NI LabVIEW高性能FPGA开发者指南

优化LabVIEW RIO 应用程序的推荐方法

Revision No. 1.1 – February 2014

NATIONAL INSTRUMENTS

目录

引言

现场可编程门阵列(FPGA)技术为专用的定制硬件提供高性能和可靠性。作为LabVIEW FPGA用户，您可以在与台式和实时系统相同的编程设计环境中利用FPGA技术。

高性能LabVIEW FPGA应用程序将NI可重配置I/O(RIO)设备的性能扩展到定时、FPGA资源、以及其他方面。通过总结常用的LabVIEW FPGA优化概念和技巧，此指南旨在帮助您创建高性能应用程序。

目标读者

如果您已经熟悉LaVIEW或者LabVIEW FPGA模块，可使用此指南来学习业内不知晓的高级LabVIEW FPGA理论，帮助您解决需要高吞吐量、精确定时控制或者更高FPGA资源效率的严苛应用需求。

“高性能”这个术语是一个相对概念，但是通常它指当您需要从设计中获得更多性能时那些使LabVIEW FPGA编程更具挑战性的底层细节。关于高性能应用程序相关概述，请参见基于FPGA的高性能设计一章。

NI RIO硬件平台可负责许多底层实现细节，所以您可以专心应对具体应用挑战。如果您有数字设计经验，并且熟悉VHDL、Verilog或者电子设计自动化(EDA)工具，此指南可帮助您理解通过与NI RIO硬件平台的高度集成，LabVIEW FPGA是如何在实现相似性能的同时提供更高生产力。

前提条件和参考资料

本指南假设您熟悉LabVIEW编程和基本的LabVIEW FPGA任务，比如创建、编译、仿真以及部署FPGA VI。请参考产品文档了解关于这些内容的更多信息。

您应该也对什么是FPGA以及FPGA的优势有一些基本了解。关于基于FPGA设计的介绍，请参见ni.com上的FPGA基础原理白皮书[1]。

本指南搜集并总结了创建和优化LabVIEW FPGA应用程序所需要的概念，无论您使用的是哪个版本的LabVIEW。参见您所用版本的LabVIEW包含的产品文档，了解API和环境细节，这是因为各个版本的LabVIEW可能会有所不同。本指南也参考了随附范例、在线教程以及其他参考资料。

本指南与受工程师欢迎的基于NI LabVIEW的CompactRIO开发者指南[2]相似， 为设计NI CompactRIO控制和监测应用提供了最佳方案。参考这本指南的使用LabVIEW FPGA自定义硬件部分了解LabVIEW FPGA编程概述。本指南还基于该部分介绍的概念指导您实现更高的性能。

更多资源

[1]	FPGA基础原理 http://www.ni.com/white-paper/6983/
[2]	基于NI LabVIEW的CompactRIO开发者指南 http://www.ni.com/comp actriodevguide/

基于FPGA的高性能设计

FPGA的优点

FPGA提供了高度并行的可自定义平台，您可以使用该平台以硬件的速度执行高级和控制任务。

相对于CPU和GPU而言，FPGA具有较慢的时钟速率，但是FPGA通过可在单时钟周期内多次连续地执行并行操作的专用来弥补时钟速率的差距。您可以在NI RIO设备上将FPGA具有的大规模并行编程特性和紧密的I/O集成结合起来，实现更高吞吐量、更好的确定性以及更短的响应时间，从而满足高速数据流、数字信号处理(DSP)、控制以及数字协议应用的需求。

高性能LabVIEW FPGA

当您在LabVIEW FPGA软件中使用标准的LabVIEW编程技巧，您就能立刻体验到基于FPGA方法的大多数优势。高级应用可能对以下一个或多个参数具有更高的要求：吞吐量、定时、资源和数值精度。

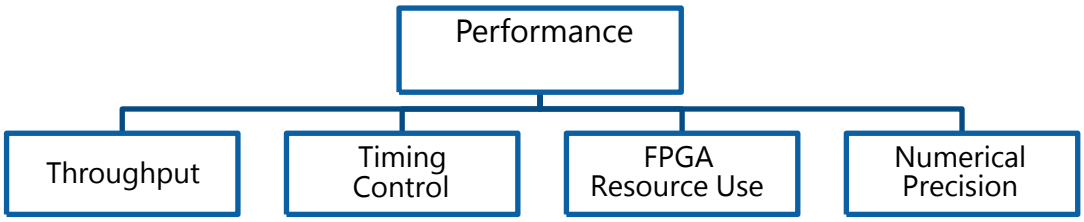

图1. 高性能RIO应用的多个维度相互关联。例如，吞吐量的增加可能会影响定时和资源占用。

这些维度通常是互相关联的。如果仅仅通过某一维度来优化设计，通常会影响到其他维度。这种影响有时是积极的，但更多是以牺牲其他维度为代价。例如，如果您只关心吞吐量，设计可能会无法满足定时的要求。因此，您需要理解这些维度以及它们之间的相互影响。本部分提供了多种考虑因素的基本概念，对相关技巧的扩展贯穿整本指南。

吞吐量

在DSP和数据处理应用领域，吞吐量是一个关键概念。通信、制造、医学、航空、国防、以及科学测量系统领域的技术进步产生了更多的数据，并要求用更短的时间进行处理。吞吐量是以单位时间完成的工作来衡量的。在大多数LabVIEW RIO应用程序中，工作是指处理或者传输采样数据，而吞吐量通常是以每秒采样数或者相当的形式诸如每秒的字节、像素、帧或者运算次数来计量。以快速傅立叶变换(FFT)作为运算函数的一个范例，这里吞吐量是以每秒所进行的FFT次数、帧或者采样次数来计量的。

本指南中吞吐量优化技巧一章对影响吞吐量的因素进行了深入的讨论。这些讨论包括时钟速率和算法并行性，以及创建LabVIEW FPGA应用程序时实现更高吞吐量的一些技巧。

定时控制

定时控制是指设置并测量系统中感兴趣事件之间的时间间隔。精确定时控制对于数字协议和高速控制应用非常重要，因为它会影响系统通信能力或者受控系统的稳定性。控制应用通常要求保证响应时间，这通常以受控系统采样和控制器的输出更新之间的时间间隔来衡量。该时间间隔也称为I/O延迟。在数字协议应用中，定时规范可能是指与数据或者传输信号相关的事件之间的目标时间、最短或最长时间。创建LabVIEW FPGA应用程序时，设计会最终转化为硬件电路，所以您可以创建具有快速定时响应和微小抖动的设计。

本指南的定时优化技巧一章对延迟以及使用LabVIEW FPGA实现更快和更精确的定时响应的技巧进行了深入探讨。

FPGA资源占用

一块FPGA芯片的硬件资源是有限的，与处理器或者单片机相比，通常FPGA更受制于存储容量。确保您的设计适合FPGA是开发过程中必须严格遵循的法则。FPGA也是由不同类型的资源组成的，包括逻辑、信号运算和存储器模块。耗尽任何一种资源都会终止整个设计的编译。

更重要的是，资源占用对吞吐量和定时限制等性能有非常大的影响。参考本指南的**资源优化技巧**一章的描述，了解组成FPGA包含的不同资源、确保设计适合FPGA的方法，以及提高性能的办法。

数值精度

数值精度是指使应用正常工作所需要的位数或者比特有效度。精度不足可能使迭代算法不断积累小的数值误差，随着时间的推移这些小的误差最终会导致完全不同的结果。代表系统变量的比特数包括整数、定点数的整数和小数部分、或者浮点数的动态范围，这些比特数可能会影响FPGA应用程序的性能和资源占用情况。

虽然本指南并不会详细介绍如何帮助您正确决定数值精度，因为经常讨论的话题通常围绕如何优化LabVIEW FPGA设计，但是这些讨论可以作为学习优化其他维度的参考。

理解NI RIO硬件平台

应用的需求可能包含前一部分讨论到的所有参数。既然这些参数相互联系，那么理解它们之间的相互影响是非常重要的，这样您就可以在编程时进行适当的取舍。本指南之后的部分对每个参数的优化进行了更详细的讨论。本部分研究了NI RIO平台的几个典型应用，并阐释了不同的硬件特征如何与具体应用实现最佳匹配。

基于PXI和PC平台的NI RIO

PXI平台

PXI (面向仪器的PCI扩展)是一种专为高性能应用而设计的坚固模块化仪器平台。它结合了PCI和PCI Express总线技术以及专用同步总线，为生产测试、军事和航空航天、机器状态监测、汽车以及工业测试等领域的应用提供高性能低成本的部署平台。PXI是一种开放的工业标准，由PXI系统联盟（PXISA）进行管理，该联盟旨在推广PXI标准，确保PXI的互通性，并维护PXI规范。

图2. PXI平台将模块化仪器、高性能处理、数据传输以及同步总线与台式或者实时应用程序开发相结合，满足高级FPGA应用程序的需求。

PXI Express利用PCI Express总线，提供点对点总线拓扑结构，使每台仪器都能以高达4 GB/s的吞吐量直接访问总线。集成定时和同步线路用于内部路由同步时钟和触发。PXI机箱采用专用的10 MHz系统参考时钟、PXI触发总线、星形触发总线，以及槽对槽局域总线。而PXI Express机箱则增加了一个100 MHz的差分系统时钟、差分信号发送和差分星形触发器，以处理高级定时和同步需求。

NI提供多种可以用于PXI平台的RIO产品，利用带宽和同步特性来实现高性能应用。

[1] 本指南中的吞吐量: GB = 10^9 比特, MB = 10^6比特, MiB = 2^{20}比特, GiB = 2^{30}比特

NI FlexRIO

NI FlexRIO是对基于PXI平台的通用研究、设计、测试、原型设计和部署设备的总称。NI FlexRIO设备包括可使用LabVIEW FPGA模块进行编程的大型FPGA，以及一些提供高性能模拟和数字I/O的适配器模块。适配器模块是可以互换的，在LabVIEW FPGA编程环境中定义I/O.

图3. NI FlexRIO设备包含了大型FPGA，可以直接连接适配器模块，提供多种I/O选项。

NI FlexRIO FPGA模块具有Xilinx Virtex-5和Kintex-7 FPGA、板载动态RAM (DRAM)以及与NI FlexRIO适配器模块的接口（为FPGA提供了I/O）。适配器模块接口除了提供必要的电源、时钟以及辅助电路来定义接口之外，还包括132条直连到FPGA引脚的通用数字I/O线。您可以配置这132条线路，以最高400 Mbit/s的速率实现单端运行或以最高1 Gbit/s的速率实现差分运行，从而获得高达66 Gbit/s (8.25 GB/s)的最大I/O带宽。

您可以在NI FlexRIO FPGA模块之间以1.5 GB/s的速率直接传输数据。最多可支持16个这样的数据传输，从而简化复杂的多FPGA通信方案，而不必耗费主机的CPU资源。参见本指南的**数据传输机制**一章，了解更多关于在NI FlexRIO设备之间直接传输数据的内容。

NI R系列多功能RIO

尽管标准的NI多功能数据采集(DAQ)板卡能够以不同的采样率测量并生成信号，但R系列多功能RIO设备却通过集成FPGA可帮助用户完成一些之前使用数据采集板卡无法完成的任务。

NI R系列多功能RIO设备具有很高性价比，它将模拟输入、模拟输出和数字I/O线以及FPGA技术集成到单个设备上。NI R系列多功能RIO设备支持PCI、PCI Express、PXI和USB总线，同时提供了带封装和只有板卡两种不同的选项。

图4. NI R系列多功能RIO设备通过可使用LabVIEW编程的FPGA扩展了通用多功能DAQ。

NI R系列多功能RIO设备的每个通道均配有专用的模数转换器(ADC)，可实现独立定时、触发和高达1 MS/s采样速率。它提供了多速率采样和独立通道触发等专用功能，这是传统DAQ硬件无法实现的。

软件设计的模块化仪器

NI推出了世界上首台软件设计的仪器——NI PXIe-5644R矢量信号收发仪(VST)。借助NI-RFSG和 NI-RFSA驱动，这台仪器可以用作射频矢量信号发生器(VSG)和矢量信号采集器。

图5. 矢量信号收发仪(VST)结合了仪器级的矢量信号发生器和采集器到一台包含大型FPGA的3槽PXI组成结构中，FPGA可以用使用LabVIEW进行自定义。

VST不仅是体积小巧的高性能射频硬件，还可以借助LabVIEW FPGA模块对FPGA进行自定义，因而具有革命性的意义。您唯一的限制因素就是应用的需求，而不会再受限于供应商定义的仪器功能。这种方法极大地提高了灵活性，并且通过基于FPGA的处理和控制功能更好地满足了应用的需求。

用于紧凑的嵌入式应用的NI RIO

NI CompactRIO

CompactRIO是一个坚固的可重配置嵌入式系统，包含三个组件：运行实时操作系统的处理器(RTOS)、可重配置的FPGA以及可互换的工业I/O模块。

图6. CompactRIO通过模块化I/O为分布式和高性能嵌入式应用提供了坚固紧凑的部署平台。

CompactRIO系统包括一个嵌入式控制器和可重配置的机箱。嵌入式控制器具有可靠的确定性行为，为LabVIEW实时应用程序提供强大独立的嵌入式执行。控制器在浮点数学运算和分析方面也具有出色的性能。

嵌入式机箱是CompactRIO系统的核心，因为它包含了可重配置的I/O FPGA。FPGA可出色地执行需要高速逻辑和精确定时的任务。它直接连接至可提供各种高性能I/O功能的I/O模块。

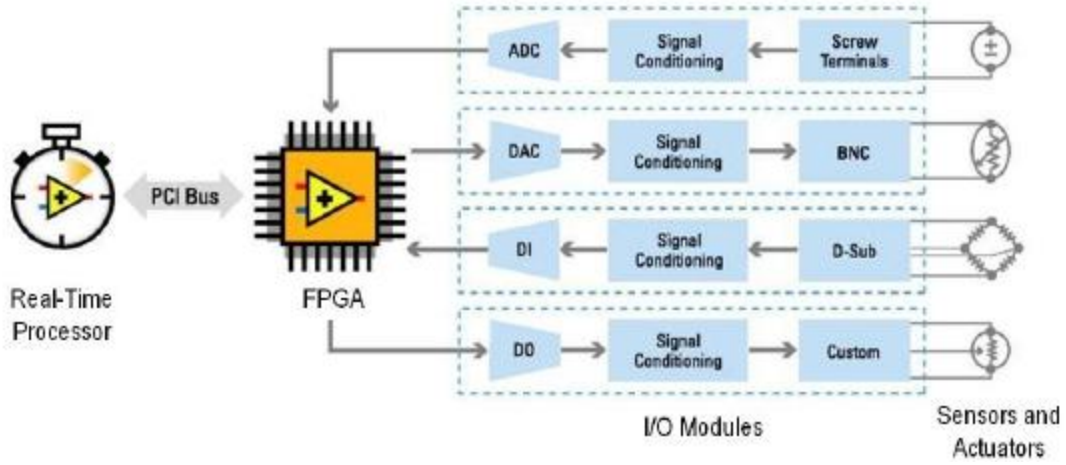

图7. 通过可以运行实时操作系统的专用处理器、具有可重配置硬件性能的FPGA以及多种模块化I/O选项，CompactRIO具有典型的LabVIEW RIO构架

NI Single-Board RIO

NI Single-Board RIO产品专为高容量和OEM嵌入式控制和需要高性能和可靠性的DAQ应用而设计。这些商用现成(COTS)硬件设备具有开放的嵌入式构架、小巧的体积和高灵活性，可以帮助用户将自行开发的嵌入式系统快速推广上市。

图8. NI Single-Board RIO使嵌入式RIO平台具有更小的外形尺寸，适用于部署应用。

.

选择FPGA平台

应用需求决定了要使用的平台和NI RIO设备系列。但是，每种设备系列有提供各种不同的选择，包括不同大小和速度的FPGA。基于LabVIEW FPGA应用程序，无论您是否拥有硬件，均可以针对任意NI RIO设备编译设计。您还可以在软件中对设计进行仿真，在编译前验证其行为。使用真实信号以硬件速度验证系统时仍需要使用硬件。用户通常使用NI RIO设备系列来采集大型FPGA设备的数据进行原型开发，之后用更小和更合适的设备优化应用的FPGA资源占用况或者性能。

只在FPGA上执行基本定时、触发和同步的应用通常可以使用更小的FPGA。需要额外信号处理的应用，比如控制、数字滤波或者复杂模拟触发，可能需要较大的FPGA来实现这些操作。

更多资源

[1]	什么是PXI? http://www.ni.com/pxi/whatis/
[2]	什么是NI FlexRIO? http://www.ni.com/flexrio/whatis/
[3]	NI R 系列多功能RIO http://www.ni.com/rseries/
[4]	什么是NI CompactRIO？ http://www.ni.com/compactrio/whatis/
[5]	什么是NI Single-Board RIO？ http://www.ni.com/singleboard/whatis/
[6]	基于USB 的NI R系列 http://sine.ni.com/nips/cds/view/p/lang/en/nid/212013
[7]	选择正确的FPGA硬件 http://www.ni.com/fpga-hardware/choose-hardware/

此页特意留白。

基于单周期定时循环的高性能编程

大部分与使用LabVIEW进行高性能FPGA编程相关的概念均涉及有效地利用单周期定时循环(SCTL)。SCTL是一个主要的LabVIEW FPGA结构，可降低资源占用，提供更高吞吐量和更精确的定时控制。SCTL提供不同的编程原理，可更紧密地匹配FPGA电路的行为，并可更好地控制FPGA上的LabVIEW代码实现。

SCTL和标准LabVIEW FPGA代码的比较

如果要理解SCTL，需要先了解标准的LabVIEW FPGA代码放入While循环或者For循环时是如何进行编译的。

使用LabVIEW FPGA模块编程时，程序框图的内容会转换成硬件，因此大体上看，程序框图上每个节点在FPGA上都会有对应的电路元件。将编码放到SCTL外面时，LabVIEW在数据流经硬件时控制这些硬件组件的执行。结果，需要更多的电路来确保这些组件仅在其输入端具有有效数据时执行。这种执行模式称为结构化数据流。

结构化数据流遵循传统的程序执行模型，即函数必须在具备所有输入参数后才能执行，当函数返回后调用程序才会停止执行。在CPU上执行代码时，CPU必须有固定的通用电路，会连续消耗代码和数据。但是，与CPU不同的是，FPGA上的代码会变成高度专用的并行电路。数据会像电信号一样流过FPGA。

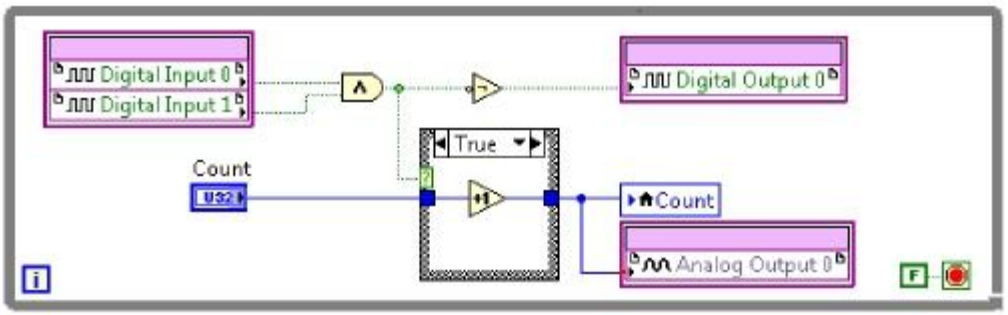

图9. 标准的LabVIEW FPGA代码被合成到电路中，当存在数据相关性时，该电路连续执行。

在While循环中编译代码时，LabVIEW逐个函数地将硬件寄存器或者小存储元件插入到时钟数据中，从而执行LabVIEW的结构化数据流。

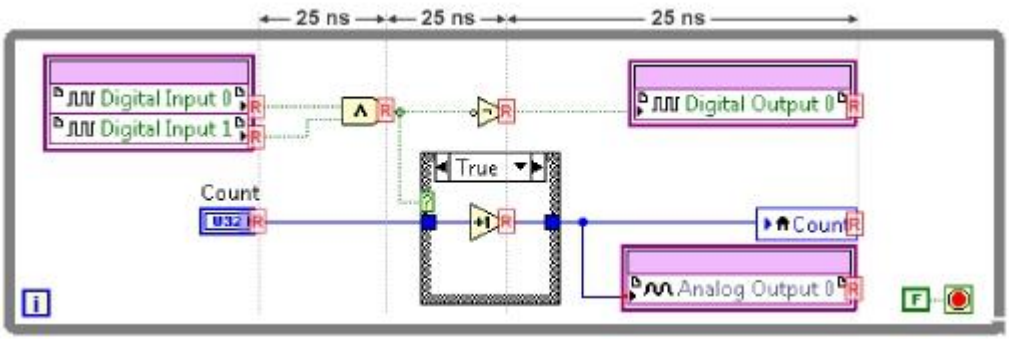

图10. 标准LabVIEW FPGA代码在每个函数上使用寄存器锁存每个顶层时钟周期的数据，该时钟通常为40 MHz（周期为25 ns）。这里硬件寄存器由用R标注的方框表示。

这些寄存器元件添加到控制函数执行中，并在数据从一个节点流向下一个节点时锁存每个时钟周期的数据。每个节点的执行需要一个或多个周期。在LabVIEW FPGA应用程序中，每次编译或者流经不同版本的LabVIEW FPGA模块时，放到SCTL外部的代码都会表现出不同的定时行为。

保存结构化数据意味着，比如针对有数据相关性的节点链，只有一部分节点可以按照给定时间主动地执行，电路的其他部分则会等待数据。

顺序执行可能看上去并没有充分地利用专用FPGA电路，但是它满足了许多应用的需求。结构化数据流模型为LabVIEW开发者提供了一种编写LabVIEW代码的方法，类似于他们在桌面上创建快捷方式一样，同时使得开发者在FPGA上执行代码而无须担心实现细节。因而，LabVIEW FPGA开发者可享受到出色的性能、确定性和I/O集成。

理解SCTL

对于创建高性能LabVIEW FPGA应用程序来说，理解SCTL及其里面节点的功能是关键。SCTL对LabVIEW FPGA应用程序来说是一种独特的结构。尽管SCTL结构和放置在其内的节点看上去几乎与LabVIEW定时循环及其内部节点一样，但是放置到SCTL内部的代码却以一种完全不同的方式执行，因为它要保证在一个特定的FPGA时钟周期内执行完。SCTL代表了以下五个主要概念：

1.结构

在LabVIEW中，结构有特别含义。结构可以在数据流经边界时提供数据范围和定义数据的转折点。相对于图上其他结构，SCTL是一种遵循结构化数据流模型的结构。特别地，SCTL只有在所有的输入或者通过隧道或者移位寄存器的连线获得数据后才能开始执行。同样地，只有其内部代码执行结束后，SCTL才能通过隧道生成输出。

2.循环

SCTL不仅是一种定义代码如何执行的结构，还是一个程序循环。它重复地执行内容，并遵循While循环规则，因此它必须至少执行一次，可以在运行时设置停止条件。

3.时钟

每个SCTL都必须有一个时钟。这个时钟定义了将用于驱动SCTL内容合成产生的电路的频率。因为SCTL内部的所有电路共享同一个时钟，它也被称为时钟域。

定义SCTL时钟的能力是设计中使用多个时钟的首要考虑因素。不同的SCTL可能使用不同的时钟定义多个时钟域。时钟速率和时钟源必须在编译时决定好，这样执行设计时才不会被更改。

4.最长迭代延迟

SCTL不仅规定了用于驱动其内部代码的时钟，而且要求其内的所有代码在一个时钟周期内执行。

5.不同的执行方式

SCTL执行方式与LabVIEW中的标准执行不同，因为SCTL内所有的代码必须在一个时钟周期内执行完毕。为了满足迭代延迟的要求，LabVIEW FPGA编译去掉了用于执行结构化数据流的流控制电路。

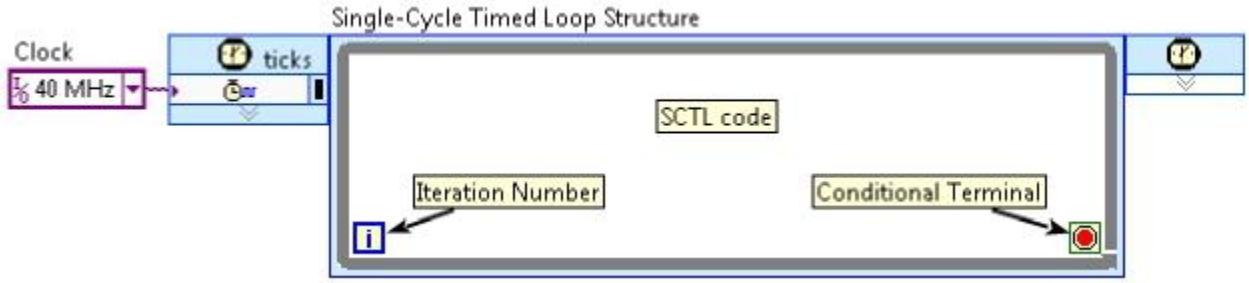

图11. SCTL循环结构规定了代码所使用的时钟并假定了一个时钟周期的最长迭代延迟。

去掉流控制电路不仅降低了FPGA资源占用，更重要的是允许程序框图能够像并行电路那样执行， SCTL中的所有数据由流动不受限制的电信号表示。信号通过逻辑进行转化，并且锁存在程序框图的某些点上，比诸如I/O、存储元件、输入控件和显示控件。

由于信号锁存在程序框图内的特定点，它们必须在下一个时钟边沿到达时处于定值，下一个时钟边沿到达标志着新迭代的开始。这种锁存行为结合循环套循环的模式，构成了由FPGA编译工具链合成的同步电路。

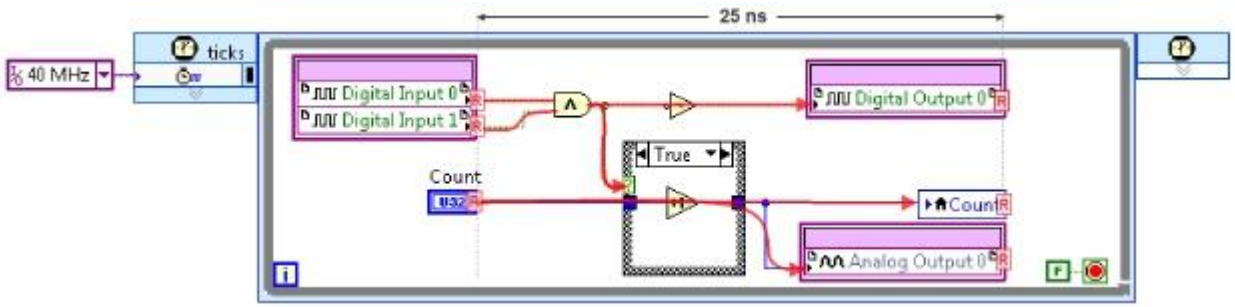

图12. SCTL代表一个同步电路，其中信号流过并只锁存到特定点上，诸如I/O，输入控件、显示控件和其他结构。SCTL规定了用于驱动内容的时钟，以及信号必须在少于一个时钟周期内在寄存器之间传输，此例中的时钟周期是25 ns。

同步电路中的术语“同步”与标准编程语言中用来指函数调用机制的同步的意义不同。同步电路是由一个时钟驱动的简单电路，而同步函数调用在函数返回结果前阻止调用程序执行。

SCTL的优势

SCTL规定的单周期迭代延时与编译时生成的VHDL代码一起传递到Xilinx编译工具链。Xilinx编译器合成SCTL代码，并把单周期要求作为电路限制条件。如果编译成功，生成的设计就一定会执行，并且在一个时钟周期内处理所有信号。这一行为和CPU上的LabVIEW定时循环不同，LabVIEW定时循环的执行周期并不确定，只能在运行时得到验证。

迭代延时约束是SCTL编程的重要部分，因为它和通常需要超过一个周期的时间来执行一部分代码的标准LabVIEW FPGA代码不同。鉴于SCTL的定时保障，您可以使用SCTL作为一种机制来规定事件和整体执行率之间的定时，如下所示：

