UltraFAST™

UltraFast Vivado HLS 方法指南

UG1197 (v2020.1) 2020 年 6 月 3 日

XILINX®

修订历史

下表列出了本文档的修订历史。

章节	修订综述
2020 年 6 月 3 日 2020.1 版	
核准用户指南标题变更	原标题：UltraFast 高效设计方法指南 现标题：UltraFast Vivado HLS 方法指南

XILINX®

目录

XILINX

第 1 章

高效设计方法指南

关于本指南

赛灵思可编程器件含有数百万个逻辑单元 (LC)，并且集成的现代复杂电子系统也与日俱增。本高效设计方法指南提供了一整套最佳做法，旨在于较短的设计周期内完成此类复杂系统的创建。

本方法指南主要围绕下列概念展开：

- 使用并行开发流程来提供有价值的差分逻辑，使您的产品在市场中脱颖而出，并提供 shell 用于将此类差分逻辑与生态系统其余部分有机整合。
- 广泛使用基于 C 语言的 IP 开发流程实现差分逻辑，使仿真速度较 RTL 仿真成倍增长，并提供时序精确且经过最优化的 RTL。
- 使用现有预验证的块级和组件级 IP 来快速构建 shell，将差分逻辑封装到系统中。
- 使用脚本来实现从设计精确性验证到 FPGA 编程在内整个流程的高度自动化。

本指南中的建议是根据多年来广泛收集的专家级用户经验总结而成的。与传统 RTL 设计方法相比，这些建议持续不断实现了各方面提升，包括：

- 设计开发时间加快 4 倍。
- 衍生设计开发时间加快 10 倍。
- 结果质量 (QoR) 提高 0.7 倍到 1.2 倍。

虽然本指南以大型复杂设计为重点，但其中论述的做法也同样适用于并已成功应用到下列各类设计中：

- 数字信号处理：
  - 图像处理
  - 视频
  - 雷达
- 汽车
- 处理器加速
- 无线
- 存储
- 控制系统

对新设计方法的需求

当今日益复杂的电子产品中所使用的先进设计正在不断对器件密度、性能和功耗的极限发起挑战，同时也对设计团队提出了挑战，要求他们在限定的预算内按时完成设计目标。

应对这些设计挑战的高效方法之一是将更多时间投入到更高的抽象层，这样即可最大程度缩短验证时间和提升工作效率。

对新设计方法的需求在下图中得到了充分体现，其中每个区域的面积分别代表设计流程中每个阶段的开发工作量的比例。

- 对传统 RTL 方法而言，大部分工作主要耗费在实现的细节工作上。
- 在高效设计方法中，大部分工作主要集中于设计系统和验证构建的系统是否正确。

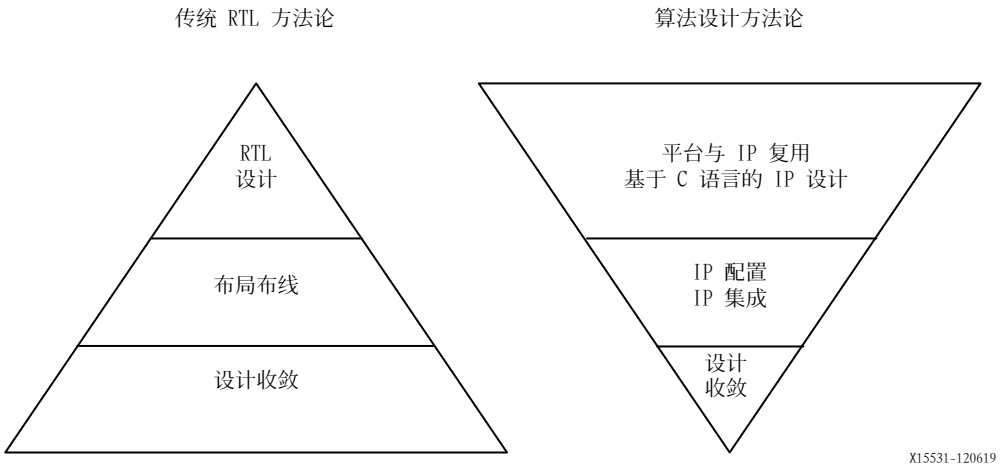

图 1-1：高效设计方法比较

传统方法

传统设计开发首先是由有经验的设计人员预估如何使用新技术来实现自己的设计、完成寄存器传输级 (RTL) 的设计捕获、通过综合和布局布线执行一些尝试以确认自己的预估是否正确，然后继续开展设计其余部分的捕获工作。在此过程中通常需要逐一对每个块进行综合，以重复确认设计的实现细节是否可接受。

确认设计能否提供所需功能的主要方法是对 RTL 进行仿真。由于 RTL 可提供极为详细的位精度和周期精度描述，虽然这些描述精度极高，但此流程仍较为缓慢且易于出错。

仅当在 RTL 中捕获设计中的所有块之后才能执行完整的系统验证，这往往会导致对 RTL 进行反复调整。在系统中的全部块验证完毕后，即可对其进行集中布局布线，此时才能完全确认先前预估的时序和面积的精确性，或者发现其中不精确的地方。这也往往会导致对 RTL 进行更改、重新启动系统验证以及重新进行实现。

设计人员现在通常需要在给定工程中实现数十万行 RTL 代码，把大部分设计时间都用在实现的细节工作上。如图 1-1 所示，设计人员把更多时间用在设计的实现上，而不是设计出使所有产品保持竞争力所需的新颖创新的解决方案。

无论是采用更新的技术以提升性能，还是采用更缓慢的技术以提供更具竞争力的定价，都意味着必须重写大部分 RTL，并且设计人员必须重新实现寄存器间的大量逻辑。

高效设计方法指南

高效设计方法沿袭了传统 RTL 方法的基本步骤，如图 1-1 所示。但是，它能够让设计人员把更多时间用来设计增值解决方案。高效设计方法的主要特性包括：

- 提出了随差分逻辑并行开发并验证的 shell 概念。此 shell 包含差分逻辑，用于捕获独立设计工程中的 I/O 外设和接口
- 使用基于 C 语言的 IP 仿真，使仿真时间相比于传统 RTL 仿真缩短多个数量级，为设计人员提供了设计理想解决方案的时间。
- 借助赛灵思 Vivado® Design Suite 利用基于 C 语言的 IP 开发、IP 复用和标准接口，实现时序收敛的高度自动化。
    - 使用 Vivado IP 目录轻松复用您自己的块级和组件级 IP，还能轻松获取赛灵思 IP，这些 IP 均已验证且已知在技术中能够有效实现。

高效设计方法中的所有步骤都能交互执行，也可使用命令行脚本来执行。所有人工交互的结果均可保存到脚本，实现从设计仿真直至 FPGA 编程的整个流程的完全自动化。根据您的设计和 RTL 系统级仿真的运行时间，该流程通常在任何 RTL 设计仿真完成之前即可在开发板上生成 FPGA 比特流并对设计进行测试。

创建衍生设计时，效率提升将更为明显。基于 C 语言的 IP 能够与不同器件、技术和时钟速度轻松对应，就像更改工具选项一样简单。完全脚本化的流程与通过 C 语言综合实现的自动时序收敛意味着能够快速完成衍生设计的验证和组装。

设计进程

下图显示了设计进程的各个步骤。

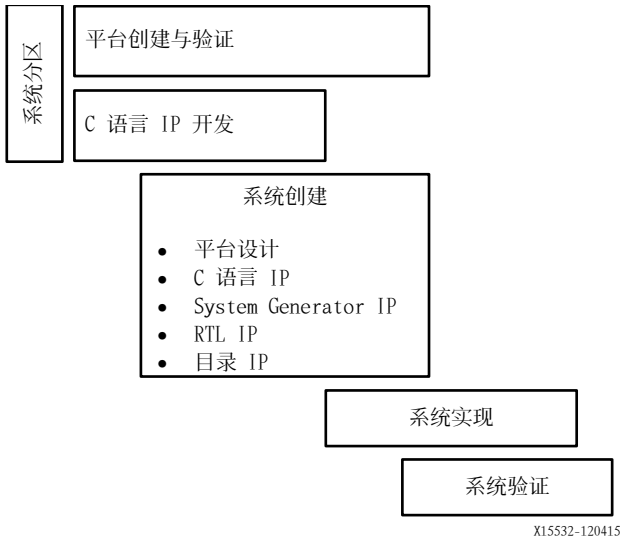

图 1-2：高效设计流程

完成第 2 章"系统设计"中所述的系统分区初始阶段之后，此设计流程的关键功能之一即开发的可重叠性。

- **Shell 开发流程：**通过使用 Vivado IP integrator 和 IP 目录，Vivado Design Suite 支持快速高效的块级集成。大部分关键的系统性能方面的工作 （包括细致的接口创建、验证和管脚分配）都可采用并行开发工作方式，并由开发者给予其所需的关注。第 3 章"shell 开发" 中对此流程进行了描述。
- **基于 C 语言的 IP 开发：**使用 RTL 仿真对完整的 1 帧视频进行仿真需要大约 1 到 2 天 （取决于设计、主机等）。使用 C/C++ 语言执行同样位级精度的仿真只需大约 10 秒钟。基于 C 语言的开发流程带来的效率提升不容忽视。如需了解有关此流程的详细描述，请参阅第 4 章"基于 C 语言的 IP 开发"。
- **系统创建：**Vivado IP integrator 和 IP 目录支持使用 shell 设计、传统 RTL IP、 System Generator IP 和赛灵思 IP 将基于 C 语言的 IP 快速组合到系统块设计中。自动化接口连接功能和系统创建的脚本化功能意味着在整个 IP 开发流程中，系统可实现快速生成和快速重新生成。如需了解有关此流程的详细描述，请参阅第 5 章"系统集成"。
- **系统实现：**通过使用经过验证的 shell 设计、基于 C 语言的 IP （针对器件和时钟频率经过自动最优化）以及现有经过验证的 IP，并采用符合业界标准的 Arm AMBA® AXI4 协议标准的接口将这些设计和 IP 全部连接在一起，即可确保最大程度缩短用于设计收敛的时间。只需单击几次鼠标或使用脚本化流程，就可以从系统块设计启动此流程。如需了解有关此流程的详细描述，请参阅第 5 章"系统集成"。
- **系统验证：**使用门级精度的 RTL 仿真和/或在开发板上进行 FPGA 编程和设计验证来执行此操作。由于使用 RTL 仿真进行系统验证，而不是在开发过程中使用迭代仿真来验证设计，因此在设计流程末只需执行一次仿真即可。如需了解有关此流程的详细描述，请参阅第 5 章"系统集成"。

访问技术文档和培训资料

在适当的时间获得正确的信息，对于按时完成设计收敛并确保整体设计成功至关重要。参考指南、用户指南、教程和视频能够帮助您尽快掌握 Vivado Design Suite。本节为您列出了技术文档和培训资料的部分来源。

使用 Documentation Navigator

Vivado Design Suite 配套提供赛灵思 Documentation Navigator （图 1-3），供您访问和管理全套赛灵思软硬件文档、培训资料和技术支持资料。借助 Documentation Navigator，您可查看赛灵思最新及过去的技术文档。您可根据版本、文档类型或设计任务来过滤其中显示的技术文档。结合搜索功能可帮助您快速找到正确的信息。 “Methodology Guides”是“Document Types”下的过滤器之一，有了它，您几乎可以瞬间找到任何方法指南。

赛灵思使用 Documentation Navigator 的“更新目录 (Update Catalog)”功能来为您提供最新的技术文档。该功能可提醒您有可用的目录更新内容，并提供有关所涉及的文档的详细信息。赛灵思建议您在出现提醒时随时更新目录，以使其保持最新状态。此外，您可以为指定的文档建立本地技术文档目录并对其进行管理。

Documentation Navigator 中包含“Design Hub View”选项卡。 “设计中心 (Design Hub)”是指与设计活动 （如应用设计约束、综合、实现、编程以及调试等）相关的文档集合。文档和视频被分门别类纳入每个设计中心内，以简化相关领域的学习过程。每个设计中心均包含“入门 (Getting Started)”部分、 “技术支持资料 (Support Resources)”部分 （包含对应流程的 FAQ）以及“其它学习资料 (Additional Learning Material)”部分。 “Getting Started”部分可为新用户提供清晰的入门指导。对已熟悉流程的用户，“关键概念 (Key Concept)”和“FAQ”部分能够为其提供 Vivado Design Suite 相关专业知识以满足其特定需求。

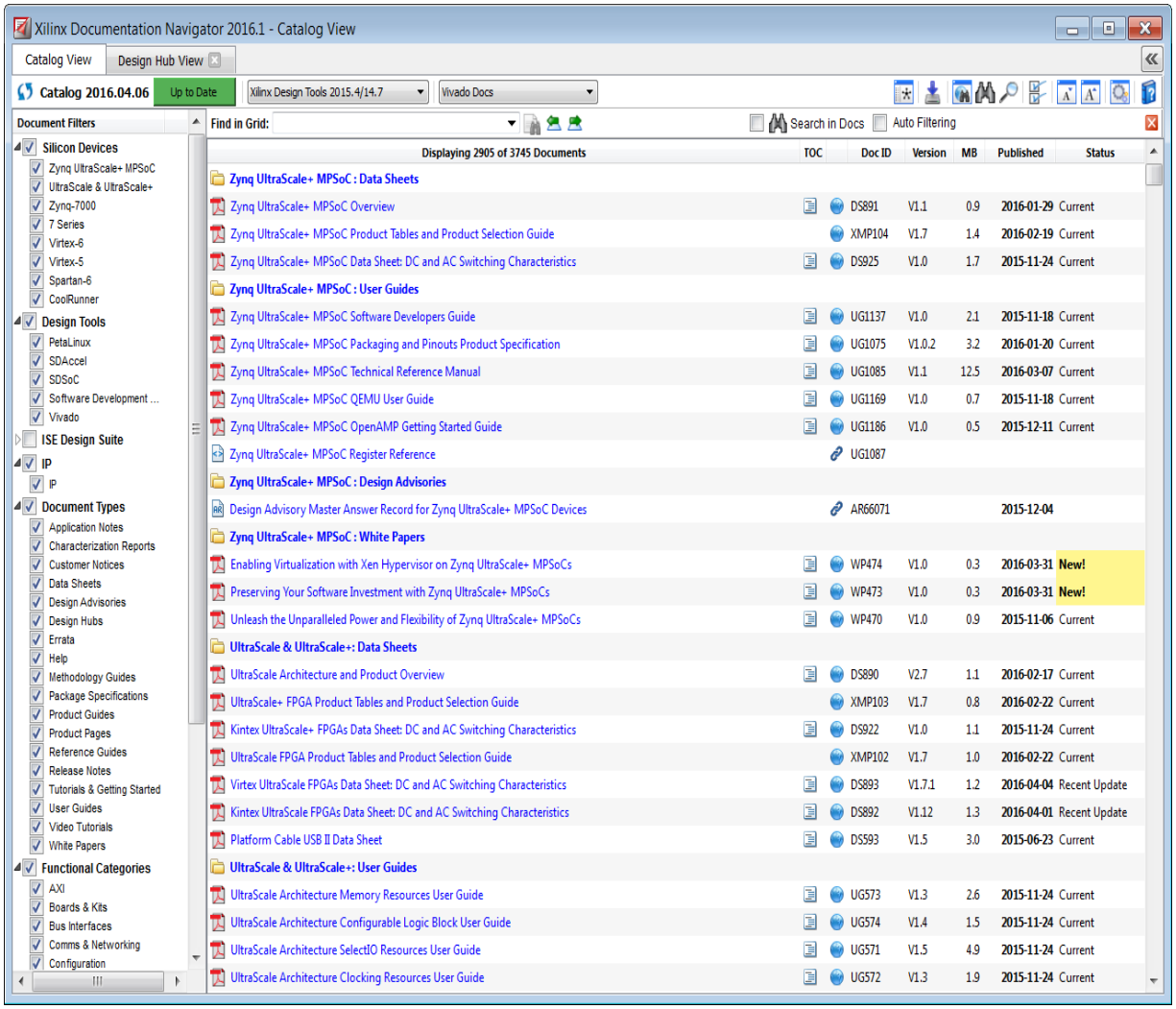

图 1-3：赛灵思 Documentation Navigator

XILINX

第 2 章

系统设计

简介

在开始工程之前，重要的是明确系统的设计和组装方法。在任何复杂的系统中，找到解决方案的途径都不唯一。这些路径由您的选择而定，包括从头开始创建的 IP 块、可复用的 IP 块、用于验证 IP 的工具和方法、系统中集成的 IP 以及系统验证方法等。

本章主要探讨可选系统分区方法，并详解 Vivado® Design Suite 中有助于系统开发流程自动化的关键特性。

系统分区

在典型设计中，位于设计边缘处的逻辑专用于与外部器件连接，一般此连接使用的是标准接口。相关示例包括 DDR、千兆位以太网、PCIe、HDMI、ADC/DAC 和 Aurora 接口。对同一家公司内的多种 FPGA 设计而言，这些接口以及用于实现它们的组件一般都是标准化接口和组件。

在高效设计方法中，此逻辑与差分逻辑彼此独立，并且此逻辑被视为 shell。下图所示即 shell 块设计示例。下图中心的阴影部分表示可添加差分逻辑或 shell 验证 IP 的区域。

