NI LabVIEW High-Performance FPGA Developer's Guide

Recommended Practices for Optimizing LabVIEW RIO Applications

Revision No. 1.1 – February 2014

TABLE OF CONTENTS

INTRODUCTION

Field-programmable gate array (FPGA) technology provides the performance and reliability of dedicated, custom hardware. As a LabVIEW FPGA user, you can take advantage of FPGA technology within the same design environment you use to program desktop and real-time systems.

High-performance LabVIEW FPGA applications push the capabilities of NI reconfigurable I/O (RIO) devices with respect to timing, FPGA resources, and other dimensions. This guide helps you create high-performance applications by offering a summary of common LabVIEW FPGA optimization concepts and techniques.

INTENDED AUDIENCE

If you are already familiar with LabVIEW or the LabVIEW FPGA Module, use this guide to learn about advanced, industry-agnostic LabVIEW FPGA concepts to help you tackle more demanding applications that require high throughput, precise timing control, or increased FPGA resource efficiency.

The term “high performance” is a relative one, but it generally refers to the low-level details that make LabVIEW FPGA programming more challenging when you need to extract even more performance from your design. Refer to the High-Performance FPGA-Based Design chapter for an overview of high-performance application concerns.

The NI RIO hardware platform takes care of many low-level implementation details so you can concentrate on solving application-specific challenges. If you have digital design experience and are familiar with VHDL, Verilog, or electronic design automation (EDA) tools, use this guide to understand how LabVIEW FPGA achieves similar performance while offering increased productivity because of its tight integration with the NI RIO hardware platform.

PREREQUISITES AND REFERENCES

This guide assumes that you are familiar with LabVIEW programming and basic LabVIEW FPGA tasks, such as creating, compiling, simulating, and deploying FPGA VIs. Refer to the product documentation for more information on these topics.

You should also have some basic understanding of what FPGAs are and how your application can benefit from their use. For an introduction to FPGA-based design, see the FPGA Fundamentals white paper on ni.com.

This guide collects and summarizes the concepts you need to create and optimize LabVIEW FPGA applications, regardless of your LabVIEW version. Refer to the product documentation included in your version of LabVIEW to learn API- and environment-specific details because the product might evolve from release to release. This guide also points to shipping examples, online tutorials, and other references when appropriate.

This guide is similar to the popular NI LabVIEW for CompactRIO Developer's Guide [2], which provides best practices for designing NI CompactRIO control and monitoring applications. Refer to the *Customizing Hardware Through LabVIEW FPGA* section of that guide for an introduction to LabVIEW FPGA programming. This guide builds on the concepts presented in that section to help you achieve higher performance.

Additional Resources

[1]	FPGA Fundamentals http://www.ni.com/white-paper/6983/
[2]	NI LabVIEW for CompactRIO Developer's Guide http://www.ni.com/compactriodevguide/

HIGH-PERFORMANCE FPGA-BASED DESIGN

ADVANTAGES OF FPGAS

FPGAs offer a highly parallel and customizable platform that you can use to perform advanced processing and control tasks at hardware speeds.

FPGAs are clocked at relatively lower rates compared to CPUs and GPUs, but they make up for the difference in clock rate with specialized circuitry that can perform multiple, sequential, and parallel operations within a single clock cycle. You can combine the massive parallel programming features associated with FPGAs with the tight I/O integration on NI RIO devices for higher throughput, greater determinism, and faster response times to tackle high-speed streaming, digital signal processing (DSP), control, and digital protocol applications.

HIGH-PERFORMANCE LabVIEW FPGA

When you use standard LabVIEW programming techniques in LabVIEW FPGA, you immediately get most of the benefits of the FPGA-based approach. Advanced applications may need to push the system even further on one or more of these related dimensions: throughput, timing, resources, and numerical precision.

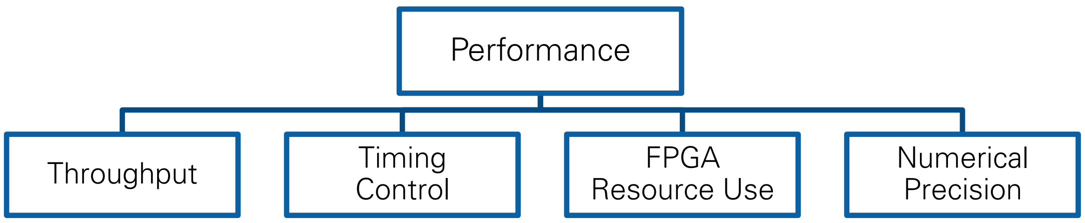


Figure 1. The multiple dimensions to high-performance RIO applications are interrelated. Increasing throughput, for example, may affect timing and resource use.

These dimensions are often interconnected. Improving your design by focusing on one of these dimensions can often affect the others, sometimes positively, but often at the expense of another dimension. For example, if you focus on throughput, your design may fail some other timing requirement; therefore, you need to understand these dimensions and how they relate to each other. This section provides basic definitions of the various considerations, and this guide as a whole expands on related techniques throughout.

THROUGHPUT

Throughput is a key concern for DSP and data processing applications. Technology advancements in communication, manufacturing, medical, aerospace and defense, and scientific measurement systems lead to increasing amounts of data that must be processed in shorter amounts of time. Throughput is measured as work per unit of time. In the majority of LabVIEW RIO applications, work refers to the processing or transfer of samples, and throughput is typically measured in samples per second or some equivalent form such as bytes, pixels, images, frames, or operations per second. The fast Fourier transform (FFT) is an example of a processing function in which throughput is measured in FFTs, frames, or samples per second.

In this guide, the Throughput Optimization Techniques chapter offers a more in-depth discussion of the factors that affect throughput, such as clock rate and algorithm parallelizability, as well as a set of techniques that can help you achieve higher throughput when creating LabVIEW FPGA applications.

TIMING CONTROL

Timing control refers to the ability to prescribe and measure the amount of time between events of interest in a system. Precise timing control is important to digital protocol and high-speed control applications because it can affect the ability of systems to communicate or the stability of the controlled system. Control applications often require response-time guarantees, which are measured as the time between the sampling of the controlled system and when the controller's outputs update. This is also referred to as I/O latency. In digital protocol applications, timing specifications may refer to the target, minimum, or maximum time between events related to the data or transmitted signals. When you create LabVIEW FPGA applications, your designs translate to a hardware circuit, so you can create designs that have fast timing responses with little jitter.

In this guide, see the Timing Optimization Techniques chapter for a more in-depth discussion about latency as well as techniques that can help you achieve faster and more precise timing responses when using LabVIEW FPGA in your applications.

FPGA RESOURCE USE

An FPGA has a finite number of hardware resources and is often more constrained in storage than a processor or microcontroller. Ensuring your design fits on an FPGA is a strict constraint on the development process. FPGAs are also made up of different types of resources, such as logic, signal processing, and memory blocks, and running out of one type of resource can prevent the whole design from compiling.

More importantly, resource use can dramatically impact performance, including throughput and timing constraints. Refer to the Resource Optimization Techniques chapter in this guide for a description of the different resources that make up an FPGA, ways to ensure your design fits on the FPGA, and methods to increase its performance.

NUMERICAL PRECISION

Numerical precision is the availability of enough digits or bits required for your application to work correctly. Insufficient precision can make iterative algorithms accumulate small numerical errors that over time lead to completely different results. The number of bits that represent system variables, including the number of bits used for integers, the integer and fractional parts of fixed-point numbers, or the dynamic range of floating-point numbers, can impact performance and resource use of your FPGA application.

Although this guide does not cover specific techniques to help you determine the right numerical precision, the discussion often arises in LabVIEW FPGA design optimization and is a consideration when learning about the other optimization dimensions.

UNDERSTANDING THE NI RIO HARDWARE PLATFORM

Your application may have requirements that include all of the considerations discussed in the previous section. Since these considerations are interrelated, it is important to understand how they interact so you can make the right trade-offs when programming. Later sections of this guide explore each optimization consideration in more detail. This section examines the different incarnations of the NI RIO platform and explains how the different hardware characteristics are best matched to specific applications.

NI RIO FOR PXI AND THE PC

THE PXI PLATFORM

PXI (PCI eXtensions for Instrumentation) is a rugged, modular instrumentation platform designed for high-performance applications. It combines PCI and PCI Express bus technologies with a specialized synchronization bus to deliver a high-performance, low-cost deployment platform for applications such as manufacturing test, military and aerospace, machine monitoring, automotive, and industrial test. PXI is an open industry standard governed by the PXI Systems Alliance (PXISA), which promotes the PXI standard, ensures interoperability, and maintains the PXI specification.

Figure 2. The PXI platform combines modular instrumentation, high-performance processing, data transfer, and synchronization buses with desktop or real-time application development to solve advanced FPGA applications.

PXI Express leverages the PCI Express bus to offer a point-to-point bus topology that gives each device its own direct access to the bus with up to 4 GB/s[1] of throughput. The integrated timing and synchronization lines are used to route synchronization clocks and triggers internally. A PXI chassis incorporates a dedicated 10 MHz system reference clock, PXI trigger bus, star trigger bus, and slot-to-slot local bus, while a PXI Express chassis adds a 100 MHz differential system clock, differential signaling, and differential star triggers to address the need for advanced timing and synchronization.

NI offers multiple RIO products that can be used on the PXI platform and that take advantage of the bandwidth and synchronization features to enable high-performance applications.

[1] Throughout this guide: GB = 10^9 bytes, MB = 10^6 bytes, MiB = 2^{20} bytes, and GiB = 2^{30} bytes

NI FlexRIO

NI FlexRIO is a general-purpose research, design, test, prototyping, and deployment device family based on the PXI platform. NI FlexRIO devices consist of a large FPGA that you can program with the LabVIEW FPGA Module, as well as adapter modules that provide high-performance analog and digital I/O. The adapter modules are interchangeable and define the I/O in the LabVIEW FPGA programming environment.

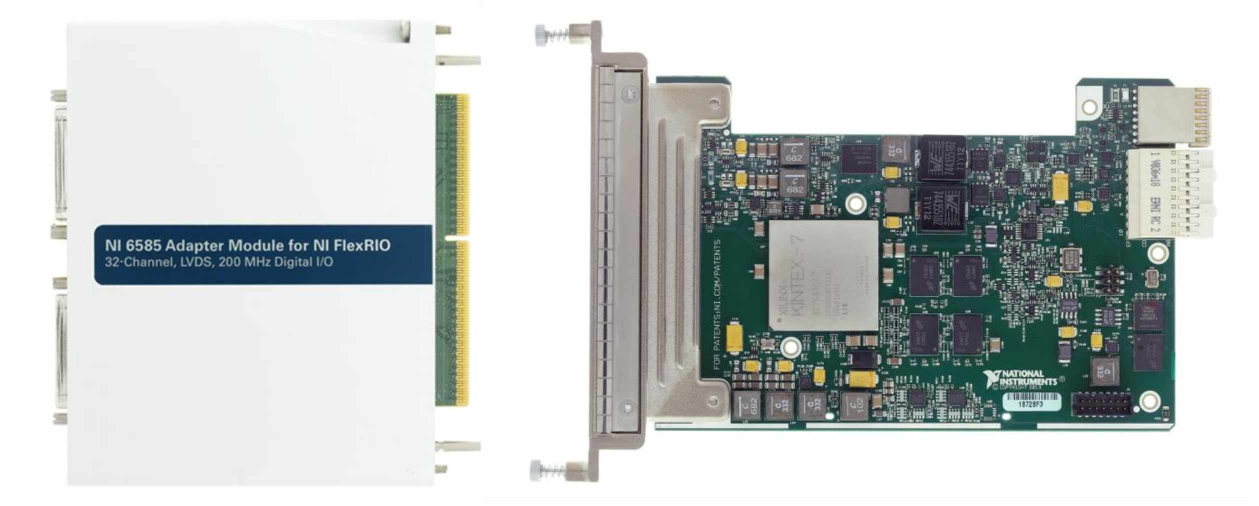

NI 6585 Adapter Module for NI FlexRIO
32-Channel, LVDS, 200 MHz Digital I/O

Figure 3. NI FlexRIO devices host large FPGAs that directly connect to adapter modules for a variety of I/O options.

NI FlexRIO FPGA modules feature Xilinx Virtex-5 and Kintex-7 FPGAs, onboard dynamic RAM (DRAM), and an interface to NI FlexRIO adapter modules that provide I/O to the FPGA. The adapter module interface consists of 132 lines of general-purpose digital I/O directly connected to FPGA pins, in addition to the power, clocking, and supplementary circuitry necessary to define the interface. You can configure these 132 lines for single-ended operation at rates of up to 400 Mbit/s and differential operation at rates of up to 1 Gbit/s for a maximum I/O bandwidth of 66 Gbit/s (8.25 GB/s).

You can stream data directly between NI FlexRIO FPGA modules at rates of 1.5 GB/s. Up to 16 such streams are supported, which simplifies complex, multi-FPGA communication schemes without taxing host CPU resources. Refer to the Data Transfer Mechanisms chapter in this guide for more information on streaming data directly between NI FlexRIO devices.

NI R SERIES MULTIFUNCTION RIO

Though standard NI multifunction DAQ boards can measure and generate a wide variety of signals at different sampling rates, R Series multifunction RIO devices go a step further by incorporating an FPGA to help accomplish tasks for which you may not have previously considered using a DAQ board.

NI R Series multifunction RIO devices offer a combination of value and performance by integrating FPGA technology with analog inputs, analog outputs, and digital I/O lines into a single device. NI R Series multifunction RIO devices support the PCI, PCI Express, PXI, and USB buses, with enclosed and board-only options available.

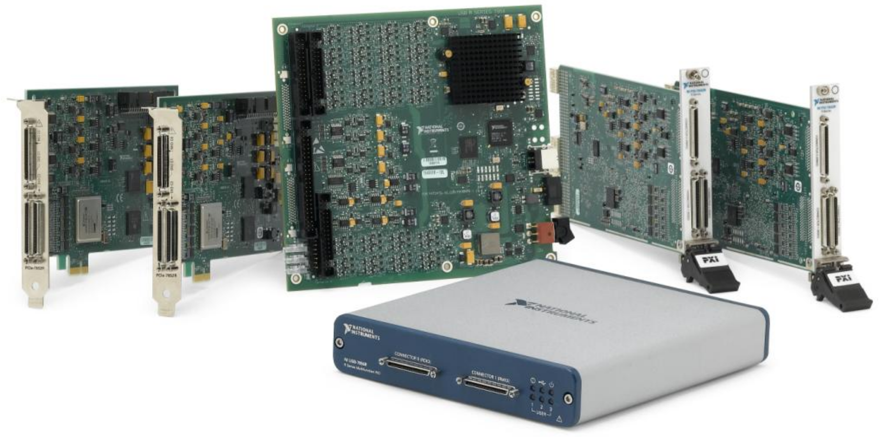


Figure 4. NI R Series multifunction devices extend general multifunction DAQ with an FPGA that's programmable in LabVIEW.

NI R Series multifunction RIO devices feature a dedicated analog-to-digital converter (ADC) per channel for independent timing and triggering and sampling rates up to 1 MS/s. This provides specialized functionality, such as multirate sampling and individual channel triggering, which are outside the capabilities of typical DAQ hardware.

SOFTWARE-DESIGNED MODULAR INSTRUMENTS

NI introduced the world's first software-designed instrument, the NI PXIe-5644R vector signal transceiver (VST), a device that can be used for RF vector signal generation (VSG) and acquisition (VSA) using the NI-RFSG and NI-RFSA drivers.

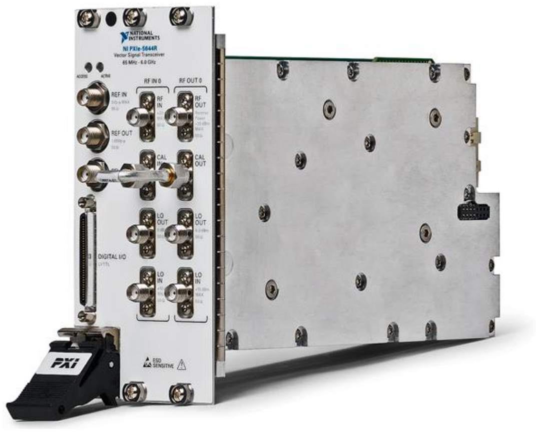


Figure 5. The vector signal transceiver (VST) combines instrument-grade vector signal generation and acquisition into a 3-slot PXI form factor with a large FPGA customizable using LabVIEW.

In addition to the small size and high performance of the RF hardware, the VST is revolutionary in that you can customize the FPGA using the LabVIEW FPGA Module, and you are limited only by your application requirements, not a given vendor's definition of what an instrument should be. This approach greatly increases flexibility and better meets application demands with additional FPGA-based processing and control.

NI RIO FOR COMPACT EMBEDDED APPLICATIONS

NI CompactRIO

CompactRIO is a rugged, reconfigurable embedded system that contains three components: a processor running a real-time operating system (RTOS), a reconfigurable FPGA, and interchangeable industrial I/O modules.

Figure 6. CompactRIO offers a rugged and compact deployment platform with modular I/O for distributed and high-performance embedded applications.

The CompactRIO system includes an embedded controller and reconfigurable chassis. The embedded controller offers powerful, stand-alone embedded execution for LabVIEW Real-Time applications, which benefit from reliable, deterministic behavior. The controller also excels at floating-point math and analysis.

The embedded chassis is at the center of the CompactRIO system because it contains the reconfigurable I/O FPGA core. The FPGA excels at tasks that require high-speed logic and precise timing. It is directly connected to I/O modules that deliver diverse high-performance I/O capabilities.

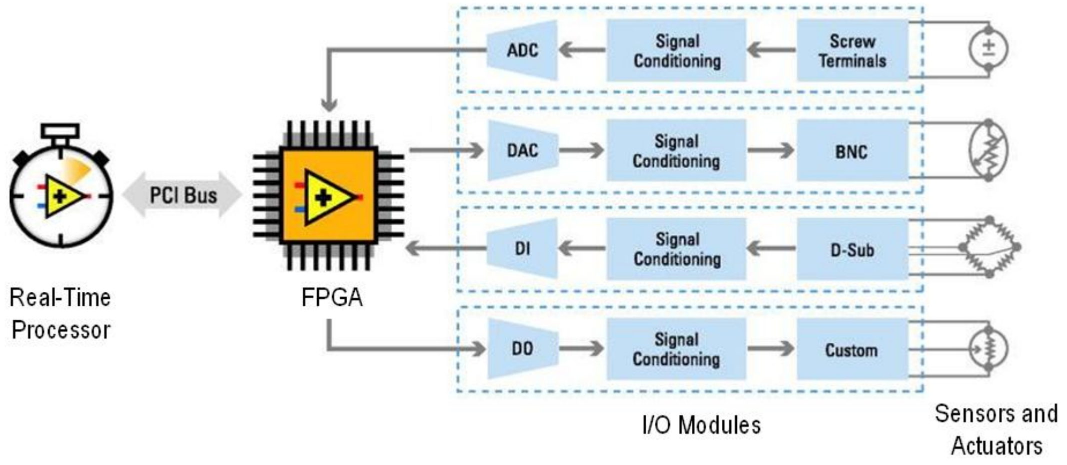


Figure 7. CompactRIO exemplifies the LabVIEW RIO architecture, with a dedicated processor that can run a real-time OS, an FPGA for reconfigurable hardware performance, and a wide variety of modular I/O options.

NI Single-Board RIO

NI Single-Board RIO products are designed for high-volume and OEM embedded control and DAQ applications that require high performance and reliability. Featuring an open and embedded architecture, small size, and flexibility, these commercial off-the-shelf (COTS) hardware devices can help you get custom embedded systems to market quickly.

Figure 8. NI Single-Board RIO takes the embedded RIO platform to smaller form factors and deployment applications.

SELECTING AN FPGA PLATFORM

Your application requirements determine which platform and NI RIO device family that you use. However, you can choose between different options within each device family, including different FPGA sizes and speeds. With LabVIEW FPGA applications, you can compile your design for any NI RIO device, regardless of whether you own the hardware. You also can simulate your design in software to verify its behavior before compilation. Hardware is still required to verify your system with real-world I/O signals at hardware speeds. Users often acquire the largest FPGA device within an NI RIO device family for prototyping and later work to optimize applications for FPGA resource use or performance to use a smaller, more appropriate device.

Applications that perform only basic timing, triggering, and synchronization on the FPGA can typically use a smaller FPGA. Applications requiring additional signal processing, such as control, digital filtering, or complex analog triggering, may need a larger FPGA to implement those operations.

Additional Resources

[1]	What Is PXI? http://www.ni.com/pxi/whatis/
[2]	What Is NI FlexRIO? http://www.ni.com/flexrio/whatis/
[3]	NI R Series Multifunction RIO http://www.ni.com/rseries/
[4]	What Is NI CompactRIO? http://www.ni.com/compactrio/whatis/
[5]	What Is NI Single-Board RIO? http://www.ni.com/singleboard/whatis/
[6]	NI R Series for USB http://sine.ni.com/nips/cds/view/p/lang/en/nid/212013
[7]	Choosing the Right FPGA Hardware http://www.ni.com/fpga-hardware/choose-hardware/

This page intentionally left blank.

HIGH-PERFORMANCE PROGRAMMING WITH THE SINGLE-CYCLE TIMED LOOP

Most of the concepts related to high-performance FPGA programming in LabVIEW involve the effective use of the single-cycle Timed Loop (SCTL). The SCTL is a key LabVIEW FPGA structure that reduces resource use and allows for higher throughput and more precise timing control. The SCTL provides a different programming paradigm that more closely matches the behavior of FPGA circuits and provides more control over the actual implementation of LabVIEW code on the FPGA.

THE SCTL VERSUS STANDARD LabVIEW FPGA CODE

To understand the SCTL, it helps to look at the way standard LabVIEW FPGA code compiles when placed inside a While Loop or For Loop.

When you program with the LabVIEW FPGA Module, the contents of your diagram are translated to hardware, so, in general, each node on your diagram has an equivalent representation on the FPGA as a circuit component. When you place code outside the SCTL, LabVIEW controls the execution of those hardware components as data flows through them. As a result, additional circuitry is needed to make sure that these components execute only when they have valid data at all of their inputs. This model of execution is referred to as structured data flow.

Structured data flow follows a traditional program execution model where a function must have all of its input parameters before it executes, and the caller blocks until the function returns. When executing code on a CPU, the CPU has fixed, general purpose, circuitry that sequentially consumes code and data. However, unlike a CPU, the code on an FPGA itself becomes a highly specialized and parallel circuit, and the data flows through it as electrical signals.

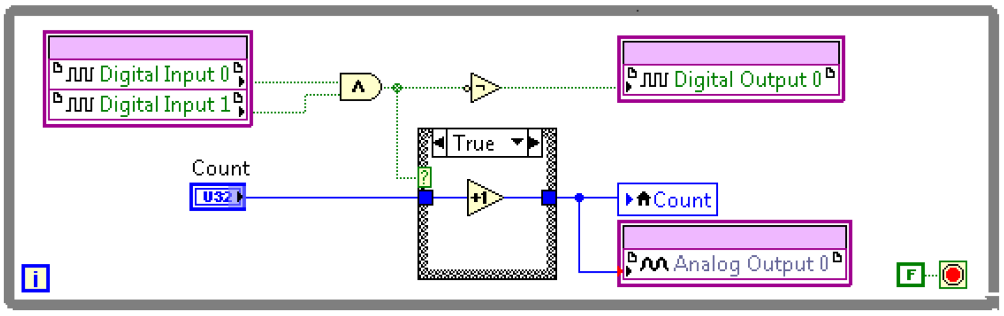


Figure 9. Standard LabVIEW FPGA code is synthesized to circuitry that executes sequentially when data dependencies exist.

When code compiles in a While Loop, LabVIEW inserts hardware registers, or small storage elements, to clock data from one function to the next, thereby enforcing the structured dataflow nature of LabVIEW.