- SCTL同一迭代内的两个事件之间的最长时间为一个循环周期。如果需要规定事件之间的最大延迟时，这一功能非常有用。
- 您可以决定、限制或者保证连续迭代中事件之间的最小周期数。这对于数字通信协议等应用中的测量和控制事件非常有用。
- 如果您知道循环速率是由编译器来进行保证，就可以规定吞吐量。循环与每次迭代运算处理的采样数据量相结合，可用于衡量吞吐量。

SCTL的约束条件

即使SCTL代码可能看上去和其他LabVIEW代码相似，但其编程和执行方式要求您调整您的思考方法以及您使用不同架构去实现更优性能的方式。

SCTL内部支持的函数和结构

SCTL迭代延迟要求会限制放置于其内的可支持LabVIEW函数和结构。

在SCTL内部需要多个循环来执行的函数也要求额外的握手信号，以在有可用的新数据或有新数据可消耗时通知SCTL内其他的逻辑。集成高吞吐量IP这一章将会讨论这些函数和握手机制。需要多个周期来执行但不包含握手信号的函数不受SCTL支持。这些函数包括：

- 某些算法函数，比如商与余数函数。
- 循环计时器和等待函数
- 一些模拟输入和模拟输出I/O节点
- 多个非重入子VI例程（参见定时优化技巧一章，了解更多关于使用非重入VI效果的信息。）
- 处理浮点数据类型的大多数函数

循环结构，比如For循环和While循环，也不允许放到SCTL内。如果您尝试在SCTL内创建需要某种迭代算法的子VI，这将是一种挑战。这种情况下，您必须在SCTL内明确地创建要用的子VI，并使用反馈结点来存储并跟踪不同迭代的值。

图13所示的代码表示的是对固定大小的数组进行排序。它大致涵盖了如何调整代码以用于SCTL。

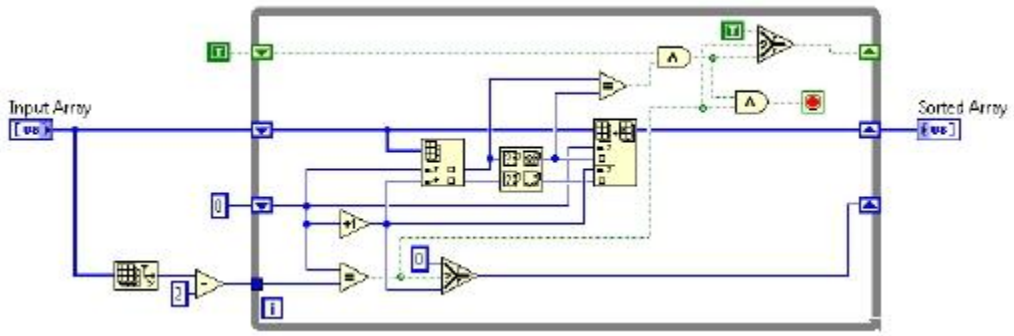

图13. 此图是一个迭代算法范例，采用标准LabVIEW代码编写，用于对数组进行排序。

图14的中SCTL删除了内部的寄存器，但添加了一个电路来跟踪穿过整个数组的索引。

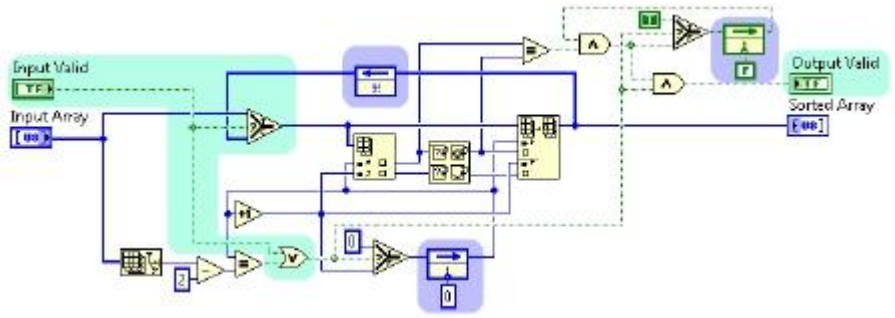

图14. 数组排序算法的SCTL称为多次同数组输入，所以它采用反馈节点来替代移位寄存器（浅蓝色部分），增加代码来跟踪穿过整个数组的索引，增加握手信号（浅绿色部分）来获悉新数据的达到，并告知下行模块何时可以消耗已排序的数组输出。

请留意图14中算法完成之前不应消耗子VI的输出，但是子VI却必须多次调用才能执行完毕。您必须添加一个信号到输出，用于指示何时数组已经排序完毕。数组排序仅仅是需要多周期执行的函数和握手信号要求的一个范例，在集成高吞吐量IP一章中将会详细阐述。

要求多个循环为迭代计算服务的函数，包括嵌套循环，通常被称为算法VI。如果要针对LabVIEW FPGA移植并优化这些VI是比较困难的。NI针对这一需求专门开发了一个LabVIEW FPGA 附加工具——LabVIEW FPGA IP生成器。您可以将它用于大型数组、嵌套循环以及浮点类型，并生成针对SCTL优化的IP。

在SCTL结构之间传输数据

在不同SCTL时钟之间传输数据可能较为困难。正如在数据传输机制一章中详细讨论的那样，以不同速率运行或者具有未知或者可变相位关系的循环会在不同时间输出并锁存数据。因此，如果没有安全的传输机制，比如块存储器的先入先出(FIFO)结构，或者在不安全的传输机制下未能添加自己的同步，则会发生数据丢失、重复或者损坏。

成功的编译

应用可能需要非常高的吞吐量或者很低的延迟。实现这些目标的常规做法是提高SCTL时钟速率。当时钟速率和资源占用增加时，工作链会更卖力地工作来维持严格的迭代延迟约束条件。这通常会导致编译时间变长，编译成功率变低。

信号在合成电路中达到稳定状态所需的时间也称为传输延迟。传输延迟由两部分组成：路由延迟和逻辑延迟。

路由延迟是指信号在电路组件之间传送所花费的时间。这些信号几乎是以光速传送，但是时钟速率通常也非常高，因此需要将这些延迟考虑进行。缆线和电路也不是理想的导体，因此时钟速率上升时会产生小负载，使数字信号丢失特性脉冲波形，进而使得时钟速率上升时保持信号沿着长线缆完整传输变得更困难。

逻辑延迟是指将信号沿逻辑组件传输所需的时间，比如信号从比较函数的输入传到输出所需的时间。

随着程序框图不断扩大，逻辑也会增加，相关组件的放置位置可能会相距更远，导致更长的传输延迟。

SCTL内具有最长传输延迟的数据路径称为关键路径。如果关键路径上的传输延迟超过SCTL时钟周期，编译就会失败。LabVIEW FPGA提供一种将编译结果映射到程序框图元件的方法，这样您就可以尝试优化失败路径。

优化技巧根据应用程序目标的不同而变化。吞吐量、延迟和资源占用是高性能LabVIEW FPGA应用程序最常见的设计影响因素。您在选择时必须有所取舍。例如，您通常可以以资源和延迟为代价来提高吞吐量。您也可以在算法允许时通过并行执行代码来降低延迟，这会导致FPGA资源占用增加。进行这些取舍时需要采用迭代方法来优化流程。接下来几章会讨论优化吞吐量、延迟和FPGA资源占用的各种方法。

更多资源

[1]	基于FPGA应用的NI LabVIEW FPGA IP Builder的VI优化和保存 http://www.ni.com/white-paper/14036/en/

此页特意留白。

吞吐量优化技术

信号、图像和通用数据处理应用对吞吐量普遍都有要求。FPGA通常用于数据流联机处理，因而必须要能够维持所需的吞吐量，有时候这种维持是无限期的。吞吐量以单位时间内处理的采样数或数据量来衡量。就本章而言，我们假定吞吐量以每秒处理或传输的采样数来衡量。

吞吐量取决于三种因素：

1. 设计所采用的时钟频率（周期/时间）
2. IP在每次调用时所能接受的采样数（采样数/调用）
3. 再次调用算法前间隔的周期数（周期/调用），亦指启动间隔（II）或启动周期

三种因素结合起来根据下述定义可以得到采样数/时间。

$$吞吐量 = \frac{时钟频率 \times 每次调用接受的采样数}{启动间隔}$$

根据等式，您可以通过以下方式提高吞吐量：

- 提高时钟频率
- 提高每次调用所处理的采样数
- 降低启动间隔

提高时钟频率

通过顶层程序框图的时钟或者SCTL时钟增加时钟频率是提高组件吞吐量最直接的方式。若能成功，在其他因素保持不变的情况下，提高时钟频率可快速引起吞吐量的线性增加。因此，如果可能的话，您应该尽量考虑提高时钟频率。

您还应意识到增加时钟频率的其他影响。在之前的章节中讨论到，较高的时钟频率增加了编译过程的难度和时间，最后将有可能无法在目标平台上生成一个满足定时约束的线路。为提高编译速度，本章的缩短关键路径部分提供了设计时一些缩短关键路径的技巧。

在某些情况下您可能无法提高时钟频率。部分设计会要求IP在执行时与系统其他部分的时钟关系相一致。例如，某些I/O模块的执行必须与I/O子系统具有相同的时钟域。采用组件级IP(CLIP)集成机制的外部DRAM端口可能也需要通过DRAM接口的时钟域进行访问。因此，您必须将设计分割成多个SCTL，对每个SCTL分别采用适当的时钟域。

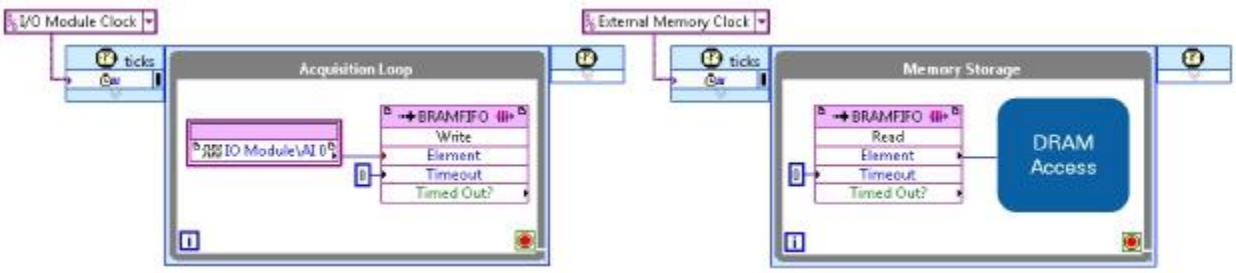

图15. 与I/O交互或者通过CLIP访问DRAM的代码必须具备适当的时钟域。如果应用程序需同时执行二者，您必须其分成多个SCTL。

或许您的设计只有某些部分需要较高的时钟频率。如果代码能在较低的时钟频率达到预期的吞吐量，可以考虑将您的设计拆分为多个循环，因为这样可以简化并加快编译过程。

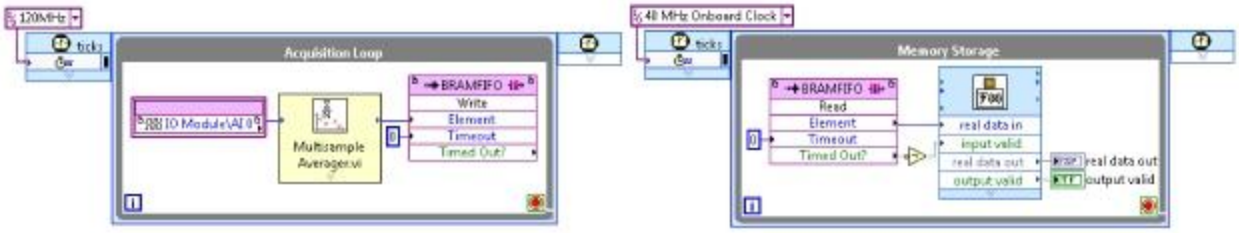

图16. 设计中的不同部分可能有不同的吞吐量要求。将设计拆分成多SCTL使得编译器在关键之处更容易达到预期的时钟频率。

在以不同时钟域运行的循环之间传递数据时，您必须要多加注意。关于此方面的技术支持，请参考数据传输机制部分。

增加每个调用所处理的采样数

VI的每次调用或者SCTL的每次迭代所处理的数据采样量在设计中通常都是固定的。但是您可以，例如，实现一种并行算法，每次调用都能从同一通道处理多个采样数据，从而在保持时钟频率和启动间隔不变的情况下，提高吞吐量。

并行化的可能性取决于算法的特性。在某些情况下，操作可在独立的数据集上显示并行，例如当您在多个I/O通道执行相同的操作时。在该情况下，由于FPGA的并行特性以及SCTL内对循环结构的使用限制，您通常会复制逻辑，以独立处理每个通道。这样就需要占用额外的资源。我们以一个LabVIEW可独立处理不同通道的应用程序为例来确定每个通道的最大值。

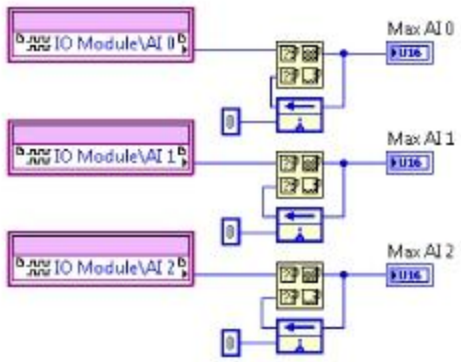

图17. 在FPGA上，通过复制每个通道的代码，您可以实现LabVIEW在多个通道上的并行运行。

一个更有趣的情况是同一数据集的采样也可以独立进行处理，就像从同一I/O通道（数据流）同时处理多个采样。这种情况只有在不需要知道之前的通道采样数据时才可能行得通。这方面的一个例子就是根据特定因子来换算通道采样数据。

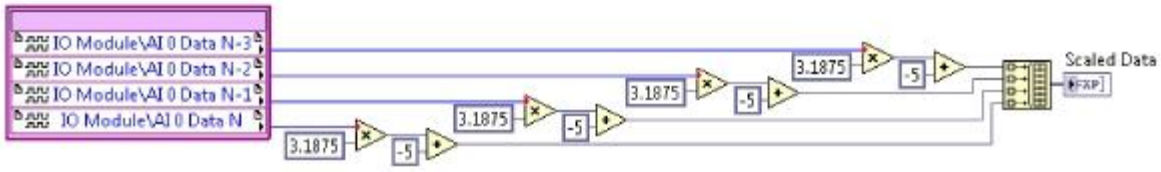

图18. 某些硬件模块在每个FPGA时钟周期中会生成多个采样，这就需要用并行方法去处理。

由于某些NI FlexRIO适配器模块会在每个FPGA周期中生成多个通道采样，因此处理一个单通道的每个采样就要用到这种并行化。这些函数称为多重采样/多周期IP。

FPGA本身就是并行设备，这使得多采样或多数据集的并行处理成为提高吞吐量的一个不错选择。FPGA也会受到资源方面的限制，并行化通常会导致资源利用的线性增加。通过复制的方式显式地并行化代码会使得代码难以读取和维护。因此，在选择并行化或每个调用处理更多地采样或循环迭代时,请务必仔细考虑清楚。文档稍后会讨论其他的并行化技术。

缩短关键路径

在优化吞吐量时，缩短SCTL关键路径的终极目标就是为了提高其时钟频率。如果SCTL关键路径的传输延迟超过SCTL时钟周期，就会导致编译失败。LabVIEW在编译失败出现时会标示出关键路径。您可以根据该信息来缩短路径的整体传输延迟，然后以相同或更高的时钟频率重新编译您的设计。

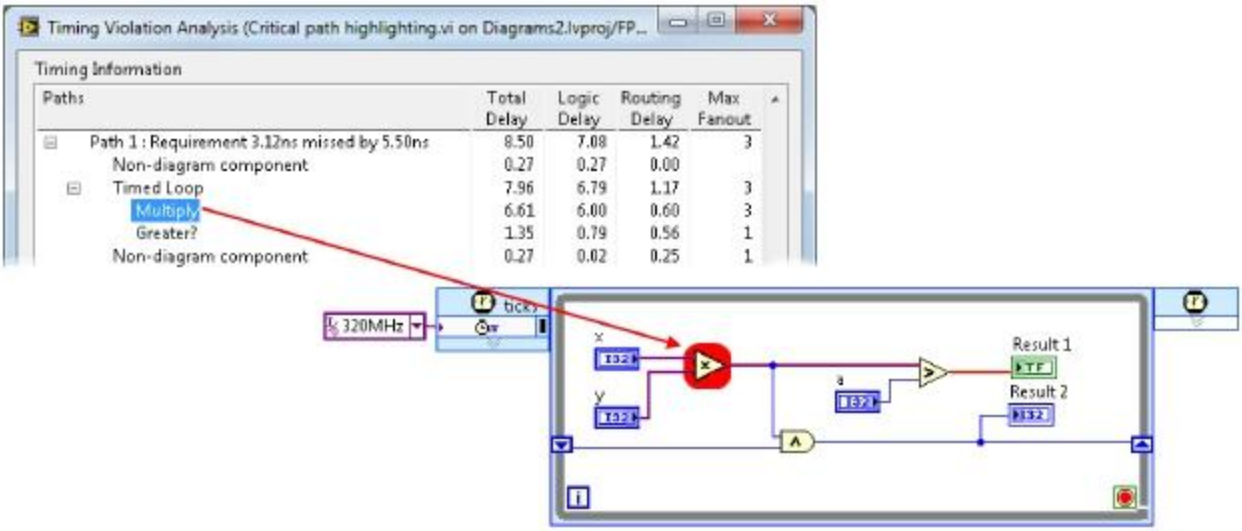

图19. LabVIEW FPGA对FPGA编译结果进行解析，并直接在程序框图上识别出不能满足定时约束的路径，以便您采取措施进行优化。

并行操作

在缩短关键路径长度时，首先要去掉不必要的相关性。找到任何由于数据相关性而需要顺序执行的路径，思考一下是否真的需要顺序运行。有时候，通过分支连线以及同时对多个拷贝的数据执行不同的操作，您可以从关键路径中移除一些操作。然后结合之后生成的结果，如图20所示。

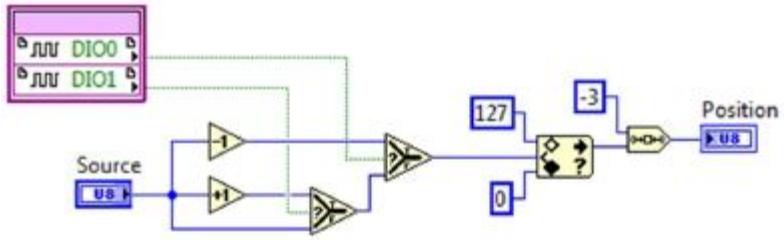

图20. 资源允许的话，您应该充分利用FPGA作为代码实现的专用硬件，在程序框图上显式地并行化独立的操作。

该方法取决于操作的性质。您或许能采用其他编码模式提高并行性，但如定时优化技术章节所提到的，这反过来会降低代码的整体延迟。

数据类型优化

用于表示数据的比特数对信号沿关键路径的传输延迟具有非常重要的影响。因此，采用能满足准确度和精密度要求的最小数据类型也能帮助您达到预期的SCTL时钟频率。

FPGA处理较大的数据类型时需要更多的逻辑，且逻辑量会随着比特数线性增加。同时，为了将这些比特数从一个逻辑传输到下一个逻辑，往往需要添加更多的电路径，而这些路径的长度可能是不同的。因此，为了让信号能够在所有路径达到稳定状态，必须减少定时约束，但这会阻碍线路以更高的频率运行。

数据类型优化在降低资源利用、提高可实现时钟频率从而增加设计的吞吐量方面是一种非常有效的方式。关于数据类型优化更详细的讨论见资源优化技术章节。

流水化设计

流水化是可用于提高代码吞吐量最常见的技术。该技术涉及到在SCTL关键路径中插入寄存器，将其分解为较短的并行代码段。较短的代码段所需执行时间减少，从而能让您增加SCTL时钟频率。

下例中，会注意到有条单数据路径，同时它也是关键路径。假设您想以200 MHz的频率执行SCTL，那么程序框图必须要能够将数值从输入传到输出，并且只允许它们停留少于5纳秒的时间。

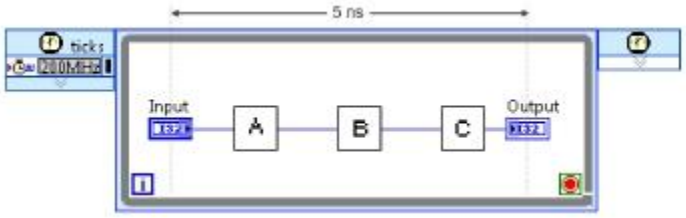

图21. 当SCTL配置为采用200 MHz时钟时，该代码序列执行时间必须少于5纳秒

假设子VI A、B和C在FPGA上执行分别需要3、4和5纳秒，那么该序列所需的总时间是12纳秒，这就将可达率限制到了大约83 MHz。200 MHz的编译在该例中就不能实现。

您可以通过以反馈节点的形式添加寄存器来优化上述VI，通常是在前向表示中，以允许每个子VI在一开始执行循环时就能开始处理数据值。前向反馈节点通过锁存并保留迭代中的数值，将执行分解为可同时并行的多个部分，从而将关键路径拆分为三个部分。缩短关键路径能够提高时钟频率，比如在此例中，缩短关键路径能够让时钟以所需的200MHz运行，将吞吐量提高了2.4倍。

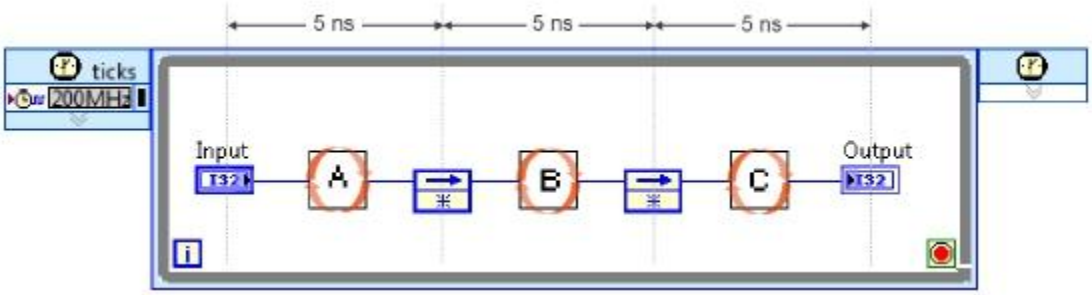

图22. 流水化允许VI并行执行。现在路径的每个部分都具有5纳秒的的延迟约束，从而使得循环能够以200MHz的频率运行。

流水化能够提高处理延时。在上例中，原程序框图的采样数据只需一个周期就能从输入传输到输出，而现在则需要三个周期。由于我们的一部分目标是拆分关键路径以提高时钟频率，通过较高的时钟频率就可以重获部分丢失的延时。然而，由于额外的逻辑延迟和关键路径的不完全划分，较高的时钟频率并不能完全补偿延时。实际上，流水化是通过增加一些延时来提高时钟频率，进而实现更高的吞吐量。

在上例中，最大期望时钟频率83 MHz所对应的初始延时应为：

$$初始延时 = \frac{1\ 个周期}{83\ MHz} = 12us$$

假设流水化设计以200MHz的频率运行需要3个周期，则其增加了25%的延时。

$$流水化延时 = \frac{3\ 个周期}{200\ MHz} = 15us$$

必须要记住的是，在吞吐量和延时之间进行权衡可能会影响一些数据流应用。

如图23所示，您可以采用时序图对执行次序进行研究。采样次数用下标数字表示，注意到子VI A开始处理第二个采样数据时VI B才开始处理子VI A上个周期的输出。

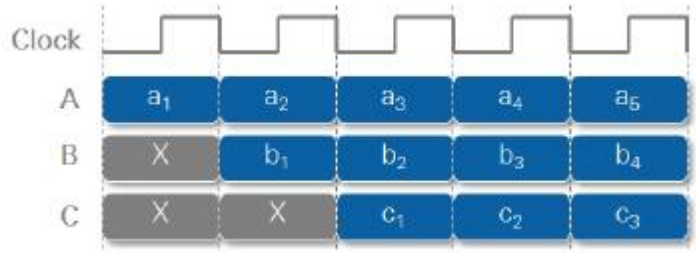

图23. 这一时序图展示了在相同迭代中，SCTL的不同部分是如何处理不同的采样数据。这使您可以为时序图设定更高的时钟频率，但是采样数据从输入传输到输出会需要更多的周期，因此您必须在开始流水线执行时先处理数据的有效性。

您可能会注意到在第个一周期中，B也会处理某些未知的数值，从而产生了无效的数据。C在前两个周期中也会出现该情况。因而，时序图中流水化会带来处理数据有效性的问题。

在流水化设计中，流水线在产生任何有效输出之前必须被“填充”或“装填”。该过程需要多个时钟周期。流水化的另一个含义就是下行区块，尤其是那些会产生副作用的区域，比如I/O、存储值的更新或通信，在流水线填充时应被告知无效数据。

向下行区块通知无效数据最简单的方式就是使用一个与数据一起传输的数据有效信号，如图24所示。

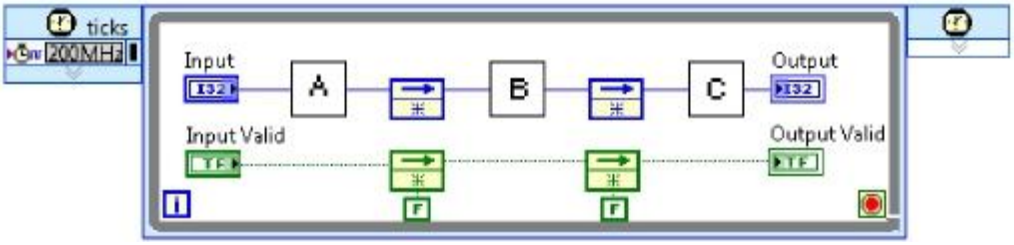

图24. 流水化设计生成一个有效采样需要多个周期，因此您可能需要一个与数据一起的布尔信号来指示输出可被下行区块消耗的时间。

由于数据路径的分支以及不同点的汇合，程序框图的复杂度不一。如果您的设计比较重视处理不同分支的数据一致性，那么您需要平衡分支的延迟以便函数能够采用适当的周期关系处理采样。图25展示了一个平衡代码分支延迟的例子，图中使用了一个寄存器节点并在配置对话中将延迟值设为2来实现分支延迟的平衡。

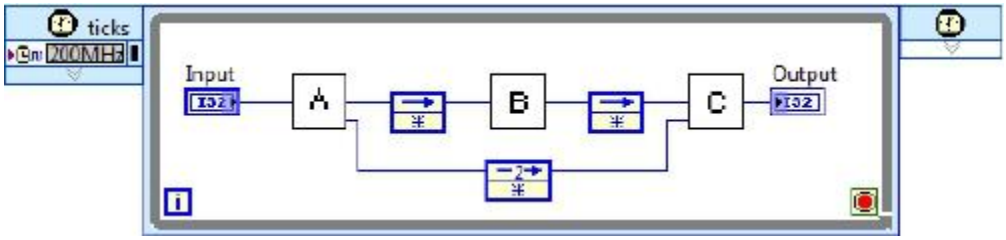

图25. 通过多次采样，反馈节点能够缓冲并延迟数据处理。这有助于平衡多个处理分支的数据延迟。

由于表示数据有效性的信号在图中是随数据流动的，因此也必须要对其进行分支并汇合。有些函数需要多个周期来消耗并生成有效数据，采用这些函数就牵涉到了更为复杂的流控制概念，该概念中数据有效性必须要在图中显式的处理。在集成高吞吐量IP章节中我们会讨论流控制技术，比如四线握手或AXI协议。