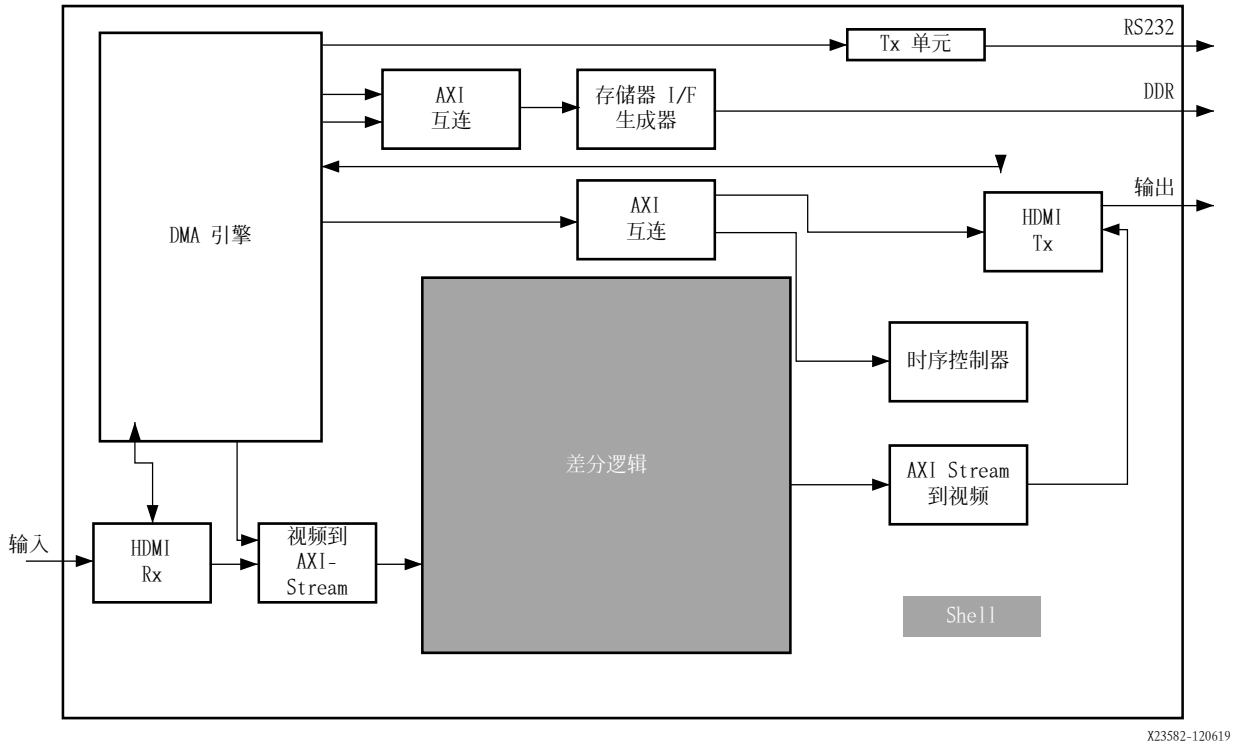

图 2-1：shell 设计示例

这种方法的主要优势有：

- shell 的开发和验证独立于设计的其余部分。
- 开发板级集成和器件管脚分配由并行开展工作的独立专门团队负责处理。
- shell 可保存并重复使用，甚至可重新编辑，便于迅速实现多种衍生设计。
- 差分逻辑的开发和验证独立于 shell。
- 预验证的 shell 和差分逻辑可迅速集成到完整系统中。

进行系统分区时，首要任务是判断在 shell 中要实现的目标以及要作为差分逻辑来实现的对象。

shell 设计

shell 设计能够为高效方法提供 2 大关键特性：

- 将标准接口逻辑与差分逻辑分离，使两者的开发和验证工作能够并行进行。
- 创建可复用的设计和 shell，可用于快速创建衍生设计。理想情况下 shell 应包含设计的标准部分，例如设计接口和接口 IP。不过 shell 也可以包含用于预处理或后处理的块。如果处理功能独立于核设计 IP 且处理功能可在多种设计中使用，那么这些块更适合布局在 shell 内。这种 shell 复用方法可便于从 shell 中移除这些块。

shell 设计的关键特性在于连接到内部设计 IP 的内部接口应使用标准接口来实现，无论您决定将何种逻辑整合到 shell 设计中都是如此。使用 AXI 等标准内部接口能增强 shell 可复用性，因为此类接口：

- 便于 shell 与尚待开发的设计 IP 相连
- 确保在验证 shell 的同时也能对内部接口进行验证
- 支持使用 “IP integrator 与标准接口” 中介绍的高效集成功能

即便您最初只考虑一个设计，基于 shell 的方法仍支持在初始设计实现后轻松创建衍生设计。

如需获取关于 shell 开发和验证的详细说明，请参阅第 3 章"shell 开发"。

IP 设计

IP 开发流程的主要特性在于它只包含用于区分产品和 shell 的 IP。

该设计 IP 并非标准 IP，需要开发。大部分开发工作主要集中在运行仿真来验证设计能否提供正确的功能。通过排除不影响所开发的新功能的标准块，即可最大程度降低此工作量并缩短仿真运行时间；这些标准块应包含在 shell 内。

下图展示了完整系统的演示，其中已将设计 IP 添加到 shell 设计中。完整系统的关键特性之一在于它可包含从不同来源开发的 IP，例如：

- 使用 Vivado HLS 从 C/C++ 生成的 IP
- 从 System Generator 生成的 IP
- 从 RTL 生成的 IP
- 赛灵思 IP
- 第三方 IP

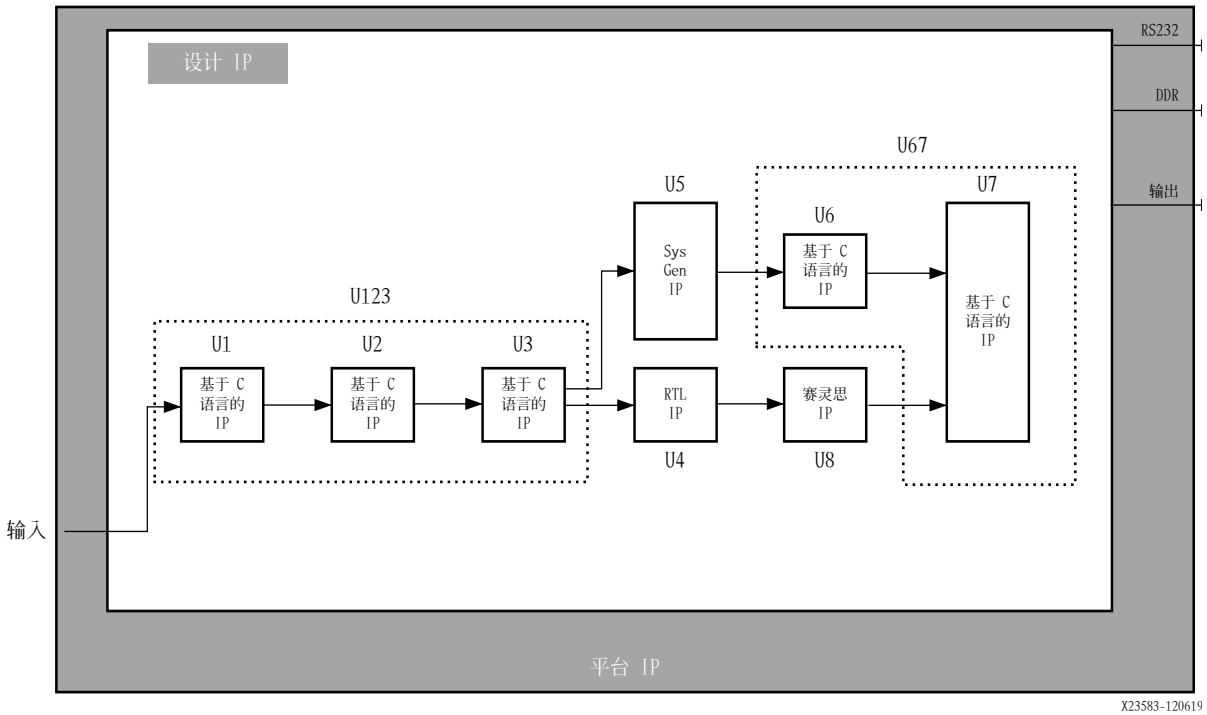

图 2-2：系统设计示例

在高效设计方法中，最显著的优势之一来自于 C 语言仿真的验证速度。从设计创建的角度来看，通过在开发过程中集中仿真 C 语言块即可显著提升效率。

- 高速 C 语言仿真便于设计人员迅速开发和验证高精度解决方案。
- 同时仿真多个 C 语言块有助于彼此验证各自的输出。
- 如果把多个 C 语言 IP 结合到单一 C 语言仿真中，则能显著提升总体效率。

图 2-2 突出表现了您在使用 C 语言 IP 时可能遇到的两难局面。块 U1、U2 和 U3 均为 C 语言 IP，可将其组合为单一顶层 U123。同样，块 U6 和 U7 可组合成单一 IP 块 U67。您可以选择下列两种方法之一：

- 创建多个较小的 C 语言 IP 块，例如 U1、 U2、 U3、 U6 和 U7。
- 创建几个大型 C 语言 IP 块，例如上图中列出的 U123 和 U67。

从设计集成的角度出发，这两种方法之间并无区别；如果 IP 块是使用 AXI 接口生成的，那么使用 IP integrator 即可将其轻松集成在一起。对刚刚接触基于 C 语言的 IP 开发的设计人员来说，较为理智的做法可能是使用较小块开展工作、学习如何对每个小块进行单独最优化、然后把多个小型 IP 块集成到一起。对已经熟悉 C 语言 IP 开发的设计人员而言，更合理的做法是生成几个大型 C 语言 IP 块。

重要提示：由此给效率带来的主要优势在于在开发期间通过一次 C 语言仿真即可对尽可能多的 C 语言 IP 块进行仿真。

在上述情况下，用于验证块 U1、 U2 和 U3 的 C 语言测试激励文件将同样用于验证 U123。 IP 生成的区别在于您将在 Vivado HLS 中把 C 语言综合的顶层设置为正常运行的 U123，或者将其设置为按顺序正常运行 U1、 U2 和 U3。

无论采用何种方法来创建这些 IP 块，每个 IP 块都应按下列方法进行独立验证：

- 采用 C/C++ 语言开发的 IP 使用 Vivado HLS 的 C/RTL 协同仿真功能进行验证，这样即可将用于验证基于 C 语言的 IP 的 C 语言测试激励文件用于验证 RTL。
- 而采用 System Generator 开发的 IP 则使用 System Generator 中提供的 MathWorks Simulink 设计环境进行验证。Simulink 环境支持通过使用预定义的仿真元素来轻松生成复杂的输入激励并分析复杂结果。采用 C/C++ 以及通过传统 RTL 生成的 IP 都能导入到 System Generator 环境中，以便充分利用此验证功能。
- 对于用 RTL 生成的 IP，您必须创建 RTL 测试激励文件来验证该 IP。
- 赛灵思和第三方供应商提供的 IP 均已经过预验证，不过您可以根据自己的配置参数集合创建测试激励文件来验证其运行结果。

对 IP 使用标准 AXI 接口能够实现 IP 彼此间的快速集成以及 IP 与 shell 设计之间的快速集成。

系统开发

虽然使用 shell 和多个 IP 块对 FPGA 设计人员而言并非新概念，但此方法通常需要开发和仿真大量 RTL，并且需要多次整合数百乃至数千独立 RTL 信号以完成下列连接：

- shell 到验证 IP 的连接
- shell 到核设计 IP 的连接
- shell 到衍生核设计 IP 的连接

鉴于在传统 RTL 设计流程中使用这种方法会因设计和验证工作量巨大导致额外增加大量工时 （而且如果是在文本编辑器中进行，还容易发生错误），因此设计团队一般选择将所有一切组合在一起进行设计和集成。

Vivado IP integrator 支持采用此方法，无需传统手动编辑 RTL 文件即可迅速完成 IP 集成。

使用这一方法的关键包括：

- Vivado IP 目录
- IP integrator 与标准接口

Vivado IP 目录

Vivado IP 目录是使用 IP 和 IP 复用的任何方法的基干。图 2-3 展示了有关高效设计方法的设计进程的另一种观点，主要展示了使用 IP 目录的位置和时间。

重要提示：使用 IP 目录是实现高效设计方法的关键。

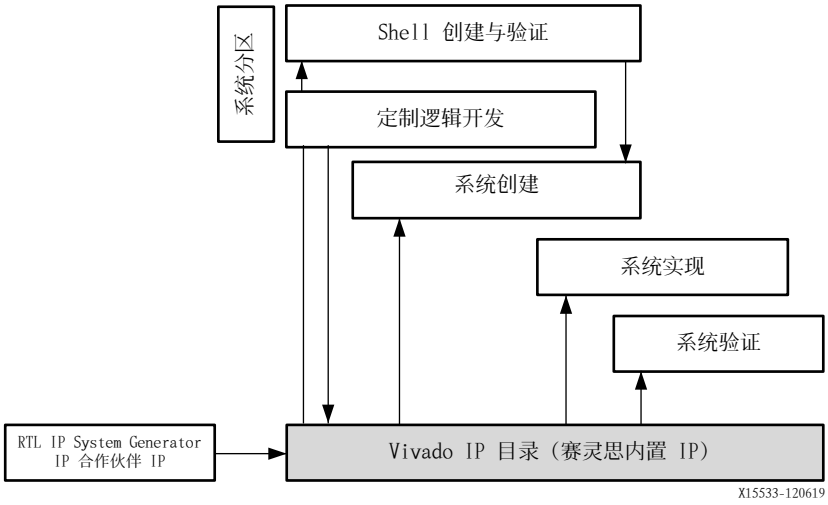

图 2-3：IP 目录与设计进程

IP 目录具有下列特性：

- 内含大约 200 个由赛灵思提供的 IP。如需了解更多信息，请参阅赛灵思 IP 页面 [参照 12]。
- 保存来自基于 C 语言的 IP 开发的输出。
- 能使用 System Generator、传统 RTL 和赛灵思合作伙伴 IP 加以强化。
- 内置大量接口 IP，支持使用传统 RTL IP，在创建 shell 时广泛使用。
- 是系统集成过程中所有 IP 块的来源。
- 提供系统集成和验证期间所使用的 RTL 实现功能。

在 shell 开发过程中，该 shell 可使用 IP 目录提供的 IP 在 IP integrator 中进行组装。其中可包括赛灵思提供的接口 IP（以太网、 VGA、 CPRI、串行收发器等）、赛灵思合作伙伴提供的 IP、作为 IP 供 IP 目录使用的传统 RTL 封装或是 Vivado HLS 和 System Generator 所创建的 IP。

如需了解有关将传统 RTL 封装为 IP 的详细信息，请参阅 《Vivado Design Suite 教程：创建和封装定制 IP》 (UG1119) [参照 5]。

如需了解有关使用来自 System Generator 的 AXI 接口创建 IP 的详情，请参阅 《Vivado Design Suite 用户指南：使用 System Generator 开展基于模型的 DSP 设计》 (UG897) [参照 6]。

Vivado HLS 的默认输出是供 IP 目录使用的封装 IP。欲知详情，请参阅“IP 封装”。

IP integrator 与标准接口

Vivado IP integrator 支持将 IP 块快速添加到画布中并完成连接，这是高效设计方法的关键要素。

重要提示：使用 Vivado IP integrator 如此高效的关键在于使用标准接口。

图 2-4 显示了 IP integrator 内捕获的块设计示例。

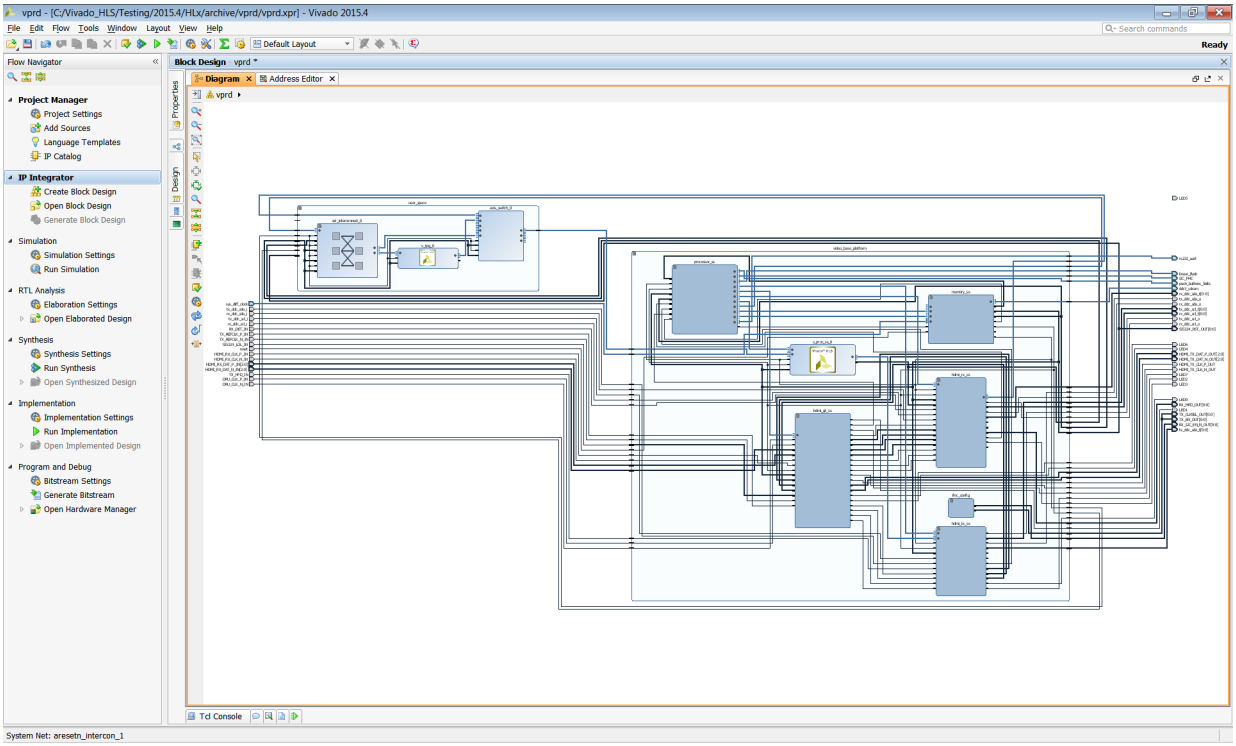

图 2-4：IP integrator 块设计

连接类型包括：

- 管脚级连接，比如时钟和复位信号。
- 总线级连接，比如 AXI、 AXI4-Lite 和 AXI4-Stream 总线。
- 开发板级连接，比如 DDR。

IP integrator 中的连接方式为使用鼠标以图形方式把每个 IP 上的管脚连接起来。除了支持基本的位级连接，还支持总线级连接并提供设计辅助功能。

下图主要体现的是总线级连接的优势。在本示例中，将把 2 个 AXI 主控制器接口连接在一起。请注意，只要建立完到第一个端口的连接，就会在图上用绿色复选标记标识出所有可能的有效连接。