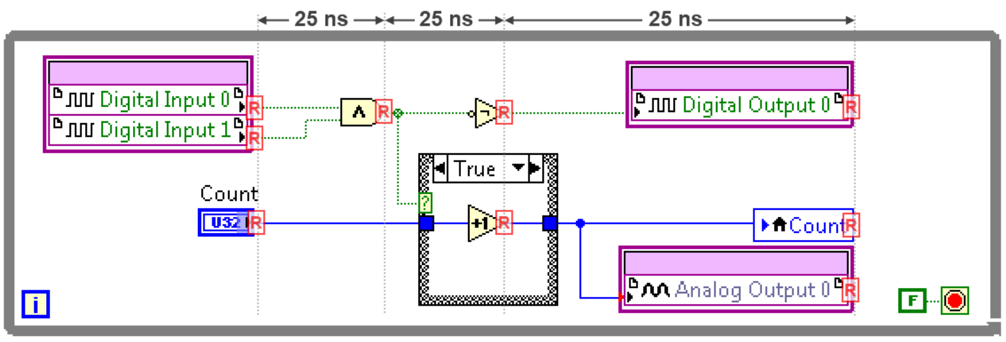


Figure 10. Standard LabVIEW FPGA code uses registers at each function to latch data on every cycle of the top-level clock, typically 40 MHz (25 ns period). Here, hardware registers are represented *by boxes labeled "R."*

These register elements are added to control function execution and latch data on every clock cycle as the data flows from one node to the next. Each node can take one or more cycles to execute. In LabVIEW FPGA applications, code placed outside the SCTL can exhibit varying timing behavior from one compilation to the next or across different versions of the LabVIEW FPGA Module.

The preservation of structured data flow means that, for chains of nodes with data dependencies, only a subset of the nodes is actively executing at a given time while the rest of the circuitry awaits data.

This sequential execution may seem like an inefficient use of the dedicated FPGA circuitry, but it suffices for many applications. The structured dataflow model provides a way for LabVIEW developers to write LabVIEW code that resembles what they create on the desktop, and it allows developers to execute that code on the FPGA without worrying about the implementation details. As a result, LabVIEW FPGA developers enjoy great performance, determinism, and I/O integration

UNDERSTANDING THE SCTL

Understanding the functionality of the SCTL and the code placed inside is key to creating high-performance LabVIEW FPGA applications. The SCTL is a structure unique to LabVIEW FPGA applications. Though the SCTL structure and the code placed inside look nearly identical to the LabVIEW Timed Loop and related internal code, the code placed inside the SCTL executes in a very different manner because it is guaranteed to execute within one clock cycle of a specific FPGA clock. The SCTL represents the following five key concepts:

1. Structure

Structures have specific meaning in LabVIEW. They provide scope and definite transition points for data as it flows through their boundaries. The SCTL is a structure and it follows the structured dataflow model with respect to other structures on the diagram. Specifically, the SCTL cannot begin executing until all of its inputs, or wires entering through tunnels or shift registers, have received data. Similarly, the SCTL does not produce outputs through its tunnels until the code inside has completed execution.

2. Loop

The SCTL is not only a structure that defines how code executes, but it is also a program loop. It repeatedly executes its contents and it obeys While Loop rules in that it must execute at least once, with a stop condition that can be set at run time.

3. Clock

Every SCTL must have a clock. The clock defines the frequency that will be used to drive the circuit synthesized from the SCTL contents. Because all circuitry inside the SCTL shares this same clock, it is also called a clock domain.

The ability to define the SCTL clock is the primary mechanism for using multiple clocks within a design. Different SCTLs may use different clocks that define multiple clock domains. The clock rate and source must be determined at compile time, so it cannot be changed once the design executes.

4. Maximum Iteration Latency

The SCTL not only specifies the clock used to drive the code it encloses but also requires that all enclosed code execute within one clock cycle.

5. Different Execution Paradigm

The execution paradigm of the SCTL differs from standard execution in LabVIEW because all the enclosed code executes in one clock cycle. To meet the iteration latency requirement, LabVIEW FPGA compilation removes flow-control circuitry used to execute structured data flow.

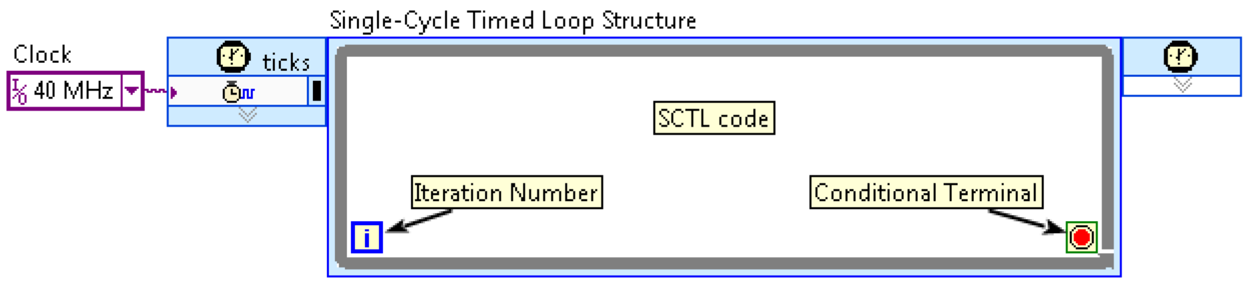


Figure 11. The SCTL is a looping structure that specifies the clock to be used by the code and assumes a maximum iteration latency of one clock cycle.

Removal of the flow-control circuitry not only lowers FPGA resource use but, most importantly, allows the diagram to behave like a concurrent circuit, where data is represented by electrical signals flowing, unrestricted, throughout the SCTL. The signals are transformed by the logic and are latched at certain points on the diagram, such as I/O, storage elements, and controls and indicators.

Because signals are latched at these specific points within the diagram, they must settle to a definite value before the arrival of the next clock edge, which marks the beginning of a new iteration. This latching behavior, combined with the looping model of the loop, creates a synchronous circuit that is synthesized by the FPGA compilation toolchain.

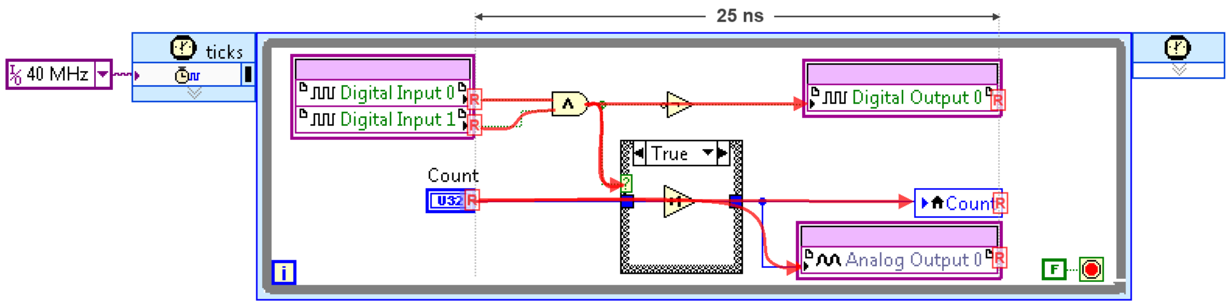


Figure 12. The SCTL represents a synchronous circuit where signals flow through and are latched only at specific points such as I/O, controls and indicators, and other constructs. The SCTL specifies the clock used to drive its contents, and signals must propagate from between registers in less than one clock cycle, which is 25 ns for this example.

The term "synchronous" in synchronous circuits differs from its meaning in standard programming languages when referring to function-call mechanisms. Synchronous circuits are simply circuits driven by a clock, while a synchronous function call blocks the caller's execution until the function returns a result.

BENEFITS OF THE SCTL

The one-cycle iteration latency that the SCTL specifies is passed down to the Xilinx compilation toolchain, along with the generated VHDL code at compile time. The Xilinx compiler synthesizes the SCTL code and treats the one-cycle requirement as a circuit constraint. If the compilation succeeds, the generated design is guaranteed to execute and settle all of its signals within one clock cycle. This behavior differs from that of the LabVIEW Timed Loop on a CPU, where the execution period is not necessarily guaranteed and must be verified at run time.

The iteration latency constraint is an important part of the SCTL paradigm because it differs from standard LabVIEW FPGA code, which often takes more than one cycle to execute a section of code. Because of this timing guarantee, you can use the SCTL as a mechanism to specify timing between events and overall execution rate, as follows:

- The maximum time between two events contained within the same iteration of the SCTL is constrained to a single loop period. This is useful if you are trying to specify a maximum latency between those events.
- You can determine, limit, or guarantee a minimum number of cycles between events in successive iterations. This is useful for measuring and controlling events in applications such as digital communication protocols.
- If you know that the loop rate is guaranteed by the compiler, you can prescribe throughput. The loop, combined with the number of samples processed per iteration, provides a measure of throughput.

RESTRICTIONS OF THE SCTL

Even though SCTL code may look similar to other LabVIEW code, the programming and execution paradigm requires an adjustment to the way you think and how you use different constructs to achieve better performance.

FUNCTIONS AND STRUCTURES SUPPORTED INSIDE THE SCTL

The iteration latency requirement of the SCTL limits LabVIEW functions and structures that are supported when placed inside an SCTL.

Functions that require multiple cycles to execute inside an SCTL also require extra handshaking signals to notify other logic in the SCTL when new data can be consumed or is available. These functions, along with the handshaking mechanisms, are discussed in the Integrating High-Throughput IP chapter. Functions that take multiple cycles to execute but do not have these handshaking signals are not supported inside the SCTL. These functions include:

- Certain arithmetic functions, such as **Quotient/Remainder**
- **Loop Timer** and **Wait** functions
- Some analog input and analog output I/O nodes
- Multiple instances of non-reentrant subVIs (refer to the Timing Optimization Techniques chapter for more information on the effect of using non-reentrant VIs)
- Most functions that deal with floating-point data types

Loop structures, such as the For Loop and the While Loop, are also not allowed inside the SCTL. This presents a challenge if you are trying to create a subVI for use within the SCTL that would have required some sort of iterative algorithm. In this case, you must create your subVI explicitly for use within the SCTL and store and track values across iterations using Feedback Nodes.

The code in Figure 13 is a basic attempt to sort a fixed-size array in place. It demonstrates in general how you might need to adapt code for use in the SCTL.

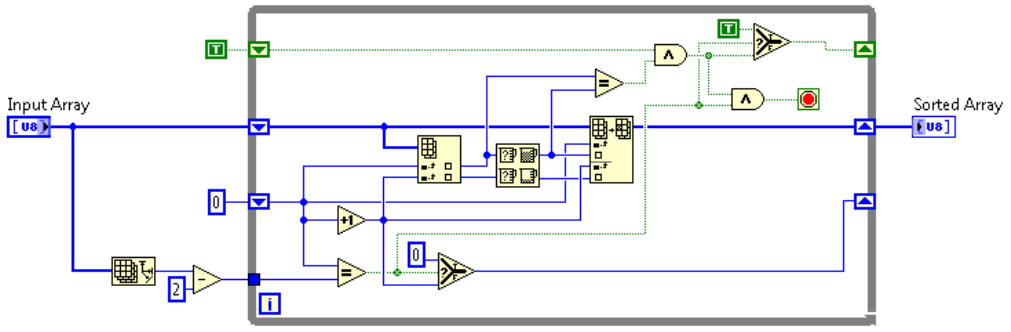


Figure 13. This diagram shows an example of an iterative algorithm, written in standard LabVIEW FPGA code, which can sort an array in place.

The SCTL version moves the registers inward and adds circuitry to keep track of the indexes as it traverses the array, as shown in Figure 14.

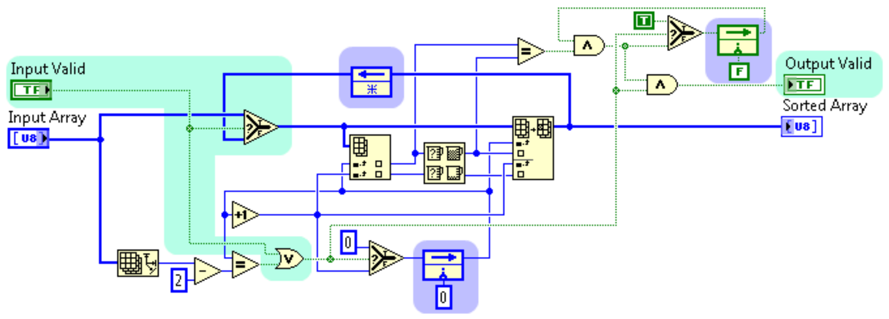


Figure 14. The SCTL version of the array-sorting algorithm is called multiple times for the same array input, so it replaces shift registers with Feedback Nodes (highlighted in light blue), adds code to keep track of the indexes as it traverses the array, and adds handshaking signals (highlighted in light green) to know when new data arrives and tell downstream blocks when it can consume the sorted array output.

Notice in Figure 14 that the output of the subVI should not be consumed until the algorithm has completed, but the subVI must be called several times to complete. You must add a signal to the output to indicate when the array has been sorted. Array sort is an example of a function that requires multiple cycles to execute and the requirement of handshaking signals, which is explained in the Integrating High-Throughput IP chapter.

Functions that require loops, including nested loops, for iterative calculations are often referred to as algorithmic VIs. These VIs can be challenging to port and optimize for LabVIEW FPGA use. NI has created a LabVIEW FPGA add-on specifically to address this need, LabVIEW FPGA IP Builder. You can use it to work with large arrays, nested loops, and floating types and to generate optimized IP for use within the SCTL.

TRANSFERRING DATA BETWEEN SCTL STRUCTURES

Passing data between different SCTL clocks can be a challenge. As discussed in more detail in the Data Transfer Mechanisms chapter, loops operating at different rates, or with an unknown or variable phase relationship, output and latch data at different times. As a result, data loss, duplication, or corruption can occur if you don't use safe transfer mechanisms, such as block memory first-in first-out (FIFO) constructs, or fail to add your own synchronization around unsafe ones.

SUCCESSFUL COMPILATION

Your application might require very high throughput or low latency. A common way to achieve these goals is to increase the SCTL clock rate. As clock rate and resource use increases, the toolchain works harder to maintain the strict iteration latency constraint, which often leads to longer compile times and lower compilation success rates.

The time it takes for signals to settle in the synthesized circuit is also known as propagation delay. Propagation delay has two components: routing delay and logic delay.

Routing delay refers to the amount of time it takes signals to travel between circuit components. These signals travel at near the speed of light, but the clock rates are also high enough that these delays must be taken into account. The traces and circuits are also not ideal conductors, so there are small loads that make digital signals lose their characteristic pulse shape, making it harder to guarantee their integrity across long traces as the clock rate increases.

Logic delay refers to the time it takes for signals to propagate through logic components, such as the time it takes for signals to go from the inputs of a comparison function to its output.

As a diagram grows, logic is added and related components might be placed farther apart, leading to larger propagation delays.

The data path with the longest propagation delay within an SCTL is referred to as the critical path. If the critical path has a propagation delay longer than the SCTL clock period, the compilation fails. LabVIEW FPGA offers a way to map compilation results back to diagram components so that you can attempt to optimize the failing path.

Optimization techniques vary depending on your application goal. Throughput, latency, and resource use are the most common design factors in high-performance LabVIEW FPGA applications, and you must consider the trade-offs when choosing between them. For example, you often can increase throughput at the expense of resources and latency. You also can reduce latency by parallelizing your code, when the algorithm permits it, which leads to increased FPGA resource use. Working through these trade-offs requires an iterative approach to the optimization process. The next few chapters provide a discussion of the various approaches to optimizing throughput, latency, and FPGA resource use.

Additional Resources

[1] Using NI LabVIEW FPGA IP Builder to Optimize and Port VIs for Use on FPGAS
http://www.ni.com/white-paper/14036/en/

This page intentionally left blank.

THROUGHPUT OPTIMIZATION TECHNIQUES

Throughput requirements are common in signal, image, and general data processing applications. FPGAs are typically used for inline processing of the data stream, so they must be able to sustain, sometimes indefinitely, a desired throughput rate, which is measured in samples or amount of data per unit of time. For the purpose of this chapter, assume that throughput is measured in number of samples processed or transferred per second.

Throughput is the result of three factors:

1. The rate of the clock you are using to drive your design (cycles/time)
2. The number of samples that the IP accepts per call (samples/call)
3. The number of cycles that must elapse before your algorithm can be called again (cycles/call); this is also referred to as the initiation interval (II) or initiation period

These three factors combine to provide samples/time per the following definition:

$$Throughput = \frac{Clock\ Rate \times Samples\ per\ Call}{Initiation\ Interval}$$

This equation shows that you can increase throughput by:

- Increasing the clock rate
- Increasing the number of samples processed per call
- Decreasing the initiation interval

INCREASING THE CLOCK RATE

The most direct way to increase component throughput is to increase its clock rate, either through the top-level diagram clock or the SCTL clock. When successful, an increase in clock rate quickly leads to a linear increase in throughput, assuming that all other factors remain the same. You should, therefore, always consider increasing the clock rate if possible.

You should also be aware that increasing the clock rate does have other implications. As discussed in previous chapters, higher clock rates make the compilation process harder and longer, eventually making it impossible to generate a circuit that meets the requested timing constraints on the target platform. The Critical Path Reduction section of this chapter provides tips on how to shorten your design's critical path so that you can compile at even higher rates.

There are cases where you may not be able to increase the clock rate of your design. Parts of your design may require IP to execute with a certain clock relationship with respect to the rest of the system. For example, some I/O blocks must execute in the same clock domain as the I/O subsystem. External DRAM interfaces using the Component Level IP (CLIP) integration mechanism may need to be accessed from the clock domain of the DRAM interface. You must, then, split your design into multiple SCTLs, using the appropriate clock domain for each.

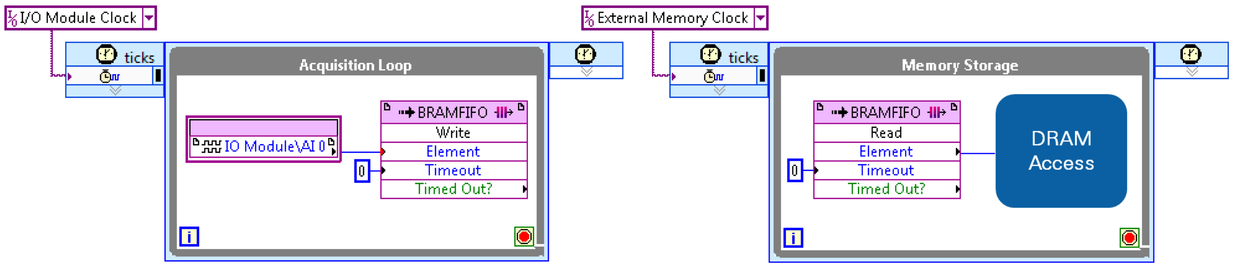


Figure 15. Code that interacts with I/O or accesses DRAM through CLIP must reside in the appropriate clock domain. If an application performs both tasks, you are forced to split it into multiple SCTLs.

Possibly only parts of your design require a high clock rate. Consider splitting your design into multiple loops if sections of code can achieve the desired throughput at lower clock rates because this can ease and speed up the compilation process.

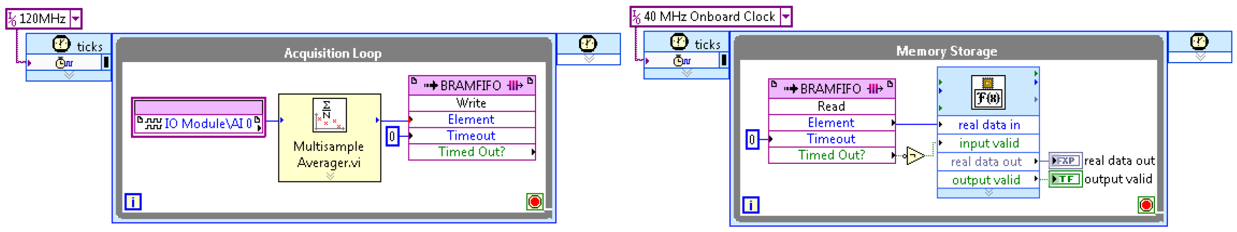


Figure 16. Parts of your design may have different throughput requirements. Splitting such designs into multiple SCTLs makes it easier for the compiler to achieve the desired clock rates where it matters.

You must be careful when passing data between loops running in different clock domains. Refer to the Data Transfer Mechanisms section for techniques to help in this area.

INCREASING THE NUMBER OF SAMPLES PROCESSED PER CALL

The number of data samples processed by each call to a VI, or iteration of an SCTL, is usually a fixed characteristic of your design. You can, for example, implement a parallel version of an algorithm that processes multiple samples from the same channel, per call, which increases throughput while keeping the clock rate and initiation interval constant.

The potential for parallelization depends on the nature of the algorithm. There are cases when the operation is explicitly parallelizable across independent data sets, such as when you are performing the same operation on multiple I/O channels. In this case, because of the parallel nature of the FPGA and the restriction of not using loop structures within the SCTL, you would typically duplicate your logic so that each channel is processed independently. This comes at the expense of additional resources. An example of an application in which LabVIEW can process channels independently would be trying to determine the maximum value for each channel.

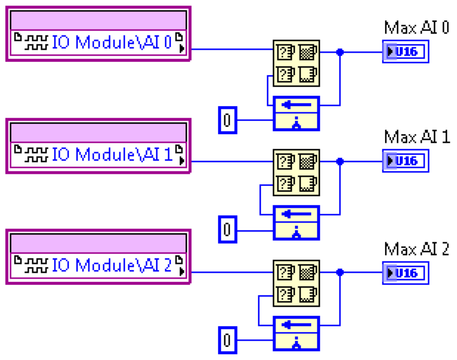


Figure 17. On an FPGA, you can make LabVIEW operate on multiple channels concurrently by replicating your code per channel.

A more interesting case occurs when samples from the same data set can be processed independently, as in the case of processing multiple samples from the same I/O channel (data stream) simultaneously. This is possible only if the operation does not require knowledge of previous channel samples. One example of this is scaling channel samples by some factor.

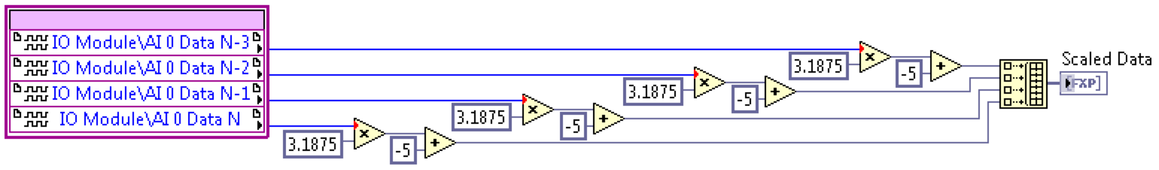


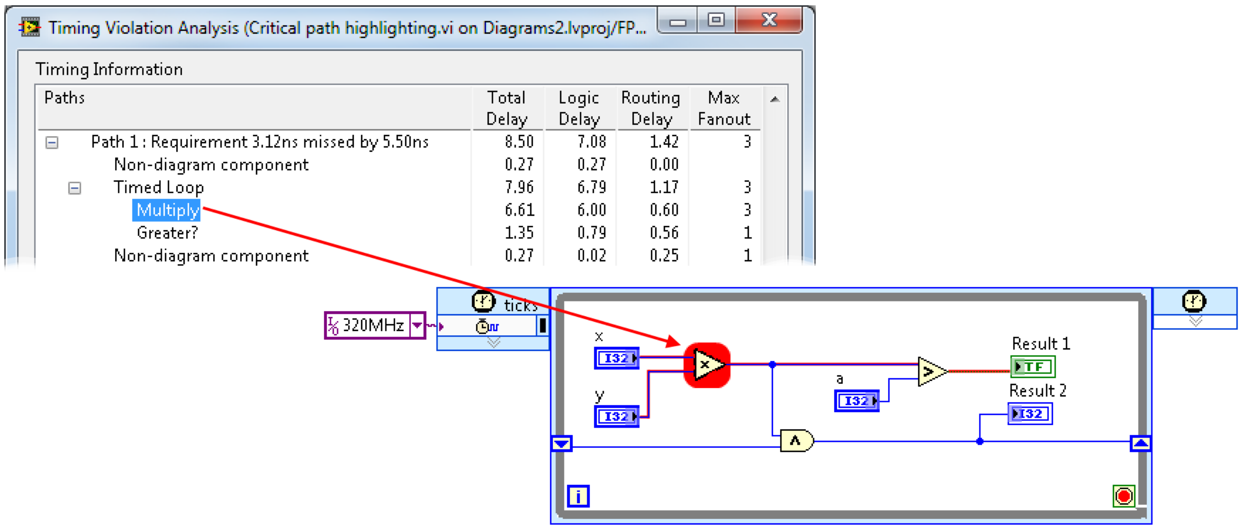
Figure 18. Some hardware modules may generate multiple samples per FPGA clock cycle, requiring a parallel approach to handle them.