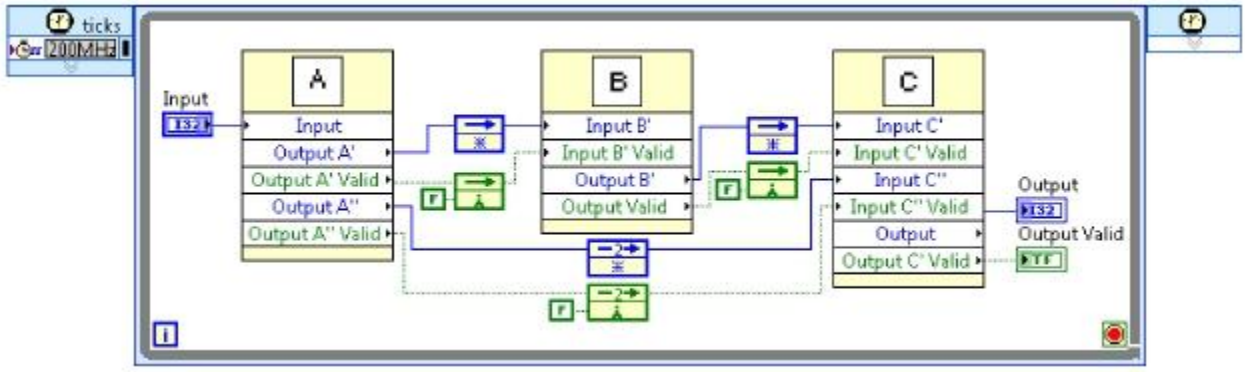

图26. 注意，数据有效性信号已添加到分支处理链中，以确保每个子VI处理的都是有效数据，且采用了适当的延迟关系，同时在输出端发出数据有效性的信号。

缩短启动间隔

当程序框图包含了每个输入需要多个周期来完成执行的IP，也就是多重采样/多周期IP时，您可以通过缩短此类IP的启动间隔来提高吞吐量。缩短某部分IP的启动间隔等同于变换代码，使其能减少调用之间的周期、以更高的频率接受采样数据。

图27显示的代码序列每两个周期处理一个采样数据。该代码包括三个阶段，分别用子VI A、B和C来表示。图中添加了流水化寄存器，以通过更高的SCTL时钟速率提高吞吐量。

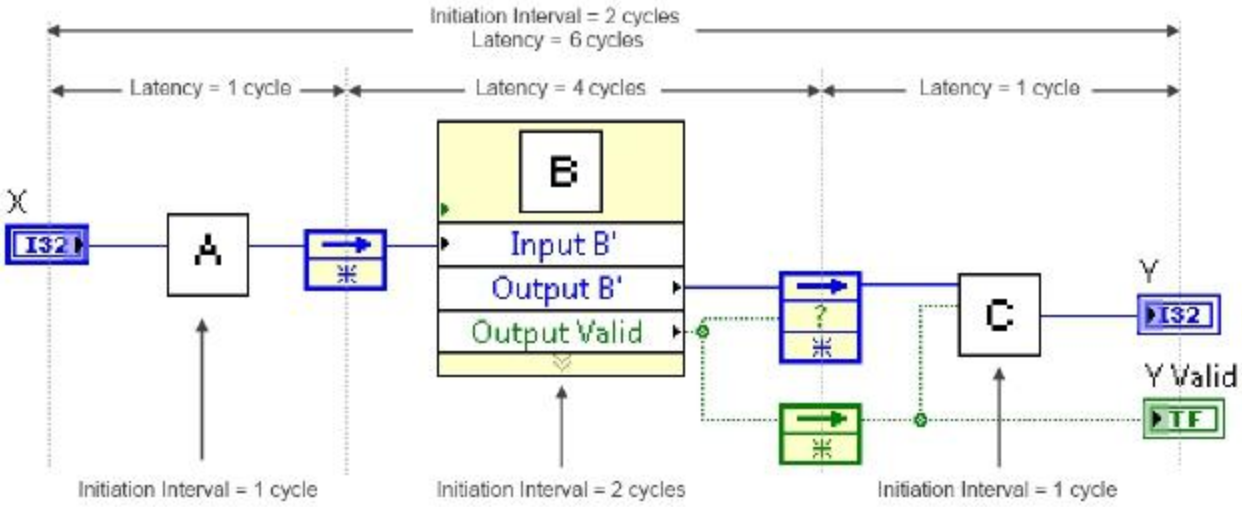

图27. 由于子VI B的原因，该代码的启动间隔是两个周期。当一个代码路径一次处理一个输入时，它可实现的最短启动间隔为路径中函数的最长启动间隔。代码路径的延时是路径中所有单个延时之和。为了简化起见，图中忽略了流水线启动时向下行区块发出数据有效性信号的代码。

该代码示例的一个重要特征就是子VI B在执行某些操作时，每个输入需要两个周期才能执行完毕，相当于一个启动间隔为两个周期。即使子VI B和C的启动间隔为一个周期，整个代码路径序列可实现的最短启动间隔也是两个周期，与链中最长的启动间隔相等。这会有两方面的影响：

1. 该代码序列有效地忽略了其他通过X端进入的采样数据。如集成高吞吐量IP章节所述，上行代码必须考虑到这些问题，或者被告知何时能采用握手信号提供新的输入。

2. 子VI B的输出有效信号必须传输给下行代码区块以表明输出的有效性。

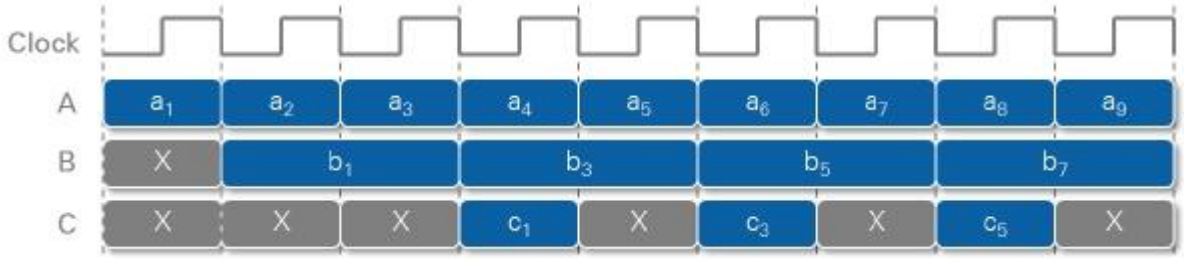

图28. 延迟可用采样数据从输入传输到输出所用的周期数来衡量，而启动间隔则以周期为单位描述了一个IP接收到一个新输入的频率，与吞吐量成反比。上例中的时序图表明代码的延迟为4个周期，启动间隔为2个周期。

请注意该图的延迟与启动间隔是不同的。该例中，任何给定的采样数据从X输入传输到Y输出需要四个周期，而时序图的X端却可以每隔一个周期接收新的输入。一种用来衡量延迟的方法是，给定采样从输入到输出所需的周期数，相比该周期数，流水化设计能在更少的周期之后就接收完数据。

制造装配线就是流水线过程的一个例子，在制造装配线上，多个工位将一个产品的不同部件装配起来，并能在给定时间内完成不同的订单。增加流水线过程的阶段意味着将其拆分为更简单的阶段，从而以更多工人和装配设备为代价，更快的完成任务并实现整体吞吐量的提高。

以十阶段的制造装配线为例，吞吐量为每分钟一个设备的话，它可能每分钟就要启动一个新设备的装配。每个单独的设备需要经过十个装配阶段，因此，若假定每个阶段持续时间为1分钟，那么整个设备生产延迟将会是10分钟。有时候我们无法将流水线阶段划分为更小的阶段。假设您不能再进一步划分流水线，就像图29中的子VI B，您或许可以将其执行进行复制和重叠，从而增加每个周期代码所能处理的采样数据量。注意前面的例子是如何进行修改的。也请注意First Call原脚本的使用会使第二个子VI B的运行延迟一个周期。

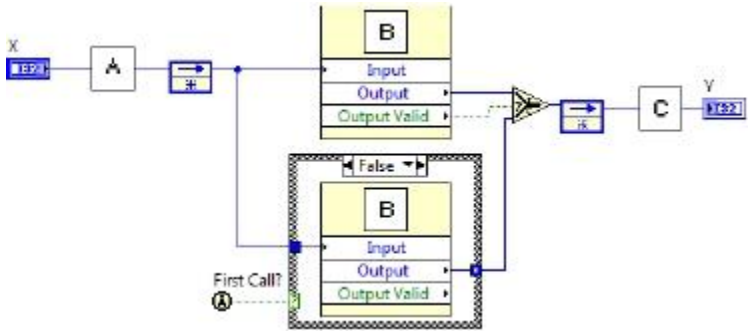

图29. 复制子VI B可使代码通过同时处理两个采样来实现每个周期处理一个采样，从而将代码整体吞吐量提升了一倍。

现在，修改后的时序图可每周期接受一个采样，以消耗更多资源为代价将原来的吞吐量扩大了一倍。注意到我们的假设是，子VI B中的操作能对每个采样独立执行。因此，保留状态或需要严格遵守采样顺序的操作就不能用这种方式并行处理。

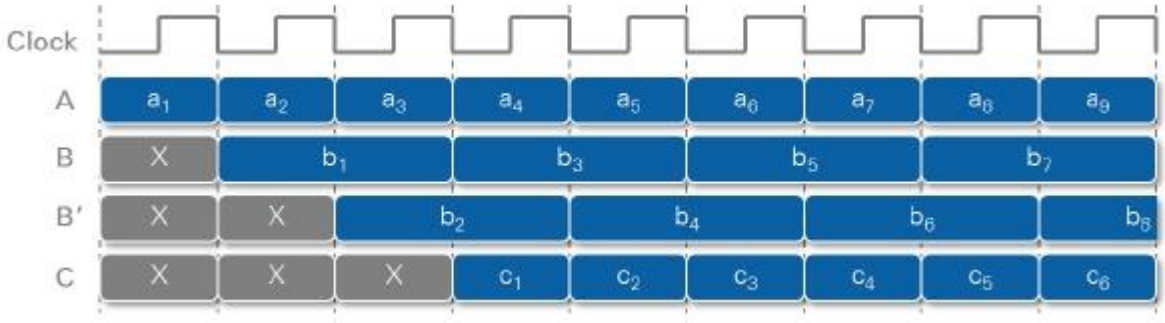

图30. 复制子VI B并错开其执行可将整体的启动间隔降低为一个周期。现在，代码就能够在SCTL的每次迭代中接收新的输入，从而将示例代码的吞吐量扩大了一倍。

该技术同时结合了流水线和复制方法。通过复制代码和状态被来实现对多个顺序采样的同时操作。这与前述提到的并行化技术有所不同，并行化是指并行代码的每个循环迭代接受多个采样，或者对相同采样并行执行不同的操作。而上例则是每个循环迭代只接受一个采样，但它以不同的完成度同时处理两个独立的采样。

集成高吞吐量IP

前一章介绍了用于编写您自己的处理IP的通用优化模式。无论何时，您应该尽可能使用现有IP来节省用于创建、验证和维护自己开发的IP的时间。本部分的概念包括常用LabVIEW FPGA IP源以及如何将其集成到您的应用程序。

推荐的LabVIEW FPGA IP源

LabVIEW FPGA有多种IP源。按照以下顺序考虑这些源：

1. LabVIEW FPGA选板

   LabVIEW FPGA选板上有由NI公司验证和优化好的用于NI RIO设备的IP。您可以用各种针对特定应用或者市场的工具包来增强这些选板的功能，避免开发专用IP。

2.集成IP生成器

   IP生成器是一种可以使用通用算法或者函数的工具，并可生成专门针对您所需性能或者功能需求的IP。

   a. Xilinx CORE生成器系统
   b. LabVIEW数字滤波器设计工具包和其他生成器
   c. LabVIEW FPGA IP生成器模块

3. LabVIEW FPGA社区

   LabVIEW用户社区提供了分享和再利用LabVIEW FPGA IP的场所。社区站点包括LabVIEW 工具网络、IPNet、NI FlexRIO以及软件设计的仪器IP社区。

4.硬件描述语言(HDL) IP

   LabVIEW FPGA提供一些机制，用于集成使用HDL或者衍生自HDL的IP，比如VHDL和Verilog。

下列部分包含从这些源文件以及IP集成技巧相关的IP范例。

LabVIEW FPGA 高吞吐量函数选板

高吞吐量FPGA函数选板提供专为高性能应用程序而设计的IP。尽管选板上的一些节点也可以用在SCTL外面，但是大多数是预期用在SCTL内部的。高吞吐量选板的大多数函数提供配置对话框。您可以借助此对话框稍微调整行为、资源占用以及性能特性。

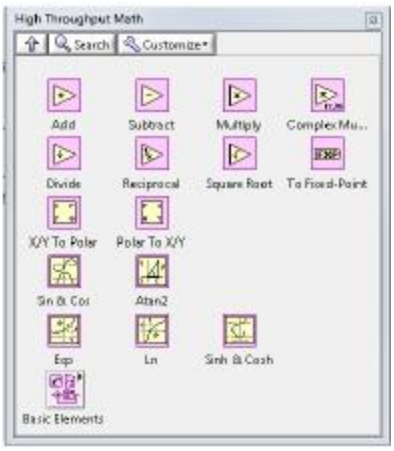

图31. 高吞吐量数学选板提供了专为SCTL内部使用而优化的函数。此选板也包含子选板，用于容纳诸如离散延迟和DSP4 8节点等专用FPGA资源。

高吞吐量函数利用专用FPGA组件和资源位置来降低逻辑和路由资源占用。这些函数还利用流水化技术来获得更高的时钟速率。

许多这些函数的文档包含如何对函数进行配置以平衡吞吐量、延迟、资源以及数值精度取舍的细节。以下介绍了其中的几个节点，阐述了节点的配置如何影响性能以及您如何使用节点来利用专用FPGA资源。

高吞吐量乘法函数

此节点用于计算定点数对的乘积。图32显示的是此节点的配置对话框。此配置对话框也为选中配置的预期性能和数值行为提供反馈。

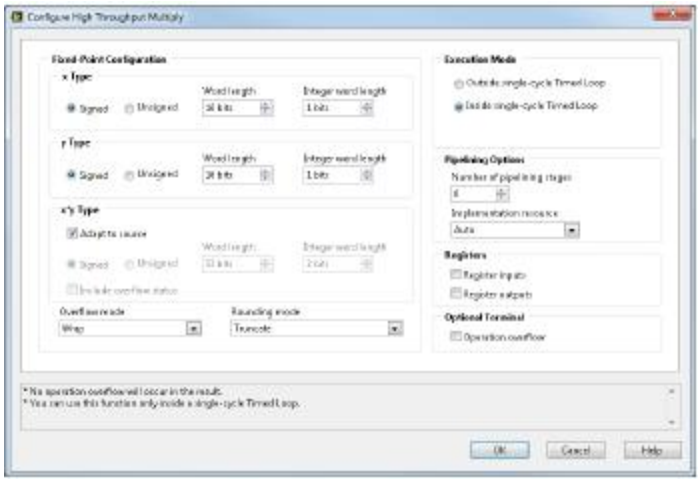

图32. 高吞吐量乘法函数提供一个详细配置对话框，您可以用它来平衡函数和性能需求。

下列配置会影响节点性能：

定点数配置

正如资源优化技巧一章讨论的那样，节点的性能和大多数逻辑和算法运算在很大程度上会受到操作数和输出宽度的影响。

溢出模式

溢出条件可能发生在输出端的数据类型不够大，无法覆盖两个输入的乘积。对于探测和报告溢出条件以及溢出条件发生时的饱和数量，需要增加额外电路。

参见产品文档了解溢出条件和何时使用每种配置的细节。通常使用Wrap模式提供更高性能，消耗更少资源。这是在数值精度和性能之间进行可能的取舍的一个范例。

取整模式

取整发生在值的预期精度大于代表它的数据类型精度时。当LabVIEW强制取整时，可能会丢失精度。

取整需要额外的逻辑，这反过来会增加对资源的占用，限制了性能。通常尽可能地使用Truncate（截位）模式更加可取，因为这种模式舍弃了最低有效位，以匹配输出的更低精度。但是请注意这种做法可能会使数值结果偏向负无穷大。这种由于取整模式造成的数值偏移是在数值精度和性能之间进行取舍的又一例子。

参见资源优化技巧一章或者LabVIEW产品文档中使用定点数类型（FPGA模块）一节，了解取整以及溢出对FPGA性能、资源占用和数值精度的影响的更多信息。

流水线、寄存器和实现资源

将乘法器针对SCTL进行配置时，它可以实现内部流水线，并将寄存器添加到输入和输出端，这种做法分解了处理路径，使编译速度更快。

您还可以通过设置在查找表（LUT）和嵌入式乘法器(DSP48块)之间自动选择来实现此函数，以此来决定编译器使用的资源类型。DSP48块是专用Xilinx FPGA元件，而LUT则是通用逻辑块。这是高吞吐量节点利用专用资源或者帮助您在不同资源类型之间平衡资源占用的一个范例。参见本章稍后的DSP48节点部分了解更多关于DSP48的信息。资源优化技巧一章会详细阐述LUT的作用和专用FPGA资源。

如图33所示，最大可实现的性能根据可利用的资源、流水线阶段数量、数据类型宽度的不同而变化。

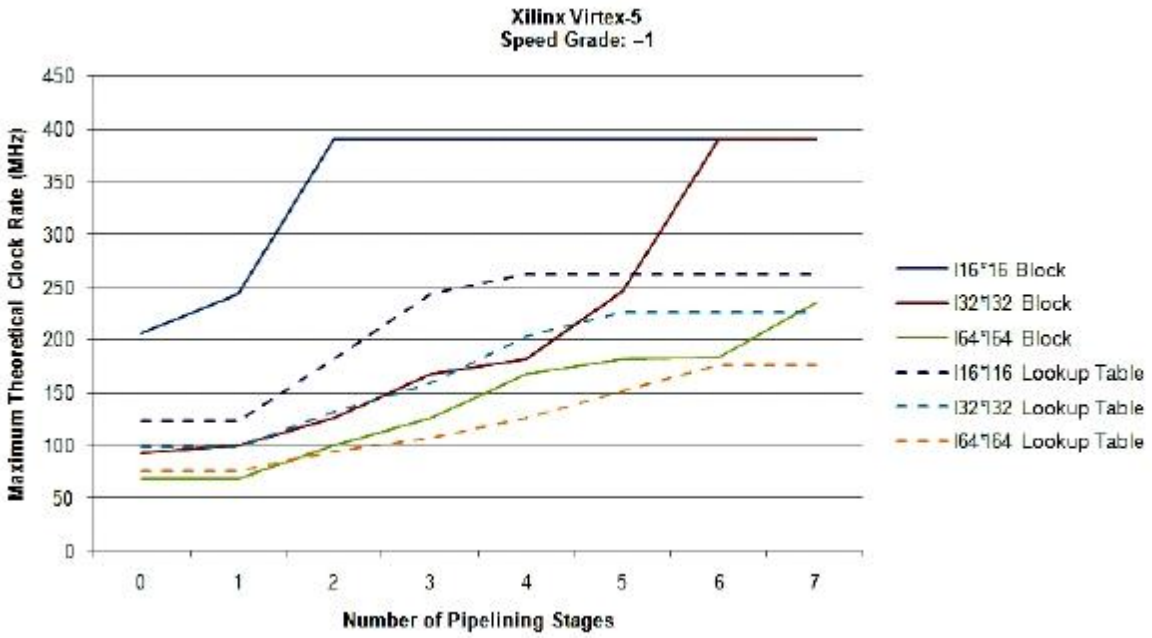

图33. 乘法器函数的最大时钟速率根据用于实现此函数的数据类型、流水线阶段数量以及资源类型不同而变化。较窄的类型结合DSP48块的使用可以提供最佳性能。

IP握手协议

将节点针对SCTL进行配置时，高吞吐量数学函数选板上的节点可以用多个周期来初始化或者生成输出，这就需要在其接口上使用握手协议。

握手协议是处理链中提示上下行模块的信号集。比如，在DSP和图像处理应用等情况中，握手协议用于提示输出的有效性或者接受新输入的意愿。

正如在吞吐量优化技巧（23页）一章中谈论到的，握手对于SCTL是必要的，因为多周期的节点需要多个周期来计算有效数据，但是SCTL会强制这些节点在每个时钟周期必须返回一些数据。因此，多周期节点不会每个时钟周期都返回有效数据。为了确保算法的数值准确度，该数据相关的节点必须知晓数据是否有效。

单周期执行的函数也必须向上行模块传输握手信息。其必要性是由于下行函数并非在执行的所有时间内都可以接受数据，所以上行代码必须控制整个数据流。

四线握手协议

高吞吐量数学函数面板上的函数支持在SCTL内部创建处理链的握手协议。因为协议有四个接线端，所以FPGA函数和VI中的握手协议有时也被叫做四线协议。此协议包含下列接线端：

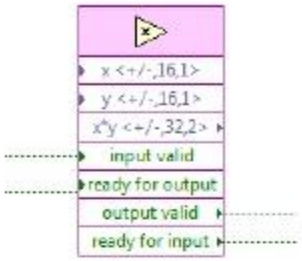

图34. 高吞吐量乘法函数是提供四线握手接口的一个函数范例。您可以使用四个信号来控制SCTL内部的有效数据流。

输入有效

握手函数需要知晓即将到来的数据的有效性。信号指定了传入此函数的数据在函数执行前是否有效。如果上行代码也实现了四线协议，那么您可以把上行的output valid信号连接到input valid接线端。

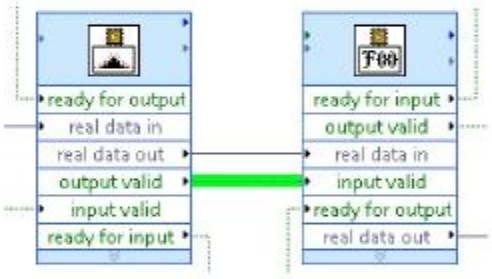

图35. 两个支持四线握手接口信号的函数通过连接上行函数的output valid信号到下行函数的input valid输入端来传输有效的数据。

输入数据可能在每个周期都是有效的。在这种情况下，您可以将“真”常量连接到此输入，如图36所示。

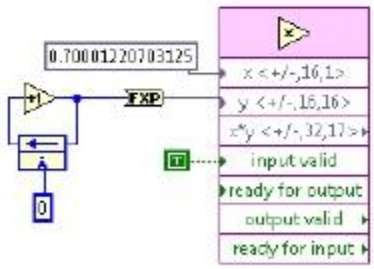

图36. 您必须连接一个值到Input Valid端，即使即将输入的数据总是有效的。

上行代码可能不会提供output valid信号，所以您必须找到其他方法来决定数据的有效性。图37所示的例子采用FIFO Read 方法，其Timeout输出作为数据有效性的显示控件。如果Timeout输出为真，那么从FIFO Read方法出来的数据就是无效的。

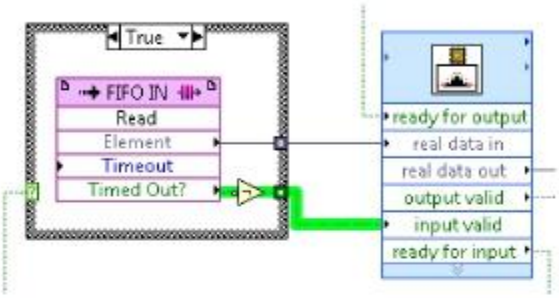

图37. 不是每个函数或者节点都提供四线握手接口，但是通常您可能会找到一种方法推断数据的有效性，正如FIFO Read方法那个案例一样。在FIFO的Timed Out?端为真时，FPGA FIFO 方法的Element输出是无效的。

输出有效

函数执行后，此输出指示相关输出信号是否包含有效数据。

下行代码通常需要确认传入数据的有效性，特别是输入对I/O、存储器或者状态变化或者数据传输操作等存在副作用。

如果下行代码也实现了四线协议，可以将函数output valid信号连接到相应的下行函数input valid端。

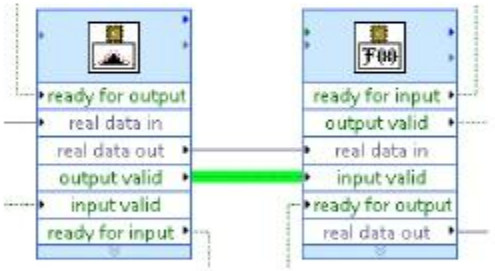

图38. 如前所述，通过连接上行函数的output valid信号到下行函数的input valid输入，四线握手函数传递出每个数据的有效性。

下行代码可能没有内置的握手端或者没有途径得知传入数据的有效性。这种情况对LabVIEW FPGA中许多I/O节点以及通信和存储结构都是成立的。在这种情况下，如果节点生成无效数据，您可以使用一个条件结构来阻止下行代码的执行，如图39所示。

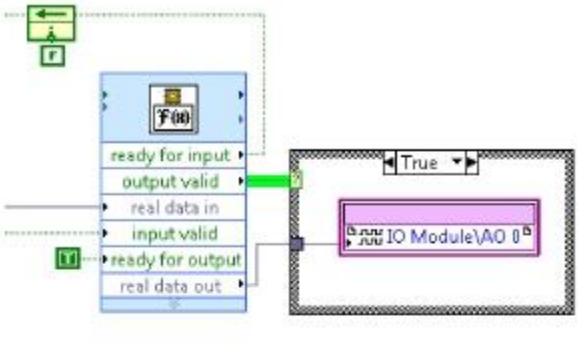

图39. 并非每个函数或者节点都会提供四线握手接口来阻止传入的无效数据被执行。基于output valid信号值，您可以使用一个条件结构来有选择性地执行下行代码。

准备输出

此输入信号可报告下行节点是否接受此节点的新输出。如果下行节点也是四线标准，那么此信号可以被理解为当前迭代准备好接受数据。

由于信号通常来自下行节点，因此必须使用一个反馈节点来阻止程序框图上的一个周期。而且反馈节点必须初始化为“假”，除非下行代码总是可以在首次调用时就接受数据。

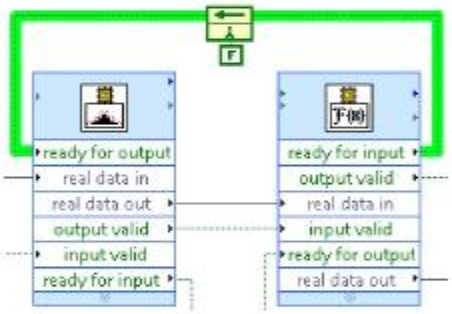

图40. ready for output输入会通知握手的函数下行代码可以接受当前循环执行产生的数据。

下行节点可能不是四线标准，但是可能包含决定是否接受当前或者下一个迭代数据的信号。FIFO Write方法就是这样的一个范例。

当最新的元素不能写入FIFO，而Timeout 端的FIFO Write方法输出“真”时，最可能的原因是FIFO已经写满了。您可以使用这个方法来保持上行块无法接受更多数据，并保留尝试写入的元素值，这样您之后才可以再次尝试写入操作，如图41所示。

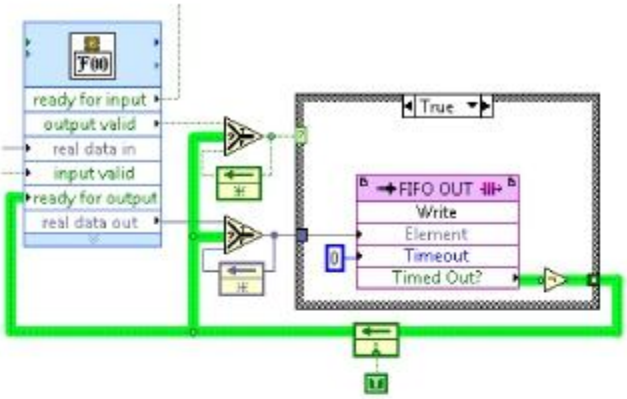

图41. FIFO Write方法中的timeout指示尝试写入时FIFO已经写满了，所以上行节点应该停止提供新的值，直至FIFO空间可用时，才会存储最新的元素。当上行代码使用握手接口时，此编码模式对于避免数据丢失是非常重要的。

