到设计中的任意时序单元之间仅存在几条路径，那么约束可以更明确（`all_registers` 可能会返回数千个单元，这取决于设计中使用的寄存器数量）：

  ```
  set_false_path -from [get_ports din] -to [get_cells blockA/config_reg[*]]
  ```

时序例外优先级规则

时序例外需遵循严格的优先级规则。最重要的规则包括：

- 约束越具体，优先级越高。例如：

  ```
  set_max_delay -from [get_clocks clkA] -to [get_pins inst0/D] 12
  set_max_delay -from [get_clocks clkA] -to [get_clocks clkB] 10
  ```

  第一项 `set_max_delay` 约束优先级更高，因为 `-to` 选项使用管脚，这比时钟更为具体。

- 例外优先级如下所示：

  1. `set_false_path`
  2. `set_max_delay` 或 `set_min_delay`
  3. `set_multicycle_path`

`set_clock_groups` 命令不视为时序例外，即使它等同于 2 个时钟之间的 2 条 `set_false_path` 命令也是如此。它的优先级高于时序例外。

`set_case_analysis` 命令和 `set_disable_timing` 命令用于禁用特定设计部分上的时序分析。其优先级高于时序例外。

如需了解有关 XDC 优先级的详细信息，请访问此链接以参阅《Vivado Design Suite 用户指南：使用约束》(UG903) 中的相应内容。

添加伪路径约束

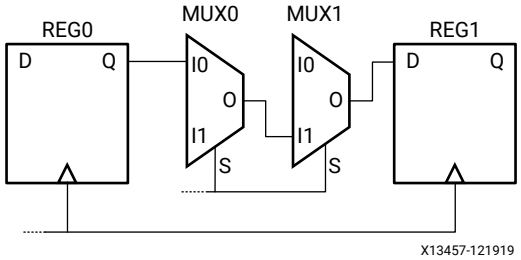
可向时序路径添加伪路径例外以忽略这些路径上的时序裕量计算。通常，要证明路径无需满足时序要求亦可正常运行是极为困难的，即使使用仿真工具也是如此。AMD 通常不建议使用伪路径，除非已正确评估并判断可接受与之关联的风险。

用例

伪路径约束的典型用例为：

- 忽略从不活动的路径上的时序。例如，如果某条路径穿过 2 个多路复用器，这 2 个多路复用器由于所选管脚连接，导致绝不允许在同一时钟周期内传输数据，则忽略该路径。

图 109：**不能激活的路径**




```
set_false_path -through [get_pins MUX0/I0] -through [get_pins MUX1/I1]
```

- 忽略异步 CDC 路径上的时序。
- 忽略设计中的静态路径。有些寄存器在应用的初始化阶段接收 1 个值后，就不再切换。当这些寄存器出现在设计的关键路径上时，可以忽略其时序，以放宽对实现工具的约束，从而有助于实现时序收敛。仅定义伪路径约束始于静态寄存器即可，无需明确指定路径端点。例如，通过添加如下伪路径约束，便可忽略从 32 位配置寄存器 `config_reg[31..0]` 到设计其余部分的路径：

```
set_false_path -from [get_cells config_reg[*]]
```

对实现的影响

所有实现步骤都对伪路径时序例外比较敏感。

对综合的影响

伪路径约束受综合支持，并且仅影响最大延迟（建立/恢复时间）路径最优化。对于设计中可安全忽略其中时序的部分以及无数据路径延迟要求的异步 CDC 路径，建议采用伪路径时序例外。

对实现的影响

所有实现步骤都对伪路径时序例外比较敏感。

添加最小和最大延迟约束

最小和最大延迟例外用于覆盖分别对应于保持/移除检查和建立/恢复检查的默认路径要求，方法是将发送沿时间和捕捉沿时间替换为约束中的延迟数值。

用例

以下描述了使用最小或最大延迟约束的部分常见原因：

- 通过收紧建立/恢复路径要求，对部分设计路径进行过约束。

  这有助于强制要求逻辑最优化或布局工具全力以赴处理部分关键路径单元，从而提升布线器的灵活性以便稍后移除最大延迟约束后满足时序要求。

- 替换多周期约束。

  对于每 N 个时钟周期存在活动的发送沿和捕获沿的路径，如需在此类路径上设置宽松的建立时间要求，此方法有效，但并不推荐使用。但只有采用这种方法才能在任一时钟周期内对多周期路径进行短时间过约束，从而帮助在布线步骤阶段实现时序收敛。例如，如果某条多周期约束为 3 的路径于布线后成为最差的违例路径并以数百 ps 的差距未能满足时序要求，

那么在最优化和布局期间原多周期路径约束将替换为如下约束，其中，14.5 对应 3 个时钟周期（每个周期 5 ns）减去 500 ps（对应于期望的裕度）：

```
set_max_delay -from [get_pins <startpointCell>/C] \
-to [get_pins <endpointCell>/D] 14.5
```

- 约束异步 CDC 路径上的最大数据路径延迟

  如需了解此技巧的详情，请参阅 定义时钟组和 CDC 约束。

不建议使用 `set_min_delay` 约束对路径强制执行额外的延迟插入，并且此方法并不常用。在裕量为正值时，保持或移除的默认最小延迟要求通常足以确保硬件正常运作。

对综合的影响

综合支持 `set_max_delay` 约束（包括 `-datapath_only` 选项）。`set_min_delay` 约束将被忽略。

对实现的影响

`set_max_delay` 约束会取代建立路径要求并影响整个实现流程。`set_min_delay` 约束会取代保持路径要求，仅当引入该约束以修复保持时间违例时才会影响布线器行为。

避免路径分段

仅当针对 `set_max_delay` 和 `set_min_delay` 命令的 `-from` 或 `-to` 选项指定的起点或端点无效时，才会引入路径分段。当 `set_max_delay` 在路径上引入路径分段后，将不再发生默认保持时间分析。如果还要对保持时间分析进行约束，请使用 `set_min_delay` 来对该路径进行约束。此规则同样适用于建立时间分析相关的 `set_min_delay` 命令。

路径分段只能由专家级用户使用，因为它会改变时序分析的基础：

- 路径分段会对分段路径上的时钟偏差计算进行拆分。
- 路径分段拆分的路径数量比用于分段的 `set_max_delay` 或 `set_min_delay` 命令约束的路径数量更多。

有效的起点和端点

应用约束时，工具在 log 日志文件中报告路径分段。您必须使用有效的起点和端点来避免此类问题：

- 起点：时钟、时钟管脚、时序单元（表示单元的有效起点管脚）、输入或输入输出端口。
- 端点：时钟、时序单元的输入数据管脚、时序单元（表示单元的有效端点管脚）、输出或输入输出端口。

如需了解有关路径分段的详细信息，请访问此链接以参阅《Vivado Design Suite 用户指南：使用约束》(UG903) 中的相应内容。

添加多周期路径约束

多周期路径例外处理必须反映设计的功能性，并且在源时钟和/或目标时钟上，只要路径并非在每个周期内都包含活动时钟沿，就必须应用此例外处理。根据使用 `-start` 选项时的源时钟或使用 `-end` 选项时的目标时钟，路径倍频器以时钟周期的形式来表示。这样尤其便于独立于时钟周期值来修改起点和端点之间的建立时间和保持时间关系。

保持时间关系始终与建立时间关系相关联。由此导致大多数情况下，修改建立时间关系后还需单独调整保持时间关系。因此才需要使用含 `-hold` 选项的第二个约束。此规则的主要例外是相移时钟之间的同步 CDC 路径：只需修改建立时间即可。

放宽建立时间要求，同时保持时间要求保留不变

当源时序单元和目标时序单元受时钟使能信号控制并且此信号每 N 个周期就会激活时钟时，此状况就会发生。以下示例中，时钟使能每 3 个周期就会激活一次，并且起点和端点时钟相同：

图 110：**使用相同时钟信号启用的触发器**

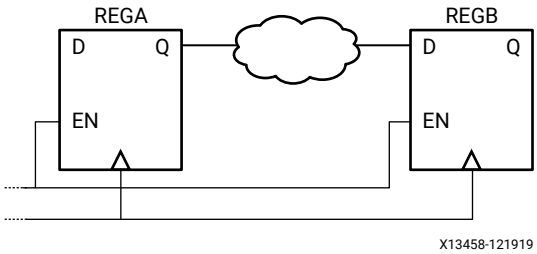
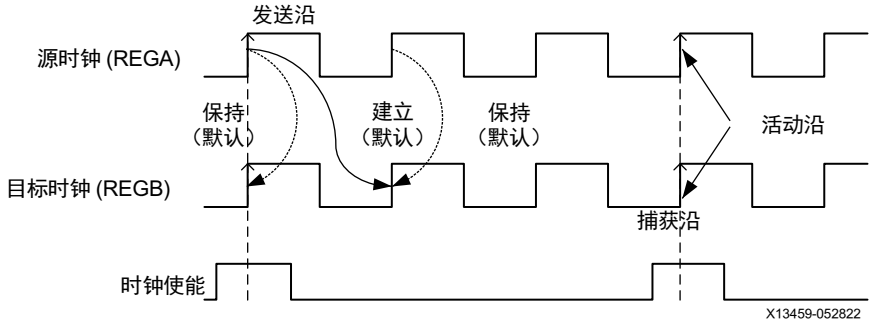


图 111：**建立/保持检查的时序图**

执行前

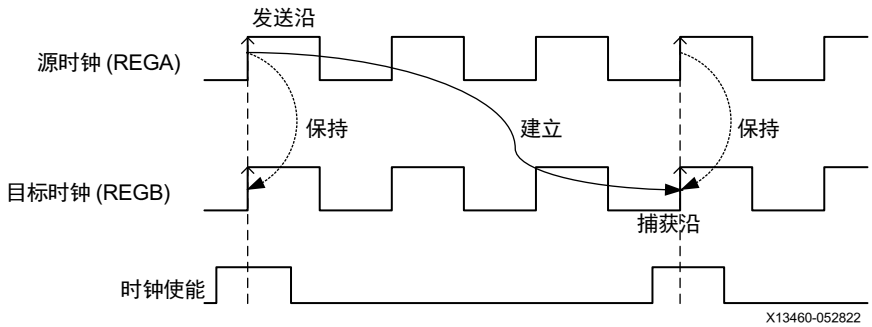


约束：

```
set_multicycle_path -from [get_pins REGA/C] -to [get_pins REGB/D] -setup 3
set_multicycle_path -from [get_pins REGA/C] -to [get_pins REGB/D] -hold 2
```

图 112：**多周期规格应用后修改的建立/保持检查**

执行后



注释：运行第 1 条命令后，当建立捕获沿移至第 3 沿（即，距离其默认位置达 2 个周期），保持沿同样移动 2 个周期。第 2 条命令用于使保持沿再次移动 2 个周期（反向）从而返回至其原始位置。

如需了解有关其他公用多周期路径场景（例如，同步时钟之间的相移和多周期路径）的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：使用约束》(UG903) 中的相应内容。

重要提示！当时钟相移不修改时钟波形，而是改为包含在时钟修改块的插入延迟中时，无需添加 1 个仅限建立的多周期路径用于对往来时钟的路径进行正确的时序约束。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

对综合与实现的影响

综合步骤支持 `set_multicycle_path` 约束，该约束可将每个时钟周期内功能上处于不活动状态的长路径约束加以放宽，从而大幅提升时序 QoR（仅针对建立时间）。

就综合而言，多周期路径例外可帮助由实现时序所驱动的算法集中处理真正关键的路径。只有在布线过程中，保持时间要求才至关重要。如果通过 `set_multicycle_path` 约束调整建立关系而不调整其对应的保持关系，那么最差保持要求一旦超过 2 或 3 ns，就会变得难以满足。这种情况会对建立裕量产生不利影响，因为在修复保持违例时布线器插入了附加延迟。