Some NI FlexRIO adapter modules generate multiple channel samples per FPGA cycle, so processing every sample of a single channel requires this kind of parallelization. These functions are referred to as multisample/multicycle IP.

FPGAs are inherently parallel devices, making parallel processing of multiple samples or data sets an attractive option for increasing throughput. FPGAs are also constrained with respect to resources, and parallelization usually leads to a linear increase in resource use. Explicitly parallelizing your code by making copies of it can lead to code that is difficult to read and maintain. Carefully consider when parallelization, or processing more samples per call or iteration of the loop, is a good option. Additional parallelization techniques are discussed later in this document.

CRITICAL PATH REDUCTION

When optimizing throughput, the ultimate goal of reducing the critical path of the SCTL is to increase its clock rate. If the SCTL critical path has a propagation delay longer than the SCTL clock period, the compilation fails. LabVIEW highlights the critical path when a compilation failure occurs. You can use that information to shorten the overall propagation delay of the path and recompile your design at the same or higher clock rates.


Timing Violation Analysis (Critical path highlighting.vi on Diagrams2.lvproj/FP...

Timing Information

Paths	Total Delay	Logic Delay	Routing Delay	Max Fanout
Path 1 : Requirement 3.12ns missed by 5.50ns	8.50	7.08	1.42	3
Non-diagram component	0.27	0.27	0.00	
Timed Loop	7.96	6.79	1.17	3
Multiply	6.61	6.00	0.60	3
Greater?	1.35	0.79	0.56	1
Non-diagram component	0.27	0.02	0.25	1

Figure 19. LabVIEW FPGA parses through the FPGA compilation results and identifies, directly on your diagram, the paths that fail to meet timing constraints so that you can take steps to optimize them.

PARALLELIZING OPERATIONS

When reducing the length of the critical path, first remove unnecessary dependencies. Look for anything that requires sequential execution due to data dependencies and question whether the operations really need to happen in order. Sometimes you can remove operations from the critical path by branching your wires and performing different operations on multiple copies of the data at the same time. Then you can combine the results later on, as shown in Figure 20.

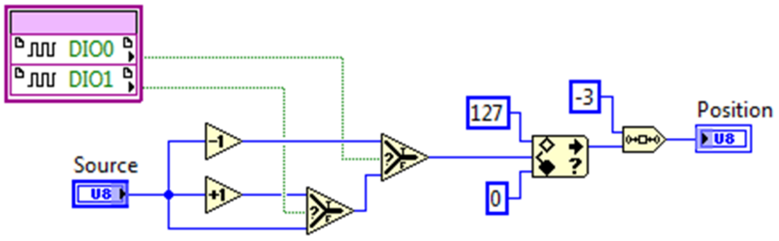


Figure 20. When resources allow, you should explicitly parallelize independent operations on the diagram to take advantage of the FPGA as a dedicated hardware implementation of your code.

This approach depends on the nature of the operations. You may be able to apply other coding patterns that increase parallelism, which in turn reduces the overall latency of the code, as discussed in the Timing Optimization Techniques chapter.

DATA-TYPE OPTIMIZATION

The number of bits used to represent data has a significant impact on the propagation delay along the critical path. As a result, using the smallest data type that meets your accuracy and precision requirements can also help you achieve the desired SCTL clock rate.

The FPGA requires more logic to process larger types and grows linearly with the number of bits. More electrical paths are also added to carry those bits from one logic section to the next, and those paths may have different lengths. Timing constraints must be relaxed to allow for signals to settle across all paths, which keeps the circuit from running at higher rates.

Data-type optimization is an effective way to decrease resource use, increase the achievable clock rate, and, as a result, increase design throughput. Data-type optimization is discussed in more detail in the Resource Optimization Techniques chapter.

PIPELINING YOUR DESIGN

Pipelining is the most common technique you can use for increasing code throughput. It involves the insertion of registers along the critical path of the SCTL to break it into shorter, concurrent sections of code. The shorter sections of code take less time to execute, which allows you to increase the SCTL clock rate.

In the following example, note the single data path, which is also the critical path. Assuming you want to execute the SCTL at 200 MHz, the diagram must be able to propagate its values, from **Input** to **Output,** and allow them to settle, within 5 ns.

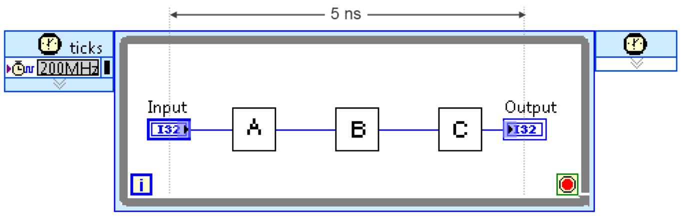


Figure 21. This code sequence must execute in less than 5 ns when the SCTL is configured to use a 200 MHz clock.

Assuming that subVIs **A**, **B**, and **C** take 3, 4, and 5 ns, respectively, to execute on the FPGA, the total time of this sequence is 12 ns, which limits the achievable rate to approximately 83 MHz. A compilation at 200 MHz fails in this case.

You can optimize the above VI by adding registers in the form of Feedback Nodes, typically in their forward representation, to allow each subVI to start working on a data value as soon as the loop begins execution. The feed-forward nodes help break up the execution into concurrent pieces by latching and retaining the values across iterations, thus splitting the critical path into three segments. Shortening the critical path allows you to increase the clock rate, and, as in this example, allows the clock to run at the desired rate of 200 MHz for a 2.4X increase in throughput.

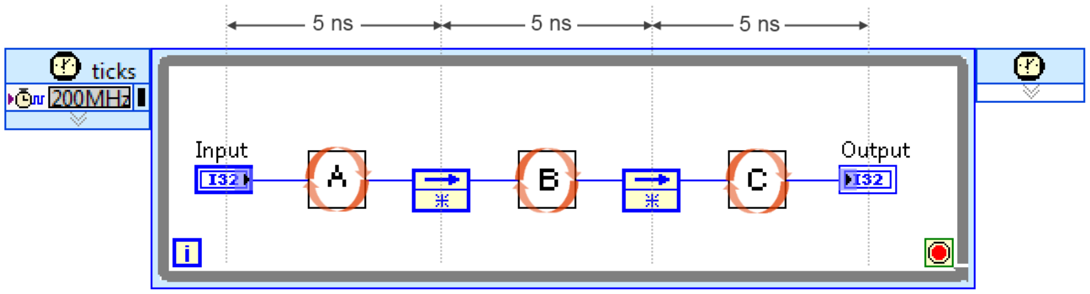


Figure 22. Pipelining allows the VIs to execute concurrently. The 5 ns latency restriction now applies to each segment of the path, making it possible to run the loop at 200 MHz.

Pipelining can increase processing latency. In the previous example, a sample traveling from input to output in one cycle of the original diagram now takes three cycles. Since part of the goal is to break up the critical path to increase clock rate, some of the lost latency is regained through a higher clock rate. However, you cannot completely offset latency with higher clock rates due to additional logic delays and imperfect partitioning of the critical path. In practice, pipelining trades off some latency for higher clock rates as a way to achieve higher throughput.

In the case of the above example, the original latency at the maximum expected clock rate of 83 MHz would have been:

$$Original\ latency = \frac{1\ cycle}{83\ MHz} = 12\ us$$

The pipelined design sees a 25 percent increase in latency, assuming it takes 3 cycles at 200 MHz:

$$Pipelined\ latency = \frac{3\ cycles}{200\ MHz} = 15\ us$$

Some streaming applications may be affected by this trade-off between throughput and latency, so it is important to keep in mind.

You can study the execution order using a timing diagram, shown in Figure 23. Samples are represented by the subscript number. Note that subVI A can start working on the second sample while subVI B works on the output of subVI A from the previous cycle.

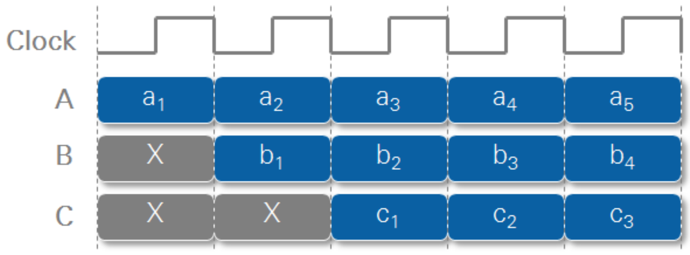


Figure 23. The timing diagram shows how different parts of the SCTL operate on different samples within the same iteration. This allows you to clock the diagram at higher rates, but it takes more cycles to propagate samples from input to output, so you must deal with data validity as the pipeline starts up.

You may notice that B operates on some unknown value during the first cycle, which can generate invalid data. C also operates on some unknown value during the first two cycles. Pipelining introduces the challenges of handling the validity of the data on the diagram.

In a pipelined design, the pipeline must be "filled" or "primed" before it produces any valid output. This process takes multiple clock cycles. Another implication of pipelining is that downstream blocks, especially those that cause side effects, such as I/O, updates to stored values, or communication, should be notified about the invalid data while the pipeline fills up.

The simplest approach to notify downstream blocks of invalid data is to use a data-valid signal that travels along with the data, as shown in Figure 24.

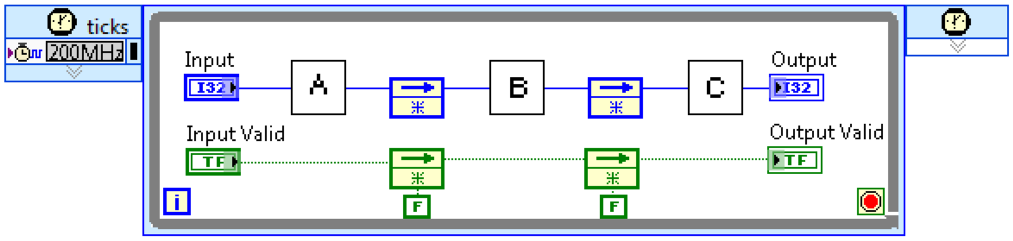


Figure 24. A pipelined design takes multiple cycles to generate a valid sample, so you likely need to accompany the data with a Boolean signal to indicate when the output can be consumed by downstream blocks.

Diagrams can be arbitrarily complex, with data paths branching and merging at different points. If data coherence across processing branches is important to your design, you need to balance the delay across them so that the functions process samples with the appropriate cycle relationship. Figure 25 shows an example of how to balance the delay across code branches by using a single register node and setting a delay value of 2 in the configuration dialog.

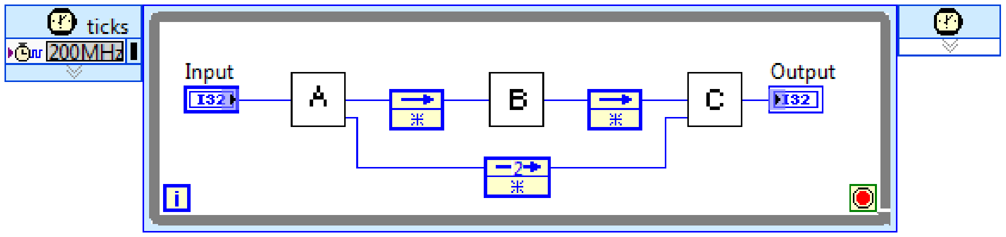


Figure 25. Feedback Nodes can buffer and delay data processing by more than one sample. This can help balance data delays across multiple processing branches.

Signals that represent the validity of the data also must be forked and joined as they follow the data flowing through the diagram. The use of functions that can take multiple cycles to consume and generate valid data leads to the more complex concept of flow control, in which data validity must be explicitly handled on the diagram. Flow-control techniques, such as four-wire handshaking or AXI protocol, are discussed in the Integrating High-Throughput IP chapter.

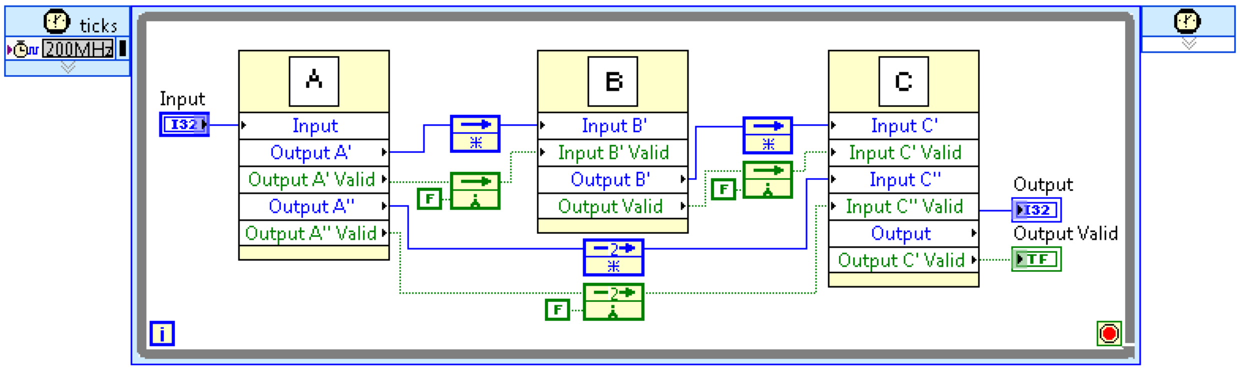


Figure 26. Note the data validity signal that has been added to the branched processing chain to make sure each subVI operates on valid data, with the appropriate delay relationship, and signaling data validity at the output.

DECREASING THE INITIATION INTERVAL

When the diagram contains IP that takes multiple cycles to execute per input, also referred to as multisample/multicycle IP, you can increase throughput by lowering the initiation interval of such IP. Lowering the initiation interval of a piece of IP equates to transforming the code so that it can accept samples more often, with fewer cycles between calls.

Figure 27 shows a sequence of code that processes a sample every two cycles. The code includes three stages, represented by subVIs A, B, and C. Pipelining registers have already been added to increase throughput through higher SCTL clock rates.

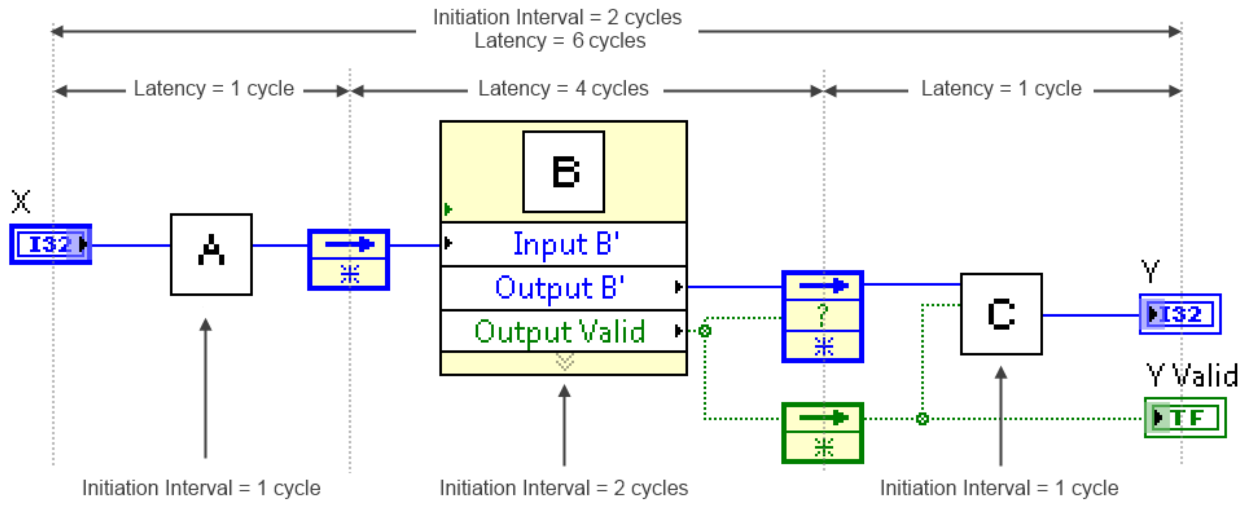

Initiation Interval = 2 cycles
Latency = 6 cycles
Latency = 1 cycle
Latency = 4 cycles
Latency = 1 cycle
X
A
B
Input B'
Output B'
Output Valid
C
Y
Y Valid
Initiation Interval = 1 cycle
Initiation Interval = 2 cycles
Initiation Interval = 1 cycle

*Figure 27. This code has an initiation interval of two cycles because of subVI **B**. When a code path processes one input at a time, its minimum achievable initiation interval is the maximum initiation interval of the functions in the path. The latency of the code path is the sum of the individual latencies in the path. For simplicity, the diagram omits the code required to signal to downstream blocks about the validity of the data while the pipeline initializes.*

An important characteristic of this code example is that subVI B performs some operation that requires two cycles for every input, which translates to an initiation interval of two cycles. Even though the initiation interval for subVIs B and C is one cycle, the minimum achievable initiation interval of the whole code path sequence is two cycles, equal to the highest initiation interval in the chain. This has two implications:

1. This sequence of code effectively ignores every other sample coming in through the X terminal. Upstream code must already take this into account or must be informed when it can provide new inputs using handshaking signals, as discussed in the Integrating High-Throughput IP chapter.
2. The Output Valid signal of subVI B must be propagated to indicate to downstream code blocks the validity of its output.

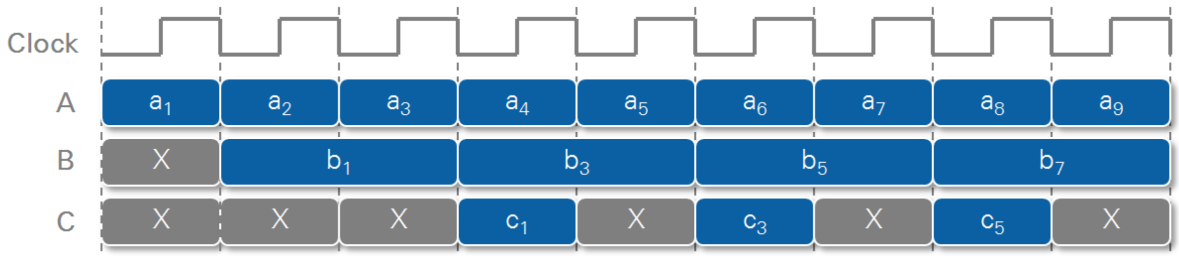


Figure 28. Latency can be measured as the number of cycles it takes a sample to go from input to output, while initiation interval describes how often, in cycles, a piece of IP can receive a new input, which is inversely proportional to throughput. The timing diagram for the previous example shows that the code has a latency of four cycles and an initiation interval of two cycles.

Notice that the latency of this diagram is different from its initiation interval. In this case, any given sample takes four cycles to travel from input X to output Y, while the diagram can accept new inputs at X every other cycle. A pipelined design accepts data after fewer cycles than the amount it takes for a given sample to go from input to output, which is one way to measure its latency.

A manufacturing assembly line is an example of a pipelined process, where multiple stations assemble different parts of a product and work on many different orders at a given time. Adding stages to a pipelined process means splitting it into simpler stages that can complete faster and increase overall throughput, at the cost of more workers and assembly equipment.

A ten-stage manufacturing assembly line, for example, may start the assembly of a new device every minute, for a throughput of one device per minute. The individual devices need to go through ten assembly stages, so assuming a duration of one minute per stage, the overall device production latency would be 10 minutes. Sometimes it is impossible to split a pipeline stage into smaller stages. Assuming that you cannot further divide the pipeline, as with subVI B in Figure 29, you may be able to duplicate and overlap its execution to increase the number of samples the code can process per cycle. Note how the previous example has been modified. Notice the use of the First Call primitive to delay the execution of the second copy of subVI B by one cycle.

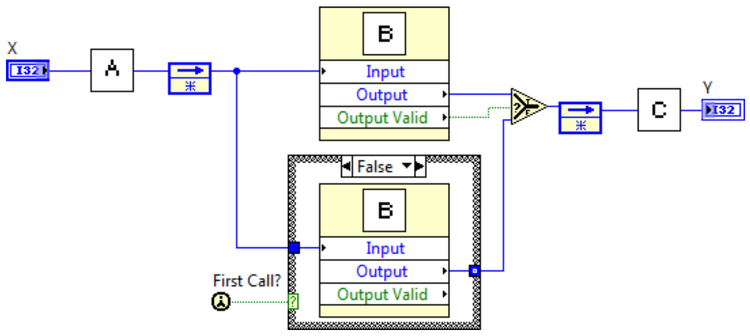


Figure 29. Duplicating subVI B allows the code to process one sample per cycle by operating on two samples concurrently, doubling the overall throughput of the code.

The modified diagram now can accept a sample on every cycle, which doubles its original throughput at the expense of more resources. Note the assumption that operations in subVI B can be performed independently on each sample. Operations that hold state or require strict sample ordering cannot be parallelized in this manner.

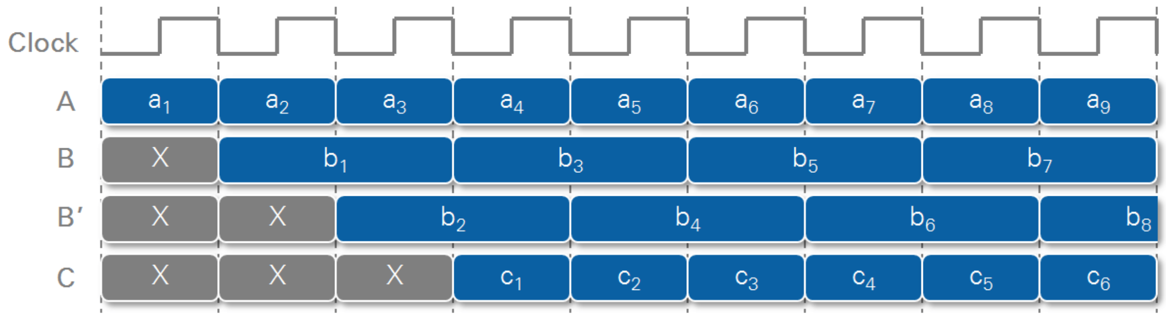


Figure 30. Duplicating subVI B and staggering its execution lowers the overall initiation interval to one cycle. The code can now accept new inputs on every iteration of the SCTL, which doubles its throughput for the code example.

This technique includes a combination of pipelining and duplication. Code and state are duplicated to operate concurrently on multiple sequential samples. This is different from the parallelization techniques covered in previous sections, where parallel code accepts multiple samples per loop iteration, or where different operations are performed on the same sample in parallel. The example above accepts one sample per loop iteration, but it processes two independent samples concurrently, each with a different degree of completion.

INTEGRATING HIGH-THROUGHPUT IP

The previous chapter covered general optimization patterns for writing your own processing IP. You should use existing IP whenever possible to save the time it takes to create, verify, and maintain your own versions. The concepts in this section cover the common sources of LabVIEW FPGA IP and how you can integrate them into your application.

RECOMMENDED SOURCES OF LabVIEW FPGA IP

There are multiple sources of IP for LabVIEW FPGA. Consider these sources in the following order:

1. LabVIEW FPGA Palettes

   The LabVIEW FPGA palettes contain IP that has been validated and optimized by National Instruments for use on NI RIO devices. You can augment the palettes with various application- or market-specific toolkits to avoid building specialized IP.

2. Integrated IP Generators

   IP generators are tools that can take a generic algorithm or function and generate IP that is specifically tailored to your performance or functional requirements. Examples of IP generators include:

   a. Xilinx CORE Generator system
   b. LabVIEW Digital Filter Design Toolkit and other generators
   c. LabVIEW FPGA IP Builder module

3. LabVIEW FPGA Community

   The community of LabVIEW users offers a place to share and reuse LabVIEW FPGA IP. Examples of community sites include the LabVIEW Tools Network, IPNet, and the NI FlexRIO and Software-Designed Instruments IP community.