准备输入

此输出告诉上行节点握手节点可以消耗下一个周期的数据。既然信号上行传输，则需要一个反馈节点来阻止程序框图的一个周期。

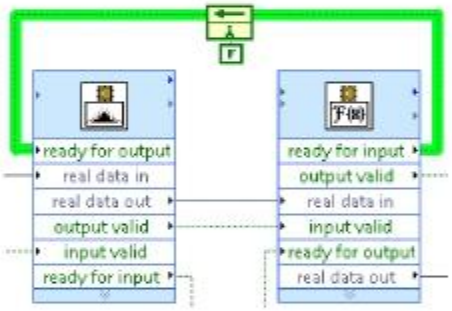

*图42. 当两个有四线接口的函数连接时，下行函数的**ready for input**信号应该通过反馈节点连回到上行节点。*

上行代码可能不兼容四线标准，或者没有握手端。这种情况下，ready for input信号可能需要重新编译或者用于控制条件结构来阻止上行节点执行。

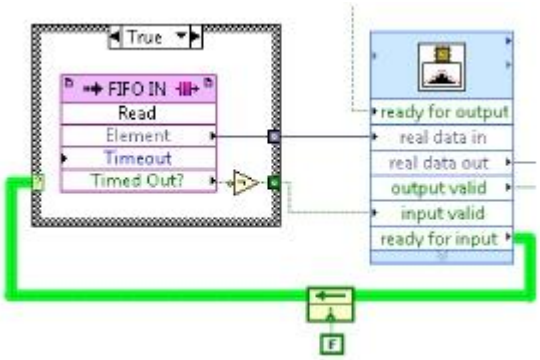

图43. 当上行代码没有四线握手接口，ready for input信号应用于控制代码执行。
在握手函数准备接受时，使用条件结构或者其他机制来确保代码仅生成有效数据。

验证握手逻辑

不正确的握手信号连线可能导致不正确的行为、数据丢失、数据损坏、数据重复以及不理想的性能。
握手连接可以通过在仿真模式下执行代码和在多个SCTL迭代中观察其行为来进行验证。

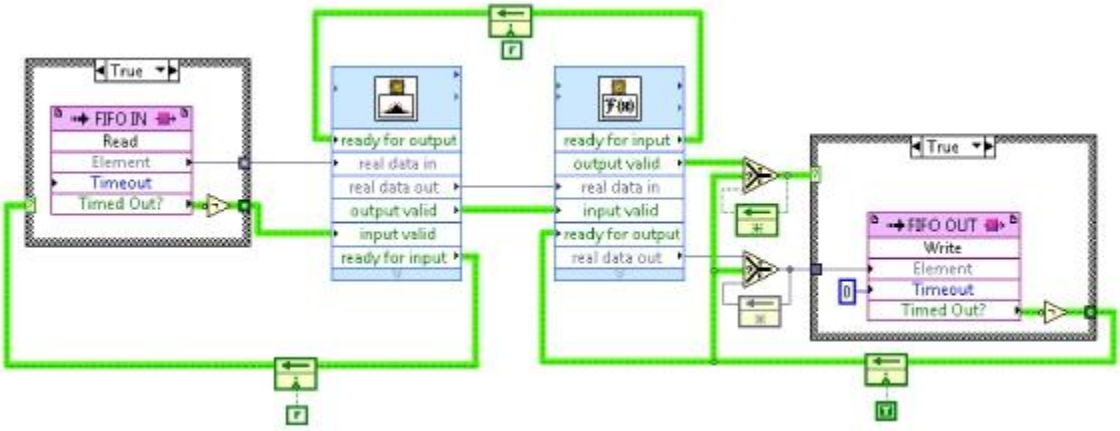

图44.此 SCTL代码从FIFO读取采样数据，并在采样数据写回FIFO执行一次加窗快速傅立叶变换，
以便在其他循环中进行进一步处理。代码包含大多数前一部分提到的四线握手情况。图上的许
多连线是帮助维持有效数据流的握手信号。验证这些信号对于保证代码功能的正确性非常重要。

您还可以使用仿真模式，通过确定采样数据通过信号链传输所需的周期数以及在IP接受新输出前经过的周期数来测量延迟和IP的启动间隔。IP接受新输出是由ready for input端进行指示的。

采样探针是一种新的LabVIEW探针，专为验证LabVIEW FPGA IP的握手和功能行为而设计。在SCTL以仿真模式执行时，采样探针提供数据的数字波形视图，如图45所示。

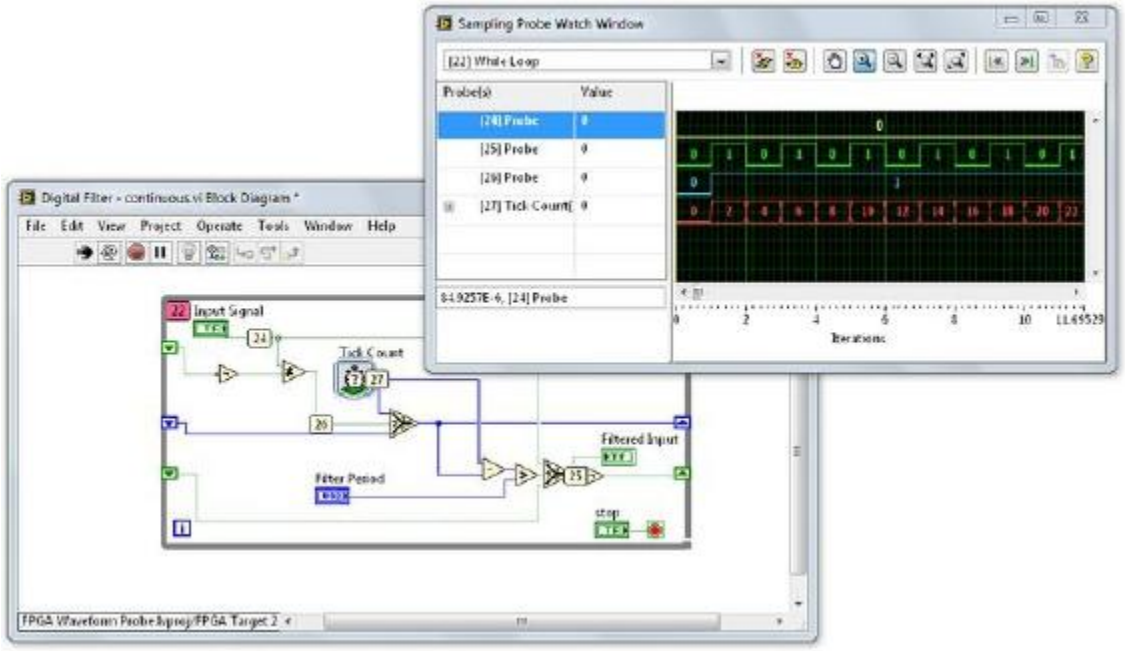

图45. 采样探针显示了仿真模式中的历史连线值。这些探针可用于验证IP的正确性和性能，包括验证任意握手信号的功能。

确定处理链的吞吐量

许多LabVIEW FPGA函数提供关于启动间隔的信息。您可以使用此信息为SCTL内部已连接的节点计算最大可用吞吐量。正如吞吐量优化技巧一章（23）中所述，节点链条的启动间隔等于序列中最长的启动间隔。启动间隔最长的节点是处理链的瓶颈。

配置对话框、即时帮助或者引用主题均有记述节点的启动间隔。有时启动间隔被称为吞吐率，但是它以周期/采样点数为单位，这和吞吐量成反比。

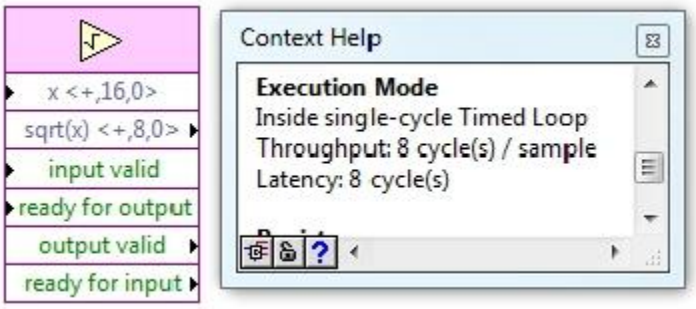

图46. 许多高吞吐量函数的即时帮助基于函数的选中配置提供了预期吞吐量（在启动间隔周期内）和延迟的记述。

DSP48节点

DSP48节点可帮助您了解如何利用LabVIEW FPGA提供的专用FPGA资源来创建高性能运算IP。

DSP48是包含于特定Xilinx FPGA产品系列（如Xilinx Virtex-5）的数字信号运算元件。您可以用该芯片来实现不同的算法运算，包括乘法累加器、乘加器和单阶或N阶计数器。您还可以将其用于实现不同的逻辑运算，比如与、或、异或运算。

Xilinx产品系列提供不同的DSP48节点，比如DSP48E或 DPS48E1，但是其基本的功能是相似的。您可以级联多片DSP48E来实现较大型的函数，比如复数乘法器和N-抽头有限脉冲响应（FIR）滤波，而不需要任何额外的FPGA结构资源。结果，基于DSP48片的IP可以用更快的时钟速率编译。

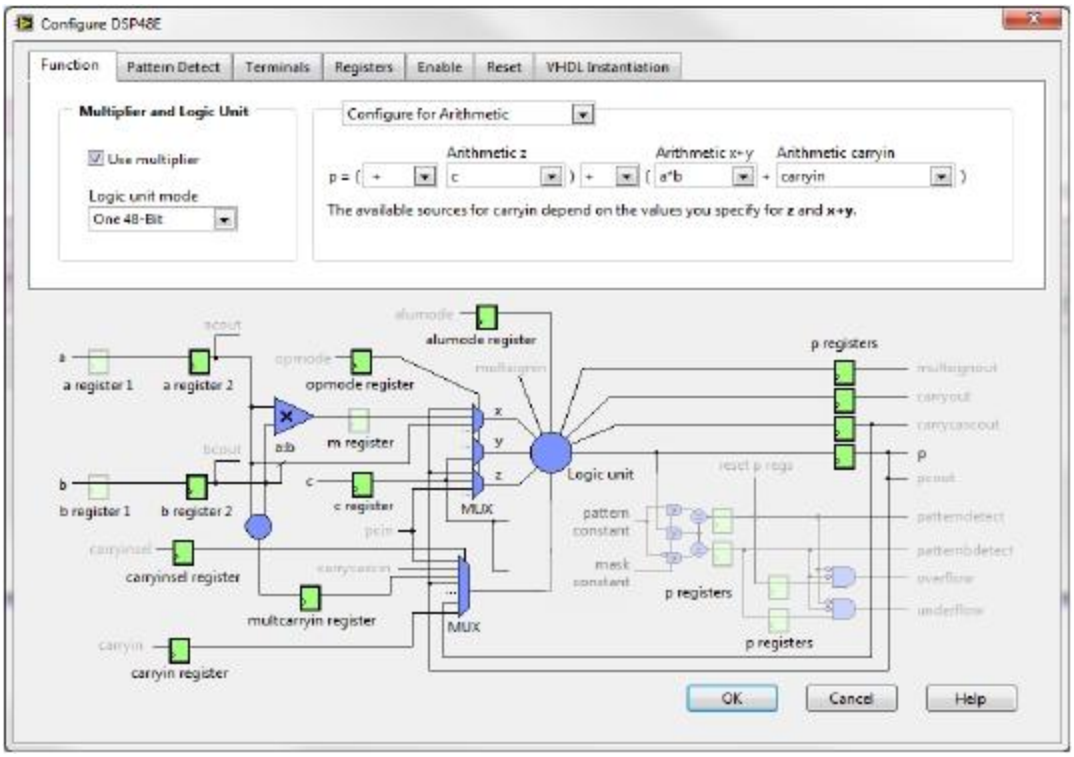

图47. 在您启用并配置不同功能时，DSP48E节点的配置对话框提供DSP48E电路的图形化视图。

由于学习此节点需要较长的时间，因此一些需要极端优化的情况才需要了解DSP48。在尝试配置此节点前，可参见《Xilinx Virtex-5 FPGA XtremeDSP的设计考量》中关于DSP48E描述和规范一章。文档请访问Xilinx网站。DSP48E应用程序一章包含了采样应用程序的清单，并提供性能、配置细节和附加说明的详细信息。如果您需要应对类似于高性能需求的挑战，可查看该章节。

您可以使用此节点完成下列采样IP任务。

基本数学应用

- 25 x 18二进制补码乘法
- 双输入48位加法
- 四输入46位加法
- 双输入48位动态加/减
- 三输入47位动态加/减
- 25 x 18 乘和48位加/减
- 扩展乘法
- 浮点数乘法和59 x 59 带符号乘法
- 25 x 18 乘和48位加层叠除

滤波

- 多相插值FIR滤波器
- 多相抽取FIR滤波器
- 多通道FIR
- 预滤波系数

高级数学应用

- 累加
- 双输入48位加积
- 动态加/减积
- 96位加/减
- 96位累加器
- MACC和MACC 扩展
- 25 x 18 复合乘法
- 35 x 25复合乘法
- 25 x 18复合MACC

逻辑和位域的应用

- 双48位输入位逻辑函数
- 动态移位器
- 18位桶型移位器
- 48位计数器
- 单指令流多数据流(SIMD) 算法
- SIMD 绝对值 (24)
- 总线主控器

模式检测应用

- 动态 C输出模式匹配
- 溢出/下溢/过饱和P[47]
- 溢出/下溢/过饱和 P[46]
- 逻辑单元样式检测
- 48位计数器自动重置

舍入应用

- 舍入决定
- 动态或者静态小数点
- 对称舍入
- 随机舍入
- 收敛舍入
- 收敛舍入：LSB 修正技巧
- 动态收敛舍入：运载修正技巧

LabVIEW FPGA也包括几个针对DSP48片设计的随附范例，作为配置节点的引用。您可以通过NI范例查找器，搜索“DSP48”关键词来找到DSP48。

快速傅立叶变换

由于FPGA资源有限，建立在SCTL内部的处理链通常以点对点为基础来处理数据，将值从一个区块传输到下一个区块。有些函数要求每次只处理一个数据块，这需要在处理采样数据之前花费多个周期来缓存采样数据。位于FPGA Math & Analysis VI和函数选板上的快速傅立叶变换(FFT)函数就是此类函数的一个范例。

FFT函数操作数据块，也称为帧，并且提供流处理，也就是连续处理选项，以及突发处模式。

在突发操作中，FFT函数一次接受并缓存一个点的数据。一旦函数搜集到足够的数据填满一帧，函数就会停止接受新数据，并开始处理缓存的帧。几个周期之后，FFT函数提供输出帧，一次一个点。在时钟开始记录最新的结果帧之后一段时间，函数开始接受新一帧的数据，如图48所示。

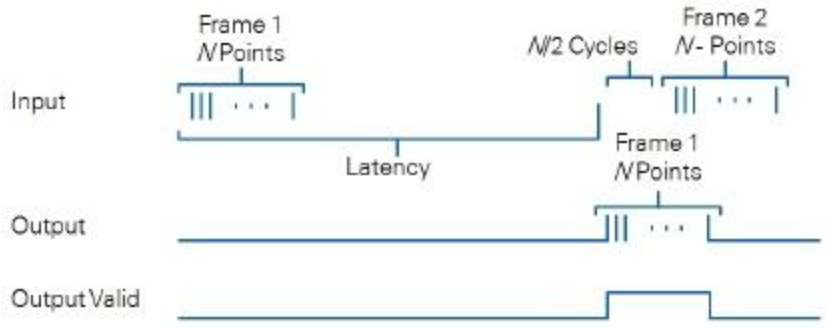

图48. FFT函数操作可以用于突发模式或者数据流模式。在突发模式，函数每次操作一帧，在它可以接受新输入帧之前，需要花费几个周期来接受、处理和输出结果。

突发模式导致启动间隔需要多个周期。FFT函数配置对话框为预期启动间隔提供反馈。启动间隔取决于FFT块的大小以及内部流水线和并行化的数据量。

在数据流模式下，FFT函数使用吞吐量优化技巧一章（23页）提及的一些原理来实现每采样点数/周期的启动间隔。这意味着它可以连续接受新采样数据。函数仍然需要在它可以处理数据前缓存数据，因此需要添加缓存和电路才能使函数接受下一个FFT输入块，同时仍可处理当前帧和输出之前的结果，如图49所示。

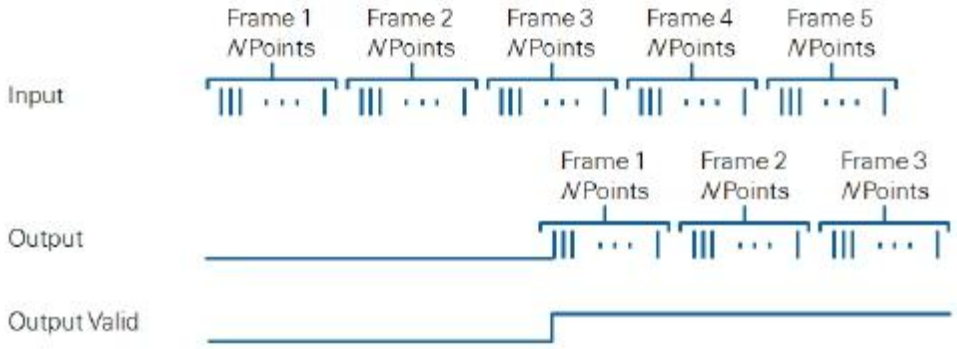

图49. 在数据流模式，FFT函数每个周期都可接受新的采样数据。FFT用于缓存采样数据、并行化FFT计算以及同步输出先前帧的结果。此模式提供了最佳吞吐量，但占用了大量的资源。

数据流模式提供最高FFT性能，允许系统对每个即将输入的采样数据执行一次FFT。由于内部运行非常复杂，并且必须不时重复此操作来保留和处理多个帧，因此该模式会占用大量资源。

XILINX内核生成器IP系统

通过Xilinx Coregen IP函数选板，LabVIEW FPGA可以直接访问Xilinx开发的内核生成器IP系统。内核生成器IP系统是按架构、领域和市场细分的IP目录，用于汽车、通信、信号处理和其他应用。Xilinx提供这些高度优化的IP块来帮助Xilinx FPGA用户集成高性能设计。这些块与LabVIEW FPGA开发环境紧密集成，并且包含一个分步配置对话框，可帮助您自定义选中内核的性能、数据类型和其他IP行为。

图50展示了Xilinx内核生成器FIR滤波内核及其配置对话框，在图中以一个LabVIEW FPGA函数表示。此内核提供配置环境，可用于自定义FIR滤波器，以满足您的具体应用需求。

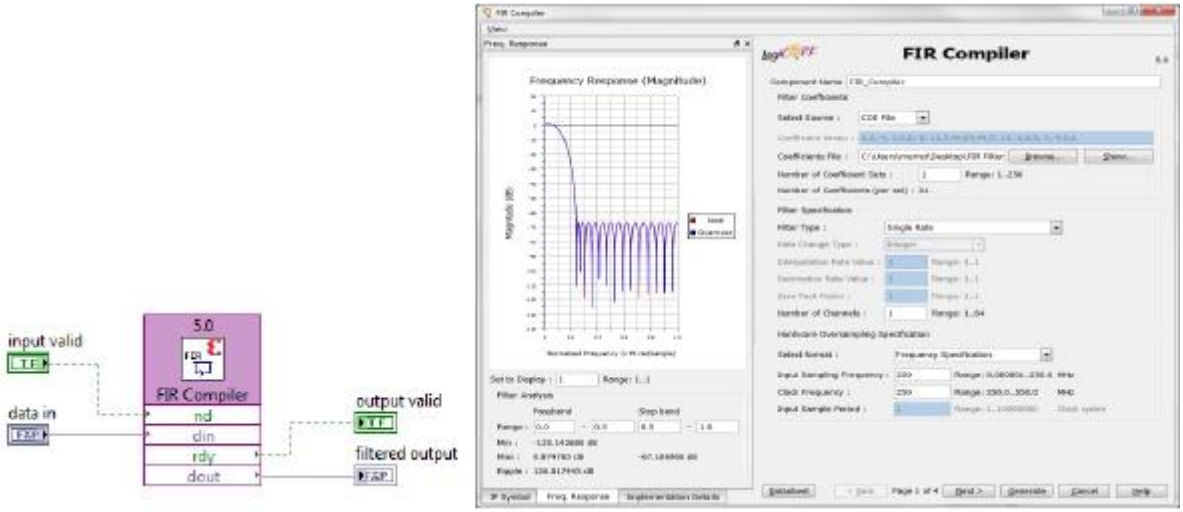

图50. LabVIEW FPGA通过Xilinx内核生成器IP系统与IP紧密集成。此函数拥有自己的选板，可以直接从LabVIEW开发环境中启动合适的Xilinx内核配置对话框。

因为这些函数是由FPGA设备制造商——Xilinx开发的，与NI提供的类似函数相比，这些函数通常可提供更多的功能、性能或者节省资源，所以可将其作为优化设计的方法之一。LabVIEW FPGA选板上有些最高级的Xilinx内核生成器IP需要购买许可证，但是如果您需要高度专用的IP，这些函数可帮助您减少实现细节和测试，绝对物有所值。

XILINX内核生成器FFT IP

Xilinx FFT内核是一个受欢迎的IP块，与LabVIEW FPGA 模块中FFT Express VI的特性互为补充。与FFT Express VI类似，Xilinx内核可以在数据流模式或者具有不同性能和资源取舍的三种突发模式下运行。另外，在数据流模式下，Xilinx FFT会消耗大量的资源。除非应用明确要求，否则可能要尽量避免使用这种模式。图51粗略展示了Xilinx每种模式的性能和资源取舍。

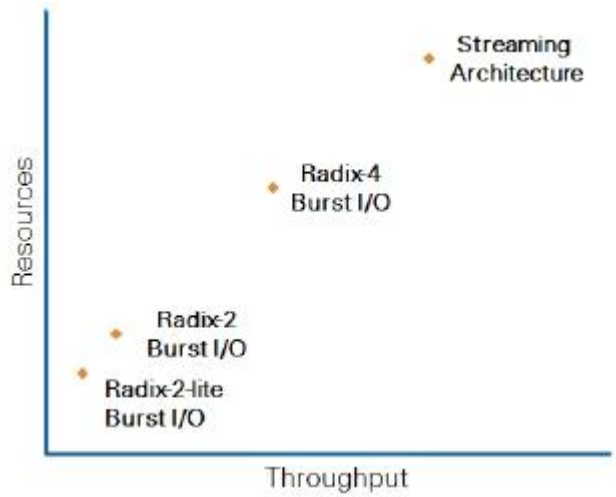

图51.Xilinx FFT的不同操作模式在吞吐量和资源之间进行了不同的取舍。突发模式使用的资源最少，但是需要多个周期来处理每个帧。

在FFT Express VI 和Xilinx内核生成器FFT之间进行选择时，请考虑Xilinx内核生成器具有以下特点：

- 针对吞吐量高度优化，所以通常会以更高的时钟速率编译
- 支持大于8192个采样点数的帧

- 在突发模式下支持多通道操作
- 支持浮点数据类型，虽然需要消耗较多资源

- 具备运行时可配置的FFT帧尺寸，可达到编译时预设置的大小（注意改变的帧大小需要重启内核）

- 正向或者反向模式下可基于每帧进行配置（您可以使用FFT函数实现正向FFT，在频率域操纵信号，然后重新使用相同的FFT函数或者硬件电路来实现反向FFT，并且把信号反变换到时间域。）

Xilinx内核生成器FFT的劣势包括以下几点：

- Xilinx内核生成器FFT的配置和使用可能比FFT Express VI困难些。
- 此函数缺少与四线协议一致的标准握手接口。例如，它没有**input valid**信号，正如这一章早前描述的那样，在这种情况下，您需要把此函数放入条件结构来避免输入无效数据。

- 从内核获得最快速性能需要一些高级配置，并且它会强制您的设计基于某些假设。例如，最高性能模式生成的帧是以反向排序采样数据。下行块必须在这种限制条件下保持吞吐率。

如果您考虑在设计中使用此内核，可阅读Xilinx LogiCORE IP快速傅立叶变换数据表的前几章。此技术文档假定您已经掌握一些数字设计知识，但是该文档资源也可帮助您获得最佳FFT性能。类似的建议也适用于LabVIEW Coregen IP选板中的其他Xilinx内核生成器IP。

AXI 协议

美国国家仪器公司针对特定硬件终端上的特定Xilinx内核开发了AXI（Advanced eXtensible Interface，高级可扩展接口）握手协议。AXI协议于1996年由ARM推出，现在已成为高性能、高频率应用中用于互连IP功能块的行业标准总线接口。

在LabVIEW FPGA模块的环境中，AXI协议与LabVIEW四线握手协议在信号连接方面相似。每部分数据都包含两种信号：一种用于值，另一种用于指示值是否有效。

使用AXI时，必须使用反馈节点来连接AXI IP。AXI协议和LabVIEW四线握手协议主要的不同之处在于信号的命名和在SCTL内互连IP时反馈节点的放置。参见LabVIEW FPGA产品文档中的在FPGA VI中连接AXI IP，更详细了解如何将AXI节点连接至其他AXI节点以及如何将AXI节点连接到其他支持四线协议的节点。

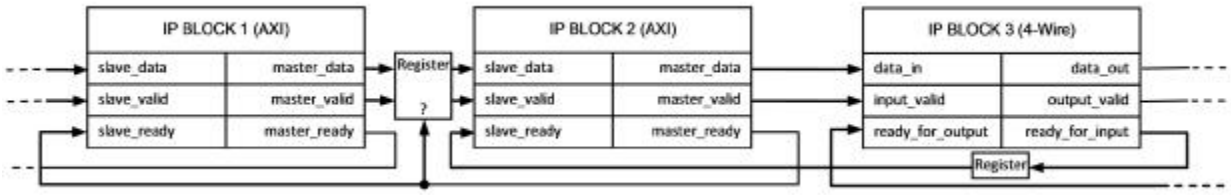

图52. AXI和四线握手信号虽然名称不同，但是含义相似。反馈节点（标注为寄存器）的放置基于IP互连类型的不同而不同。

集成HDL IP

硬件描述语言(HDL)是基于文本的语言，用于设计针对FPGA和专用集成电路(ASIC)的硬件组件。这些语言提供标准的方式来表示逻辑、信号、连接、并发、定时和其他硬件概念。同时它们可显示结构、行为以及底层实现细节，无需设计者门对门地定义电路。

在这本指南中，HDL IP指的是使用VHDL 或者Verilog编写的IP，或者是用此类语言生成的网表文件。

VHDL 和Verilog在数字设计行业有着悠久的历史，并且被大多数EDA工具支持。它们代表了基于FPGA的设计的传统方式。网表这一术语指的是HDL设计中预先编译的输出。网表的表现形式更接近于最终电路，因为它直接定义了FPGA硬件元件以及元件之间的连接。网表有时也可用作一种无需展示HDL源代码的IP集成或分享方法。

您可以从许多地方获得HDL IP。您的组织可能正在使用传统的数字设计工具，这种情况下您可能已经获得HDL IP或者具备相关专业技术来开发HDL IP。您还可以从一些社区网站，比如opencores.org，找到开源的HDL IP。其他数字设计公司也会开发针对特定市场的HDL IP，使用此种IP需要授权许可。

如果您已经有IP或者认为用HDL创建IP会更容易的话，您可能想把HDL IP集成到LabVIEW FPGA设计中。HDL为底层硬件和工具链设置提供了接口，所以当您需要极致性能和资源控制时，这为您提供了另一种选择。

将HDL IP导入到LabVIEW FPGA有两种机制：

通过IP集成节点(IPIN)
作为组件级别IP(CLIP)

选择IPIN或CLIP

IPIN 和CLIP机制之间有许多不同之处，但使用哪一种机制取决于您希望集成的IP类型。

使用IPIN来集成IP本质上是使用函数和算法，或者作为一个处理函数，每次调用时如果提供输入，IP就会生成输出。

使用CLIP来集成IP作为一个独立异步的组件，比如与应用程序的多个部分交互的IP，或者与外部数字组件连接的接口，比如存储器、SPI或者I2C。

IPIN必须放到SCTL内部，并遵循LabVIEW调用模型，即数据通过节点传送，节点可能生成输出数据，以备其它节点消耗。IPIN不能直接通过HDL代码访问I/O，所以它必须接受或者生成作为输出或者输入所需的值。将IPIN放入SCTL内部时，它代表了一个单独的IP例程。如果您去掉两个节点，就会有两个IP例程。