常见错误

以下是必须避免的常见错误：

- 多周期路径功能并非在每个时钟周期内都处于活动状态，在此情况下放宽“建立时间”要求，但却不将“保持时间”调整恢复到相同的发送沿和捕获沿。

  保持要求会变得非常高（大多数情况下为至少 1 个时钟周期），且无法满足。

- 在设计中错误的时间点之间设置多周期路径例外处理。

  当假定从起点单元到端点单元只有一条路径时会发生上述情况。在有些情况下也不尽然。端点单元可包含多个数据输入管脚，包括时钟使能和复位管脚，这些管脚在至少 2 个连续时钟沿上处于活动状态。

  因此，AMD 建议指定端点管脚而非单元（或时钟）。例如，端点单元 REGB 有 3 个输入管脚：C、EN 和 D。只有 REGB/D 管脚需要由多周期路径例外处理进行约束，EN 管脚不需要，因为在每个时钟周期它都会发生变化。如果将约束连接至单元而不是管脚，那么包括 EN（时钟使能）管脚在内的所有有效的端点管脚都在约束的考虑范围内。

  为安全起见，AMD 建议您始终使用如下语法：

```
set_multicycle_path -from [get_pins REGA/C] \
-to [get_pins REGB/D] -setup 3
set_multicycle_path -from [get_pins REGA/C] \
-to [get_pins REGB/D] -hold 2
```

其他高级时序约束

通过设置一些其他时序约束可忽略和修改默认时序分析，欲知详情，请参阅以下章节。

案例分析

案例分析命令常用于通过在逻辑中设置约束来描述设计中的功能模式，例如，要使用哪些配置寄存器。此命令可应用于输入端口、信号线、层级管脚或叶节点输入管脚。此常量值通过逻辑传输，用于关闭从不活动的任何路径上的分析。其作用与伪路径例外的工作方式相似。

最常见的示例是将多路复用器选择管脚设置为 0 或 1，这样仅允许 1 个或 2 个多路复用器输入通过此管脚传输。以下显示的是在穿过 `mux/S` 和 `mux/I1` 管脚的路径上关闭分析的示例：

```
set_case_analysis 0 [get_pins mux/S]
```

禁用时序

禁用时序命令可关闭时序数据库中的时序 arc，从而完全阻止通过此 arc 进行任何分析。禁用的时序 arc 可通过 `report_disable_timing` 命令来报告。

注意！请谨慎使用禁用时序命令。它会造成大量路径中断！

数据检查

`set_data_check` 命令用于在设计中 2 个管脚之间设置建立时序或保持时序检查的等效检查。例如，此约束可用于报告异步接口上的时序。实现工具会忽略此命令，它仅用于时序报告，通常仅限专家级用户使用。

最大时间借用量

`set_max_time_borrow` 命令用于设置锁存器可从下一阶段（锁存后逻辑）借用的最大时间量，以及提供给其上一阶段（锁存前逻辑）的最大时间量。通常不建议使用锁存器，因为在硬件中难以对其进行测试和确认。该命令应由专家级用户使用。

为抖动指定 RAM 活动

USER_RAM_AVERAGE_ACTIVITY 约束所指定的值表示可开关（启用/禁用）的器件上的所有 UltraRAM 和块 RAM 的平均频率。该值供 Vivado 工具用于在静态时序分析中对 RAM 开关所引发的电源噪声进行建模，并计算全局时钟的抖动。在静态时序分析中，所报告的抖动属于时钟不确定性的一部分。如不指定 USER_RAM_AVERAGE_ACTIVITY，那么 Vivado 工具计算所得的 RAM_AVERAGE_ACTIVITY 值将用于抖动计算，可能增加时序收敛的难度。

重要提示！AMD 建议您为设计计算 USER_RAM_AVERAGE_ACTIVITY 以降低时序收敛难度。如需了解有关 RAM 活动抖动和计算设计的 USER_RAM_AVERAGE_ACTIVITY 的更多信息，请访问此链接以参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003) 中的相应内容。

Vivado 工具会基于所使用的 RAM 器件资源、连接到活动信号的 RAM 使能管脚以及 RAM 的工作频率，来执行扁平设计的 RAM 活动估算。Vivado 工具假定 DFX 设计的消极因素默认值为 320 MHz。Vivado 工具计算所得的 RAM 活动值将应用于顶层 `[current_design]` 对象上的只读 RAM_AVERAGE_ACTIVITY 属性。为了减少 DFX 设计中的消极因素，您可为可重配置分区的 RAM 活动规划预算，并计算设计的 USER_RAM_AVERAGE_ACTIVITY，欲知详情，请访问此链接以参阅《Versal 自适应 SoC 时钟资源架构手册》(AM003) 中的相应内容。

要指定 USER_RAM_AVERAGE_ACTIVITY 约束，请将该约束应用于 XDC 约束中的顶层 `[current_design]` 对象：

```
set_property USER_RAM_AVERAGE_ACTIVITY 160 [current_design]
```

下表显示了 RAM 开关所导致的额外时钟不确定性可能对时序收敛产生的影响。在此示例中，设计中包含的全局时钟运行频率分别为 300 MHz、400 MHz 和 500 MHz。RAM_AVERAGE_ACTIVITY 为 320，计算所得 USER_RAM_AVERAGE_ACTIVITY 约束为 160。应用约束会降低每个时钟域中所有路径的时钟不确定性，导致设计更加难以实现时序收敛。

表 10：**消极默认时钟不确定性与指定 RAM 活动时钟不确定性的比较**

时钟域	RAM_AVERAGE_ACTIVITY 320 时钟不确定性	USER_RAM_AVERAGE_ACTIVITY 160 时钟不确定性	时钟不确定性降低
300 MHz	0.103 ns	0.073 ns	-0.030 ns
400 MHz	0.089 ns	0.066 ns	-0.023 ns
500 MHz	0.078 ns	0.058 ns	-0.020 ns

定义功耗和散热约束

在 Vivado 工具或 Vitis 环境中开发设计时，您必须确保自己的设计在供电和热处理解决方案的约束范围内，这些约束通常是根据电源设计管理器 (PDM) 工具所确定的早期功耗估算来判定的。由于更改供电和热处理解决方案代价不菲，因此确保设计正确完成约束至关重要。

AMD 建议，至少应用总功耗预算、最大工艺和最差情况结温，使用以下 XDC 约束来创建最差情况功耗分析：

```
set_operating_conditions -design_power_budget <Power in Watts>
set_operating_conditions -process maximum
set_operating_conditions -junction_temp <Max Tj based on Temp Grade>
```

电源/功耗提示：为了执行最差情况功耗估算，截至热处理解决方案的 Theta Ja (ΘJa) 已知之前，AMD 建议将 Tj 设置为目标温度范围允许的最大值。Theta Ja 可基于热处理仿真结果按如下公式来计算：ΘJa = (Tj – Ta)/Pd。单位为每瓦摄氏度 (°C/W)。

已知热处理设计的 Theta Ja 之后，即可实现最准确的功耗估算。您可使用以下约束将 Theta Ja 和应用支持的最高环境温度 (Ta) 应用于 report_power 以替换结温设置。使用这些约束即可允许 report_power 更准确地估算结温，从而提供更准确的静态功耗估算。

```
set_operating_conditions -design_power_budget <Power in Watts>
set_operating_conditions -process maximum
set_operating_conditions -ambient_temp <Max Supported by Application>
set_operating_conditions -thetaja <Increase in Tj for every W dissipated
C/W>
```

此外，您可以使用 XDC 约束来指定供电设计。使用此方法即可允许 report_power 报告总功耗裕度、检查电源轨的功耗估算并基于指定的估算和电源轨整合来报告裕度。如需了解有关这些约束的更多信息，请参阅《Vivado Design Suite 用户指南：功耗分析与最优化》(UG907)。

```
create_power_rail <power rail name> -power_sources {supply1, supply2 ,..}
add_to_power_rail <power rail name> -power_sources {supply1, supply2, ..}
set_operating_conditions -supply_current_budget {<supply rail name>
<current budget in Amp>} -voltage {<supply rail name> <voltage>}
```

电源/功耗提示：请确保使用功耗报告文本获取最详细的电源轨约束报告。

定义物理约束

物理约束用于控制布局规划、特定布局、I/O 分配、布线器以及类似功能。确保已为每个管脚指定 I/O 位置和标准。以下用户指南中涵盖了有关物理约束的信息：

- 如需了解有关锁定布局和布线（包括宏的相关布局）的信息，请参阅《Vivado Design Suite 用户指南：使用约束》(UG903)。
- 如需了解布局规划相关信息，请参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906)。
- 如需了解配置相关信息，请参阅《Vivado Design Suite 用户指南：编程和调试》(UG908)。

使用 IMUX 寄存器约束

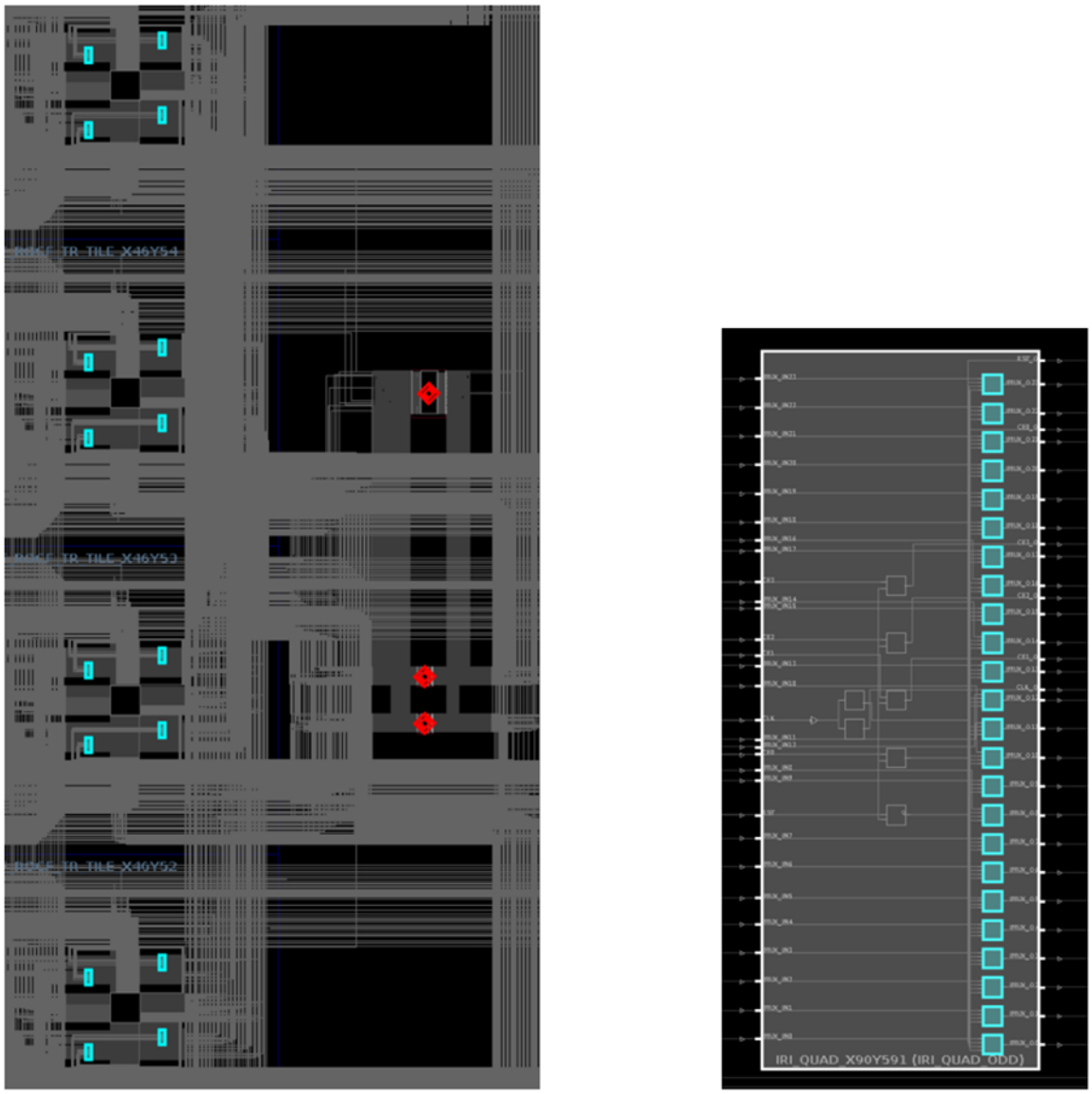
IRI_QUAD site 中存在可选的触发器阶段，用于驱动 PL 中的硬核宏。这些 IMUX 寄存器 (IMR) 按每个接口拼块 4 个 IRI_QUAD site 的方式来组织，并由输入布线 MUX 驱动。

