采用 Vivado 高层次综合开展 FPGA 设计的简介

UG998 (v1.1) 2019 年 1 月 22 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
2019 年 1 月 22 日 1.1 版	
常规更新	编辑更新。
“DSP 块”	更新有关 DSP 块的信息。
“存储元件”	新增有关 UltraRAM 的信息。
2013 年 7 月 2 日 1.0 版	
初始赛灵思版本。	不适用

目录

XILINX

第 1 章

引言

简介

软件是所有应用的基础。现如今，无论是娱乐、游戏、通信还是医药，人们日常使用的大量产品都始于软件模型或原型设计。软件工程师的使命就是根据系统的性能和可编程性限制来判断将项目投向市场的最佳实现平台。为了履行这一使命，软件工程师需要借助编程技巧以及各种硬件处理平台的帮助。

编程方面数十年的发展使面向对象的编程代码复用以及并行计算范例的算法性能双双取得了质的飞跃。编程语言、框架和工具的进步使软件工程师能够快速完成原型设计，并测试多种不同方法以解决具体问题。这种对于快速设计解决方案原型的需求带来了 2 个有趣的问题。第 1 个问题是如何分析和量化各种算法，这个问题在其它作品中已经得到了广泛的讨论，在本指南中就不再赘述了。第 2 个问题是在何处执行算法，在本指南中将从现场可编程门阵列 (FPGA) 的角度来探讨这个问题。

对于在何处运行算法的讨论正逐渐向并行化和并发化的方向靠拢。虽然以并行和并发方式执行软件程序的想法由来已久，但在处理器和应用专用集成电路 (ASIC) 设计领域的某些趋势的推波助澜下，这个话题又再度引发了关注。为了进一步提升软件算法的性能，软件工程师曾面临 2 个选择：定制集成电路或 FPGA。

第一种选择最为昂贵，即将算法交给硬件工程师，进行定制电路实现。这个选择的成本基于如下要素：

- 电路制造成本
- 将算法转换为硬件的时间成本

尽管制造工艺阶段技术在功耗、计算吞吐量和逻辑密度方面已经实现了长足的发展，但为应用制造定制集成电路或 ASIC 的成本仍居高不下。在每个工艺处理节点上，制造成本不断增加，导致从经济角度只有交付数量达百万级的应用才值得一试。

第二个选择是使用 FPGA，它可解决 ASIC 制造方面固有的成本问题。FPGA 支持设计人员使用由基本可编程逻辑元件组成的现成组件来创建算法的定制电路实现。此平台可节省功耗，发挥小型制造节点的性能优势，同时避免 ASIC 开发工作中所伴随的成本和复杂性。与 ASIC 一样，以 FPGA 实现的算法可以发挥定制电路的固有并行性质。

模型编程

硬件平台的编程模型是推动其采用背后的主要推手之一。软件算法通常是以 C/C++ 或其它高级语言采集的，此类语言可提炼计算平台的详细信息。这些语言支持快速迭代、增量改进和代码可移植性，这些特征对于软件工程师都是至关重要的。最近数十年来，这些语言所采集的算法的快速执行助推了处理器和软件编译器的发展。

最初，软件运行时间方面的提升是围绕 2 个核心概念来实现的：增加处理器时钟频率和使用专用处理器。多年来，一般都需要等待一整年才能开发出全新一代的处理器作为加速执行的途径。随着全新时钟频率不断提升，软件程序运行速度也越来越快。对于大量应用而言，通过处理器时钟频率实现速度递增虽然在某些情况下是可接受的方法，但并不足以向市场交付可行的产品。

对于这类应用，专用处理器应运而生。虽然数字化信号处理器 (DSP) 和图形处理单元 (GPU) 之类的专用处理器类型多种多样，但所有这些处理器都能够执行以高级语言 （例如 C 语言）编写的算法，并且具备专用功能的加速器，用于改善其目标软件应用的执行。

随着标准处理器和专用处理器设计范例方面最近的转变，这两类处理器都已不再依靠提升时钟频率来加速程序，并且都双双提升了每个芯片的处理核数。多核处理器将程序并行性作为提升软件性能的主要手段。软件工程师现在必须按能够提升并行性以换取性能提升的方式来构建算法。算法设计中所需的技巧使用的正是 FPGA 设计的基本要素。 FPGA 与处理器之间的主要区别在于编程模型。

曾经，FPGA 的编程模型的核心是寄存器传输级 (RTL) 描述，而并非 C/C++。这种设计采集模型与 ASIC 设计完全兼容，它类似于软件工程设计中的汇编语言编程。图 1-1 显示了采用 RTL 作为设计采集方法的传统 FPGA 设计流程，其中展示了编程模型差异对于不同计算平台的实现时间和可实现的性能的影响。

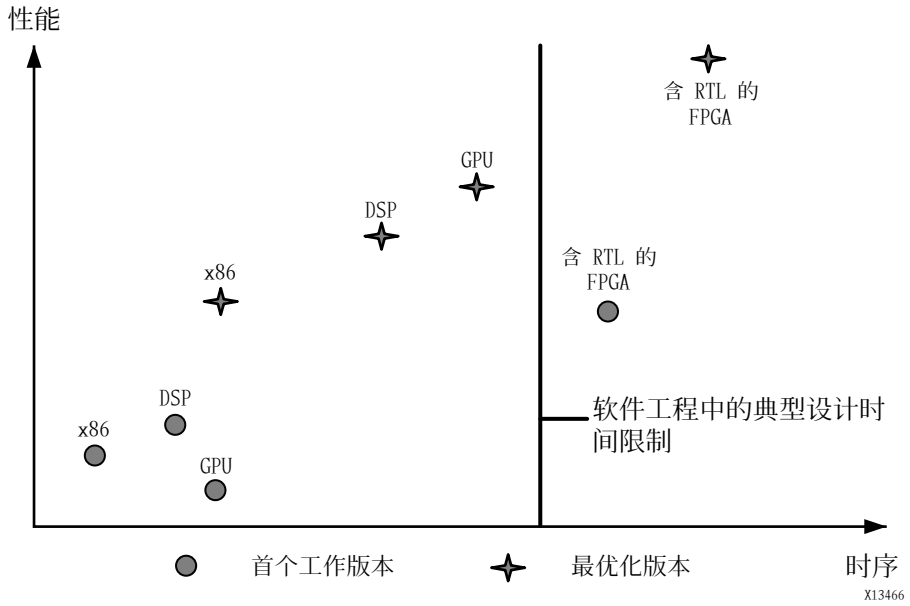


图 1-1：利用 RTL 设计输入的设计时间与应用性能对比

如图 1-1 中所示，对于项目设计周期而言，标准处理器和专用处理器完成软件程序的初始工作版本都相对较快。完成初始可运行版本后，还必须开展额外的开发工作才能在任何实现平台上获得最大限度的性能。

此图还显示了在 FPGA 平台上开发相同软件应用所需的时间。相比于同阶段的标准处理器和专用处理器，该平台上的应用的初始版本和优化版本的性能优势明显。采用 RTL 编码的 FPGA 优化应用能够达成最高性能的实现。

但达成此实现所需的开发时间超出了典型软件开发工作的范围。因此，只有当应用的性能要求无法通过任何其它方式来满足时， FPGA 方可一展身手，例如，具有多个处理器的设计。

最近赛灵思所取得的技术进步彻底消除了处理器与 FPGA 之间的编程模型差异。就像 C 语言和其它高级语言为不同处理器架构所提供的编译器一样，赛灵思 Vivado® 高层次综合 (HLS) 编译器能为以赛灵思 FPGA 为目标的 C/C++ 程序提供同样的功能。图 1-2 将 Vivado HLS 编译器的结果与软件工程师可用的其它处理器解决方案进行了对比。

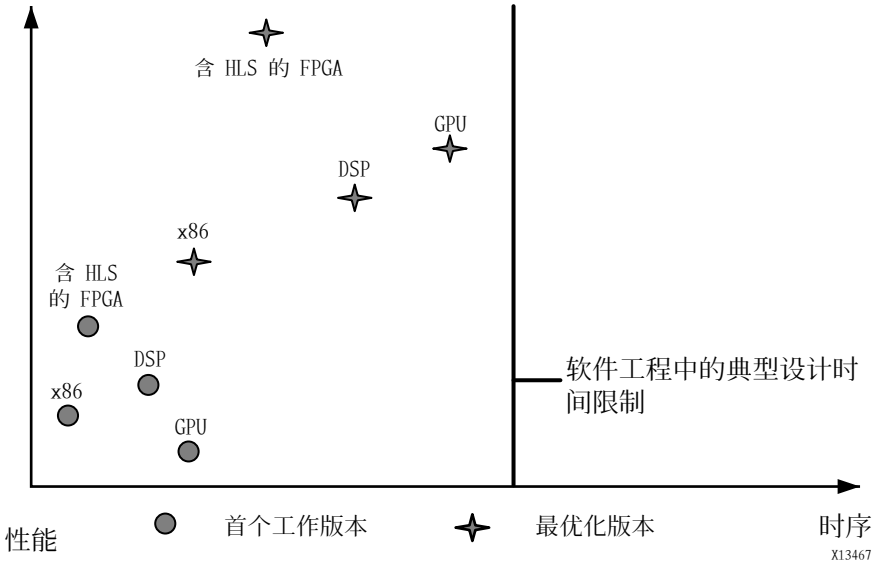


图 1-2：利用 Vivado HLS 编译器的设计时间与应用性能对比

本指南的组织结构

针对相同的 C/C++ 应用， FPGA 与其它处理器之间存在明显的性能差异。本指南中的下列章节描述了这种显著的性能差异背后的原因，并介绍了 Vivado HLS 编译器的工作方式。

第 2 章：关于 FPGA

第 2 章"关于 FPGA"介绍了 FPGA 中可用的计算元件及其与处理器的比较。其中涵盖了诸如 FPGA 内存层级、逻辑元件以及这些元件的相互关联方式等话题。

第 3 章：硬件设计的基本概念

处理器与 FPGA 之间的硬件差异会影响对应每个目标的编译器的工作方式。第 3 章"硬件设计的基本概念"涵盖了适用于 FPGA 设计和基于处理器的设计的基本硬件概念。理解这些概念将有助于设计人员指引 Vivado HLS 编译器创建出最佳的处理架构。

第 4 章：Vivado 高层次综合

第 4 章"Vivado 高层次综合"介绍了赛灵思 Vivado HLS 编译器。本章使用前两章中的概念描述了为 FPGA 编译 C/C++ 程序的方式。本章聚焦于编译器在 FPGA 内提炼并行能力、组织内存并连接多个程序的方式。

第 5 章：围绕计算的算法

虽然有关算法分析早已存在大量文献，但围绕计算的算法与围绕控制的算法之间的细微差异在很大程度上依赖于实现平台。第 5 章"围绕计算的算法"提供了适用于 FPGA 的围绕计算的算法的定义，并提供了示例和最佳实践建议。

第 6 章：围绕控制的算法

围绕控制的算法可在处理器和 FPGA 上实现。所选实现方法取决于算法所需的反应时间。第 6 章"围绕控制的算法"提供了围绕控制的算法的实现选项综述，并提供了用户数据报协议 (UDP) 包处理的联网示例。

第 7 章：软件验证和 Vivado HLS

就像所有其它编译器一样， Vivado HLS 编译器输出的质量与正确性取决于输入软件。第 7 章"软件验证和 Vivado HLS"对推荐适用于 Vivado HLS 编译器的软件质量技巧进行了回顾。它提供了典型编码错误示例及其对于 Vivado HLS 编译的影响，以及每个问题的可能的解决方案。其中还包含一个有关在 C 级无法完全验证程序行为时的变通方法的段落。

第 8 章：多个程序的集成

就像大部分处理器都会运行多个程序来执行应用一样， FPGA 同样可以构建多个程序或模块来执行特定应用。第 8 章"多个程序的集成"描述了如何在 FPGA 中连接多个模块以及如何通过单一处理器来控制这些模块。其中着重介绍了赛灵思 Zynq®-7000 片上系统 (SoC)，该系统将 FPGA 结构与 Arm® Cortex™-A9 处理器结合在一起。借助使用者和生产者示例，本章还展示了完整的系统开发、集成和设计取舍。

第 9 章：完整应用的验证

对于 FPGA，完整应用会创建一个硬件系统。此系统的 FPGA 结构中可包含 1 个或多个模块以及在处理器上执行的代码。第 9 章"完整应用的验证"提供了建议和最佳实践，以确保正确执行目标应用。

XILINX

第 2 章

关于 FPGA

简介

FPGA 是一种集成电路 (IC)，制造完成后即可针对不同算法进行编程。现代化 FPGA 器件包含多达 200 万个逻辑单元，可通过配置来实现各种软件算法。虽然传统 FPGA 流程更类似于常规 IC 而不是处理器，但 FPGA 相比于 IC 开发工作可提供显著的成本优势，并且大部分情况下可提供同等级的性能。相比于 IC，FPGA 的另一大优势就是它具备动态重配置能力。此流程就像在处理器中加载程序一样，可以影响 FPGA 结构中可用的部分或全部资源。

使用 Vivado® HLS 编译器时，重要的是了解 FPGA 结构中有哪些可用资源以及这些资源彼此间以何种方式进行交互来执行目标应用。本章提供了有关 FPGA 的基本信息，这些信息是指导 Vivado HLS 为任意算法提供最佳计算架构所不可或缺的。

FPGA 架构

FPGA 的基础结构由以下元件组成：

- **查找表 (LUT)**：该元件负责执行逻辑运算。
- **触发器 (FF)**：此寄存器元件用于存储 LT 的结果。
- **线路**：这些元件用于将元件彼此相连。
- **输入/输出 (I/O) 焊盘**：这些物理端口负责在 FPGA 上输入和输出数据。

这些元件的组合构成了 FPGA 基础架构，如图 2-1 中所示。虽然此结构已足以实现任意算法，但生成的实现的效率受到计算吞吐量、必需资源和可实现的时钟频率的限制。

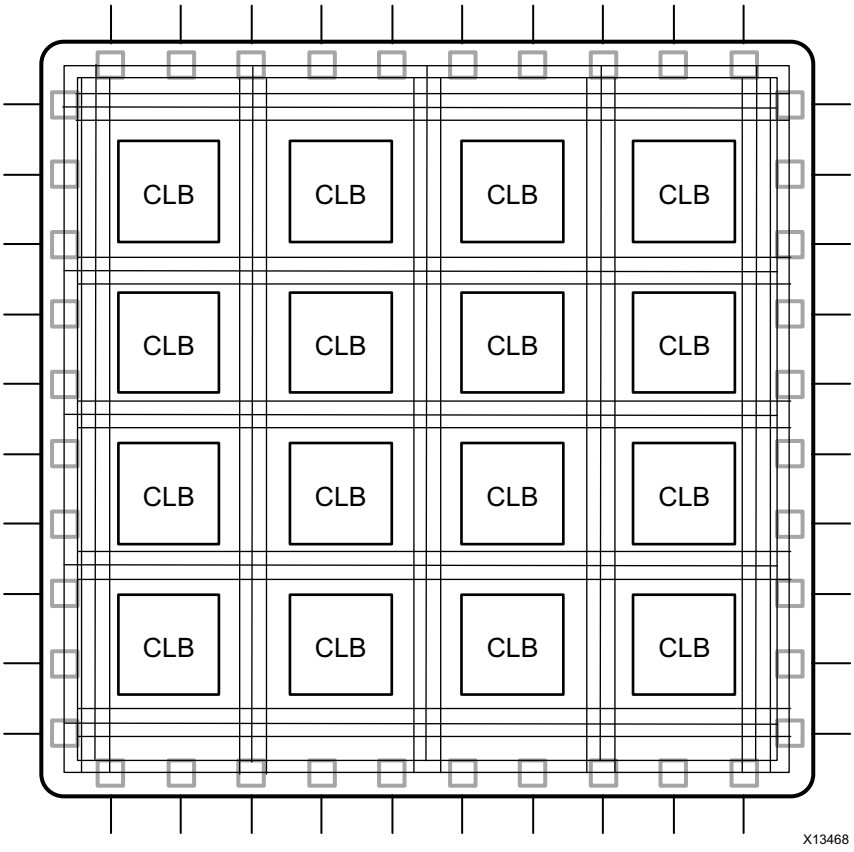


图 2-1：FPGA 基础架构

当代 FPGA 架构将基础元件与其它旨在提高器件的计算密度和效率的计算和数据存储块整合在一起。后文将对这些额外元件进行介绍，具体包括：

- 分布式数据存储器的嵌入式内存
- 用于以不同时钟速率来驱动 FPGA 结构的锁相环 (PLL)
- 高速串行收发器
- 片外内存控制器
- 乘法累加块

通过组合这些元件使 FPGA 得以灵活实现处理器上运行的任意软件算法，并生成了当代 FPGA 架构，如图 2-2 中所示。

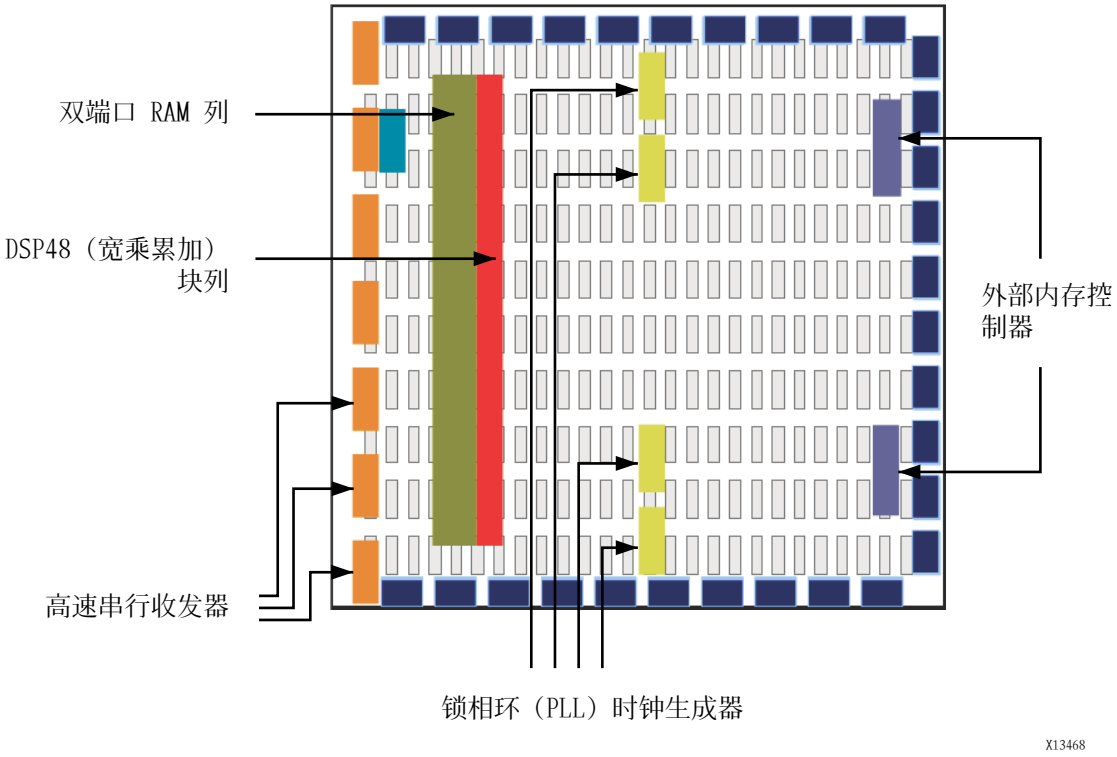


图 2-2：当代 FPGA 架构

LUT

LUT 是 FPGA 的基本构建块，能够实现 N 个布尔值变量的任意逻辑功能。本质上，此元件是一个真值表，其中通过输入的不同组合来实现不同功能，从而生成各种输出值。真值表的大小限制为 N，其中 N 表示 LUT 的输入数量。对于常规 N 个输入的 LUT，该表访问的内存位置数量为：

$$2^N$$ 等式 2-1

允许该表实现的功能数量为：

$$2^{N^N}$$ 等式 2-2

注释：赛灵思 FPGA 器件中 N 的典型值为 6。

LUT 的硬件实现可视作为连接到一组多路复用器的内存单元集合。LUT 的输入充当多路复用器上的选择器位，以便在给定时间点选择结果。时刻牢记此表达式至关重要，因为 LUT 既可用作为功能计算引擎，也可用作为数据存储元件。图 2-3 显示了 LUT 的此功能表达式。

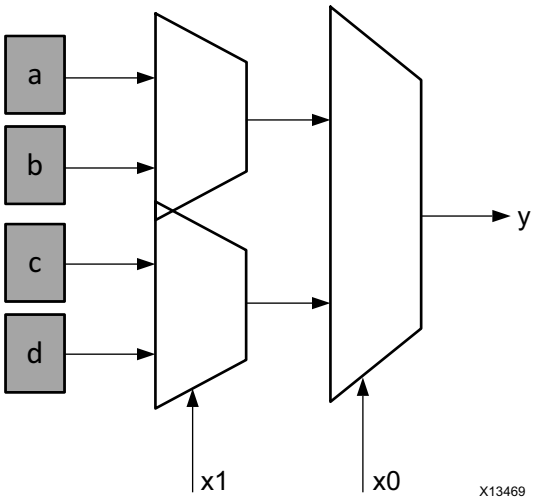


图 2-3：LUT 作为内存单元集合的功能表达式

触发器

触发器是 FPGA 结构中的基本存储单元。此元件始终与 LUT 配对以辅助执行逻辑流水线化和数据存储。触发器的基本结构包括数据输入、时钟输入、时钟使能、复位和数据输出。正常运行期间，数据输入端口处的任意值均处于锁存状态并传递到时钟的每个脉冲上的输出。时钟使能管脚的用途是允许触发器保存特定值以供多个时钟脉冲使用。当时钟和时钟使能都等于 1 时，新数据输入仅锁存并传递至数据输出端口。图 2-4 显示了触发器的结构。