与IPIN相反，CLIP不会出现在程序框图上，因为它只存在于LabVIEW项目中。CLIP在FPGA VI外面运行，您可以多次复制给定的CLIP，并将这些CLIP实例化。LabVIEW FPGA框图通过CLIP附件节点与CLIP IP交互，这与I/O节点类似，并且允许您读写CLIP接口寄存器。在NI FlexRIO硬件中，CLIP也可以访问I/O引脚。这种类型的CLIP也称为“嵌入的”CLIP。

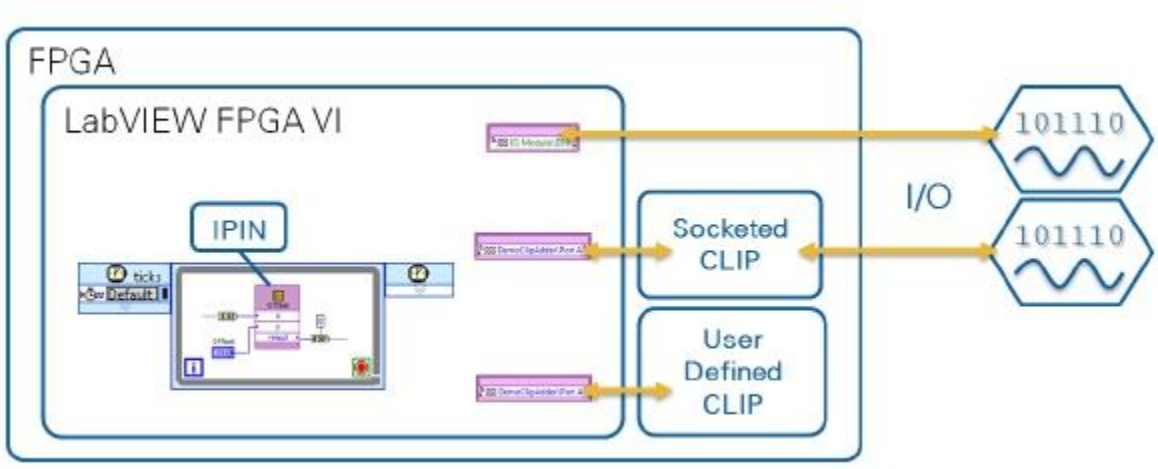

图53. CLIP和IPIN是在LabVIEW FPGA中的HDL集成机制。当IP可以作为函数模型时适合选择IPIN，而当IP必须作为组件执行时最好选择CLIP。取决于您的RIO设备类型，CLIP也可以访问I/O。

下列表格总结了CLIP 和IPIN之间的其他一些不同之处：

CLIP	IPIN
支持多个时钟域，并可以用FPGA VI创建和共享时钟	SCTL定义了包含IPIN节点的时钟
支持与FPGA VI的连接	属于LabVIEW FPGA程序框图的一部分
支持某些设备上的FPGA I/O访问	无法在HDL IP内访问 I/O
支持约束文件	接受浮点和其他高级数据类型
支持第三方模拟器，比如Mentor Graphics QuestaSim 和Xilinx ISim	支持LabVIEW FPGA的嵌入式仿真

参见产品文档和产品包含的教程范例，了解关于如何使用集成机制的更多信息。

集I成P到软件设计的仪器

借助NI矢量信号收发仪（VST）和其他软件设计的仪器，您可通过LabVIEW FPGA模块对行为进行自定义。在VST这个例子中，如果NI-RFSG 和 NI-RFSA驱动程序的功能不能满足您的需求时，有两个软件堆栈选项可供选择。其一是使用仪器驱动FPGA扩展来执行常规和部分修正。另一个选项则是使用基于LabVIEW的开放软件堆栈完全自定义行为。

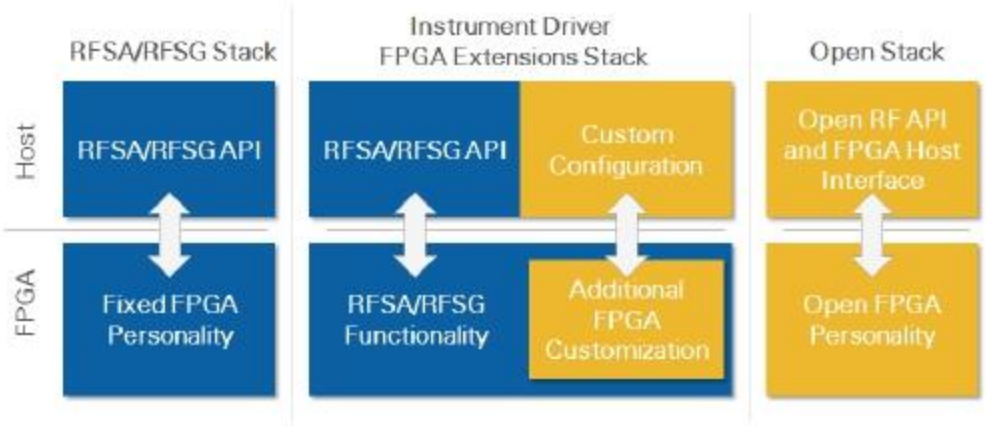

图57. VST可以与标准RFSA/RFSG仪器驱动、完全开放的软件堆栈或者仪器驱动的FPGA扩展来实现需要标准仪器驱动功能的常见自定义行为。

开放堆栈选项包含一系列采样项目，提供了用于实现常见应用程序的完整开放构架。堆栈的主机和FPGA端都可以在需要时进行修改。如果您想要从头开始定义主机和FPGA端时，可以采用这个方法，这使您可以设计完全自定义的仪器，利用设备的前端I/O功能，包括由NI的联盟伙伴Averna提供DOCSIS信道仿真仪。

仪器驱动的FPGA扩展

基于仪器驱动的FPGA扩展，您可以使用标准的仪器驱动，比如VST例子中的NI-RFSA和 NI-RFSG驱动，同时您还可以修改和扩展FPGA上的仪器功能。

堆栈的FPGA部分实现了标准仪器驱动预期的仪器功能，并清晰地标出您可以覆盖或者扩展特定功能的地方，比如触发、滤波和其他信号处理。

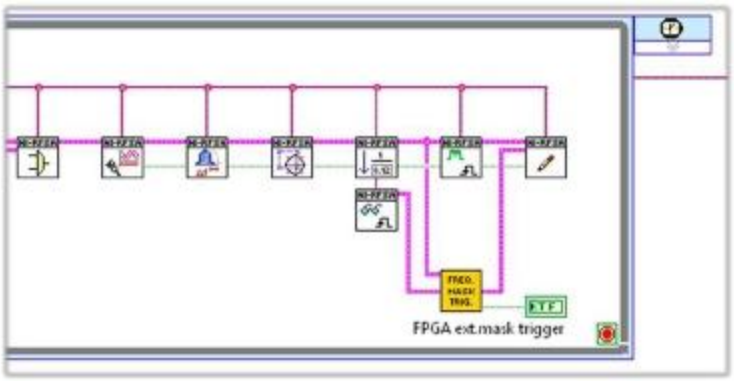

图55. 仪器驱动的FPGA扩展包含了与标准RFSA/RFSG仪器驱动API兼容的FPGA代码。同时，此方法也可帮助您识别和自定义FPGA的关键部分。

集成来自社区的IP

一些高性能LabVIEW FPGA IP的外部资源可以帮助您节省实现产品不具有的功能的时间。本部分着重讲述了一些最相关的IP源。

NI FlexRIO和软件设计仪器IP社区

该社区是一个针对软件设计仪器和NI FlexRIO设备的范例和IP中心库。从I2C到全通道仿真范例，您可以为这些高性能设备找到特定的协议和函数以及一群可帮助解答问题的热心人。

许多软件设计仪器的范例均是基于产品中的LabVIEW项目范例。这些范例利用了仪器驱动FPGA扩展的概念或者NI FlexRIO仪器开发资源库，比如IP构件块。

NI FlexRIO仪器开发资源库是LabVIEW主机和FPGA代码的集合，这些代码是设计来提供仪器里常用的FPGA功能，比如采集引擎、DRAM接口、触发逻辑以及相关的主机API。这个资源库是模块化的，所以您可以只选择您需要的组件。这个资源库也提供高效的实现方案，必要时您和可以修改提供的代码来满足您独特的应用需求。

IPNET 和LabVIEW工具网络

IPNet是社区和NI开发人员为LabVIEW FPGA专门创建的IP和范例集合。LabVIEW工具网络为不同的设备和应用程序提供了经认证的第三方插件，其中有些是为LabVIEW RIO平台特别开发的。

在创建LabVIEW FPGA IP之前，您应该先到IPNet和LabVIEW工具网络进行快速搜索。

更多资源

[1]点对点(P2P)数据流介绍
http://www.ni.com/white-paper/10801/en

[2] Xilinx Virtex-5 FPGA XtremeDSP设计考量
http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[3] Xilinx LogiCORE IP快速傅立叶变换数据表
http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf

[4]连接FPGA VI中的AXI IP
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/xilinxip_using/

[5] OpenCores
www.opecores.org

[6] DOCSIS信道仿真仪(DCE)—Averna
http://sine.ni.com/nips/cds/view/p/lang/en/nid/211379

[7]使用仪器驱动的FPGA扩展程序
http://www.ni.com/white-paper/14648/en/

[8] NI FlexRIO仪器开发资源库
https://decibel.ni.com/content/docs/DOC-15799

[9] NI LabVIEW数字滤波设计工具包
http://sine.ni.com/nips/cds/view/p/lang/en/nid/209040

[10]用于软件设计仪器和NI FlexRIO的范例和IP
https://decibel.ni.com/content/groups/software-designed-instrument-and-ni-flexrio-examples-and-ip

[11]仪器驱动的FPGA扩展介绍
http://www.ni.com/white-paper/14646/en/

[12] The LabVIEW FPGA IPNet
http://www.ni.com/ipnet/

[13] LabVIEW工具网络
http://www.ni.com/labview-tools-network/

此页特意留白。

定时优化技巧

应用程序通常需要指定感兴趣的事件之间的时间间隔。LabVIEW FPGA应用程序中，事件通常与I/O相关。在基于NI FPGA的设备中，由于处理是在硬件里进行，并且很靠近I/O，因而具有快速反应时间。本章介绍了多种技巧来帮助您更精确地控制和测量事件之间的定时。

延迟对于控制应用非常重要，在控制应用各种，系统的状态是由控制器逻辑进行采样和处理的，之后在一定的时间内生成控制输出。在控制应用中，我们通常希望获得更短的延迟。

数字协议应用也规定必须遵循定时约束条件，以实现与外部设备的正确通信。在这种情况下，时间之间的精确定时和可重复性是满足稳定和为正确的数据传输提供电子信号的要求所必需的。

通过SCTL确定和规定延迟

处理代码路径的总时间是它所有组件的延时之和。SCTL提供内置定时保证，因为每个迭代所需的时间刚好为一个时钟周期。跨多个SCTL迭代事件之间的延迟可通过以下公式计算出：

延迟=周期/时钟速率

这意味着您可以通过改变时钟速率或者感兴趣的事件之间的周期数来规定延迟。

SCTL时钟速率定义了定时分辨率。时钟速率越高，定时分辨率越高，但这样会使设计的编译更困难。时钟速率的增加量应该只需满足达到预期分辨率所需的量。

您可以通过在SCTL内的多个迭代之间扩散时间同时保持与所需分辨率对应的时钟速率来增加SCTL内事件之间的延迟。最基本的方法是当数据在SCTL内部传输时插入反馈节点，增加延迟周期。

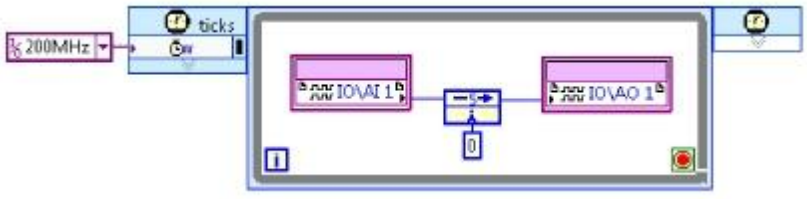

图56. 通过插入一个延迟值为五的反馈节点，我们可以实现在输入和输出之间额外25ns的延迟。

LabVIEW的状态机模式提供一种更高级的方法来控制事件之间的时间，因为每个状态在进入下一个状态前可以对经过的周期数进行跟踪。保证的迭代延迟提供一种方法来控制不同状态下事件之间的定时。

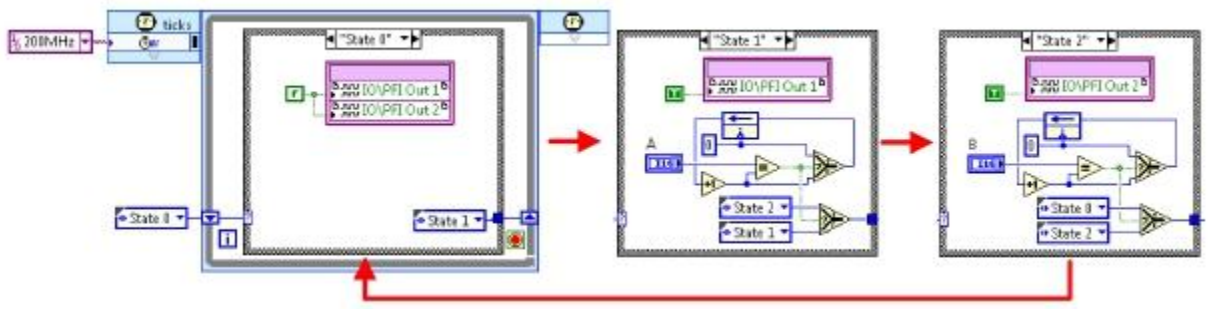

图57. 通过记录周期数，状态机模式使用SCTL控制数字输出事件之间的定时。第一个状态，状态0，使两条数字输出线无效，进入状态1。状态1使PFI Out 1生效，在进入状态2之前等待A个周期。状态2在返回状态0之前使PFI Out2有效B个周期。I/O事件之间的定时是由参数A和B决定的，200 MHz SCTL 时钟决定了分辨率为5ns。

要缩短事件之间的时间间隔较是困难，特别当事件已经涉及多个SCTL迭代，而且延迟也已经大于预期值。最佳路径取决于关系中周期数的量级。如果周期数已经比较小，那么减少几个周期可能对延迟有很显著的影响。如果周期数比较大，比如有成千上万个，那么您也很难减少足够多的周期来对延迟产生可测的影响。这种情况下，通过非流水线的方式增加时钟速率可能是最佳方法。此章剩下的部分主要讲述帮助您降低延迟的技巧。

通过并行化来降低延迟

正如吞吐量优化技巧（23）一章中所述，您可以利用FPGA高度并行的本质来执行同步操作，这以占用额外的资源为代价缩短整体延迟。并行化也可以缩短关键路径，这可以获得更高时钟速率，而且不需要流水线寄存器或者增加延迟的时钟周期。

并行化可能需要对代码进行较大的调整，根据算法的特性，对代码进行较大的调整可能非常困难或者不可能实现。不幸的是，我们没有一个保证算法并行化的方法。但是，您可以采用一些常见的并行化模式。

.

当数据是由独立的数据流（比如I/O通道）构成时，可通过复制程序框图上的代码来匹配独立元素或者通道的数量，从而轻松实现最大值、最小值和取平均等运算的并行化。

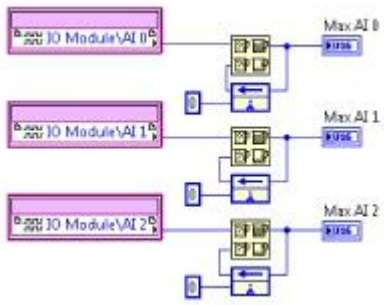

图58. 利用FPGA设备内在的并行性，您可以对独立数据元素执行多个同步操作。

如果运算可以对同一个数据集内的子集执行，您可以以同一比例多次复制此运算来降低延迟。例如，对图像数据进行阈值运算时，可以并行运算每个像素。复制代码n次可以将图像阈值运算的延迟降低n倍。

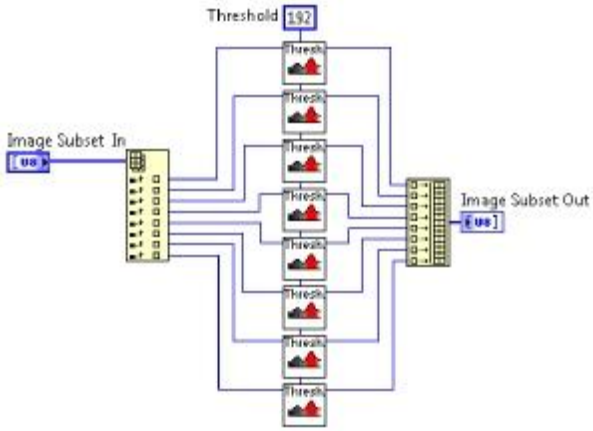

图59. 某些运算可以独立处理同一数据流多个部分。此代码将阈值运算同步应用到八个连续的图像像素，增加了吞吐量，降低延迟8倍。

即使没有明显的数据分离选项来进行复制操作，您也可以实现部分并行。当您使用具有相关性的运算来集合多个项目，比如乘法和加法，您可以创建树状结构来减少组合路径的长度。图60所示的范例展示了如何使用树状结构来更快速地添加数组中的元素。

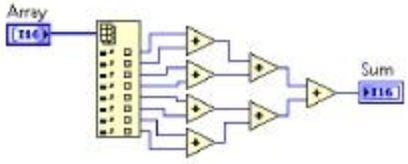

图60. 此树状结构提供了对数加速。处理n个元素所需的级数大约是log2 (n)。这比更常见的另一选项要好，即连续运算n对元素。

您可以用类似的模式类确定一组数值的最大值和最小值。如图61所示是两个数状结构的部分合并结果。

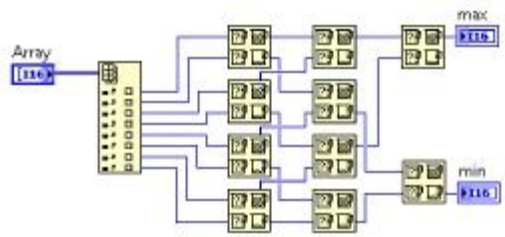

图61. 此图利用了一个理念，那就是多次使用同一逻辑函数(Max & Min)，选出两个数中较大的一个同时也指明了较小的那一个。双树状结构降低了运算延迟，降低倍数即为元素的数量。

对于运算会生成多个值的情况，您可以使用点阵结构进一步利用树状模式。比如当您想要用“泡泡”分类算法对数组元素进行分类时，如图62所示。

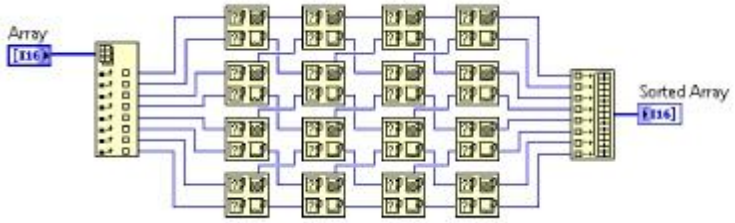

图62. 泡泡分类算法以点阵形式实现，利用了部分并行机制和运算结合性来计算一个时钟周期的固定大小。

此方法会比原来的在高性能单周期定时循环运算（15）一章中首次提到的数组分类算法快得多。原来的算法要求在数组的n个元素里有高达n个通过来分类数据。图62可运行一个SCTL周期，取决于时钟速率和数组大小。

删除流水线寄存器

正如吞吐量优化技巧（23）一章中所述，流水线方法会以增加延迟为代价来提高吞吐量。因此，您可以通过删除关键延迟路径上的流水线寄存器降低延迟。您也可以检查由于函数而隐式增加的流水线级数，比如高吞吐量FPGA选板上的那些函数。

优化数据类型

在资源优化技巧一章中讨论到推荐方法可能对减小传输延迟有明显的效果，这让您可以用更少的流水线级数来增加SCTL时钟速率，降低整体延迟。

资源优化技巧

尽管FPGA在性能和逻辑元素计数方面遵循摩尔定律，但是它们仍然在资源方面相对受到约束。用户必须认真管理FPGA资源，这是因为FPGA资源不仅会影响设计的适用性，而且也会对可实现的时钟速率和性能有着极大的影响。

FPGA资源类型

学习不同FPGA的芯片组件可以帮助您理解在设计日益复杂化时如何分配资源以及可能的进一步优化选择。FPGA主要是由两种组件构成的：可配置逻辑块或者片以及具有专用功能的资源，比如I/O或者内存块。

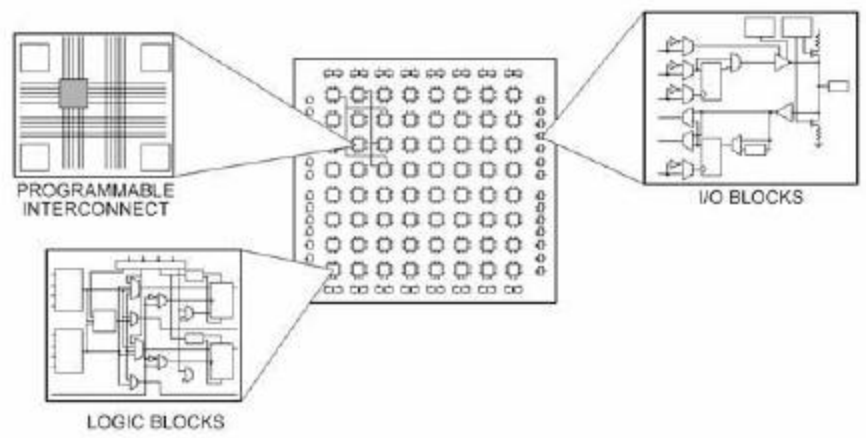

图63. FPGA资源可以分为逻辑块或片以及专用块，比如I/O或者内存块。这些块之间可通过编程互连路由信号。

逻辑资源对不同的片进行分组，创建出可以实现逻辑函数的可编程块。一片包含固定数量的查找表（LUT）、触发器和多路复用器。

- LUT是FPGA上硬连接的一组逻辑门。LUT保存了一个针对每种输入组合的预定义输出列表，提供一种快速的方法来检索逻辑运算的输出。
- 多路复用器，也被称为mux，是在两个或更多输入之间进行选择并输出选中输入的电路。
- 触发器是可以实现两种稳定状态的电路，代表一个比特。多个触发器可以组合成一个寄存器来存储比特模式，比如程序框图上的数值。FPGA上的寄存器包含一个时钟、输入数据、输出数据和一个启用信号端口。在每个时钟周期，输入值都被锁住并存储到内部，同时更新输出以匹配内部存储的数据。FPGA VI使用寄存器来实现许多LabVIEW构架，包括反馈节点和循环移位寄存器。

不同的FPGA产品系列具有不同的片和LUT。例如， Virtex-II FPGA的一片包含两个LUT和两个触发器，但是Virtex-5 FPGA的一片就包含四个LUT和四个触发器。LUT的输入数量通常是两到六个，取决于不同的FPGA产品系列。

专用FPGA资源用于执行非常具体的功能，比如DSP、时钟管理、更大型的存储元件（内存块）以及I/O。因为这些资源通常较为稀缺，根据FPGA产品系列的不同从几百到几千不等，所以必须谨慎管理这些资源。

块随机存取存储器，也叫做内存块，是嵌入到FPGA用于存储数据的。在合成存储器和FIFO时，LabVIEW尝试使用块RAM。数据传输机制一章会更详细地介绍存储器和FIFO项。您可以指定LabVIEW应该用于实现这些项和其他项的资源类型，以便在FPGA资源将被耗尽时手动平衡设计资源。

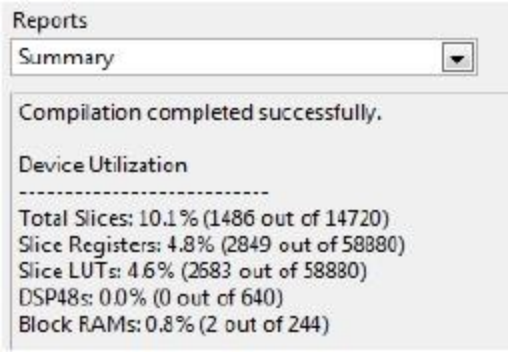

图64. LabVIEW FPGA编译结果会显示资源类型和此设计占用的资源比例。

填满FPGA

在设计中添加逻辑时，编译可能因为下列原因失败：

- FPGA耗尽了某种资源类型
- 编译器无法找到在组件之间创建路由的途径。
- 传输和逻辑延迟以请求的时钟速率阻止设计编译

FPGA每种资源类型的资源是有限的。当编译器尝试合成设计时，它可能先把一种资源用完。此外，逻辑资源是一起打包到片里的，如果某些部分是特别配置过的话，给定片的其余部分可能不能用，这使得完全利用某种特别资源类型很困难。您有时可以重新平衡不同类型的资源来使设计符合要求，正如本章稍候资源平衡一部分所述那样。

编译器的工作是将所有的组件放到FPGA芯片上。当进行小型设计时，编译器在放置资源方面更有效率。当配置满足定时要求，编译器的配置效率可能会低一些。

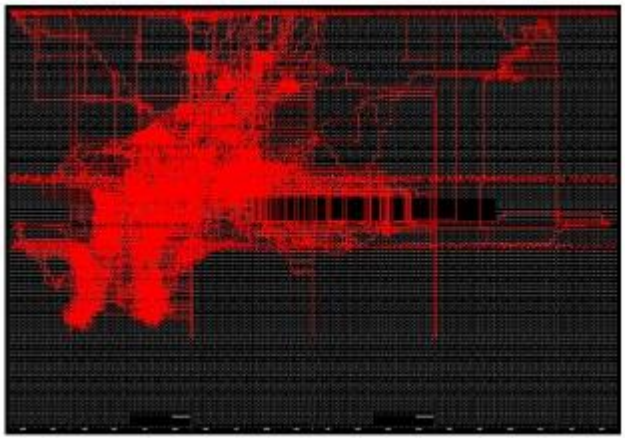

图65. 此图展示了一个FPGA设计的硬件布局范例。更小型的设计更容易合成，但当请求的定时约束条件满足时，资源配置的效率会降低。

随着逻辑密度增加，组件之间的距离会更远，导致路由变得更加困难，而且逻辑和传输延迟会增加。这导致了编译时间变长，失败率更高，甚至之前以同样的时钟速率成功实现编译的设计部分也可能会失败。

图66. 随着FPGA逐渐填满，放置组件和为设计组件找到更短的路由路径变得更加困难，这通常会导致定时失败。

资源优化可帮助编译器找到并改进设计不同部分之间的路由，使合成设计成为可能。通过缩短路由路径和降低路由延时，资源优化和平衡也可帮助您实现更高的时钟速率。下列部分提供了减少资源占用和平衡资源的建议，帮助您最大程度利用FPGA。

通过数据类型优化资源

标准类型优化

FPGA上用于代表值的比特数有时被称为数据类型宽度，在LabVIEW FPGA应用程序中，这是影响资源占用和性能的最大因素。更宽的数据类型要求更多的电路来维持和路由设计不同部分之间的值。更宽的数据类型还会快速消耗资源，使得同时满足用于在FPGA上传输值比特的不同路径的定时需求变得更加困难。

大多数LabVIEW FPGA函数需要占用的资源与输入和输出比特数成正比。在**加法**函数中，输入比特数越大，该函数需要的资源就越多。更多的资源占用导致更长的传输延迟和逻辑延迟，从而限制了整体最大时钟速率。