如需了解有关使用 IMUX 寄存器改善硬核宏时序的更多信息，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

IMR 可以通过寄存驱动到硬核块（如 RAM、DSP、GT_QUAD 等）的信号来最优化接口时序。

以下左图显示了用于驱动一组 RAMB18 站点和 RAMB36 站点（标记为红色）的 IRI_QUAD 站点（青色高亮）。右侧缩放视图显示了一个含 IMUX 寄存器（青色高亮）的 IRI_QUAD。

图 113：**IRI_QUAD 中的 IMR 示例**



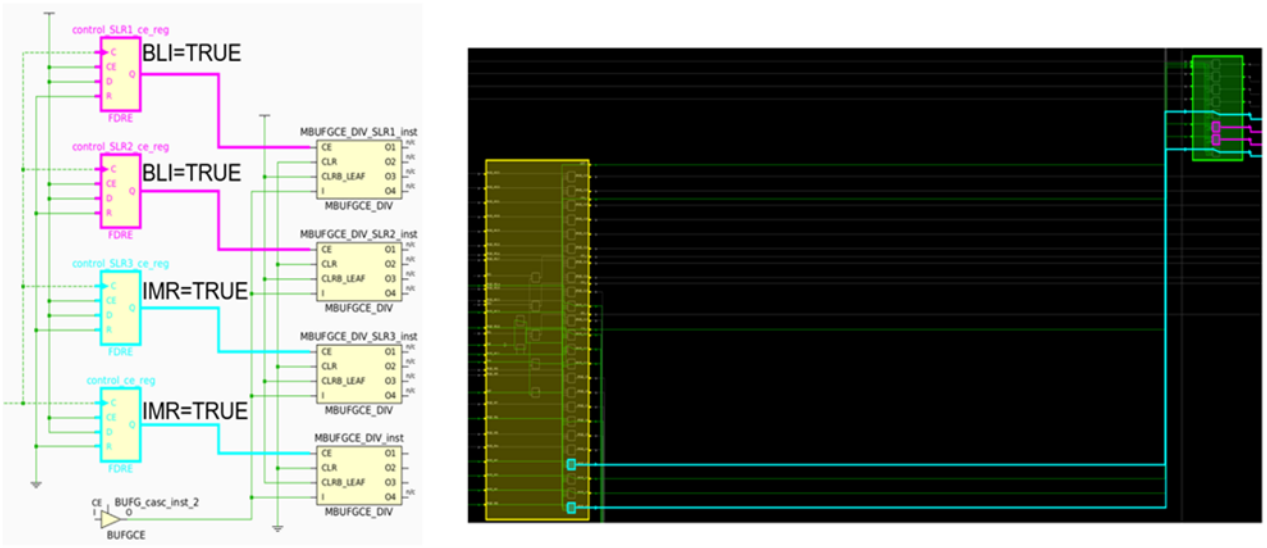
IMR 触发器资源存在如下限制：

- IMR 仅支持具有异步清除 (FDCE) 或同步复位 (FDRE) 的触发器。
- IMR 仅支持 INIT 值为 0 的 FDCE 和 FDRE。
- 任一站点 (site) 内的所有 IMR 触发器都必须共享相同活动的 `CLR` 信号或不活动的 `GND R` 信号。
- 仅当 `CLR` 管脚和 R 管脚绑定到 `GND` 时，才能混用 FDCE 和 FDRE。
- IRI_QUAD 站点内的所有 IMR 触发器都必须共享相同的 `CE` 信号。
- 只能使用数据 IMR BEL。控制 IMR BEL 在 Versal 器件中不受支持。

默认情况下，如果触发器仅连接到硬核块，那么这些触发器不布局在 IMR 触发器资源内。IMR 约束必须供 Vivado 工具用于将触发器布局在 IMR 触发器资源内。如果由于违反上述限制导致无法满足 IMR 约束，则触发器布局在 CLB 内。CLB 触发器的信号线如果使用 IMR 来与各资源对接，则会执行 IMR 触发器资源的直通式布线。在以下示例中，IMR 约束用于驱动 BLI 寄存器的 IRI_QUAD 内的触发器，这些 BLI 寄存器用于驱动 MBUFGCE_DIV/CE。选中 BLI（洋红色高亮）站点和 IMR（青色高亮）站点即可为驱动 MBUFGCE_DIVE/CE 管脚的寄存器获取最优化的建立时间/保持时间窗口。

```
set_property IMR TRUE [get_cells {control_SLR1_ce_reg control_SLR2_ce_reg}]
```

图 114：**IMR 约束示例**



使用边界逻辑接口约束

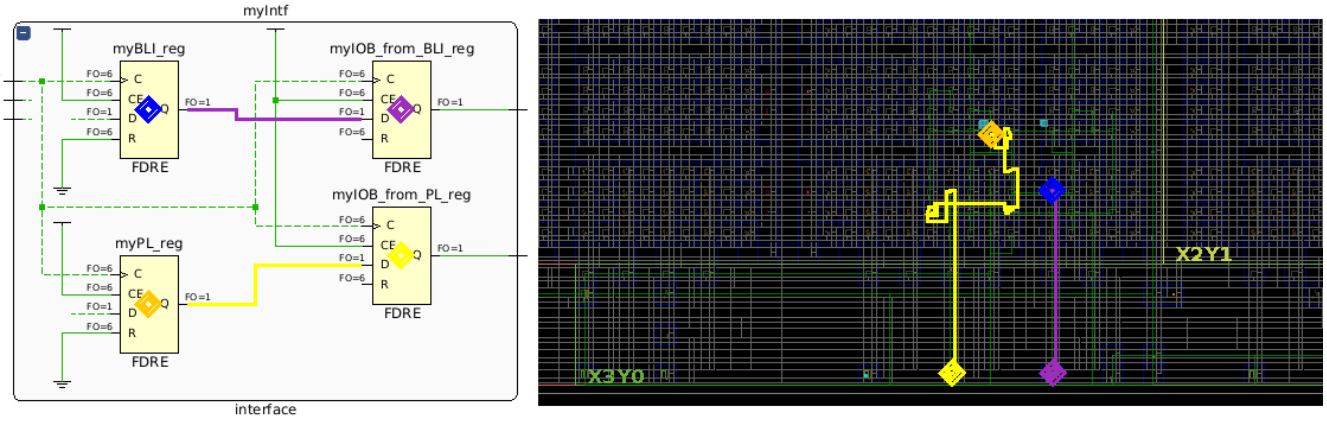
在可编程逻辑 (PL) 与高性能 XPIO 之间以及在 PL 与 AI 引擎接口拼块之间的边界逻辑接口 (BLI) 中存在触发器阶段。BLI 触发器资源可对出入 PL 的信号进行寄存，从而最优化接口时序。BLI 触发器资源存在如下限制：

- BLI 仅支持具有异步清除 (FDCE) 或同步复位 (FDRE) 并且 R 管脚绑定到 GND 的触发器。
- BLI 仅支持 INIT 值为 0 的 FDCE 和 FDRE。
- 任一站点 (site) 内的所有 BLI 触发器都必须共享相同的活动 CLR 信号或不活动的 GND R 信号。
- 仅当 CLR 管脚和 R 管脚绑定到 GND 时，才能混用 FDCE 和 FDRE。
- 任一站点 (site) 内的所有 BLI 触发器都必须共享相同的 CE 信号。

默认情况下，如果触发器仅连接到 XPIO bank 或 AI 引擎接口拼块资源，那么这些触发器不布局在 BLI 触发器资源内。BLI 约束必须供 Vivado 工具用于将触发器布局在 BLI 触发器资源内。如果由于违反上述限制导致无法满足 BLI 约束，则触发器布局在 PL 内。PL 触发器的信号线如果使用 BLI 来与各资源对接，则会执行 BLI 触发器资源的直通式布线。在以下示例中，BLI 约束供 BLI 资源中的触发器用于驱动 XPIO IOB 中的触发器。此示例还演示了驱动 XPIO IOB 中的触发器的 PL 触发器，并显示了从 BLI 穿越 BLI 触发器资源的布线。

```
set_property BLI TRUE [get_cells myIntf/myBLI_reg]
```

方程 1：**BLI 约束示例**



如需了解有关 XPIO bank 中的 BLI 的更多信息，请访问此链接以参阅《Versal 自适应 SoC SelectIO 资源架构手册》(AM010) 中的相应内容。

如需了解有关 BLI 约束的更多信息，请参阅《Vivado Design Suite 属性参考指南》(UG912) 中的 BLI。

Dynamic Function eXchange 的布局规划约束

最优化的布局规划对于在 DFX 设计内确保时序收敛和避免可布线性问题至关重要。与 DFX 布局规划中的 I/O bank 相关的规则也会影响管脚分配规划。以下是 DFX 设计中进行布局规划时需要考量的关键领域：

- Pblock

  进行 DFX 分区布局规划时，将最大量的资源分配到 RP Pblock，在静态区域中保留最少量资源用于满足平台编译需求。

  在多个 RP 间可通过水平分割（上半部分和下半部分）时钟区域来共享同一个时钟区域，但部分 Versal 器件中 PL 顶层存在的半个时钟区域除外。

  可编程单元 (PU) 粒度通常即为拼块 (tile) 本身，但 I/O 除外，其粒度是整个 I/O bank。如此精细粒度可在布局规划中提升灵活性。如需了解有关 PU 的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909) 中的相应内容。

  为了避免静态信号线渗透到 RP，可以为该静态区域创建 Pblock，并在其中启用约束布线。但此方法需额外考量静态逻辑的可布线性。欲知详情，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)。

- I/O

  在 Versal 器件中，在某一 I/O bank 内声明一个静态 I/O 会强制此 I/O bank 的所有 I/O site 归入此静态区域，但仍可重新配置时钟设置资源。在前述器件中，I/O 和时钟设置资源捆绑在同一个 PU 内，因此所有 I/O 和时钟设置资源均全部为静态资源或者全部为重配置资源，且不允许处于不同域中。

- 时钟设置

  如果 RP 使用远程时钟拼块，并且需要跨越另一个 RP，那么会根据此拼块在 RP 间的拆分方式来限制时钟轨道的使用。使用可能位于 RP 本地的 GT 时钟区域内的 DPLL 来代替位于器件底部的水平时钟区域的 MMCM。

  如需了解有关时钟区域的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909) 中的相应内容。

- 分区管脚

在 DFX 设计中，可重配置模块 (RM) 与静态区域之间的信号称为“边界信号”。所有 RM 管脚都必须包含分区管脚位置约束 (PPLOC)，此约束由布局器存入边界信号。唯一例外是硬核原语之间的专用路径。分区管脚是 PL 上的物理接口，用于分割边界信号的静态部分和可重配置部分。如需了解有关 PPLOC 的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909) 中的相应内容。

由于对应边界信号线的布线始终强制穿过分区管脚，因此分区管脚的存在会缩减布线器的解空间。为了缓解这一问题，DFX 流程包含扩展布线。扩展布线是为 RP 添加布线占板面积，此 RP 可包含来自静态区域的布线拼块。如需了解有关扩展布线的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909) 中的相应内容。