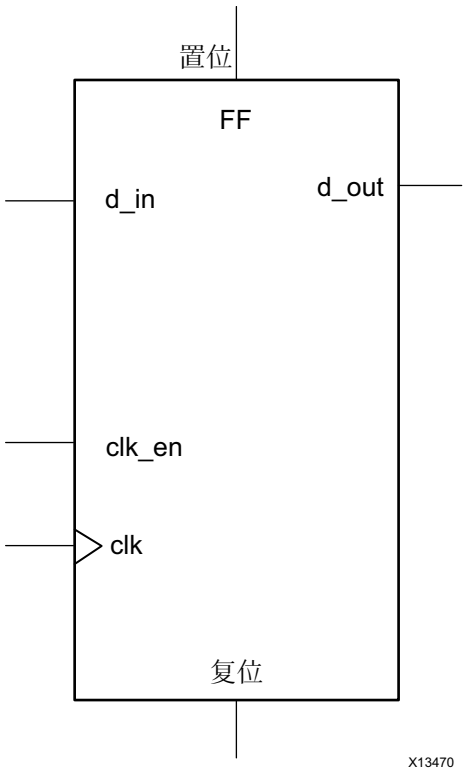


图 2-4：触发器结构

DSP 块

赛灵思 FPGA 中提供的最复杂的计算块为 DSP 块，如图 2-5 中所示。 DSP 块是嵌入 FPGA 结构的算术逻辑单元 (ALU)，由 3 个不同的块链接而成。DSP 中的计算链由加法/减法单元组成，这些加法/减法单元连接到乘法，最后再连接到加法/减法/累加引擎。这条计算链支持单一 DSP 单元实现如下形式的功能：

$$p = a \times (b + d) + c \quad \text{等式 2-3}$$

或

$$p += a \times (b + d) \quad \text{等式 2-4}$$

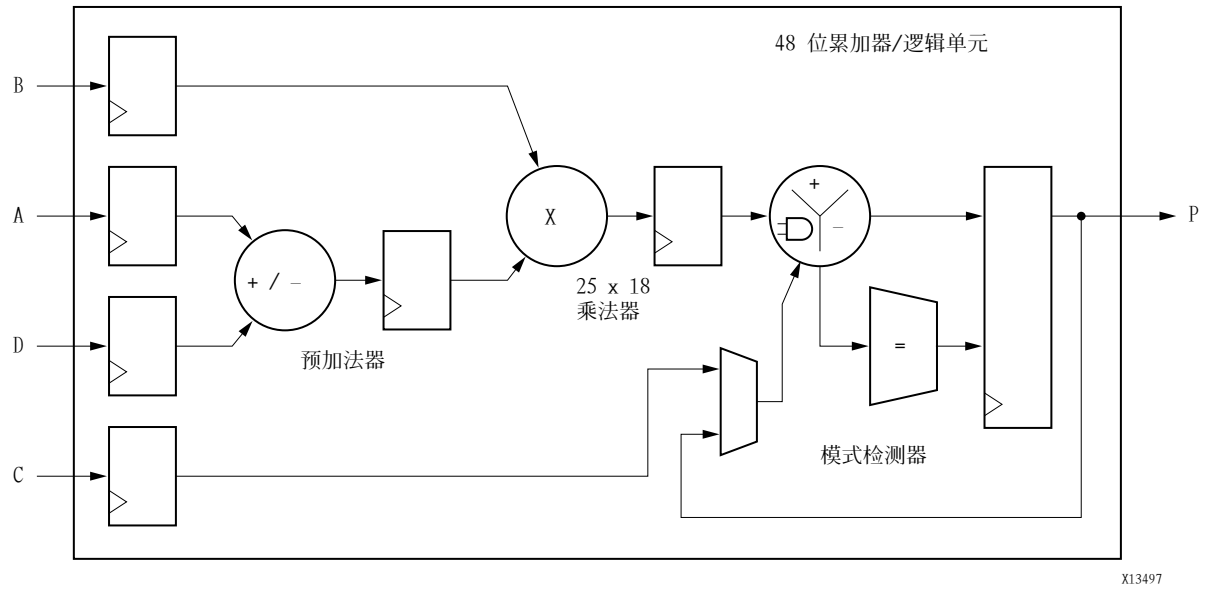


图 2-5：DSP 块的结构

存储元件

FPGA 器件包含嵌入式内存元件，可用作为随机存取内存 (RAM)、只读内存 (ROM) 或移位寄存器。这些元件包括块 RAM (BRAM)、 UltraRAM 块 (URAM)、 LUT 和移位寄存器 (SRL)。

BRAM 属于双端口 RAM 模块，通过例化到 FPGA 结构中来为相对较大的数据集提供片上存储空间。器件中提供的 2 种类型的 BRAM 内存可保存 18 k 或 36 k 比特的数据。这些内存的可用数量因器件而异。这些内存的双端口性质支持对不同位置进行同时钟周期并行访问。

就 C/C++ 代码中呈现数组的方式而言，BRAM 可实现 RAM 或 ROM 二选一。唯一差异出现在数据写入存储元件时。在 RAM 配置中，电路运行时间期间可随时读写数据。相反，在 ROM 配置中，电路运行时间期间只能读取数据。 ROM 数据写入 FPGA 配置，且无法以任何方式进行修改。

UltraRAM 块为双端口同步 288 Kb RAM，具有深度为 4,096 位且宽度为 72 位的固定配置。这些块可在 UltraScale+ 器件上使用，可提供的存储容量比 BRAM 多 8 倍。

如前文所述， LUT 是小型内存，其中真值表内容在器件配置期间写入。由于赛灵思 FPGA 中 LUT 结构的灵活性，这些块可用作为 64 位内存，通常称之为“分布式内存”。这是 FPGA 器件上提供的最快的内存类型，因为它可在结构中任意部分例化，以改进实现的电路的性能。

移位寄存器是彼此相连的一连串寄存器。此结构的用途是提供数据复用和计算路径 （例如，用于滤波器）。例如，基本滤波器由乘法器相连而成，用于将单个数据样本与一组系数相乘。通过使用移位寄存器来存储输入数据，内置数据传输结构即可将数据样本移至每个时钟周期上彼此相连的下一个乘法器。图 2-6 显示了移位寄存器示例。

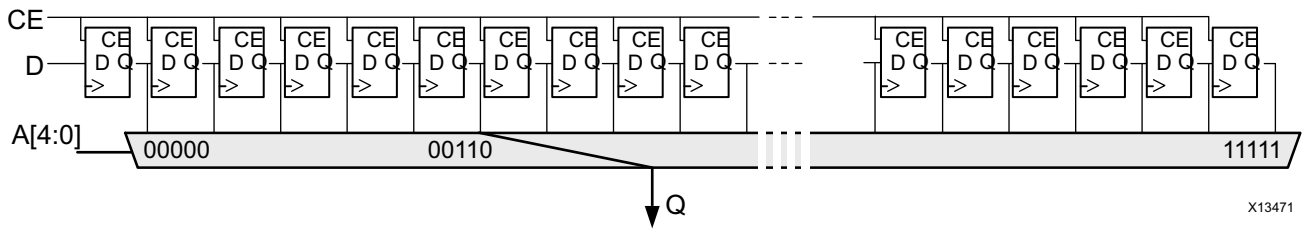


图 2-6：可寻址的移位寄存器结构

FPGA 并行性与处理器架构对比

相比于处理器架构，构成 FPGA 结构的结构在应用执行中可支持实现高度并行性。由 Vivado HLS 编译器为软件程序生成的定制处理架构可提供不同的执行范例，判断是否要将应用从处理器移植到 FPGA 时必须考量此执行范例。为详解 FPGA 执行范例的优势，本章提供了处理器程序执行的简单回顾。

处理器上的程序执行过程

各种类型的处理器均可通过一连串指令来执行程序，这些指令可转换为软件应用的实用计算方式。这一系列指令是由处理器编译器工具 （如 GNU Compiler Collection (GCC)）生成的，这些工具将以 C/C++ 表达的算法转换为处理器原生的汇编语言结构。处理器编译器负责提取如下形式的 C 函数：

$$z = a + b;$$ 等式 2-5

并将其转换为如下汇编代码：

```
ADD $R1,$R2,$R3
```

图 2-7：以汇编代码表述的计算方式

图 2-7 中的汇编代码用于定义加法运算以计算处理器内部寄存器相关的 z 值。此代码声明计算的输入值存储在寄存器 R1 和 R2 中，计算结果存储在寄存器 R3 中。此代码很简单，它并非用于表述计算 z 值所需的所有指令。此代码仅用于处理已到达处理器的数据的计算。因此，编译器必须创建其它汇编语言，以便将来自中央内存的数据加载到处理器的寄存器中并将结果重新写入内存。用于计算 z 值的完整汇编程序如下所示：

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

图 2-8：用于计算 z 的完整汇编程序

图 2-8 中的代码显示即便是简单的运算 （例如，将两个值相加）也会生成多个汇编指令。每个指令的计算时延都因指令类型而异。例如，根据 a 和 b 的位置，完成 LD 操作所需的时钟周期数都不尽相同。如果这些值包含在处理器高速缓存中，那么这些加载操作可在数十个时钟周期内完成。如果这些值包含在双倍数据率 (DDR) 主内存中，那么这些运算需耗时数百到数千个时钟周期才能完成。如果这些值包含在硬盘驱动器中，那么完成加载操作所需时间可能更长。因此习惯于追踪高速缓存命中率的软件工程师花大量时间来重构算法，以期能增加内存中数据的空间局部性，从而提高高速缓存命中率并缩短每条指令所耗处理器时间。

重要提示：通过在 FPGA 中实现算法重构操作以更好拟合可用处理器高速缓存，可以为软件工程师节省大量精力。

FPGA 上的程序执行过程

FPGA 本质上是并行处理结构，能够实现可在处理器上运行的任意逻辑和算术功能。主要区别在于用于将软件描述转换为 RTL 的 Vivado HLS 编译器不受高速缓存和统一内存空间的限制。

Vivado HLS 将 z 的计算过程编译到符合输出运算符大小要求的多个 LUT 中。例如，假定在原始软件程序中，变量 a、b 和 z 定义为短数据类型。此类型定义了 1 个 16 位数据容器，并由 Vivado HLS 作为 16 个 LUT 来实现。

注释：一般情况下， 1 个 LUT 等同于 1 个计算位。

用于计算 z 的 LUT 均为此运算专用。不同于所有计算共享同一个 ALU 的处理器， FPGA 实现可在软件算法中为每次计算例化 1 组独立 LUT。

除了按计算分配专用 LUT 资源外，FPGA 与处理器的内存架构和内存访问成本都不同。在 FPGA 实现中，Vivado HLS 编译器将内存分配到多个存储 bank 中，并且尽可能靠近运算的使用点。这导致远超处理器容量的瞬时内存带宽。例如，赛灵思 Kintex®-7 410T 器件有总计 1,590 个 18 k 比特 BRAM 可用。就内存带宽而言，此器件的内存布局可以为软件工程师提供每秒 0.5M 比特的容量 （寄存器级）和每秒 23T 比特的容量 （BRAM 级）。

就计算吞吐量和内存带宽而言，Vivado HLS 编译器通过调度、流水线化和数据流程过程来实践 FPGA 结构的各项功能。虽然这些过程对于用户都是透明的，但都是软件编译过程中不可或缺的阶段，汇总了软件应用中能实现的最佳电路级实现。

调度

调度是确定不同运算之间的数据和控制依赖关系的过程，用于判定执行每项运算的时间。在传统 FPGA 设计中，这是一个手动过程，对于硬件实现，它也被称为软件算法的并行化。

Vivado HLS 可分析相邻运算之间的依赖关系和跨时间运算的依赖关系。这使编译器可将运算分组以便在相同时钟周期内执行，并设置硬件以允许函数调用重叠。函数调用执行的重叠可解除处理器要求当前函数调用完全完成后才能开始对同一组运算执行下一次函数调用的限制。此过程称为“流水线化”，在下文和后续章节中对此进行了详细说明。

流水线化

流水线化是一种数字设计技巧，支持设计人员避免数据依赖关系，提升算法硬件实现中的并行性。原始软件实现中的数据依赖关系将保留用于实现功能等效性，但所需电路将拆分为连续的独立阶段。链接在一起的所有阶段都可在同一时钟周期上并行运行。唯一差异在于每个阶段的数据源。计算过程中每个阶段都会从上一个时钟周期内的前一个阶段计算所得结果接收其数据值。例如，要计算以下函数， Vivado HLS 编译器会例化 1 个乘法器块和 2 个加法器块：

$$y = (a \times x) + b + c \qquad \text{等式 2-6}$$

图 2-9 显示了此计算结构和流水线化的工作。它显示了示例函数的 2 种实现。上方实现是计算无流水线化情况下的 y 值结果所需的数据路径。此实现的行为与对应 C/C++ 函数行为的相似之处在于，计算开始时，所有输入值都必须已知，并且每次只能计算 1 个 y 值结果。下方实现显示了相同电路的流水线化版本。

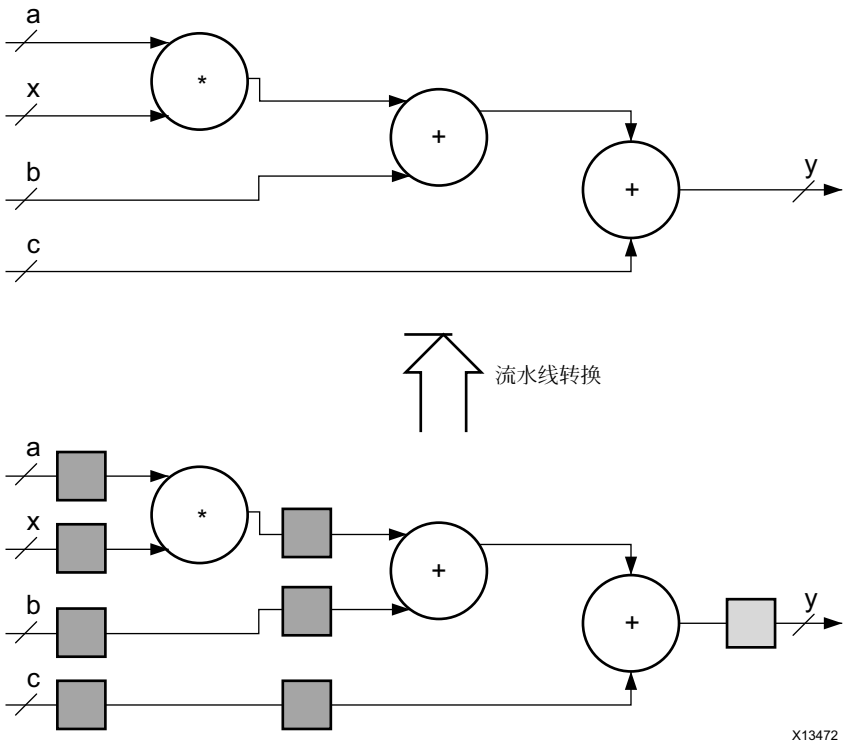


图 2-9：计算函数的 FPGA 实现

图 2-9 中的数据路径中的框表示由 FPGA 结构中的触发器块实现的寄存器。每个框都计作为 1 个时钟周期。因此，在流水线化版本中，每个 *y* 值结果的计算都需要 3 个时钟周期。通过添加寄存器，可将每个块分隔为 1 个独立的计算时间段。这意味着具有乘法器的段和具有 2 个加法器的段可并行运行，从而减少函数的总体计算时延。通过并行运行数据路径的 2 个段，此块实际上可并行计算 y 值和 y′ 值，其中 y′ 值是下次执行等式 2-6 的结果。y 的初始计算也称为“流水线填充时间 (pipeline fill time)”，需耗时 3 个时钟周期。完成初始计算后，在每个时钟周期的输出处会提供新的 y 值，因为计算流水线包含对应当前 y 计算和后续 y 计算的重叠的数据集。

图 2-10 显示的流水线化架构中，原始数据 （深灰色）、半计算数据 （白色）和最终数据 （浅灰色）同时存在，在每个阶段自己的寄存器集中会捕获该阶段的结果。因此，虽然此类计算方式的时延长达数个周期，但每个周期都可生成 1 个新的结果。

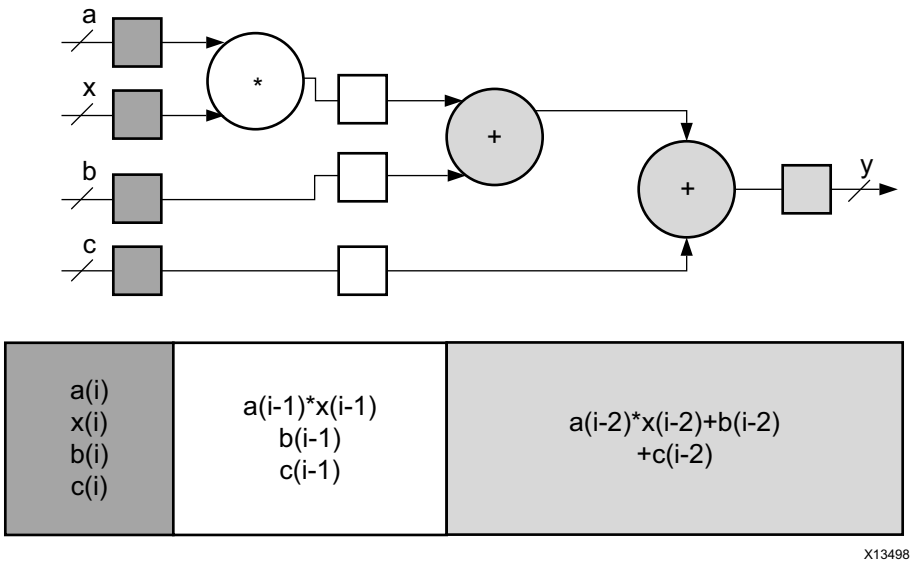


图 2-10：流水线化架构

数据流

数据流是另一种数字设计技巧，其概念与流水线化相似。数据流的目标是在粗颗粒度层面上表述并行化。就软件执行而言，这种变换适用于在单一程序内并行执行函数。

Vivado HLS 通过根据程序的不同函数的输入和输出来评估这些函数间的交互，从而提炼出这一层面的并行机制。最简单的并行化案例是函数在不同数据集上进行处理但彼此不进行通信。在此情况下，Vivado HLS 会为每个函数分配 FPGA 逻辑资源，然后独立运行各块。软件程序中常见的较为复杂的情况是，由 1 个函数为另 1 个函数提供结果。这种情况被称为“使用者/生产者情境 (consumer-producer scenario)”。

Vivado HLS 支持 2 种使用者/生产者情境使用模型。在首个使用模型中，生产者创建完整的数据集，然后使用者即可开始操作。并行性是通过例化一对 BRAM 内存 （排列为内存 bank 乒乓）来实现的。在函数调用持续时间内，每个函数都只能访问 1 个内存 bank （乒或乓）。当新函数调用开始时， HLS 生成的电路会为生产者和使用者同时切换内存连接。此方法可保证功能正确性，但会限制跨函数调用可实现的并行度。

在第 2 个使用模型中，使用者可使用来自生产者的部分结果开始工作，并且可实现的并行度已提升，在单次函数调用中即可包含执行操作。 Vivado HLS 为 2 个函数生成的模块将通过使用先入先出 (FIFO) 内存电路来连接。此内存电路充当软件编程中的 1 个队列，从而在模块之间提供数据级同步。在函数调用中的任意时间点，2 个硬件模块都在同时执行其编程操作。唯一例外是使用者模块会等待生产者提供部分可用数据后再开始计算。在 Vivado HLS 术语中，使用者模块的等待时间称为“时间间隔”或“启动时间间隔 (II)”。

XILINX

第 3 章

硬件设计的基本概念

简介

处理器与 FPGA 的关键差异之一是处理架构是否固定。此差异直接影响编译器处理每个目标的方式。处理器采用固定的计算架构，编译器负责判定可用处理架构中运行软件应用的最佳方式。性能是应用映射到处理器功能的效率高低和正确执行应用所需的处理器指令数量的指标。

相反，FPGA 类似于包含大量构建块的空白候选清单。Vivado® HLS 编译器负责从最适合软件程序的各种构建块创建处理架构。指导 Vivado HLS 编译器创建最合适的处理架构的过程需要有关硬件设计概念的基本知识。

本章涵盖了适用于基于 FPGA 的设计和基于处理器的设计的常用设计概念，并解释了这些概念之间的关系。本章不涵盖 FPGA 设计的详细内容。与处理器编译器相同， Vivado HLS 编译器可在 FPGA 逻辑结构中处理算法实现的低层次细节。