重要提示：IP integrator 禁止建立非法连接。这样可避免在使用手动编辑执行该流程的过程中常见的典型连接错误。

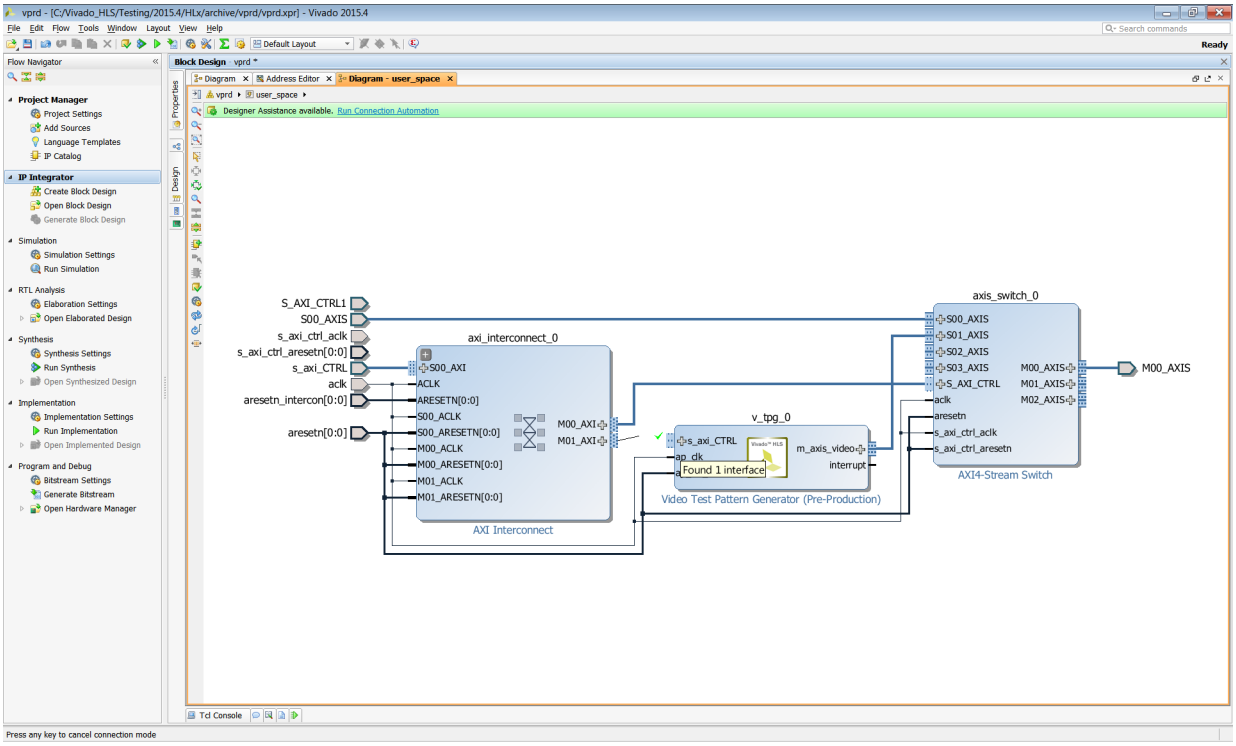

图 2-5：自动连接功能

通过使用标准 AXI 接口和 IP integrator 还会带来另一个有助于提升效率的特性，即自动生成 AXI Interconnect IP。图 2-6 展示了连接结果：

- 一个块上的 AXI 输出
- 另一个块上的 AXI4-Stream 输入

IP integrator 支持自动添加 AXI Interconnect IP，以将主控制器类型的接口连接到流传输类型的接口。

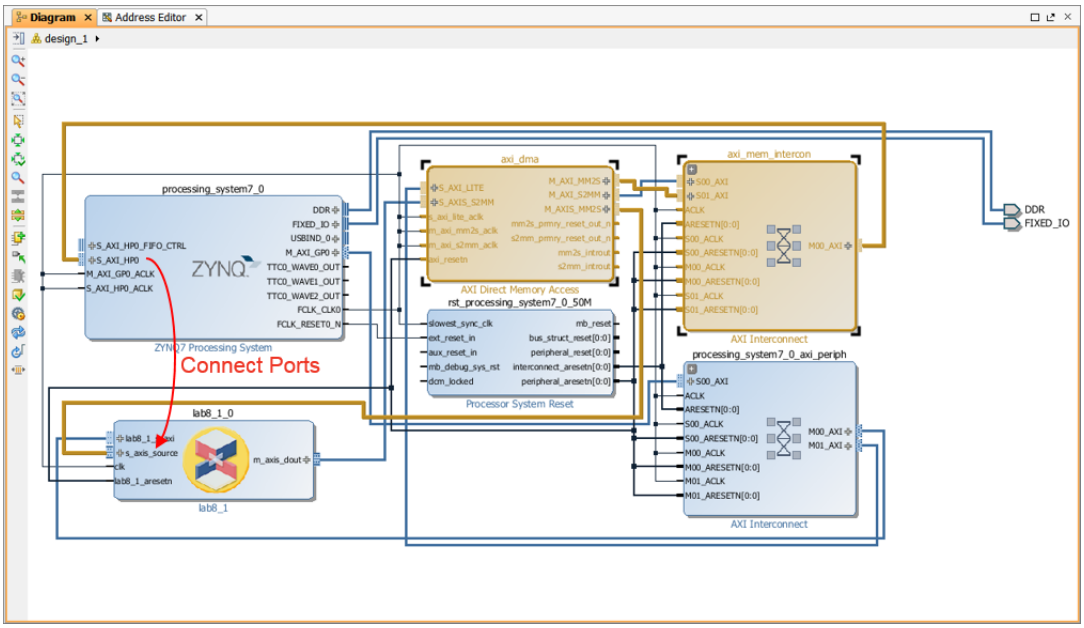

图 2-6：自动添加 AXI Interconnect IP

该 AXI Interconnect IP 在 IP 目录中提供，可手动添加，但 IP integrator 能自动完成这项任务。此外，如果把最终块设计保存为脚本， Tcl 命令就会指出需要连接到哪个管脚。

提示：当升级到新版本的 Vivado Design Suite 和赛灵思 IP 时，请重新运行脚本以确保其使用的是最新的互连逻辑。

在设计中使用标准接口的最后一种情况是为开发板级连接提供的设计辅助功能。除了允许选择目标器件外， Vivado Design Suite 还可以选择目标板。 IP integrator 具有开发板感知能力，能够自动完成开发板级连接。

在得到设计人员确认后， IP integrator 能自动完成 IP 与 FPGA 管脚之间的连接 （开发板连接）。

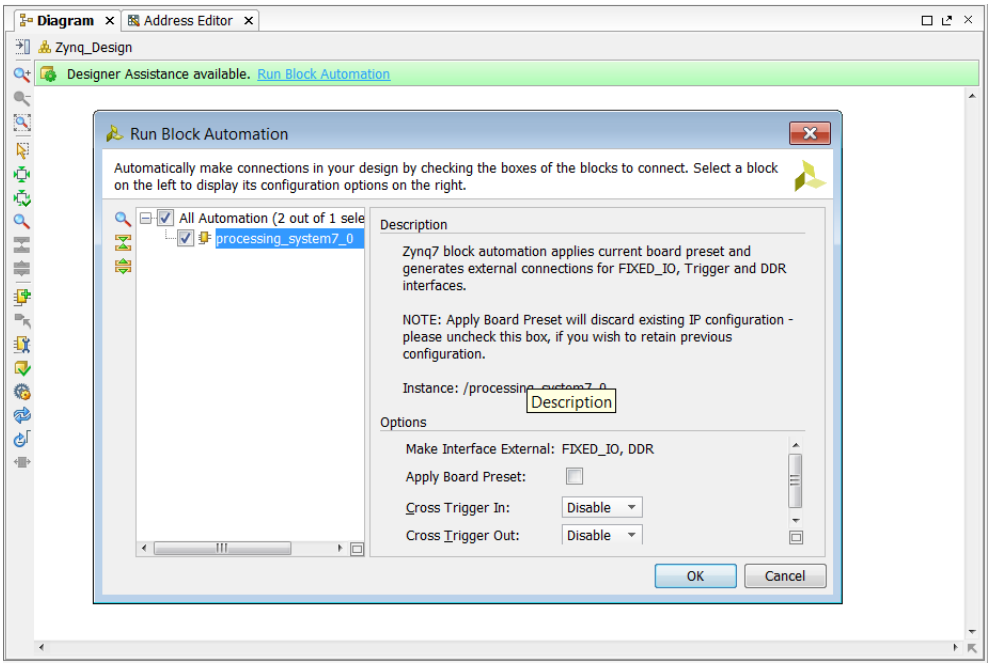

图 2-7：块自动化设置功能

IP integrator 能自动将 IP 集成到块级原理图。其它特性包括使用“验证”功能的“设计规则检查”和为 AXI Interconnect IP 自动添加时钟和复位逻辑。充分利用这一自动化功能和支持高效 shell 方法的关键在于使用标准接口和 AXI 接口开展片上通信。

第 3 章

shell 开发

简介

高效设计方法之所以能提供效率优势，关键就在于使用 shell。shell 设计包含所有标准接口和处理块，用于提供核设计 IP 与系统其余部分之间的连接，并与核设计 IP 并行开发。

整合 shell 设计的方法可实现多项重要的效率提升，例如：

- 它支持设计接口和 I/O 管脚分配的开发工作独立于核设计之外单独进行。
- 它支持在核设计 IP 就绪之前就开展设计接口的验证工作。
- 由于设计更小，它可缩短接口的验证时间；它并不包含一般情况下占据大部分系统逻辑的核设计 IP。
- 它提供了一套高效设计复用方法，支持轻松创建衍生设计。

图 3-1 中简单演示了 shell 设计方法。该方法的一个重要特性正是 shell 设计的复用功能。

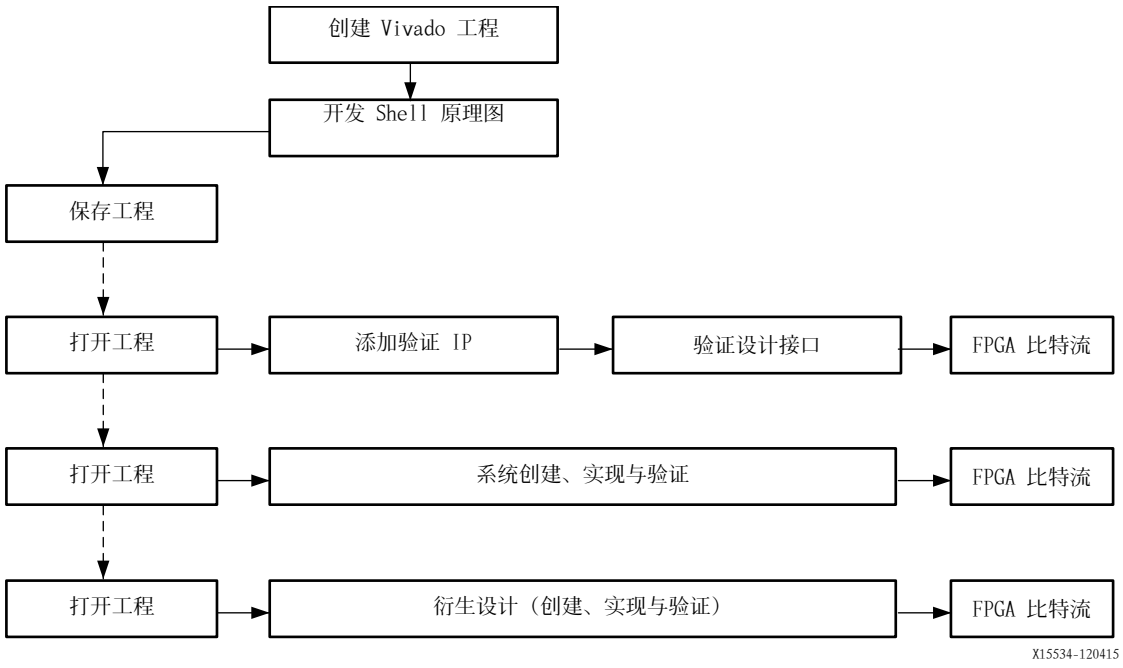

图 3-1：shell 方法

shell 开发包含 2 个同等重要的过程：shell 设计和 shell 验证。

shell 设计

shell 设计仅包括设计边缘部分，如上图所示，而且其采用的形式必须有利于设计复用。 shell 将可保存并可重新打开，以构成多个工程的基础。

为实现如上图所示的流程所需的设计复用功能，在 IP integrator 中，应将 shell 设计作为块设计加以捕获，以便可将其保存并重新打开，从而构成其它设计工程的基础。

组装现有 IP

在 IP integrator 中使用 IP 目录提供的 IP，以块设计的形式组装 shell 设计。

重要提示：准备创建 shell 时，请将 shell 设计中要使用的任何现有 RTL 或公司特有的 IP 封装为 IP 以供在 IP 目录中使用。这样您即可在 shell 块设计中添加此 IP。

如需了解有关如何进行块封装以供 IP 目录使用的详细信息，请参阅 《Vivado® Design Suite 教程：创建和封装定制 IP》 (UG1119) [参照 5]。

shell 设计工程

组装 IP 后，请创建 Vivado RTL 工程。

培训：如需了解有关创建 Vivado RTL 工程的详细信息，请参阅 《Vivado Design Suite QuickTake 视频：创建不同类型的工程》。

创建 Vivado 工程时：

- 请将工程指定为 RTL 工程，并选中“Do not specify any sources at this time”。 shell 设计源就是您在 IP 目录中封装的 IP。
- 理想情况下，请选择赛灵思开发板作为目标。赛灵思开发板所用器件的 I/O 均已完成配置。这样，您即可尽快着手开发自己的定制开发板，您也可以利用 IP integrator 中的设计自动化 (Designer Automation) 功能进行 I/O 连接。

如不指定赛灵思开发板作为目标，那么您还需要为目标器件指定 I/O 连接。请访问此链接以参阅 《UltraFast 设计方法指南 （适用于 Vivado Design Suite）》 (UG949) [参照 7]。

如果在开发过程中使用的是您自己的定制板，建议您创建一个开发板文件以便详述开发板连接情况，并支持使用 IP integrator 中的设计自动化功能，这将极大简化开发板级连接。如需获取有关开发板文件的详细信息，请访问此链接以参阅 《Vivado Design Suite 用户指南：系统级设计输入》 (UG895) [参照 9] 中的“使用 Vivado Design Suite 平台开发板流程”。

工程创建后，使用 Flow Navigator 中的“Create Block Design”按钮来打开 IP integrator 并创建新的块设计。在 IP integrator 窗口中，指定 IP 存储库来源，并使用“Add IP”按钮开始进行组装 shell。

shell 完成后，使用 `write_bd_tcl` 命令把整个块设计保存为 Tcl 脚本。该脚本包含从头开始重新生成块设计所需的一切。块设计和 Vivado 工程保存后即可用于后续验证和系统开发阶段。

在 Documentation Navigator 的“设计中心 (Design Hubs)”选项卡中提供了有关管脚分配、 IP integrator 以及 Vivado Design Suite 中的其它功能的详细信息。如需了解更多信息，请参阅“使用 Documentation Navigator”。

shell 验证

完成 shell 设计创建后即可继续进行 shell 验证。在验证过程中， shell 设计会重新打开，并将验证 IP 添加到设计中，以确认接口正常工作。

shell 验证工程

验证 shell 设计的第一步是使用以下两个选项之一创建新的验证工程。

- 打开用于 shell 设计的 Vivado 工程，使用“File > Save Project As”在新工程中保存 shell 设计。
- 创建一个新的 Vivado RTL 工程 （无 RTL 源代码）并选择相同的目标器件或开发板。然后选择“Create Block Design”，并在控制台中找到使用 `write_bd_tcl` 保存的 Tcl 脚本，将其用于在新工程中重新生成 shell 块设计。