4. Hardware Description Language (HDL) IP

   LabVIEW FPGA offers mechanisms for integrating IP written in, or derived from, HDL such as VHDL and Verilog.

The following sections cover IP examples from these sources as well as related IP integration techniques.

LabVIEW FPGA HIGH THROUGHPUT FUNCTION PALETTES

The high throughput FPGA function palettes provide IP that is especially tuned for high-performance applications. Although some of the nodes in these palettes can be used outside the SCTL, they are mostly intended for use within the SCTL. Most of the functions in the high throughput palettes provide a configuration dialog, where you can tweak behavior, resource use, and performance characteristics.

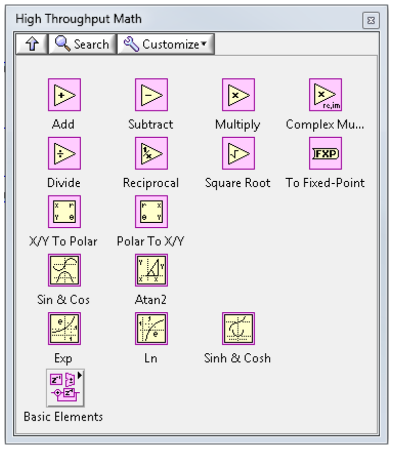


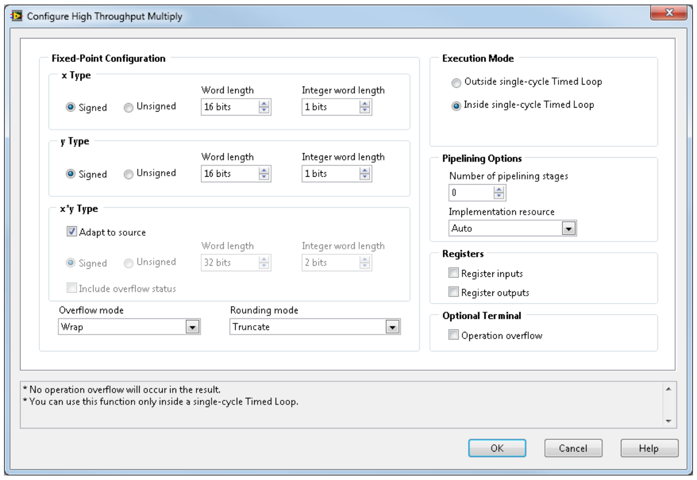
Figure 31. The High Throughput Math palette offers functions specifically optimized for use within the SCTL. It also provides subpalettes that expose specialized FPGA resources such as the Discrete Delay and DSP48 nodes.

The high throughput functions leverage specialized FPGA components and resource locality, which reduces logic and routing resource use. They also use pipelining techniques to achieve higher clock rates.

The documentation for many of these functions contains details about how to configure them to balance throughput, latency, resource, and numerical precision trade-offs. Below is a discussion of a couple of these nodes that shows how their configuration may affect performance and how you can use the nodes to leverage specialized FPGA resources.

THE HIGH THROUGHPUT MULTIPLY FUNCTION

This node computes the product of fixed-point number pairs. Figure 32 shows the configuration dialog for the node. This configuration dialog also provides feedback on the expected performance and numerical behavior for the selected configuration.


Configure High Throughput Multiply

Fixed-Point Configuration

x Type: [✓ Signed] Unsigned; Word length 16 bits; Integer word length 1 bits

y Type: [✓ Signed] Unsigned; Word length 16 bits; Integer word length 1 bits

x*y Type: [✓ Adapt to source]; [✓ Signed] Unsigned; Word length 32 bits; Integer word length 2 bits; Include overflow status

Overflow mode: Wrap

Rounding mode: Truncate

Execution Mode: Outside single-cycle Timed Loop [✓ Inside single-cycle Timed Loop]

Pipelining Options: Number of pipelining stages 0; Implementation resource Auto

Registers: Register inputs; Register outputs

Optional Terminal: Operation overflow

* No operation overflow will occur in the result.
* You can use this function only inside a single-cycle Timed Loop.

Figure 32. The High Throughput Multiply function presents a detailed configuration dialog you can use to balance functional and performance needs.

These settings affect node performance:

Fixed-Point Configuration

The performance of the node, and most logic and arithmetic operations, as discussed in the Resource Optimization Techniques chapter, is greatly affected by the width of the operands and output.

Overflow Mode

Overflow conditions can occur when the type of output terminal may not be large enough to hold the result of multiplying the two inputs. Additional circuitry is necessary to detect and report overflow conditions as well as to saturate numbers when overflow conditions occur.

Refer to the product documentation for more details on overflow conditions and when to use each setting. In general, using Wrap mode provides higher performance and consumes fewer resources. This is an example of a possible trade-off between numerical precision and performance.

Rounding Mode

Rounding occurs when the expected precision of a value is greater than the precision of the type that represents it. LabVIEW coerces the value for you, which can result in a loss of precision.

Additional logic is needed to perform rounding, which, in turn, increases resource use and limits performance. In general, it is preferable to use Truncate mode when possible because it simply discards the least significant bits to accommodate for the lower precision of the output. Note, however, that this can bias numerical results toward negative infinity. This numerical bias due to rounding mode is another example of the trade-off between numerical precision and performance.

Refer to the Resource Optimization Techniques chapter and the *Using the Fixed-Point Data Type (FPGA Module)* topic in the LabVIEW product documentation for more information on the effect of rounding and overflow on FPGA performance, resource use, and numerical precision.

Pipelining, Registers, and Implementation Resources

When configured for use inside an SCTL, the multiplier can implement internal pipelining and add registers to its input and output terminals, which breaks up the processing path to allow for compilation at higher rates.

You also can determine the type of resource that the compiler uses by specifying if it should automatically choose between lookup tables (LUTs) and an embedded multiplier (DSP48 block) to implement the function. DSP48 blocks are specialized Xilinx FPGA elements, while LUTs are general-purpose logic blocks. This is an example of how high throughput nodes can leverage specialized resources or help you balance resource use across resource types. Refer to The DSP48 Node subsection later in this chapter for more information on the DSP48. The Resource Optimization Techniques chapter explains in more detail the role of LUTs and specialized FPGA resources.

As shown in Figure 33, the maximum attainable performance varies depending on the implementation resource, the number of pipeline stages, and the data-type width.

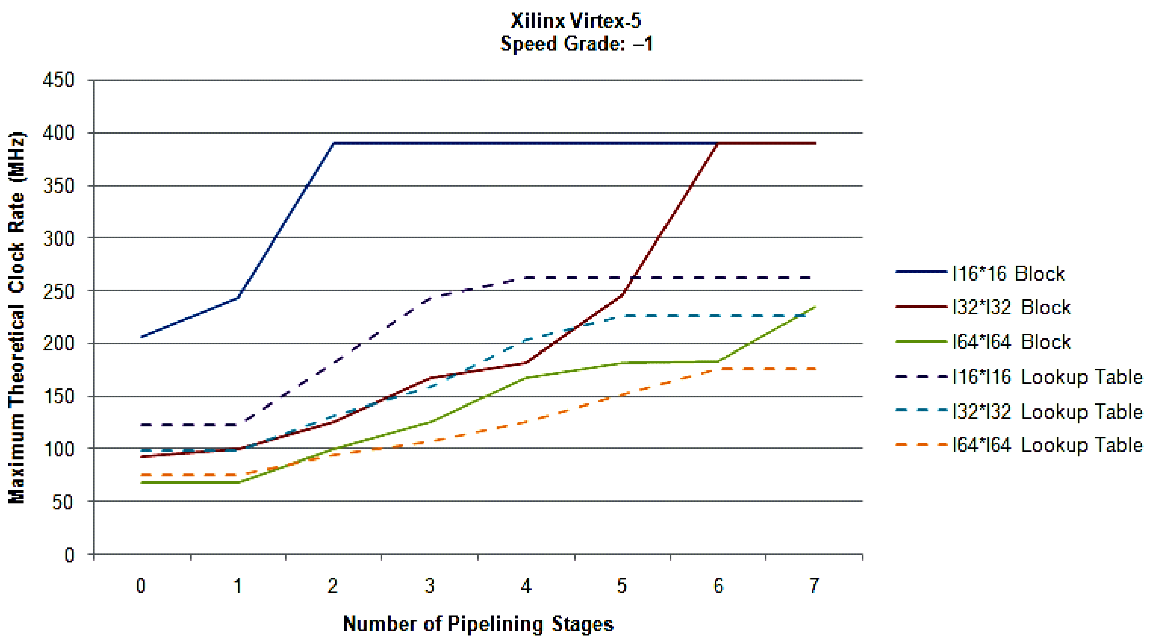


Figure 33. The maximum clock rate for a multiplier function depends on the data type, the number of pipeline stages, and the type of resource used to implement the function. Narrower types, combined with the use of the DSP48 block, offer the best performance.

IP HANDSHAKING PROTOCOLS

When configured for use within the SCTL, nodes on the High Throughput Math Functions palette can take multiple cycles to initialize or generate output, which drives the need for handshaking protocols on their interface.

Handshaking protocols are a collection of signals that inform upstream or downstream blocks in processing chains, such as in the case of DSP and image processing applications, about the validity of their outputs or their willingness to receive new inputs.

As discussed in the Throughput Optimization Techniques chapter, handshaking is necessary in an SCTL because multicycle nodes need more than one cycle to compute valid data, and the SCTL forces these nodes to return some data every clock cycle. Therefore, multicycle nodes do not return valid data every clock cycle. To ensure the numerical accuracy of an algorithm, nodes that depend on this data must know whether the data is valid.

Functions that can execute in one cycle must also be able to propagate handshaking information upstream. This is necessary because downstream functions may not be able to receive data at all times during execution, so the inflow of data must be controlled by upstream code.

THE FOUR-WIRE HANDSHAKING PROTOCOL

Functions on the High Throughput Math Functions palette support a handshaking protocol for building processing chains inside the SCTL. Because this protocol involves four terminals, handshaking in FPGA functions and VIs is sometimes known as the four-wire protocol. This protocol involves the following terminals:

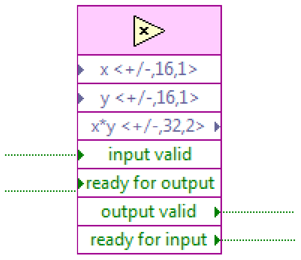


Figure 34. The High Throughput Multiply function is an example of a function that offers a four-wire handshaking interface. You use the four signals to control the flow of valid data inside the SCTL.

INPUT VALID

The handshaking function needs to know about the validity of incoming data. This signal specifies if the data passed to the function is valid before it executes. If upstream code also implements the four-wire protocol, you can connect the upstream **output valid** signal to the **input valid** terminal.

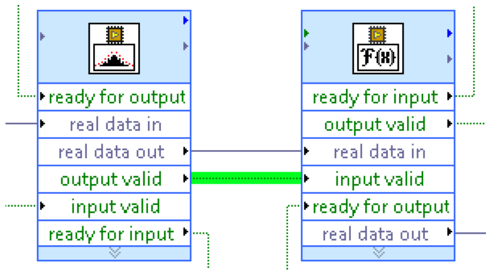


*Figure 35. Two functions supporting the four-wire handshaking interface signal the transfer of valid data by connecting the **output valid** signal of the upstream function to the **input valid** input of the downstream one.*

Incoming data might be known to be valid on every cycle. In this situation, you can wire a **TRUE** constant to this input, as shown in Figure 36.

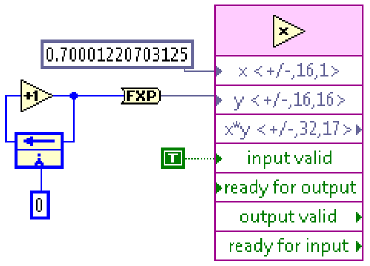


*Figure 36. You must wire a value to the **Input Valid** terminal even if incoming data is expected to always be valid.*

Upstream code may not provide an **output valid** signal, so you must find other mechanisms to determine the validity of the data. Figure 37 shows an example where the **Timeout** output of a **FIFO Read** method acts as an indicator of data validity. If the **Timeout** output is true, then the data coming out of the **FIFO Read** method is invalid.

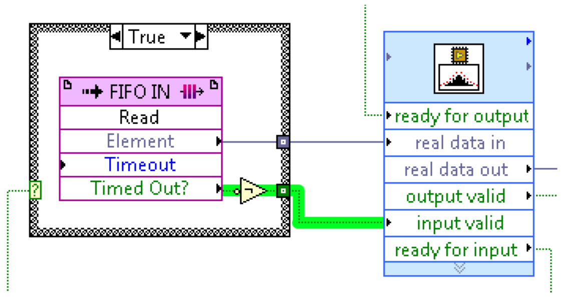


*Figure 37. Not every function or node provides a four-wire handshaking interface, but you can often find a way to infer the validity of the data, as in the case of the FIFO Read method. The **Element** output of the FPGA FIFO method is invalid when the FIFO **Timed Out?** terminal is TRUE.*

OUTPUT VALID

After the function executes, this output indicates if the associated output signal carries valid data.

Downstream code often needs confirmation of the validity of data being passed to it, especially if there will be side effects such as I/O, storage or state changes, or data transfer operations.

If downstream code also implements the four-wire protocol, you can connect the **output valid** signal of the function to the **input valid** terminal of the appropriate downstream function.

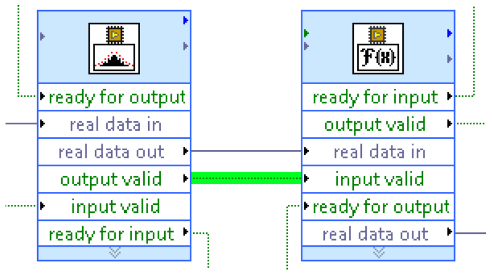


*Figure 38. As previously shown, four-wire handshaking functions convey the validity of each piece of data by connecting the **output valid** signal of the upstream function to the **input valid** input of the downstream function.*

Downstream code may not have a built-in handshaking terminal or a way to be informed about the validity of the data being passed. This is the case with many I/O nodes as well as communication and storage constructs in LabVIEW FPGA. In this situation, you can use a Case structure to prevent downstream code from executing when the node generates invalid data, as shown in Figure 39.

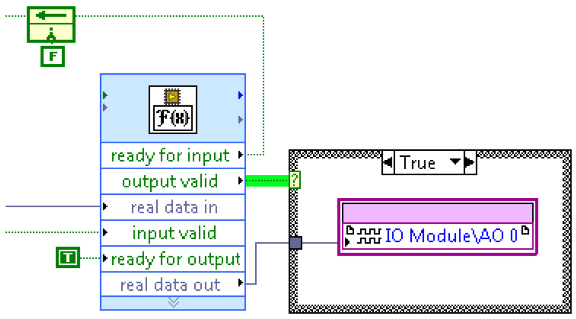


Figure 39. Not every function or node provides a four-wire handshaking interface to prevent it from executing when invalid data is passed. You can use a Case structure to selectively execute downstream code based on the value of the ***output valid*** *signal.*

READY FOR OUTPUT

This input signal reports if downstream nodes can accept new output from this node. If downstream nodes are also four-wire compliant, then this signal can be interpreted as their readiness to accept data on the current iteration.

Since the signal usually comes from downstream nodes, you must use a Feedback Node to prevent a cycle on the diagram. You also must initialize the Feedback Node to **FALSE** unless downstream code can always accept data on its first call.

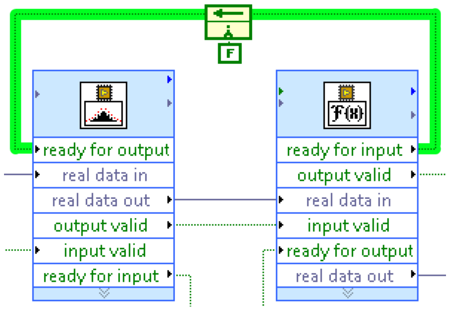


Figure 40. The ***ready for output*** *input tells the handshaked function that downstream code can accept data produced in the current execution of the loop.*

Downstream nodes may not be four-wire compatible but may contain a signal that determines if they can accept data on the current or next iteration. The **FIFO Write** method is an example of this.

The **Timeout** terminal of the **FIFO Write** method outputs **TRUE** when the most recent element cannot write to the FIFO, most likely because it is full. You can use this as a way to keep upstream blocks from feeding more data and to preserve the attempted element value so that you can retry the write operation later, as shown in Figure 41.

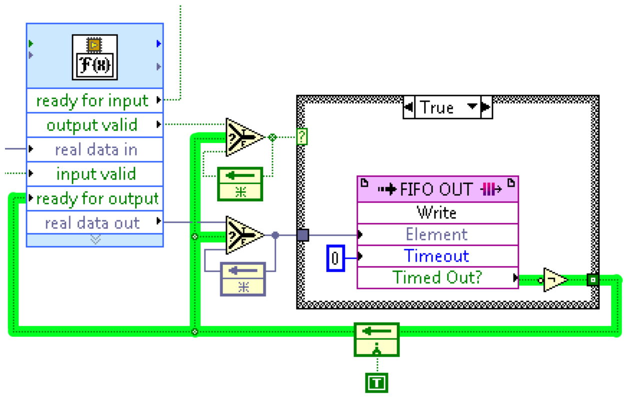


Figure 41. A timeout on the FIFO Write method indicates that the FIFO is full after the attempted write, so upstream nodes should stop providing new values and the most recent element should be stored until space becomes available on the FIFO. This coding pattern is important to prevent data loss when upstream code uses a handshake interface.

READY FOR INPUT

This output tells upstream nodes if the handshaked node can consume data on the next cycle. Since the signal propagates upstream, a Feedback Node is required to prevent a cycle in the diagram.

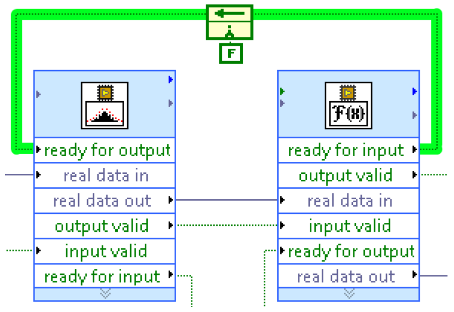


*Figure 42. When two functions with a four-wire interface are connected, the **ready for input** signal of the downstream functions should be wired back to the upstream one through a Feedback Node.*

Upstream code may not be four-wire compatible or may not have any handshaking terminals. In such cases, the **ready for input** signal may need to be reinterpreted or used to control a Case structure to keep upstream nodes from executing.

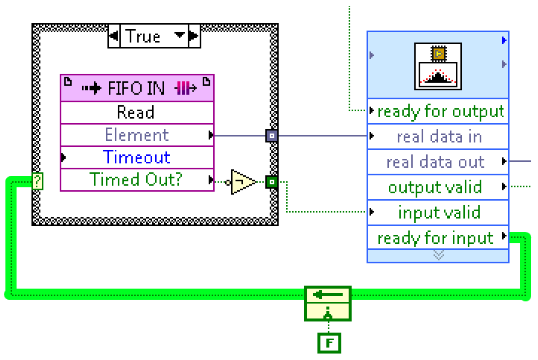


*Figure 43. When upstream code does not have a four-wire handshaking interface, the **ready for input** signal should be used to control code execution. Use a Case structure or other mechanism to make sure the code generates only valid data when the handshaked function is ready to receive it.*

VERIFYING HANDSHAKING LOGIC

Incorrect wiring of handshaking signals can lead to incorrect behavior, data loss, data corruption, data duplication, and suboptimal performance. You can verify the handshaking connections by executing the code in Simulation mode and observing its behavior over multiple SCTL iterations.

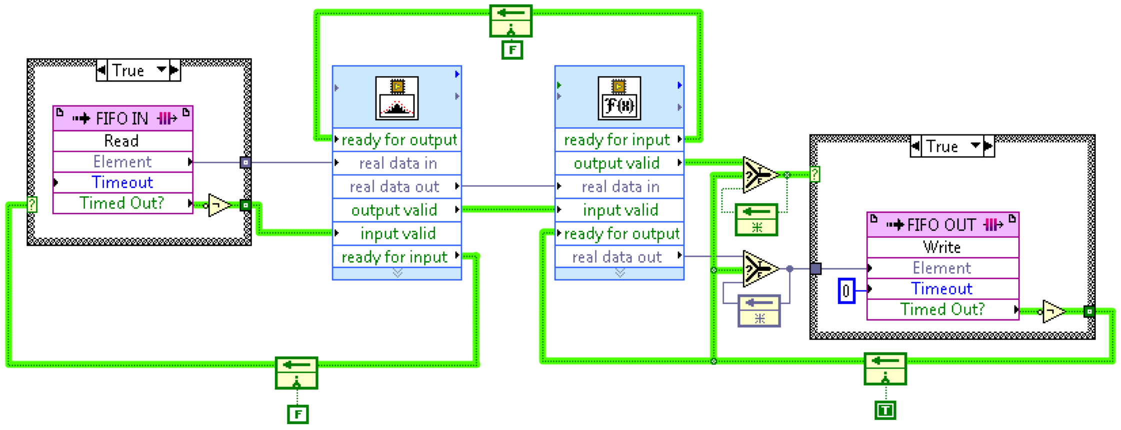


Figure 44. This SCTL code reads samples from a FIFO and performs a windowed FFT before writing the samples back to a FIFO for further processing in another loop. The code contains most of the four-wire handshaking scenarios mentioned in the previous section. Many of the wires on this diagram are handshaking signals that help maintain the flow of valid data. Verification of these signals is important to guarantee the correct functionality of the code.

You also can use Simulation mode to measure the latency and initiation interval of your IP by determining how many cycles it takes for a sample to propagate through the chain, and how many cycles must elapse before the IP accepts new inputs, which are signaled by the **ready for input** terminal.

The Sampling probe is a new LabVIEW probe specifically designed to validate the handshaking and functional behavior of LabVIEW FPGA IP. The Sampling probe provides a digital waveform view of your data when the SCTL executes in Simulation mode, as shown in Figure 45.

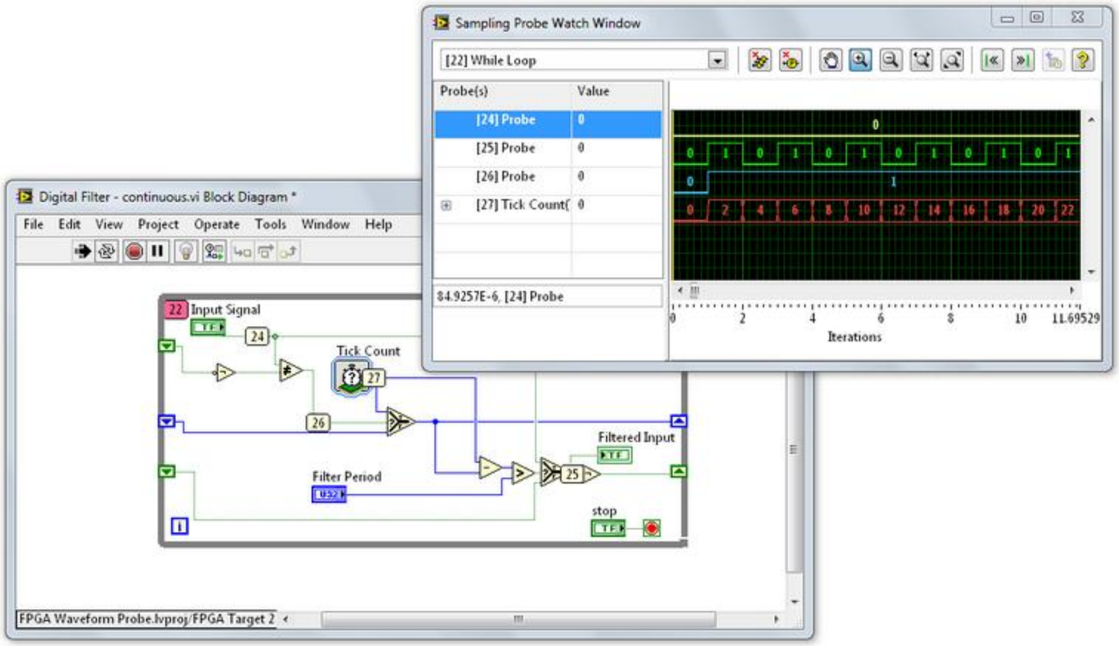
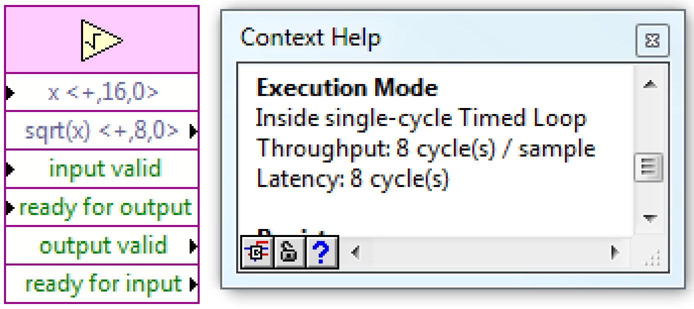

Sampling Probe Watch Window
[22] While Loop
Probe(s)
Value
[24] Probe
[25] Probe
[26] Probe
[27] Tick Count(
84.9257E-6, [24] Probe
Iterations
Digital Filter - continuous.vi Block Diagram *
File Edit View Project Operate Tools Window Help
Input Signal
Tick Count
Filter Period
Filtered Input
stop
FPGA Waveform Probe.lvproj/FPGA Target 2

Figure 45. Sampling probes show historical wire values in Simulation mode. They are useful for verifying the correctness and performance of your IP, including verifying the functionality of any handshaking signals.