您也可以降低分区管脚的局部密度，以避免在将来实现可重配置模块变体时遇到布线困难。有多个属性（如 HD.PARTPIN_RANGE 和 HD.PARTPIN_LOCS）可用于控制分区管脚的布局。

如需了解有关 Pblock 准则、I/O 管脚分配、时钟设置和 PPLOC 缩减的更多信息，请参阅《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)。

第 10 章

设计实现

选定器件、选择并配置 IP 且编写 RTL 和约束条件后，下一步即为实现。实现通过综合和布局布线来编译设计，然后生成用于对器件进行编程的文件。实现过程可能包含一些迭代循环。本章将介绍各个实现步骤，并着重强调需特别注意的事项，同时给出识别和消除具体瓶颈的要诀和技巧。

重要提示！您必须定期确认综合与实现均正常完成，不含任何错误，仅含最少量的时序违例，然后才能为 AMD Vitis™ 工具添加新的块或生成平台。

注释：这些实现步骤是在 Vitis 环境流程中自动运行的。您可按本章中所述方法，使用 Vitis 命令行选项和配置文件来改善时序收敛和可实现的最高时钟频率。如需了解更多信息，请参阅《Vitis 统一软件平台文档》。

运行综合

综合步骤将采用 RTL 和时序约束，并生成功能上等同于 RTL 的最优化网表。通常，综合工具可以采用任何合规 RTL，并为其创建逻辑。综合需要现实的时序约束。

注释：如果您使用 Vivado IP integrator 创建设计并且已在 BD 文件中设置您的 IP，那么当您实现此 BD 文件时，Vivado 工具会自动运行综合。

如需了解有关综合的更多信息，请参阅以下资源：

- 《Vivado Design Suite 用户指南：综合》(UG901)
- Vivado Design Suite QuickTake 视频：设计流程概述

注释：如需了解有关时序约束的更多信息，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

相关信息

设计约束

综合流程

在 AMD Vivado™ Design Suite 中，可运行以下章节中所述的综合流程，这些流程各有各的利弊取舍。

全局综合

在全局综合流程中，整个设计通过单次运行来完成综合，其优势如下：

- 允许综合工具执行最大程度地最优化。由于综合工具已发现整个设计，该工具可在层级间进行最优化，这是其他流程无法做到的。
- 简化综合运行后的分析操作。

此流程的缺点在于编译时间较长。每次运行综合后，都会重新运行整个设计。但可通过使用增量综合来缓解此缺点的不利影响。

注释：如果设计包含 XDC 约束，那么必须将对象引用到顶层设计。

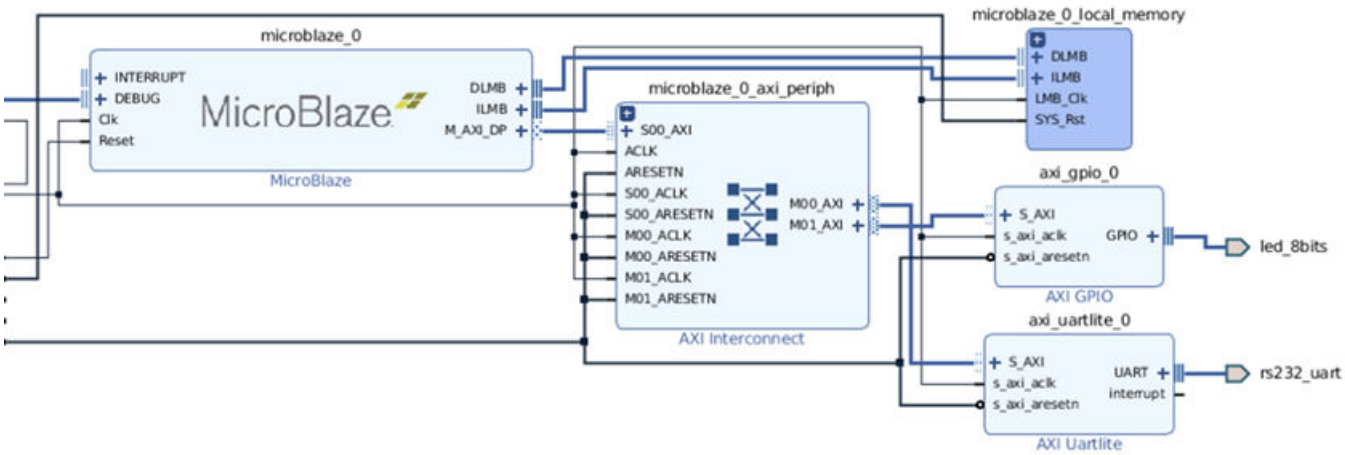
块设计综合

块设计综合流程支持您使用定制 IP 和 AMD IP 创建复杂系统。在此流程中，将使用 Vivado IP integrator 创建块设计 (BD) 文件。AMD IP 或定制 IP 将被添加到此 BD 文件中并作为系统进行连接。此流程具有以下优点：

- 将大量功能封装到小型设计中。
- 支持集中精力处理整个系统，而不是系统的各部分。
- 简化并加速设计的设置和综合。

下图显示了块设计示例。

图 115：**块设计示例**



创建块设计时，可使用非关联 (OOC) 综合模式或全局综合模式来运行综合。如果使用非关联综合模式，那么块设计将独立于设计其余部分进行单独综合。这样可在修改 BD 文件外部的层级时加速完成重新综合。如果使用全局综合模式，则每次都会编译并综合整个设计。全局综合模式更易于设置，因为它在全局级别设置约束。但使用该模式可能导致重新综合时运行时间更长。您可使用增量综合来缩短运行时间。

非关联综合

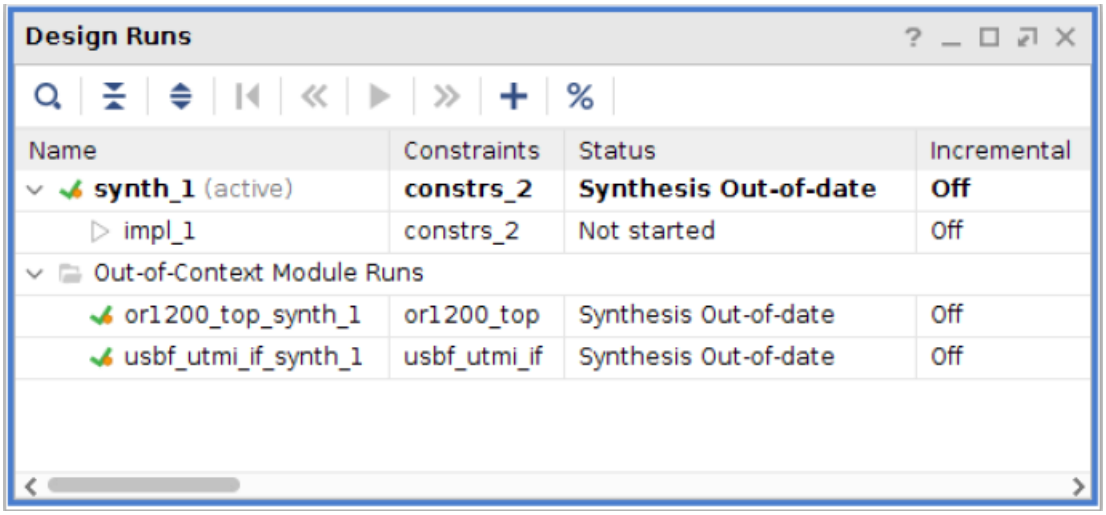
在非关联 (OOC) 综合流程中，某些层级与顶层分离，单独进行综合。非关联层级首先进行综合。然后，运行顶层综合，并且每次非关联运行都作为黑盒来处理。AMD IP 通常采用非关联综合模式来运行。所有非关联综合都运行完成并且顶层综合运行完成后，Vivado 工具会在您打开已综合的顶层设计时从所有综合运行执行设计汇编。此流程具有以下优点：

- 缩短后续综合运行的编译时间。仅对您指定的运行进行重新综合，其他运行保持不变。
- 确保进行设计更改时的稳定性。仅对包含更改的运行进行重新综合。

此流程的缺点在于需要额外进行设置。您必须谨慎选择将哪些模块设置为非关联综合模块。任何额外 XDC 约束都必须单独定义，并且仅限用于非关联综合运行。

下图所示设计包含 1 个顶层综合运行 (synth_1) 和 2 个低层非关联综合运行。

图 116：**含 2 个非关联综合运行的顶层综合**



如需了解有关设置非关联综合运行的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

增量综合

您可使用增量综合来复用现有综合结果。此方法能够将典型的综合编译时间缩短 50%。此方法配合增量实现流程一起使用时，可缩短总体编译时间并提升时序收敛一致性。

对设计进行综合时，会将其细分为多个 RTL 分区。增量综合会复用来自前一轮综合运行的 RTL 分区。RTL 分区通常是随逻辑层级一起创建的。仅当设计足够大、综合创建至少 4 个 RTL 分区并且每个分区包含至少 50000 个实例时，才会运行增量综合。这些实例包含逻辑层级和 RTL 原语。

以下是使用 `synth_design -incremental_mode <value>` 命令时可用的不同模式：

- `off`：不运行增量综合。
- `quick`：最快获得结果，但不执行跨边界最优化。此模式会限制逻辑的工作频率。
- `default`：启用大部分逻辑最优化，包括跨边界最优化。非增量综合，显著缩短编译时间。
- `aggressive`：启用所有最优化。非增量综合，显著缩短编译时间。

通常建议仅对低性能设计使用 `quick` 模式。由于无跨边界最优化，因此典型设计的时钟频率会降低。但如果精心构造设计且设计包含已寄存的分层边界，那么可能不会影响可实现的时钟频率。由于跨边界最优化受限，因此在给定区域内进行 RTL 更改才会导致在该区域内发生重新综合。在其他综合分区内执行的更改不会触发超出该分区范围的更改。由此导致增加复用，并能更快获得综合结果。对于其他模式，则情况并非如此，RTL 更改可能触发超出单元所在分区范围的其他分区发生再综合。当超出 50% 的分区发生修改时，将触发完全重新综合。

对于高性能设计，建议使用 `default`、`aggressive` 和 `off` 模式。在 `aggressive` 和 `off` 模式下会启用更多最优化，这可能导致进行更多再综合，但 QoR 会更高。

为了更加积极地应对编译时间问题，可将增量综合与 OOC 综合进行比对。OOC 综合通常供 IP 使用，并自动执行设置。含增量综合的全局综合所提供的优势不仅体现在允许跨边界最优化上，还体现在编译时间上。值得考量的领域包括：

- 编译时间

  OOC 综合更快，因为它减少了每次细化的代码量。仅当在 OOC 模块内修改 RTL 时，才会细化 RTL。

- Performance

  增量综合的最高时钟频率优先于 OOC 综合，因为此模式允许跨 OOC 边界执行最优化。

- 设置

  对于非 IP 流程，创建 OOC 综合运行时，必须在从更高的模块传递泛型/参数的同时创建封装文件。此外，还必须创建独立时序约束文件，并以 OOC 级别的端口作为目标。增量综合则不存在这些要求。