时钟频率

处理器时钟频率是判定特定算法的执行平台时要考量的首要事项之一。常用指导原则是时钟频率越高，算法执行速率性能越高。虽然这可能是选择处理器的有效首要规则，但实际上它具有误导性，可能导致设计人员在选择处理器和 FPGA 时选择错误。

这一常用指导原则出现误导的原因在于处理器与 FPGA 之间的时钟频率的名义差异。例如，比较处理器与 FPGA 的时钟频率时，经常会遇到表 3-1 中所示的比较。

表 3-1：最大时钟频率示例

处理器	FPGA
2 GHz	500 MHz

简单分析表 3-1 中的值可能会误导设计人员认为处理器的性能为 FPGA 的 4 倍。这种简单分析错误假定这些平台间的唯一差异就是时钟频率。但这些平台之间还存在其它差异。

处理器与 FPGA 之间的首个主要差异就在于软件程序的执行方式。处理器能够在 1 个公共硬件平台上执行任何程序。此公共平台包含处理器核，并定义所有软件都必须拟合的固定架构。而编译器则具有针对处理器架构的内置认知，可将用户软件编译为一组指令。生成的一组指令始终都可按相同的基本顺序来执行，如图 3-1 中所示。

图 3-1：处理器指令执行阶段

无论处理器类型为标准处理器还是专用处理器，指令的执行始终相同。用户应用的每条指令都必须经历以下阶段：

1. 指令访存 (IF)
2. 指令解码 (ID)
3. 执行 (EXE)
4. 内存操作 (MEM)
5. 回写 (WB)

在表 3-2 中总结了每个阶段的用途。

表 3-2：指令处理阶段

阶段	描述
IF	从程序内存获取指令。
ID	对指令进行解码来判定运算和运算符。
EXE	在可用硬件上执行指令。在标准处理器中，这意味着算术逻辑单元 (ALU) 或浮点单元 (FPU) 二选一。在此指令处理阶段中，专用处理器在标准处理器功能的基础上增加了固定功能加速器。
MEM	使用内存操作为后续指令访存数据。
WB	将指令结果写入局部寄存器或全局内存。

大部分现代化处理器都包含指令执行路径的多个副本，并且能够以一定程度的重叠来运行指令。由于处理器中的指令通常彼此相关联，因此指令执行硬件副本之间的重叠并不完美。最好情况下，仅限通过使用处理器引入的开销阶段才能重叠。负责应用计算的 EXE 阶段则按顺序执行。按顺序执行的原因与 EXE 阶段中的有限资源以及指令间的依赖关系有关。

图 3-2 显示的处理器可按半并行顺序执行多条指令。这对于处理器而言实属最佳情况，所有指令都可尽可能快速执行。即使在此最佳情况下，处理器仍受到限制，每个时钟周期仅限 1 个 EXE 阶段。这意味着每个时钟周期都会将用户应用向前移动 1 个操作步骤。即使编译器判定全部 5 个 EXE 阶段都应并行执行，处理结构也会阻止此行为。

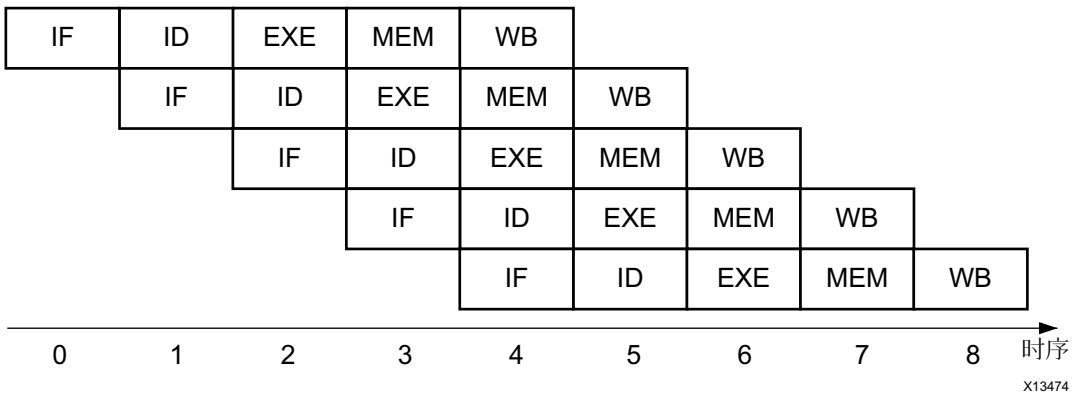


图 3-2：含多个指令执行单元的处理器

FPGA 不在公共计算平台上执行所有软件。它仅在对应每个程序的定制电路上执行该程序。因此，更改用户应用就会更改 FPGA 中的电路。不同于图 3-1，在 FPGA 中处理的 EXE 阶段如图 3-3 中所示。 MEM 阶段是否存在取决于应用。

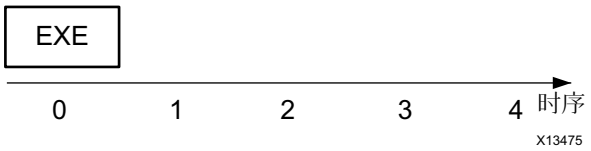


图 3-3：FPGA 指令执行阶段

因应这一灵活性优势， Vivado HLS 编译器无需考量平台中的开销阶段，并且可寻找各种途径来最大限度提升指令并行度。按图 3-2 中所示假定，在图 3-4 中演示了在 FPGA 中执行相同软件的剖析过程。

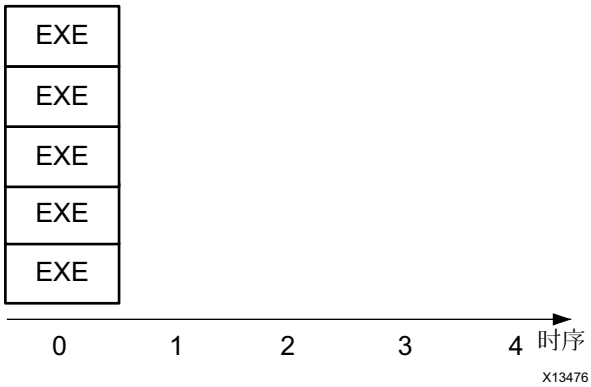


图 3-4：含多个指令执行单元的 FPGA

根据图 3-2 与图 3-4 的比较可以发现， FPGA 的名义性能优势达到处理器的 9 倍。实际数字始终因应用而异，但通常对于计算密集型应用， FPGA 可展现出至少 10 倍于处理器的性能。

仅关注时钟频率所隐藏的另一个问题是软件程序的功耗。功耗的近似计算方式如下：

$$P = \frac{1}{2}cFV^2 \qquad \text{等式 3-1}$$

如等式 3-1 中所示，功耗与时钟频率之间的关系受到经验数据的支持，相同计算工作负载下，处理器的功耗较 FPGA 更高。通过根据软件程序创建定制电路， FPGA 即可以更低的时钟频率运行，同时最大限度提升各操作间的并行度，而不会像处理器中一样出现指令解释开销。

建议：在处理器与 FPGA 之间进行选择时，建议根据吞吐量和时延而不是最大时钟频率来分析应用需求和计算工作负载。

时延和流水线化

时延表示完成任一指令或一组指令以生成应用结果值所需的时钟周期数。使用图 3-1 中所示的基本处理器架构时，任一指令的时延为 5 个时钟周期。如果应用包含总计 5 条指令，则此简单模型的总体时延为 25 个时钟周期。即，完成 25 个时钟周期后，应用结果才可供使用。

应用时延在 FPGA 和处理器中都是关键的性能指标。在这两大平台上，时延问题是通过使用流水线化来解决的。在处理器中，流水线化意味着在完成当前指令前即可将下一条指令发送到执行中。这样即可允许指令集处理中所需的开销阶段发生重叠。图 3-2 中显示了流水线化对于处理器所能实现的最佳结果。通过重叠指令执行，处理器可以为含 5 条指令的应用实现 9 个时钟周期的时延。

在 FPGA 中，不存在与指令处理相关联的开销周期。时延是根据运行原始处理器指令的 EXE 阶段所需的时钟周期数来度量的。对于图 3-3 中所示情况，时延为 1 个时钟周期。并行机制在时延中同样发挥着重要作用。对于包含完整的 5 条指令的应用， FPGA 时延同样为 1 个时钟周期，如图 3-4 中所示。对于 FPGA 的 1 个时钟周期时延，流水线化的优势可能并不那么显而易见。但在 FPGA 中采用流水线化的原因与处理器中并无不同，即为了提升应用性能。

正如前文所述，FPGA 是一个包含大量构建块的空白候选清单，必须将这些构建块连接起来才能实现应用。Vivado HLS 编译器可直接连接这些构建块，也可以通过寄存器来连接。图 3-5 显示了图 3-3 中使用 5 个构建块实现的 EXE 阶段的实现过程。

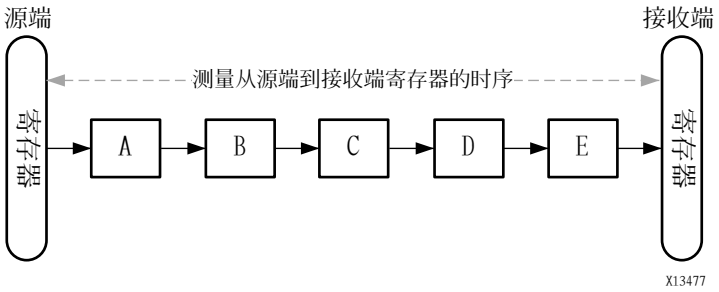


图 3-5：FPGA 实现 （无流水线）

FPGA 中的运算时序即单一信号从源端寄存器传输到接收端寄存器所耗费的时间。假定图 3-5 中的每个构建块都需要 2 ns 执行完成，当前设计需要 10 ns 来实现其功能。时延仍为 1 个时钟周期，但时钟频率限制为 100 MHz。100 MHz 频率限制衍生自 FPGA 中的时钟频率定义。对于 FPGA 电路，时钟频率定义为源端和接收端寄存器间的最长信号传输时间。

FPGA 中的流水线化是插入更多寄存器以将大型计算块细分为较小的分段的过程。这种计算分区会导致时钟周期数增加从而增加时延，但因允许定制电路以更高的时钟频率运行而提升性能。

图 3-6 显示了图 3-5 中完成流水线化后处理架构的实现。完整的流水线化意味着在 FPGA 电路中每个构建块间插入寄存器。添加寄存器会将电路的时序要求从 10 ns 降低到 2 ns，从而使最大时钟频率达到 500 MHz。此外，通过将计算过程细分为多个绑定寄存器的独立区域，即可使每个块时钟保持繁忙，从而给应用吞吐量带来积极影响。

流水线化有 1 个问题，即电路的时延。图 3-5 的原始电路时延为 1 个时钟周期，但时钟频率较低。与此相反，图 3-6 的电路的时延为 5 个时钟周期，而时钟频率则较高。

重要提示：由于流水线化而导致的时延是在 FPGA 设计器件需要考量的取舍之一。

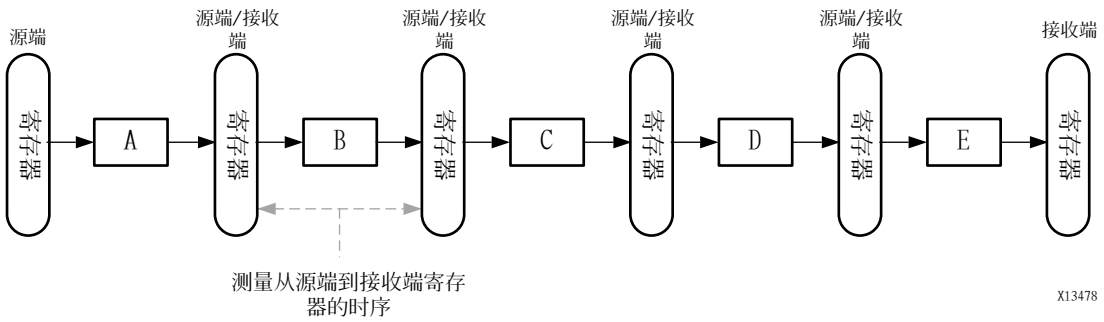


图 3-6：FPGA 实现 （含流水线）

吞吐量

吞吐量是用于判定实现的总体性能的另一个指标。它表示处理逻辑接受下一个输入数据样本所耗费的时钟周期数。对于该值，请谨记，电路的时钟周期会改变吞吐量数值的意义。

例如，图 3-5 和图 3-6 所示的实现在两次输入数据采样之间都需要 1 个时钟周期。关键差异在于，图 3-5 中的实现所需输入采样时间间隔为 10 ns，而图 3-6 中的电路所需输入数据采样时间间隔仅为 2 ns。当基础时间已知后，显而易见，第 2 种实现的性能更高，因为它可接受更高的输入数据速率。

注释：分析处理器上执行的应用时，同样可以使用本章中所述的吞吐量定义。

内存架构与布局

所选实现平台的内存架构是可影响软件应用的物理要素之一。内存架构用于确定可实现的性能上限。在某些性能点，处理器或 FPGA 上的所有应用都会受内存限制，与可用计算资源的类型和数量无关。 FPGA 设计中的策略之一是了解内存限制的范围及其受数据布局和内存组织方式的影响。

在基于处理器的系统中，软件工程师必须在本质相同的内存架构上拟合应用，与处理器的具体类型无关。这种共通性可以简化应用移植过程，但会牺牲性能。软件工程师所熟悉的公用内存架构由慢速、中速或快速内存 （依据将数据输入处理器所耗用的时钟周期数来区分）组成。在表 3-3 中提供了这些内存分类的定义。

表 3-3：内存类型定义

内存类型	定义
慢速	大空间存储设备，例如，硬盘驱动器
中速	DDR 内存
快速	片上高速缓存 （大小因特定处理器而异）

该表中所示的内存架构假定为用户提供单一大型内存空间。在此内存空间内，用户负责进行程序数据存储区域的分配和取消分配。数据的物理位置及其在不同层级间移动的方式由计算平台处理，并且对用户透明。在此类系统中，提升性能的唯一方式是尽可能增加高速缓存中数据的复用。

为实现此目标，软件工程师必须花大量时间来查看高速缓存追踪，重构软件算法以增加数据局部性，并管理内存分配以最大限度降低程序的即时性内存占用。虽然所有这些技巧都可在各种处理器间广泛应用，但结果却不可复制。必须根据运行软件程序的每个处理器来对软件程序进行微调才能最大限度提升性能。

借助基于处理器的内存处理经验在 FPGA 内处理内存时，软件工程师遇到的第一种差异是不存在固定的片上内存架构。基于 FPGA 的系统可连接到慢速和中速内存，但在可用的快速内存方面却展现出巨大差异。即， Vivado HLS 编译快速内存架构以在算法中找到最合适的数据布局，而不是对软件进行重构以最大限度利用现有高速缓存。由此生成的 FPGA 实现可能具有 1 个或多个大小不同的内部 bank，并且可独立相互访问。

图 3-7 中的代码示例展示了为应对程序中的内存需求而提供的最佳实践建议。

处理器代码

```
void foo(......)
{
   int *A = (int *)malloc(10 * sizeof(int));
....   free(A);
}
```

FPGA 代码

```
void foo(......)
{   int A[10];
....}
```

图 3-7：处理器和 FPGA 代码示例

FPGA 代码缺少动态内存分配，这可能会令经验丰富的软件工程师也大吃一惊。对于基于处理器的系统而言，由于底层采用固定内存架构，因而使用动态内存分配早已成为最佳实践准则之一。

与此方法相反， Vivado HLS 编译器会根据应用构建专用内存架构。这种专用内存架构是根据程序中的内存块大小和在程序执行的整个过程中数据的使用方式来定制的。当前最佳 FPGA 编译器 （如 Vivado HLS）要求在编译时可完整分析应用的内存要求。

静态内存分配的优势在于 Vivado HLS 可以多种不同方式为数组 A 实现内存。根据算法中的计算方式，Vivado HLS 编译器可将 A 的内存作为寄存器、移位寄存器、 FIFO 或 BRAM 来实现。

注释：尽管限制使用动态内存分配，但 Vivado HLS 编译器完全支持指针。如需了解有关指针支持的详情，请参阅第 4 章中的“指针”。

寄存器

内存的寄存器实现是可行的最快内存结构。按此实现方式， A 的每个条目都会变为一个独立实体。每个独立实体都会嵌入计算以供使用，而无需对逻辑进行寻址，也不存在其它延迟。

移位寄存器

在处理器编程方面，移位寄存器可被视为一种特殊队列。在此实现中， A 的每个要素都会在计算过程的不同部分被多次使用。移位寄存器的关键特征是在每个时钟周期上都可访问 A 的每个要素。此外，将所有数据项都移至下一个相邻存储容器只需 1 个时钟周期即可。

FIFO

FIFO 可被视为含单一入口点和单一出口点的队列。此类结构通常用于在程序循环或函数之间传输数据。其中不涉及寻址逻辑，实现详细信息完全由 Vivado HLS 编译器处理。

BRAM

BRAM 是嵌入 FPGA 结构的随机存取内存。赛灵思 FPGA 器件包含大量此类嵌入式内存。内存精确数量因器件而异。在处理器编程方面，此类内存可被视作为具有如下限制的高速缓存：

- 无法实现高速缓存一致性，存在冲突，并且在处理器高速缓存中经常会发现高速缓存缺少追踪逻辑。
- 只能在器件通电期间保留其值。
- 支持同周期并行访问 2 个不同内存位置。

XILINX

第 4 章

Vivado 高层次综合

简介

赛灵思 Vivado® 高层次综合 (HLS) 编译器所提供的编程环境类似于在标准处理器和专用处理器上为进行应用开发所提供的编程环境。Vivado HLS 与处理器编译器采用相同的关键技术来进行 C/C++ 程序的解释、分析和优化。主要差异在于应用的执行目标。

将 FPGA 作为执行结构时， Vivado HLS 支持软件工程师针对吞吐量、功耗和时延来优化代码，而无需应对单一内存空间的性能瓶颈和有限的计算资源所带来的问题。这样即可将计算密集型软件算法实现为真正的产品，而不只是功能演示。本章介绍了 Vivado HLS 编译器的工作方式及其与传统软件编译器的差异。

以 Vivado HLS 编译器为目标的应用代码使用的类别与所有处理器编译器都相同。Vivado HLS 对所有程序都进行如下方面的分析：

- 运算
- 条件语句
- 循环
- 函数

重要提示：Vivado HLS 可编译几乎任何 C/C++ 程序。 Vivado HLS 的唯一编码限制在于含单一内存空间的处理器中常见的动态语言结构。使用 Vivado HLS 时，需要考量的主要动态结构是内存分配和指针，如本章中所述。

运算

运算表示计算结果值过程中所涉及的应用的算法组件和逻辑组件。此定义有意排除了比较语句，因为这些语句在“条件语句”中处理。

处理运算时， Vivado HLS 与其它编译器的主要差异在于对设计人员所施加的限制。对于处理器编译器，固定处理架构意味着用户只能通过限制运算依赖关系和操纵内存布局来最大限度提升高速缓存性能。相比之下， Vivado HLS 不受固定处理平台的限制，可根据用户输入来构建特定于算法的平台。这使 HLS 设计人员能够给应用性能带来吞吐量、时延和功耗方面的影响，正如本章示例所示。

图 4-1 显示了计算 F[i] 结果中所涉及的 3 项运算。

```
A[i] = B[i] * C[i];
D[i] = B[i] * E[i];
F[i] = A[i] + D[i];
```

图 4-1：3 项运算的代码示例

通过使用处理器，生成的执行剖析类似于图 4-2。此应用剖析仅聚焦中央处理单元 (CPU) 内指令处理的 EXE 阶段。这是指令处理中处理器与 FPGA 共有的唯一阶段。在此示例中，执行追踪采用顺序方式，原因在于执行平台，而非算法。根据算法， A[i] 和 D[i] 的值可按任意顺序计算，也可同时计算。唯一算法限制在于，这两个值都必须先于 F[i] 计算。