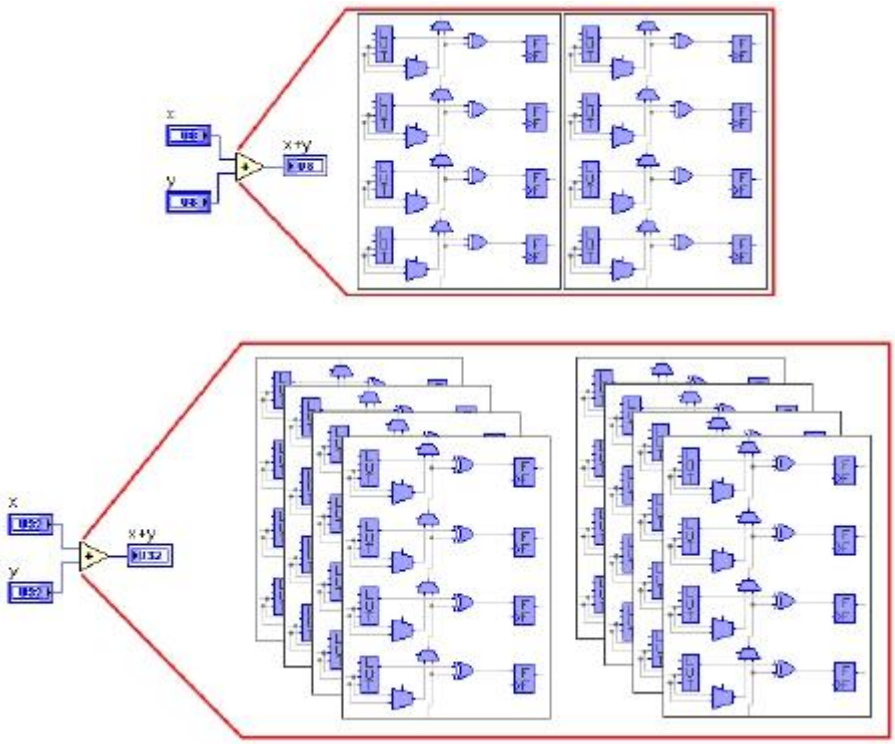

图67. 一个简单的8位加法函数可以用八个LUT以及一些一些针对溢出而设计的专用进位链逻辑来实现，每个LUT针对每组相加的比特。在某些FPGA设备系列中，LUT可能分布在两片上。一个32位的加法函数需要占用四倍的资源。

许多用LUT实现的函数遵循这个模型。例如，减法是用二进制的补码加法实现的。当输入增加宽度时，比较函数也需要更多的资源。

片上的LUT构成了FPGA的大多数可重配置逻辑，也可以将它们用于实现许多不同类型的运算。尽管逻辑片相对充足，但当FPGA填满时逻辑片之间的路由就变得更加困难。因此，实际上是不可能用尽设备上所有可用的片。

有些函数不能使用LUT单独实现。对数运算要求多个簿记资源，而乘法运算则需要使用DSP片。片和专用FPGA元件之间的比例取决于函数，并会影响FPGA资源布局。因此，当设计在FPGA芯片中实现时，可能会先用尽某些资源类型。

数据类型优化可以用于数据和任何设计中用来代表状态或者作为信息传递的值。例如，使用无符号8位整型(U8)来选择条件结构中的条件时，会出现256种情况。使用默认的LabVIEW类型I32可能产生超过您实际需要的比特，导致资源浪费。

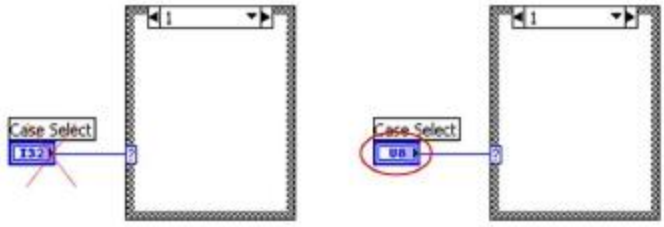

图68. 您可以根据设计每个部分的需求，选择最小的可代表不同数值的数据类型来节约资源。条件结构默认选择器的数据类型为I32类型，但是大多数的条件结构只包含一部分情况，用U8就足够了。

定点数据类型组合

定点数据类型具有浮点数据类型的部分灵活性，但是又保持整型算术的大小和速度优势。定点数字在用户指定的范围和精度内代表有理数。您可以为定点数值指定1到64位（含64位）之间的任何大小。您还可以将定点数配置为有符号或者无符号。

大多数支持定点类型的算术函数可根据输入宽度自动决定输出宽度。在决定函数输出宽度时，LabVIEW用一种保守的方法尝试阻止溢出和保持精度，即扩展用于代表整数和小数部分的比特数。尽管在数值方面是安全的，但这会导致下行类型宽度迅速增加，进而增加资源占用。

如果您知道流经运算代码的数据范围和精度，您可以通过手动指定函数沿处理路径的输出宽度来控制定点宽度的增加。这可能是一个枯燥乏味且需要需要仔细验证的过程。NI推荐通过使用一整套单元测试来设计算法并验证其功能。一旦基本的算法支持较长宽度的输入值，可通过调整函数输出宽度和使用一组功能测试来验证数值行为来具体指定宽度。

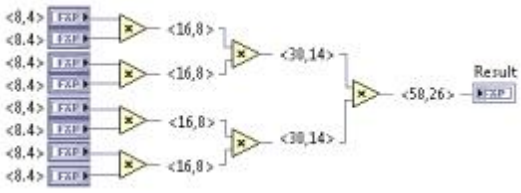

图69. 定点输出宽度可以快速增加，取决于运算的类型。在这个例子中，LabVIEW自动传输数据宽度并把数据宽度从一组8位的操作数扩展为一个58位的结果。您可以通过手动重写函数输出宽度来控制宽度的增加。

优化数组和簇

数组运算占用的资源和数组的大小成正比。例如，当数组以一定标量值增长时，该数组中每个元素都对加法运算进行实例化。簇和更大的数据类型也适用同样的原则。通过最小化数组大小和簇类型，您可以节约FPGA资源，提高性能。

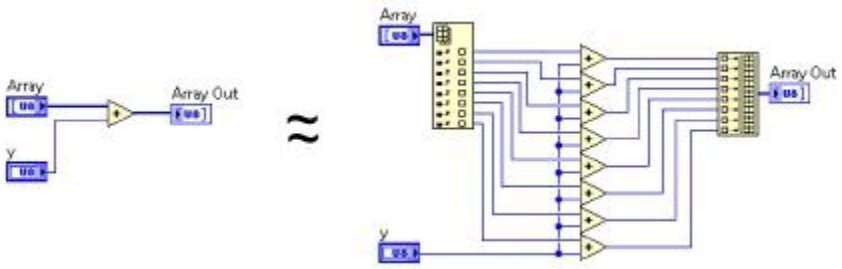

图70.通过复制数组里的每个元素的运算来实现数组运算。这会导致FPGA资源占用显著增加。

在编译时，有些运算只解决FPGA的路由问题。数组索引使用一个经编译器优化的常数值作为储存元素和消耗存储值的逻辑之间的路径。数组里的其他元素如果不需要使用，可能会被编译器删除。

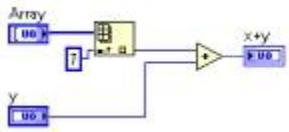

图71. 此范例在数组输入的索引7加入一个数。没有片逻辑与检索索引7相关。逻辑仅要求把索引7加到y。

把输入控件连接到索引数组函数需要额外的逻辑来路由选中数组元素。即使索引的值在运行时保持不变，编译器必须插入电路来处理索引值在运行时会变化的情况。

最小化前面板使用的输入控件和显示控件的数量

在优化FPGA资源占用时，您应该最小化在顶层FPGA VI前面板上创建的输入控件和显示控件的数量。正如数据传输机制一章中检验的那样，顶层FPGA VI上每个前面板对象都用作一个寄存器项，用于FPGA和主机系统之间的通信，需要存储和地址逻辑。因此您应该尝试去掉任何不使用的前面板对象，或者用图常量来替换这些对象。

创建一个大型数组或者簇作为前面板输入或者显示控件会非常耗费资源，应该尽量避免。如果您需要传输主机的十几个数组元素或将十几个数组元素传输至主机，正如数据传输机制一章中阐释的那样，使用FPGA主机界面FIFO。如果前面板对象是作为VI内部传输数据的途径，您可以用全局变量替换它们以节省资源。

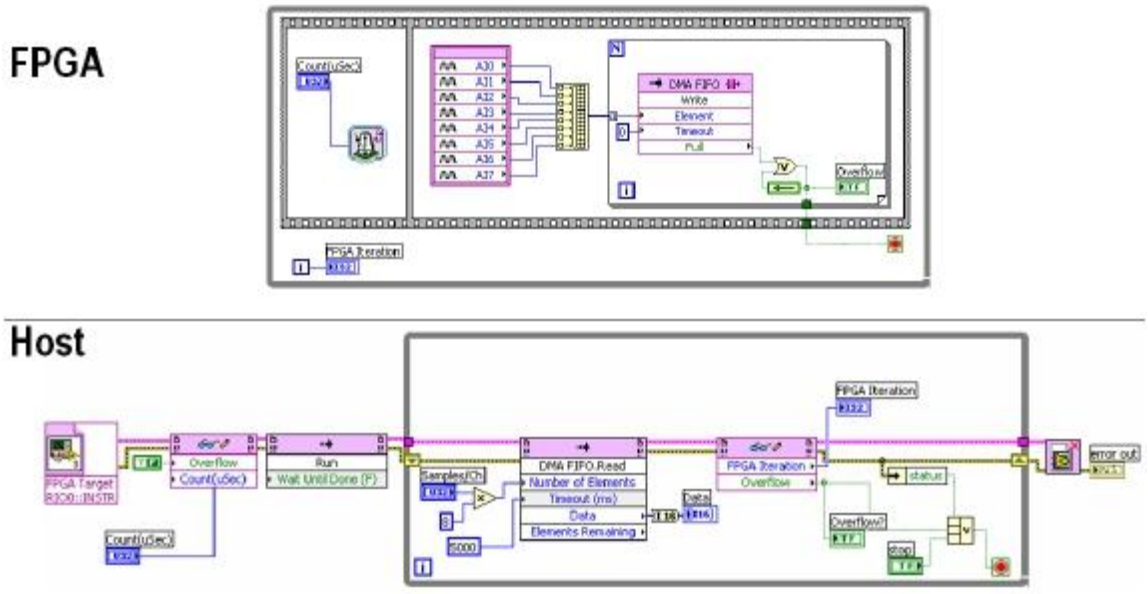

图72. 此例展示了使用Host Interface DMA FIFO传输模拟采样数据数组到主机。

调节输出溢出和舍入选项

溢出选项

正如集成高吞吐量IP一章所述，如果输出端的数据类型不够大，无法容纳运算结果，就会发生溢出。通常情况下我们不希望溢出，因此LabVIEW会改变输出类型宽度来避免溢出。但是，您可以使用算法及算法将处理的数据来使用较小的输出类型进行运算，降低FPGA资源占用。当您手动指定输出类型时，您可以配置大多数函数，使用Saturate 或者 Wrap 模式来处理溢出。

Saturate 模式决定了值是否在输出类型的范围之内，并强制将值调整到合适的限制范围。这需要电路来检测溢出条件，并且为输出挑选出合适的溢出值。

Wrap 模式就是舍弃有效位直到值处于输出范围内。这种模式要求较少的FPGA资源，但是在溢出条件发生时，它会导致明显数值差异。

当您为定点算术运算手动指定输出类型时，为了满足数值的准确性，LabVIEW默认为Saturate溢出模式。集成高吞吐量IP一章所讨论的高吞吐量数学函数默认使用Wrap溢出模式以提高性能和资源效率。您可以调整每个函数的溢出模式来实现资源占用和数值精度之间的最佳平衡。

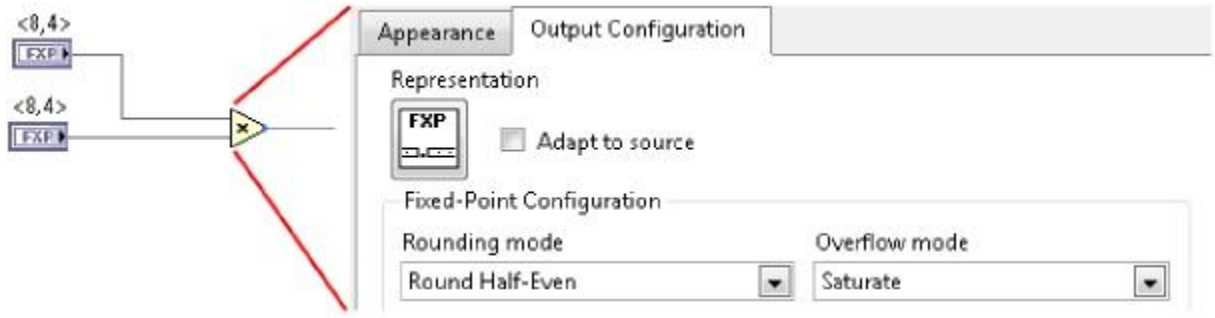

图73. 当您手动指定输出类型宽度时，标准定点算术函数输出端会出现蓝色的点，默认为Saturate溢出模式和Round-Half-Even舍入模式，以提高数值准确性。

舍入选项

舍入发生在值的预期精度大于它代表的数据类型精度时。LabVIEW自动挑选输出宽度来保持数值的精确度。不准确的操作符，比如除和平方根，总是需要某种程度的舍入。使用Truncate， Round-Half-Up或者Round-Half-Even模式，您可以通过配置函数来实现舍入。

Truncate 模式仅仅是删除最低有效位，因此不需要任何FPGA资源。但是，对大多数数据流，此模式产生了最大的平均误差，因为它的斜率趋向于零。

Round-Half-Up模式将值舍入至数据类型可以代表的最接近值。如果要舍入的值正好在两个有效值之间，那么这种模式会取两个值中较高的那个值。LabVIEW为输出值加一位，然后再进行截断。与Truncate模式相比，这种舍入模式可生成更准确的输出值，对更大的值有更小的偏差。但是，Round-Half-Up也对性能和资源有更大的影响，因为它需要与输出类型的比特宽度成正比的额外逻辑。

Round-Half-Even模式将值舍入至数据类型可以代表的最接近值。如果要舍入的值正好在两个有效值之间，那么LabVIEW会检查值在舍入后的最低有效位。如果该位为0，则此模式会取输出类型代表的两个值中较小的那个值。如果该位非0，则此模式取两个值中较大的那个值。相较其他两种模式，Round-Half-Even模式产生更精确的输出值，但是它需要最多的FPGA资源，并且对性能有最消极的影响。

图74. 当您指定高吞吐量数学函数的输出类型时，它们默认处于Wrap溢出模式和Truncate舍入模式，以保证性能和资源效率。

当您指定函数输出类型宽度时，为了保证数值准确性，LabVIEW默认标准定点算术运算的模式为Round-Half-Even模式。正如集成高吞吐量IP一章讨论到的，高吞吐量数学函数默认处于Truncate模式，以保证性能和资源效率。您可以调整每个函数的输出宽度和舍入模式来实现资源占用和数值精准之间的最佳平衡。

初始化反馈节点

移位寄存器在进入每个循环结构时初始化。像移位寄存器那样，反馈节点的初始值可以配置，反馈节点的初始化方式也会影响性能和资源占用。

没有连线的反馈节点默认在Compile or Load时初始化。FPGA比特文件加载到FPGA后，Compile or Load分配会默认值给数据类型作为反馈节点的初始化值。如果在反馈节点的初始化端连接一个值，则会将初始化模式切换到First Call.。

如果您希望每次运行FPGA VI时，反馈节点都重置到初始值，则可选择First Call.选项。First Call.初始化使用了多路复用器在初始值和当前值之间进行选择，导致稍微较高的资源占用。大多数应用加载FPGA比特文件，每个设备电源周期运行一次。所以，用Compile or Load初始化是更好的选择。如果您的主机应用每次运行都加载FPGA比特文件，或者重复使用前一次执行的值是可以接受的，您就可以使用Compile or Load选项，而无需使用多路复用器，进而稍微提高了该代码路径的可实现时钟速率。

图75. 由于反馈节点集可以通过Compile or Load初始化，您可以使用更少的资源实现更快的编译速度。

在使用First Call初始化模式时，您可以选择把多路复用器添加到寄存器之前或者之后。这是一个高级优化选项，如果在将关键路径分成均等的多个延迟部分以获得更快的编译速率遇到困难时，则可使用该选项。把多路复用器移动到寄存器任何一侧都会使传输延迟和逻辑延迟从一个部分移位到另一个部分，这使得在分布路径延迟时允许更细的粒度。

资源平衡

在您可能用尽某种资源类型的情况，资源平衡可以帮助您解决编译失败的问题。通过指定或者促使设计某些部分使用某种资源类型，您可以更有效率地占用FPGA资源。再平衡机制也可帮助您以更快的速率运行您的设计，因为不同的资源类型以不同的时钟速率运行，并且在不同资源类型之间需要不同数量的路由逻辑。

反馈节点、存储器、FIFO和寄存器项也会显著占用资源，但是正如数据传输机制一章所述，通常它们会用于易失性存储和通信任务。

反馈节点

移位寄存器和反馈节点用多个触发器来实现的，除非编译器可以将一或多个触发器优化到一个LUT中。如果您的代码不依赖反馈节点的初始值来正常运行，您可以通过反馈节点Properties 对话框的FPGA Implementation页面选择Ignore FPGA reset method选项节省额外的资源。这允许编译器删除寄存器的重置逻辑，并用移位寄存器查找表(SRL)取代触发器来实现。SRL可以将多个延迟结合到一个LUT，这可以显著降低与触发器相关的FPGA资源占用。

分布式延迟函数与反馈节点的行为类似，但是它使用了SRL来实现多个周期的延迟。离散延迟函数在运行时支持不同的延迟量，对于标量和整型、定点和布尔型数据类型数组、簇和数组簇，最多支持预先设置的最大延迟量。

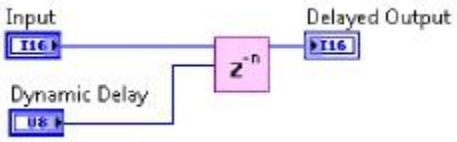

图76. 离散延迟块独立运算布尔型数据，但是它们是使用SRL实现的，所以您可以用它们替代反馈节点以平衡资源占用。

FPGA FIFOS

FPGA FIFO可以用作存储器或者作为在FPGA的循环间传递数据的方法。通过在触发器、LUT或者内存块(BRAM)之间进行选择，您可以指定用于平衡FPGA上资源的FIFO的资源类型。

触发器具有最快性能，但是您应该只将它们用于小型FIFO，大小最多100字节，因为它们会消耗每个存储元件相当多的FPGA逻辑。

内存块FIFO使用分布在FPGA芯片上的专用内存块来存储数据。对于300字节到数千字节范围内的FIFO可使用这个选项，FIFO的大小取决于FPGA设备。正如数据传输机制一章所述，内存块FIFO是可以跨时钟域传输数据的唯一FIFO。与其他两种实现方式相比，它们也存在较长的延迟。

在使用内存块FIFO时，您可以为FIFO控制逻辑指定实现资源。较新的FPGA配有内置的FIFO控制器，可以显著节省FPGA资源，与使用片构造的控制逻辑相比，可支持更高的时钟速率。

数组常数

您可以通过指定LabVIEW如何实现数组常数来优化FPGA应用程序。您应该考虑选择内存块来实现数组常数，除非您需要其他内存类型的优势，或者需要为其他任务释放内存块。内存块不会消耗FPGA逻辑资源，而且相对于其他类型的内存，会以高时钟速率编译。

默认情况下，LabVIEW会基于具体的代码模式来自动决定使用内存块、LUT、还是触发器来实现数组常数。存储在反馈节点的数组可能使用了内存块作为实现资源，可以看作RAM。使用内存块时，只有一个读取程序和一个写入程序可以用SCTL内的标准数组函数访问数组。访问内存块需要占用一整个时钟周期，所以反馈节点必须正好放在每次访问后。

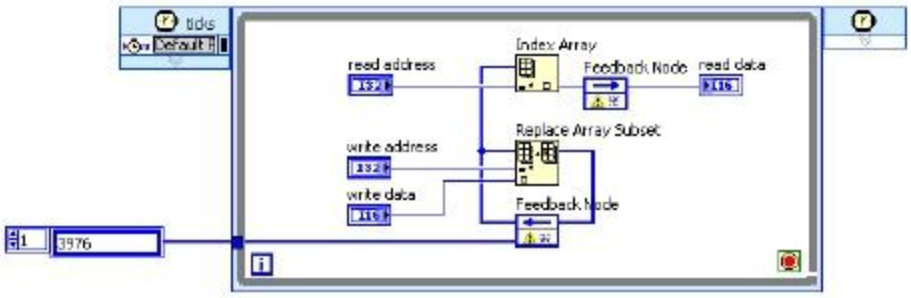

图77. 此数组常数作为内存块实现，并存储在反馈节点中。这允许在一个SCTL周期内使用标准数组函数来进行读写。

如果只需要读取访问，在内存块作为数组实现资源时，您可以使用一个索引数组函数和反馈节点把数组立即放到SCTL内部。

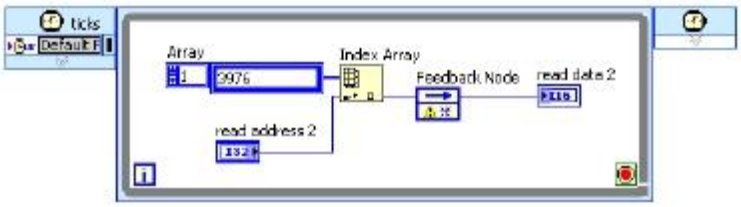

图78. 只读数组也可以使用上述模式在内存块上实现。

双端口访问内存项

在Memory Properties对话框的Interfaces页面将内存配置为双端口读取访问，可以降低内存块资源占用，减少执行时间。

通过两个读取端口，您可以同步访问两个不同地址的数据。您可以放置两个内存方法节点来读取一个结构内的双端口内存，比如SCTL或者FPGA VI的不同地方。

多路复用逻辑

逻辑复用是另一种优化资源的方式。您可以通过以更高的时钟速率运行处理代码以及之后对其进行多路复用访问，使用同一电路来处理一段时间内独立的数据流。一种在SCTL内实现这一目的的方法是将处理逻辑移动到一个独立的SCTL，这个 SCTL的时钟速率需要为原SCTL时钟速率的倍数。您必须在这些时钟域使用内存块FIFO来传输数据，然后使用一组通用块来进行采样处理。

因为簿记和通信开销，代码多路复用可能仅仅适用于大型运算任务。通信开销指的是在两个循环之间移动采样数据时产生的延迟、资源和可能税率。簿记开销指的是必须添加以跟踪哪些采样数据需要在给定时间进行处理的逻辑。处理代码取决于不同时间的状态，由于这些代码会在独立的数据流之间交替，因此必须保存和检索相应的状态信息。

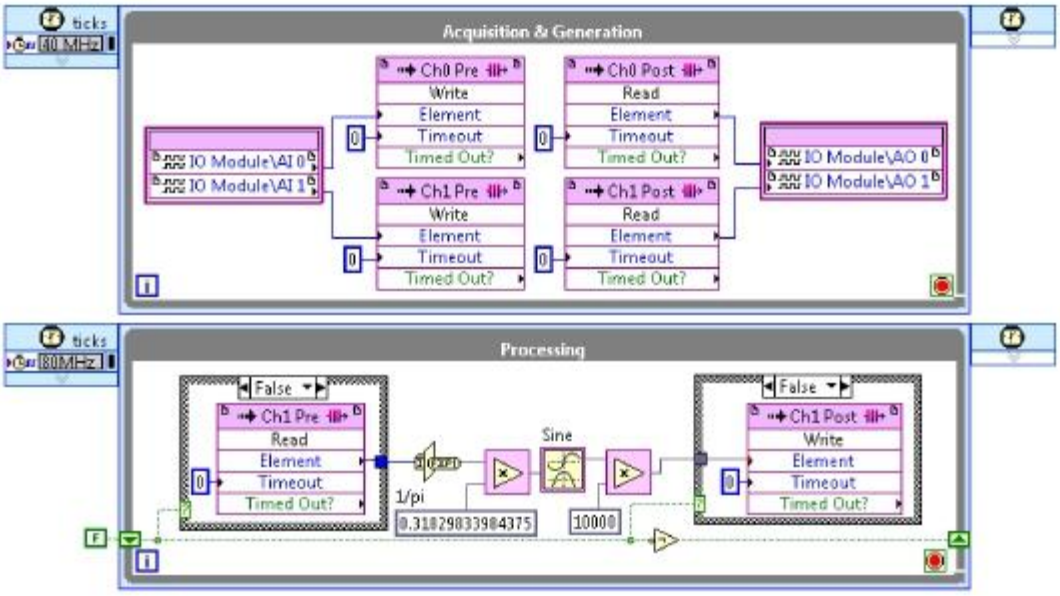

图79. 运算代码一段时间内多路复用于两个独立的数据通道。采集循环读取I/O模块，把两个样本通过专用FIFO传输到运算循环。运算循环以采集循环两倍的速度运行，交替读取每个FIFO，进行运算，然后把结果写会到相应的返回FIFO。这种方法以部分通信和簿记开销为代价将处理占用的资源减半。

DSP 运算非常适用于多路复用，因为与数字逻辑任务相比，它们通常消耗更多的资源。如果您需要耗尽FPGA空间的话，一些代价昂贵的运算，比如除、平方根以及其他复杂IP，也是多路复用不错选择。

当您使用Xilinx 内核生成器IP或者使用LabVIEW FPGA IP生成器附件创建您自己的IP时，浮点操作也可能在SCTL内。单精度浮点操作可能显著占用资源，但是可以通过动态类型范围来简化某些算法的实现。因此，单精度浮点运算也是多路复用的不错选择。

在SCTL之外运行时，您可以将子VI设置为非重入来实现多路复用，并在程序框图内对其进行多次复制。LabVIEW会对复制的每个子VI进行访问仲裁，从而实现序列化的执行，提高资源利用效率。对于许多控制和测量应用，FPGA比I/O快速得多，所以序列化访问通常能够满足吞吐量和延迟需求，同时节省FPGA资源。

使用SCTL节省资源

即使您现存的FPGA应用程序不包含SCTL，您也可以考虑使用SCTL来优化资源。LabVIEW会自动优化SCTL内部的代码，与While循环内部的相同代码相比，这种执行更快速，消耗更少的FPGA空间。

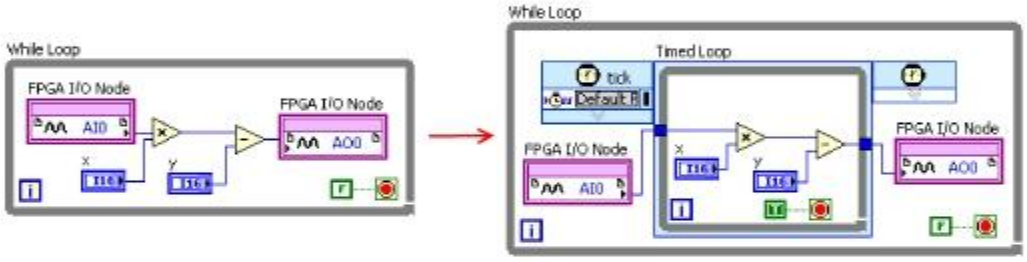

图80. 您可以在一个While循环内使用SCTL。将一个常数连接到SCTL停止条件或者使用内部的四线握手IP Output Valid信号来停止循环，将结果向下传递到其他非SCTL代码。

更多资源

[1]使用NI LabVIEW FPGA IP Builder优化并将VI用于FPGA
http://www.ni.com/white-paper/14036/en/

[2]优化数组常数的内存占用
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_array_memory_implement/

此页特意留白。

数据传输机制

数据传输是高性能LabVIEW FPGA应用程序用于实现系统内并行进程的一个重要方面。这些并行进程以一个FPGA、多个FPGA或者主机系统上并行循环的形式出现。从系统的角度总体来看，通信可以发生在FPGA内、FPGA和主机之间、FPGA设备之间或者FPGA设备和其他仪器之间。