如需了解更多信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901)。

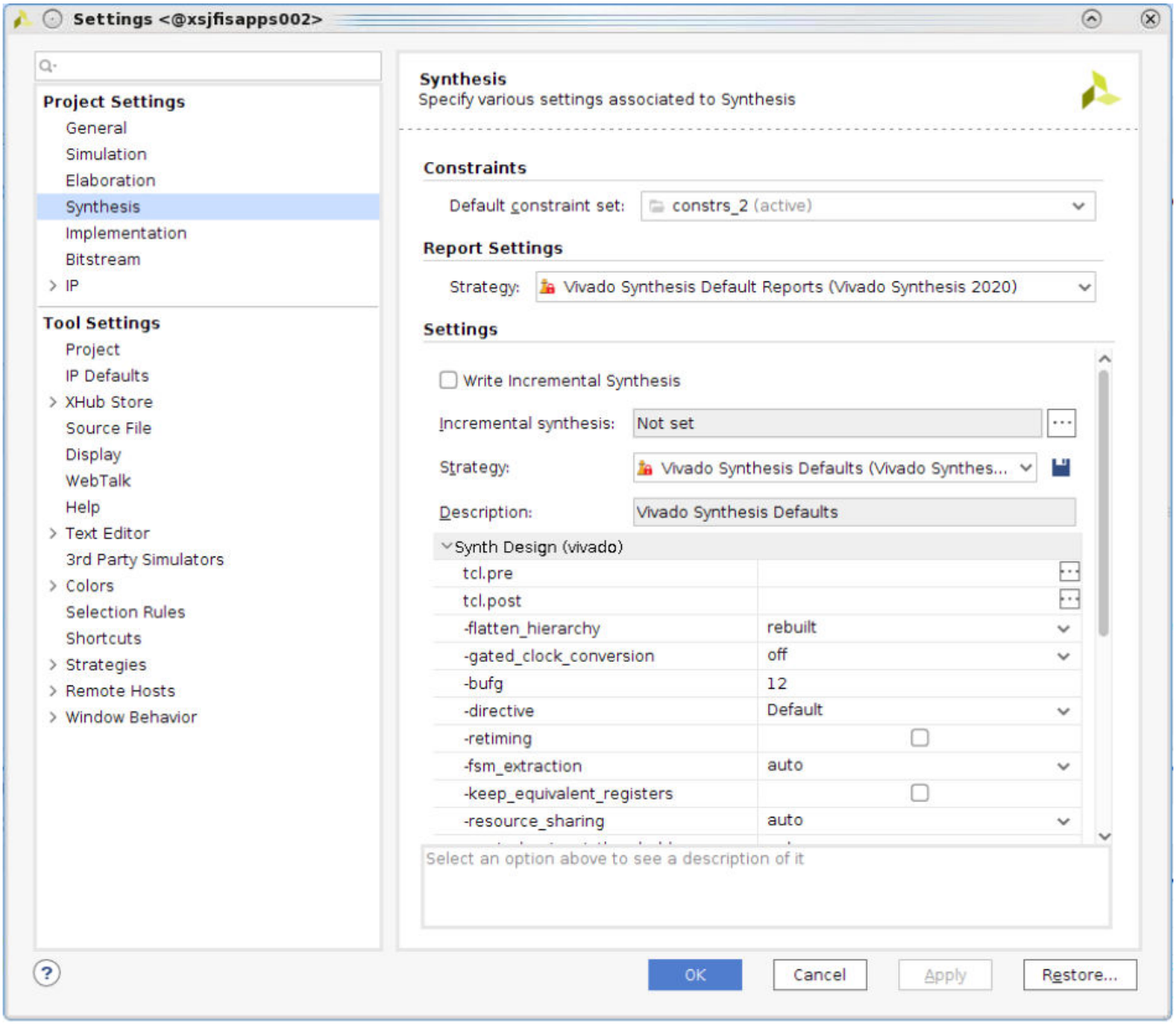
综合最优化

默认情况下，Vivado 综合所应用的最优化措施旨在为尽可能更多的设计产生最佳的结果。但您可以按下列章节中所述方式来调整默认综合最优化操作。

综合设置

您可使用 Vivado Design Suite 综合设置来选择多项影响整个设计的全局设置。这些设置将影响逻辑推断方式以及增量综合的运行方式。AMD 建议开始设计时使用默认选项，随后根据设计的具体需求来更改相应选项。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

图 117：**Vivado 综合设置**



综合属性

综合属性支持您以特定方式控制逻辑推断。虽然综合算法致力于为最大数量的设计提供最佳结果，但设计要求常常不尽相同。在此情况下，您可使用属性来更改设计以改进 QoR。如需了解有关综合所支持的属性的信息，请访问此链接以参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的相应内容。

注释：在为新器件调整设计之前，AMD 建议复查先前针对旧器件运行的设计中的所有综合属性。

使用 KEEP、DONT_TOUCH 和 MAX_FANOUT 属性时，请注意以下章节中所述的特殊注意事项。

KEEP 和 DONT_TOUCH

KEEP 和 DONT_TOUCH 均为非常有价值的设计调试属性。当对象被赋予这两种属性时，工具就不会最优化该对象。

- KEEP 供综合工具使用，并且不作为网表中的属性来进行传递。KEEP 可用于保留特定信号，例如，用于在综合期间关闭信号上的特定最优化。
- DONT_TOUCH 供综合工具使用，并可供传递到布局布线工具以确保对象永不进行最优化。

请谨慎使用这些属性：

- 接收 RAM 输出的寄存器上如果存在 KEEP 属性，就会导致该寄存器无法并入 RAM，从而阻止对块 RAM 进行推断。
- 在驱动更高层级上的三态输出或双向信号的层级上，请勿使用这些属性。如果驱动信号和三态条件都位于当前层级，那么将不会推断 IOBUF，因为工具必须更改层级才能创建 IOBUF。
- 禁用最优化的属性通常会导致电路增大以及功耗增大。AMD 建议有节制地使用这些控制，如不再需要，可将其移除。

此外，请注意将 DONT_TOUCH 放置在信号上与将其放置在层级上的差异：

- 如果将该属性放置在信号上，将保留该信号。
- 如果将该属性放置在层级上，该工具不会触碰该层级的边界，并且该层级中不会发生常数传输。但该层级内部的最优化将予以保留。

MAX_FANOUT

MAX_FANOUT 强制综合通过复制逻辑来满足扇出限制。该工具能够复制逻辑，但不能复制输入或黑盒。因此，如果在设计的直接输入所驱动的信号上放置 MAX_FANOUT 属性，那么该工具将无法处理约束。

请谨慎分析 MAX_FANOUT 所在的信号。如果将 MAX_FANOUT 布局在由含 DONT_TOUCH 的寄存器所驱动的信号上，或者驱动的信号所在层级与 DONT_TOUCH 属性所在层级不同，那么将无法遵循 MAX_FANOUT 属性进行操作。

综合在执行首次复制时会为复制的单元追加 `_rep`，为后续复制追加 `_rep__0`、`_rep__1`，以此类推。通过在单元上选择“Edit” → “Find”（编辑 > 查找），可以在综合后网表中看到这些单元。

重要提示！在综合期间，请谨慎使用 MAX_FANOUT。AMD Vivado™ 工具中的 `place_design` 和 `phys_opt_design` 命令可执行基于布局的复制，此操作比在综合内执行逻辑复制更有效。如果需要使用特定扇出，通常有必要花时间和精力来手动编码以生成额外寄存器。

块级综合策略

借助 Vivado 综合，您可使用各种策略和全局设置来自定义设计的综合方式。大多数情况下，这些选项均为全局选项并且影响整个设计。您可使用块级综合策略，通过自上而下的流程采用不同全局选项来对不同层级进行综合。相比于自下而上编译，此流程更快且更易于执行。您可以为整个设计设置约束，而不必先为较低层级设置约束然后再针对顶层进行重新设置。

在 XDC 文件中使用以下语法设置块级综合策略：

```
set_property BLOCK_SYNTH.<option_name> <value> [get_cells <instance_name>]
```

其中：

- `<option_name>` 为要设置的选项。
- `<value>` 为要分配给该选项的值。
- `<instance_name>` 为要在其中设置该选项的层级实例。

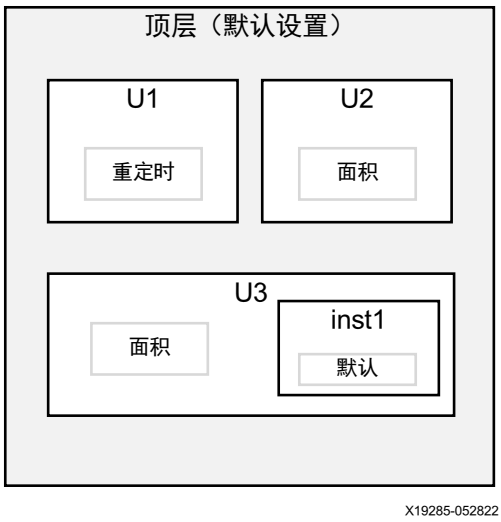
注释：这些属性始终在层级实例上进行设置。这样即可通过不同选项对已多次例化的模块或实体进行综合。

例如，您可以在 XDC 文件中设置以下策略：

```
set_property BLOCK_SYNTH.RETIMING 1                  [get_cells U1]
set_property BLOCK_SYNTH.STRATEGY {AREA_OPTIMIZED}  [get_cells U2]
set_property BLOCK_SYNTH.STRATEGY {AREA_OPTIMIZED}  [get_cells U3]
set_property BLOCK_SYNTH.STRATEGY {DEFAULT}          [get_cells U3/inst1]
```

Vivado 综合的执行方式如下图所示。

图 118：**块级综合策略示例**



为了尝试不同的选项，您可以在同一实例上设置多种 BLOCK_SYNTH 属性。例如：

```
set_property BLOCK_SYNTH.STRATEGY {ALTERNATE_ROUTABILITY} [get_cells inst]
set_property BLOCK_SYNTH.FSM_EXTRACTION {OFF} [get_cells inst]
```

当使用 IP 时，您可以使用下列块级综合策略：

- 如果 IP 采用全局编译，那么您可在 IP 顶层使用该策略。
- 对于非关联 IP，则不能使用该策略，因为此 IP 是 1 个黑盒。应改为在编译 IP 时使用全局设置。

注释：如需了解有关该功能以及受支持的策略和选项的更多信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901)。

综合后的步骤

确保综合过程中您已获得的网表质量优良，这样它就不会在下游造成问题。在继续执行实现流程的剩余步骤之前，应检查如下重要内容。

检查和清理 DRC

`report_drc` 命令可运行设计规则检查 (DRC) 以寻找常见设计问题和错误。需要进行多重规则检查。默认规则检查如下：

- 综合后网表
- I/O、BUFG 和其他特定的布局需求
- 属性和 MGT、IODELAY、MMCM、PLL 的连线以及其他原语

建议：设计进程中应尽早检查和纠正 DRC 违例，以避免在实现流程的后期出现时序或逻辑相关的问题。

提示：对于可安全忽略的 DRC 违例，您可以使用豁免机制来豁免此类违例。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

运行 Report Methodology

Vivado 工具中提供了“Report Methodology”（方法论报告），专用于检查是否符合方法论指南要求。这些工具根据所处的设计进程阶段运行不同的检查：

- RTL 设计：RTL lint 样式检查
- 综合设计和实现设计：网表、约束和时序检查

在“Project Mode”（工程模式）下，这些工具默认会在实现（`opt_design` 或 `route_design`）期间自动运行“Report Methodology”。要手动运行这些检查，请使用以下任一方法：

- 在 Tcl 提示符处，打开要确认的设计，并输入以下 Tcl 命令：`report_methodology`
- 要从 Vivado IDE 运行这些检查，请打开要确认的设计，然后选择“Reports” → “Report Methodology”（报告 > 方法论报告）。