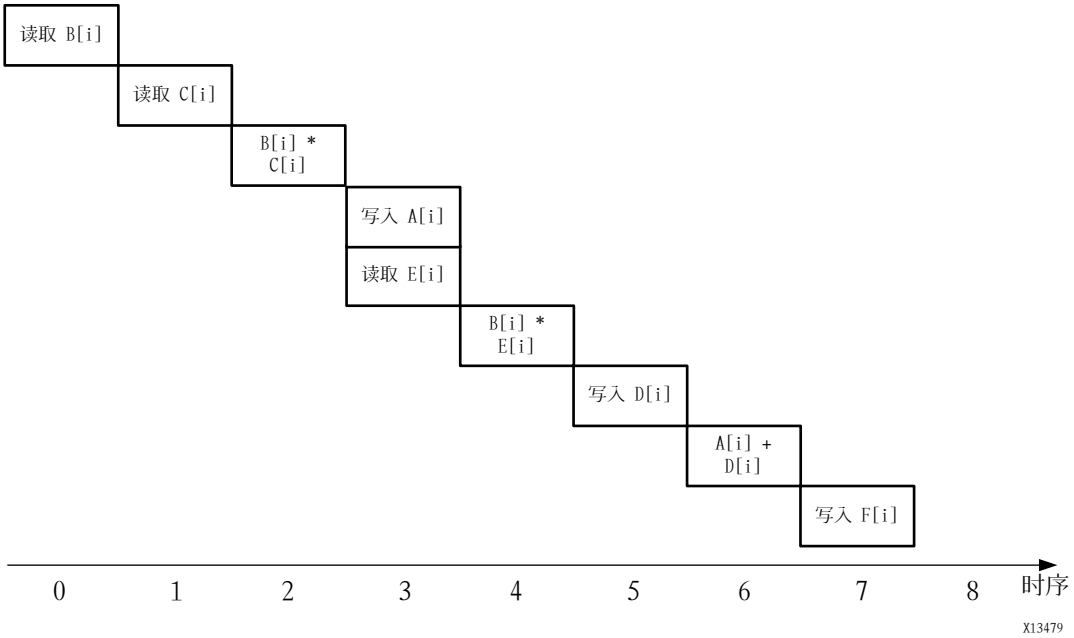


图 4-2：处理器上的代码执行示例

图 4-3 显示了在 FPGA 上使用 Vivado HLS 默认设置对图 4-1 中的代码进行编译的结果。生成的执行剖析与处理器相比，相似之处在于乘法和加法同样按顺序方式发生。采用此默认行为的原因是为了最大限度减少实现用户应用所需的构建块数量。虽然 FPGA 没有固定处理架构，但每个器件可保存的构建块的最大数量是有限的。因此，设计人员可以对 FPGA 资源进行评估，按器件比较应用性能和应用数量。

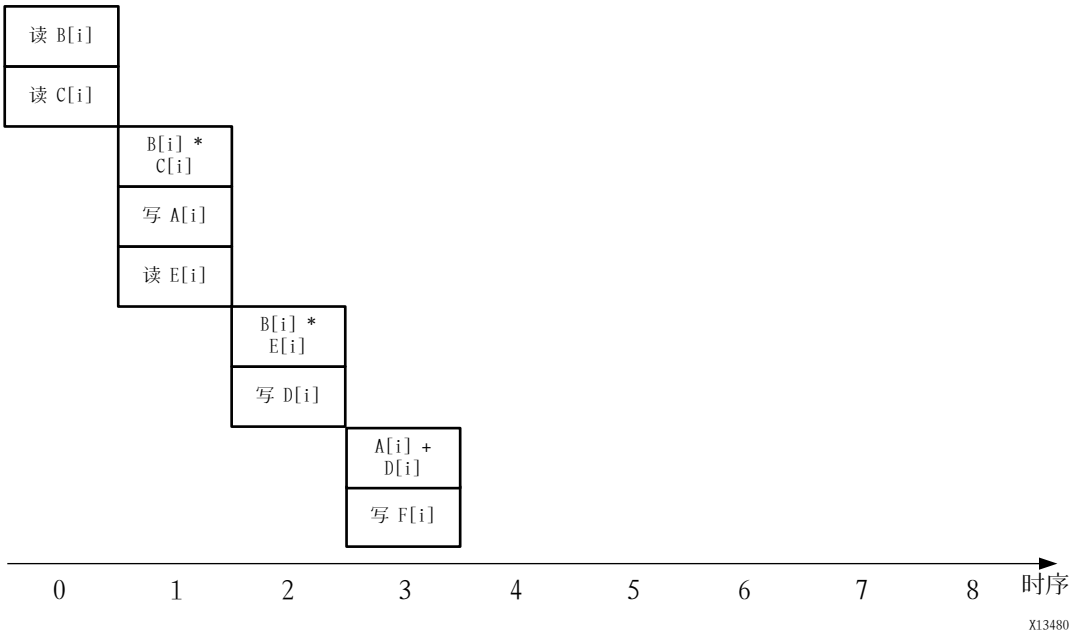


图 4-3：FPGA 上 HLS 代码的默认执行

即使采用默认行为，由于为算法创建的定制内存架构，其实现的性能同样优于处理器执行。在处理器上，数组 A、 B、C、 D、 E 和 F 存储在单一内存空间内，每次只能访问其中一个数组。相比之下， HLS 可检测这些内存并为每个数组创建独立内存 bank，从而在数组 B 和数组 C 之间生成读操作重叠。

通过在时钟周期 1 内调度数组 E 的读操作，可显示来自 Vivado HLS 的自动资源优化措施之一。对于内存运算，Vivado HLS 会分析包含数据和计算期间所使用的值的内存 bank。虽然在时钟周期 0 期间可读取数组 E，但 Vivado HLS 会将内存运算自动置于尽可能靠近数据使用位置，从而减少电路中的临时数据存储量。由于使用 E 值的乘法器在时钟周期 2 之前不运行，因此将读访问调度为早于时钟周期 1 发生是没有意义的。

Vivado HLS 帮助用户控制生成的电路大小的另一种方法是提供数据类型用于调整变量大小。与所有编译器相似，Vivado HLS 可为用户提供对整数、单精度和双精度数据类型的访问权。这样即可将软件快速移植到 FPGA，但可能隐藏因处理器中可用的 32 位和 64 位数据路径而导致的算法低效问题。

例如，假定图 4-1 中的代码在数组 B、C 和 E 中只需 20 位值。在原处理器代码中，同样大小的位要求数组 A、D 和 F 能够存储 64 位值以避免发生任何精度损失。 Vivado HLS 可按现状编译代码，但这导致低效的 64 位数据路径耗用的资源量超过算法所需的量。

图 4-4 显示了如何使用 Vivado HLS 任意精度的数据类型来重写图 4-1 中的代码的示例。通过使用这些数据类型即可快速完成确保算法准确性所需的最低精度的软件级分析和验证。除减少实现计算所需的资源量外，使用任意精度数据类型还可减少完成运算所需的逻辑级数。这样即可减少设计时延。

```
ap_int<40> A[10], D[10];
ap_int<41> F[10];
ap_int<20> B[10], C[10], E[10];
...
A[i] = B[i] * C[i];
D[i] = B[i] * E[i];
F[i] = A[i] + D[i];
```

图 4-4：使用 HLS 任意精度类型进行编码的示例

如第 3 章"硬件设计的基本概念"中所述，流水线化或者将计算细分为绑定寄存器的较小区域是 FPGA 设计实现目标时钟频率的基本技巧。根据运算大小，此最优化由 Vivado HLS 自动实现。 Vivado HLS 将大型运算符细分为多个计算阶段，因而导致电路时延相应增加。

条件语句

条件语句是程序控制流程语句，通常作为 if、 if-else 或 case 语句来实现。这些编码结构是大部分算法不可或缺的一部分，受包括 HLS 在内的所有编译器的完整支持。编译器间的唯一差异在于实现这些类型的语句的方式。

对于处理器编译器，条件语句转换为分支运算，这可能导致上下文切换。引入分支会破坏最大指令执行封装 （如图 3-2 中所示），因为由此引入的依赖关系会影响从内存访存的下一条指令的判定。这种不确定性导致处理器执行流水线中出现泡沫，并直接影响程序性能。

在 FPGA 中，条件语句对性能并不存在与处理器相同的潜在影响。 Vivado HLS 会创建条件语句的每个分支所描述的所有电路。因此，条件软件语句的运行时间执行涉及在 2 个可能的结果之间进行选择而不是上下文切换。

循环

循环是用于表述迭代计算的常用编程结构。一个常见的误解是使用 HLS 之类的编译器时不支持循环。虽然对于 FPGA 的早期版本的编译器确实如此，但 HLS 已完全支持循环，甚至可执行超越标准处理器编译器能力范畴的变换。图 4-5 显示了简单循环示例。

```
for(i=0; i < 10; i++)
{
      A = A + (B[i] * C[i]);
}
```

图 4-5：循环代码

为便于演示，假定循环每次迭代需 4 个时钟周期，与实现平台无关。在处理器上，强制编译器按顺序调度循环迭代，总计运行时间为 40 个周期，如图 4-6 中所示。

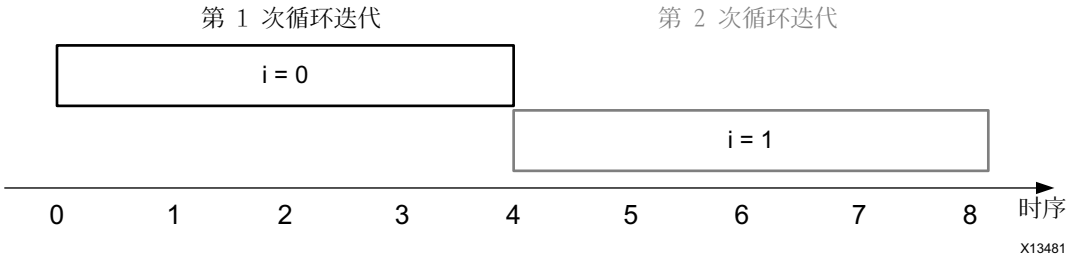


图 4-6：处理器上的循环迭代调度

HLS 不存在此限制。因为 HLS 会为算法创建硬件，因此它可通过将迭代流水线化来更改循环的执行剖析。循环迭代流水线化可将运算并行化的概念从循环迭代内扩展至跨迭代进行。

为减少迭代时延， Vivado HLS 应用的首个自动最优化是对应循环迭代主体的运算符并行化。第二项最优化是循环迭代流水线化。此最优化需要用户输入，因为它影响 FPGA 实现的资源使用和输入数据速率。

HLS 的默认行为是按处理器的调度顺序来执行循环，如图 4-6 中所示。这意味着图 4-5 中的代码的处理时延为 40 个周期，输入数据速率为每 4 个周期一次。在此示例中，输入数据速率根据从输入对 B 和 C 的值进行采样的速度快慢程度来定义。

HLS 可将循环迭代并行化或流水线化，以降低计算时延并增大输入数据速率。用户通过设置循环启动时间间隔 (II) 来控制迭代流水线化的级别。循环启动时间间隔 (II) 用于指定连续循环迭代的开始时间之间的时钟周期数。图 4-7 显示了将 II 值设置为 1 之后生成的循环调度。

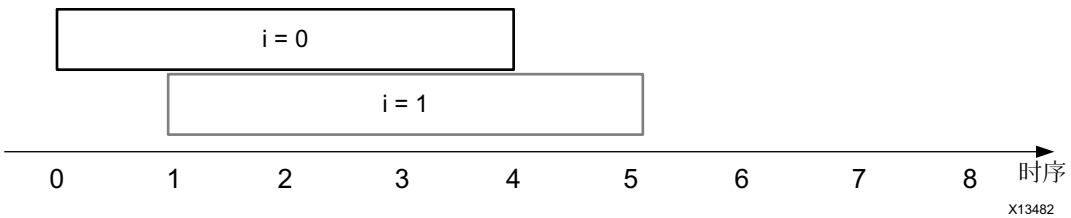


图 4-7：II = 1 时的循环迭代调度

为实现此结果， HLS 可分析循环迭代 0 和 1 之间的数据依赖关系和资源争用，并自动解决问题，如下所示：

- 为解决数据依赖关系， HLS 可更改循环主体中的运算之一，或者查询用户是否存在算法更改。
- 为解决资源争用， HLS 可例化更多资源副本或者查询用户是否存在算法更改。

在表 4-1 中总结了循环流水线化对执行特征的影响。

表 4-1：不同编译器的循环执行剖析

编译器	循环执行时延	输入数据速率
处理器	40	每 4 个时钟周期 1 次
默认 HLS	40	每 4 个时钟周期 1 次
HLS， II = 1	14	每个时钟周期 1 次

函数

函数为编程层级，可包含运算符、循环和其它功能。 HLS 和处理器编译器中的函数的处理方式与循环中的处理方式类似。

在 HLS 中，循环与函数之间的主要差异在于术语。 HLS 可将循环与函数的执行都并行化。对于循环，这种变换方式通常称为流水线化，因为在运算符与循环迭代之间存在明确的层级。对于函数，循环主体外部与循环内部的运算位于相同层级上下文中，这可能导致使用“流水线化”一词时发生混淆。为避免使用 HLS 时可能出现的混淆，函数调用执行的并行化称为“数据流最优化 (dataflow optimization)”。

数据流最优化会指示 HLS 为给定程序层级的所有函数创建独立的硬件模块。这些独立硬件模块能够在数据传输期间并行执行并自我同步。

动态内存分配

动态内存分配是 C 和 C++ 编程语言中可用的内存管理技巧之一。借助此方法，用户即可在程序运行时间内按需分配尽可能多的内存。已分配的内存大小因程序执行而异，且从中央物理内存池进行分配，如第 3 章"硬件设计的基本概念"中所述。在表 4-2 中显示了通常与动态内存分配关联的函数调用。

表 4-2：动态内存管理中使用的函数

C	C++
malloc()	new()
calloc()	delete()
free()	

如第 3 章"硬件设计的基本概念"中所述，FPGA 不含固定内存架构供 HLS 编译器在其中拟合用户应用。HLS 会改为根据算法的独特需求来综合内存架构。因此，提供给 HLS 编译器以供在 FPGA 中实现的所有代码都只能使用编译时间可分析的内存分配。

为帮助用户确保提供给 HLS 的所有代码均可同步，编译器会在分析设计前执行编码合规性检查。此编码合规性检查会标记不适合 HLS 使用的所有编码样式。用户负责手动更改代码并移除动态内存分配的所有实例。

图 4-8 中的代码会在内存中分配区域，每个区域存储 10 个 32 位的值。

```
int *A = malloc(10*sizeof(int));
```

图 4-8：动态内存分配

虽然此编码示例明确声明内存分配不变，但 HLS 代码合规性阶段不会分析 malloc 语句的内容。HLS 无法对包含表 4-2 中的任意关键字的代码进行综合，即使分配不变也是如此，如图 4-8 中的示例所示。有 2 种方法可用于修改此代码以符合 HLS 标准。以下代码示例演示了这些方法并解释了其对 FPGA 实现的影响。

图 4-9 中的代码显示了 C/C++ 程序执行的自动内存分配。HLS 严格按照 C/C++ 规定的行为来实现此内存样式。这意味着为存储数组 A 所创建的内存仅限于存储执行包含该数组的函数调用期间的有效数据值。因此，函数调用负责在每次使用前以有效数据填充 A。

```
int A[10];
```

图 4-9：符合 HLS 标准的自动内存分配

图 4-10 中的代码显示了 C/C++ 程序执行的静态内存分配。此类型的内存分配行为表明数组 A 的内容跨函数调用有效，直至程序完全关闭为止。处理 HLS 时，只要电路通电，针对数组 A 实现的内存包含的数据就有效。

```
static int A[10];
```

图 4-10：符合 HLS 标准的静态内存分配

自动和静态内存分配技巧都可提升处理器上运行的算法的总体软件内存占用率。在 C/C++ 中为 FPGA 实现指定算法时，最重要的注意事项是用户应用的总体目标。即，编译到 FPGA 时，主要目标并非创建尽可能最佳的软件算法实现。而是在使用 HLS 之类的工具时，确保算法捕获方式允许工具推断尽可能最佳的硬件架构从而达成尽可能最佳的实现结果。

指针

指针是到内存内位置的地址。 C/C++ 程序中指针的常见用例包括函数参数、数组处理、指针对指针以及强制类型转换。这种语言结构固有的灵活性使其成为 C/C++ 代码的实用且常用的要素。 HLS 编译器支持以在编译时间完全可分析的方式来使用指针。可分析的指针使用方式即完全可通过纸笔计算来表述和计算而无需运行时间信息的使用方式。

图 4-8 中的代码显示如何使用指针来引用内存中的动态分配区域。如前文所述， HLS 不支持这种使用方式，因为指针的目标地址仅在程序执行期间已知。这并不意味着使用 HLS 编译器时不支持将指针用于内存管理。图 4-11 显示了有效的编码样式，其中即使用指针来访问内存。

```
int A[10];
int *pA;

pA = A;
```

图 4-11：使用指针管理数组访问

此代码之所以有效，是因为指针 pA 的所有用法均可分析并映射回数组 A。由于数组 A 是通过自动内存分配创建的，HLS 可完整确定 A 的属性。

内存和指针的另一个受支持的模型是访问外部内存。使用 HLS 时，针对函数参数进行的任意指针访问均暗指变量或外部内存。HLS 将外部内存定义为编译器生成的 RTL 范围之外的任意内存。这意味着内存可能位于 FPGA 中的其它函数中或者作为片外内存 （例如， DDR）的一部分。

在图 4-12 所示代码中，函数 foo 是 HLS 的顶层模块，其中 data_in 作为参数。根据对 data_in 的多次指针访问，HLS 推断此函数参数属于外部内存模块，必须通过硬件级别的总线协议才能访问。诸如 Advanced eXtensible Interface (AXI) 协议之类的总线协议用于指定多个函数彼此连接并通信的方式。

```
void foo(int *data_in,...)
{
   int item1, item2, item3;

  item1 = *data_in;
  item2 = *(data_in + 1);
  item3 = *(data_in + 2);
  ...
}
```

图 4-12：指向外部内存的指针

XILINX

第 5 章

围绕计算的算法

简介

虽然有关算法分析早已存在大量文献，但围绕计算的算法与围绕控制的算法之间的细微差异在很大程度上依赖于实现平台。本章在 Vivado® HLS 编译器和 FPGA 环境中对围绕计算的算法予以定义。此处还包含示例和最佳实践建议，用于演示如何最大限度提升 HLS 生成的实现的性能。

围绕计算的算法即按任务进行一次性配置的算法，任务执行期间无法更改其行为。硬件中的任务与 C/C++ 程序中的函数调用相同。任务大小由 HLS 用户控制。

建议：总体上，建议根据算法中的自然分工来设置任务大小。

图 5-1 显示了 Sobel 边缘检测操作的代码。这是围绕计算的算法示例，可划分为不同大小的任务。此算法属于二维筛选操作，通过计算 x 向和 y 向中每个像素的颗粒度来计算图像中某个区域的边缘。正如此代码所示，可通过 HLS 将其编译为 FPGA 实现。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
             for(colOffset= -1; colOffset <= 1; colOffset++){
                x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
             }
          }
     edge_weight = ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

图 5-1：Sobel 边缘检测算法 - 任务选择 1

为正确优化此算法，设计人员必须首先明确任务大小。任务大小用于判定所需的硬件模块的配置生成频率以及新数据批次的接收频率。图 5-2 和图 5-3 展示了图 5-1 中代码的 2 种可能的任务定义。或者也可以选择将图 5-1 中的代码定义为 1 项任务。

任务选择 2（图 5-2）创建的硬件模块仅用于颗粒度计算。颗粒度计算在 3x3 的像素窗口内执行，不支持行或图像帧的概念。此项选择的问题在于其执行的工作量与算法的自然分工之间存在不匹配。Sobel 边缘检测在完整的图像范围内执行。这意味着，对于这种任务大小选择而言，设计人员必须判定如何将图像划分为 HLS 构建的任务处理器所需的 3x3 像素片。该算法功能需要处理器或其它硬件模块才能完成。

```
for(rowOffset = -1; rowOffset <= 1; rowOffset++){
      for(colOffset= -1; colOffset <= 1; colOffset++){
                  x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                  y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
      }
}
```

图 5-2：Sobel 边缘检测算法的任务选择 2

任务选择 3 （图 5-3）为按任务处理整个像素行。这是基于任务选择 1 的改进方法，因为它实现算法的完整功能所需的额外模块数量较少。此方法还将与控制处理器的交互减少到每行一次。将任务大小调整为每次处理一行的问题在于底层操作需要处理多个行才能计算出结果。对于此选项，可能需要采用复杂的控制机制来将图像行序列化为 HLS 生成的硬件模块。

```
for(j = 0; j < width; j++){
        x_dir = 0;
        y_dir = 0;
        if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
        for(rowOffset = -1; rowOffset <= 1; rowOffset++){
            for(colOffset= -1; colOffset <= 1; colOffset++){
               x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
               y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
            }
         }
      edge_weight =ABS(x_dir) + ABS(y_dir);
      output_image[i][j] = edge_weight;
}
```

图 5-3：Sobel 边缘检测算法的任务选择 3

任务选择 1 （图 5-1）是此算法的最佳选择，因为它与图 5-1 所示代码中表述的每次函数调用的完整图像相匹配。此项选择即围绕计算的任务，因为生成的 FPGA 实现的配置在处理任一图像帧时都是固定的。处理的图像大小可随图像帧而改变，但任务开始后即不可变。

确定任务的适当大小后，用户必须使用 HLS 编译器选项来最优化算法实现。对于图 5-1 中的代码，FPGA 实现的目标图像大小为 1080 个像素 （每秒 60 帧）。这可以解释为硬件模块能够以 150 MHz 的时钟频率来处理 1920 x 1080 个像素，每个时钟周期传入数据率为 1 个像素。

数据率最优化

在 HLS 编译器中，代码最优化从基线编译开始。基线编译的目的是确定实现瓶颈所在位置，并设置参考点来测量不同最优化的影响。基线编译会以尽可能少的 FPGA 资源和最低的输入数据率来构建算法实现。在本章示例中，基线编译生成的传入数据率为每 40 个时钟周期 1 个像素。

使用 HLS 编译器时，在生成的实现中增加输入数据率和并行度的方法是采用流水线最优化。正如第 2 章"关于 FPGA"和第 3 章"硬件设计的基本概念"中所述，流水线化可将大型计算分割为可并行执行的较小的阶段。适用于循环后，流水线化即可设置循环的启动时间间隔 (II)。