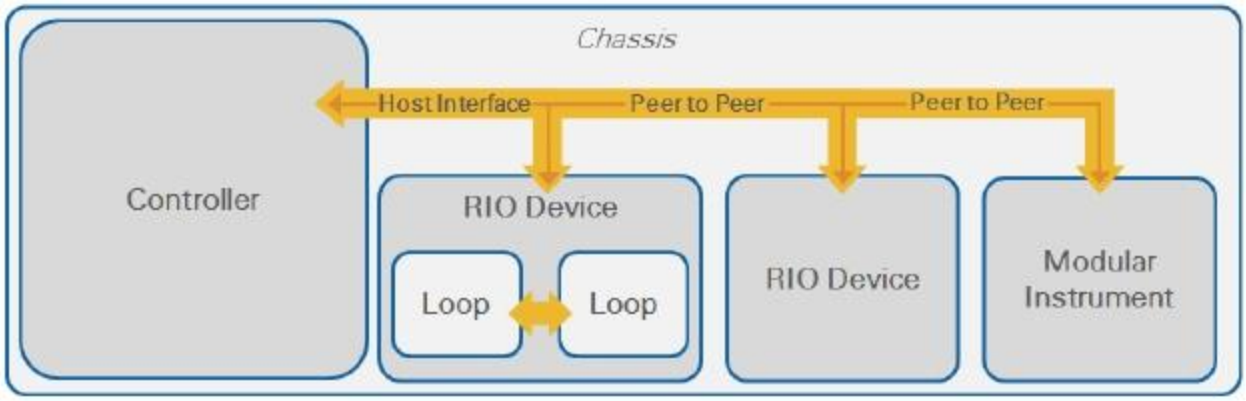

图81. 数据传输可能在FPGA内部、FPGA设备之间、FPGA设备和其他设备之间或者FPGA设备和主机之间发生。

这一章总结了不同数据传输机制及其相关性能特点。

数据传输机制的吞吐量和延迟

当应用涉及多个设备或者一个设备上的多个进程时，数据传输机制的效率影响应用的性能。这个部分分析了数据传输机制性能相关的概念，此机制会在接下来几个部分中讨论。

您可以模拟任何数据传输机制，作为两个进程（一个源，一个汇）之间的交换，其中数据从源通过某个通道流向汇。

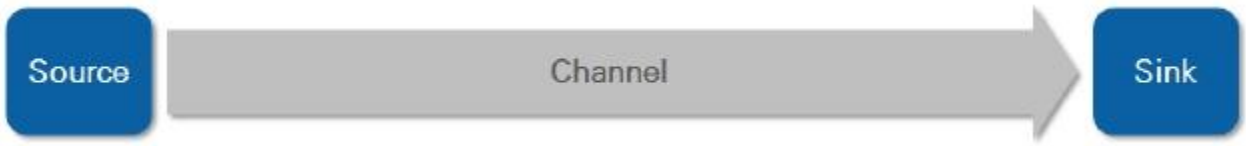

图82. 数据传输可以用两个节点交换数据通过一个通道来模拟。源产生数据，汇消耗数据。

源、汇以及通道决定了数据传输机制吞吐的有效性。源决定每个时间单位可以提供多少数据。通道必须有足够的带宽让数据流过。在数据达到时，汇必须能够存储或者处理数据。

带宽是指给定通道配置下最大可实现的吞吐量。带宽和吞吐量不同：吞吐量指的是该通道实际传输的数据量。您可能需要使用特别传输模式，或者写入特定大小的数据来实现接近通道带宽的吞吐率。

数据源可能间歇地突发地传输数据。如果突发模式是可预期的，您可以以通道用于传输数据的时间百分比来定义其占空比。在传输初始化前可以定义最小数据量，也就是用通道或源定义最小传输大小。

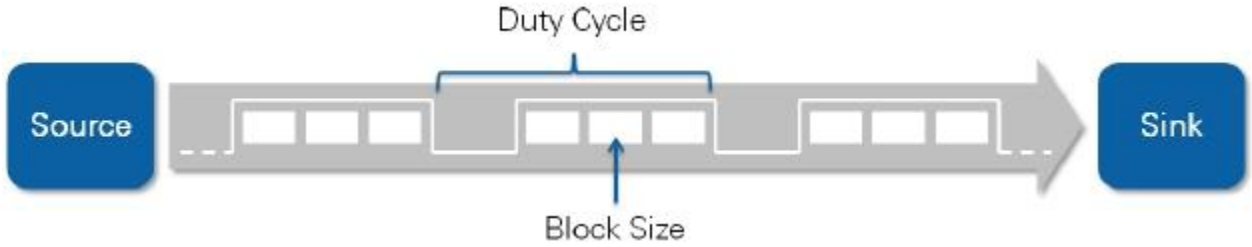

图83. 突发数据传输可能由一到多个块组成。如果突发是统一的、周期性的，通道有效吞吐量就是通道带宽和占空比的乘积。

突发数据、不可预测传输模式以及其他瞬时通道或源的条件，比如总线冲突或者抖动，会导致峰值和平均吞吐量。平均吞吐量是以无限时间内的可持续吞吐量来定义的。如果源、汇或者通道不能处理预期的平均吞吐量，那么传输最终会失败。峰值吞吐量可能有时会超过平均吞吐量，但是不会超过通道带宽。

只有在汇使用缓存来临时存储和批处理数据，以峰值或平均吞吐率接收数据时，数据传输才会成功。

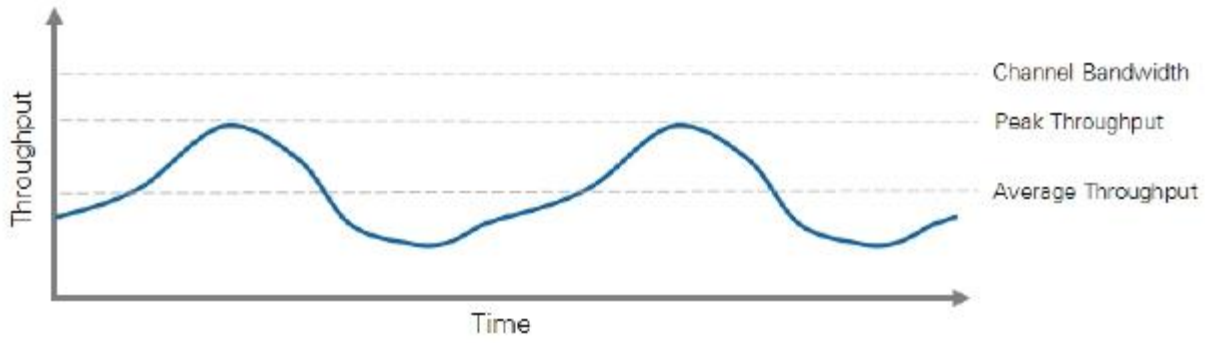

图84. 有效吞吐量可能随着时间变化，但是不可能超过通道带宽。源和汇必须处理平均吞吐量来保持无限传输。源和汇也必须能够临时处理峰值吞吐量的情况，通常是通过缓存。

缓存可以吸收预期的突发数据和出乎预料的瞬时情况，这样就有助于维持理想的平均吞吐量。较大的缓存会以增加资源占用和更高的延迟为代价吸收更多的瞬时事件。

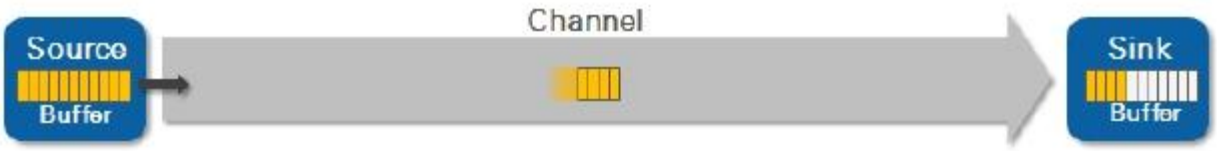

图85. 缓存通过将数据临时存储到源或者汇来帮助处理瞬时情况。

数据传输延迟与传输途中的数据量以及通信通道延迟成正比。FPGA内部通信和本地通信总线，比如PCI Express，通常具有较快速度，但是在给定时间内无法保存太多数据。因此，大多数的延迟来自源和汇缓存区的数据。可以通过将预期的传输数据量乘以通信机制的平均吞吐量来估计延迟。

FPGA内部传输数据

多循环和多速率设计

我们常常将高性能LabVIEW应用程序分布在框图里的多个循环中。下列是多循环分布的一些考虑因素：

任务分离——一个好的编程做法是尽可能地将逻辑上分离的任务分配到独立的循环中，这可以使代码维护更容易和提高可读性。如果用此方法构建，循环就扮演了独立进程的角色，利用FPGA的并行性实现并行操作。多个运算循环可能在同一个数据流里运行，所以理解在它们之间的传递数据对性能的影响是很重要的。

强制的时钟域——某些FPGA结构只能用于特定的时钟域，这强制应用程序的某些部分也只能处于特定的时钟域。一个典型的例子是在应用程序与FPGA外部的某些组件，比如I/O或者DRAW交互。从高速输入源读取数据并将传数据到DRAM的应用程序必须至少包含两个循环：一个为输入；另一个用于写入DRAM。这样就要求循环之间具有通信机制。

时钟域优化——在特定I/O构架强制您使用特定时钟域时，您需要知道剩余处理任务可能处于其他时钟域。例如，对数据流取平均值或取十分之一时，您可以采用一个吞吐量比较低的数据流，这意味着您可能能够以较低速率为下行函数进行时钟定时。较低的速率有助于降低编译时间和资源占用，同时也增加了编译成功的几率。

资源优化技巧（59）一章介绍了该方法的一个范例，在该范例中，处理函数在一个独立的循环中运行，可通过共享来节省资源。

不同循环类型——最终，在您的FPGA设计中，您可能需要不同循环类型。并不是应用程序的每个部分都需要SCTL的性能优势。因此，您可能需要将应用程序分到多个循环来避免对SCTL具有相对更严格的要求。

将应用程序分到多个循环中可能会带来数据通信挑战。SCTL通信包含的逻辑如果采用不同的时钟时，就会以不同的时钟域执行。当SCTL定义了不同的时钟域时，如果您在选择和配置通信机制时不够仔细，可能会发生数据丢失或者覆盖。

不同通信机制提供不同的同步度和缓存策略来维持数据一致性，通常以牺牲性能或者资源占用为代价。额外的同步机制，比如数据标签和同步变量，可以用于保证循环之间的执行顺序，并且在必要时增强通信机制。

循环间通信机制

局部和全局变量

变量把信息从一个循环传输到另一个循环。变量并不需要缓存，所以仅在需要最新写入的值时变量才有用。在SCTL内部使用的变量只能限制用于一个写入程序，但允许用于多个读取程序。

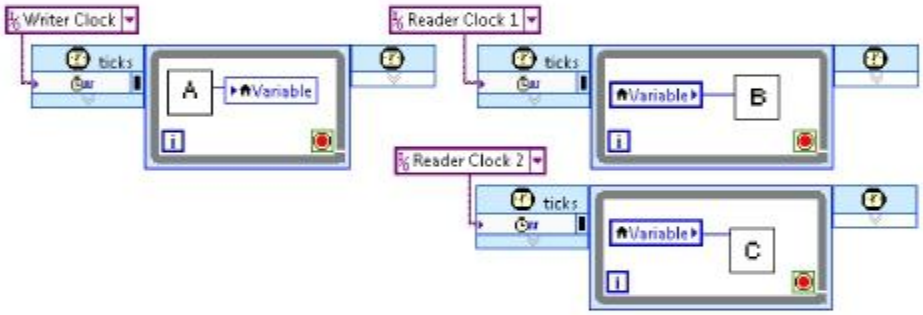

图86. 变量用于存储和发布跨循环的值。

变量访问范围根据变量类型的不同而不同。局部变量只能被定义这些变量的VI访问。在FPGA上运行的所有VI都可以访问全局变量。局部变量以输入控件或者显示控件的形式出现在前面板。因此，正如资源优化技巧一章讨论的那样，如果您需要节省资源，全局变量或者寄存器是更好的选择。

数据一致性是通过给定变量所有的位来保证。换言之，写入值被接收程序读取时，所有位都是一致的。多个变量之间的一致性不一定能保证，因此对于多个变量的值非常重要的情况，您必须将其绑定作为簇或者叠加同步机制的一部分。正如错误！找不到引用源一章所述，如果您需要使用一个变量而且需要同步写入程序和读取程序时，可考虑使用握手项。

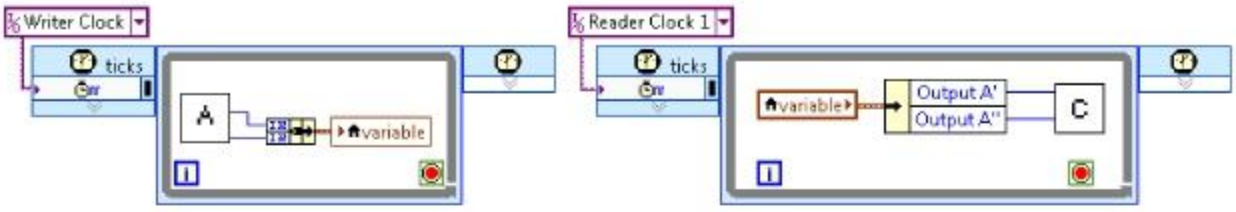

图87. 此范例使用簇变量保证跨循环共享多个值的一致性。

关于延迟，变量可能需要几个周期才能将其值从一个写入程序传递到任意读取程序。周期数根据读取程序不同而有所不同。这意味着如果您需要多个写入程序在给定时间共享同一变量值，您必须采取适当的步骤，可采用额外同步或者标签的形式。此外，您可能不知道在循环之间传输值所需耗费的时间。因此，您需要告诉代码或者提供额外的同步机制来检测数据的写入或读取时间。

由于变量并不提供内置缓存或者同步机制，因此它们通常不包括数据流。因此，它们的吞吐量特性并不像其延迟具有相关性。

寄存器项

寄存器项与变量在功能、资源占用和性能方面都相似。与变量不同的是，您可以通过使用VI-Defined Register Configuration（VI定义的寄存器配置）节点来获取寄存器的引用。通过将引用连线传递到子VI，寄存器项引用可用于编写可复用代码。这些寄存器项的引用在编译时进行解析。

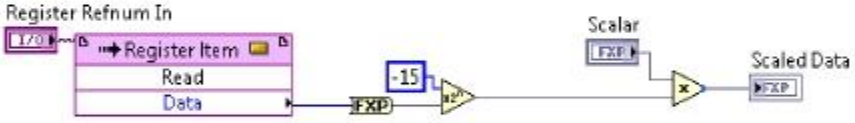

图88. 您可以使用寄存器引用连线编写用于处理共享数据的可复用子VI。该子VI可以用于整个设计，但是可从不同的寄存器读取数据，取决于连接到调用程序VI的引用值。

FIFO项

通过先进先出访问策略，FIFO用于循环间高吞吐量无损传输。FIFO可缓存数据，可用于当源或者汇可能生成或者接受突发数据的情况。

FIFO可由项目或者VI定的，具有相似的吞吐量和延迟特性。项目定义的FIFO用于传输不同VI的循环之间的数据，但是需要LabVIEW项目在编译时提供全局FIFO定义和配置参数。

VI定义的FIFO是直接在程序框图上使用VI-Defined FIFO Configuration（VI定义的FIFO配置）节点进行定义。每次在程序框图中实例化母VI时，此节点就会实例化FIFO。这个配置节点还提供了一个引用，可以用来编写可复用代码和在VI之间传输数据。

FIFO提供一个 Time Out?端，用于指示读或者写方法的成功。读取数据时出现超时则表明FIFO是空的且Element的值未定义，因此该值不应该用于下行节点。该信号等同于四线协议中的output valid否定值，详情可参考集成高吞吐量IP一章（33页）。

图89. SCTL内部的FIFO读取操作必须使用值为0的超时。输出为Time Out?则表明这是一个空FIFO。

写入数据时出现超时则表明FIFO满了且此元素不可写。如果应用程序不允许数据丢失，则应该添加逻辑来缓存要写入的值，然后重试写入操作。

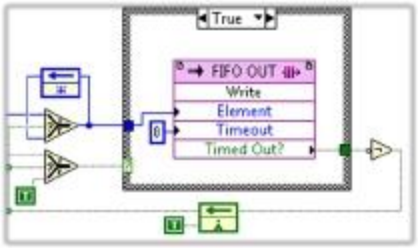

图90. 额外的逻辑必须用于处理将数据写入FIFO导致超时的情况。使用反馈节点可以缓存要写入的值，阻止下行代码产生额外值直至FIFO有空间，写入操作才算成功。

FIFO提供了一种查询在给定时间可读写元素的数量的方法。这些方法增加了可跟踪并显示FIFO元素计数的电路。额外电路会降低SCTL可以达到的时钟速率。同样，这些方法也可用于FIFO与生成突发数据的IP交互的情况。如果预期的突发数据大小未知，建议使用Timed Out?端，根据需要允许握手发生。

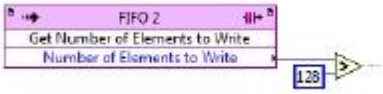

图91. Get Number of Elements to Write方法通常用于阻止突发IP执行，直到FIFO可以接受预期的突发输出数据。

正如资源优化技巧一章所论及，您可以为项目和VI定义的FIFO指定资源实现类型。

在共享相同时钟域的SCTL之间传递数据时，采用触发器实现方式可提供最佳性能，但却会使用较多数量的片，因此该方法应该用于小型FIFO（最大100字节左右）。基于LUT的FIFO提供了折中的FIFO选项，大小为100到300字节。

内存块是唯一一种支持跨时钟域传输数据的FIFO。换言之，内存块FIFO支持以不同的速率或者使用源自不同时基的时钟跨程序框图传输数据。您可以仅从一个时钟域把数据写入内存块FIFO，然后再从另一个时钟域读取数据。

在从同一时钟域的SCTL读写FIFO时，LabVIEW使用一种更简便的同步实现方式，可缩短延迟以及提高时钟速率。

在实现相对深的FIFO（大于300字节）时可使用内存块FIFO。在写入的数据到达读取程序之前，内存块FIFO可以有高达六个时钟周期的延迟。

握手项

握手项是使用握手来实现单个写入程序和单个读取程序之间无损传输的非缓存机制。握手项与FIFO相似，仅包含一个元素，但是握手机制可绕开阻止FIFO以这种方式进行配置的实现细节。握手项只能用在SCTL内部，可以用于跨时钟域通信。

与FIFO相似，握手项提供了内容清理的选项。它们也可以提供一种实现非破坏性的读取方式，这时读取程序读取握手项的内容，而不清空内容，或者告知写入程序数据已读，然后发出一个明确的确认信息，这样写入程序就能提供一个新的值。这在必须同步读取程序和写入程序的情况下提供了一种更佳的变量选择。

握手项需要几个时钟周期来跨时钟域传输值。这直接导致了通信延迟。与变量和寄存器项一样，握手项不提供缓存，所以它们不是为数据流而设计的，其吞吐量性能不像延迟那样具有相关。

与寄存器和FIFO一样，握手项可以由VI和项目定义。每种类型在吞吐量和迟延方面的性能特性都是一样。与FIFO相比，握手项消耗更少的FPGA资源，而且不消耗内存块。

内存项

内存项是首要的FPGA存储机制，在FPGA内部传输数据。它们提供通用寻址空间，可通过应用程序的不同部分进行访问。

内存项的资源实现类型可以配置为以下任一情况：

内存块——内存项使用内存块编译，相对比用其他内存类型，时钟速率更快。您可以针对读写访问或者双端口读取访问配置内存块。您也可以在一个时钟域内使用通过内存块实现的内存项来写入数据，然后从另一个时钟域读取数据。在这种实现中，每个内存项您只能使用一个写入程序节点和一个读取程序节点。内存块存储数据时并不需要消耗FPGA逻辑资源。

当您在多个时钟域使用通过内存块实现的内存项时，同步读写相同地址是可能的。但是，这样做会导致读取不正确的数据。内存项没有内置握手机制，所以您需要额外的握手代码，如果您想要避免读取程序和写入程序在同一项上发生冲突，则需要保证内存内的数据一致性，或者以无损方式传输数据。

通常来说，对于无损传输较为重要的情况，建议采用FIFO机制。对于随机数据访问较为重要的情况，可使用内存项，而且写入程序只知道最近的一致内存状态，而不需要知道一段时间内的每个一致内存状态。

内存块实现要求整个时钟周期读取请求的地址。因此您必须将反馈节点直接放置在数据输出之后，在下个时钟周期读取Data值。注意这会使整个读取操作延迟增加一个周期。

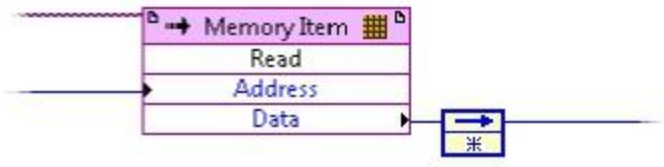

图82. 使用内存块的内存项检索一个值需要一整个周期，所以您必须立刻在读取操作的下游放置一个反馈节点。

查找表(LUTs)——LUT，也叫做分布式RAM，由硬连线到FPGA上的逻辑门构成。由于LUT可以作为逻辑资源或者内存运行，因而需要消耗逻辑资源。

当内存项是由LUT实现时，您不需要直接把Read输出连接到未初始化的移位寄存器或者反馈节点。您也可以在您提供Address参数的同一个周期内读取内存项的数据。

在访问SCTL内部的内存时，或者您需要在一个周期内读取数据时，又或者您剩下的内存块有限时，建议使用LUT作为内存项。

动态RAM (DRAM)——DRAM是FPGA外部的一种内存形式，在某些NI RIO设备上可用。DRAM提供大量的存储空间，从成百上千到成千上万兆字节不等。因此您可以使用它来存储FPGA无法存储的数据。但是，由于DRAM位于FPGA外面，因此应用程序在一个单独的时钟周期内无法从DRAM接收数据。您必须使用Request Data 和Retrieve Data methods来接收数据。在多个读取请求发生冲突时，顺序访问内存可以防止确定性定时。

图93. 读取DRAM是在Retrieve Data方法的Output Valid信号为真时通过给定地址请求数据，然后在几个周期后对其进行检索来实现。

在多个读取请求发生冲突时，顺序访问内存可以防止确定性定时。动态RAM必须也周期性地刷新，当刷新内存地址和读写请求发生冲突时，这会导致额外的不确定性。

为了优化DRAM性能，建议在突发情况下（比如每个时钟周期）发送数据请求或者将数据写入DRAM。

SCTL间通信机制小结

下列表格总结了每种SCTL间通信机制的特点。

传输方式	有损	SCTL存取程序数量	FPGA资源	跨时钟域1	主要应用
全局变量	是	1个写入程序 N个读取程序	逻辑	是	在一个VI内或多个VI之间共享当前变量值
局部变量					在一个具有前边板项的VI内共享当前变量值
寄存器项	是	N个写入程序 2,3 N个读取程序	逻辑	是4	写入在一个VI内或多个VI之间共享当前寄存器值的可复用IP
内存项	是	N个写入程序2,3,8 N个读取程序2,3	LUTs	否	快速随机读/写访问一个时钟域内的小型内存（数百个字节）
			BRAM	是4,5,6,7,9	随机读/写访问至少两个时钟域内的中型内存（数千个字节）
			DRAM	是5,6,7	较长延迟，随机读写访问至多两个时钟域内的大型内存（兆字节）
握手项	否	1个写入程序 1个读取程序	逻辑	是	同步写入在最多两个时钟域内共享一个值的可复用IP
FIFOs	否10	N个写入程序2,3 N个读取程序2	BRAM	是3,4,6,7	在至多两个时钟域跨循环进行无损大型数据传输和存储缓存
			LUTs	否	在一个时钟域内进行无损中型传输和存储缓存
			触发器		,在一个时钟域内进行快速无损小型传输和存储缓存

1 跨时钟域指的是此机制是否可以用于在不同时钟域的SCTL之间传输数据。所有这些机制可以用于在同一时钟域的SCTL之间传输数据。
2 如果您需要多个存取程序的话，必须明确禁用仲裁。
3 对于多个写入程序，您必须实现自身同步机制来阻止数据崩坏或者由于写入程序冲突造成的不正确寻址。
4 所有的写入程序都必须在同一个时钟域。
5 所有的读取程序都必须在同一个时钟域。
6 您必须启用Dual-clock选项来实现跨时钟域。
7 对于多个读取程序，您必须自行开发同步机制来阻止数据崩坏或者由于读取程序冲突造成的错误寻址。
8 使用内存块的内存项支持多个SCTL读取程序。使用双端口的BRAM接口只有一个写入程序和一个读取程序来最小化访问抖动。
9 当您在多个时钟域内使用通过内存块实现的内存项，同步读写相同地址是可能的。但是这样会导致读取的数据不正确。
10 Timed Out?端用于决定读写方法是否成功。

在FPGA和主机系统之间传输数据

FPGA主机界面聚集了FPGA和主机之间主要的数据传输机制。基于主机的处理在LabVIEW RIO应用程序中扮演关键角色，所以理解不同FPGA界面机制的延迟和吞吐特性是很重要的。本部分介绍了这些机制的性能方面以及推荐的应用。

您可能使用FPGA主机界面在主机和FPGA之间传输数据来：

- 实现您用FPGA无法实现的更多数据处理
- 实现FPGA设备不能执行的运算，比如双精度和扩展精度浮点算法
- 创建有FPGA设备作为大型系统组件的多层应用
- 记录数据到磁盘

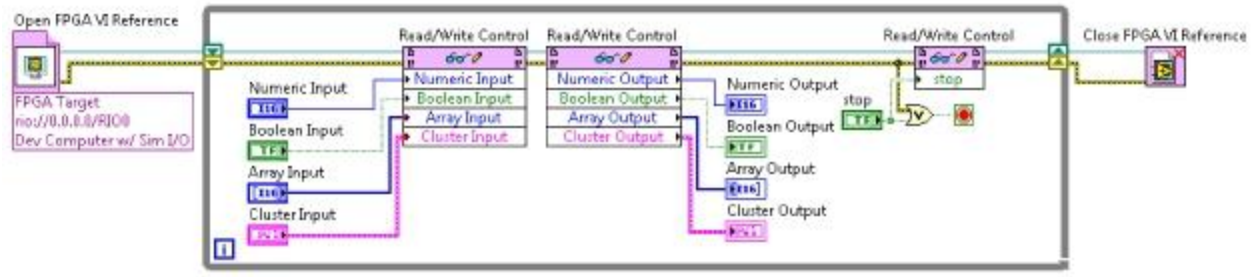

图94. FPGA主机界面是在FPGA和主机之间传输数据的主要机制。它使用面向对象的方法，顶层FPGA VI输入控件和显示控件通过属性节点来访问。方法可以通过界面上执行不同的操作来调用，比如DMA FIFO访问。

顶层VI输入控件和显示控件

LabVIEW FPGA模块用于创建顶层FPGA VI上每个输入控件和显示控件与一组硬件寄存器之间的寄存器映射。FPGA主机界面提供了对寄存器的读写访问。熟悉驱动程序和底层I/O编程的人员可以把这看作到FPGA的内存映射寄存器访问（MMRA）。

读写输入控件函数支持标量数据（比如数值或者布尔输入控件）和结构化数据（比如数组和簇）。正如资源优化技巧一章所介绍的，不要在FPGA VI的前面板过度使用数组，因为它们会消耗大量的FPGA资源。

顶层FPGA VI输入控件和显示控件与FPGA通信具有最低延迟。尽管取决于状态机， FPGA的读写通常在微秒范围内完成。这为交换状态和设定配置参数提供了有效方法。使用多核主机并将其中一个CPU内核专用于特定寄存器的读取，以实现最低延迟和更紧凑的不同。使用LabVIEW Real-Time主机可提供良好的平均延迟，增加确定性，这把最大访问延迟限定在FPGA。