为确保验证设计的复杂性可控，可能需要使用多个验证工程。下图展示了 shell 验证设计示例。本示例仅测试单个接口。

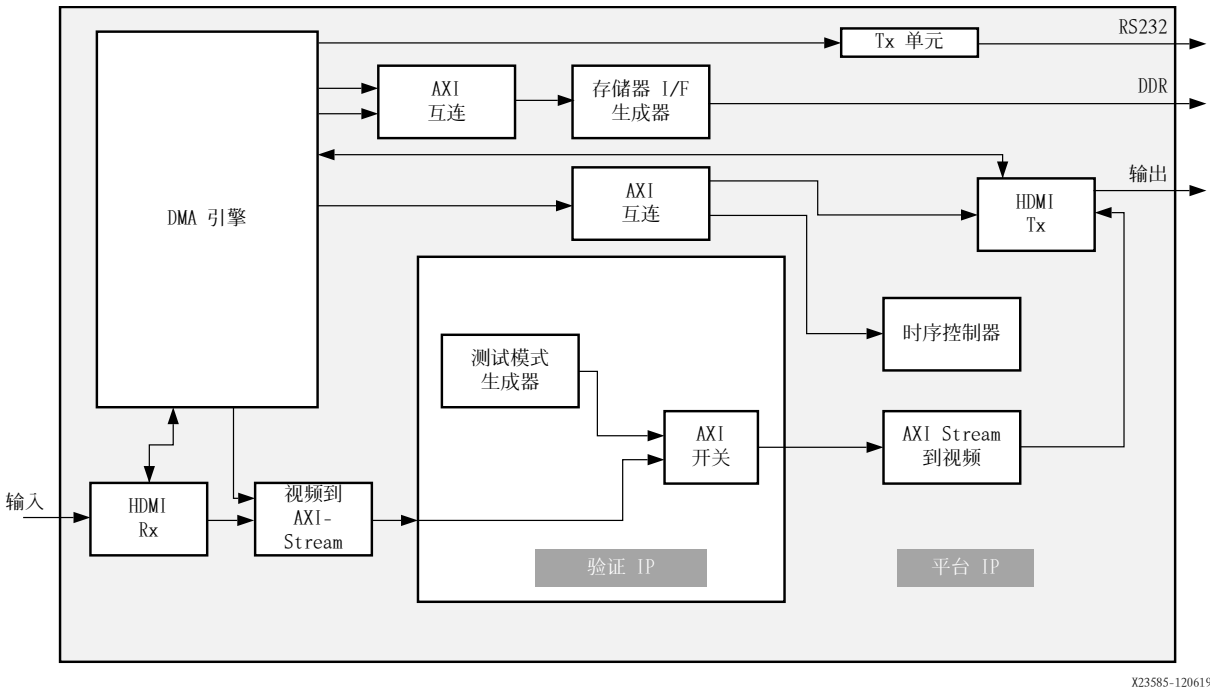

图 3-2：shell 验证示例

验证 IP

将验证 IP 从 Vivado IP 目录添加到 shell 设计中以验证设计。

本指南中讨论的下列所有技巧都可用于开发验证 IP：RTL、System Generator 或基于 C 语言的 IP。以下示例显示了采用标准 AXI 接口 IP 的情况下，如何使用一个小型 C 语言文件在 AXI4-Stream 流接口上快速创建含 N 个样本输出的 HANN 窗口。如以下代码示例所示，只需将接口指令从 `axis` 改为 `m_axi`，即可实现一个 AXI 存储器映射接口：

```
void verify_IP_Hann(float outdata[WIN_LEN]) {
// Specify AXI4-Stream output
#pragma HLS INTERFACE axis port=outdata
// Alternative output AXI4M (commented out)
//#pragma HLS INTERFACE m_axi port=outdata

    float coeff[WIN_LEN];
    coeff_loop:for (int i = 0; i < WIN_LEN; i++) {
       coeff[i] = 0.5 * (1.0 - cos(2.0 * M_PI * i / WIN_LEN));
    }

    winfn_loop:for (unsigned i = 0; i < WIN_LEN; i++) {
      outdata[i] = coeff[i];
     }
}
```

如需了解有关如何使用 Vivado HLS 在其它 IP 之间创建接口块的信息，请参阅 《有关使用 Vivado IP integrator 集成基于 AXI4 的 IP 的方法的应用指南》 (XAPP1204) [参照 10]。

验证 shell

如果在仿真源中添加了顶层测试激励文件，在进行 FPGA 编程前即可通过仿真来验证 shell 设计。

使用 RTL 仿真进行 shell 验证需要创建 RTL 测试激励文件。该测试激励文件同样可用于对完全集成的设计进行验证。如果使用多个验证工程来验证 shell，则应扩展该测试激励文件以验证所有接口。

为验证 FPGA 上的具体接口，可在设计中添加额外的信号级调试探针。

在块设计中工作时，右键单击菜单便可轻松标记信号线以便调试。在硬件运行期间，可对标记调试的信号进行分析：将 ILA 核添加到设计中，以便在 FPGA 中捕获信号并扫描输出以供分析。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) [参照 8]。

最终设计随后通过 Vivado 设计流程进行处理，直至生成比特流。 shell 验证完毕后，对 shell 设计进行的任何修改都应传回最初的源 shell 设计工程，但验证 IP 的修改除外。至此， shell 设计已准备就绪，可用于核设计 IP 的集成。

XILINX

第 4 章

基于 C 语言的 IP 开发

简介

在高效设计流程中，生成核设计 IP 的主要方式是使用基于 C 语言的 IP 和通过高层次综合 (HLS) 把 C 语言代码转化为 RTL。基于 C 语言的 IP 开发流程所带来的优势如下：

- C 语言验证提供的一流仿真速度
- 自动生成经优化且时序精确的 RTL
- 能够使用库中的现有 C 语言 IP
- 使用 IP integrator 即可轻松将生成的 RTL IP 集成到完整系统中

本章将讨论如何创建、验证、综合、分析基于 C 语言的 IP 以及如何对其进行最优化并将其封装为可供 IP 目录使用的 IP。为实现这些目标所采用的方法就是借助 Vivado® 高层次综合 (HLS) 这一由 Vivado Design Suite 提供的工具。

Vivado HLS 设计流程如下图所示。设计流程步骤如下：

1. 编译、执行 （仿真）和调试 C 语言算法。

   注释：在高层次综合中，运行编译后的 C 语言程序被称为 C 语言仿真。 执行 C 语言程序即可对此功能进行仿真，以验证算法功能正常。
2. 将 C 语言程序综合为 RTL 实现，期间可以选择使用用户最优化指令。
3. 生成详细报告并分析设计。
4. 使用按钮式流程验证 RTL 实现。
5. 将 RTL 实现封装为一套选定的 IP 格式。

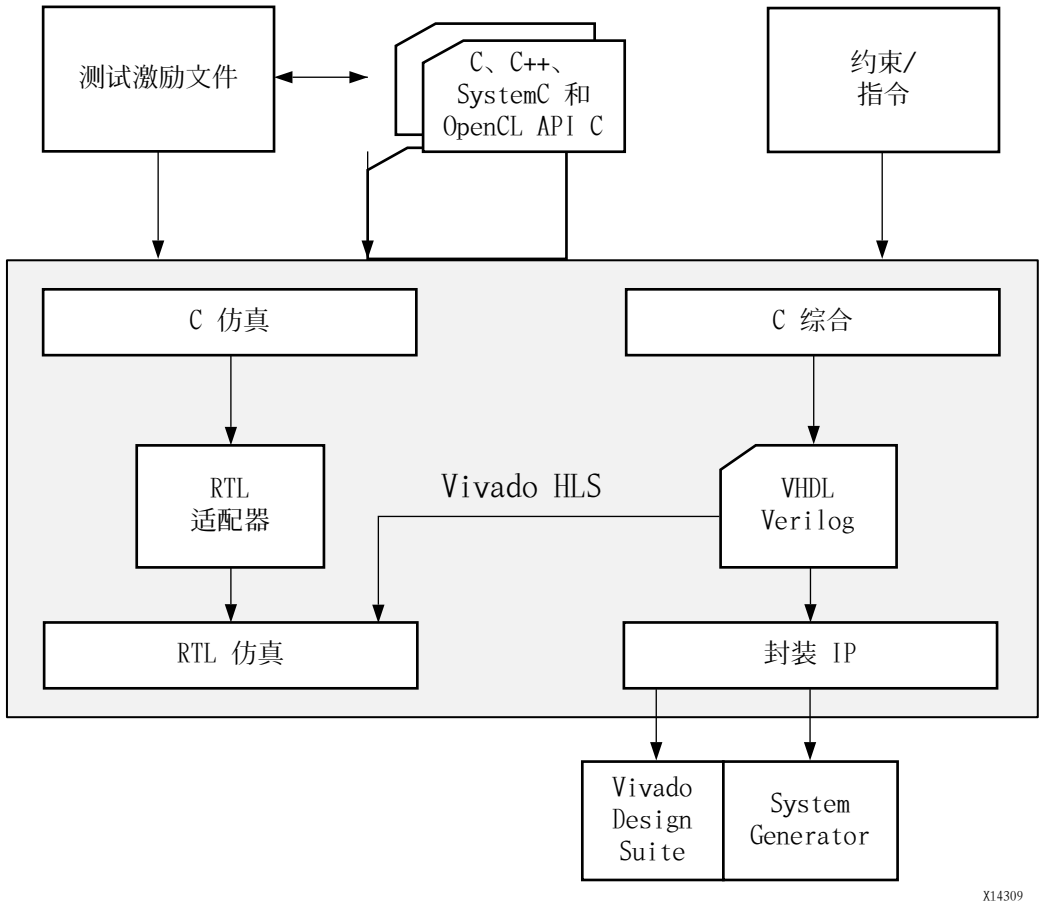

图 4-1：Vivado HLS 设计流程

如需了解有关 Vivado HLS 的详情，请参阅《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 2]。本章介绍的是高效使用 Vivado HLS 的方法。

快速的 C 语言验证

与在 RTL 中仿真相比，在 C 语言中仿真相同的算法速度可快上数倍。

以标准的视频算法为例。典型的 C 语言视频算法先处理完整的 1 帧视频数据，然后把输出图像与基准图像做比较，确认结果的正确性。这个算法的 C 语言仿真一般耗时 10 到 20 秒。 RTL 实现的仿真通常需要几个小时到一天 （数天）不等，具体取决于帧数和设计的复杂性。

借助软件仿真的速度优势， C 语言在设计开发过程中应用越普遍，工作效率就越高。设计人员正是在这一层级上开展实际的设计工作：调整算法、数据类型和位宽以验证和确认设计的正确性。

该流程的其余部分属于开发工作：使用工具链在 FPGA 中实现正确的设计。Vivado Design Suite 和高效设计方法所带来的优势在于为开发流程实现了高度自动化。

完成初始 FPGA 设计实现后，通常可创建完整的全新比特流以对 FPGA 进行编程，这样做比执行全系统 RTL 仿真更快，如需了解有关如何使用脚本化流程来执行此操作，请参阅第 5 章"系统集成"。

为最大程度提升基于 C 语言的 IP 流程的效率，首先应明确以下几点：

C 语言测试激励文件

每一个 C 语言程序的顶层都是 `main()` 函数。Vivado HLS 会对 `main()` 层级下的每个函数进行综合。将由 Vivado HLS 进行综合的函数被称为“设计函数 (Design Function)”。详情请参阅图 4-2。

- 设计函数下的所有函数均由 Vivado HLS 进行综合。
- “设计函数”层级外的全部内容被称为“C 语言测试激励文件”。

C 语言测试激励文件包括 `main()` 下的所有 C 语言代码，用于为“设计函数”提供输入数据以及接受“设计函数”的输出数据以便确认其精确性。

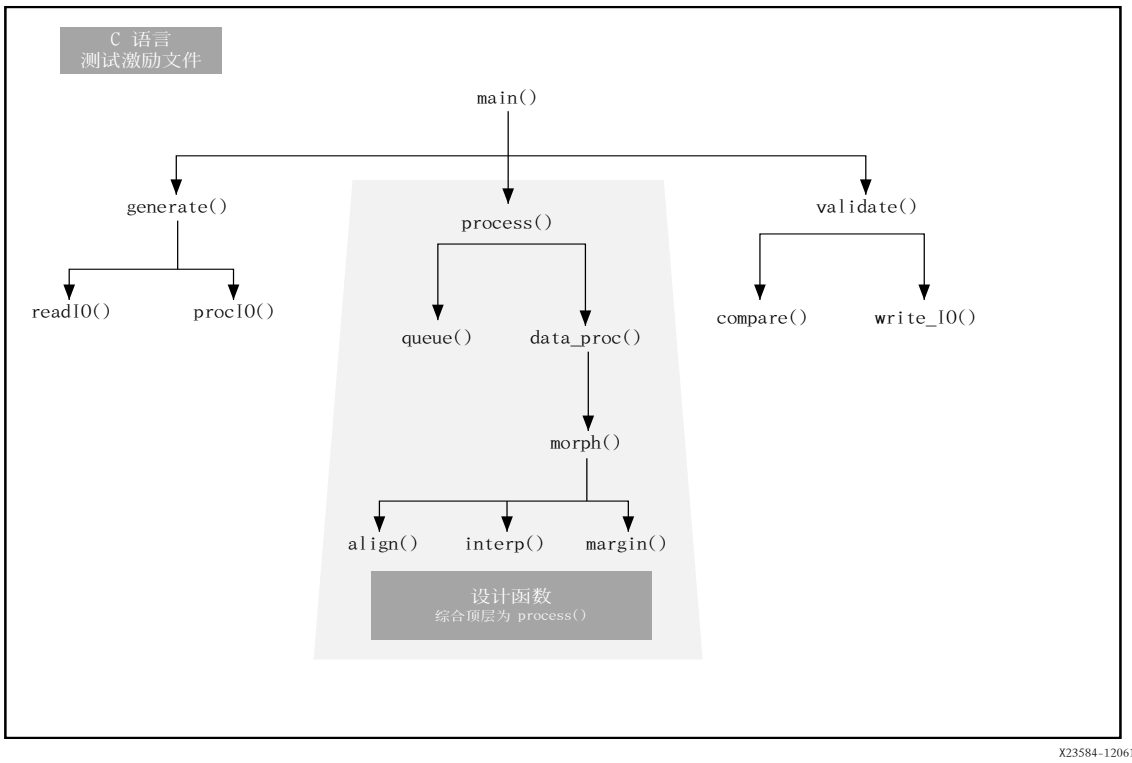

图 4-2：C 语言测试激励文件

Vivado HLS 设计流程中新用户最常见的错误是在不使用 C 语言测试激励文件和不执行 C 语言仿真的情况下，就对其 C 语言代码进行综合。以下代码着重演示了这一类错误。在该嵌套循环示例中存在什么错误？

```
#include "Nested_Loops.h"

void Nested_Loops(din_t A[N], dout_t B[N]) {

  int i,j;
  dint_t acc;

  LOOP_I:for(i=0; i < 20; i++){
    LOOP_J: for(j=0; j < 20; j++){
      if(j=0) acc = 0;
        acc += A[i] * j;
      if(j=19) B[i] = acc / 20;
    }
  }
}
```

该代码未能综合成期望的结果，因为条件语句求值结果为 FALSE 且 J 在 LOOP_J 第一次迭代结束时设置为 19。条件语句的正确表达应为 j==0 且 j==19 （使用 == 取代 =）。前述代码示例能够正常无误完成编译、执行和综合。但是该代码无法发挥预期的作用，而仅靠粗略目测评估无法轻易发现这个问题。

在开发者每天不断使用 C/C++、Perl、Tcl、Python、Verilog 和 VHDL 语言中的一种或多种语言的时代，不仅难以察觉此类小错误，更难以发现功能性错误，而在综合后发现此类错误的难度极大且极为耗时。

C 语言测试激励文件只是一个程序，用于调用要综合的 C 语言函数、为其提供测试数据并测试其输出的正确性；它可在综合前编译和运行，并且在综合前验证期望的结果。

您一开始可能会认为直接进行综合能节省时间，但在设计方法中使用 C 语言测试激励文件所带来的好处巨大，相比之下用于创建测试激励文件的时间不值一提。

自检测试激励文件

Vivado HLS 支持在综合前开展 C 语言仿真以验证 C 语言算法，同时支持在综合后进行 C/RTL 协同仿真，以验证 RTL 设计实现。在这两种情况下，Vivado HLS 均使用函数 main() 的 return 值确认结果的正确性。理想的 C 语言测试激励文件应包含以下代码示例中所示的结果检查属性。用于综合的函数输出将保存在 results.dat 文件中并与期望的正确结果进行比对，也就是本示例中所谓的“理想”结果。

```
int main () {
  ...
  int retval=0;
  fp=fopen("result.dat","w");
  ...
  // Call the function for synthesis
  loop_perfect(A,B);

  // Save the output results
  for(i=0; i<N;++i) {
    fprintf(fp, "%d \n", B[i]);
  }...

  // Compare the results file with the golden results
  retval = system("diff --brief -w result.dat result.golden.dat");
  if (retval != 0) {
    printf("Test failed  !!!\n");
    retval=1;
  else {
    printf("Test passed !\n");
  }

  // Return 0 ONLY if the results are correct
```

```
    return retval;
}
```