建议：要识别常见的设计问题，请在首次对设计进行综合时运行此报告。添加重要模块、发生重大约束变更或者重大时钟电路变更后，再次运行此报告。

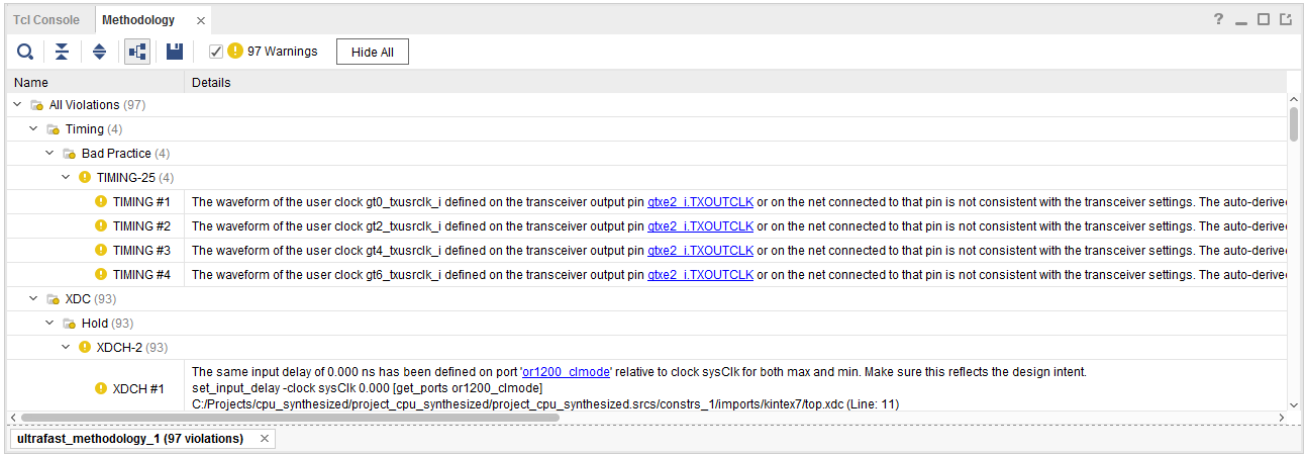
注释：对于 AMD 提供的 IP 核，已经对违例进行过评估与核查。

在“Methodology”（方法论）窗口中已列出所有违例情况，如下图所示。如果不需要修复特定方法论违例，请确保您明确了解此违例及其影响，以及此违例不会对您的设计产生负面影响的具体原因。

重要提示！您必须解决所有“Critical Warnings”（严重警告）和大部分“Warnings”（警告），以确保 QoR 结果良好、时序分析准确性高，并且满足硬件可靠性和稳定性要求。欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。对于可安全忽略的方法论检查违例，您可以使用豁免机制来豁免此类违例。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

注释：当原语的所有输入或输出路径的建立时序都大于 1 ns 时，将不会报告与 RAMB 和 DSP 原语可选流水打拍（SYNTH-6、SYNTH-11、SYNTH-12 和 SYNTH-13）相关的方法论检查。

图 119： **“Methodology”窗口**



如需了解有关运行 Report Methodology 的更多信息，请参阅《Vivado Design Suite 用户指南：系统级设计输入》(UG895)。另请访问此链接以参阅《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906) 中的相应内容。

复查综合 Log 日志

您必须复查综合 log 日志文件并确认该工具提供的所有消息与您期望的设计用途相匹配。请特别关注“Critical Warnings”（严重警告）和“Warnings”（警告）。大多数情况下，需修复“Critical Warnings”（严重警告）才能实现可靠的综合结果。

注意！如果某条消息出现超过 100 次，那么该工具仅将前 100 次写入综合 log 日志文件。您可通过 Tcl 命令 `set_param messaging.defaultLimit` 来更改 100 次的限值。

检查时序约束

您必须提供简单标准的时序约束以及时序例外（如果适用）。错误的约束将导致编译时间过长、出现最高时钟频率问题以及硬件故障。

建议：复查所有时序约束相关的“Critical Warnings”（严重警告）和“Warnings”（警告），这些警告表示尚未加载或者未能正确应用约束。

相关信息

对设计约束进行组织以便执行编译

评估综合后的结果质量

“Report QoR Assessment”（QoR 评估报告）可将逻辑层次检查、使用率检查和最常用的时钟拓扑结构检查组合到单一汇总报告中，以便为您提供总体设计评估。此报告可帮助您了解时序收敛问题的严重性。AMD 建议您在含正确时序约束的设计上完成任何重大网表更新后，在综合后的设计上运行此报告。

“Report QoR Assessment”可提供 1 到 5 之间的评分，用于表示设计实现时序收敛的可能性。下表显示了每个评分的定义。得分为 1 和 2 表示不可能满足时序收敛，得分为 3 表示满足时序收敛可能性较低。因此，低分意味着需要继续努力实现时序收敛。

表 11：**QoR 评估报告评分**

得分	含义
1	设计将可能无法完成实现。
2	设计将能够完成实现，但无法满足时序。
3	设计将可能无法满足时序。
4	设计将可能满足时序。
5	设计将能够满足时序。

在此报告中，详情表提供了有关评分的基本信息。详情表中的阈值对于器件而言并非绝对限制。这些阈值仅表示何时达成时序收敛的难度可能增加。超出其中任一项的阈值后，达成时序收敛的难度就会显著增大。

请制定相应计划以便纠正“Report QoR Assessment”中标记待查的任意项。其中许多项都可使用“Report QoR Suggestions”（QoR 建议报告）自动解决。

遵循准则解决剩余违例问题

重要提示！综合后，分析时序以识别重大设计问题，必须先解决这些问题，才能继续后续流程。

HDL 更改对 QoR 影响最大。因此，最好在实现前先解决问题，以便实现更快速的时序收敛。分析时序路径时，请特别关注以下问题：

- 最常见的错误（即错误最多的时序路径中出现的单元或信号线）
- 源自于未寄存的块 RAM 的路径
- 源自于 SRL 的路径
- 包含未寄存的级联 DSP 块的路径
- 含大量逻辑层次的路径
- 含大扇出的路径

注释：欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

处理高层次逻辑

识别长逻辑路径有助于诊断较难解决的 QoR 挑战。经过评估的综合后信号线延迟接近于最佳布局。要评估存在高层次逻辑延迟的路径是否满足时序要求，可生成无信号线延迟的时序报告。如果路径仍无法满足无信号线延迟的时序要求，那么在这些路径上就无法实现时序收敛。

注释：欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

检查使用率

分别检查 LUT、FF、块 RAM 和 DSP 组件的使用率至关重要。如果块 RAM 使用率较高，而 LUT/FF 使用率较低，那么设计仍可能出现布局问题。`report_utilization` 命令会生成全面的使用率报告，并分章节逐一罗列所有设计对象。

注释：在综合之后，由于设计流程后续执行的最优化，使用率可能发生改变。

复查时钟树

本节讨论了如何复查时钟树，包括时钟缓冲器使用率和时钟树拓扑结构。

时钟缓冲器的使用

`report_clock_utilization` 命令用于提供有关时钟原语使用率的详细信息。请观察架构时钟设置规则以避免出现下游布局问题。布局约束无效或者区域时钟缓冲器扇出过高可能导致布局器中出现问题。对于时钟缓冲器使用率过高的设计，可能需要锁定时钟生成器和部分区域时钟缓冲器以帮助完成布局。

对于需要极为严格的时序关系的部分接口，有时最好为需要极为严格的时序关系的信号（例如，源同步接口）锁定特定资源。总而言之，除非存在如上所述的特殊原因，否则作为设计的出发点，只需锁定 I/O 即可。

注释：欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

时钟树拓扑结构

请遵循以下建议来处理时钟树：

- 运行 `report_clock_networks` 命令以在详细的树视图中显示时钟网络。
- 按可最大限度降低偏差的方式来使用时钟树。
- 对于 PLL 和 MMCM 的输出，请使用相同类型的时钟缓冲器来最大限度降低偏差。
- 寻找可能意外引发额外延迟和/或偏差的级联 BUFG 元件。

实现设计

Vivado Design Suite 实现包括在器件资源上对网表进行布局布线，同时满足设计的逻辑、物理和时序约束所需的所有步骤。如需了解有关实现的更多信息，请参阅下列资源：

- 《Vivado Design Suite 用户指南：实现》(UG904)
- Vivado Design Suite QuickTake 视频：设计流程概述

使用工程模式对比使用非工程模式

您可使用工程模式或非工程模式来运行设计实现。工程模式可提供工程基础架构，如运行管理、文件集管理、报告生成和交叉探测。非工程模式可提供简单集成，并且由 Tcl 脚本驱动，此类脚本必须在整个流程中显式调用所需的全部报告。如需了解有关这些模式的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892) 中的相应内容。

策略

策略供 Vivado Design Suite 用于控制由“工程模式”下的综合和实现运行所生成的工具选项和报告。您可以使用相关策略对实现目标进行调整并控制所生成的报告。如需了解有关策略的更多信息，请参阅《Vivado Design Suite 用户指南：实现》(UG904)。

建议：请首先尝试默认策略 - Vivado 实现默认设置。该策略可在编译时间与设计的最高工作时钟频率之间实现良好的平衡。

注释：策略因工具和版本而异。在某些情况下，策略所需编译时间可能较长。

指令

指令为下列实现命令提供了不同行为模式：

- `opt_design`
- `place_design`
- `phys_opt_design`
- `route_design`

首先使用默认指令。当设计接近完成时，使用其他指令来浏览设计的解空间。每次只能指定一项指令。如需了解有关指令的更多信息，请参阅《Vivado Design Suite 用户指南：实现》(UG904)。

迭代流程

在非工程模式下，您可以使用不同选项通过各种最优化命令进行迭代。例如，可先运行 `phys_opt_design -directive AggressiveFanoutOpt`，再运行 `phys_opt_design -directive AlternateFlowWithRetiming`，以便对不满足时序要求的已布局的设计运行不同的物理综合最优化。

以迭代方式运行 `phys_opt_design` 可以改进时序约束。`phys_opt_design` 命令会尝试对顶层时序问题路径进行最优化。通过以迭代方式运行 `phys_opt_design`，可借助最优化来令更多关键路径获益。在布线后阶段中运行 `phys_opt_design` 会对可能未布线的任何信号线进行重新布线。因此，在布线后运行 `phys_opt_design` 后，无需再显式运行 `route_design`。

使用检查点分析不同阶段的设计

Vivado Design Suite 使用物理设计数据库来存储布局布线信息。设计检查点文件 (`.dcp`) 支持您在设计流程中的关键节点保存（`write_checkpoint` 命令）和复原（`read_checkpoint` 命令）此物理数据库。检查点是处于流程中特定节点的设计快照。在“工程模式”下，Vivado 工具会自动生成设计检查点文件，并将其存储在实现运行目录中。这些文件可在 Vivado 工具的单独实例中打开。

此设计检查点文件包含以下内容：

- 当前网表，包括实现期间执行的任何最优化
- 设计约束
- 实现结果

在设计流程的其余部分中，可使用 Tcl 命令来运行检查点设计。无法通过新设计源来对其进行修改。

以下是常见的检查点使用示例：

- 保存结果，以便您稍后返回并对这部分流程进行进一步分析。
- 尝试运行 `place_design`，使用多项指令并为每项指令保存检查点。这样即可支持您选择具有最佳时序结果的 `place_design` 检查点，以便用于执行后续实现步骤。

如需了解有关检查点的更多信息，请参阅《Vivado Design Suite 用户指南：实现》(UG904)。

使用交互报告文件

打开检查点后，可将生成的报告读取到 Vivado IDE 中，并在其中立即对报告进行分析。要生成报告，请使用以下报告命令并附加 `-rpx <filename.rpx>` 选项：

```
report_timing_summary
report_timing
report_power
report_methodology
report_drc
```

打开检查点之后，您可以使用“Reports” → “Open Interactive Report”（报告 > 打开交互报告）来打开交互式报告文件。

注释：在“工程模式”下，可以自动生成并打开交互报告。

建议：生成报告时，对 RPX 文件有大小限制。因此，AMD 建议使用 `catch` 命令来防止出现可能导致流程停止的错误。例如：`catch {report_timing_summary -rpx timing_summary.rpx -file timing_summary.rpt}`。

使用增量实现流程

在 Vivado Design Suite 中，您可以使用增量实现来复用现有布局和布线数据，从而缩短实现的编译时间，并提升结果的可预测性。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

建议：在设计周期的关键阶段中难以对流程脚本执行更改时，增量实现最为有用。确保您的流程脚本在设计周期初即包含增量实现，这样即可在关键时期启用增量实现。

注释：为进一步缩短编译时间和提升 QoR，还可使用增量综合。

相关信息

增量综合

自动增量实现模式

您可使用自动增量实现模式来激活增量实现流程，同时允许 Vivado 工具延迟运行增量实现直至获取有关参考检查点和当前设计的更多信息为止。当发出 `read_checkpoint` 命令时，Vivado 工具会判定使用默认流程算法还是使用增量流程算法来运行实现流程。自动模式提供了易于使用的按钮，因为工具负责管理参考设计数据以供用于增量实现。