DETERMINING PROCESSING CHAIN THROUGHPUT

Many LabVIEW FPGA functions provide information about their initiation interval. You can use that information to calculate the maximum achievable throughput for a series of connected nodes inside an SCTL. As introduced in the Throughput Optimization Techniques chapter, the initiation interval of a chain of nodes is equal to the highest initiation interval in the series. The node(s) with the highest initiation interval is the bottleneck of the processing chain.

The initiation interval of a node is documented within the configuration dialog, Context Help, or reference topic. Sometimes the initiation interval is referred to as throughput, but it uses units of cycles/sample, which is inversely proportional to throughput.



Execution Mode
Inside single-cycle Timed Loop
Throughput: 8 cycle(s) / sample
Latency: 8 cycle(s)

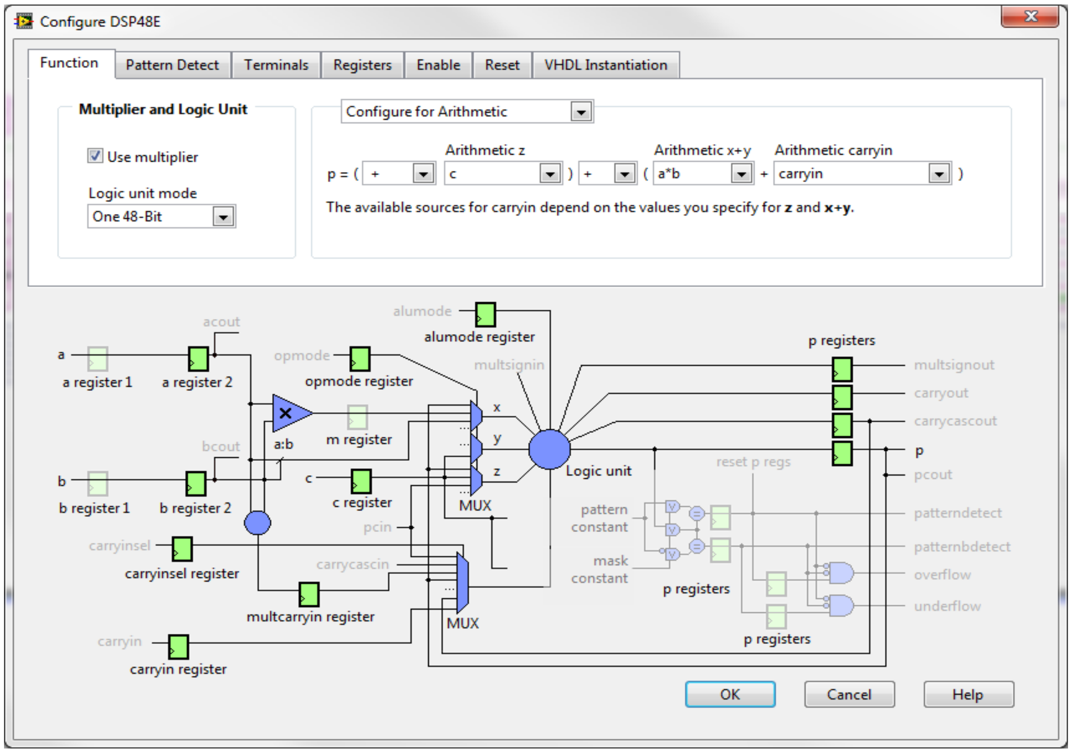
Figure 46. The Context Help for many high-throughput functions documents the expected throughput (in initiation interval cycles) and latency, based on the selected configuration of the function.

THE DSP48 NODE

The DSP48 node is an example of how LabVIEW FPGA exposes specialized FPGA elements that you can use to create high-performance processing IP.

A DSP48 is a digital signal processing element on certain Xilinx FPGA device families such as the Xilinx Virtex-5. You can use this slice to implement different kinds of arithmetic operations, including a multiply-accumulator, multiply-adder, and a one- or N-step counter. You also can use the slice to implement different kinds of logic operations, such as pattern AND, OR, and XOR operations.

Xilinx device families offer variations of the DSP48 node, such as the DSP48E or DPS48E1, but their basic functionality is similar. You can cascade multiple DSP48E slices to implement larger functions, including complex multipliers and n-tap finite impulse response (FIR) filters, without using any additional FPGA fabric resources. As a result, IP built out of DSP48 slices can compile at high clock rates.


Configure DSP48E

Function | Pattern Detect | Terminals | Registers | Enable | Reset | VHDL Instantiation

Multiplier and Logic Unit

☑ Use multiplier

Logic unit mode: One 48-Bit

Configure for Arithmetic

p = (+ | Arithmetic z: c) + (Arithmetic x+y: a*b + Arithmetic carryin: carryin)

The available sources for carryin depend on the values you specify for **z** and **x+y**.

Figure 47. The configuration dialog for the DSP48E node offers a graphical view of the DSP48E circuitry as you enable and configure different functionality.

Because of the steep learning curve associated with this node, the DSP48 should be reserved for cases where extreme optimization is required. Refer to the *DSP48E Description and Specifics* chapter of the *Xilinx Virtex-5 FPGA XtremeDSP Design Considerations* before attempting to configure this node. The document is on the Xilinx website at xilinx.com. The *DSP48E Applications* chapter contains a long list of sample applications, along with detailed information on their capabilities, configuration details, and caveats. This chapter is worth reviewing if you need to solve similar challenges with high-performance requirements.

You can accomplish the below sample IP tasks with this node.

Basic Math Applications ▪ 25 x 18 Two's Complement Multiply ▪ Two Input 48-Bit Addition ▪ Four Input 46-Bit Addition ▪ Two Input 48-Bit Dynamic Add/Subtract ▪ Three Input 47-Bit Dynamic Add/Subtract ▪ 25 x 18 Multiply Plus 48-Bit Add/Sub ▪ Extended Multiply ▪ Floating Point Multiply and 59 x 59 Signed Multiply ▪ 25 x 18 Multiply Plus 48-Bit Add Cascade ▪ Division **Filters** ▪ Polyphase Interpolating FIR Filter ▪ Polyphase Decimating FIR Filter ▪ Multichannel FIR ▪ Preloading Filter Coefficients **Advanced Math Applications** ▪ Accumulate ▪ Two Input 48-Bit Add Accumulate ▪ Dynamic Add/Sub Accumulate ▪ 96-Bit Add/Subtract ▪ 96-Bit Accumulator ▪ MACC and MACC Extension ▪ 25 x 18 Complex Multiply ▪ 35 x 25 Complex Multiply ▪ 25 x 18 Complex MACC	**Logic and Bit Field Applications** ▪ Two 48-Bit Input Bitwise Logic Functions ▪ Dynamic Shifter ▪ 18-Bit Barrel Shifter ▪ 48-Bit Counter ▪ Single Instruction Multiple Data (SIMD) Arithmetic ▪ SIMD Absolute Value (24) ▪ Bus Multiplexer **Pattern Detect Applications** ▪ Dynamic C Input Pattern Match ▪ Overflow/Underflow/Saturation Past P[46] ▪ Overflow/Underflow/Saturation Past P[47] ▪ Logic Unit and Pattern Detect ▪ 48-Bit Counter Auto Reset **Rounding Applications** ▪ Rounding Decisions ▪ Dynamic or Static Decimal Point ▪ Symmetric Rounding ▪ Random Round ▪ Convergent Rounding ▪ Convergent Rounding: LSB Correction Technique ▪ Dynamic Convergent Rounding: Carry Correction Technique

LabVIEW FPGA also includes a set of shipping examples for the DSP48 slice that serves as a reference for configuring the node. You can find the DSP48 examples by going to the **NI Example Finder** and searching for the term "DSP48."

THE FAST FOURIER TRANSFORM

Because of the limited resources on the FPGA, processing chains built within the SCTL typically operate on data on a point-by-point basis, streaming values down from one block to the next. Some functions require operations on blocks of data one at a time, which takes multiple cycles to buffer samples before they can be processed. The fast Fourier transform (FFT) function from the FPGA Math & Analysis VIs and Functions palette is an example of such a function.

The FFT function operates on blocks of data, also called frames, and provides the option to do so in a streaming, or continuous, manner as well as in a burst pattern.

During Burst operation, the FFT function accepts and buffers data that is provided one point at a time. Once the function has collected enough data to fill a frame, it stops accepting new samples and begins processing the buffered frame. Some number of cycles later, the FFT function provides its output frame, one point at a time. The function accepts samples for a new frame some time after starting to clock out its most recent result frame, as shown in Figure 48.

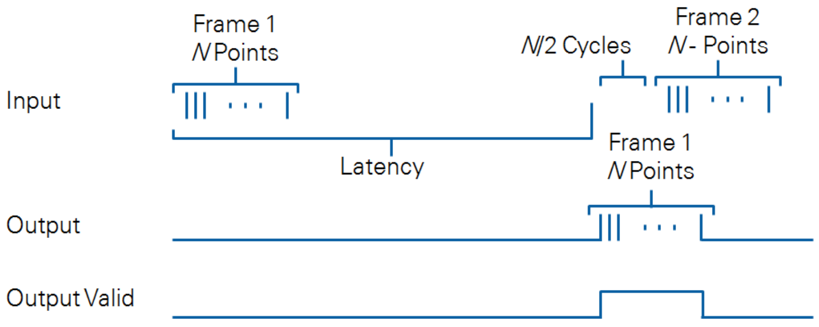


Figure 48. The FFT function operates in Burst or Streaming mode. In Burst mode, the function operates on one frame at a time, taking several cycles to accept, process, and output results before it can accept a new input frame.

Burst mode leads to an initiation interval of multiple cycles. The FFT function configuration dialog provides feedback on the expected initiation interval. The initiation interval depends on the FFT block size and the amount of internal pipelining and parallelization.

In Streaming mode, the FFT function uses some of the principles covered in the Throughput Optimization Techniques chapter to achieve an initiation interval of one sample/cycle, which means it can continually accept new samples. The function still needs to buffer the data before it can process it, so buffering and circuitry are added for the function to accept the next FFT input block while it is still processing the current frame and outputting the previous results, as shown in Figure 49.

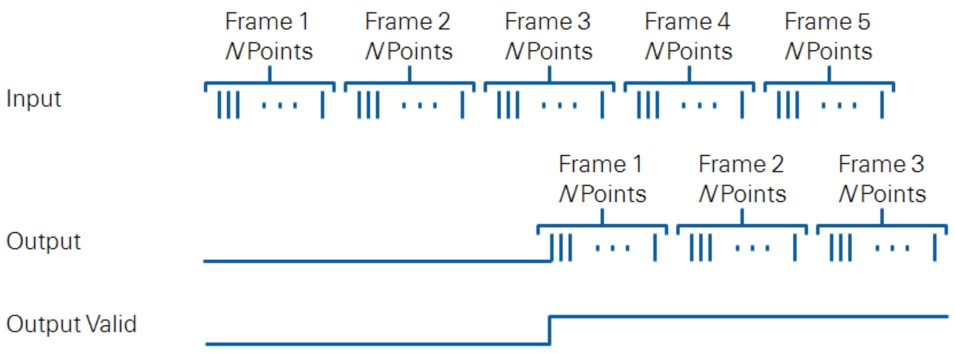


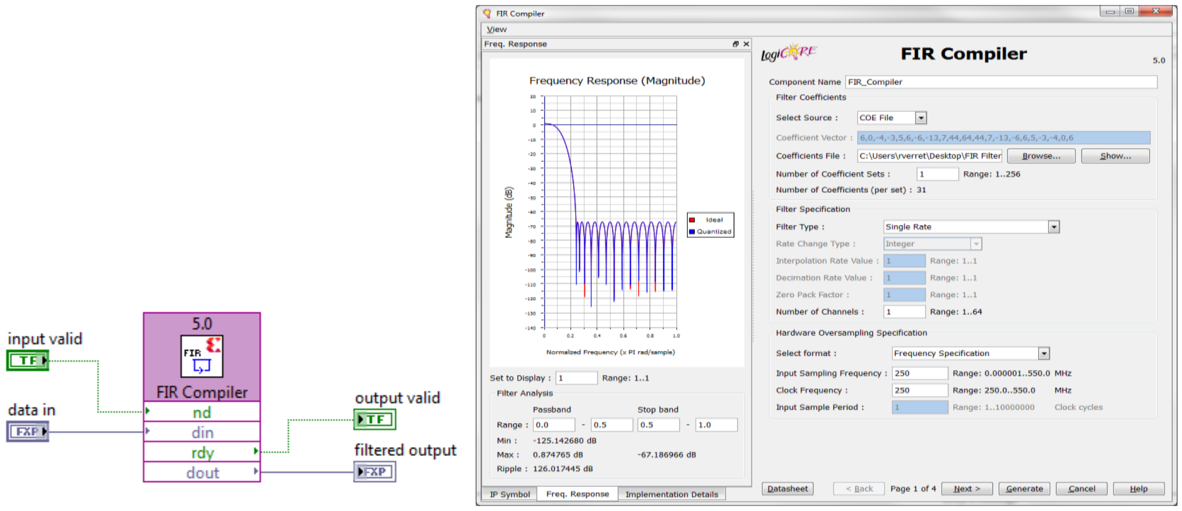
Figure 49. In Streaming mode, the FFT function accepts new samples on every cycle. Internally, the FFT is designed to buffer samples, parallelize the FFT computation, and simultaneously output results for previous frames. This mode provides the best throughput, but it comes at a significant resource cost.

Streaming mode offers the highest FFT performance, allowing the system to perform an FFT on every incoming sample. This comes at a significant cost in resources because the internal machinery is complex and must be replicated to hold and process multiple frames at time.

THE XILINX CORE GENERATOR IP SYSTEM

LabVIEW FPGA provides access to Xilinx's CORE Generator IP system directly from the Xilinx Coregen IP Functions palette. The CORE Generator IP system is a catalog of architecture-, domain-, and market-specific IP for automotive, communications, signal processing, and other applications. Xilinx provides these highly optimized IP blocks to help Xilinx FPGA users stitch together high-performance designs. The blocks are tightly integrated with the LabVIEW FPGA development environment and include a step-by-step configuration dialog where you can customize the selected core with respect to performance, data type, and other IP-specific behavior.

Figure 50 shows the Xilinx CORE Generator FIR filter core, represented on the diagram as a LabVIEW FPGA function, as well as its configuration dialog. The core provides configuration settings that you can use to customize the FIR filter to meet your specific application requirements.



Set to Display : 1 Range: 1..1

Filter Analysis

	Passband		Stop band
Range :	0.0 - 0.5		0.5 - 1.0
Min :	-125.142680 dB		
Max :	0.874765 dB		-67.186966 dB
Ripple :	126.017445 dB		

Component Name FIR_Compiler

Filter Coefficients

Select Source : COE File

Coefficient Vector : 6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6

Coefficients File : C:\Users\rverret\Desktop\FIR Filter Browse... Show...

Number of Coefficient Sets : 1 Range: 1..256

Number of Coefficients (per set) : 31

Filter Specification

Filter Type : Single Rate

Rate Change Type : Integer

Interpolation Rate Value : 1 Range: 1..1

Decimation Rate Value : 1 Range: 1..1

Zero Pack Factor : 1 Range: 1..1

Number of Channels : 1 Range: 1..64

Hardware Oversampling Specification

Select format : Frequency Specification

Input Sampling Frequency : 250 Range: 0.000001..550.0 MHz

Clock Frequency : 250 Range: 250.0..550.0 MHz

Input Sample Period : 1 Range: 1..10000000 Clock cycles

Figure 50. LabVIEW FPGA integrates tightly with IP from the Xilinx CORE Generator IP system. The functions have their own palettes and launch the appropriate Xilinx CORE configuration dialogs right from the LabVIEW development environment.

Because these functions are built by Xilinx, the FPGA device manufacturer, they often offer additional functionality, performance, or resource savings when compared to similar functions built by NI, so it is worth considering them as a way to optimize your design. Some of the most advanced Xilinx CORE Generator IP included in the LabVIEW FPGA palettes require a licensing fee, but the implementation and test savings are often worth the price if you need highly specialized IP.

XILINX CORE GENERATOR FFT IP

The Xilinx FFT Core is an example of a popular IP block that complements the features of the FFT Express VI in the LabVIEW FPGA Module. Similarly to the FFT Express VI, the Xilinx core is capable of operating in Streaming mode or one of three different Burst modes with varying performance and resource trade-offs. Additionally, the Xilinx FFT consumes a significant number of resources when operating in Streaming mode. You might consider avoiding that mode unless absolutely required by the application. The diagram in Figure 51 reproduces Xilinx's rough guidance on the performance and resource trade-offs for each of the modes.

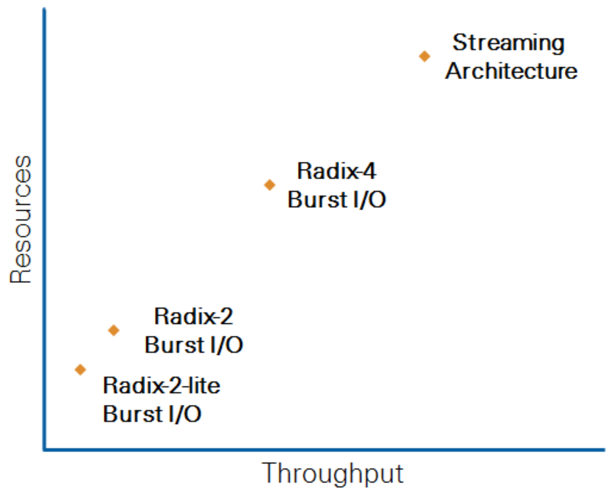


Figure 51. The different modes of operation for the Xilinx FFT provide trade-offs between throughput and resources. Burst mode uses the fewest resources, but it takes multiple cycles to process each frame. Streaming mode can process data continuously, but it uses significant resources.

When choosing between the FFT Express VI and the Xilinx CORE Generator FFT, consider that the Xilinx CORE Generator FFT:

- Is highly optimized for throughput, so it can usually compile at high clock rates
- Supports frame sizes greater than 8192 samples
- Supports multichannel operation when in Burst mode
- Supports floating-point data types, although at a considerable resource cost
- Has an FFT frame size that is run-time configurable up to the amount predetermined at compile time (note that changing the frame size requires a core reset)
- Can be configured in forward or reverse mode on a per frame basis (You can use the FFT function to perform the forward-FFT, manipulate the signal in the frequency domain, and then reuse the same FFT function/hardware circuitry to perform the inverse FFT and convert the signal back to the time domain.)

Xilinx CORE Generator FFT challenges include the following:

- The Xilinx CORE Generator FFT can be harder to configure and use than the FFT Express VI.
- The function lacks a standard handshaking interface that resembles the four-wire protocol. For example, it does not have an **input valid** signal, so you might have to place the function in a Case structure, as described earlier in this chapter, to avoid feeding invalid data to it.
- Getting the fastest performance out of the core requires some advanced customization, and it forces certain assumptions on your design. For example, the highest performance mode produces frames with their samples sorted in reverse order. Downstream blocks must be able to deal with this constraint in order to maintain the throughput rate.

Consider reading the first few chapters of the Xilinx LogiCORE IP Fast Fourier Transform data sheet if you are considering using the core for your designs. The document is technical and assumes some digital design knowledge, but it is a useful resource for getting the best FFT performance. Similar advice applies to the other Xilinx CORE Generator IP in the LabVIEW Coregen IP palette.

AXI PROTOCOL

National Instruments exposes the AXI, or Advanced eXtensible Interface, handshaking protocol for certain Xilinx CORE Generator IPs on specific hardware targets. Introduced by ARM in 1996, the AXI protocol has become an industry-standard bus interface for interconnecting functional blocks of IP for high-performance, high-frequency applications.

In the context of the LabVIEW FPGA Module, the AXI protocol is similar to the LabVIEW four-wire handshaking protocol with respect to signal connections. For every piece of data there are two signals: one for the value and another to indicate whether the value is valid.

When you use AXI, you must use a Feedback Node when connecting AXI IP. The main difference between the AXI protocol and the LabVIEW four-wire handshaking protocol is the naming of the signals and the placement of Feedback Nodes when interconnecting IP in the SCTL. Refer to the *Interfacing AXI IP in FPGA VIs* topic of the LabVIEW FPGA product documentation for a detailed description of how to connect AXI nodes to other AXI nodes, as well as how to connect AXI nodes to nodes that support the four-wire protocol.

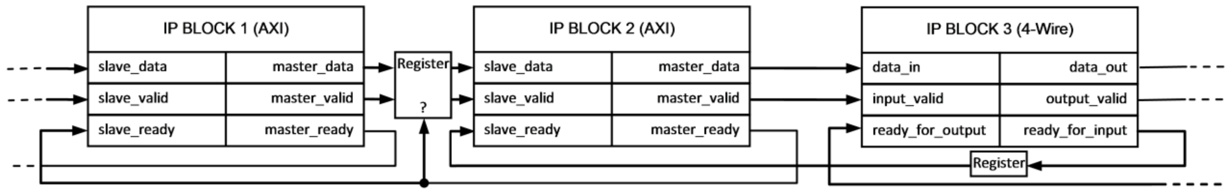


Figure 52. The AXI and four-wire handshaking signals have different names but similar meanings. Feedback Node (labeled as Register) placement varies depending on the type of IP interconnecting.

INTEGRATING HDL IP

Hardware description languages (HDLs) are text-based languages used to design hardware components for FPGAs and application-specific integrated circuits (ASICs). These languages provide a standard way of representing logic, signals, connections, concurrency, timing, and other hardware concepts. They also express structural, behavioral, and low-level implementation details without having the designer specify the circuit on a gate-by-gate basis.

In the context of this guide, HDL IP refers to IP written in VHDL or Verilog, or a net-list file generated from such languages.

VHDL and Verilog have a long history in the digital design industry and are supported by most EDA tools. They represent the traditional way of creating designs on FPGAs. Net-list is a term used to refer to the precompiled output of an HDL design. The net-list representation is closer to the final circuit because it directly specifies FPGA hardware elements and the connectivity between them. Net-lists are sometimes used as a way to integrate or share IP without actually revealing the original HDL code.

You can obtain HDL IP from a variety of places. Your organization may already use traditional digital design tools, in which case you may already have HDL IP or may have the expertise to develop it. There are also community sites, such as opencores.org, where you can find open source HDL IP. Other digital design companies create and license market-specific HDL IP.

You may want to integrate HDL IP into your LabVIEW FPGA design if you already have the IP or believe that it would be easier to create the IP using an HDL. An HDL provides access to low-level hardware and toolchain settings, so it is another option to consider when you need extreme performance and resource control.