在本 Vivado HLS 设计流程中， `main()` 函数的 `return` 值含义如下：

- 零值：结果正确。
- 非零值：结果不正确。

建议：由于系统环境（例如 Linux、Windows 或 Tcl）会对 `main()` 函数的返回值进行解读，赛灵思建议将返回值的范围约束为 8 位以保障可移植性和安全性。

通过使用自检测试激励文件，您无需创建测试激励文件来验证 Vivado HLS 的输出是否正确。用于 C 语言仿真的测试激励文件在 C/RTL 协同仿真期间也会自动使用，并且该测试激励文件还会验证综合后的结果。

C 语言中有多种途径来检查结果是否有效。在上述示例中，用于综合的函数输出将保存到 `result.dat` 文件，并与含期望结果的文件进行比较。这些结果还可以与未标记为用于综合的相同函数做比较（测试激励文件运行时在软件内执行比较）或与测试激励文件计算所得值做比较。

重要提示：如果测试激励文件的 `main()` 函数中没有 `return` 语句，C 语言标准会把 `return` 值设为 0（零值）。因此 C 语言和 C/RTL 协同仿真往往会报告仿真通过，哪怕结果错误也是如此。请检查结果，确认仅当结果正确时才返回零值。

花时间创建自检测试激励文件可确保 C 语言代码中没有明显错误，并且无需创建 RTL 测试激励文件来验证综合输出是否正确。

高位精度的数据类型

随 Vivado HLS 提供有任意精度数据类型，可指定任意宽度的变量。例如可以把变量定义为 12 位、22 位或 34 位宽度。使用标准的 C 语言数据类型的情况下，这些变量分别应为 16 位、32 位和 64 位。使用标准的 C 语言数据类型往往造成使用不必要的硬件来实现所需的精度。例如仅需 34 位时却用 64 位硬件来实现。

使用任意精度数据类型的一个更明显的优势是可以使用这些新的位宽和经过分析的高位精度结果来执行 C 语言算法仿真。例如您可能想设计一种含 10 位输入和 14 位输出的滤波器，同时您判定该设计可使用 24 位累加器。执行 C 语言仿真可在数分钟内用数万样本对滤波器进行仿真，并快速确认输出的信噪比是否可接受。您将能够快速判断累加器是否过小，也可以验证使用更小且更高效的累加器是否仍能提供所需的精度。

重要提示：高位精度 C 语言仿真是验证设计的最快途径。

高效方法即使用标准 C 语言数据类型启动您的初始设计，然后确认算法性能是否符合设计。随后移植 C 语言代码，以使用任意精度数据类型。为确保这种移植到硬件效率更高的数据类型上的操作能够安全高效地执行，前提是存在 C 语言测试激励文件用于检查结果，这样您才能迅速验证更小但更高效的数据类型是否能够胜任需求。当您熟悉任意精度类型的使用后，一般就可以在新的 C 语言工程开始时直接使用任意精度数据类型了。

使用 C 语言测试激励文件的优势，以及在设计方法中不使用它所带来的效率损失，都非常显而易见。

请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2] 中所述的 Vivado HLS 示例，其中随附了 C、C++ 或 SystemC 语言测试激励文件。这些示例可通过复制和修改，用于创建 C 语言测试激励文件。其中包括使用任意精度数据类型的 C 语言函数。

C 语言对综合的支持

理解综合支持哪些语言是 Vivado HLS UltraFast 设计方法的重要组成部分。 Vivado HLS 为 C、 C++ 和 SystemC 提供全面支持。 C 语言仿真享有全面支持，但不能够把每一种描述都综合为等效的 RTL 实现。

在审核代码能否在 FPGA 中实现时，应牢记以下两大原则：

- FPGA 有固定数量的资源。功能必须在编译时固定。硬件中的对象不能动态创建和删除。
- 与 FPGA 的全部通信必须通过输入输出端口进行。 FPGA 不具备底层操作系统 (OS) 或操作系统资源。

不受支持的构造

系统调用

综合不支持系统调用。这些调用用于与执行 C 语言程序的操作系统进行交互。在 FPGA 中不存在需要与之通信的底层操作系统。系统调用的示例包括 `time()` 和 `printf()`。

部分常用函数会被 Vivado HLS 自动忽略，无需将其从代码中移除。这些函数是：

- `abort()`
- `atexit()`
- `exit()`
- `fprintf()`
- `printf()`
- `perror()`
- `putchar()`
- `puts()`

除了移除任何不受支持的代码之外，还有一种方法是禁止其进入综合。 `__SYNTHESIS__` 宏在执行综合时由 Vivado HLS 自动定义。

该宏可用于在运行 C 语言仿真时包含代码，在执行综合时排除该代码。

```
#ifndef __SYNTHESIS__
  // The following code is ignored for synthesis
  FILE *fp1;
  char filename[255];
  sprintf(filename,Out_apb_%03d.dat,apb);
  fp1=fopen(filename,w);
  fprintf(fp1, %d \n, apb);
  fclose(fp1);
#endif
```

注释：仅限在要综合的代码中使用 `__SYNTHESIS__` 宏。请勿在测试激励文件中使用该宏，因为 C 语言仿真或 C RTL 协同仿真不会遵循其指示进行操作。

如果需要操作系统提供信息，则数据必须作为实参传递给顶层函数以供综合使用。随后系统其余部分将负责把该信息提供给综合后的 IP 块。一般采用的方法是把该数据端口实现为连接到 CPU 的 AXI4-Lite 接口。

动态对象

动态对象不能综合。函数调用 `malloc()` 和 `alloc()`、预处理器 `free()` 以及 C++ `new` 和 `delete` 都能动态创建或删除操作系统存储器映射中的存储器资源。FPGA 中可用的唯一存储器资源是块 RAM 和寄存器。块 RAM 是数组综合时创建的，数组中的值必须在一个或者数个时钟周期间保持不变。当变量存储的值必须在一个或者数个时钟周期内保持不变时，就会创建寄存器。必须使用固定大小的数组或变量替代任何动态存储器分配。

与动态存储器使用限制一样，Vivado HLS 不支持在综合中动态创建或删除的 C++ 对象。这包括动态多态性和动态虚拟函数调用。在运行时，无法动态创建可能生成新硬件的新函数。

出于类似原因，综合中也不支持递归。在编译时，所有对象的大小都必须已知。在使用模板时，可为递归提供有限支持。

除了 `std::complex` 等标准数据类型外，综合中也不支持 C++ 标准模板库 (STL)。这些库内包含的函数会大量使用动态存储器分配和递归。

SystemC 语言构造

`SC_MODULE` 不能进行内部嵌套，也不能从另一个 `SC_MODULE` 衍生。

不支持 `SC_THREAD` 构造 （但支持 `SC_CTHREAD`）。

受有限支持的构造

顶层函数

综合中支持使用模板，但不支持将其用于顶层函数。

C++ 类对象不能在综合中直接用于顶层。该类必须例化为顶层函数。

综合中支持指针到指针，但不支持将其用作顶层函数的实参。

指针支持

Vivado HLS 支持原生 C 语言类型间的指针强制转型，但不支持通用指针强制转型，比如为区分结构类型而仅限指针间强制转型。

Vivado HLS 支持指针数组，前提是每个指针都指向标量 (scalar) 或标量数组。指针数组不能指向其它指针。

递归

在 FPGA 中仅支持使用模板进行递归。在综合中执行递归的关键是使用大小为 1 的终端类在递归中实现最终调用。

存储器函数

`memcpy()` 和 `memset()` 均受支持，但其限制条件是使用的值必须为 `const`。

- `memcpy()`：用于总线突发操作或含 const 值的数组初始化。 memcpy 函数只能用于在实参与顶层函数之间进行值的双向复制。
- `memset()`：用于含常量给定值的聚集初始化。

无论是不支持用于综合的任何代码还是仅为其提供有限支持的代码，都必须经过修改后才能进行综合。

如需获取有关语言支持的详细信息，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

使用经硬件最优化的 C 语言库

Vivado HLS 为常用的 C 语言函数提供了多个 C 语言库。 C 语言库中提供的函数一般都已预先经过最优化，可确保在综合时以高性能且高效率的方式完成设计实现。

请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2] 中的“高层次综合 C 语言库”，其中提供了有关 Vivado HLS 随附的所有 C 语言库的详细信息，但强烈建议您评估这些 C 语言库中哪些 C 函数适用于自己的方法。

Vivado HLS 包含下列 C 语言库：

- 任意精度数据类型
- HLS 流传输库
- 数学函数
- 线性代数函数
- 数字信号处理 (DSP) 函数
- 视频函数
- IP 库

理解 Vivado HLS

在阅读本指南后续章节之前，有必要了解一些与基于 C 语言的 IP 最优化相关的关键 HLS 概念。本章节提供了这些概念的简介。

性能指标

Vivado HLS 可根据自身默认的综合行为和约束快速创建最理想的设计实现。时钟周期是主要的约束， Vivado HLS 将其与目标器件规格相结合来判断每个时钟周期内可完成的操作次数。

在满足时钟频率约束后， Vivado HLS 所使用的性能指标按最优化重要性排序如下：

- **启动时间间隔 (II)**：表示新输入之间的时钟周期数量。它代表了吞吐量以及设计读取并处理下一项输入的速度。
- **时延**：表示生成输出所需的时钟周期数量。在实现最小间隔后，或者没有明确指定内部目标的情况下， Vivado HLS 会尽可能最大程度减小时延。
- **面积**：在实现最小时延后， Vivado HLS 会尽可能最大程度减小面积。

在报告性能指标时，报告的是整个函数的指标。例如，如果函数包含标量输入，II=3 表示该函数每 3 个时钟周期处理 1 个样本。但如果该函数包含由 N 个元素组成的输入数组，则 II=N 表示每 N 个时钟周期处理 N 个元素：即每个时钟周期处理 1 个样本的比率值。

最优化指令可用于指示 Vivado HLS 创建设计，并对上述指标进行排序，例如，强制优先 （于吞吐量）减小面积或时延。如无最优化指令， Vivado HLS 会根据这些目标并使用下述默认综合行为来创建初始设计。

接口综合

顶层函数的实参可通过可选 I/O 协议综合为数据端口。 I/O 协议指与数据端口关联的一个或多个信号，用于在数据端口与系统中的其它硬件块之间自动同步数据通信。

例如，在握手协议中，数据端口会伴随一个有效端口用于指示何时数据有效并可供读写，另包含一个确认端口用于指示已成功完成数据读写。

请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2] 中提供的 I/O 协议的完整描述，但这些接口包括 AXI、 AXI4-Stream 和 AXI4-Lite 接口，因此使用 IP integrator 即可轻松将 IP 集成到系统中。

此外，默认情况下，仅针对顶层函数本身实现 I/O 协议。该协议用于控制 IP 开始操作的时间，并指示 IP 完成操作的时间或者 IP 准备好接受新输入数据的时间。该可选 I/O 协议可实现为 AXI4-Lite 接口以便用微处理器控制该设计。

函数综合

在最终 RTL 设计中，函数被综合为层级块。 C 语言代码中的每个函数在最终 RTL 中都显示为一个唯一的块。一般情况下最优化操作止于函数边界，部分最优化指令具有允许指令跨函数边界生效的递归选项或行为。

使用最优化指令即可使用内联函数。这样可以永久性去除函数层级，实现更好的逻辑最优化。函数还可以通过流水线化来提高其吞吐量性能。

函数可以调度为尽早执行。以下示例显示了两个函数：`foo_1` 和 `foo_2`。

```
void foo_1 (a,b,c,d,*x,*y) {
  ...
  func_A(a,b,&x);
  func_B(c,d,&y);
}
```

在 `foo_1` 函数中， `func_A` 函数和 `func_B` 函数之间不存在数据依赖关系。虽然这两个函数在 C 语言代码中按顺序出现，但在 Vivado HLS 实现的架构中，这两个函数在第一个时钟周期内同时开始处理数据。

```
  void foo_2 (a,b,c,*x,*y) {
    int *inter1;
     ...
     func_A(a,b,&inter1,&x);
     func_B(c,d,&inter1,&y)
}
```

在 `foo_2` 函数中，函数之间存在数据依赖关系。内部变量 `inter1` 由 `func_A` 传递给 `func_B`。在本示例中 Vivado HLS 必须将 `func_B` 函数调度为在 `func_A` 函数执行完成后才开始执行。

循环综合

循环默认保持“收起”状态。这意味着 Vivado HLS 在循环主体中对该逻辑仅综合一次，然后按顺序执行该逻辑，直至抵达循环终止条件为止。循环可以“展开”以便并行执行所有操作，但这样会为循环硬件创建多个复本，或者也可以将循环流水线化以提升性能。

循环一般调度为按顺序执行。在以下示例中，SUM_X 循环与 SUM_Y 循环之间没有依赖关系，但两者始终调度为按其在代码中显示的顺序执行。

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
             dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     SUM_X:for (i=0;i<xlimit; i++) {
          X_accum += A[i];
          X[i] = X_accum;
}

     SUM_Y:for (i=0;i<ylimit; i++) {
          Y_accum += B[i];
          Y[i] = Y_accum;
     }
}
```

示例 4-1： 顺序循环

逻辑综合

默认情况下，始终将函数和循环中的逻辑综合调度为尽早执行。 Vivado HLS 始终尽可能最大程度缩短时延，同时实现设计约束。C 语言代码中的运算符（如 +、* 和 / 等）都综合为硬件核。Vivado HLS 会自动选择最合适的核以实现综合目标。最优化指令 RESOURCE 可用于明确指定使用哪个硬件核来实现特定操作。

数组综合

默认情况下， Vivado HLS 将数组综合为块 RAM。

在 FPGA 中，块 RAM 以多个块的形式提供，每个块均由 18K 位原语元素组成。每个块 RAM 根据需求使用特定数量的 18K 原语元素来实现该数组。例如由 1024 种 `int` 类型组成的数组需要 1024 * 32 位 = 32768 位块 RAM，相当于需要 32768/18000 = 1.8 个 18K 块 RAM 原语来实现该块 RAM。虽然 Vivado HLS 报告每个数组综合为一个块 RAM，但该块 RAM 可能包含多个 18K 原语块 RAM 元素。

在默认条件下 Vivado HLS 不会试图把较小的块 RAM 组合成单个大型块 RAM，也不会把大型块 RAM 分区成多个较小的块 RAM。但是可通过使用最优化指令来实现此操作。 Vivado HLS 可能自动把小数组分区为单独的寄存器，以提升结果质量。

Vivado HLS 会根据综合目标自动判断是使用单端口还是双端口块 RAM。例如，如果有助于最大程度缩短时间间隔或时延，则 Vivado HLS 会使用双端口块 RAM。要明确指定是使用单端口还是双端口块 RAM，可使用 RESOURCE 最优化指令。

流水线函数、循环和任务

实现高性能设计的关键在于使用 PIPELINE 和 DATAFLOW 最优化指令来将函数、循环和任务流水线化。

下图演示了流水线化的概念说明。在不使用流水线化的情况下，操作会顺序执行直至函数完成。然后开始执行函数的下一步操作或下一项传输事务。使用流水线时，一旦硬件资源变为可用，下一项传输事务就会即刻启动。

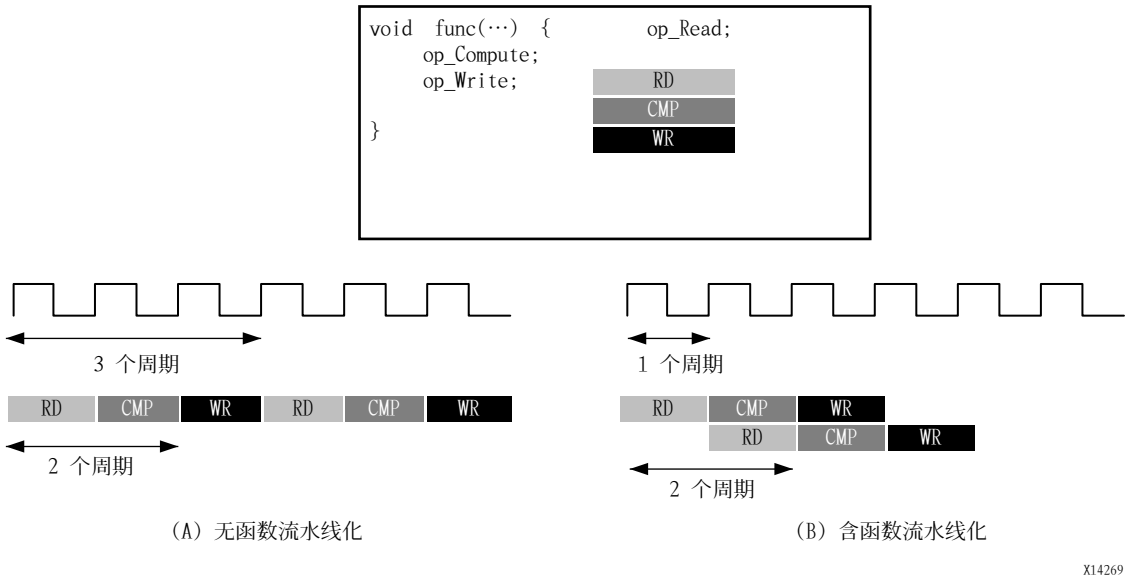

图 4-3：流水线行为

PIPELINE 指令可用于函数或循环，这样即可以最小的面积开销提升吞吐量 （最大限度缩短 II）。

函数和循环均被视为任务。 DATAFLOW 指令用于将任务“流水线”化，使其在数据依赖关系允许的情况下并行执行。

图 4-4 演示了任务流水线化的概念视图。综合完成后，默认行为是先执行和完成 func_A，然后是 func_B，最后是 func_C。不过您可以使用 DATAFLOW 最优化指令将每个函数调度为一旦数据可用就即刻执行。

在本示例中原始函数的时延和时间间隔为 8 个时钟周期。在使用数据流 (dataflow) 最优化后，时间间隔缩短到仅 3 个时钟周期。本例中所示的任务为函数，但您也可以在函数之间、在函数与循环之间以及在循环之间执行数据流最优化。

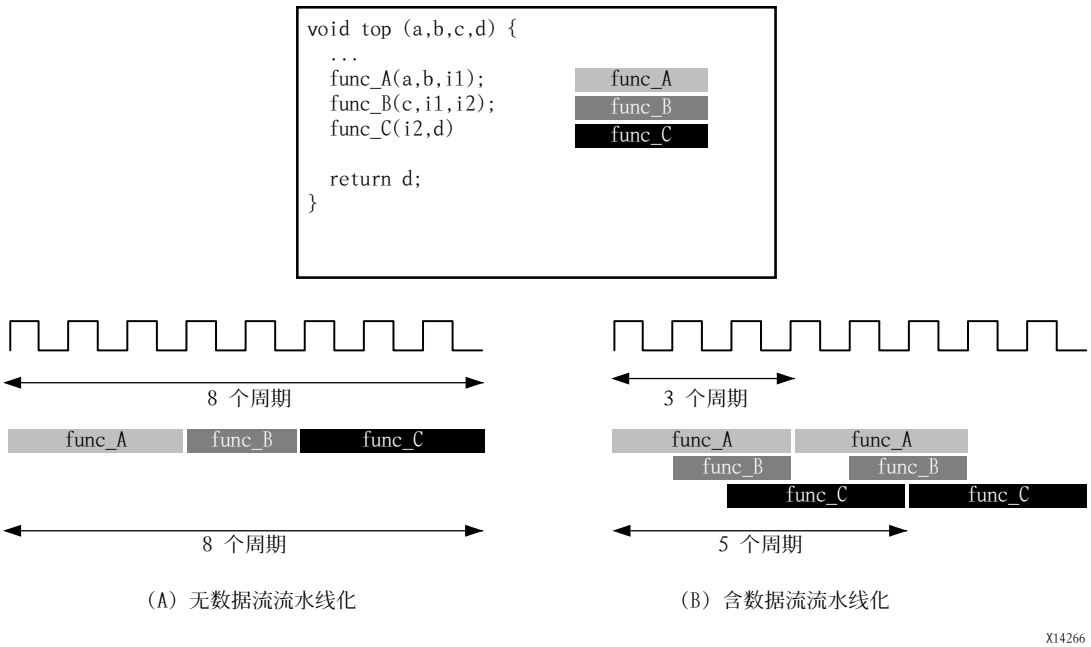

图 4-4：数据流最优化

Vivado HLS 资源

Documentation Navigator 中的 Vivado HLS 设计中心为进一步了解 Vivado HLS 提供了方便的渠道，其中包括：

- 讲解具体操作的 QuickTake 视频
- 关于设计流程各方面的教程
- Vivado HLS 用户指南
- 多个应用指南

如需了解有关设计中心的更多信息，请参阅“使用 Documentation Navigator”。

最优化方法

除前述章节讨论的默认综合行为， Vivado HLS 还提供了多项最优化指令和配置，用于引导综合达成所期望的结果。本节将介绍用于通过最优化来实现高性能设计的常规方法。

在使用 Vivado HLS 进行设计最优化时可能面临多重目标。该方法假定您要创建的设计具备最高的性能，每个时钟周期处理一个新输入数据样本，因此先考虑如何解决这些最优化需求，然后再考虑其它用于缩短时延或减少资源的需求。

下一节“HLS 最优化方法”将探讨如何将本节所述方法应用于其它 C 语言代码架构。

如需了解有关本节所述的最优化的详细说明，请参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2] 中的下列章节：

- “管理接口”，请参阅此链接。
- “设计最优化”，请参阅此链接。

强烈建立先复查本方法并全面了解高层次综合最优化之后，再查看特定最优化的详细信息。

HLS 最优化方法

图 4-5 中显示了 Vivado HLS 的最优化方法。首先验证 C 语言代码是否功能正常，这尤其重要。其余的步骤将在下文中具体介绍，可大致分为：确定接口、设计流水线化、通过数据结构最优化解决阻碍最优化流水线的问题、然后解决任何时延和面积问题。

图 4-5：HLS 最优化方法

以下是最优化指令的完整列表。列表左侧显示的是 Tcl 命令，右侧是可直接包含在 C 语言代码中的等效编译指示指令：