注释：自动增量实现模式比运行默认增量实现流程更保守，支持在运行增量实现流程时为 QoR 提供更好的维护。

工程模式

在工程模式下，Vivado 工具可管理检查点的更新以及所使用的算法。要在工程模式下启用自动增量实现模式，请右键单击“Design Runs”（设计运行）窗口，然后选择“Set incremental Compile” → “Automatically use the checkpoint from the previous run”（设置增量编译 > 自动使用上一轮运行的检查点）。

等效的 Tcl 命令为：

```
set_property AUTO_INCREMENTAL_CHECKPOINT 1 [get_runs <runName>]
```

非工程模式

在非工程模式下，Vivado 工具可管理要使用的算法，但您必须判定是否要更新检查点。要在“非工程模式”下启用自动增量实现模式，请使用 `-auto_incremental` 选项。以下是一条示例命令：

```
read_checkpoint -incremental -auto_incremental <reference>.dcp
```

更新检查点时，请在实现流程末尾使用以下命令来避免 WNS 劣化至可接受的下限以下：

```
if {[get_property SLACK [get_timing_path -setup]] > -0.250} {
    file copy -force <postroute>.dcp <reference>.dcp
}
```

增量指令和目标 WNS

您可使用增量指令来指定实现流程的目标 WNS。目标 WNS 可判定实现工具是尝试收敛时序还是尝试实现与参考检查点相同等级的时序收敛。当实现流程使用默认算法时，将忽略增量指令并使用 `place_design` 和 `route_design` 指令。

下表显示了每条增量指令以及对应的目标 WNS 行为。

表 12：**增量指令与目标 WNS 行为**

增量指令	目标 WNS 行为
RuntimeOptimized	与参考检查点相同
TimingClosure	0.000
Quick	流程不受时序驱动，布局受相关逻辑驱动

注释：增量指令将取代先前版本的指令映射。

配置增量流程

您可以使用 `config_implementation` 命令配置增量流程。要查看此命令的默认值和当前值，请使用 `report_config_implementation` 命令。要更新这些值，请使用 `config_implementation` 命令。下面给出 1 个示例：

```
report_config_implementation
config_implementation { {incr.ignore_user_clock_uncertainty true} }
```

注释：您可采用上述外层括号中所示方法来对键值对进行分组，这样即可同时更新多个元素。

您可配置：

- 最小阈值，用于自动增量流程中的单元匹配、信号线匹配和 WNS。
- 综合与实现的行为，前提是不满足自动增量流程条件。在综合开始运行时以及在为实现执行 `read_checkpoint -incremental` 期间，会执行此项检查。它可设为 `Terminate` 以停止此流程，或者设为 `SwitchToDefaultFlow` 以退出增量流程，而以默认流程设置来继续运行。
- 流程是否忽略用户时钟不确定性约束，此类约束通常用于对布局器进行过约束，并强制执行更紧密的布局。

并行运行

为提升使用默认流程满足时序要求的可能性，常用的方法是实现多轮次的并行运行，每个轮次均采用不同的布局器指令。对于增量流程，该指令会指示需收敛时序还是维持时序。为实现各种不同结果，请为期望的增量指令设置不同的参考点目标。

编译时间注意事项

在自动增量实现模式下或高复用模式下使用 RuntimeOptimized 指令时，如果设计复用率不低于 95%，则编译时间可减半。随复用率下降，编译时间的优势也会一并下降。除非执行影响关键路径的更改，否则通常编译时间可预测。

在自动增量实现模式下或高复用模式下使用 TimingClosure 指令时，需耗费更多时间运行额外算法以达成时序收敛。使用此模式可能导致编译时间增加，当拥塞区域内难以达成时序收敛时或者无法满足时序要求时尤其如此。如果参考检查点可满足时序，那么缩短编译时间的效果与使用先前所述的 RuntimeOptimized 指令的效果类似。

打开综合设计

综合后的第一步是将网表从综合设计读取到存储器中并应用设计约束。您可以根据使用的流程通过各种方式打开综合设计。欲知详情，请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

逻辑最优化 (opt_design)

Vivado Design Suite 逻辑最优化可以对当前存储器中的网表进行最优化。由于这是设计汇编的第 1 个视图（RTL 和 IP 块），通常可以进一步对设计进行最优化。默认情况下，`opt_design` 命令会执行逻辑裁剪 (logic trimming)、移除不含负载的单元、传输常量输入和执行块 RAM 功耗最优化。此外，还可选择执行其他最优化操作，例如，重新映射，即将 LUT 串联组合，减少 LUT 从而降低路径深度。

最优化分析

`opt_design` 命令用于生成消息以详述每个最优化阶段的结果。完成最优化后，可运行 `report_utilization` 来分析使用率的提升情况。为了更好地分析最优化结果，请选中 `-verbose` 和 `-debug_log` 选项并重新运行 `opt_design`，以获取有关每次最优化对逻辑产生的影响以及用户约束阻止部分最优化操作的方式的完整详细信息。欲知详情，请访问此链接和此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

布局 (place_design)

Vivado Design Suite 布局器引擎将来自网表的单元置于目标 AMD 器件中的特定站点 (site) 上。

布局分析

布局后使用“Timing Summary”（时序汇总）报告检查关键路径。

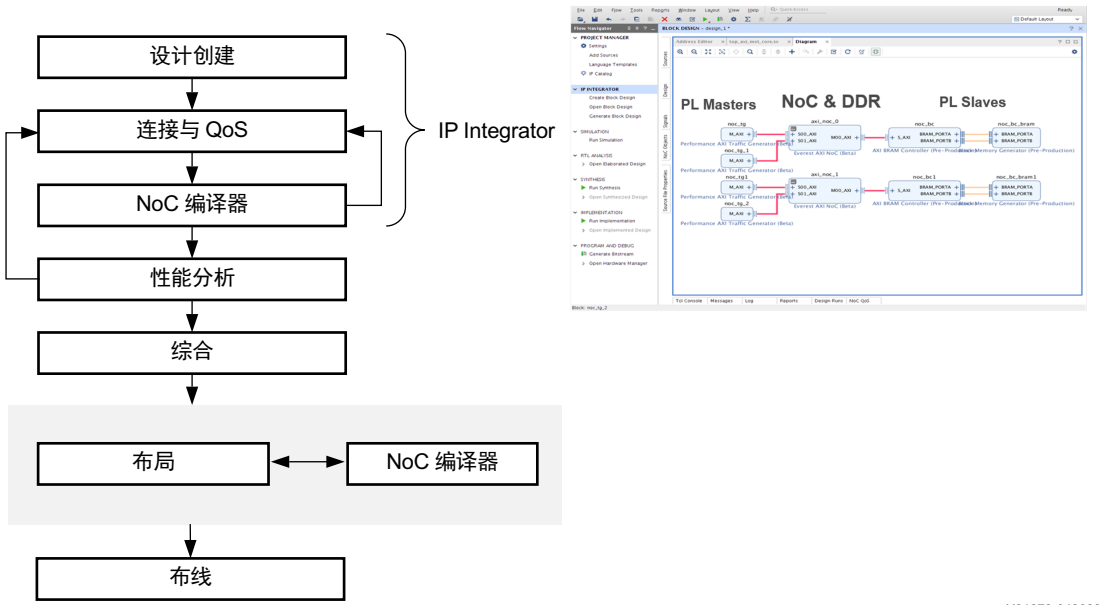
- 含超大建立时间时序负裕量的路径可能需要检查约束以确保完整性和正确性，或者逻辑重组以实现时序收敛。
- 对于含超大保持时间时序负裕量的路径，其主要成因是约束错误或时钟拓扑错误，因此需在进入布线设计之前将其进行修复。
- 具有较小的保持时间时序负裕量的路径有可能通过布线器修复。您还可先运行 `place_design`，然后再运行 `report_clock_utilization` 来查看按时钟区域划分时钟资源和负载计数的报告。

欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容，并获取有关布局的更多信息。

布局期间运行 NoC 编译器

Vivado IP integrator 会在块设计确认期间调用 NoC 编译器，以生成 NoC 布局布线解决方案，满足 QoS 要求。如果来自 IP integrator 的解决方案无法充分满足设计实现要求，那么在设计布局期间可调用 NoC 编译器来生成新的解决方案以满足实现要求。

图 120：**NoC 编译器流程**



下列实现要求可能导致在设计布局期间调用 NoC 编译器：

- 应用于 PL 的物理位置约束或 Pblock 约束将影响 NoC NMU/NSU 布局
- 解析 CIPS 与 NoC 之间的 NoC 接口，以便正确分配至目标器件
- DDR 存储器控制器接口的顶层端口分配将导致 DDR 存储器控制器分配发生更改
- 对可编程逻辑进行全局布局，这可能影响 NoC NMU/NSU 布局

提示：在 IP integrator 中，您可以将 DDR 存储器控制器的位置约束到 NoC 视图中的相应 site 位置，以便反映要在设计布局期间执行的分配。这样即可改善 IP integrator 与完全实现的设计之间的 NoC QoS 结果关联。

物理最优化 (phys_opt_design)

物理最优化属于流程中的可选步骤，其用于对设计的负时序裕量路径执行时序驱动的最优化。最优化内容涉及复制、重定时、修复保持时间以及布局提升。由于物理最优化会自动执行所有必要的网表和布局变更，因此在 `phys_opt_design` 之后不需要 `place_design`。

判断是否需要物理综合

为了判定物理综合对于设计是否有益，布局后请对时序进行评估。分析不满足要求的路径以确认扇出情况。高扇出的关键路径可受益于扇出最优化。此外，执行 `route_design` 后，如果涉及多个块 RAM 的大型 RAM 块的高扇出数据、地址和控制信号线未能满足时序要求，则同样可能受益于“Forced Net Replication”（强制信号线复制）。如需了解有关物理综合的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

布线 (route_design)

Vivado Design Suite 布线器在已布局的设计上执行布线，并在已布线的器件上执行最优化，以解决保持时间违例问题。默认情况下，布线器执行最优化的方式是在编译时间与设计的工作时钟频率之间实现平衡，同时缓解拥塞。部分布线器指令牺牲编译时间来换取更好的最高设计时钟频率，并积极减少拥塞。如需了解有关布线的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：实现》(UG904) 中的相应内容。

布线分析

时序约束错误通常会导致信号线的布线结果欠佳。在尝试布线器设置前，请确保已确认布线器所看到的约束和时序图。布线前，复查已完成布局的设计的时序报告，以确认时序和约束。

时序约束不良的常见示例包括跨时钟路径和错误的多周期路径，这些错误导致布线延迟插入，并需要修复保持时间。拥塞区域可通过在 RTL 综合中进行针对性的扇出最优化或通过物理最优化来解决。您可以保留全部或部分设计层级，以防止跨边界最优化并降低网表密度。或者，也可以使用布局规划约束来减轻拥塞。

欲知详情，请访问此链接以参阅《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388) 中的相应内容。

布线编译时间

您可以使用 `route_design -ultrathreads` 选项，以牺牲可重复性为代价缩短编译时间。该选项给予布线器额外的自由来执行多个线程，从而能够更快地完成布线，但每次的结果又略有不同。相同后续运行之间的裕量相差无几，但可节省大量编译时间。仅当环境不需要严格的可重复结果时，方可考虑采用该选项来缩短布线器编译时间。