循环 II 可通过影响启动 i+1 迭代所需的时钟周期数来控制循环的输入数据率。设计人员可以选择在算法代码中应用流水线最优化的位置。图 5-4 显示了将流水线编译指示应用于窗口计算的过程。

注释：如需了解有关 HLS 编译器中可用的编译指示的详细信息，请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 1]。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
              for(colOffset= -1; colOffset <= 1; colOffset++){
#pragma HLS PIPELINE
                  x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                  y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
              }
          }
     edge_weight =ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

图 5-4：应用于窗口计算的循环流水线编译指示

图 5-4 中的示例显示的是将流水线最优化作为编译器编译指示直接应用于算法源代码。对于代码中的该级别而言，流水线编译指示的影响是每个时钟周期计算 3x3 筛选窗口中的 1 个域。因此，要执行 3x3 窗口中的乘法计算需要 9 个时钟周期，要生成结果像素则还需要 1 个时钟周期。在应用级别，这意味着输入采样率为每 10 个时钟周期 1 个像素，这不足以满足应用要求。

图 5-5 显示了将流水线编译指示应用于跨越多个图像列的 j 循环的过程。通过在此循环上应用流水线，HLS 实现可达到每个时钟周期 1 个像素的输入数据率。为达成此新输入数据率，编译器首先完全展开窗口计算循环，以便所有颗粒度乘法均可并行执行。展开过程会例化额外硬件，并在输入图像上将每个时钟周期的内存带宽要求增加至 9 次内存操作。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
#pragma HLS PIPELINE
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
              for(colOffset= -1; colOffset <= 1; colOffset++){
                 x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                 y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
              }
          }
     edge_weight =ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

图 5-5：应用于 J 循环的循环流水线编译指示

虽然 HLS 编译器可以检测到所需内存带宽高于算法中所表述的带宽，但编译器无法自动执行任何更改来影响算法正确性。在此示例中，流水线最优化所需的 9 次并行内存存取操作无法通过超出 HLS 生成模块边界的内存来得到满足。

无论外部内存上有多少端口，HLS 生成模块都只能连接到单个端口，且每个时钟周期内该端口只能执行一次传输事务。因此，必须修改算法，将内存带宽要求从模块输入端口移到由 HLS 编译器生成的内存。此内部内存类似于处理器中的高速缓存。对于 Sobel 边缘检测之类的图像处理算法，此本地内存称为“行缓存 (line buffer)”。

行缓存属于多 bank 内部内存，可支持生成实现在每个时钟周期内并行访问来自 3 个不同行的像素。开始任意计算操作前，实现行缓存的算法都必须分配时间来为结构填充充足的数据，以满足计算要求。这意味着要满足每个计算结果 9 次访问的内存要求，算法必须考量数据穿越行缓存的移动过程以及算法更改所生成的额外带宽。

图 5-6 显示了图像像素穿越行缓存的移动过程。

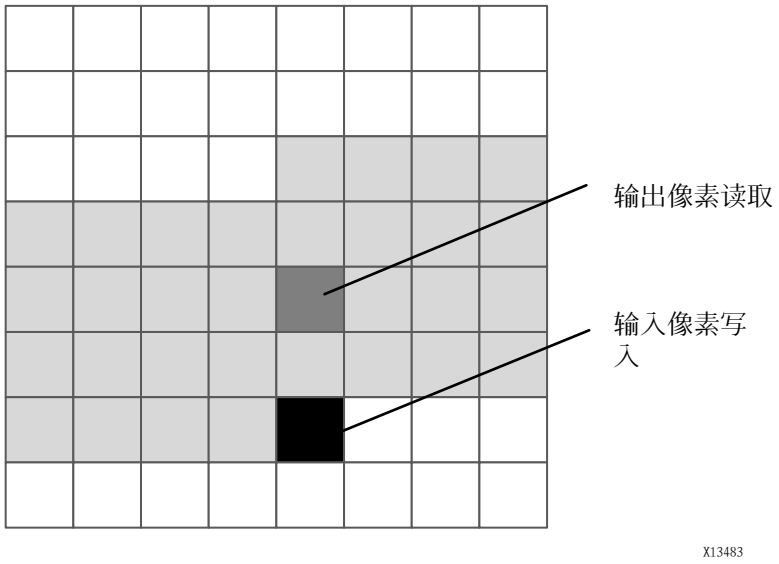


图 5-6：行缓存中的数据移动

浅灰色框表示此内存结构当前存储的像素。此块仅用于存储正常执行功能所需的最小数量的像素，并非用于存储整个图像。如前文所述，添加此内存结构会导致输入像素采样与输出像素计算之间出现延迟。对于 3x3 窗口操作 （如图 5-5 中代码所示），行缓存必须存储 2 个完整的图像行以及第 3 行的前 3 个像素，然后才能计算第一个输出像素。深灰色框和黑色框用于表示此时延。黑色框会高亮来自原始图像的下一个输入像素的写入位置。深灰色框可显示输出图像中当前计算的像素的位置。

HLS 使用来自 FPGA 结构的 BRAM 实现行缓存。这种双端口内存元素按 bank 排列组合，每个 bank 对应于 1 行。因此，可用于算法计算的内存带宽将从原先每个时钟周期 1 个像素提高 3 倍，达到每个时钟周期 3 个像素。这仍不足以满足每个时钟周期 9 个像素的要求。

为满足每个时钟周期 9 个像素的要求，除了行缓存外，设计人员还必须向算法源代码添加内存窗口。内存窗口是使用来自 FPGA 结构的 FF 资源实现的存储元素。此内存中每个寄存器均可供所有其它寄存器独立访问和同时访问。从逻辑角度上，由 FF 元素组成的内存可采用最适合 C/C++ 中的算法描述的任何形式。

图 5-7 显示了 Sobel 边缘检测算法的内存窗口。

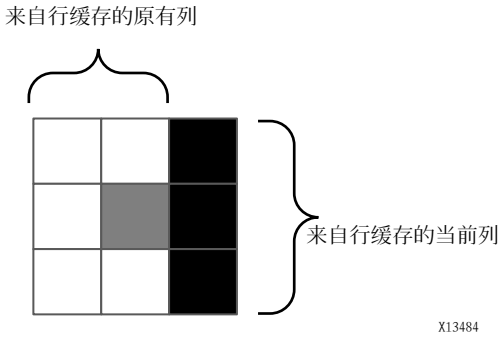


图 5-7：内存窗口

灰色中心像素高亮的像素即为计算其颗粒度的像素。黑色列表示行缓存所提供的 3 个像素。在每个时钟周期内，窗口内容都会向左移，以便为来自行缓存的新的列预留空间。窗口内存的数据复用和分布实现可以提供算法所需的 9 次内存操作。此内存不会对设计造成额外时延。窗口数据移动操作与行缓存的数据移动操作并行执行。

图 5-8 中显示的是通过分层内存架构从输入到计算的总体数据移动过程。

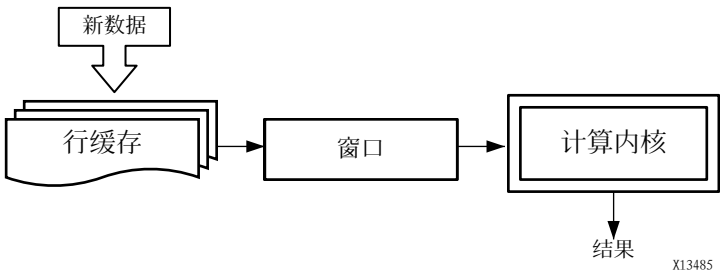


图 5-8：从输入到计算的数据移动

图 5-9 显示的是实现分层内存架构所需的算法代码更改。此分层架构允许 HLS 生成的实现达成每个时钟周期 1 个像素的输入数据率。在此代码示例中，算法的计算内核位于 sobel_operator 函数中。此代码的主要更改在于将行和列循环各自扩展一个迭代。此扩展会考量行缓存导致的额外任务执行时间。此外，行缓存写入操作受基于原始图像边界的 if 条件的保护。算法输出写操作基于输出图像定位，此定位与原始图像相比存在 1 行和 1 列的偏移。

如图 5-9 中所示，围绕计算的应用可包含嵌入式控制语句，形式包括 for 循环、 if-else 语句等。这种算法的关键特征在于任务执行期间，其函数和行为是固定的。 HLS 生成模块根据给定配置来处理数据批次。不同任务的配置可更改，但任务执行期间不可更改。

提示：行缓存操作库包含在 HLS 编译器所提供的视频库内。如需了解更多信息，请参阅 《Vivado Design Suite 用户指南：高层次综合》(UG902) [参照 1]。

```
for(row = 0; row < rows+1; row++){
     for(col = 0; col < cols+1; col++){
          if(col < cols){
                    buff_A.shift_up(col);
                    temp = buff_A.getval(0,col);
          }
          if(col < cols & row < rows) {
                buff_A.insert_bottom(rgb2y(input_pixel[row][col]),col);
          }
          buff_C.shift_right();
          if(col < cols){
                 buff_C.insert(buff_A.getval(2,col),0,2);
                 buff_C.insert(temp,1,2);
                 buff_C.insert(rgb2y(tempx),2,2);
          }
          if( row <= 1 || col <= 1 || row > (rows-1) || col > (cols-1)){
                 edge.R = edge.G = edge.B = 0;
          }
          else{
                 edge = sobel_operator(&buff_C);
          }
          if(row > 0 && col > 0){
                  AXI_PIXEL output_pixel;
                  output_pixel.data = (edge.B, edge.G);
                  output_pixel.data = (output_pixel.data, edge.R);
                  out_pix[row-1][col-1] = output_pixel;
         }
       }
     }
}
```

图 5-9：含行缓存的 Sobel 边缘检测代码

XILINX

第 6 章

围绕控制的算法

简介

围绕控制的算法是可在任务执行期间根据系统级因素进行更改的算法。围绕计算的算法在任务执行期间对所有数据值应用相同操作，而围绕控制的算法则根据当前输入端口状态来判定其操作。本章描述了利用 Vivado® HLS 编译器来最优化这种类型的应用的最佳实践。

以 C/C++ 来表述控制

描述最佳实践前，重要的是了解在 C 语言和 C++ 语言中表述控制的方式。

循环

循环是用于表述迭代计算的基本编程结构。像所有编译器一样，HLS 同样允许以 for 循环、while 循环和 do-while 循环的方式来表述循环。在使用 Vivado HLS 编译的所有应用类型中都支持此构造。正如第 5 章"围绕计算的算法"中的 Sobel 边缘检测示例中所示，循环是捕获 C/C++ 中的计算密集型算法所不可或缺的。

图 6-1 显示了 for 循环的示例以及 Vivado HLS 编译的影响。它展示了 Vivado HLS 编译在单次 FPGA 实现过程中生成计算逻辑和控制逻辑的过程。不同于 FPGA 结构的前几代代码编译器， Vivado HLS 编译器不区分控制语言构造和计算语言构造。利用此图中的代码， HLS 能为循环中的数学运算生成流水线化数据路径。这种实现可以通过在循环迭代期间和跨循环迭代都并行执行计算来降低执行时延。除此逻辑外， Vivado HLS 实现还会嵌入循环控制器逻辑。循环控制器逻辑会规定用于计算 y 值的硬件执行次数。

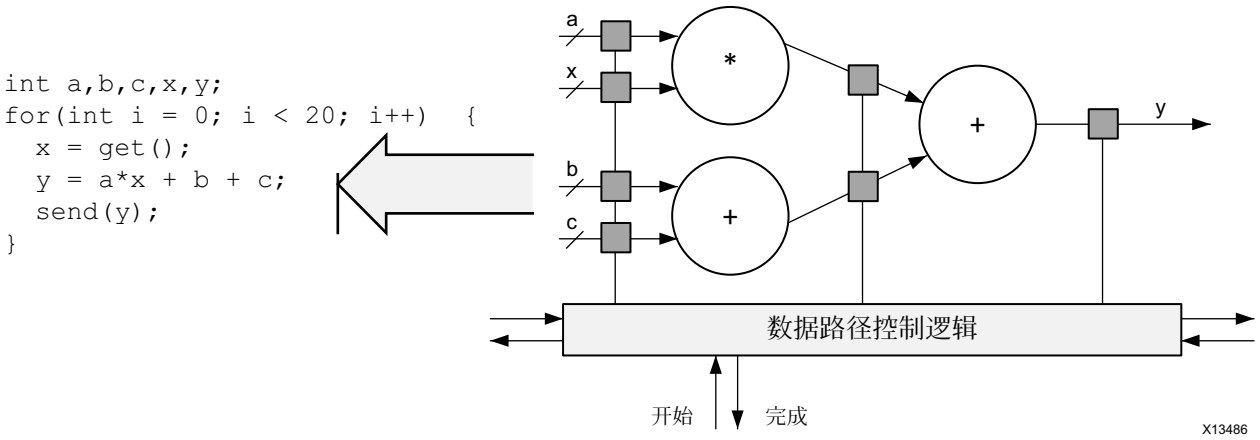


图 6-1：循环示例

条件语句

条件语句通常以 C 和 C++ 中的 if-else 语句来表述。在硬件实现中，这导致二选一结果：根据触发器值生成 2 个结果，或者生成 2 条执行路径。这种实用构造允许设计人员在变量级别或函数级别充分发挥对于算法的控制能力。这 2 种用例都受到 HLS 编译器的充分支持。

图 6-2 显示的示例 if-else 语句中， if 语句会在算法中的 2 个不同函数之间进行选择。 Vivado HLS 编译器生成的实现会为 function_a 和 function_b 分配 FPGA 资源。这两条硬件电路将并行运行并平衡，从而在同一个时钟周期内生成结果。原始源代码中的条件触发器将从计算所得 2 个结果中进行选择。

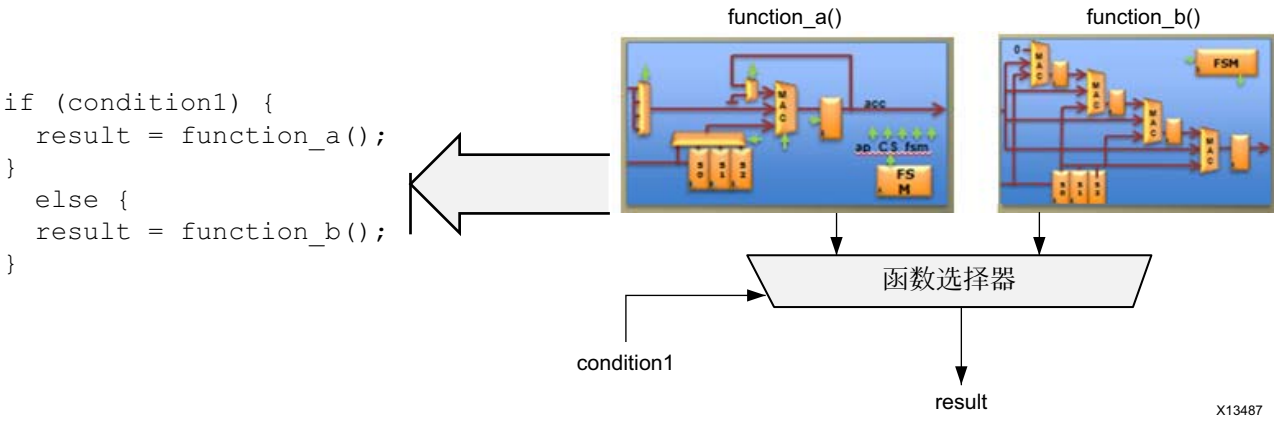


图 6-2：If-Else 示例

Case 语句

Case 语句用于根据输入变量值来定义程序中操作或时间的具体顺序。虽然此构造也可在围绕计算的算法中使用，但在围绕控制的应用中其使用更普遍，在此类应用中系统级更改可直接影响模块执行。并且，在大部分使用模型中， case 语句均可显式定义程序控制区域间的转换。

图 6-3 显示了 case 语句示例以及 Vivado HLS 的编译结果。编译器会将 case 语句转换为硬件有限状态机 (FSM)。 FSM 数组表示状态转换，且对应于代码示例中的 case 转换。 FSM 中的每个状态还包含程序控制区域中的计算逻辑。

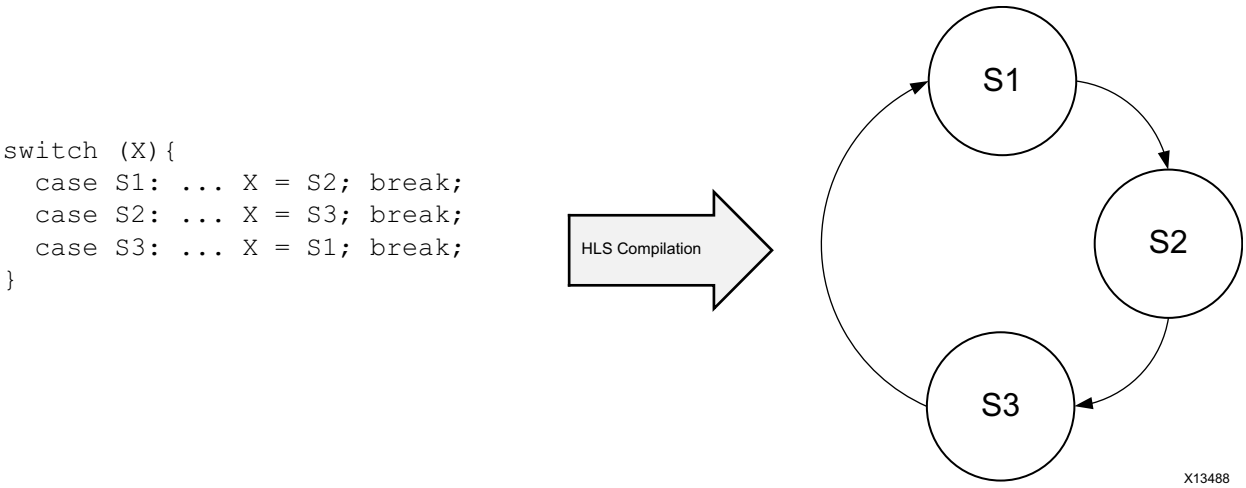


图 6-3：Case 语句示例

控制系统分类

使用代码构造捕获围绕控制的应用后，下一步设计人员需要判断用于运行应用的平台。原先通常选择处理器作为最合适的平台。正如 Zynq®-7000 SoC 所示，在许多用例中，处理器仍是最佳选择。但是， HLS 编译器解决了因状态机最优化和复杂性而导致的 FPGA 结构中控制算法实现的瓶颈问题。设计人员可以选择在处理器上还是在 FPGA 结构中 HLS 生成的定制控制器上运行相同的控制算法。这两者的选择取决于算法响应时间要求和 FPGA 结构资源的用量。

表 6-1 显示了按外部事件响应时间分类的控制算法。

表 6-1：控制系统分类

控制类型	时钟周期中的执行预算	建议的实现
非常慢	≥ 1,000,000	X86 处理器、 DSP 或 Zynq-7000 SoC
慢速	100,000 - 1,000,000	X86 处理器、 DSP 或含 HLS 生成的加速器的 Zynq-7000 SoC
中速	1,000 - 100,000	含 HLS 生成的加速器的 Zynq-7000 SoC
快速	≤ 1,000	HLS 生成的定制控制器

对于所需响应时间非常慢的设计，实现的最佳选择为处理器。此选择可以提供更多空间以便将围绕计算的算法编译到 FPGA 结构中。图 6-4 显示了控制响应时间非常慢的系统示例。

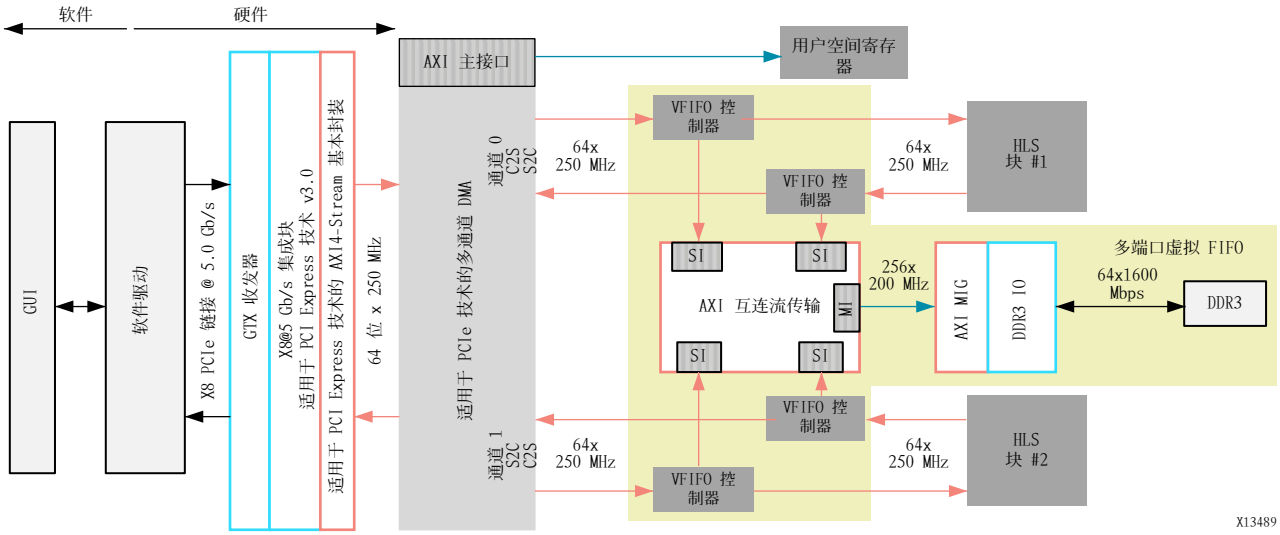


图 6-4：非常慢的控制示例

对于需中等速度的设计，如“慢速”或“中速”类别中所示，实现可以选择采用处理器或者采用 FPGA 结构中的定制逻辑。在此类情况下，控制算法具有 1 个必须作为硬件模块来实现的关键函数。对于此类系统，硬件协同处理器的作用是弥补控制处理器中的通信时延或处理速度不足的问题。图 6-5 显示了需要硬件协同处理元件的系统示例。