```
set_directive_allocation           - Directive ALLOCATION
set_directive_array_map            - Directive ARRAY_MAP
set_directive_array_partition      - Directive ARRAY_PARTITION
set_directive_array_reshape        - Directive ARRAY_RESHAPE
set_directive_data_pack            - Directive DATA_PACK
set_directive_dataflow             - Directive DATAFLOW
set_directive_dependence           - Directive DEPENDENCE
set_directive_expression_balance   - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline               - Directive INLINE
set_directive_interface            - Directive INTERFACE
set_directive_latency              - Directive LATENCY
set_directive_loop_flatten         - Directive LOOP_FLATTEN
set_directive_loop_merge           - Directive LOOP_MERGE
set_directive_loop_tripcount       - Directive LOOP_TRIPCOUNT
set_directive_occurrence           - Directive OCCURRENCE
set_directive_pipeline             - Directive PIPELINE
set_directive_protocol             - Directive PROTOCOL
```

```
set_directive_reset                - Directive RESET
set_directive_resource             - Directive RESOURCE
set_directive_stream               - Directive STREAM
set_directive_top                  - Directive TOP
set_directive_unroll               - Directive UNROLL
```

通过配置可修改默认的综合行为。配置没有等效的编译指示。在 GUI 中使用“Solution > Solution Settings > General”菜单来设置配置。可用配置的完整列表如下：

```
config_array_partition             - Config the array partition
config_bind                        - Config the options for binding
config_compile                     - Config the optimization
config_dataflow                    - Config the dataflow pipeline
config_interface                   - Config command for io mode
config_rtl                         - Config the options for RTL generation
config_schedule                    - Config scheduler options
```

拥有所有的最优化指令和综合配置列表固然好。掌握其使用方法则更好。

步骤 1：初始最优化

下表显示的是您应首先考虑添加到设计中的指令。

表 4-1：最优化策略步骤 1：初始最优化

指令和配置	描述
INTERFACE	指定如何根据函数描述创建 RTL 端口。
DATA_PACK	把结构体的数据字段打包到字宽更宽的单一标量中。
LOOP_TRIPCOUNT	用于含变量边界的循环。提供估算的循环迭代计数。这对综合没有影响，只对报告功能有影响。
Config Interface	该配置用于控制与顶层函数实参无关联的 I/O 端口，支持从最终 RTL 中去除未使用的端口。

该设计接口一般由系统中的其它块定义。因为 I/O 协议的类型有助于确定综合所实现的结果，因此建议先使用 INTERFACE 指令指定此类型，然后再继续进行设计最优化。

如果该算法以流传输方式访问数据，可以考虑使用某个流传输协议来确保高性能运行。

提示：如果 I/O 协议完全被外部块固定，永远无法修改，可考虑将 INTERFACE 指令作为编译指示直接插入 C 语言代码。

当在顶层实参列表中使用结构体时，会将其分解为单独的元素，且结构体的每个元素都会实现为单独的端口。在某些情况下可使用 DATA_PACK 最优化把整个结构体实现为单个数据字，从而生成单一 RTL 端口。如果结构体包含大型数组，则应谨慎处理。数组的每个元素都在数据字中实现，这可能导致数据端口过宽。

设计首次综合后的一个常见问题是报告文件中时延和时间间隔显示为问号“?”而非数字值。如果设计中的循环含有变量循环绑定， Vivado HLS 将无法确定时延，并使用“?”来表示此状况。

要解决此问题，请使用分析透视图或综合报告来定位综合无法报告数字值的最底层循环，然后使用 LOOP_TRIPCOUNT 指令施加一个估算的行程计数 (tripcount)。这样就可以报告时延和时间间隔值，也可以对采用不同最优化方法的解决方案进行比对。

注释：带变量绑定的循环无法完全展开，且导致层级中位于其上方的函数和循环无法流水线化。这个问题将在下一节中讨论。

最后，在用于综合的函数范围内，一般情况下全局变量可供读写，并且在最终 RTL 设计中无需用作为 I/O 端口。如果全局变量用于在 C 语言函数上输入或输出信息，那么建议您使用接口配置将其作为 I/O 端口予以公开。

步骤 2：性能流水线

创建高性能设计的下一阶段是函数、循环和任务流水线化。下表显示了可用于流水线化的指令。

表 4-2：最优化策略步骤 2：性能流水线

指令和配置	描述
PIPELINE	通过允许在循环或函数中并行执行操作，从而降低启动时间间隔。
DATAFLOW	支持任务层次的流水线化，允许函数和循环并行执行。用于最大程度缩短时间间隔。
RESOURCE	指定用于在 RTL 中实现变量 （数组、算术运算或函数实参）的资源 （核）。
Config Compile	支持循环根据自身的迭代计数自动执行流水线化。

在最优化流程的这个阶段，建议您创建尽量多的并行操作。您可将 PIPELINE 指令应用于函数和循环。您可在包含函数和循环的层次使用 DATAFLOW 指令，使其并行工作。

推荐的策略是采用自下而上的工作方式，并注意下列情况：

- 部分函数和循环内含子函数。如果子函数未流水线化，位于其上层的函数在流水线化后可能表现为性能提升有限。未流水线化的子函数将成为制约因素。
- 部分函数和循环内含子循环。使用 PIPELINE 指令时，该指令会在下层层级中自动展开所有循环。这样会产生大量逻辑。在下层层级中执行循环流水线化可能更为合理。
- 含变量绑定的循环无法展开，位于这些循环上层层级中的任何循环和函数都无法进行流水线化。为解决这一问题，可先将这些循环流水线化，然后再使用 DATAFLOW 最优化来最大限度提升包含此循环的函数的性能。此外也可重写循环，移除变量绑定。

最优化流程中此阶段的基本策略是尽可能将更多任务 （函数与循环）流水线化。如需了解有关需要流水线化的具体函数和循环以及 DATAFLOW 指令的应用范围的详细信息，请参阅“最优化策略”。

对具有大量循环或大量嵌套循环的设计，编译配置功能提供了一种方法，可根据设计的循环迭代计数对设计中的全部循环自动进行流水线化。如需了解更多信息，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

您也可以在运算符层级应用流水线化，虽然此方法并不常用。例如， FPGA 中的线路布线会引发大量且无法预测的延迟，导致设计难以按所需时钟频率来实现。在这种情况下，您可使用 RESOURCE 指令将特定操作 （例如，乘法器、加法器和块 RAM）加以流水线化。

RESOURCE 指令可指定用于实现 C 语言代码中的操作的具体的硬件核。如果指定实现的资源的时延大于 1，则会导致 Vivado HLS 对此操作使用额外的流水线阶段。 RTL 综合能利用这些额外的流水线阶段来改善总体时序。

下列操作支持流水线化的实现：

- 有多阶 (*nS) 核可用的标准算术运算
- 浮点运算
- 用块 RAM 实现的数组

步骤 3：通过结构最优化提升性能

C 语言代码包含的某些描述可能阻碍函数或循环根据性能要求进行流水线化。在某些情况这需要进行代码修改，但大部分情况下，这些问题都可使用其它最优化指令来解决。

以下示例显示了如何使用最优化指令来提升流水线化的性能。在此初始示例中，将向循环添加 PIPELINE 指令以提升循环的性能。

```
#include "bottleneck.h"
```

```
dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

当上述代码完成综合后，会输出以下消息：

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
I
```

如果流水线化不能满足所需性能，解决问题的关键在于采用分析透视图来核查设计。如需了解有关使用“分析透视图 (Analysis Perspective)”的详细信息，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。分析透视图中该设计示例的视图如下图所示。

- 存储器 （块 RAM）访问在该图中以高亮显示。这些访问对应的是上述代码中的 mem 数组。
- 每次访问耗时两个周期：1 个周期用于生成地址， 1 个周期用于读取数据。
- 由于每个块 RAM 最多只有 2 个数据端口，周期 C1 中只能启动 2 次存储器读取。
- 第 3 次和第 4 次存储器读取只能在周期 C2 中开始。
- 下一组存储区读取操作最早可从周期 C3 开始。这意味着该循环只能有 II=2：循环的下一组输入只能每 2 个周期读取一次。

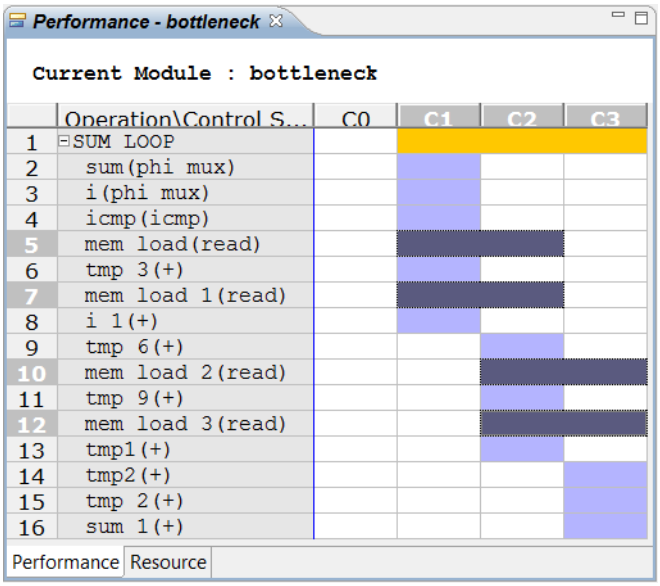

图 4-6：因端口数过少导致流水线失败

可以通过对 mem 数组使用 ARRAY_PARTITION 指令来解决存储器端口限制问题。该指令用于把数组分区为较小数组，从而提供更多数据端口，并改善数据结构以实现性能更高的流水线。

通过使用以下所示的附加指令，可将 mem 数组分区为 2 个双端口存储器，这样即可在 1 个时钟周期内完成全部 4 次读取操作。在数组分区时可以有多种选择。此处循环分区因数为 2，这可确保第 1 个分区包含来自原始数组的元素 0、2、 4，第 2 个分区可包含元素 1、 3、 5 等。使用双端口块 RAM，即可在 1 个时钟周期中读取元素 0、 1、 2 和 3。

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

在尝试循环和函数流水线化时，可能会遇到其它的此类问题。下表列出的指令有助于减少数据结构中的瓶颈，因此可能可以解决此类问题。

表 4-3：最优化策略步骤 3：通过结构最优化提升性能

指令和配置	描述
ARRAY_PARTITION	把大型数组分区为多个较小数组或分区为单独的寄存器，以改善数据访问并消除块 RAM 瓶颈。
DEPENDENCE	用于提供附加信息，这些信息可用于克服循环附带的依赖关系并支持循环流水线化（或以较低时间间隔流水线化）。
INLINE	内联函数，消除所有函数层级。用于跨越函数边界实现逻辑最优化，通过减少函数调用开销来改善时延/时间间隔。
UNROLL	展开 for 循环，创建多个独立操作而非单个操作集。
Config Array Partition	该配置用于判断包括全局数组在内的数组分区方式，以及分区是否会影响数组端口。
Config Compile	控制综合专用最优化功能，例如自动循环流水线化和浮点运算最优化。
Config Schedule	判断综合调度阶段的工作量以及输出消息的详细程度，并指定是否应在流水线化任务中放宽 II 以实现时序收敛。
CONFIG_UNROLL	允许自动展开低于指定循环迭代次数的所有循环。

除 ARRAY_PARTITION 指令之外，用于数组分区的配置也可用于数组自动分区。

用于编译的配置可用于循环层级的自动流水线化。进行循环流水线化时，可能需要用 DEPENDENCE 指令移除暗含的依赖关系。此类依赖关系将通过 SCHED-68 消息进行报告。