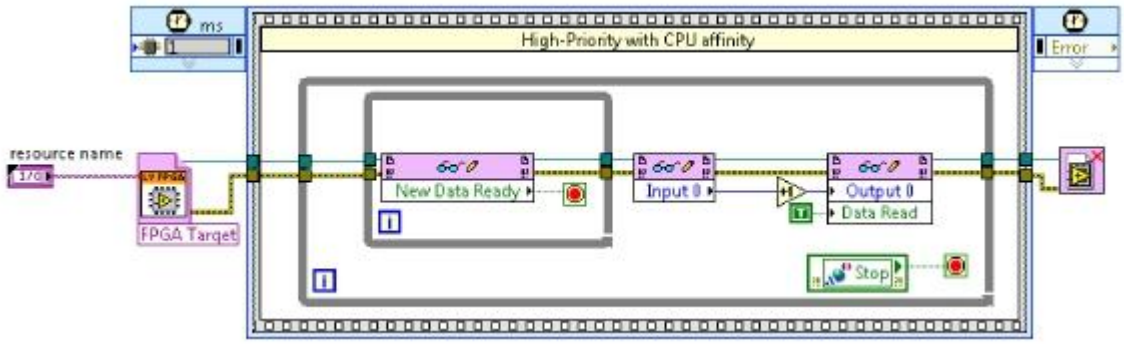

图95.这个定时顺序结构为主机端的循环分配CPU亲和性和高优先级，主机端的循环会对FPGA VI输入控件进行轮询。这种方法为FPGA和主机之间传输时间敏感型数据提供了最低延迟。

请记住FPGA更新主机界面寄存器的速度要比主机读取寄存器的速度更快。因此主机应用程序很容易丢值。如果您想保持多个读取的数据一致性，或者想让数据在FPGA和主机端进行无损数据传输，额外的握手或者不同的同步机制是必需的。FPGA主机接口FIFO直接访问主机内存，并且是在FPGA和主机之间传输大量信息的推荐方法。

FPGA主机接口FIFO（直接内存访问）

FPGA主机接口FIFO使用直接内存访问(DMA)来缓存和高速传输数据到主机系统内存，几乎不需要处理器的参与。与前面板输入控件和显示控件相比，在发送大块数据时，这是一种有效机制。

FPGA主机接口FIFO的API与标准FPGA FIFO的API相似。主机接口FIFO使用单向传输机制，您可以将其配置为主机到FPGA或者FPGA到主机两种方向。主机接口FIFO的 API根据从主机端访问还是从FPGA端访问而有所不同。

图96. 主机接口FIFO不考虑FPGA和主机系统之间DMA通信的复杂性。此范例展示您如何从FPGA端一次写入一个元素，并从主机端一次读多个元素。

配置FPGA主机接口FIFO实现最佳性能

使用主机接口FIFO，您的代码必须处理并且应对瞬变状态来获得可能的最佳吞吐量。正如本部分稍候将会讨论的，FIFO可以事先填充好数据，并通过控制DMA的初始化顺序、源和汇来最小化传输过程初期发生错误的可能性。

一个DMA通道包括两个FIFO缓存。一个在主机上；一个在FPGA设备上。主机端和FPGA端都有各自的缓存区，只要满足某些条件，DMA引擎可将数据从一端传输到另一端。

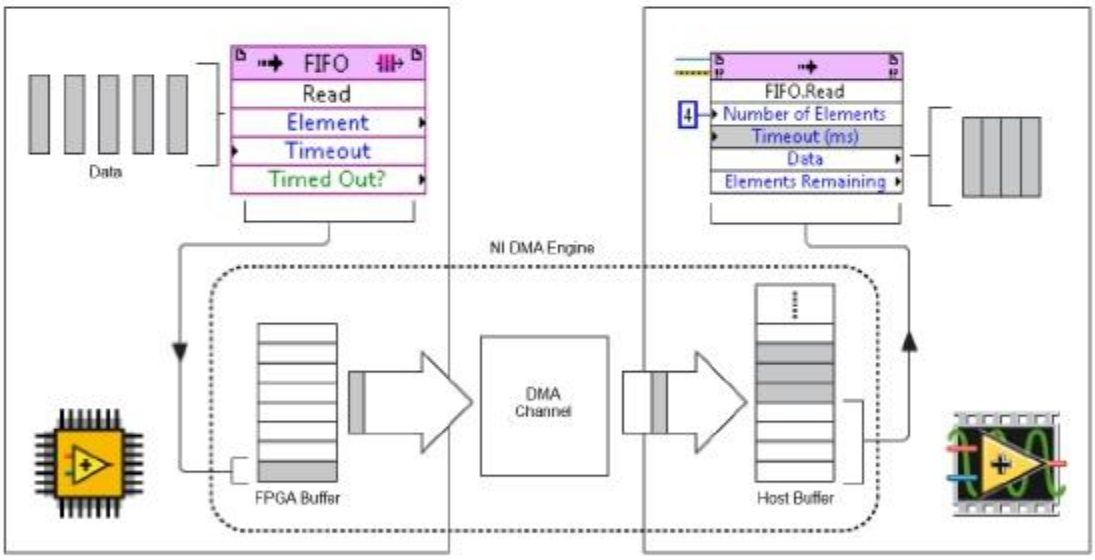

图97. DMA引擎通过局域总线传输数据，直接访问主机内存，以极少主机CPU负荷实现接近总线带宽的传输速度。

主机端缓存区可以在运行时传输过程开始之前通过主机应用程序进行配置。目前主机端缓存区的默认大小是大于10000隔元素，并且是FPGA端大小的两倍。如果您遇到溢出或者下溢错误，NI推荐您把缓存大小增加到4096个元素的倍数。建议您将主机端缓存至少设置为您一次想要读写的元素数量的五倍。如果延迟不成问题的话，可将主机端缓存区增大，以容纳数秒钟的数据。

FPGA端的缓存区是由FPGA上的逻辑资源实现的。您可访问LabVIEW项目中FIFO项的FIFO属性对话框，在进行编译之前配置FPGA端缓存的大小。FPGA端缓存默认为1023个元素的深度。因为FPGA能够以比主机更快的速度维护缓存区，因为FPGA通常受限于资源，NI推荐保持缓存大小不变，除非有明显证据表明增加缓存可以提高性能。当您的设计接近尾声时，如果延迟不成问题，您可以考虑在编译过程允许的条件下，尽可能增加FPGA端的缓存。

在维护和提供数据到DMA引擎方面，FPGA通常比主机系统要快得多，所以在进行FPGA FIFO传输时，主机CPU需要更努力地运行来跟上FPGA的速度。在临时缓存空间上，FPGA也比主机受到更多的资源约束。因此CPU应该尽可能频繁地被读取来保持较小的FPGA缓存。

在FPGA传输数据到主机时，CPU尝试保持FPGA缓存尽可能为空，这样如果瞬态条件短暂地影响CPU或者局域总线，FPGA就有空间来存放数据。当主机传输数据到FPGA时，主机的目标是保持FPGA缓存尽可能满，这样在瞬态条件发生时FPGA就有足够的数据进行处理。在从FPGA传输数据到主机时，您需要理解的是DMA引擎仅在主机执行Read 或者 Start方法才启动。如果DMA引擎不运行，则FPGA快速溢出FIFO缓存；因此，在FPGA端将数据写入FIFO之前，您应该通过调用主机的Read 或者 Start方法来启动DMA。

一旦传输开始，主机通过调用Read方法读取主机端缓存的数据。如果主机端缓存满了，则DMA引擎停止传输数据，FPGA端的FIFO报告溢出，作为超时条件。

主机端Read方法使用轮询机制在超时前检查请求的数据量。通过在主机VI的定时循环或者定时序列结构内放置一个Read方法，您可为此进程分配一个专用的CPU内核，以高CPU负载为代价提供最佳性能。如果主机端保持DMA数据流的速度，您可能希望使用低超时值，在读取调用不返回数据时，明确调用Wait Microsecond函数，这样您就可以用其他系统进程共享CPU时间。

将数据从主机传输到FPGA时，或者必须让FPGA总是有数据来处理或者输出时，在开始执行从FPGA端读取数据这一进程之前将数据写入FIFO。在这种情况下，较大型的主机端缓存更适合用于应对瞬态条件，但是它们会增加传输延迟。

HOST界面FIFO延迟和吞吐量

正如之前所讨论的，延迟和传输中数据量和额外的机制开销成正比。大部分传输中的数据不是在FPGA就是在主机端缓存里。

在从FPGA传输数据到主机时，主机端缓存应该在稳定状态几乎为空。此外，FPGA端缓存应该相对较小，所以延迟是由DMA引擎传输数据块到主机的频率决定的。DMA引擎传输数据到主机，只要满足任一下列条件：

- FPGA端缓存1/4满
- FPGA缓存至少有512字节（一个完整的PCI Express数据包）
- DMA控制器的驱逐定时器触发——此定时器的周期差不多为一微秒。

将数据从主机传输到FPGA时，主机端缓存应该在稳定状态几乎为空。此外，FPGA端缓存应该仍然为空，所以平均延迟大致是由主机端缓存大小除以平均吞吐量来决定的。平均吞吐量取决于FPGA从DMA FIFO读取的频率，受到系统总线最大带宽的限制。

FPGA DMA系统带宽取决于总线技术和您选择的平台。NI采用最新的总线技术后，总线带宽迅速增加，所以建议您阅读产品文档，确保您了解了最新的规格。NI硬件产品的当前带宽能力范例包括以下：

- 通过利用PXI Express机箱的PCI Express技术，NI FlexRIO仪器当前提供最高的基于FPGA的DMA吞吐量。
  - 基于NI FlexRIO系列的NI PXIe-797xR使用四个PCI Express 2.0 lane来实现1.7 GB/s的单向吞吐量和每个方向各1.2 GB/s的双向数据传输。
  - 基于NI FlexRIO系列的NI PXIe-796xR可以实现800 MB/s其他端至FPGA，838 MB/s FPGA至其他端以及每个方向各800 MB/s的双向数据传输速度。
- NI cRIO-9068使用高速AXI总线提供单个DMA通道300 MB/s的速率或者16个DMA通道300 MB/s的总速率。
- 多核NI cRIO-9082使用PCI总线将FPGA和处理器相连，提供90 MB/s 到 100 MB/s的带宽。

减少主机界面FIFO的缓存复制

读写主机界面FIFO时，需要对LabVIEW和NI-RIO驱动内存缓存之间的数据进行复制。这些额外的数据复制会消耗CPU周期，从而限制使用FIFO创建的应用程序的大小。LabVIEW FPGA模块扩展了DMA FIFO API，通过结合LabVIEW中的数据值引用概念和Acquire Data Region 和 Acquire Write Region FIFO方法，提供了对NI-RIO驱动端DMA缓存的直接访问。3

使用域API，您可以获得NI-RIO缓存的数据值引用，直接操作In Place Element结构范围内的数据。读取结构内的引用数据并不会复制这些数据，任何写入操作直接进入到驱动端的缓存。使用数据值引用可以允许驱动程序复用内存，以便进行之后的操作，使用完之后您必须删除数据值引用。

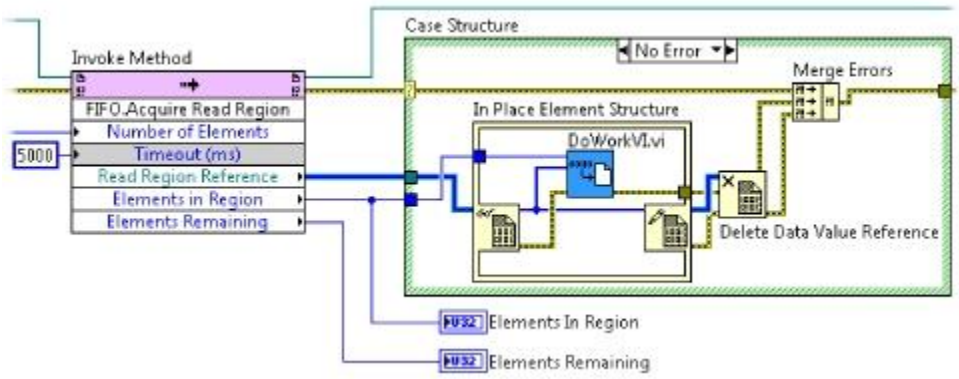

图98. 此程序框图使用通过Acquire Read Region 方法进行配置的Invoke Method函数，采集一个区域的数据，以便从缓存读取数据。然后，通过一个In Place Element结构，它把引用传递到读取区域，然后在读取区域内进行进一步处理，而无需复制缓存的数据。

中断

LabVIEW FPGA设备可以使用硬件中断来同步主机。对于主机需要等待来自FPGA的信号（比如您轮询一个显示控件的状态来查看某个事件是否发生）的应用程序，中断是非常有用的。这需要特定的主机处理能力来连续地从显示控件读取数据。中断提供一种无须轮询的方法，其中主机不需要询问设备的状态，从而释放了CPU和总线资源。

由于操作系统开销，中断确实比轮询要付出更高的代价。当中断发生时，操作系统通常会询问某些设备来找出中断来源，这样中断延迟就取决于所选择的总线、CPU或者操作系统。

采用高性能嵌入式控制器的PXI系统具有最低平均中断延迟，通常为几微秒。CompactRIO控制器的中断延迟通常为几十到一百微秒。由于LabVIEW Real-Time具有确定的操作系统响应，控制器使用LabVIEW Real-Time可提供一致的中断延迟。

设备间数据传输

点对点数据流

在PXI上，您可以绕过主机处理器，使用点对点数据流直接把数据从FPGA设备高速传输到另一个FPGA设备。您还可以在FPGA和非FPGA设备（比如数字化仪器、任意信号发生器和NI其他模块化仪器）之间使用点对点数据流。

点对点数据流利用PCI Express技术，通过在PXI机箱的设备之间设置一个直接单向通信路径来运行。直接连接对于在一个机箱上的多个NI FlexRIO设备之间分布高性能LabVIEW FPGA应用程序非常有用。如果您的应用程序耗尽NI FlexRIO设备上的所有空间，您只需添加一个板卡到机箱中，并将数据传输到板卡中进行进一步处理。

由于NI的多个模块化仪器都支持点对点数据流，您可以使用这些高性能仪器直接采集数据并将数据传输到NI FlexRIO设备进行处理。例如，基于PXI Express的高吞吐量数字化仪可以将数据传输到FPGA仪器进行频谱分析，这释放了主机处理器来执行其他任务。PXI Express使用树状拓扑结构。因此通过仔细选择设备的插槽，您可以使用点对点数据流直接将数据从数字化仪发送到FPGA设备，而不影响系统中的其他点对点传输。

点对点数据流代表一个FIFO接口，与LabVIEW FPGA中其他FIFO相似。此接口还可决定FIFO中的采样点数或者可用插槽的数量以及点对点数据流的状态。您可以使用此API将数据从一个设备写入，几微秒之后从另一个设备读取。

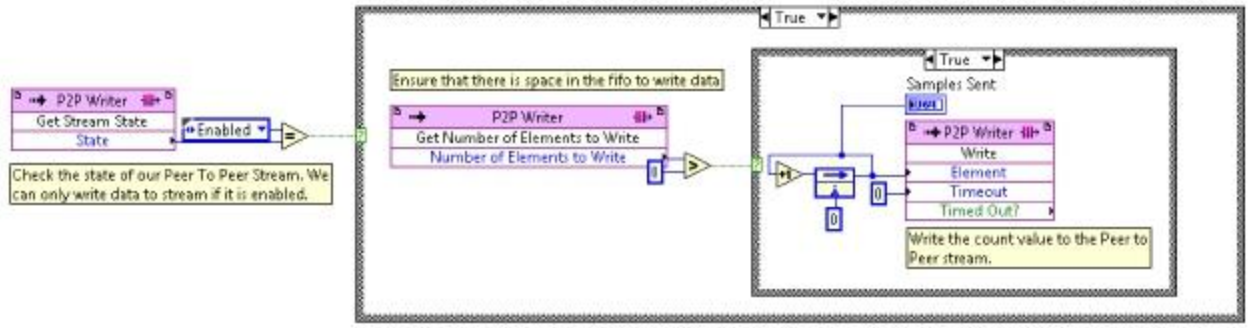

图99. FPGA端的点对点数据流API提供一个与主机接口和FIFO接口相似的接口，具有查询同级缓存区空间或可用元素数量以及数据流状态的能力。

您可以使用将点对点终点连接来实现数据流的API，在主机系统里配置点对点连接。配置完成后，数据在FPGA之间传输，主机就不再参与了。

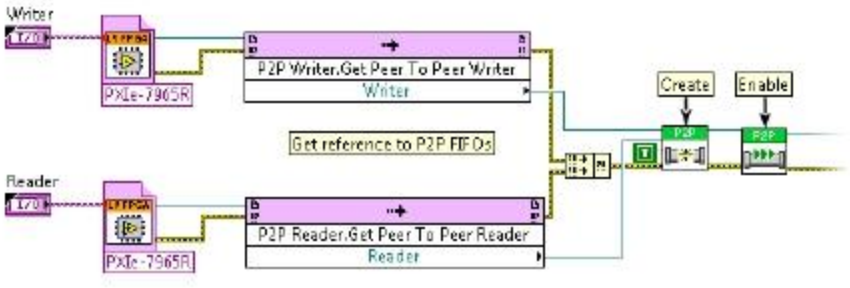

图100. 点对点数据流通过一个主机端API进行配置和监测。

点对点吞吐量取决于设备、机箱、插槽配置以及所使用的控制器。在为点对点数据流应用选择和配置硬件时，考虑以下部分讨论到的范例和因素。

点对点数据流的设备选择考虑要素

点对点设备必须能够按照所需的速率提供或者接受数据，这反过来对其总线接口和处理逻辑速度提出了要求。速度最慢的点限制了整体可达速率。

- 基于NI FlexRIO设备的NI PXIe-797xR支持PCI Express 2.0通信，这为点对点通信提供了最高吞吐量。这些设备利用四条PCI Express线，单向传输吞吐量可达1.5GB/s。

- 基于NI FlexRIO设备的NI PXIe-796x使用PCI Express 1.0 卷道，以超过800 MB/s的速率读写模块的数据。当双向同步进行数据流时，FPGA模块在每个方向的速率可以超过700 MB/s。

点对点数据流的机箱选择考虑要素

PCI Express背板通过机箱交换路由数据，提供高带宽点对点连接，实现了点对点数据流。带宽取决于机箱插槽中与同一个PCI Express开关直接连接的模块的开关。

NI PXIe-1085机箱通过八条可用于任何插槽的PCI Express 2.0卷道提供最大4 GB/s速率。设备可能无法使总线饱和，但是如果您决定将来升级设备，则可具备所需的带宽。相似地，PXI Express 2.0设备可以支持PXI Express 1.0的机箱，但是会受到PXI Express 1.0带宽的限制。

机箱设计也使带宽取决于您为设备选择的插槽。当点对点数据流系统中的模块没有连接到同一个PXI Express开关时，数据必须通过背板或者主机控制器开关来传输，这使得吞吐量取决于这些开关的能力。

NI PXIe-1085机箱具有两个PCI Express开关，通过八条PCI Express 2.0卷道连接，可实现4 GB/s的开关间带宽。当一个点插入插槽2到9之间的任意槽而其他点插入在插槽11到18之间的任意槽时，开关间连接由点对点连接共享。以此方法添加足够的点最终可使开关间连接达到饱和，但这会影响吞吐量。因此推荐您在尽可能在同一开关上放置点对点数据流对。

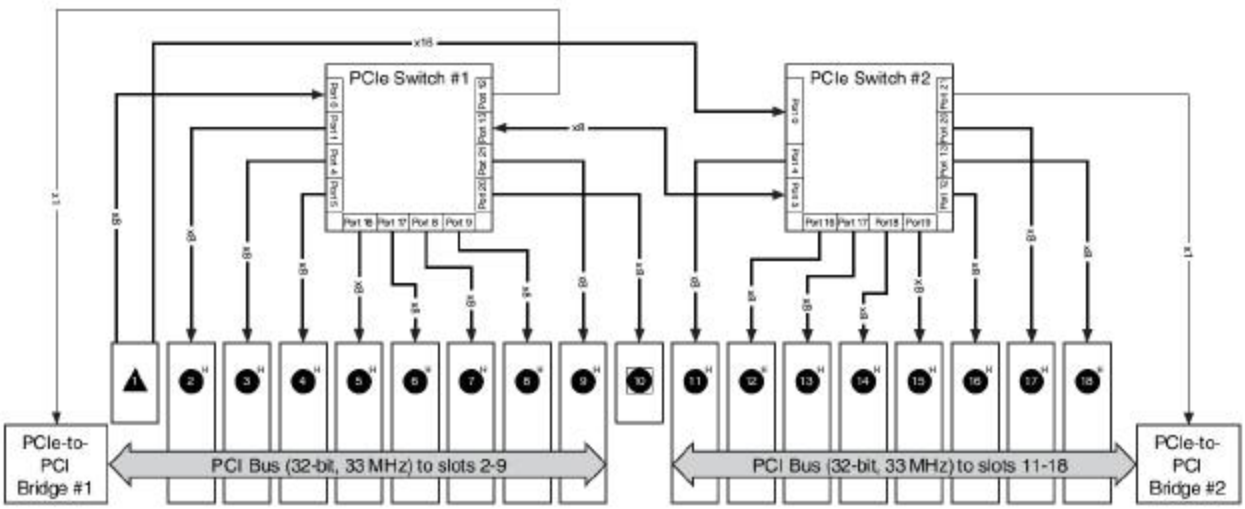

图101. NI PXIe-1085机箱背板有两个PCI Express 2.0机箱，与18个机箱槽互连。

对于NI PXIe-1065机箱，机箱包括两个部分。在槽9到14的设备直接通过背板上的PCI Express开关直接相互通信。在槽7和8的设备必须通过主机控制器板载芯片开关，这使得带宽取决于控制器。

较早的控制器，比如NI 8130，在点对点数据流从机箱的一部分流到另一部分时将带宽限制在大约640 MB/s。有些控制器，比如NI 8108，不支持点对点数据流，但是数据流仍然可以在设备与机箱开关直接连接时发生。

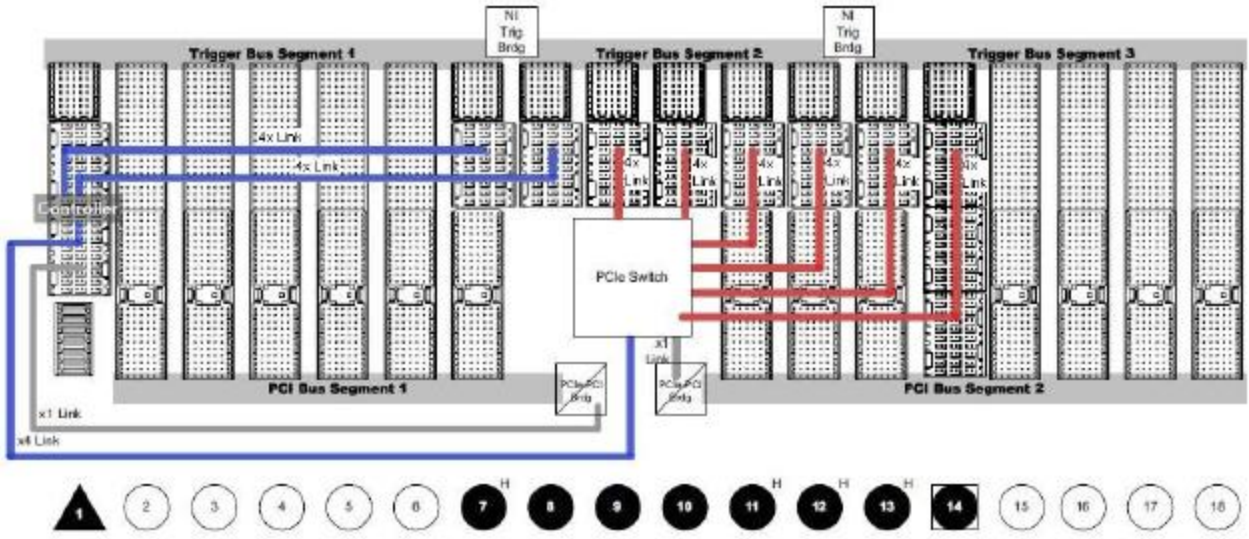

图102. 在NI PXIe-1065机箱背板上，连接槽7和8的设备在执行点对点数据流时，必须通过插槽1的主机控制器开关，这使得吞吐量取决于控制器。

当多个点对点数据流同步运行时，带宽由链路上同一方向的数据流平均共享。由于需要应答传输会数据源的消息，反向同步流会降低其他数据流的吞吐量。

最后，在尝试解决需要跨多个FPGA的数据流的闭环控制应用时点对点延迟很重要。点对点延迟通常是2 µs 到 4 µs，但是如果系统中的数据流相互竞争流量，延迟可能会增加到10 µs或者更多。因此如果延迟很重要的话，您可尝试使用专用PCI Express开关来隔离点对点流量

更多资源

[1]	[1] http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_dma_best_practices DMA最佳实践
[2]	[2] http://www.ni.com/white-paper/10801/en 点对点数据流介绍
[3]	3] http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgahelp/fpga_zerocopy_dma/ 提供访问DMA FIFO(FPGA 模块)的效率

此页特意留白。

下一步

LabVIEW产品文档详细介绍了如何完成此指南中讨论到的任务。硬件手册也包含NI RIO设备特性和性能特征的有价值信息。

您可通过NI的服务主页面——ni.com/support快速访问手册、KnowledgeBase文档、教程、范例代码、社区论坛、技术支持和客户服务。

正式培训

您可以从NI获得高吞吐量LabVIEW应用程序的面授培训。高吞吐量LabVIEW FPGA教程更详细地介绍了此指南中包括的许多主题，并附有指导型练习，以检验您是否理解相关内容。

本指南还介绍了高性能LabVIEW FPGA应用程序的相关主题，侧重阐述单周期定时循环内的编程。并非每个应用程序或者应用程序的每个部分都要求SCTL编程。标准LabVIEW FPGA教程包含许多在SCTL外部进行LabVIEW FPGA编程所需的基本概念。此教程提供面授培训，但是包含于在线自学培训中， 标准服务计划(SSP)的订阅者可免费访问。

评估NI RIO平台

您可以下载LabVIEW和LabVIEW FPGA模块的评估版本，即使您没有NI硬件也可以创建、仿真和编译FPGA设计。您也可以获取LabVIEW RIO评估工具包[4]——评估NI RIO平台的低成本选择。

NI联盟伙伴和服务

NI合作伙伴可以帮助您设计、集成和部署您的应用程序。NI联盟伙伴网络项目囊括了全球700多家独立的第三方公司，旨在为工程师提供基于图形化系统设计的完整解决方案和高品质产品。

最后，您可以寻求NI现场工程师的帮助，讨论如何使用NI平台解决您的具体应用问题。

更多资源

[1] [1] http://sine.ni.com/tacs/app/overview/p/ap/of/lang/en/ol/en/oc/us/pg/1/sn/n24:16770/id/2158/
高吞吐量LabVIEW FPGA培训

[2] [2] http://sine.ni.com/tacs/app/overview/p/ap/of/lang/en/ol/en/oc/us/pg/1/sn/n24:4769,n8:4398/id/1597/
LabVIEW FPGA培训

[3] [3] http://www.ni.com/white-paper/14457/en/
在线自学培训

[4] 4] http://www.ni.com/rioeval/
NI LabVIEW RIO架构评估选项

[5] [5] http://www.ni.com/alliance/
NI联盟伙伴网络

修订和反馈

NI致力于提供高质量内容，欢迎您提供任何评论。如果有任何关于修订此指南的反馈，请随时发送邮件至hprioguide@ni.com。

如果你认为某些地方需要解释，请提交问题到LabVIEW FPGA论坛，这里有应用、支持和研发的工程师可以回答您的问题，且所有读者都可以看到。以“fpga”开头发送消息到“LabVIEW”版块，并添加“hprioguide”标签可以更快得到回复。

修订版本n	日期	改动小结
1.0	02/14/2014	第一版
1.1	02/25/2014	集成高吞吐量IP章节 ▪ 截断向负无穷的取整偏差 ▪ 溢出和舍入不需要调整除非覆盖输出类型 资源优化章节 ▪ 溢出仅强制发生，不舍入 ▪ 溢出和舍入不需要调整除非覆盖输出类型 ▪ 函数会取较大的值，不需要比较逻辑。 ▪ 离散延迟函数现在支持不同的数据类型。. ▪ 离散延迟函数支持动态延迟。 微小的语法更正、说明和图修改。