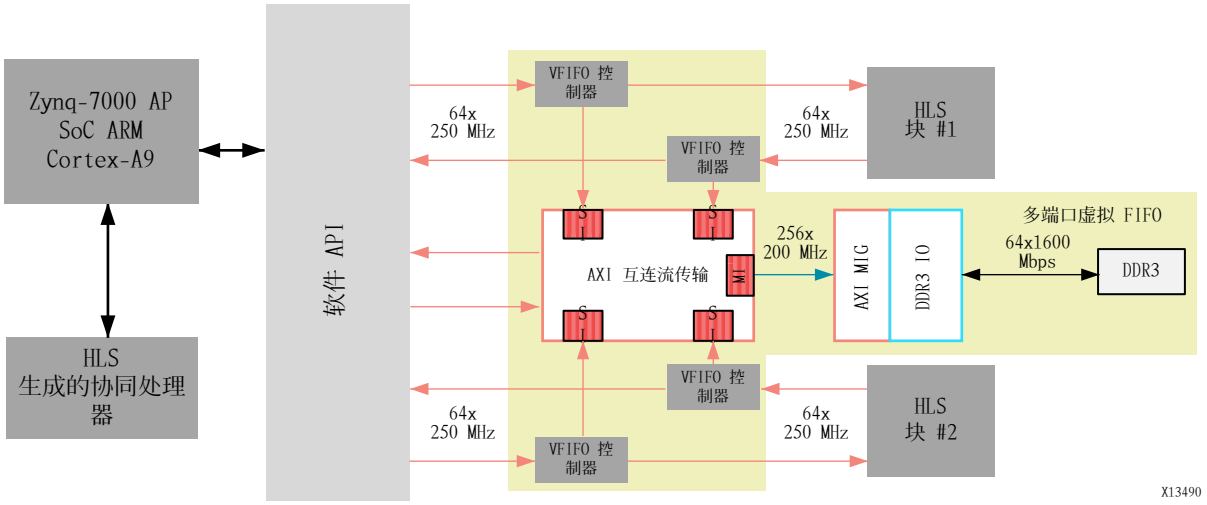


图 6-5：含 HLS 生成的协同处理器的系统示例

最后一个围绕控制的应用类别为“快速”响应时间类别。此类别对应的控制应用所需的响应时间和计算吞吐量均高于处理器可提供的范围。因应 HLS 编译器的引入，归入此类别的算法范围已扩展。例如， HLS 编译器正被越来越多地用于为 Zynq-7000 SoC 生成处理器加速器模块。

UDP 包处理

用户数据报协议 (UDP) 属于计算机网络应用中使用的无状态数据传输协议。此协议不保证包交付，也不处理丢失包的恢复。而是通过有线或无线通道以尽可能快的速度进行包传输。此协议可实现的数据速率使其成为网络电话、视频流传输和其它数据速率重要性高于数据传输中的每个包的接收完整性的应用的适用标准。

虽然此协议不保留包交付和状态追踪记录，但它仍属于围绕控制的应用。 UDP 包处理器的控制领域包括：

- 以线路传输速率来解析传入数据包
- 响应来自网络的控制包
- 格式化数据包以供传输
- 处理传输通道中断

所有这些控制领域都可生成图 6-6 中所示的复杂状态机。在引入 HLS 编译器前，这种级别的复杂控制始终交由处理器来处理，即使因此牺牲性能也不例外。选择这种实现方式的主要原因是在手动设计流程中如此大小的 FSM 难以得到有效表述和平衡。

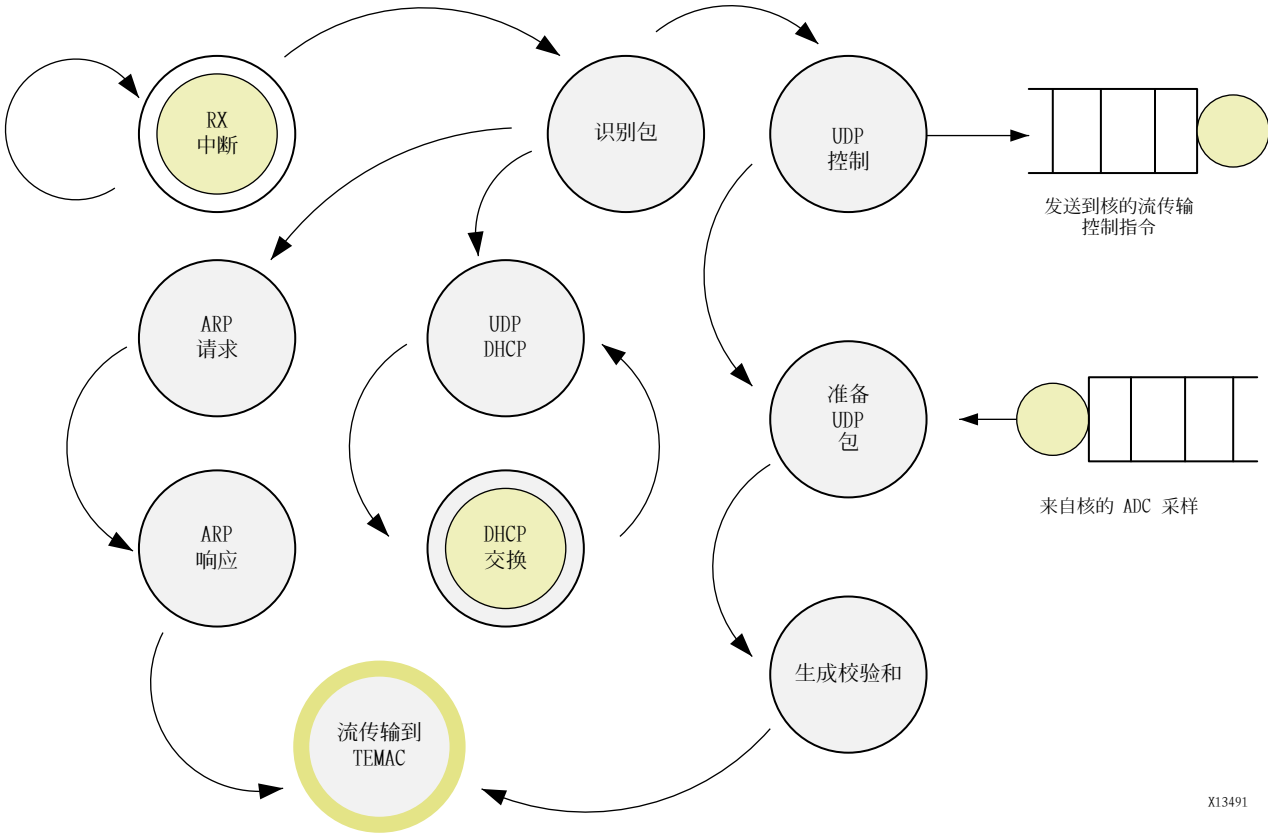


图 6-6：UDP 包处理 FSM

如图 6-6 中所示， UDP 包处理 FSM 是复杂的互连状态网络。每个状态都对应包处理的不同阶段。除状态间的复杂交互之外，每个状态都可能因系统级事件而中断。这些事件可能触发来自应用的状态信息请求，或者重新配置下一个包的处理方式。不同于围绕计算的应用，包处理的任务大小无法精确定义。每个包都必须分析，这意味着只要器件通电，任务持续时间就是无限的。 UDP 处理 FSM 的实现从顶层函数签名开始。

图 6-7 显示了 UDP 包处理引擎的顶层函数签名，目标是使用 HLS 编译器完成 FPGA 实现。在此函数中，使用数组对该模块与系统其余部分之间的物理通信缓存进行建模。此外还需注意，使用 volatile 关键字来标记不属于数组的每个函数变量。如图 6-6 中所示，此控制器必须能够在执行的任意阶段内处理来自系统的中断。此要求的问题在于 C 和 C++ 中指定的函数变量行为。

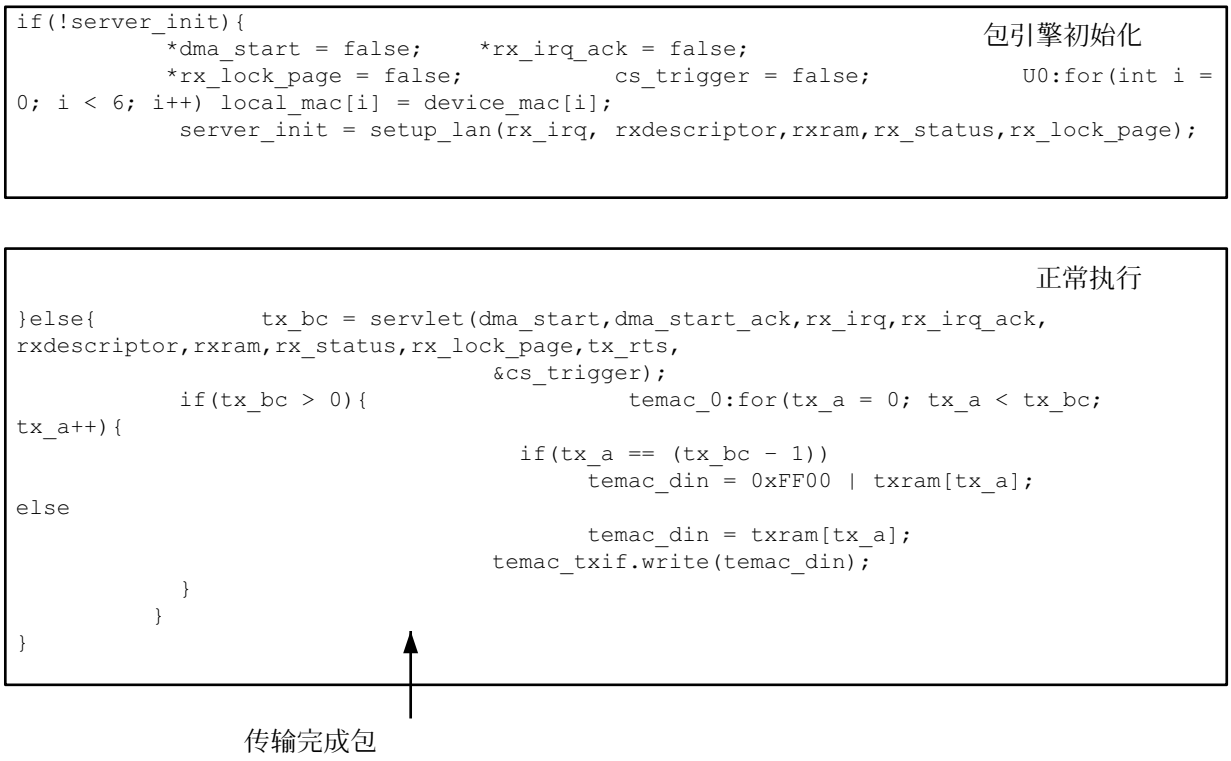
```
bool udp_proc(const Xuint8 device_mac[6],
                           const Xuint32 rxdescriptor[RX_DESCRIPTOR_RAM_SIZE],
                           const Xuint8 rxram[MAX_RX_RAM],
                           const volatile bool *dma_start_ack,
                           const volatile bool *rx_irq,
                           const volatile bool Xuint8 *rx_status,
                           const volatile bool *tx_rts,
                           volatile bool *dma_start,
                           volatile bool *rx_irq_ack,
                           volatile bool *rx_lock_page)
```

图 6-7：UDP 处理器函数签名

在 C 和 C++ 中，使用函数调用时，在函数内存空间内的本地副本中对函数变量进行采样和存储。这意味着不仅可能将相同变量存储在多个内存空间内，而且 C/C++ 程序直到下次函数调用时才能检测变量值变更。 volatile 关键字正是此问题的语言解决方案。嵌入式软件开发人员都很熟悉此构造，它会告知 C/C++ 编译器，在函数调用期间，变量值可能发生更改。因此，每次在代码中使用 volatile 变量时，都必须从函数端口直接访问此变量。虽然此语言构造修复了数据访问问题，但它并未移除变量的内部副本。

跨内存空间复制数据的问题可通过 const 限定符来解决。将此限定符应用于函数端口时，编译器即可避免在函数内存空间内创建变量的本地副本。改为在变量端口上直接执行读写操作。在硬件中，使用 const volatile 限定符还可支持系统在执行任务期间对外部输入进行响应，并减少响应时延。

图 6-8 显示了封装 UDP 控制 FSM 的主要处理过程的代码。




```
if(!server_init){
          *dma_start = false;     *rx_irq_ack = false;
          *rx_lock_page = false;           cs_trigger = false;          U0:for(int i =
0; i < 6; i++) local_mac[i] = device_mac[i];
           server_init = setup_lan(rx_irq, rxdescriptor,rxram,rx_status,rx_lock_page);
```

```
}else{            tx_bc = servlet(dma_start,dma_start_ack,rx_irq,rx_irq_ack,
rxdescriptor,rxram,rx_status,rx_lock_page,tx_rts,
                                  &cs_trigger);
            if(tx_bc > 0){                  temac_0:for(tx_a = 0; tx_a < tx_bc;
tx_a++){
                                    if(tx_a == (tx_bc - 1))
                                         temac_din = 0xFF00 | txram[tx_a];
else
                                         temac_din = txram[tx_a];
                                  temac_txif.write(temac_din);
            }
          }
}
```

图 6-8：UDP FSM 主要函数

UDP 控制 FSM 的执行可分为初始化和正常执行阶段。一旦 FPGA 实现完成复位，就会即刻开始初始化阶段。在此阶段中，状态标记设置为默认值，并从内存加载块的媒体访问控制 (MAC) 地址。 MAC 地址是唯一网络标识，动态主机配置协议 (DHCP) 地址即分配到该网络标识。当 UDP 控制器可广播其地址后，它就会开始处理网络控制包以向该网络请求并寄存内部协议 (IP) 地址。在网络中正确寄存控制器后，它会切换到正常运行模式，并开始生成 UDP 包。除特定功能外，此代码还可演示在单个围绕控制的应用内控制编码元素和计算编码元素的组合方式。

图 6-8 中的代码显示的是基于 2 个执行阶段的单层控制层级。实际上，围绕控制的应用比此示例更复杂，并展现出层级控制结构。以处理器表述控制层级的相同方式来捕获控制层级的能力正是 HLS 与用于硬件的其它软件编译器之间的主要差异之一。

图 6-9 显示了针对 HLS 编译器来表述层级控制的方式示例。此图是图 6-8 中的 servlet 函数的 1 个分段。servlet 函数可控制初始化后 UDP 控制器的所有操作阶段。正如此代码所示，该模块与系统级信号进行持续性交互，以判定下一步操作。此外，此编码样式可保留嵌套的 case 语句，并混用处理器代码中常用的计算函数。这有助于捕获 C/C++ 中的功能，并辅助实现从处理器到 FPGA 的代码移植。

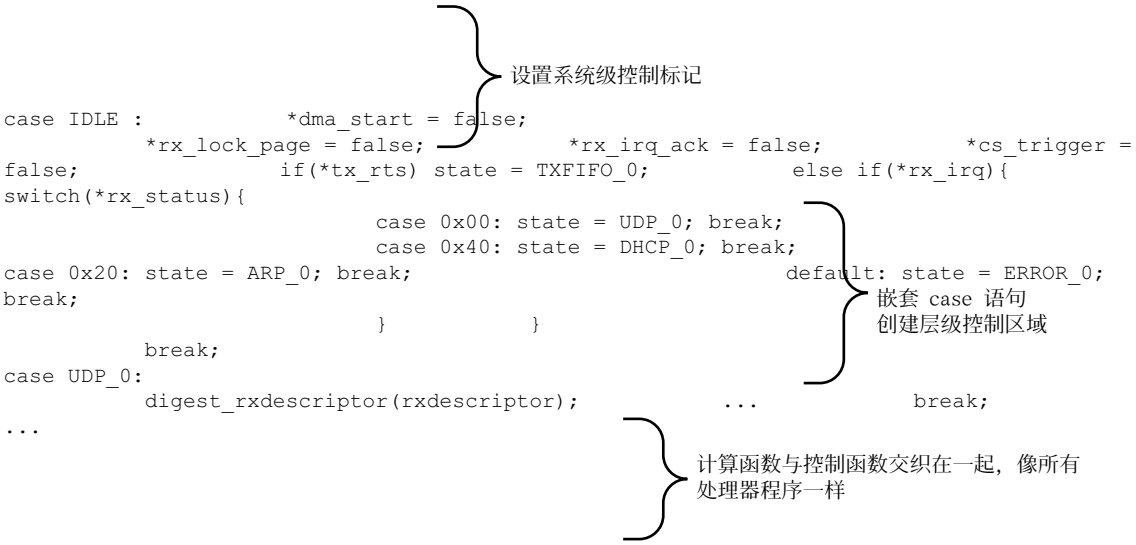


图 6-9：UDP 处理中的层级控制区域

在含 HLS 编译器的 FPGA 上可编译和实现围绕控制的应用 （例如， UDP 处理器）。因此，要判断是否实现此类代码，只需判断是需要控制代码还是需要应用中的所有其它函数即可。通过使用 HLS 编译器开发完整应用，用户即可判断围绕控制的函数和围绕数据的函数在设计中为达成不同性能点所需的资源量。由于 HLS 编译器生成多种 what-if 场景，因此可浏览设计变量 （例如，吞吐量对比面积对比时延）。

XILINX

第 7 章

软件验证和 Vivado HLS

简介

就像处理器编译器一样， Vivado® HLS 编译器输出的质量与正确性取决于输入软件。本章对适用于 Vivado® HLS 编译器的推荐软件质量保证技巧进行了回顾。它提供了典型编码错误示例及其对于 HLS 编译的影响，以及每个问题的可能的解决方案。其中还包含一个有关在 C/C++ 仿真级别无法完全验证程序行为时的变通方法的段落。

软件测试激励文件

任何 HLS 生成的模块的验证都需要使用软件测试激励文件。软件测试激励文件可提供如下重要功能：

- 证明以 FPGA 实现为目标的软件可正常运行且不会出现分段错误
- 证明算法的功能正确性

无论是在 HLS 中还是在任何其它编译器中，分段错误都会导致问题。但导致此问题的编码错误的检测方法不尽相同。在基于处理器的执行中，分段错误是由于程序尝试访问处理器未知的内存位置而导致的。此错误最常见的原因是用户程序尝试访问内存中与指针地址关联的位置，而内存尚未分配并连接到该指针。在运行时间根据以下事件顺序采用相对较为直接的方式检测此错误：

1. 处理器检测内存访问违例并通知操作系统 (OS)。
2. OS 向导致错误的程序或进程发送信号。
3. 接收到来自 OS 的错误信号后，程序会终止，并生成核转储文件以供分析。

在 HLS 生成的实现中分段错误较为难以检测，因为不存在监控程序执行的处理器或操作系统。分段错误的唯一指标为电路生成的结果值不正确。这本身并不足以判断分段错误的根本原因，因为有多个问题可能触发计算结果错误。

建议：使用 HLS 时，建议设计人员确保软件测试激励文件在处理器上没有任何问题的前提下编译并执行函数。这样可保证 HLS 生成的实现不会导致分段错误。

软件测试激励文件的另一项用途是证明以 FPGA 执行为目标的算法的功能正确性。对于生成的硬件实现，HLS 编译器仅保证功能与原始 C/C++ 代码等效。因此，需要存在强大的软件测试激励文件以最大限度减少硬件验证工作。

强大的软件测试激励文件具备在算法的软件实现上执行数千甚至上百万次数据集测试的能力。这使设计人员能够以极高的置信度等级来断言算法捕获方式正确。但即使拥有大量测试矢量，有时在 FPGA 设计的硬件验证期间，仍可能在 HLS 生成的输出中检测到错误。在硬件验证期间检测到功能错误意味着软件测试激励文件不完整。通过对 C/C++ 执行过程应用攻击性测试矢量即可找到算法中的错误语句。

重要提示：不得在生成的 RTL 中直接修复错误。功能正确性方面的任何问题都是由于软件算法的功能正确性直接导致的结果。

提示：如果软件测试激励文件通过 HLS 来实践的算法以 FPGA 实现为目标，则编码样式并无任何限制。软件工程师可自由使用任何有效的 C/C++ 编码样式或构造来对算法的功能正确性进行完整彻底的测试。