```
@W [SCHED-68] Target II not met due to carried dependence(s)
```

INLINE 指令可用于移除函数边界。它可用于将逻辑或循环上移一个层级。在实现函数中的逻辑流水线化的过程中，将其上层函数的逻辑一并包含在流水线化操作中可能效率更高，并且，将一连串循环所在层级上移可能更便于将这些循环与其它循环一并进行数据流处理。

如果循环无法以所需启动时间间隔加以流水线化，则可能需要使用 UNROLL 指令。如果某个循环只能按 II=4 进行流水线化，那么它将把系统中的其它循环和函数约束在 II=4 的水平。在某些情况下确有必要展开循环，虽然这样会创建更多逻辑，但是可以消除潜在瓶颈。

调度配置用于提升调度信息的详细程度并控制调度中的工作量。使用 verbose 选项时，如果调度无法满足约束，Vivado HLS 会列出关键路径。

一般情况下，通过增加调度工作量来帮助改善调度的用例较为罕见，但仍提供此选项以备不时之需。如果最优化指令和配置无法用于改善启动时间间隔，那么可能需要修改代码。如需获取相关示例，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

步骤 4：缩短时延

当 Vivado HLS 完成最大程度缩短启动时间间隔后，会自动尽可能缩短时延。下表中列出了有助于缩短时延或指定特定时间的最优化指令。

在循环和函数流水线化时一般无需使用这些指令，因为在大多数应用中时延并非关键，通常吞吐量才是关键。如果循环和函数未流水线化，则吞吐量将受到时延限制，因为只有在上一个任务完成后，下一个任务才会开始读取下一组输入。

表 4-4：最优化策略步骤 4

指令	描述
LATENCY	允许指定最小和最大时延约束。
LOOP_FLATTEN	允许把嵌套循环折叠为已改善时延的单一循环。
LOOP_MERGE	合并连续循环，以缩短总体时延、增加共享和提升逻辑最优化。

LATENCY 指令用于指定所需的时延。循环最优化指令可用于扁平化循环层级或把多个连续循环合并在一起。对于时延而言，该指令的作用体现在通常进入和退出循环需耗费 1 个时钟周期。循环间过渡次数越少，设计完成所需的时钟周期数就越少。

步骤 5：缩小面积

在满足所需性能目标 （或 II）后，下一步是在保持性能不变的情况下缩小面积。

如果已使用 DATAFLOW 最优化且 Vivado HLS 无法判定设计中的任务是否在流送数据，那么 Vivado HLS 会使用乒乓缓存 (ping-pong buffer) 来实现数据流任务之间的存储器通道。如果设计已完成流水线化且数据正在从一个任务流送到下一个任务，那么使用数据流配置 `config_dataflow` 把默认存储器通道中使用的乒乓缓存转换为 FIFO 缓存即可显著缩小面积。随后可把 FIFO 深度设置为需要的最小值。

数据流配置 `config_dataflow` 可指定所有存储器通道的默认实现方式。您可以使用 STREAM 指令指定哪些单独的数组将实现为块 RAM 以及哪些数组将实现为 FIFO。

如果设计使用 `hls::stream` I/O 协议来实现，存储器通道默认将采用深度为 1 的 FIFO，并且无需使用数据流配置，但如果任务输出的数据量大于其耗用的数据量 （例如，内插），则可使用 STREAM 指令增大 FIFO 的深度。

下表列出了尝试最大程度减少设计实现所使用的资源时可考量的其它指令。

表 4-5：最优化策略步骤 5

指令	描述
ALLOCATION	指定所使用的操作、核或函数的数量限制。这样会强制共享硬件资源并可能增大时延。
ARRAY_MAP	把多个较小的数组结合成单个大型数组以帮助减少块 RAM 资源数量。
ARRAY_RESHAPE	把数组从多元素数组重塑为字宽更宽的数组。用于在不增加使用的块 RAM 数量的前提下提升块 RAM 访问。
LOOP_MERGE	合并连续循环，以缩短总体时延、增加共享和提升逻辑最优化。
OCCURRENCE	在对函数或循环进行流水线化时使用，用于指定某个位置的代码执行速度低于外围函数或循环中的代码执行速度。

表 4-5：最优化策略步骤 5 （续）

指令	描述
RESOURCE	指定将特定的库资源 （核）用于实现 RTL 中的变量 （数组、算术运算或函数实参）。
STREAM	指定在数据流最优化期间把特定存储器通道实现为 FIFO 或 RAM。
Config Bind	判断综合绑定阶段的工作量，可用于在全局层面最大限度减少使用的运算数量。
Config Dataflow	该配置用于指定数据流最优化中的默认内存通道和 FIFO 深度。

ALLOCATION 和 RESOURCE 指令用于限制操作数量，选择哪些核 （或资源）供实现操作使用。例如，您可将函数或循环限制为只能使用 1 个乘法器，并指定使用流水线化乘法器来实现它。绑定配置用于在全局层面限制特定操作的使用。

重要提示：最优化指令仅在指定范围内适用。

如果 ARRAY_PARITION 指令用于提升启动时间间隔，可以考虑改为使用 ARRAY_RESHAPE 指令。ARRAY_RESHAPE 最优化执行的是与数组分区类似的任务，但重塑最优化功能会把数组分区创建的元素重新结合成含更宽数据端口的单个块 RAM。

如果 C 语言代码内含相似索引的一连串循环，那么使用 LOOP_MERGE 指令合并这些循环可以实现部分最优化。

最后，如果流水线区域内的某一段代码相比于该区域其它部分只需以更低的启动时间间隔运行即可，则可使用 OCCURENCE 指令来指示对该逻辑进行最优化，降低其运行速率。

最优化策略

最优化方法普遍适用于所有类型的 C 语言代码。获得高性能设计的关键最优化指令是 PIPELINE 指令和 DATAFLOW 指令。本节将详细讨论如何将这些指令应用于各类 C 语言代码架构。

从根本上来说，存在 2 种类型的 C 语言函数。分别是基于帧的 C 语言函数和基于样本的 C 语言函数。

这 2 种风格无论使用哪一种，所生成的 RTL IP 都几乎相同：区别在于应用最优化指令的方法。具体选择的风格由用户自行判断：建议使用最便于捕获您的描述的风格。

基于帧的 C 语言代码

基于帧的 C 语言代码的概要示例如下。这种编码风格的主要特点是该函数在每个传输事务中可处理多个数据样本 （1 帧数据），这里 1 个传输事务即表示完整执行 1 次 C 语言函数。

```
void foo(
  data_t in1[HEIGHT][WIDTH],
  data_t in2[HEIGHT][WIDTH],
  data_t out[HEIGHT][WIDTH] {


Loop1: for(int i = 0; i < HEIGHT; i++) {
  Loop2: for(int j = 0; j < WIDTH; j++) {
    out[i][j] = in1[i][j] * in2[i][j];
    Loop3: for(int k = 0; k < NUM_BITS; k++) {
  }
}
}
```

该数据一般以数组形式提供，但也可作为指针或 `hls::stream` 来提供。使用指针算法可多次访问指针。`hls::stream` 也可在函数内多次访问。

基于帧的编码风格的另一个特点是一般使用循环来访问和处理数据。上述代码是这种情况的经典示例。

在尝试流水线化任何 C 语言代码时，您应在处理数据样本的层级上布局流水线指令。根据此原则讨论上面示例中的每一个层级，有助于掌握布局流水线指令的最佳做法。

函数层级： 该函数接受数据帧作为输入 （in1 和 in2）。如果该函数以 II=1 进行流水线化，即每个时钟周期读取一组新输入，就会告知工具在单个时钟周期内读取 in1 和 in2 的所有 HEIGHT*WIDTH 值。

这种设计可能与您的期望不符。

在应用 PIPELINE 指令后，其下层层级中的所有循环 （即本例中 foo 以下的全部内容）都必须展开。这就是流水线化的要求：流水线内部不能出现顺序逻辑。这样会创建逻辑的 HEIGHT*WIDTH*NUM_ELEMENT 复本，形成一个大型设计。

数组可以作为多种类型的接口来实现：

- 块 RAM 接口 （默认）
- AXI 接口
- AXI4-Lite 接口
- AXI4-Stream 接口
- FIFO 接口

由于数据按顺序进行访问，作为 AXI4-Stream 接口、双向握手或 FIFO 接口。块 RAM 接口可实现为每个时钟周期提供 2 个样本的双端口接口。另一种接口类型每个时钟周期只能提供 1 个样本。这样就会产生瓶颈。

HLS 最优化方法的步骤 3 能够帮助您克服这一瓶颈。为了在同一时钟周期内访问所有数据值，该数组必须分区为单独的元素，以创建 HEIGHT*WIDTH 端口 （如果每个端口都是双端口块 RAM 接口，那就是该数量的一半），这样即可在同一时钟周期中读取所有端口。输出端口的情况类似。

注释： 如需了解有关最优化方法的更多信息，请参阅“最优化方法”。

这将是一种高度并行化的设计，但规模也很大。

Loop1 层级：Loop1 中的逻辑能处理二维矩阵中完整 1 行。把 PIPELINE 指令置于此处所创建的设计会尝试在每个时钟周期内处理 1 行。同样，这也将展开位于其下层的循环，并创建额外逻辑。

这种大规模并行设计比第一种更慢且规模更小。

Loop2 层级：循环中的逻辑会尝试处理数组中的 1 个样本。如果设计需每个时钟周期处理 1 个样本，则可在此层级执行流水线化。

这样会导致 Loop3 完全展开，但由于 Loop2 每个时钟周期处理 1 个样本，因此这是必须满足的要求，通常也是必要的。在典型设计中， Loop3 中的逻辑一般是移位寄存器或数据字内部的处理位。要在每个时钟周期内执行 1 个样本，这些进程应并行执行，因此需要将循环展开。

该设计每个时钟周期处理 1 个数据样本，并且仅在需要实现这种规模等级的数据吞吐量时才创建并行逻辑。

Loop3 层级：如上文所述，在此情况下 Loop3 中的逻辑一般会负责位级任务或数据移位任务，在此类层级上将对每个数据样本进行操作。例如，如果 Loop3 包含移位寄存器操作且 Loop3 已流水线化，就会告知工具每个时钟周期移位 1 个数据值。设计只会返回到 Loop2 中的逻辑且在所有样本移位完成后读取下一输入。

本例中理想的流水线化位置是 Loop2。

在处理基于帧的代码时，您应考虑在循环层级进行流水线化，并且一般情况下流水线化的对象为在样本层级进行操作的循环。如有疑虑，可在 C 语言代码中添加 1 条 print 命令，并使用 C 语言仿真来确认您是否希望在每个时钟周期内在此层级执行操作。

根据上文的详细说明，通常在基于帧的设计中， ARRAY_PARTITION 指令可用于将数组分区为较小的块（或对位于接口上的数组，可使用该指令将其分区为多个端口）以便消除性能瓶颈。

基于样本的 C 语言代码

基于样本的 C 语言代码的概要示例如下。这种编码风格的主要特征是函数在每个传输事务期间处理 1 个数据样本。

```
void foo (data_t *in, data_t *out) {

  static data_t acc;