There are two mechanisms for importing HDL IP into LabVIEW FPGA:

1. Through the IP Integration Node (IPIN)
2. As Component-Level IP (CLIP)

CHOOSING BETWEEN IPIN AND CLIP

There are many differences between the IPIN and CLIP mechanisms, but the decision to use one over the other should be based on the kind of IP you want to integrate.

Use the IPIN to integrate IP that is more functional and algorithmic in nature or works as a processing function, where some input is provided and the IP generates some output on every call.

Use the CLIP to integrate IP that acts as an independent, asynchronous component, such as IP that interacts with multiple parts of your application, or an interface to some external digital component, such as memory, SPI, or I^2C.

The IPIN, which must be placed inside the SCTL, follows the LabVIEW call model, where data is passed into the node and the node might generate output data for other nodes to consume. The IPIN cannot access I/O directly from the HDL code, so it must accept or generate the values that it needs as inputs or outputs. When you place an IPIN in the SCTL, it represents a single instance of your IP. If you drop two nodes, you have two instances of the IP.

In contrast to the IPIN, you do not see the CLIP on your diagram because it is represented only in the LabVIEW project. The CLIP lives outside the FPGA VI, and you can instantiate multiple copies of a given CLIP. LabVIEW FPGA diagrams interact with the CLIP IP through CLIP accessory nodes, which are similar to I/O nodes and allow you to write to or read from the CLIP interface registers. On NI FlexRIO hardware, the CLIP also provides access to I/O pins. This type of CLIP is also known as a “socketed” CLIP.

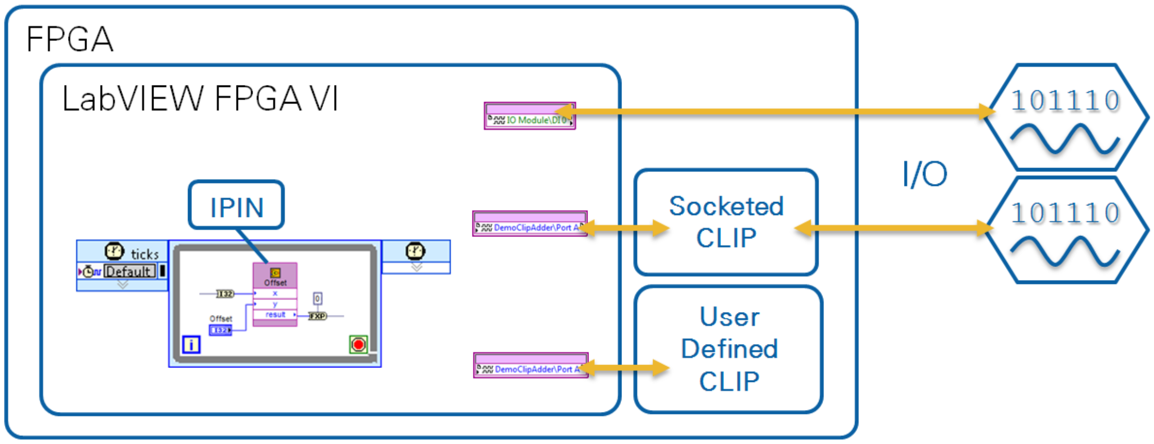


Figure 53. CLIP and IPIN are the HDL integration mechanisms in LabVIEW FPGA. IPIN is appropriate when the IP can be modeled as a function, while CLIP is a better option when the IP must execute as its own component. CLIP might have access to I/O, depending on your RIO device type.

The following table summarizes some of the other differences between CLIP and IPIN.

CLIP	IPIN
Supports multiple clock domains and can create and share clocks with the FPGA VI	The SCTL specifies the clock of contained IPIN nodes
Supports interfacing with the FPGA VI	Is part of the LabVIEW FPGA diagram
Supports access to FPGA I/O on certain targets	Cannot access to I/O from within the HDL IP
Supports constraint files	Accepts floating-point and other more advanced data types
Supports using a third-party simulator such as Mentor Graphics QuestaSim and Xilinx ISim	Supports the built-in simulation in LabVIEW FPGA

Refer to the product documentation and the tutorial examples included in the product for more information on how to use these integration mechanisms.

INTEGRATING IP INTO SOFTWARE-DESIGNED INSTRUMENTS

With the NI vector signal transceiver (VST) as well as other software-designed instruments, you can customize behavior through the LabVIEW FPGA Module. In the case of the VST, for example, there are two software stack options when you want to go beyond the NI-RFSG and NI-RFSA driver capabilities. The first option is to perform common and partial modifications using instrument driver FPGA extensions. The second option is to fully customize the behavior using the open software stack based on LabVIEW.

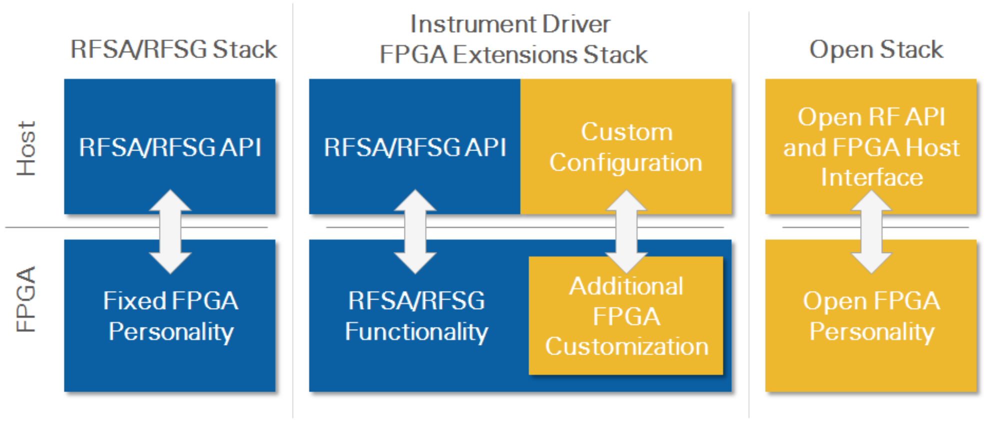


Figure 54. The VST can be used with the standard RFSA/RFSG instrument drivers, the completely open software stack, or the intermediate option of instrument driver FPGA extensions for common customizations that require the benefit of the standard instrument driver functionality.

The open stack option is complemented by a set of sample projects that provide complete, open architectures for implementing common applications. Both the host and FPGA side of the stack can be modified as necessary. You can take this approach when you want to start from scratch for both the host and FPGA sides, which enables you to design a completely custom instrument that leverages the front-end I/O capabilities of the device, including the DOCSIS Channel Emulator from Averna, a National Instruments Alliance Partner.

INSTRUMENT DRIVER FPGA EXTENSIONS

With instrument driver FPGA extensions, you can use the standard instrument drivers, such as the NI-RFSA and NI-RFSG drivers in the case of the VST, while still modifying and extending instrument functionality on the FPGA.

The FPGA section of the stack implements the instrument functionality that is expected by the standard instrument driver and clearly identifies the places where you can override or extend specific functionality, such as triggering, filtering, and other signal processing.

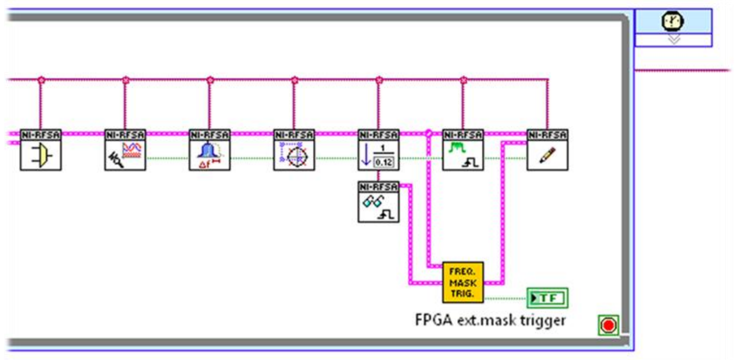


Figure 55. Instrument driver FPGA extensions expose FPGA code that is compatible with the standard RFSA/RFSG instrument driver API. At the same time, this approach identifies and helps you customize key parts of the FPGA.

INTEGRATING IP FROM THE COMMUNITY

There are a few external sources of high-performance LabVIEW FPGA IP that can save you from implementing functions not included in the product. This section highlights some of the most relevant sources.

NI FlexRIO AND SOFTWARE-DESIGNED INSTRUMENTS IP COMMUNITY

This community serves as a central repository for examples and IP for NI software-designed instruments and NI FlexRIO devices. From I^{2}C to full channel emulation examples, you can find specific protocols and functions for these high-performance devices as well as a community of people who can help answer your questions.

Many of the examples for the software-designed instruments are based on the LabVIEW sample projects included with the product. These examples leverage the concept of instrument driver FPGA extensions or use the NI FlexRIO Instrument Development Library as IP building blocks.

The NI FlexRIO Instrument Development Library is a collection of LabVIEW host and FPGA code that is designed to provide FPGA capabilities commonly found in instruments, such as acquisition engines, DRAM interfaces, and trigger logic, along with the associated host APIs. The library is modular, so you can choose only the components you need. The library also delivers efficient implementation and allows you to modify the provided code, if necessary, to meet your unique application needs.

IPNET AND THE LabVIEW TOOLS NETWORK

IPNet is a collection of IP and examples specifically created for LabVIEW FPGA by the community and NI developers. The LabVIEW Tools Network provides access to certified, third-party add-ons for different applications and devices, including some specifically created for the LabVIEW RIO platform.

You should perform a quick search of IPNet and the LabVIEW Tools Network before creating your LabVIEW FPGA IP.

Additional Resources

[1] An Introduction to Peer-to-Peer Streaming
http://www.ni.com/white-paper/10801/en

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

[3] Xilinx LogiCORE IP Fast Fourier Transform Data Sheet
http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf

[4] Interfacing AXI IP in FPGA VIs
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/xilinxip_using/

[5] OpenCores
www.opecores.org

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

[7] Using Instrument Driver FPGA Extensions
http://www.ni.com/white-paper/14648/en/

[8] NI FlexRIO Instrument Development Library
https://decibel.ni.com/content/docs/DOC-15799

[9] NI LabVIEW Digital Filter Design Toolkit
http://sine.ni.com/nips/cds/view/p/lang/en/nid/209040

[10] Examples and IP for Software-Designed Instruments and NI FlexRIO
https://decibel.ni.com/content/groups/software-designed-instrument-and-ni-flexrio-examples-and-ip

[11] An Introduction to Instrument Driver FPGA Extensions
http://www.ni.com/white-paper/14646/en/

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

[13] The LabVIEW Tools Network
http://www.ni.com/labview-tools-network/

This page intentionally left blank.

TIMING OPTIMIZATION TECHNIQUES

Applications often need to specify the amount of time between events of interest. In LabVIEW FPGA applications, these events are usually related to I/O. NI FPGA-based devices provide fast response times because the processing occurs in hardware in close proximity to the I/O. This chapter covers various techniques to help control and measure the timing between events with greater accuracy and precision.

Latency is important in control applications, where the status of the system is sampled and processed by controller logic, and then control outputs are generated within a certain amount of time. Lower latency is often desirable in control applications.

Digital protocol applications also specify timing constraints that must be observed to properly communicate with some external device. In this case, precise timing and repeatability between events is needed to satisfy the settling and hold requirements of electrical signals for correct data transmission.

DETERMINING AND SPECIFYING LATENCY WITH THE SCTL

The total timing of a processing code path is the sum of the latency of its components. The SCTL provides a built-in timing guarantee because each iteration takes exactly one clock cycle. The latency between events across multiple SCTL iterations is expressed as follows:

$$Latency = \frac{cycles}{clock\ rate}$$

This means that you can prescribe latency by changing the clock rate or the number of cycles between events of interest.

The SCTL clock rate defines your timing resolution. Higher clock rates provide greater timing resolution but can make it harder to compile the design. You should increase your clock rate only as much as needed to achieve the desired resolution.

You can increase latency between events in an SCTL by spreading the time over multiple iterations of the SCTL while sticking with a clock rate that provides the appropriate resolution. The most basic way to do this is to insert Feedback Nodes to add cycles of delay as the data propagates within the SCTL.

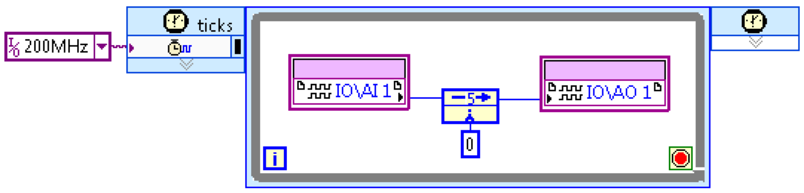


Figure 56. An additional 25 ns of delay between input and output are achieved by inserting a Feedback Node with a delay value of five.

The state machine pattern in LabVIEW provides a more advanced approach for controlling time between events because each state can keep track of the number of elapsed cycles before moving on to the next state. The guaranteed iteration latency provides a way to control the timing between events in different states.

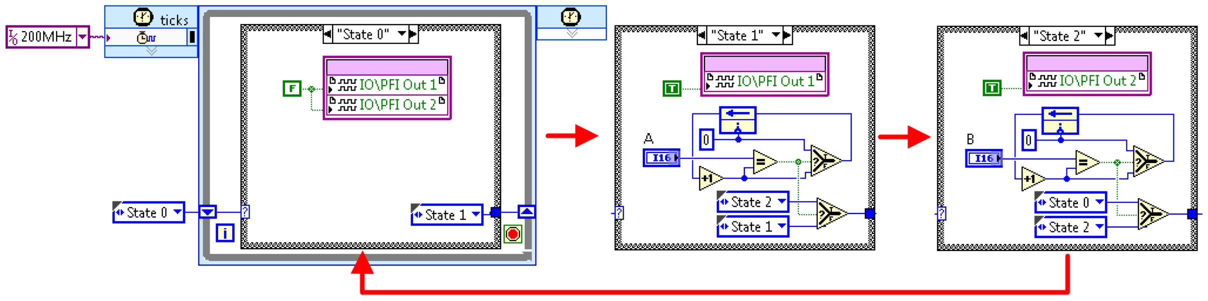


*Figure 57. A state machine pattern using the SCTL controls timing between digital output events by counting the number of cycles. The first state, **State 0**, deasserts two digital output lines and proceeds to **State 1**. **State 1** asserts **PFI Out 1** and waits **A** cycles before advancing to **State 2**. **State 2** asserts **PFI Out 2** for **B** cycles before returning to **State 0**. Timing between the I/O events is determined by parameters **A** and **B**, with the 5 ns resolution, determined by the 200 MHz SCTL clock.*

Attempting to reduce the amount of time between events is difficult, especially when the events already span multiple SCTL iterations and the latency is already larger than desired. The most promising path depends on the magnitude of the quantities in the relation. If the number of cycles is already small, then removing a few cycles could have a significant impact on latency. If the number of cycles is already large, in the thousands for example, then it might be difficult to remove enough cycles to have any measureable impact on latency. In this case, increasing the clock rate, through means other than pipelining, may be the best approach. The rest of the chapter concentrates on techniques that can help you reduce latency.

REDUCING LATENCY THROUGH PARALLELIZATION

As mentioned in the Throughput Optimization Techniques chapter, you can take advantage of the highly parallel nature of the FPGA to perform operations simultaneously, which shortens overall latency at the expense of additional resources. Parallelization also can shorten the critical path, which allows for higher clock rates without the use of pipelining registers and additional clock cycles of delay.

Parallelization might require significant changes to your code, and can be difficult or impossible to achieve depending on the nature of the algorithm. Unfortunately, there is not a guaranteed recipe for algorithm parallelization. There are, however, some common parallelization patterns that you can adopt.

When data is made up of independent streams such as I/O channels, operations like maximum, minimum, and average are easily parallelizable by replicating the code on the diagram to match the number of independent elements/channels.

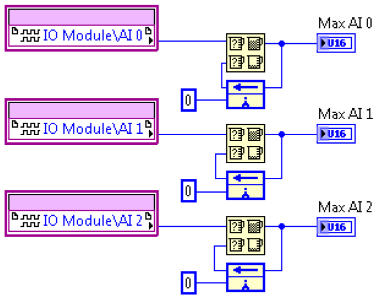


Figure 58. You can perform multiple, simultaneous operations on independent data elements to take advantage of the inherent parallelism in FPGA devices.

If the nature of the operation is such that it can be performed on subsets within the same data set, you can replicate the operation many times to reduce latency in the same proportion. For example, when applying a threshold operation on image data, every pixel can be operated on in parallel. Replicating the code **n** times will shorten the latency of the image threshold operation by a factor of **n**.

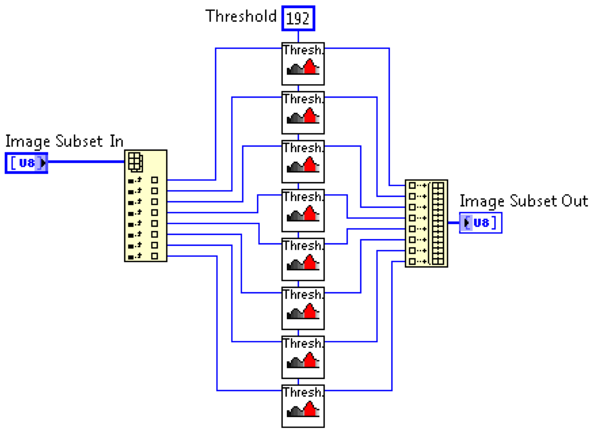


Figure 59. Certain operations can process multiple parts of the same data stream independently. This code applies a threshold operation to eight sequential image pixels simultaneously, which increases throughput and decreases latency by a factor of eight.

You can still achieve partial parallelization even when there are no obvious data separation options to replicate an operation. When you aggregate multiple items using an operation that is associative in nature, such as multiplication and addition, you can create a tree structure to reduce the length of the combinatorial path. The example in Figure 60 shows how the elements in an array can be added more quickly using a tree structure.

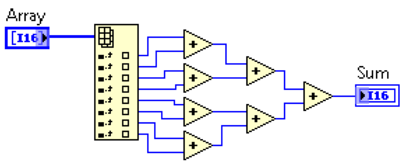


*Figure 60. The tree structure provides a logarithmic speedup. The number of stages required to process **n** elements is roughly $\log_2$ (**n**). This is better than the more obvious alternative, where **n** pairs are processed sequentially.*

You can apply a similar pattern to determine the minimum and maximum of a set of numbers, which is also associative, and, as shown in Figure 61, results in the partial combination of two tree structures.

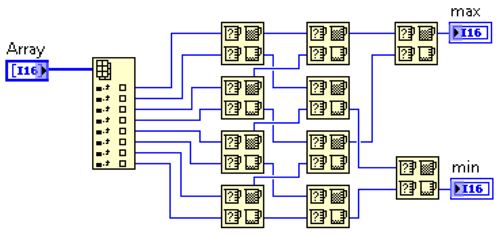


*Figure 61. This diagram leverages the idea that picking the larger of two numbers also identifies the smaller one, using the same logic function (**Max & Min**) for more than one purpose. The dual tree structure shortens operation latency by a logarithmic factor with respect to the number of elements.*

You can further generalize the tree pattern to a lattice structure in the case where the operation generates multiple values, such as when you want to sort array elements using the “bubble” sort algorithm, as shown in Figure 62 below.

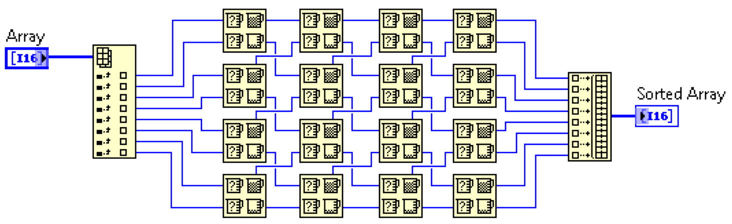


Figure 62. This bubble sort algorithm, implemented in lattice form, leverages partial parallelism and operation associativity to process a fixed size in one clock cycle.

This pattern is significantly faster than the original array sort algorithm first shown in the High-Performance Programming With the Single-Cycle Timed Loop chapter, which required up to n passes through the array of n elements to sort the data. Figure 62 can run in one SCTL cycle, depending on the clock rate and the array size.

REMOVING PIPELINING REGISTERS

As explained in the Throughput Optimization Techniques chapter, pipelining increases throughput at the expense of increased latency. As a result, you can reduce latency by removing any pipeline registers in the critical latency path. You should also check for any pipelining stages implicitly added by functions, such as those in the high-throughput FPGA palette.

OPTIMIZING DATA TYPES

The recommendations discussed in the Resource Optimization Techniques chapter can have a significant effect on shortening propagation delay, which allows you to increase the SCTL clock rate with fewer pipeline stages and decrease overall latency.

RESOURCE OPTIMIZATION TECHNIQUES

Even though FPGAs have followed Moore's law in terms of performance and logic element count, they are still relatively constrained when it comes to resources. FPGA resources must be carefully managed not only because they can keep your design from fitting, but also because they can have a dramatic impact on achievable clock rate and performance.

FPGA RESOURCE TYPES

Learning about the different FPGA fabric components can help you understand how resources are allocated as the design grows and what options you may have for further optimization. An FPGA is primarily made up of two types of components: configurable logic blocks, or slices, and resources with specialized functionality, such as I/O or block RAM.

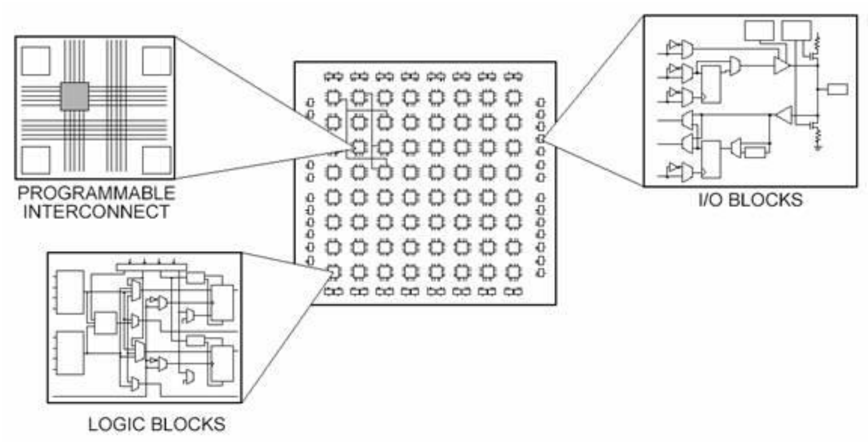


Figure 63. FPGA resources can be categorized into logic blocks or slices and specialized blocks, such as I/O or block RAM. Programmable interconnects route signals between these blocks.

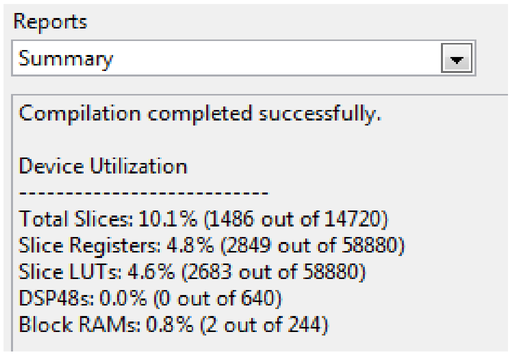
Logic resources are grouped in slices to create configurable blocks that can perform logic functions. A slice contains a set number of LUTs, flip-flops, and multiplexers.

- A LUT is a collection of logic gates hard-wired on the FPGA. LUTs store a predefined list of outputs for every combination of inputs and provide a fast way to retrieve the output of a logic operation.
- A multiplexer, also known as a mux, is a circuit that selects between two or more inputs and outputs the selected input.
- A flip-flop is a circuit capable of two stable states and represents a single bit. Flip-flops can be grouped into a register to store bit patterns, such as numeric values on the diagram. A register on the FPGA has a clock, input data, output data, and an enable signal port. On every clock cycle the input value is latched, stored internally, and the output is updated to match the internally stored data. FPGA VIs use registers to implement many LabVIEW constructs, including Feedback Nodes and loop shift registers.