代码覆盖率

代码覆盖率表示设计中由测试激励文件代码来实践的语句的百分比。像 gcov 之类的工具可生成该指标，以便标示用于实践算法的测试矢量的质量。

测试激励文件最低限度必须达成至少 90% 的代码覆盖率得分才能被视为合格的算法测试。这意味着测试矢量可触发 case 语句、if-else 条件语句和 for 循环中的所有分支。除总体覆盖率指标外，代码覆盖率工具所生成的报告还可提供有关任一函数中哪些部分已执行、哪些部分未执行的信息。

图 7-1 显示了利用 gcov 测试的应用示例。

测试平台代码

```
int main()
{
   int i;
   int B[10];
   int C[10];
   int result;

   for(i=0; i < 10; i++){
        B[i] = i;
        C[i] = i;
   }
   result = example(B,C);
   return result;
}
```

算法代码

```
int example(int B[10], int C[10])
{
   int i;
   int A=0;

   for(i=0; i < 10; i++){
         A += B[i] * C[i];
         if(i == 11)
              A = 0;
   }
   return A;
}
```

图 7-1：代码覆盖率应用示例

要运行 gcov，需以其它标记编译代码，此类标记应可生成对程序执行进行剖析所需的信息。假定图 7-1 中的代码显示在 `example.c` 文件中，可按图 7-2 中所示命令序列来运行 gcov。

```
gcc -fprofile-arcs -ftest-coverage example.c
./a.out
gcov example.c
```

图 7-2：gcov 命令序列

gcov 结果表明 92.31% 的程序行已执行，这满足 HLS 的最低 90% 的代码覆盖率要求。但 gcov 可提供更有趣的结果，即执行每行代码的次数，如表 7-1 中所示。

表 7-1：gcov 代码分析示例

执行次数	代码行
-	int example(int B[10], int C[10])
1	{
-	int i;
1	int A = 0;
11	for(i=0; i < 10; i++){
10	A += B[i] * C[i];
10	if(i == 11)
NEVER	A = 0;
-	}
1	return A;
-	}

结果显示从未执行 for 循环中发生的 A = 0 赋值。此声明会警告用户，并提供可能原因以及拦截此赋值的条件语句。根据图 7-1 中表述的循环边界，拦截条件语句 i == 11 永远无法判定为 true。算法必须检查此行为是否符合期望。HLS 会检测 C/C++ 中不可达成的语句 （例如， A 赋值为 0），将此类语句判定为死码并从电路中删除。

未初始化的变量

未初始化的变量是由于编码样式欠佳所导致的，在此类编码中设计人员未在声明点将变量初始化为 0。图 7-3 显示了含未初始化的变量的代码片段示例。

```
int A;
int B;
...
A = B * 100;
```

图 7-3：未初始化的变量代码片段

在此代码片段示例中，变量 A 从不会引发任何问题，因为在读取该变量前已对其赋值。此问题是由变量 B 导致的，该变量尚未赋值即已用于计算。这种 B 使用方式在 C 和 C++ 中均归类为未定义行为。虽然部分处理器编译器可通过在声明点向 B 自动赋值 0 来解决此问题，但 HLS 并不使用这种类型的解决方案。

HLS 假定可从生成的实现中优化掉用户代码中的所有未定义行为。这样可触发最优化级联效应，导致电路缩减为空。用户可通过关注生成实现的 RTL 文件是否为空来检测此类错误。

检测此类错误的更好的方法是使用代码分析工具，如 valgrind 和 Coverity。这两种工具都会在用户程序中标记未初始化的变量。就像所有软件质量问题一样，在使用 HLS 编译代码前，必须先解决所有未初始化的变量问题。

出界内存访问

在 HLS 中，内存访问应表述为针对数组执行的操作或者针对外部内存 （通过指针）的操作。对于出界内存访问，应注意 HLS 转换为内存块的数组。图 7-4 显示了具有出界内存访问的代码示例。

```
int A[10];
...
for(i = 0; i < 11; i++){
      A[i] = i + 5;
}
```

图 7-4：出界内存访问示例

此代码尝试将数据写入数组 A，而该数组位于已分配的内存范围之外。在处理器编译器中，此类地址溢出会触发地址计数器复位为 0。这意味着在处理器执行图 7-4 所示代码时，位置 A[0] 的内容为 15，而不是 5。虽然其结果存在功能错误，但此类错误通常不会导致程序崩溃。

对于 HLS，访问无效地址会触发一系列事件，从而导致生成的电路中出现不可恢复的运行时间错误。由于 HLS 实现假定已对软件算法进行了正确验证，因此生成的 FPGA 实现中不包含错误恢复逻辑。因此，图 7-4 中所示代码的实现会向用于存储数组 A 的值的 BRAM 资源元素生成无效的内存地址。随后，此 BRAM 会发出与 HLS 实现的期望不符的错误条件，且该错误将无法得到解决。 BRAM 中无法解决的错误会导致系统挂起，只能通过设备重启来解决。

要在电路编译前捕获此类错误，建议通过动态代码检查工具 （如 valgrind）来执行该工具。 Valgrind 是旨在对 C/C++ 程序进行检查和剖析的工具套件。 valgrind Memcheck 工具可执行经过编译的 C/C++ 程序并监控执行期间的所有内存操作。此工具会标记以下关键问题：

- 使用的未初始化的变量 （图 7-3）
- 无效的内存访问请求 （图 7-4）

建议：使用 HLS 来编译软件函数以供 FPGA 执行前，建议设计人员先解决动态代码检查工具所标记的所有问题。

协同仿真

C/C++ 程序分析和功能测试工具会捕获影响 HLS 实现的大部分问题。但是，这些工具无法验证并行化后 C/C++ 顺序程序是否能保持功能正确性。在 HLS 编译器中通过协同仿真进程可解决此问题。

在协同仿真进程中，通过软件仿真期间所使用的 C/C++ 测试激励文件来实践生成的 FPGA 实现。 HLS 会按对用户透明的方式来处理 C/C++ 测试激励文件与生成的 RTL 之间的通信。在此进程中， HLS 会调用硬件仿真器 （如 Vivado 仿真器）来仿真 RTL 在器件上的运行方式。此仿真的主要目的是检查确认用户提供的并行化指导信息不会破坏算法的功能正确性。

默认情况下， HLS 在并行化前遵循所有算法依赖关系，以确保功能与原 C/C++ 表示法等效。如果无法完全分析算法依赖关系，那么 HLS 会采取保守方法并遵循依赖关系。这可能导致编译器生成保守实现，无法实现应用的目标性能。图 7-5 显示了触发 HLS 中的保守行为的代码示例。

```
for(i=0; i < M; i++){
      A[k+i] = A[i] + .....;
       B[i] = A[i] * .....;
}
```

图 7-5：触发保守的 HLS 实现的依赖关系示例。

此代码会显示在数组 A 和 B 上运行的循环以及在数组 A 上发生的分析问题。在数组 A 中建立索引的过程依赖于循环变量 i 和变量 k。在此示例中，变量 k 表示编译时某个函数参数的值未知。因此， HLS 可以证明写入 A[k+i] 的操作发生位置与读取计算 B[i] 时所使用的 A[i] 的位置不同。基于此不确定性，HLS 会采用相应的算法依赖关系，强制 A[k+i] 和 B[i] 的计算过程按原 C/C++ 源代码中所述顺序发生。用户能覆盖此依赖关系，并强制 HLS 生成并行计算 A[k+i] 和 B[i] 的电路。此覆盖的效果仅影响生成的电路，因此可供协同仿真认证。

使用协同仿真时请谨记，这是处理器上执行的并行硬件仿真。因此，它比 C/C++ 仿真慢约 10,000 倍。此外，还需谨记，协同仿真的目的并不是验证算法的功能正确性。而是检查确认用户给予 HLS 编译器的指导信息并未破坏算法。

建议：建议仅在算法功能验证期间使用的一小部分测试矢量上运行协同仿真。

当无法执行 C/C++ 验证时

HLS 用例主要发生在可通过 C/C++ 仿真来完全验证其功能正确性的算法中。但在某些用例中，执行 HLS 编译前仍无法完全验证算法的 C/C++ 表示法。图 7-6 显示了此类代码示例。

```
case IDLE :
            *dma_start = false;
            *rx_lock_page = false;
            *rx_irq_ack = false;
            *cs_trigger = false;

            if(*tx_rts) state = TXFIFO_0;
            else if(*rx_irq){
                  switch(*rx_status){
                              case 0x00: state = UDP_0; break;
                              case 0x40: state = DHCP_0; break;
                              case 0x20: state = ARP_0; break;
                              default: state = ERROR_0; break;
                  }
            }
            break;
```

图 7-6：使用 Volatile 类型的代码示例

此代码显示了以 C 语言描述的 UDP 包处理引擎的片段。在此示例中，声明的所有指针均含 volatile 关键字。volatile 关键字常用于器件驱动开发，它可警告编译器，指针连接到的存储元件在函数执行期间可能发生更改。每次在源代码中指定此类指针时，都必须对其进行读取或写入。 volatile 关键字还会将合并指针访问的传统编译器最优化关闭。

volatile 数据的问题在于，在 C/C++ 仿真中无法完全验证代码行为。 C/C++ 仿真无法在测试的函数执行过程中更改指针的值。因此，只能在 HLS 编译后通过 RTL 仿真来完全验证此类代码。用户必须编写 RTL 测试激励文件才能在所有情况下为 C/C++ 源代码中的每个 volatile 指针测试生成的电路。在此情况下，不适合使用协同仿真，因为它受到 C/C++ 仿真中可用的测试矢量的限制。

XILINX

第 8 章

多个程序的集成

简介

就像大部分处理器都会运行多个程序来执行应用一样，FPGA 同样可以例化多个程序或模块来执行某一应用。本章主要聚焦如何在 FPGA 中连接多个模块以及如何使用处理器来控制这些模块。本章中的示例使用赛灵思 Zynq®-7000 SoC 来演示处理器与 FPGA 结构之间的互连。

Zynq-7000 SoC 是以低功耗软件执行为目标的首个新型器件。此器件将 Arm® Cortex™-A9 多核处理器与 FPGA 结构组合到单个芯片中。此器件中的集成级别可消除与协同处理器或加速解决方案相关联的通信时延和瓶颈。此器件还可彻底消除在处理器上运行的代码与 Vivado® HLS 为 FPGA 编译的代码之间通过 PCIe® 网桥传输数据的需求。改为使用 Advanced eXtensible Interface (AXI) 协议实现这 2 个计算域之间的互连。

AXI

AXI 是 Arm 高级微控制器总线架构 (AMBA®) 系列微控制器总线的一部分。此标准旨在定义系统中的模块间的数据传输方式。适用于 Zynq-7000 SoC 上运行的应用的 AXI 通信用例包括：

- 内存映射从接口
- 内存映射主接口
- 直接点对点数据流

注释：如需了解有关 AXI 及其对应赛灵思 FPGA 的实现方式的更多信息，请参阅 《AXI 参考指南》 (UG761) [参照 2]。

内存映射从接口

AXI4-Lite 是内存映射从连接，使用的通信机制与基于处理器的系统中的器件驱动相同。处理器代码通过向器件驱动发出函数调用来访问从加速器核。由 Vivado HLS 自动生成的器件驱动通过访问加速器中的寄存器来配置并触发任务执行。这些寄存器驻留在处理器的内存空间中，也可不借助驱动直接访问。

FPGA 结构中的从加速器无法自行启动任何数据传输。尤其是，此类结构不允许加速器启动与主内存的数据传输以完成其任务。图 8-1 中显示了此结构的传输事务图示。此图显示了传输事务期间时钟周期的耗用位置。设计人员了解传输事务顺序和时序预算后即可正确判断此接口的适用性及其对应用性能的影响。

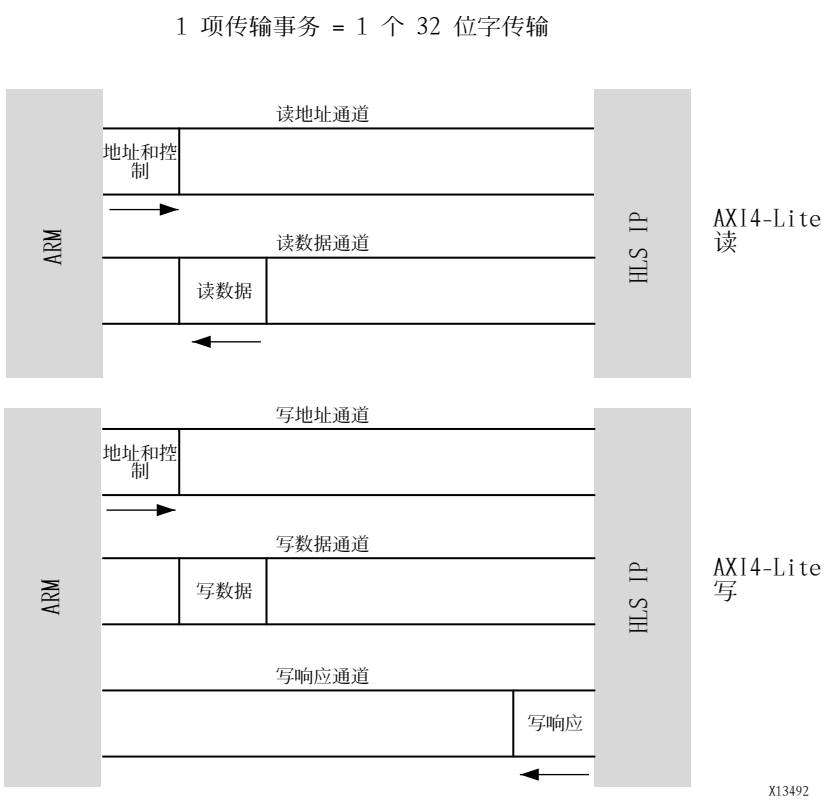


图 8-1：AXI4-Lite 传输事务图示

内存映射主接口

AXI4 是内存映射的主接口，支持 HLS 生成的模块启动到 DDR 内存等器件的数据传输事务，而无需处理器干预。对于含此接口的块，处理器无需从主内存赋值和传输数据，这样即可提高应用计算吞吐量。

请谨记，无法从处理器访问与 AXI4 接口关联的函数端口。因此，建议含 AXI4 接口的模块最好包含部分连接到 AXI4-Lite 接口的函数参数。从接口支持处理器与基本地址进行通信，函数应从该地址访存其任务数据。完成传输事务基本地址设置后，即可从内存与加速模块之间的数据传输中移除处理器。

图 8-2 显示了 AXI4 接口的传输事务时序图。此图显示了传输事务顺序和关联的开销，以支持设计人员判断此接口是否适合特定应用。

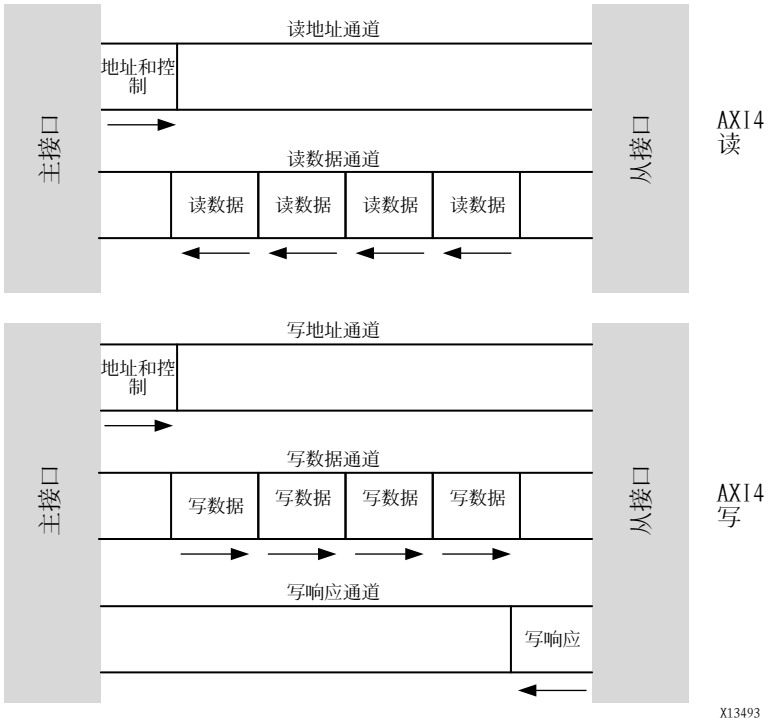


图 8-2：AXI4 传输事务图示

直接点对点数据流

AXI4-Stream 是 FPGA 结构中 2 个模块之间的直接点对点通信通道。就像 AXI4 一样，此传输通道在处理器内存空间内不可见。它也不含与从内存对数据进行寻址和访存操作相关联的任何开销。而是改为在模块间通过 FIFO 来进行数据传输。

AXI4-Stream 是编译到 FPGA 结构上的函数之间的首选数据传输通道，与软件开发中的函数间队列相似。连接到此类数据传输通道的函数可并行运行，并根据通道状态进行自我同步。在数据流输入处连接的函数称为“生产者 (producer)”，只要通道内有空间即可传输数据。在数据流输出处连接的函数称为“使用者 (consumer)”，只要通道显示不为空，即可接收数据。

生产者和使用者均可与 AXI4-Stream 通道进行独立交互。根据通道状态，函数可完成传输事务或者等待通道就绪。只要函数的聚合吞吐量功能满足系统级要求，就不会丢失或跳过数据。

图 8-3 显示了 AXI4-Stream 数据传输通道的传输事务时序图。此通道不提供寻址逻辑，存储空间量由用户定义。默认情况下， AXI4-Stream 深度为 1，即将生产者和使用者置于彼此的 Lock-Step 中。通过更改 AXI4-Stream 通道中的存储空间量可影响生产者与使用者之间的耦合程度。

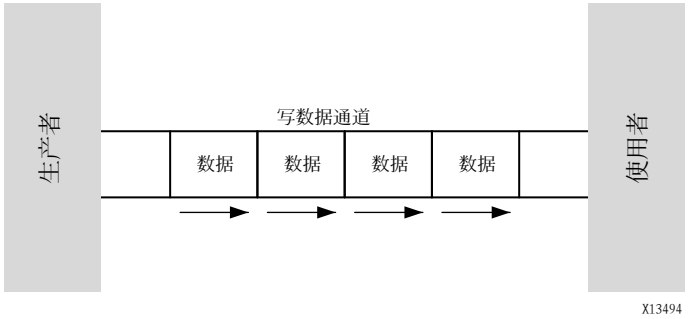


图 8-3：AXI4-Stream 传输事务图示

设计示例：Zynq-7000 SoC 上运行的应用

此设计示例显示了如何将处理器代码转换为 Zynq-7000 SoC 上运行的应用。此示例逐一介绍了移植过程中的如下每个步骤：

- 对处理器代码进行分析和分区
- 在 Vivado HLS 中编译程序
- 在 Vivado IP 集成器中构建系统
- 连接处理器代码和 FPGA 结构函数

注释：Zynq-7000 器件内的 Arm Cortex-A9 处理器可支持单程序执行，也可支持完整的操作系统 （如 Linux）。在这 2 种操作用例中，构建应用所需的步骤都是相同的。因此，本示例着重介绍单程序执行模型来演示应用移植过程。

对处理器代码进行分析和分区

以 Zynq-7000 器件为目标的大部分软件应用最初都属于标准 x86 处理器或 DSP 处理器上运行的应用。因此，设计移植的第一步是针对 Arm Cortex-A9 处理器编译程序，并分析其性能。 Arm 处理器上运行的程序的性能分析数据可提供相关指导信息，以供设计人员选择采用处理器还是 FPGA 结构来对原始代码进行分区。

图 8-4 显示了此示例的原始处理器代码。

```
#include <iostream>
#include "strm_test.h"

using namespace std;

int main(void)
{
   unsigned err_cnt = 0;
   data_out_t hw_result, expected =0
   strm_data strm_array[MAX_STRM_LEN];
   strm_param_t strm_len = 42;

   // Generate expected result
   for(int i = 0; i < strm_len; i++){
         expected += i + 1;
    }

    producer(strm_array,strm_len);
    consumer(&hw_result,strm_array,strm_len);

    // Test result
    if(hw_result != expected){
        cout << "!!!ERROR";
        err_cnt++;
    }else{
         cout << "*** Test Passed";
    }
    cout << "-expected:" << expected;
    cout << " Got:" << hw_result << endl;
    return err_cnt;
}
```

图 8-4：处理器代码

此设计包含 1 个主函数，用于调用 2 个子函数：生产者和使用者。编译到 Arm 处理器后，程序性能分析方法有 2 种：

- 测量时序

  此方法包括利用定时器来检测代码和在处理器上对每个子函数的执行进行定时。

- 使用代码剖析工具

  这种侵入性较低的方法使用 gprof 之类的工具来测量用于某一函数的时间量，并提供有关调用该函数的次数的统计数据。

在此示例中， gprof 的结果表明生产者和使用者函数为应用中的性能瓶颈。因此，决定在 FPGA 结构中实现这 2 个函数。将函数标记为 FPGA 实现后，必须分析函数端口以判断最合适的硬件接口。

图 8-5 显示了生产者函数的签名。

```
void producer(strm_data_t strm_out[MAX_STRM_LEN],strm_param_t strm_len)
```