  Loop1: for (int i=N-1;i>=0;i--) {
  acc+= ..some calculation..;
  }

  *out=acc>>N;
}
```

在基于样本的函数中，数据以标量、指针或 `hls::stream` 变量的形式提供。

指针或 `hls::stream` 在任一函数内可多次访问，但在基于样本的函数中则只能访问一次。

基于样本的编码风格的另一个特征是函数往往内含静态变量：多次调用函数 （例如，累加器或样本计数器）时，此变量的值必须保持不变。

要实现 II=1，即每个时钟周期读取一个数据值，该函数必须流水线化。这将展开所有循环并创建额外逻辑，但这是不可避免的。如果 `Loop1` 已流水线化，则至少需要 N 个时钟周期才能完成。完成后，函数才能读取下一个 x 输入值。

如果使用的 C 语言代码在样本层级执行操作，则最佳策略往往是把函数流水线化。由于基于样本的设计中的循环一般在执行移位寄存功能的数组上进行操作，因此一般会把这些数组分区为单独的元素以确保在一个时钟周期内完成所有样本的移位：否则移位操作会被限定为在双端口块 RAM 上读写样本。

这里的解决方法是对 `foo` 函数进行流水线化。这样即可得到每个时钟周期处理 1 个样本的设计。

RTL 验证

Vivado HLS 内部的 RTL 验证流程是完全自动运行的。在 RTL/C 语言协同仿真期间，将复用 C 语言仿真所使用的 C 语言测试激励文件，综合后的函数则由 RTL 设计替代。使用正确的接口协议进出 RTL 设计的数据的排序操作则是由 Vivado HLS 自动执行的。

由于复用 C 语言测试激励文件，因此无需创建 RTL 测试激励文件。

有部分设计选项可能会妨碍 RTL/C 语言协同仿真。要执行 RTL/C 协同仿真，必须符合以下条件。

- 顶层函数必须使用 `ap_ctrl_hs` 或 `ap_ctrl_chain` 块级接口加以综合。
- 或者必须采用纯组合型设计。
- 或顶层函数的启动时间间隔必须为 1。
- 或接口必须全部是使用 `ap_fifo`、 `ap_hs` 或 `axis` 接口模式来完成流传输和实现的数组。

如果上述任一条件未得到满足， C/RTL 协同仿真就会中止并提示以下消息：

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3) designs with
array streaming or hls_stream ports.
```

```
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

IP 封装

在设计完成之后，会使用 Vivado HLS 的“导出 RTL (Export RTL)”功能创建一个适用于 IP 目录的 IP 封装。对包含 AXI4-Lite 接口的设计，该 IP 封装内含有用于接口编程的必要软件驱动程序文件。

Vivado HLS 提供多种封装选项。要使用高效的 IP integrator 方法，应使用 IP 目录格式，同时接口应为 AXI 接口。

设计分析与最优化

任何设计方法的必备一环是使用高效的设计分析与改进流程。如需获取有关使用 Vivado HLS 进行 C 语言仿真、 C 语言调试、综合、 RTL 验证和 IP 封装的过程的详细描述，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

生成设计和提升设计性能的流程可总结为：

- 执行 C 语言代码仿真并验证设计是否正确。
- 执行初始设计综合。
- 执行设计性能分析。
- 创建新解决方案并添加最优化指令。
- 分析新解决方案的性能。
- 继续创建新解决方案和最优化指令，直至满足要求为止。
- 验证 RTL 是否正确。
- 把设计封装为 IP 并包含到您的系统中。

最高效的方法是使用 C 语言仿真来验证设计并在综合前确认结果是否正确。 C 语言仿真速度恰恰是高层次设计流程的主要优势所在。相比于费时费力调试性能问题结果却发现问题源于设置错误，确认 C 语言设计是否正确效率更高。

确保报告的实用性

在实现初步综合结果后，第一步是审核结果。如果综合报告内含任何未知值 （显示为问号“?”），就必须加以解决。要判断最优化指令是否能提升设计性能，关键在于比较解决方案：时延值必须已知才能进行比较。

如果循环有变量绑定， Vivado HLS 就不能判断完成循环所需的迭代数量。由于变量绑定的存在，即便一次循环迭代的时延已知， Vivado HLS 也不能判断完成此循环的全部迭代的时延。

审核设计中的循环。在综合报告中，审核“Latency > Details > Loops”部分中的循环。从循环层级中报告未知时延的最低层循环开始，因为这一未知值会沿层级向上传输。变量绑定循环可能位于层级中的较低层。审核综合报告的“Latency > Details > Instance”部分，查看是否有子函数显示为未知值。打开显示时延值未知的任何函数的报告，重复上述流程，直至找出变量绑定循环。

除了使用综合报告，还可以使用“分析透视图 (Analysis Perspective)”。

在找到变量绑定循环后，添加 LOOP_TRIPCOUNT 指令以指定该循环的迭代计数，或使用 C 语言代码中的断言来指定限值。如需了解更多信息，请访问此链接以参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

如果使用 LOOP_TRIPCOUNT 指令，可考虑将该指令作为 `pragma` 添加到源代码中，因为该指令在每个解决方案中都需要。

如果您发现其它循环有变量绑定，请为这些循环设定迭代限值，否则请重复综合，使用同样的自下而上方法，直至顶层报告包含真实的数值为止。

设计分析

设计分析可使用三种不同技巧来执行。

- 综合报告。
- 分析透视图。
- RTL 仿真波形。

提示：在分析结果之前，请查看控制台窗口或日志文件，查看已执行、已跳过或已失败的最优化。

综合报告和分析透视图可用于分析时延、时间间隔和资源估算。如果现在存在多个解决方案，请使用 GUI 中的比较报告按钮并列比较这些解决方案。该功能与分析透视图一样，只能在 GUI 中使用，但请谨记，在 GUI 中可使用 `vivado_hls -p project_name` 打开使用批处理模式创建的工程并加以分析。

此时层级方法一样有用。从顶层开始，判断哪些任务对时延、时间间隔或面积影响最大，然后深入核查这些任务。按递归方式沿层级向下查找，直至找到您认为能够或应该对实现您的目标发挥更理想性能的循环或函数。在改进这些函数或循环后，产生的改进效果会沿层级向上传递。

与综合报告相比，使用分析透视图更便于沿设计层级进行上下移植。此外，分析透视图还能针对与 C 语言代码交叉关联的调度操作和资源使用情况提供详细视图。

在深入探查细节之前，最好先使用分析透视图中的详细调度视图查看宏层级行为，这是很有用的。这些操作一般按 C 语言代码的执行次序列出。请谨记， Vivado HLS 会尝试把所有内容都调度到时钟周期 1 中，并争取在 1 个时钟周期内完成。

- 如果看到操作总体呈现从左上到右下漂移，原因可能是代码中固有的数据依赖关系或任务执行顺序造成的。每项操作都必须在前一项操作完成后才能开始。
- 如果看到操作调度为逐一顺序执行，然后突然发生大量操作同步执行 （或与此相反的情况），说明可能存在瓶颈（比如 I/O 端口或 RAM 端口）导致设计必须持续等待，随后所有操作并行执行。

除了综合报告和分析透视图，也可使用 RTL 仿真波形帮助分析设计。在 RTL 验证期间可以保存走线文件并使用合适的查看器查看。请参阅 《Vivado Design Suite 教程：高层次综合》 (UG871) [参照 3] 中的 RTL 验证教程部分，以获取详细

信息。此外，导出 IP 封装并在 `project_name/solution_name/impl/ip/verilog` 或 `vhdl` 文件夹中打开该 Vivado RTL 工程。如果已执行 C/RTL 协同仿真，该工程会包含一个 RTL 测试激励文件。

使用 RTL 进行设计分析时务必谨慎操作。如果更改 C 语言代码或添加最优化指令，在重新执行综合后，得到的 RTL 设计及其名称可能发生变化。每次有新设计生成都需要花时间重复了解 RTL 细节，而且会使用显著不同的名称和结构。

总之，建议沿层级向下查找可供进一步最优化的任务。

设计最优化

在开展任何最优化之前，建议在工程中创建新的解决方案。使用解决方案可以比较不同的结果。不仅可以比较结果，还可以比较日志文件乃至输出 RTL 文件。

高性能设计的基本最优化策略是：

- 创建初始或基准设计。
- 流水线化循环和函数。
- 解决限制流水线化的任何问题，例如数组瓶颈和循环依赖关系 （使用 ARRAY_PARTITION 和 DEPENDENCE 指令）。
- 应用 DATAFLOW 最优化以同时执行循环和函数。
- 有时可能需要对代码进行调整以满足性能要求。
- 缩小数据流存储器通道的大小并使用 ALLOCATION 和 RESOUCE S 指令进一步缩小面积。

总之，目标是始终优先满足性能，其次是缩小面积。如果策略的目的是利用更少的资源创建设计，只需要忽略用于提升性能的步骤即可。

在整个优化流程中，强烈建议在综合后审核控制台输出 （或日志文件）。如果 Vivado HLS 无法实现最优化的指定性能目标，它会自动放宽目标 （时钟频率除外）并以能够满足的目标创建设计。重要的是，审核综合的输出以了解已完成的最优化。

如需了解有关应用最优化的具体详情，请参阅 《Vivado Design Suite 用户指南：高层次综合》 (UG902) [参照 2]。

XILINX

第 5 章

系统集成

简介

就像 shell 设计开发一样，高效系统集成方法的关键也同样是利用 Vivado® IP 目录 (IP Catalog) 和 IP 集成器 (IP integrator)。遵循前文中高层次设计方法的步骤的情况下，该阶段的设计进程包含以下几项：

- 经预先验证的 shell。板级接口已开发完成并验证成功。
- 一批已封装 （供 Vivado IP 目录使用）的 IP 块，这些 IP 块经过预先验证，确认可提供设计核心功能。
- 用于 IP 级接口的 AXI 接口，支持利用 IP integrator 的设计辅助功能来自动创建设计。
- 已创建好的用于验证 shell 的系统级测试激励文件。

并行开发和验证的系统组件现已准备就绪，可用于系统集成。

完成初始系统集成后，您现已具备自动执行这整个流程并轻松生成更多新设计所需的一切。

初始系统集成

设计集成进程可概括如下：

1. 基于 shell 设计创建新的 Vivado 工程
2. 添加所有 IP 块，并使用 IP integrator 连接 IP
3. 验证系统并处理设计，将其实现至 FPGA 比特流。

系统集成工程

使用如下所示任一参考 shell 设计创建新的系统设计集成工程：

1. 打开 Vivado 工程的 shell 设计并选择“File > Save Project As”以在新工程中保存此 shell 设计。
2. 创建一个新的 Vivado RTL 工程 （无 RTL 源代码）并选择相同的目标器件或开发板。然后选择“Create Block Design”，并在控制台中找到使用 `write_bd_tcl` 保存的 Tcl 脚本，将其作为源文件用于在新工程中重新生成 shell 块设计。
3. 将所有核设计 IP 块都添加到工程 IP 存储库中。

自动执行的系统集成

IP integrator 可提供设计辅助功能以帮助完成系统集成。建议打开 shell 设计并将所有 IP 块都添加到画布中。如果 AXI 接口已全部用于所有 IP 和 shell 接口，设计辅助功能将被激活，并提供连接建议。

设计辅助功能会识别合法连接。除了简单自动建立连接外，它还可以自动添加任何必需的 AXI 互连逻辑，例如将 AXI4-Stream 接口连接到 AXI 内存映射端口。如需获取设计辅助功能的完整描述，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) [参照 8]。

培训：如果在设计中使用了多个时钟域，请参阅 《Vivado Design Suite QuickTake 视频：在 Vivado IP 集成器中使用多个时钟域》。

为设计辅助功能不支持的对象 （如标量信号以及任何非 AXI 总线接口等）建立连接，从而完成块设计。

最后，使用验证设计功能来确保设计没有设计规则违例。如需了解使用内存映射接口或处理器的设计的相关信息，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) [参照 8]。

当设计完成并成功验证后，赛灵思建议您使用 `write_bd_tcl` 命令。`write_bd_tcl` 命令可以在 Tcl 文件中保存重新生成整个系统所需的所有内容。

系统验证与实现

在 IP integrator 中完成系统块设计后，您可以通过生成输出文件和为设计创建顶层 HDL 封装程序来开展整个系统的验证和实现。如需了解上述使用 IP integrator 方式的完整流程，请参阅 《Vivado Design Suite 教程：高层次综合》 (UG871) [参照 3] 中的下述章节：

- 第 9 章“在 IP integrator 中使用 HLS IP”
- 第 10 章“在 Zynq SoC 设计中使用 HLS IP”

现在，即可使用专为 shell 验证而创建的 RTL 测试激励文件来执行系统级验证。 IP 块和 shell 设计等各个部分均已单独经过预验证。现在的任务就是验证整个系统。在初始阶段，请您通过下列方式重点确认系统级连接：

1. 确保 shell 为处理流水线中第一个块的输入提供的数据正确无误。
2. 确保第一个块向下一块提供的输出正确无误。

注释：使用最小的数据量，确保系统级仿真尽可能快速运行。这时请您先集中精力确认块连接。

验证顶层连接后，便可执行完整详尽的系统仿真。

重要提示：此高级方法可提供一种有效途径来对未通过系统级验证的任何 IP 快速进行重新设计，然后以高度脚本化的方式快速重新生成整个系统，详情请参阅“自动执行的系统集成”。

在系统完全完成验证后，便可将设计实现至比特流。如果大部分设计 IP 都是使用 Vivado HLS 通过 C/C++ 创建的，那么 RTL 已经自动完成时序收敛，以确保 RTL 综合期间可满足时序要求。

注释：经过 RTL 综合后，如果 IP 块之间存在不满足时序要求的时序路径，请考虑在 HLS 期间使用 `INTERFACE` 指令来寄存接口端口。

为了缩短运行时间，建议在生成输出文件时使用“Out of Context per IP”模式。该选项可生成一个经综合并缓存的输出文件，仅当 IP 块发生更改的情况下才会重新执行综合，从而缩短系统实现时间。

自动执行的系统集成

虽然只在系统集成阶段才执行完整的系统级验证，这一点或许令人担忧，但实际上它正是这种方法的优势之一。一次完整的 RTL 系统级仿真可能耗时较长，在工程开发中执行多次这样的仿真会占用大量工程开发时间，这通常是开发过程中最耗时的任务。

此方法将重点放在：

- 并行开发
- 使用 C/C++ 仿真来验证设计 IP，以使验证速度提升多个数量级
- 创建并验证块级 IP
- 复用现有已验证的 IP

由于 Vivado Design Suite 的高度自动化，使得这一方法卓有成效。本节内容展示了如何使用 Tcl 脚本快速轻松重新创建整个系统，即使系统集成过程中发现了问题也不会有影响。

Vivado 工程自动化

在 Vivado IDE 中执行的所有操作均以 Tcl 命令的形式捕获到工程日志文件中。在批处理模式下，这些命令允许您重复所有操作，从而显著缩短执行任务所需时间。通过此方法，利用这种自动生成 Tcl 命令的方式可支持实现下列任务的高度自动化：

- 创建工程
- 在工程中添加 IP
- 系统仿真
- 系统实现

以下代码样本演示了如何轻松自动完成创建项目、在工程中添加 IP 存储库、创建块设计、将工程处理至比特流的流程。

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0

# Set the paths to auto-adjust to the local directory
# Define project and IP repository locations
set my_files [pwd]
set projdir $my_files/project_1
set repo_dir $my_files/../my_ip/ip
puts "Using project directory $projdir"
puts "Using repository directory $repo_dir"

# Create the Project
set projname project_1
create_project -force $projname $projdir -part $my_part
set_property board_part $my_board_part [current_project]

# Create IP repository
set_property ip_repo_paths $repo_dir [current_fileset]
update_ip_catalog -rebuild

# Create the block design
source ./design_IPI.tcl
```

```
# Create output products and HDL wrapper
generate_target all [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd]
make_wrapper -files [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd] -top
add_files -norecurse
$projdir/$projname.srcs/sources_1/bd/$design_name/hdl/${design_name}_wrapper.v
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

# Implement the bitstream
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
```

可直接从工程日志文件复制用于执行上述操作的 Tcl 命令。或者，可以使用“Save > Write Project Tcl”在 Vivado 中轻松生成流程脚本，并自动执行 shell 创建、 shell 验证和系统集成的流程。

Vivado HLS 自动化

Vivado HLS 使用 IDE 为每个工程创建 1 个 Tcl 文件。以下代码样本演示了适用于 C 语言设计流程中下列所有步骤的 Tcl 命令：仿真 C 语言代码、使用最优化指令综合 C 语言代码、验证 RTL 是否正确、创建 IP 封装，以及确认 RTL 综合满足时序要求。

```
# Create a project and add files
open_project proj_matrixmul
set_top DESIGN_TOP
add_files matrixmul.cpp
add_files -tb matrixmul_tb.cpp

# Create a solution
open_solution "fast"
set_part {xc7z020clg484-1}
create_clock -period 4 -name default

# Add optimization directives
set_directive_pipeline "cholesky/"
set_directive_array_reshape -type complete -dim 2 "matrixmul" a
set_directive_array_reshape -type complete -dim 1 "matrixmul" b

# Simulate, Synthesize, Verify and package outputs
csim_design
csynth_design
cosim_design
export_design -format ip_catalog -evaluate verilog
```

可通过编辑此自动生成的 Tcl 文件来为 C 语言 IP 开发流程的任何部分实现自动化。例如，可以创建仅执行 C 语言仿真的脚本。验证设计后，如上所示完整脚本即可用于将设计综合成为封装 IP。

IP integrator 自动化

IP integrator 的 `write_bd_tcl` 命令不仅可保存 Tcl 脚本以重现您的操作，同时也会对脚本进行最优化，使其仅创建最终块设计。只需执行该脚本即可重新创建块设计。由于使用 IP 存储库中的 IP 重新创建块设计，因此只要此 IP 更新，就会使用最新的 IP 重建设计。这种程度的自动化使您能够快速重建块设计：

- 可在新设计工程中重新生成 shell 设计、执行修改，并轻松创建新 shell。
- 可在新验证工程中重新生成 shell 设计，并将验证 IP 轻松添加到设计中。
- 可在系统集成工程中重新生成 shell 设计，并将核设计 IP 集成到系统中。

整个方法中的每个步骤都能以高效方式重新执行。

完整系统自动化

通过使用 Makefile 执行脚本即可进一步提升效率。 Makefile 能指定一系列依赖关系。例如，下列任务必须依序执行：

- 任务 A：在 C 语言中仿真 IP。
- 任务 B：在 IP 目录中综合 IP。
- 任务 C：在系统级集成 IP。

当使用 Makefile 来执行任务 C 时，它会进行自动检查，以确定是否存在来自任务 B 的输出。如果输出不存在，它将尝试执行任务 B，并检查是否存在来自任务 A 的输出，以此类推。

使用 Makefile 来执行 Tcl 脚本同上述流程类似，因此，可以直接更新任何 IP 或 shell 设计，并发出一条命令来重新创建整个系统，需执行下列任一操作时，此命令将停止执行：

- 复查 C 语言仿真的结果
- 重新创建 shell 设计以添加验证 IP
- 重新综合 IP、通过 RTL 仿真来验证 IP 以及重新构建系统。
- 完成 FPGA 编程后。

这种程度的自动化可将此方法中所有一切都链接到单一高效的流程中，也正是因为这个原因，此方法的各部分才能并行执行并等待至系统集成完成后再执行系统级仿真。创建系统的第一个版本后，系统的整个生成过程将完全自动化。

面向未来的设计

使用高效设计方法的最后一个优势是可从初始设计轻松创建衍生设计。这一效率提升主要是由两大因素促成的：

- 使用 C 语言开发 IP
- 采用自动化实现流程

使用 C 语言开发 IP

除了本指南中所概述的使用 C 语言进行 IP 开发的主要优势之外，另一个优势就在于轻松实现设计重定向：根据相同来源创建衍生设计。

在上文 Vivado HLS 脚本示例中，对于含 250 MHz 时钟的 Zynq®-7000 SoC 器件采用了以下 Tcl 命令：

```
set_part {xc7z020clg484-1}
create_clock -period 4 -name default
```

用于创建时序精确的 IP 块，以便在其中完成设计实现，例如，对于 300 MHz 的 Kintex® UltraScale™ 器件，只需更新最优化约束即可：

```
set_part {xcku025-ffva1156-2-i}
create_clock -period 300MHz -name default
```

此外无需任何改动。 Vivado HLS 只需创建按目标技术的选定频率实现的设计即可。设计采用更快的技术的情况下只需更少的时钟周期就能完成 （反之，采用更慢的技术，则需要更多的时钟周期），但无需重新编码或重新最优化。

使用 C 语言代码创建的内容越多，设计也就更易于应用于新技术和/或时钟频率。原有 RTL 块可能仍需重新实现，以适应不同的时序参数。

自动执行的实现流程

如果通过完全脚本化的流程来实现 FPGA，则通过脚本参数化来创建衍生设计即可进一步改善设计复用并提升效率。

上述脚本示例中使用了下列代码。此示例使用了 Vivado 脚本，通过 Vivado HLS 脚本和 IP integrator 脚本也可以实现同样的改进效果。

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0
```

如需仅用一个顶层工程参数脚本来完成工程中的所有设置，请更改先前代码以匹配下列内容：

```
# source project-level parameters
source project_top.tcl
# Set project parameters
set my_part $target_device
set my_board_part $target_board
```

本示例中 `project_top.tcl` 的内容为：

```
set target_device xc7z020clg484-1
set target_board xilinx.com:zc702:part0:1.0
```

修改这一脚本会使工程以高度自动化的方式进行重新定位并重新实现。

其它相关注意事项：

- 如果您遵循 C 语言测试激励文件的有关建议，则将自动检查 C 语言仿真。
- Vivado HLS 会基于新参数生成新 IP。
- 由 Vivado HLS 创建的 RTL 将通过 RTL 仿真来自动验证。
- Vivado 工程生成脚本将基于更新参数创建一个新项目。
- IP integrator 脚本像之前一样建立块连接，然后设计自动化功能将使用最合适的连接。
- 您可以在系统集成步骤选择暂停流程，然后修改设计。
- 已完成的系统将使用更新的目标器件和时钟频率实现到比特流。

增强 Tcl 脚本以适应参数化程度，这样将显著提升您创建统包性衍生设计的能力。

XILINX

附录 A

附加资源与法律声明

赛灵思资源

如需了解答复记录、技术文档、下载以及论坛等支持性资源，请参阅赛灵思技术支持。

解决方案中心

如需获取设计周期各阶段有关器件、软件工具和 IP 等的技术支持，请参阅赛灵思解决方案中心。相关专题包括设计辅助、建议和故障排除提示等。

Documentation Navigator 与设计中心

赛灵思 Documentation Navigator (DocNav) 提供了访问赛灵思文档、视频和支持资源的渠道，您可以在其中筛选搜索信息。打开 DocNav 的方法：

- 在 Vivado IDE 中，单击“Help > Documentation and Tutorials”。
- 在 Windows 中，单击“Start > All Programs > Xilinx Design Tools > DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

赛灵思设计中心 (Xilinx Design Hubs) 提供了根据设计任务和其它话题整理的文档链接，您可以使用这些链接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。
- 在赛灵思网站上，查看设计中心页面。

参考资料

1. 《采用 Vivado® 高层次综合开展 FPGA 设计的简介》 (UG998)
2. 《Vivado Design Suite 用户指南：高层次综合》 (UG902)
3. 《Vivado Design Suite 教程：高层次综合》 (UG871)
4. 《Vivado Design Suite 用户指南：版本说明、安装和许可》 (UG973)
5. 《Vivado Design Suite 教程：创建和封装定制 IP》 (UG1119)

6. 《Vivado Design Suite 用户指南：采用 System Generator 开展基于模型的 DSP 设计》 (UG897)
7. 《UltraFast 设计方法指南 （适用于 Vivado Design Suite）》 (UG949)
8. 《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》 (UG994)
9. 《Vivado Design Suite 用户指南：系统级设计输入》 (UG895)
10. 《使用 Vivado IP integrator 集成基于 AXI4 的 IP 的方法》 (XAPP1204)
11. Vivado Design Suite 技术文档
12. 赛灵思 IP 页面

培训资料

赛灵思提供多种多样的培训课程和 QuickTake 视频，可帮助用户进一步了解有关本文档中提出的概念。使用以下链接获取相关培训资料：

1. 基于 C 语言的设计：采用 Vivado HLS 工具开展高层次综合培训课程
2. 基于 C 语言的 HLS 编码硬件设计人员培训课程
3. 基于 C 语言的 HLS 编码软件设计人员培训课程
4. Vivado Design Suite QuickTake 视频教程
5. Vivado Design Suite QuickTake 视频：Vivado 高层次综合
6. Vivado Design Suite QuickTake 视频：Vivado 高层次综合入门
7. Vivado Design Suite QuickTake 视频：验证 Vivado HLS 设计
8. Vivado Design Suite QuickTake 视频：创建不同类型的工程

请阅读：重要法律提示