Different FPGA families implement slices and LUTs differently. For example, a slice on a Virtex-II FPGA includes two LUTs and two flip-flops, but a slice on a Virtex-5 FPGA includes four LUTs and four flip-flops. The number of inputs to a LUT, commonly two to six, also depends on the FPGA family.

Specialized FPGA resources perform very specific functions, such as DSP, clock management, larger storage elements (block RAM), and I/O. Because these resources are usually scarce, numbering in the hundreds to few thousands depending on the FPGA family, they must be managed carefully.

Block random access memory, or block RAM, is embedded throughout the FPGA for storing data. LabVIEW attempts to use block RAM when synthesizing memory and FIFOs. Memory and FIFO items are explained in more detail in the Data Transfer Mechanisms chapter. You can specify the type of resource LabVIEW should use to implement these and other items to manually balance design resources as the FPGA approaches full capacity.



```
Compilation completed successfully.

Device Utilization
---------------------------
Total Slices: 10.1% (1486 out of 14720)
Slice Registers: 4.8% (2849 out of 58880)
Slice LUTs: 4.6% (2683 out of 58880)
DSP48s: 0.0% (0 out of 640)
Block RAMs: 0.8% (2 out of 244)
```

Figure 64. LabVIEW FPGA compilation results show resource types as well as percentage used by the design.

FILLING UP THE FPGA

As you add logic to your design, compilation might fail for the following reasons:

- The FPGA runs out of a specific resource type
- The compiler fails to find a way to create routes between components
- Propagation and logic delays prevent design compilation at the requested clock rates

The FPGA has a finite number of resources for each resource type. As the compiler attempts to synthesize the design, it may run out of one type of resource first. Additionally, logic resources are packed together into slices and some parts of a given slice may be unusable when the rest is configured in a specific manner, which makes full utilization of a specific resource type difficult. You sometimes can rebalance resources across types to make the design fit, as discussed in the Resource Balancing section, later in this chapter.

The compiler's job is to place all of your components on the FPGA fabric. When the design is small, the compiler is more successful at placing resources and can even be slightly inefficient in placement if a configuration meets the timing requirements.

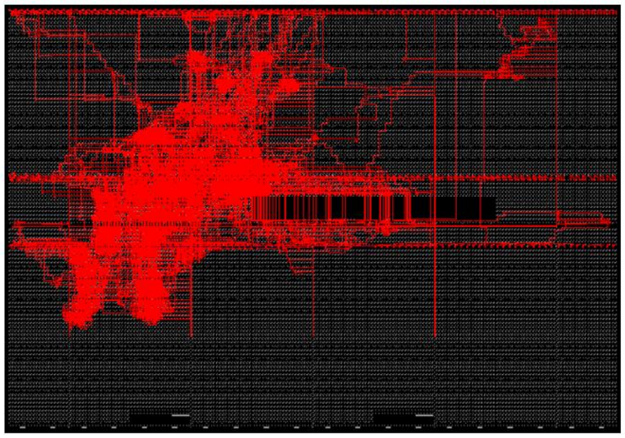


Figure 65. This shows a sample hardware layout of an FPGA design. Smaller designs are easier to synthesize and may even be inefficient with respect to resources as long as the requested timing constraints are met.

As the density of logic increases, components are placed further apart, routing becomes more difficult, and logic and propagation delays increase. This leads to longer compile times and makes compilation failures more likely, even for parts of the design that previously compiled successfully at the same clock rate.

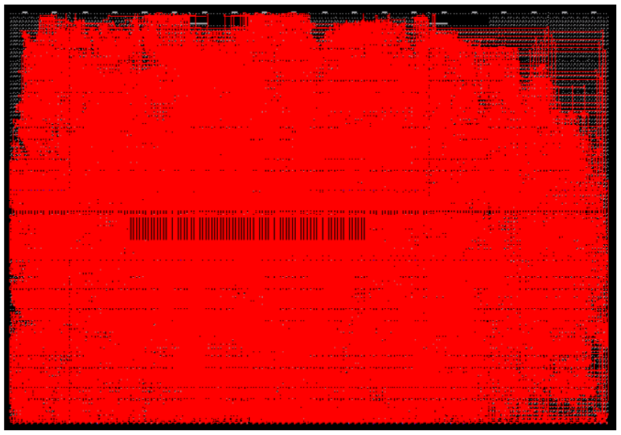


Figure 66. As the FPGA fills up, it becomes difficult to place and find short routes between design components, which often leads to timing failures.

Resource optimization helps the compiler find and improve routes between parts of your design, which makes synthesis possible. Resource optimization and balancing also can help achieve higher clock rates by allowing for shorter routes with smaller routing delays. The following sections provide resource-reduction and resource-balancing tips to help you get the most out of the FPGA.

OPTIMIZING RESOURCES THROUGH DATA TYPES

SCALAR TYPE OPTIMIZATION

The number of bits used to represent values on the FPGA, sometimes called data-type width, is the biggest factor driving resource use and performance in LabVIEW FPGA applications. Wider data types require more circuitry to hold and route values across your design. They also quickly consume resources and make it more difficult to simultaneously meet timing across the individual traces used to propagate each value bit across the FPGA.

Most LabVIEW FPGA functions require resources proportional to the number of bits of its inputs and outputs. In the case of the **Add** function, the larger the inputs, the more resources the function requires. More resources results in a longer logic delay component of the propagation delay, which limits the overall maximum clock rate.

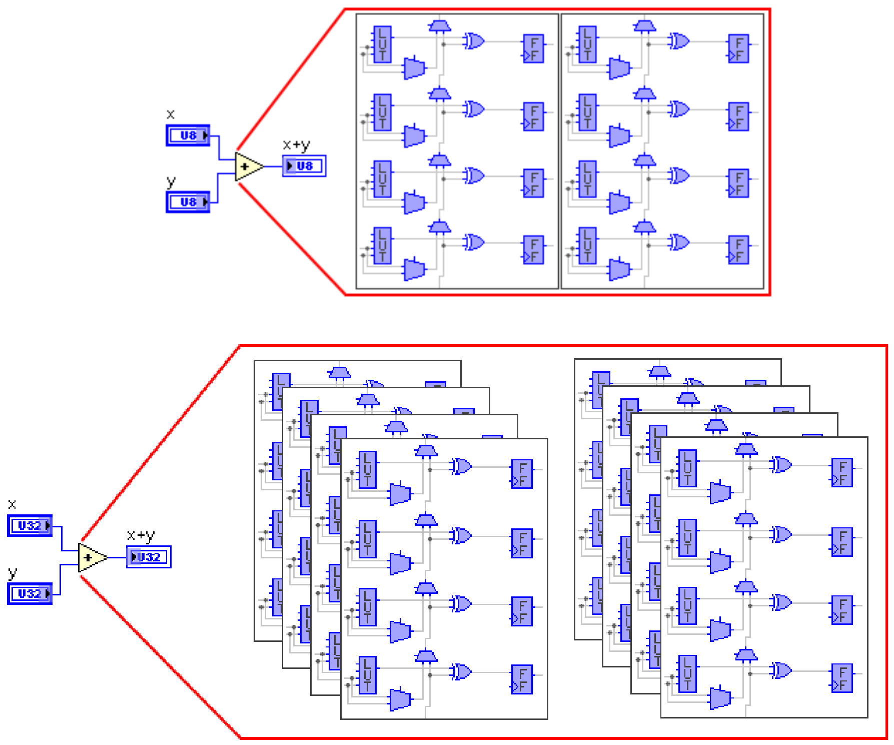


*Figure 67. A simple 8-bit **Add** function can be implemented using eight LUTs, one for each pair of added bits, and some dedicated carry-chain logic to account for overflows. On some FPGA device families, these LUTs may be distributed across two slices. A 32-bit **Add** function uses four times the resources.*

Many functions implemented using LUTs follow this model. For instance, subtraction is implemented as two's complement addition. Comparison functions also require more resources as the width of their inputs increases.

LUTs in slices make up most of the reconfigurable logic of the FPGA, and they can be used to implement many different kinds of operations. Although logic slices are relatively plentiful, routing between them becomes more difficult as the FPGA fills up. As a result, it is practically impossible to use all of the slices available on a device.

Some functions cannot be implemented using LUTs alone. Logarithm math requires multiple bookkeeping resources and multiplication uses DSP slices. The ratio between slices and specialized FPGA elements depends on the function and affects the resource layout of the FPGA. As a result, your design, when implemented in fabric, might run out of some resource types before others.

Data-type optimization applies to data and any other values you use in your design to represent state or pass as messages. For example, using an unsigned 8-bit integer (U8) to select between cases in a Case structure provides 256 different cases. Using the default LabVIEW type of I32 is probably more bits than you really need and leads to wasted resources.

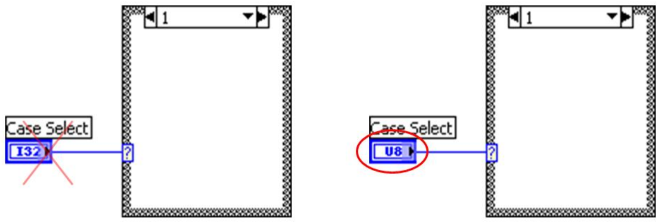


Figure 68. You can save resources by choosing the smallest data type that can represent the different values needed by each part of your design. Case structures default to an I32 type for their selectors, but most Case structures include only a handful of cases, making a U8 sufficient.

FIXED-POINT TYPE BUILD-UP

The fixed-point data type provides some of the flexibility of the floating-point data type but maintains the size and speed advantages of integer arithmetic. Fixed-point numbers represent rational numbers within a user-specified range and with user-specified precision. You can specify any size between 1 and 64 bits, inclusive, for a fixed-point number. You also can configure fixed-point numbers as signed or unsigned.

Most arithmetic functions that support fixed-point types automatically determine an output width based on the width of their inputs. When determining function output width, LabVIEW takes a conservative approach that attempts to prevent overflows and preserve precision by expanding the number of bits used to represent integer and fractional parts. Although numerically safe, this leads to the quick growth of downstream type widths and, as a result, increases resource use.

If you know the range and precision of data that flows through processing code, you can curb fixed-point width growth by manually specifying the output width for functions along the processing path. This can be a tedious process that requires careful verification. NI recommends designing the algorithm and verifying its functionality through the use of a comprehensive set of unit tests. Once the basic algorithm works for wide input values, specialize it by tweaking function output widths and verifying numerical behavior using a set of functional tests.

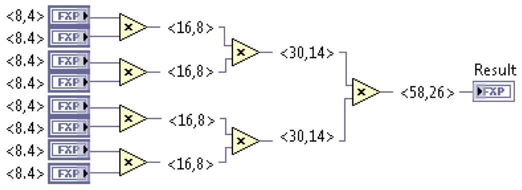


Figure 69. Fixed-point output width can grow quickly depending on the type of operation. In this case, LabVIEW automatically propagates and expands data widths from a set of 8-bit operands to a 58-bit result. You can manually curb this growth by overriding function output widths.

ARRAY AND CLUSTER OPTIMIZATION

Array operations use resources in direct proportion to the array size. For example, when incrementing an array by some scalar value, an add operation is instantiated for each element in the array. The same principle applies to clusters and larger data types. You can save resources on the FPGA and increase performance by minimizing the size of array and cluster types.

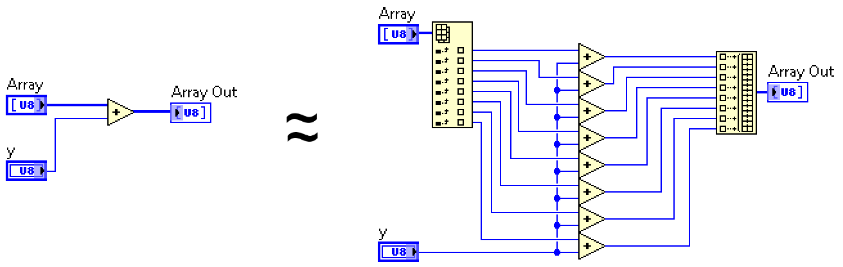


Figure 70. Array operations are implemented by replicating the operation for each element in the array. This can lead to a significant increase in FPGA resource use.

Some operations resolve to nothing more than a route on the FPGA at compile time. Array indexing using a constant value is optimized by the compiler as just a route between the storage element and the logic that consumes the stored value. Other elements in the array might be removed by the compiler if unused.

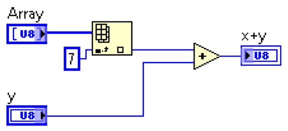


*Figure 71. This example adds a number to index 7 of the array input. There is no slice logic associated with retrieving index 7. Logic is only required to add index 7 to **y**.*

Connecting a control to the index array function requires additional logic to route the selected array element. Even if the value of index is held constant at run time, the compiler must insert circuitry to handle the case where the index value changes at run time.

MINIMIZING THE USE OF FRONT-PANEL CONTROLS AND INDICATORS

When optimizing FPGA resource use, you should minimize the number of controls and indicators you create on the front panel of your top-level FPGA VI. As examined in the Data Transfer Mechanisms section, each front-panel object on the top-level FPGA VI serves as a register item for communication between the FPGA and the host system, which requires storage and address logic. You should, therefore, try to remove any unused front-panel objects or replace them with diagram constants.

Creating a large array or cluster as a front-panel indicator or control is also resource intensive and should be avoided. Use FPGA Host Interface FIFOs, as explained in the Data Transfer Mechanisms chapter, if you need to transfer more than a dozen or so array elements to or from the host. If front-panel objects are used as a way to transfer data within the VI, you can replace them with global variables to save resources.

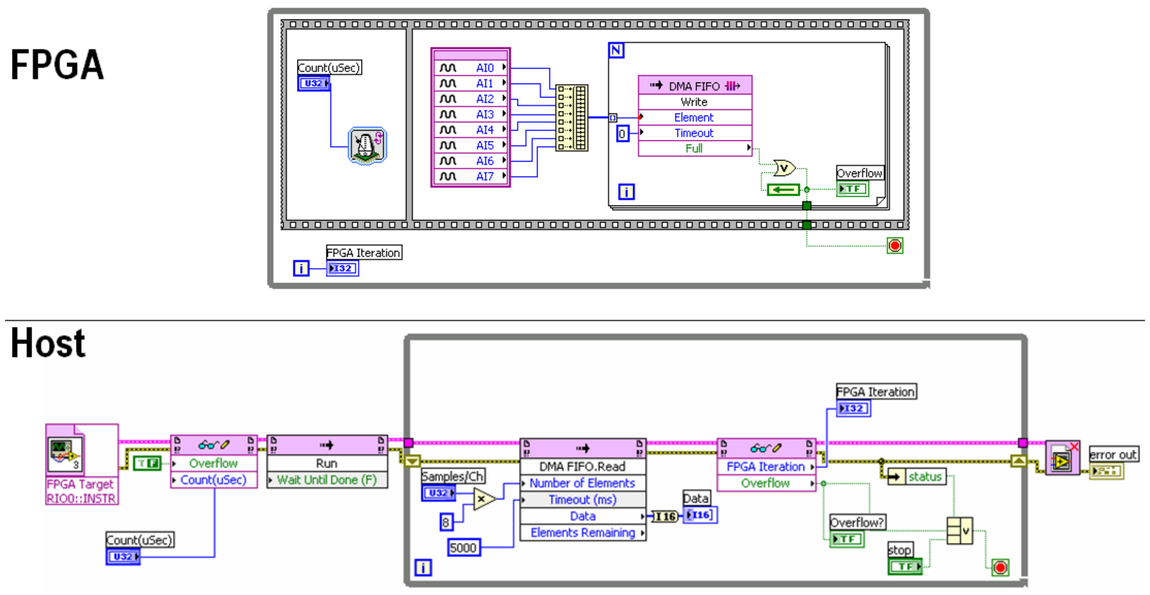


Figure 72. This example shows the use of Host Interface DMA FIFOs to transfer an array of analog samples to the host.

TWEAKING OUTPUT OVERFLOW AND ROUNDING OPTIONS

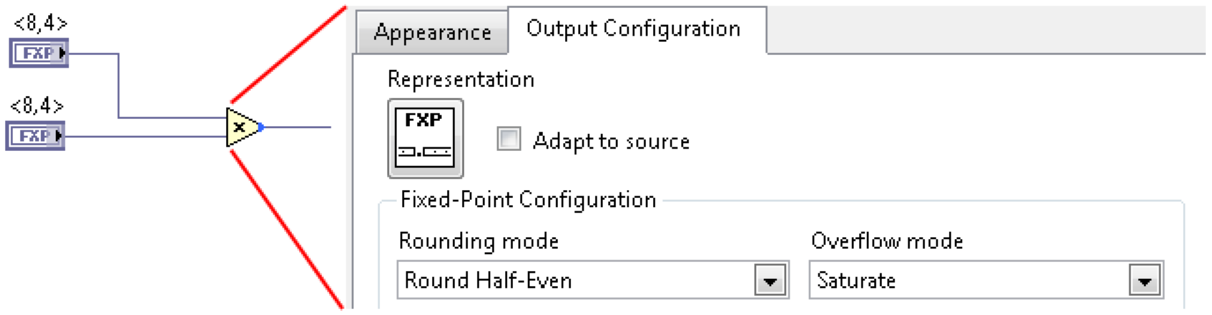
OVERFLOW OPTIONS

As explained in the Integrating High-Throughput IP chapter, overflow conditions occur when the type of output terminal is not large enough to hold the result of the operation. In general, because overflows are typically undesirable, LabVIEW adapts the output type width to avoid them. You can, however, use knowledge of the algorithm and the data it will process to use smaller output types and reduce FPGA resource use. You can configure most functions to handle overflow using **Saturate** or **Wrap** mode when you manually specify their output type.

Saturate mode determines if the value is within the range of the output type and coerces the value to the appropriate limit. This requires circuitry to detect overflow conditions and pick the appropriate overflow value for the output.

Wrap mode simply discards significant bits until the value is in the range of the output. This mode requires fewer FPGA resources, but it can lead to significant numerical differences when an overflow condition occurs.

When you manually specify the output type for standard fixed-point arithmetic operations, LabVIEW defaults to **Saturate** overflow mode to favor numerical accuracy. The High Throughput Math functions, discussed in the Integrating High-Throughput IP chapter, default to **Wrap** overflow mode to favor performance and resource efficiency. You can adjust the overflow mode on a per function basis to achieve the right balance between resource use and numerical precision.



Adapt to source: ☐

Fixed-Point Configuration
Rounding mode: Round Half-Even
Overflow mode: Saturate

*Figure 73. A blue dot shows up on the output terminal of standard fixed-point arithmetic functions when you manually specify their output type width, defaulting to **Saturate** overflow mode and **Round-Half-Even** rounding mode to favor numerical accuracy.*

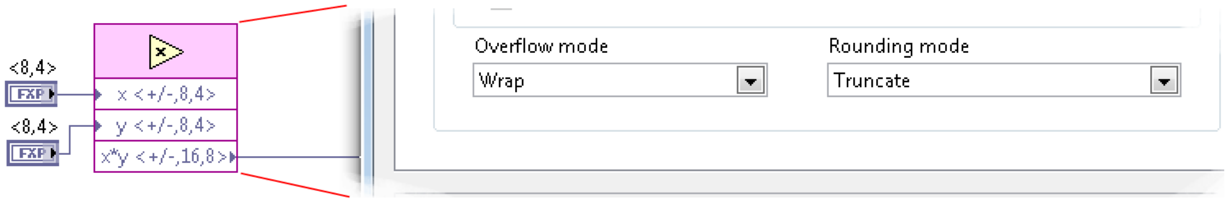
ROUNDING OPTIONS

Rounding occurs when the expected precision of a value is greater than the precision of the type that represents it. LabVIEW automatically picks output widths to maintain numerical precision. Inexact operators such as **Divide** and **Square Root** always require rounding of some sort. You can configure functions to perform rounding using **Truncate** , **Round-Half-Up**, or **Round-Half-Even** modes.

Truncate mode simply removes least-significant bits and, therefore, does not require any FPGA resources. However, this mode produces the largest mean error for most data streams because of its bias toward zero.

Round-Half-Up mode rounds to the nearest value that the type can represent. If the value to round is exactly between two valid values, this mode rounds the value up to the higher of the two values. LabVIEW adds one bit to the output value and then truncates it. This rounding mode produces more accurate output values than **Truncate** mode, with a smaller bias toward greater values. However, **Round-Half-Up** also has a greater impact on performance and resources because it requires additional logic proportional to the bit width of the output type.

Round-Half-Even mode rounds the value to the nearest value that the type can represent. If the value to round is exactly between two valid values, LabVIEW checks the bit of the value that becomes the least significant bit after rounding. If the bit is 0, this mode rounds the value to the smaller of the two values that the output type can represent. If the bit is not 0, this mode rounds the value to the larger of the two values. **Round-Half-Even** mode produces more accurate output values than the other modes, but it requires the most FPGA resources and has the most negative impact on performance.



Overflow mode: Wrap
Rounding mode: Truncate

*Figure 74. When you specify the output type width of High Throughput Math functions, they default to the **Wrap** Overflow mode and **Truncate** Rounding mode to favor performance and resource efficiency.*

When you specify function output type width, LabVIEW defaults to **Round-Half-Even** mode for standard fixed-point arithmetic operations to favor numerical accuracy. The High Throughput Math functions, discussed in the Integrating High-Throughput IP chapter, default to **Truncate** rounding mode to favor performance and resource efficiency. You can adjust the output width and rounding mode on a per function basis to achieve the right balance between resource use and numerical accuracy and precision.

INITIALIZING FEEDBACK NODES

Shift registers initialize every time the loop structure is entered. Like shift registers, the initial value of Feedback Nodes can be configured, and the way you initialize them can impact performance and resource use.

Unwired Feedback Nodes initialize on **Compile or Load** by default. The **Compile or Load** assigns the default value for the data type as the initialization value for the Feedback Node as soon as the FPGA bitfile is loaded onto the FPGA. If you wire a value to the initialization terminal of the Feedback Node, it switches its initialization mode to **First Call**.

The **First Call** option is useful if you want the Feedback Node to reset to the original value every time you run your FPGA VI. **First Call** initialization uses a multiplexer to select between initial and current values, which leads to slightly higher resource use. Most applications load the FPGA bitfile and run it once per device power cycle, so initializing on **Compile or Load** is a better option. If your host application loads the FPGA bitfile every time it runs, or if reusing the value from a previous execution is acceptable, you can use the **Compile or Load** option to save a multiplexer and get a slight increase in achievable clock rate for that code path.

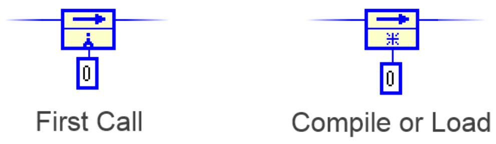


*Figure 75. With Feedback Nodes set to initialize on **Compile or Load**, you use fewer resources and can compile at higher rates.*

When using the **First Call** initialization mode, you select if the multiplexer is added before or after the register. This is an advanced optimization option that can help if you are having trouble breaking up the critical path into even delay segments to compile at higher rates. Moving the multiplexer to either side of the register shifts propagation and logic delay from one segment to the other, which allows for finer granularity when distributing path delays.

RESOURCE BALANCING

Resource balancing can help you resolve compilation failures in which you run out of a specific resource type. You can achieve more efficient FPGA resource use by specifying or encouraging certain resource types for parts of your design. Rebalancing might also help you run your design at higher rates since different resource types can run at different clock rates and require different amounts of routing logic between them.

Feedback Nodes, memory, FIFO, and register items also can take up a significant amount of resources, but they are often used for volatile storage and communication tasks, as explained in more detail in the Data Transfer Mechanisms chapter.

FEEDBACK NODES

Shift registers and Feedback Nodes are implemented using flip-flops unless the compiler can optimize one or more into a LUT. If your code does not depend on the initial value of the Feedback Node to function correctly, you can save additional resources by selecting the **Ignore FPGA reset method** option on the **FPGA Implementation** page of the Feedback Node **Properties** dialog. This allows the compiler to remove the reset logic from the register and implement it using shift register lookup tables (SRLs) instead of flip-flops. SRLs can combine multiple delays into a single LUT, which can significantly reduce FPGA resources relative to flip-flops.