图 8-5：生产者函数签名

生产者函数包含如下端口：

- strm_out

  此端口是用于函数输出的数组，连接到使用者函数中的对应输入。由于生产者和使用者函数都作为顺序队列来访问此数组，因此最佳硬件接口为 AXI4-Stream。

- strm_len

  此函数参数为输入，必须由处理器提供。因此，此端口必须映射到 AXI4-Lite 接口上。

图 8-6 显示了使用者函数的函数签名。

```
void strm_consumer(data_out_t *dout, strm_data_t
            strm_in[MAX_STRM_LEN], strm_param_t strm_len)
```

图 8-6：使用者函数签名

使用者函数包含如下端口：

- strm_in

  此数组端口与生产者函数连接到同一数组。因此，此端口必须连接到 AXI4-Stream 接口。

- strm_len

  此函数参数的用途与生产者函数中的参数用途相同。此端口与生产者函数中的端口同样作为 AXI4-Lite 接口来实现。

- dout

  这是输出端口。由于设计中没有任何其它 FPGA 结构，因此唯一的选择是将该值传输回处理器。通过发出处理器中断来执行将数据从 FPGA 直接传输到处理器的操作。确认中断后，处理器就会查询其内存空间是否包含此数据。dout 函数参数必须映射到 AXI4-Lite 接口方可从处理器程序访问。

在 Vivado HLS 中编译程序

识别要在 FPGA 结构中运行的函数后，设计人员会为 Vivado HLS 编译准备源代码。在此示例中，生产者函数和使用者函数作为 FPGA 结构中的独立模块来实现。在 FPGA 结构中，每个编译工程均可生成 1 个模块。因此，在此示例中，设计人员必须运行 2 次 HLS，以生成对应模块。

建议：处理多个工程或模块时，建议将源代码分为多个不同文件。这种简单的技巧可以避免任一模块的编译问题影响设计中的其它模块。

HLS 编译可使用工具命令语言 (Tcl) 脚本文件来控制。 Tcl 脚本文件与编译生成文件 (Makefile) 相似，可指示编译器要实现哪个函数以及作为目标的 FPGA 器件。

图 8-7 显示了生产者函数的 HLS 编译的 Tcl 脚本文件。

```
## Project Setup
open_project producer_prj
set_top producer
add_file strm_producer.cpp
add_file -tb strm_consumer.cpp
add_file -tb strm_test.cpp

### Solution Setup
open_solution "solution1"
set_part {xc7z020clg484-1}
create_clock -period 5

### Compilation
csynth_design
export_design -format ipxact
```

图 8-7：生产者函数示例 HLS 脚本文件

此脚本分为以下几个部分：

- 工程设置

  此部分包含源文件和要编译的函数名称。指导 Vivado HLS 编译器操作是将指令或编译指示应用于设计源代码的迭代过程。设计的每次连续优化称为 1 个解法。所有工程至少包含 1 个解法。

- 解法设置

  本部分用于确立时钟频率和要为其编译软件函数的器件。如果设计人员通过使用指令来指导编译器，那么解法指令包含在脚本的此部分中。

- 编译

  此部分用于驱动 RTL 生成和封装。将 HLS 程序汇编为 1 个完整的 Zynq-7000 器件应用需要使用 Vivado IP 集成器这一系统构建工具。 IP 集成器要求将模块打包到软件对象文件的等效文件中。

  注释：如需了解有关 IP 和 IP 集成器的更多信息，请参阅 《Vivado Design Suite 用户指南：采用 IP 进行设计》 (UG896) [参照 3] 和 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》 (UG994) [参照 4]。

生产者和使用者函数的最优化需要编译指示才能确定生成模块及其接口的并行化。图 8-8 显示了生产者函数的最优化代码。

```
#include “strm_test.h”

void producer(strm_data_t strm_out[MAX_STRM_LEN],strm_param_t strm_len)
{
//Interface Behavior
#pragma HLS INTERFACE ap_none port=strm_len
#pragma HLS INTERFACE ap_fifo port=strm_out

//Interface Mapping
#pragma HLS RESOURCE variable=strm_out core=AXIS metadata=“-bus_bundle OUTPUT_STREAM”
#pragma HLS RESOURCE variable=strm_len core=AXIS4LiteS metadata=“-bus_bundle CONTROL_BUS”

for(int i = 0; i < strm_len; i++){
#pragma HLS PIPELINE
     strm_out[i] = i + 1;
    }
}
```

图 8-8：生产者函数的最优化版本

生产者函数通过流水线编译指示来实现并行化。这样创建的实现中的 i 与 i+1 迭代开始时间可相隔 1 个时钟周期。除流水线编译指令外，此代码还可显示接口编译指示的使用方式。

接口编译指示可定义 FPGA 结构中模块的连接方式。定义过程分为接口行为和接口映射。在此示例中，流程如下：

1. strm_out 端口的 ap_fifo 接口编译指示将数组转换为硬件 FIFO。
2. 物理 FIFO 通过资源编译指示映射到 AXI4-Stream 接口。
3. strm_len 函数参数首先分配到 ap_none 接口行为，然后映射到 AXI4-Lite 接口。

   注释：AXI4-Lite 接口处理来自处理器的 strm_len 值的正确排序。因此， HLS 生成的模块无需在此端口上强制执行其它同步。

图 8-9 显示了使用者函数的代码。此函数的最优化和编译指示与生产者函数相同。

```
#include "strm_test.h"

void consumer(data_out_t *dout, strm_data_t strm_in[MAX_STRM_LEN],
                          strm_param_t strm_len)
{
//Interface Behavior
#pragma HLS INTERFACE ap_none port=dout
#pragma HLS INTERFACE ap_none port=strm_len
#pragma HLS INTERFACE ap_fifo port=strm_in

//Interface Mapping
#pragma HLS RESOURCE variable=strm_in core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
#pragma HLS RESOURCE variable=strm_len core=AXIS4LiteS metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE variable=dout core=AXIS4LiteS metadata="-bus_bundle CONTROL_BUS"

data_out_t accum = 0;

    for(int i = 0; i < strm_len; i++){
#pragma HLS PIPELINE
     accum += strm_in[i];
    }
    *dout = accum;
}
```

图 8-9：生产者函数的最优化版本

在 Vivado IP 集成器中构建系统

Vivado IP 集成器是用于系统构建的赛灵思 FPGA 设计工具。此工具的用途之一是将 HLS 编译器生成的块连接到执行用户应用的处理平台中。在软件开发方面， IP 集成器类似于将所有程序对象组合到单一比特流中的连接器。比特流是用于对 FPGA 结构进行编程的二进制文件。

连接处理器代码和 FPGA 结构函数

在 IP 集成器中创建 FPGA 结构编程二进制文件后，设计人员必须创建在处理器上运行的软件。该软件的用途是初始化 FPGA 结构函数、启动执行并接收来自结构的结果。要使整体应用的功能与原始处理器代码等效，在 FPGA 结构中运行的每个函数都需要 Arm Cortex-A9 处理器上运行的代码中的下列功能：

- 地址映射
- 初始化
- 起始函数
- 中断服务例程 (ISR)
- 处理器例外表中的中断寄存
- 用于运行系统的新的主函数

此功能适用于 FPGA 结构中运行的生产者函数和使用者函数。因此，在图 8-10 中仅显示生产者函数的代码。

```
XStrm_producer_Config producer_config={
            0,
            XPAR_STRM_PRODUCER_TOP_0_S_AXI_CONTROL_BUS_BASEADDR
};
```

图 8-10：处理器程序空间内的硬件函数配置

此代码显示了处理器程序空间中的生产者硬件模块的配置。第 1 个参数用于声明要在结构中访问的生产者函数的实例。由于结构中只有 1 个生产者例化，因此该参数值为 0。基本地址定义由 IP 集成器中的系统构建步骤来提供。此地址表示可从处理器访问的内存空间内的内存映射加速器的位置。

图 8-11 显示的初始化函数用于使生产者硬件模块可供处理器上运行的程序使用。

```
int ProducerSetup(){
       return XStrm_producer_Initialize(&producer,&producer_config);
}
```

图 8-11：硬件函数的初始化

图 8-12 将生产者硬件模块设置为开始任务执行。该函数负责将模块中断设置为已知状态并开始任务执行。

```
void ProducerStart(void *InstancePtr){
         XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;
         XStrm_producer_InterruptEnable(pProducer,1);
         XStrm_producer_InterruptGlobalEnable(pProducer);
         XStrm_producer_Start(pProducer);
}
```

图 8-12：硬件函数开始

图 8-13 中显示的 ISR 描述了处理器如何响应来自 FPGA 结构中的生产者函数的中断。ISR 内容取决于具体应用。此代码显示与 Zynq-7000 器件中 HLS 生成的模块进行正确交互所需的最小 ISR。

```
void Producer(void *InstancePtr){
         XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;

         //Disable the global interrupt from the producer
         XStrm_producer_InterruptGlobalDisable(pProducer);
         XStrm_producer_InterruptDisable(pProducer,0xffffffff);

         //clear the local interrupt
         XStrm_producer_InterruptClear(pProducer,1);

         ProducerDone = 1;
         //restart the core if it should be run again
         if(RunProducer){
             ProducerStart(pProducer);
         }
}
```

图 8-13：中断服务例程

所有中断服务例程都必须寄存在处理器例外表中。完成处理器中断控制器初始化后，主程序即可开始执行用户应用。图 8-14 显示了如何为 Zynq-7000 器件配置例外表。

```
int SetupInterrupt()
{
   //This function sets up the interrupt on the ARM
   int result;
   XScuGic_Config *pCfg =
   XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
   if(pCfg == NULL){
          print("Interrupt Configuration Lookup Failed\n\r");
          return XST_FAILURE;
   }
   result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
   if(result != XST_SUCCESS){
            return result;
   }
   //self test
   result = XScuGic_SelfTest(&ScuGic);
   if(result != XST_SUCCESS){
              return result;
   }
   // Initialize the exception handler
   Xil_ExceptionInit();
   //Register the exception handler
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
   //Enable the exception handler
   Xil_ExceptionEnable();
   //Connect the Producer ISR to the exception table
   result = XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INT,
                                                  (Xil_InterruptHandler)ProducerIsr,&producer);
   if(result != XST_SUCCESS){
                  return result;
    }
    //Connect the Consumer ISR to the exception table
    result = XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR,
                                                 (Xil_InterruptHandler)ConsumerIsr,&consumer);
    if(result != XST_SUCCESS){
                  return result;
    }
    //Enable the interrupts for the Producer and Consumer
    XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INTR);
    XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR);
     return XST_SUCCESS;
}
```

图 8-14：处理器例外表的配置

图 8-15 显示了应用的新的主程序。完成硬件设置和处理器环境配置后，对于此示例，处理器上已无任何其它计算。所有计算均已通过使用 HLS 编译迁移到 FPGA 结构。在此情况下，处理器用途为在每个硬件模块上启动任务并在模块完成任务后收集结果。

```
int main()
{
  Init_platform();

  print("Producer Consumer Example\n\r");
  int length;
  int status;
  int result;
  length = 50;
  printf("Length of stream = %d\n\r",length);

  status = ProducerSetup();
  if(status != XST_SUCCESS){
     print("Producer setup failed\n\r");
  }
  status = ConsumerSetup();
  if(status != XST_SUCCESS){
      print("Consumer setup failed\n\r");
  }
  //Setup the interrupt
  status = SetupInterrupt();
  if(status != XST_SUCCESS){
       print("Interrupt setup failed\n\r");
   }

  XStrm_consumer_SetStrm_len(&consumer, length);
  XStrm_producer_Set_Strm_len(&producer,length);

  ProducerStart(&producer);
  ConsumerStart(&consumer);

  while(!ProducerDone) print("waiting for producer to finish\n\r");
  while(!ConsumerResult) print("waiting for consumer to finish\n\r");

  result = XStrm_consumer_GetDout(&consumer);
  printf("Consumer result = %d\n\r",result);
  print("Finished\n\r");

  cleanup_platform();

  return 0;
}
```

图 8-15：处理器主函数

XILINX

第 9 章

完整应用的验证

简介

在 FPGA 设计中，完整的应用系指可实现由某项设计的软件表示法所捕获的功能的硬件系统。在 FPGA 上使用 Vivado® HLS 编译器可实现的系统分 2 种主要类别：

- 独立计算系统
- 基于处理器的系统

独立计算系统

独立计算系统是由 1 个或多个 HLS 生成的模块所创建的 FPGA 实现，这些模块连接在一起可实现软件应用。在此类系统中，算法配置是固定的，在器件配置期间加载。由 HLS 编译器生成的模块连接到外部 FPGA 管脚进行数据传输和接收传输事务。这是可验证的最简单的系统类型。独立系统的验证分为如下几个阶段：

- 模块验证
- 连接验证
- 应用验证
- 器件验证

模块验证

在第 7 章"软件验证和 Vivado HLS"中详述了 HLS 生成块的模块验证。在软件仿真和协同仿真中完成块的完整功能正确性验证后，设计人员必须测试块的系统内部容错能力。

软件仿真和协同仿真都聚焦于测试封闭环境下算法的功能正确性。即通过测试算法和编译的模块来确保以理想方式处理所有输入和输出时的功能正确性。这种程度的彻底测试有助于确保数据提供给模块后的正确性。它还可排除模块的内部处理核导致错误的可能性，从而帮助降低后续阶段的验证压力。此方法无法处理的唯一模块级问题是模块是否能从与其接口的错误握手完全恢复。

系统内部测试用于测试 HLS 生成的模块对于错误切换其输入输出端口的响应方式。此测试的目的是排除因 I/O 行为导致错误而崩溃或者对测试中的模块产生不利影响的可能性。此方法中的各类错误用例包括：

- 勘误表时钟信号切换
- 复位操作和随机复位脉冲
- 输入端口以不同速率接收数据
- 输出端口以不同速率进行采样
- 接口协议违例

这些测试作为系统级行为示例，可确保 HLS 生成的模块在所有情况下都能按期望的方式运作。此阶段所需的测试量取决于接口类型和集成方法。通过使用 HLS 默认设置来生成兼容 AXI 的接口，设计人员即可避免编写详尽的系统级错误行为测试激励文件。兼容 AXI 的接口已由 HLS 编译器开发者经过完整测试和验证。

连接验证

连接验证是用于检查应用中的模块彼此正确相连的一系列任务。与模块验证相同，所需测试量取决于系统集成方法。正如第 8 章"多个程序的集成"中所提到的，可手动汇编应用，或者也可利用 FPGA 设计工具来辅助汇编。

在 Xilinx® System Generator 和 Vivado IP 集成器设计流程中都提供了 FPGA 设计工具辅助功能。这些图形化模块连接工具可处理模块连接相关的方方面面操作。在这些流程中，每项工具都会检查应用中每个模块的端口类型和协议合规性。如果每个模块都已完成模块验证，就不必再通过其中任一流程来执行其它用户指示的连接测试。

手动集成流程需要用户在 RTL 中编写应用顶层模块，并手动连接构成应用的每个模块的 RTL 端口。这是最易于出错的流程，必须加以验证。通过使用 HLS 编译器默认设置和为每个模块端口都生成 AXI 接口可减少所需测试量。

对于围绕 AXI 接口构建的系统，通过使用总线功能模型 (BFM) 可验证连接情况。 BFM 可提供经赛灵思验证的 AXI 总线行为和点到点连接。这些模型可用于流量生成器，以便在 RTL 仿真过程中帮助证明 HLS 生成的模块的连接是否正确。

重要提示：请谨记，此仿真的目的只是为了检查连接情况以及数据是否能正常流经系统。连接验证步骤并不验证应用的功能正确性。

应用验证

应用验证是在 FPGA 器件上运行应用的最后一个步骤。流程中之前的步骤聚焦于检查构成应用的各算法的质量以及检查连接情况是否正常。应用验证则聚焦于检查原始软件模型与 FPGA 结果是否匹配。如果应用仅包含 1 个 HLS 生成的模块，那么此阶段与模块验证相同。如果应用由 2 个或更多个 HLS 生成的模块组成，那么验证流程会从原始软件模型开始验证。

设计人员必须从软件模型提取应用输入和输出测试矢量以供在 RTL 仿真中使用。由于硬件实现的构造是通过多个阶段来验证的，应用验证无需采用详尽的仿真。仿真可运行的测试矢量并无限制，设计人员可自行判断 FPGA 实现中所需的矢量数量。

器件验证

在 RTL 中使用自动或手动集成流程汇编程序后，设计会经历额外的编译阶段以生成对 FPGA 进行编程所需的二进制文件或比特流。在 FPGA 设计术语中，将 RTL 编译为比特流的过程称为逻辑综合、实现和比特流生成。生成比特流之后，即可对 FPGA 器件进行编程。硬件正确运行的时间量达到设计人员指定的时间量后，即表示应用验证完成。

基于处理器的系统

对于模块和连接验证阶段，基于处理器的系统与独立系统的验证流程是相同的。主要差异在于某部分应用在处理器上运行。在 Zynq®-7000 SoC 中，这意味着有部分应用在嵌入式 Arm® Cortex™-A9 处理器上运行，而部分应用则由 HLS 编译为在 FPGA 结构上执行。这种分区可能会导致验证困难，但可通过使用如下方法解决：

- 硬件在环 (HIL) 验证
- 虚拟平台 (VP) 验证

硬件在环验证

HIL 验证是一种在 FPGA 结构中对接受测试的系统部分执行仿真的验证方法。在 Zynq-7000 SoC 中，以处理器为目标的应用代码在器件中的实际 Arm Cortex-A9 处理器上执行。使用 HLS 编译的代码则在 RTL 仿真中执行。

图 9-1 显示了 Zynq-7000 器件的 HIL 验证概况。此图中的系统是一种实验性设置，其中包含 ZC702 评估板 （当前为商用板）和 Vivado 仿真器。此图还介绍了处理系统 (PS) 和可编程逻辑 (PL) 单元的概念。 PS 表示 Arm Cortex-A9 双核处理器，也称为处理子系统。 PL 表示 Zynq-7000 器件内的 FPGA 逻辑，即器件中 HLS 生成的模块映射到的部分。

提示：HIL 验证要求单板可访问处理器，此技术适用于任何 Zynq-7000 SoC 开发板。

图 9-1：Zynq-7000 SoC 的 HIL 验证概况

HIL 验证相比于其它验证的主要优势在于：

- 处理器模型与实际处理器之间不存在仿真不一致现象
- 处理器上运行的代码按 FPGA 器件的速度来执行

- 每个生成的模块在 RTL 仿真过程中的操作过程清晰可见

使用 HIL 验证时，牢记此方法的性能特性至关重要。虽然处理器在实际硬件中运行，但 FPGA 结构完全在设计人员的工作站上进行仿真。如第 8 章"多个程序的集成"中所述，RTL 仿真是一个相对缓慢的过程。因此，建议仅将 HIL 验证用于验证处理器与 FPGA 结构之间的主要交互，而非应用中的每个用例。使用 HIL 验证进行检查的关键应用行为包括：

- 将 Vivado HLS 驱动集成到处理器代码上
- 编写从 PS 到 PL 的配置参数
- 中断从 PL 到 PS 的连接

在软件算法的 RTL 实现中， Vivado HLS 编译器会生成所需的软件驱动，以供处理器与生成的硬件模块进行通信。来自 Vivado HLS 的驱动将负责处理加速器启动和停止、配置以及数据传输。此驱动可供 Linux 和独立软件应用使用。

注释：独立软件应用是所含处理器仅执行单个程序而无需操作系统支持的系统。

虚拟平台验证

虚拟平台技术是公认的软硬件重叠开发方法，可用于 Zynq-7000 SoC。虚拟平台是针对应用以及运行应用的硬件平台的软件仿真。用于设计的 PL 部分的模型可采用 C、 C++、 SystemC 或 RTL。此仿真平台可用作为建议的其它验证阶段的代理，以满足硬件实现的不同程度的逼真性要求。

虚拟平台的最快用例是从提供给 Vivado HLS 编译器的 C/C++ 源代码对以 PL 为目标的应用模块进行仿真。此设置可生成功能正确性仿真，以支持设计人员测试算法的计算方式是否正确。由于模块通过 Vivado HLS 进行最优化和编译，因此生成的 RTL 可替代模块的软件版本以支持连接测试和时序驱动仿真。

重要提示：请谨记，添加 RTL 模块会影响虚拟平台上的运行时间，导致减慢执行速度。

器件验证

器件验证的目的是检查处理器与 FPGA 结构进行交互的所有应用用例。与独立执行一样，此进程需在 Zynq-7000 SoC 运行完整应用并保持一段时间。此测试的目的是检查所有应用中有关设计的 PS 与 PL 部分之间的交互情况的极限用例。

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

如需了解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”视图。
- 在赛灵思网站上，请参阅设计中心页面。

注释：如需了解有关 Documentation Navigator 的更多信息，请参阅赛灵思网站上的 Documentation Navigator 页面。

参考资料

1. 《Vivado Design Suite 用户指南：高层次综合》 (UG902)
2. 《AXI 参考指南》 (UG761)
3. 《Vivado Design Suite 用户指南：采用 IP 进行设计》 (UG896)
4. 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》 (UG994)

5. Vivado Design Suite 技术文档 (www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design_tools/vivado_design_suite.html)

请阅读：重要法律提示