第 11 章

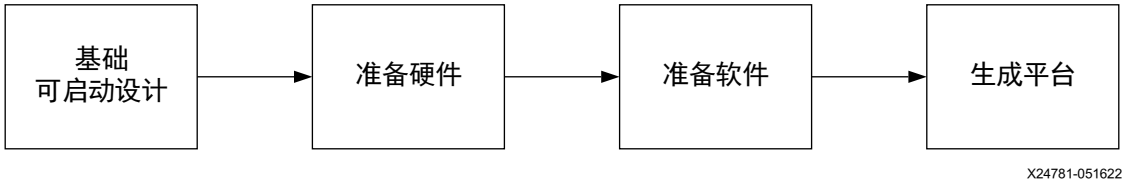
Vitis 环境嵌入式平台创建方法

在 AMD Vitis™ 环境的加速应用开发流程中，工程分为 2 个不同要素：平台和处理子系统。平台包含基本 IP 块（例如，用于 SoC 的 PS、NoC 和用于 AMD Versal™ 自适应 SoC 的 AI 引擎）以及开发板接口 IP 块（例如，高速 I/O 和存储器控制器）。处理子系统包含特定于应用的系统部分，这部分由可编程逻辑与 AI 引擎块组成。此方法有助于分离关注点、促进并发开发并鼓励复用。应用开发者能够免于应付平台的低层次细节，转而专注于解决处理子系统的细节问题。平台开发者则能够集中应对系统初始化和 I/O 性能调整，而无需担心处理子系统的问题。这意味着应用开发者能够在不同平台上集成子系统，并且可在不同处理子系统间复用平台。

AMD 为 AMD Alveo™ 卡和嵌入式评估板提供了预构建平台。您可从下载中心下载这些平台。此方法论的核心是将平台与子系统解绑并加以妥善利用，这同时也是 Vitis 环境所提供的生产力增益的核心。对于嵌入式设计，AMD 建议采用并行开发进程，即应用团队使用 AMD 预构建平台开始处理子系统，而平台团队则独立处理定制平台的初始化。以此方式开展工作即可快速取得进展。使用预构建平台意味着可使用经过预验证且已知有效的基本平台来对子系统进行独立开发、集成和测试。当子系统达到足够先进且稳定的状态后，该子系统即可与自定义平台的相应版本进行集成。总而言之，此方法能够显著精简系统集成进程。

下图显示了如何创建自定义的嵌入式平台。

图 121：**平台创建**



要创建平台，您必须具备可启动的基础设计作为起点。此设计可以是 AMD 基础平台设计、现有正常运作的设计或者也可以是从头开始创建的设计。在可启动的基础设计中，必须包含下列基础组件：

- 从 AMD Vivado™ Design Suite 导出的基础硬件设计
- 基础软件设计，其中包含 Linux 内核、根文件系统和设备树

通过 Vivado Design Suite 设计获得正常运行的硬件和开发板后，将其转换为 Vitis 环境平台需要为基础组件添加属性以满足 Vitis 环境要求。总体上，平台创建由以下步骤组成：

1. 在 Vivado Design Suite 工程中添加硬件接口参数和中断支持，并导出 XSA。
2. 更新软件平台组件以启用应用加速软件栈（启用 XRT、更新设备树等）。
3. 使用 XSCT 命令或 Vitis IDE 封装并生成平台。

注释：平台创建流程通常为迭代式，并且会在整个工程过程中创建多个版本的平台。在工程的早期阶段，您可以创建具有精简功能集的平台，以便于在开发板上测试处理子系统。在工程的后期阶段，您可能需要在平台上进行迭代，以响应规格更改或提高整体 QoR。

注释：您也可以基于现有存储库平台来创建平台。此操作实际上就是将存储库平台的副本复制到本地目录。但 GUI 或 XSCT 工具不支持 DFX 平台。

Vitis 环境使用硬件工程中的属性来识别平台中的资源，并将内核链接至平台。Vitis 环境使用软件栈来控制内核。

如需了解有关 Vitis 环境内的嵌入式平台创建的详细信息，请参阅 Vitis 统一软件平台文档。要获取分步指示信息，请参阅 Vitis 平台创建教程。

将功能映射到平台和子系统

使用 Vitis 环境时，在子系统和平台之间会对 PL 功能加以分割。在平台中必须包含必需的基础架构 IP（例如，CIPS、NoC 和外部 I/O 控制器）。但其他 PL 块的位置可由您自行选择。总之，AMD 强烈建议在子系统中包含系统处理链中所涉及的所有块和 IP。此方法能够显著提升设计模块化并启用更多自动化功能。在对紧密结合的 PL 和 AI 引擎块进行开发、调试和最优化时，此等卓越的灵活性尤为重要。欲知详情，请访问此链接以参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的相应内容。

附录 A

附加资源与法律声明

查找其他文档

文档门户

AMD 自适应计算文档门户是旨在使用您的网页浏览器提供健全的文档搜索和导航的在线工具。要访问文档门户，请转至 https://docs.xilinx.com。

注释：单击链接将打开英语版本，但您可从下拉列表中选择简体中文版本（如可用）。请注意，简体中文版本可能比英语版本旧。

Documentation Navigator

Documentation Navigator (DocNav) 是预安装的工具，支持访问 AMD 自适应计算文档、视频和支持资源，您可在其中通过筛选和搜索来查找信息。要打开 DocNav，请执行以下操作：

- 在 AMD Vivado™ IDE 中，单击“Help” → “Documentation and Tutorials”。
- 在 Windows 上，单击“Start”（开始）按钮并选中“Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

注释：如需了解有关 DocNav 的更多信息，请参阅《Documentation Navigator 用户指南》(UG968)。

注释：您无法从 DocNav 访问简体中文版本。请使用设计中心网页。

设计中心 (Design Hub)

AMD 设计中心提供了根据设计任务和其他主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

- 在 DocNav 中，单击“Design Hubs View”选项卡。
- 转至设计中心网页。

支持资源

如需获取答复记录、技术文档、下载以及论坛等支持资源，请访问技术支持。

参考资料

以下技术文档是非常实用的补充资料，可配合本指南一起使用：

1. 《Versal 自适应 SoC GTY 和 GTYP 收发器架构手册》(AM002)
2. 《Versal 自适应 SoC 时钟资源架构手册》(AM003)
3. 《Versal 自适应 SoC 可配置逻辑块架构手册》(AM005)
4. 《Versal 自适应 SoC SelectIO 资源架构手册》(AM010)
5. 《Versal 自适应 SoC 技术参考手册》(AM011)
6. 《Versal 自适应 SoC 封装和管脚分配架构手册》(AM013)
7. 《Versal 自适应 SoC CPM CCIX 架构手册》(AM016)
8. 《Versal 自适应 SoC GTM 收发器架构手册》(AM017)
9. 《SmartConnect LogiCORE IP 产品指南》(PG247)
10. 《Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP 产品指南》(PG313)
11. 《Advanced I/O Wizard LogiCORE IP 产品指南》(PG320)
12. 《适用于 Versal 自适应 SoC 的 Clocking Wizard LogiCORE IP 产品指南》(PG321)
13. 《Versal Adaptive SoC Transceivers Wizard LogiCORE IP 产品指南》(PG331)
14. 《Versal Adaptive SoC Integrated Block for PCI Express LogiCORE IP 产品指南》(PG343)
15. 《Versal Adaptive SoC DMA and Bridge Subsystem for PCI Express 产品指南》(PG344)
16. 《Versal Adaptive SoC PCIe PHY LogiCORE IP 产品指南》(PG345)
17. 《Versal Adaptive SoC CPM Mode for PCI Express 产品指南》(PG346)
18. 《Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express 产品指南》(PG347)
19. 《Control, Interface and Processing System LogiCORE IP 产品指南》(PG352)
20. 《Vivado Design Suite Tcl 命令参考指南》(UG835)
21. 《Versal 自适应 SoC PCB 设计用户指南》(UG863)
22. 《Vivado Design Suite 用户指南：设计流程概述》(UG892)
23. 《Vivado Design Suite 用户指南：使用 Tcl 脚本》(UG894)
24. 《Vivado Design Suite 用户指南：系统级设计输入》(UG895)
25. 《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896)
26. 《Vivado Design Suite 用户指南：I/O 管脚分配和时钟规划》(UG899)
27. 《Vivado Design Suite 用户指南：逻辑仿真》(UG900)
28. 《Vivado Design Suite 用户指南：综合》(UG901)
29. 《Vivado Design Suite 用户指南：使用约束》(UG903)
30. 《Vivado Design Suite 用户指南：实现》(UG904)
31. 《Vivado Design Suite 用户指南：设计分析与收敛技巧》(UG906)

32. 《Vivado Design Suite 用户指南：功耗分析与最优化》(UG907)
33. 《Vivado Design Suite 用户指南：编程和调试》(UG908)
34. 《Vivado Design Suite 用户指南：Dynamic Function eXchange》(UG909)
35. 《Vivado Design Suite 属性参考指南》(UG912)
36. 《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994)
37. 《AI 引擎工具和流程用户指南》(UG1076)
38. 《AI 引擎内核与计算图编程指南》(UG1079)
39. 《Vivado Design Suite 用户指南：创建和封装定制 IP》(UG1118)
40. 《Versal 自适应 SoC 设计指南》(UG1273)
41. 《Versal 自适应 SoC 系统集成和确认方法指南》(UG1388)
42. 《Vitis 统一软件平台文档：应用加速开发》(UG1393)
43. 《Vitis 高层次综合用户指南》(UG1399)
44. 《Vitis 统一软件平台文档：嵌入式软件开发》(UG1400)
45. 《Versal 自适应 SoC 系统和解决方案规划方法指南》(UG1504)
46. 《Versal 自适应 SoC 开发板系统设计方法指南》(UG1506)

修订历史

下表列出了本文档的修订历史。

章节	修订综述
2023 年 11 月 15 日 2023.2 版	
系统设计类型	在表格中添加 AIE-ML 链接。
Dynamic Function eXchange 的设计规划注意事项	更新描述。
基于 DFX 的 Vitis 加速平台开发的设计规划注意事项	更新第 5 项最小实现描述。
适用于 AI 引擎核可编程逻辑集成的设计规划注意事项	新增章节。
块 RAM 的性能/功耗利弊取舍	更新图示描述。
Versal 器件时钟设置	更新图示，并添加时钟布线注释。
多时钟缓冲器 (MBUFG)	更新代码块。
使用 CLOCK_ROUTE_GUIDE 约束	更新 CLOCK_ROUTE_GUIDE 描述。
使用 IMUX 寄存器约束	新增章节。
Dynamic Function eXchange 的布局规划约束	添加时钟设置描述。
使用增量实现流程	移除有关运行时间减半的描述。
2023 年 5 月 24 日 2023.1 版	
文档标题	标题更改为《Versal 自适应 SoC 硬件、IP 和平台开发方法指南》(UG1387)。
系统设计类型	添加 HBM。
传统设计流程的设计规划注意事项	添加设计流程图。

章节	修订综述
基于平台的设计流程的设计规划注意事项	添加设计流程图。
Dynamic Function eXchange 的设计规划注意事项	添加设计流程图和基于 BDC 的 DFX 设计描述。
基于 DFX 的 Vitis 加速平台开发的设计规划注意事项	在平台设置中添加 DFX 和非 DFX 嵌入式平台创建描述。
有关采用 Versal 器件 IP 进行设计的建议	添加 AM017 参考资料。
针对不同 Versal 器件设计拓扑结构的建议	在 BD 设计中添加 CIPS 和 NoC 描述。
第 4 章：使用 RTL 创建设计	添加 UG899 链接和描述。
时钟原语	添加 AM017 参考资料。
增大 Fmax 的准则	添加章节。
低扇出时钟	添加 BUFG_GT 描述。
使用 CLOCK_ROUTE_GUIDE 约束	添加章节。
USER_CLOCK_ROOT 分配	添加 USER_CLOCK_ROOT 分类。
同步 CDC	添加 CDC 路径输入/输出时钟注释。
NoC 注意事项	添加 SSI 技术描述。
第 8 章：采用 HBM 器件进行设计	添加章节。
增量综合	更新至 50000 个实例。
参考资料	添加 AM017 参考资料。

请阅读：重要法律声明