The Discrete Delay function behaves similarly to a Feedback Node, but it uses SRLs to implement multiple cycles of delay. The Discrete Delay function supports varying the amount of delay at run time, up to a preconfigured maximum, for scalars and arrays of integer, fixed-point, and Boolean data types, clusters, and arrays of clusters.

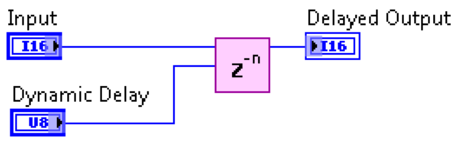


Figure 76. Discrete Delay blocks operate solely on Boolean data, but they are implemented using SRL resources, so you can use them in place of Feedback Nodes to balance resource use.

FPGA FIFOS

FPGA FIFOs can be used as storage or as a way to pass data between loops on the FPGA. You can specify the type of resource used to implement FIFOs on the FPGA to balance resources by choosing between flip-flops, LUTs, or block memory (BRAM).

Flip-flops provide the fastest performance, but you should use them only for small FIFOs, up to 100 bytes in size, because they consume considerable FPGA logic per storage element.

LUTs are more resource efficient than flip-flops, so they are adequate for larger FIFOs in the 100 byte to 300 byte range.

Block memory FIFOs store data using specialized blocks of memory distributed throughput the FPGA fabric. You can use this option for FIFOs larger than 300 bytes and up to several kilobytes, depending on the FPGA device. As described in the Data Transfer Mechanisms chapter, block memory FIFOs are the only FIFO implementation that can transfer data across clock domains. They also have longer latency than the other two implementation options.

You can specify the implementation resource for the FIFO control logic when using block memory FIFOs. Newer FPGAs are equipped with built-in FIFO controllers that can save a significant amount of FPGA resources and support higher clock rates than control logic using slice fabric.

ARRAY CONSTANTS

You can optimize your FPGA application by specifying how LabVIEW implements array constants. You should consider choosing a block memory implementation for array constants, unless you need the advantages of a different type of memory or need to free up block memory for other tasks. Block memory does not consume FPGA logic resources and tends to compile at a high clock rate, with respect to other types of memory.

By default, LabVIEW automatically decides to implement an array constant in block memory, LUTs, or flip-flops based on specific coding patterns. Arrays stored in Feedback Nodes might use block RAM as the implementation resource and can be treated as RAM. When using block RAM, only one reader and one writer can access the array using the standard array functions inside the SCTL. Accessing block RAM also takes one whole clock cycle, so a Feedback Node must be placed right after each access.

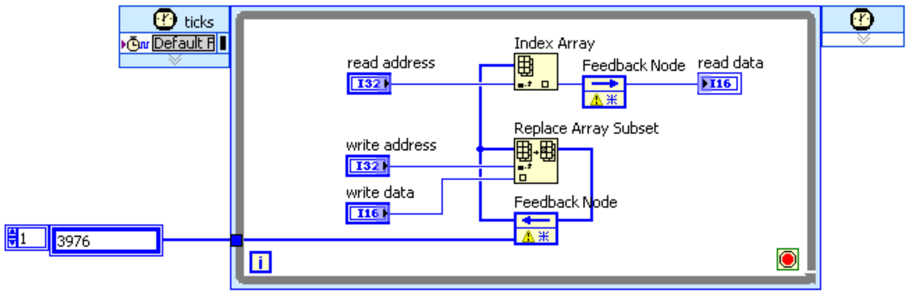


Figure 77. This array constant is implemented as block RAM and stored in the Feedback Node, which allows read and write access within one cycle of the SCTL by simply using the standard array functions.

If only read access is required, you can place the array inside the SCTL with an Index Array function and a Feedback Node immediately after to promote block RAM as the array implementation resource.

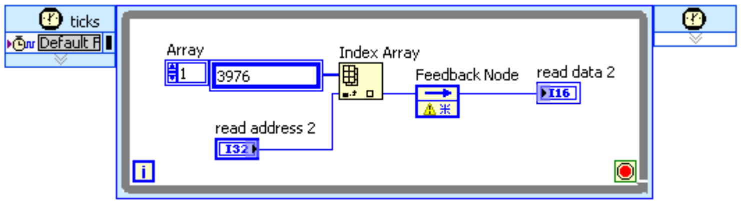


Figure 78. Read-only arrays also can be implemented in block RAM by using the above pattern.

DUAL-PORT ACCESS TO MEMORY ITEMS

Configure the memory for dual-port read access on the **Interfaces** page of the **Memory Properties** dialog to reduce the use of block memory resources and improve execution time.

With two read ports you can access data simultaneously on two different addresses. You can place the two Memory Method nodes that read the dual-ported memory within one structure, such as an SCTL, or in different places in the FPGA VI.

MULTIPLEXING LOGIC

Logic reuse is another way to optimize resources. You can use the same circuitry to process independent data streams over time by running the processing code at higher clock rates and multiplexing access to it over time. One way to achieve this with the SCTL is to move your processing logic to a separate SCTL with a clock rate that is a multiple of the original SCTL clock rate. You must use block RAM FIFOs to transfer data across these clock domains and, then, use a common set of blocks to alternate sample processing.

Code multiplexing might make sense only for relatively large processing tasks because of bookkeeping and communication overhead. Communication overhead refers to the latency, resource, and potential rate tax incurred by moving samples between two loops. Bookkeeping overhead refers to logic that must be added to keep track of which sample needs to be processed at a given time. Processing code that relies on state over time must also store and retrieve the appropriate state information, as it alternates between independent data streams.

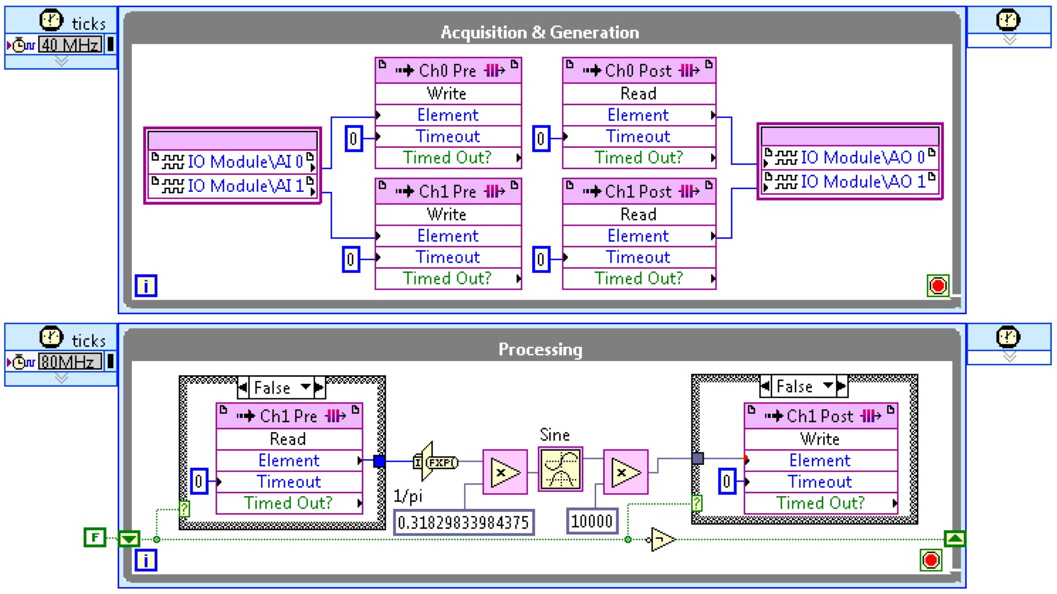


Figure 79. Processing code is multiplexed over time across two independent channels of data. The acquisition loop reads from the I/O module and transfers both samples over dedicated FIFOs to the processing loop. The processing loop runs at twice the rate of the acquisition loop and alternates reading from each FIFO, performing some processing and then writing the result back to the appropriate return FIFO. This approach halves the resources needed for processing at the expense of some communication and bookkeeping overhead.

DSP operations are good candidates for multiplexing because they typically consume more resources than digital logic tasks. Expensive operations, such as division, square root, and other complex IP, are good candidates for multiplexing if you are running out of space on the FPGA.

Floating-point operations are also possible in the SCTL when you use Xilinx CORE Generator IP or create your own IP using the LabVIEW FPGA IP Builder add-on. Single-precision floating-point operations can use a significant number of resources but can simplify the implementation of certain algorithms due to the dynamic range of the type. As a result, single-precision floating-point operations are also good candidates for multiplexing.

When working outside the SCTL, you can multiplex subVIs by making them nonreentrant and place multiple copies on your diagram. LabVIEW arbitrates access to each copy, which results in serialized execution and increased resource efficiency. FPGAs are much faster than I/O for many control and measurement applications, so serialized access can often still meet throughput and latency requirements while conserving FPGA resources.

USING THE SCTL AS A WAY TO SAVE RESOURCES

Even if your existing FPGA application does not include an SCTL, you might consider using one to optimize resources. LabVIEW automatically optimizes code inside the SCTL to execute more quickly and consume less space on the FPGA, compared to the same code inside a While Loop.

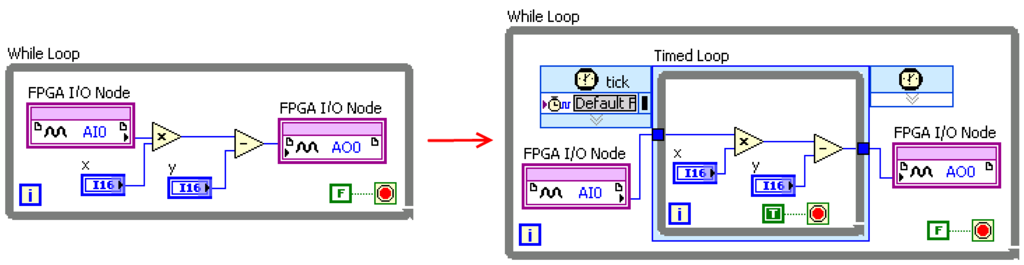


*Figure 80. You can use an SCTL within a While Loop. Wire a constant to the SCTL Stop condition or use the **Output Valid** signal of four-wire handshake IP inside to stop the loop and pass the results down to other non-SCTL code.*

Additional Resources

[1]	Using NI LabVIEW FPGA IP Builder to Optimize and Port VIs for Use on FPGAS http://www.ni.com/white-paper/14036/en/
[2]	Optimizing Memory Usage of Array Constants http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_array_memory_implement/

This page intentionally left blank.

DATA TRANSFER MECHANISMS

Data transfer is an important aspect of high-performance LabVIEW FPGA applications that are implemented as concurrent processes within a system. These concurrent processes take the form of parallel loops on one FPGA, across multiple FPGAs, or on the host system. From this general view of the system, communication can take place within the FPGA, between the FPGA and the host, between FPGA devices, or between FPGA devices and other instruments.

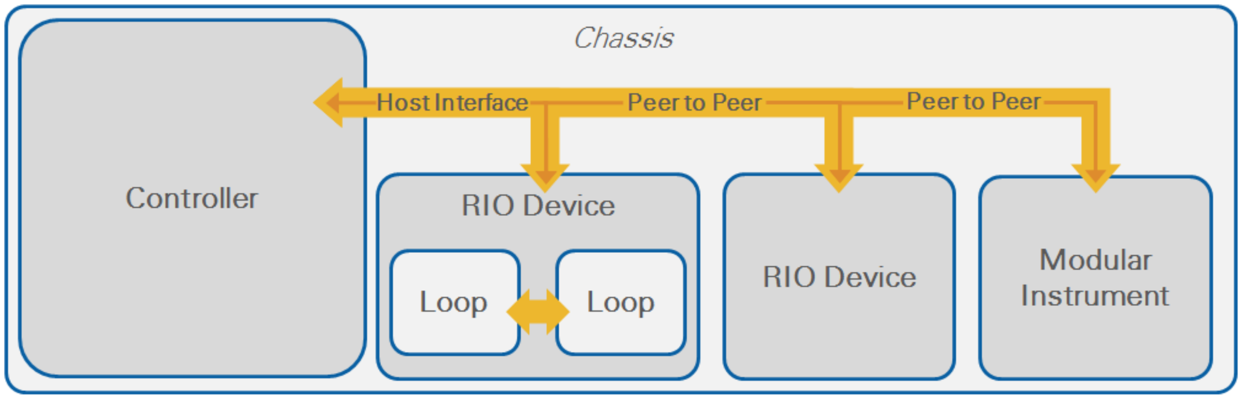


Figure 81. Data transfer can occur between loops within the FPGA, between FPGA devices, between FPGA devices and other devices, or between FPGA devices and the host system.

This chapter provides a summary of the different data transfer mechanisms and their relevant performance characteristics.

THROUGHPUT AND LATENCY OF DATA TRANSFER MECHANISMS

The efficiency of data transfer mechanisms affects application performance when the application spans multiple devices or processes on a device. This section covers concepts for analyzing the performance of data transfer mechanisms discussed in later sections.

You can model any data transfer mechanism as the exchange between two processes (a source and a sink), where the data flows from the source to the sink over some channel.

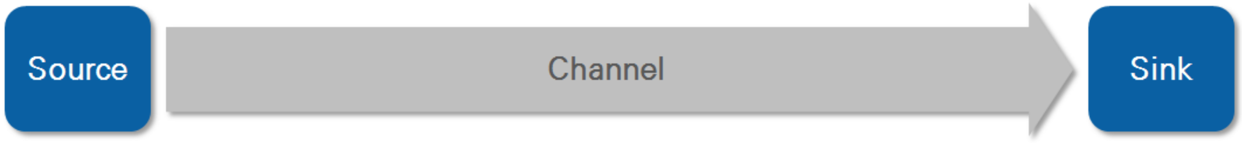


Figure 82. Data transfer can be modeled as two peers exchanging data over a channel. The source produces the data and the sink consumes it.

Source, sink, and channel determine the effective throughput of the data transfer mechanism. The source determines how much data can be provided per unit of time. The channel must have enough bandwidth to allow that amount of data to flow through it. The sink must be able to store or process the data as it arrives.

Bandwidth is the maximum achievable throughput for a given channel configuration. Bandwidth is different from throughput—throughput refers to the actual amount of data passing through the channel. You may need to use specific transfer patterns or write data in specific sizes to achieve throughput rates that approach the channel bandwidth.

A data source might transmit data intermittently, in bursts. If the burst pattern is predictable, you can define its duty cycle as the percentage of time the channel is busy transferring data. There might also be a minimum transfer size, defined by the source or channel as the minimum amount of data needed before a transfer initiates.

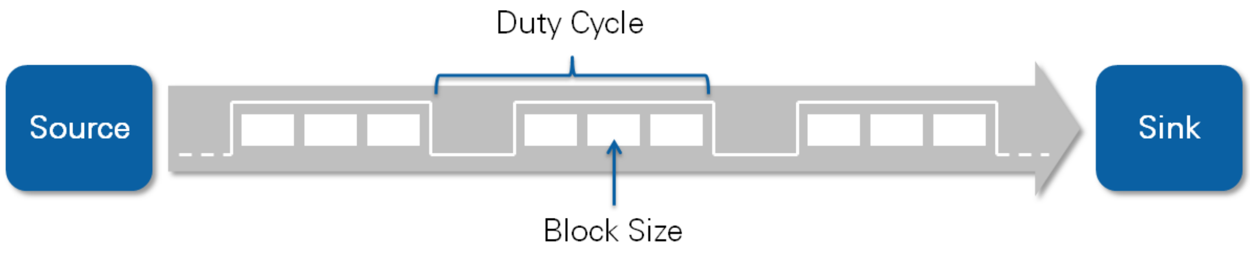


Figure 83. Data transfer bursts can consist of one or more blocks. When bursts are uniform and periodic, the effective throughput of the channel is the channel bandwidth times the duty cycle.

Data bursts, unpredictable transfer patterns, and other transient channel or source conditions, such as bus collisions or jitter, lead to peak and average throughput. Average throughput is defined as the sustainable throughput over an indefinite amount of time. If the source, sink, or channel cannot handle the desired average throughput, then the transfer eventually fails. Peak throughput can temporarily exceed the average throughput, but it cannot exceed the channel bandwidth.

Data transfer only succeeds if the sink is able to accept data at peak throughput rates, or at the average throughput rates using buffers to temporarily store and process the data in batches.

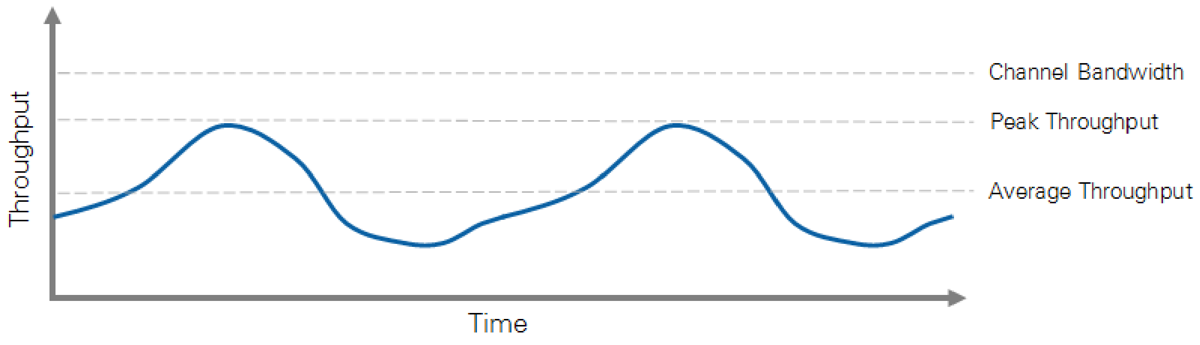


Figure 84. The effective throughput may vary over time, but it may not exceed the channel bandwidth. Source and sink must handle the average throughput to sustain the transfer indefinitely. Source and sink must also handle temporary peak throughput conditions, usually through buffering.

Buffers absorb expected data bursts as well as unexpected transient conditions, thus helping to sustain the desired average throughput. Larger buffers can absorb larger transient events at the cost of increased resource use and higher latency.

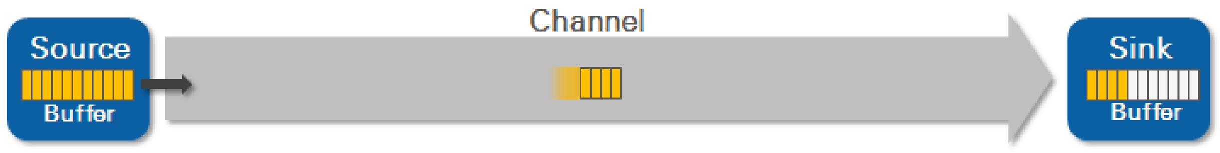


Figure 85. Buffering helps handle transient conditions by temporarily storing data on the source or sink.

Data transfer latency is directly proportional to the amount of data in flight plus the latency of the communication channel. Intra-FPGA communication and local communication buses, such as PCI Express, are usually fast and do not hold much data in the channel at any given time. As a result, most of the latency stems from data in the source and sink buffers. You can estimate the latency by multiplying the expected amount of data in flight by the average throughput of the communication mechanism.

TRANSFERRING DATA WITHIN THE FPGA

MULTILOOP AND MULTIRATE DESIGNS

It is common to distribute high-performance LabVIEW applications across multiple loops on the diagram. The following are considerations for when to distribute across multiple loops:

Task Separation—A good programming practice is to map logically separate tasks into independent loops when possible, which eases the maintenance and readability of the code. When architected this way, loops take on the role of independent processes, which leverage FPGA parallelism for concurrent operation. Multiple processing loops may act on the same data streams, so it is important to understand the performance implications of passing data between them.

Forced clock domains—Certain FPGA constructs can be used only in specific clock domains, which forces parts of your application to be in a specific clock domain. This is typically the case when applications interact with components external to the FPGA, such as I/O or DRAM. An application that reads from a high-speed input source and streams the data to DRAM is forced to have at least two loops: one for input and another to write to DRAM. This, in turn, requires a communication mechanism between the loops.

Clock-domain optimization—When specific I/O constructs force you to use a specific clock domain, you need to know that the rest of the processing tasks may be placed in other clock domains. For example, after averaging or decimating a data stream, you can work with a lower throughput stream, meaning that you might be able to clock downstream functions at lower rates. The lower rates help reduce compile times and resource use as well as increase the odds of a successful compilation.

The Resource Optimization Techniques chapter covers an example of this approach where, to save resources, a processing function is shared over time by running it in a separate loop.

Different loop types—Finally, you might use different loop types throughout your FPGA design. Not every part of an application requires the performance benefits of the SCTL. As a result, you might decide to split your application into multiple loops to avoid the relatively more strict requirements of the SCTL.

Separating your application into multiple loops can introduce data-communicating challenges. Logic within communicating SCTLs executes in different clock domains if they use different clocks. When SCTLs define different clock domains, there is potential for data to get lost or overwritten if you are not careful when picking and configuring the communication mechanism.

Different communication mechanisms provide varying degrees of synchronization and buffer policies to maintain data coherence, often at the expense of performance or resource use. Additional synchronization mechanisms, such as data tagging or synchronization variables, can be used to guarantee execution ordering between loops and to augment communication mechanisms when necessary.

INTERLOOP COMMUNICATION MECHANISMS

LOCAL AND GLOBAL VARIABLES

Variables broadcast information from one loop to another. Variables are not buffered, so they are useful when only the latest written value is needed. Variable use in the SCTL is restricted to a single writer, but multiple readers are allowed.

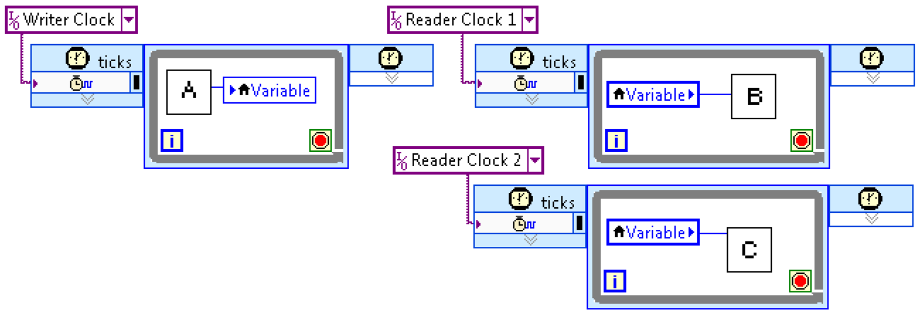


Figure 86. Variables are used to store and publish a value across loops.

Access scope for variables varies depending on the type of variable. Local variables are accessible only within the VI that defines them. Global variables are accessible from any VI running on the FPGA. Local variables have front-panel presence, in the form of controls or indicators. As a result, global variables or registers are a better alternative if you need to save resources, as discussed in the Resource Optimization Techniques chapter.

Data coherence is guaranteed for all the bits in a given variable. In other words, the written value is coherent across all of its bits when read by the receiver. Coherence across multiple variables is not guaranteed, so you must bundle them as part of a cluster or overlay synchronization mechanisms if the value of multiple variables at a time is important. Consider using a handshake item, as explained in the **Error! Reference source not found.** section of this chapter, in cases where you would use a variable but eed to synchronize a writer and a reader.

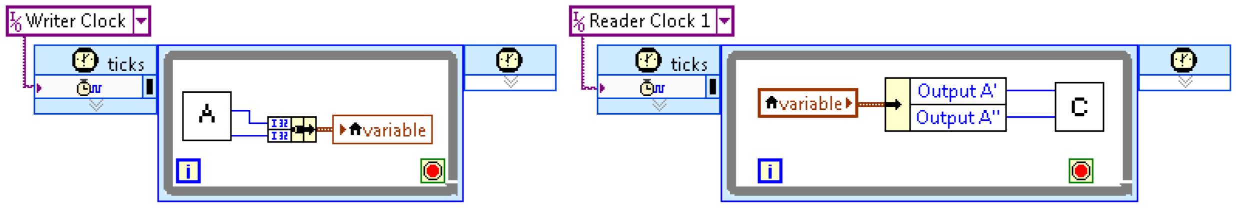


Figure 87. This example uses a cluster variable to guarantee coherence when sharing multiple values across loops.

In terms of latency, a variable may take a few cycles to propagate its value from the writer to any reader, and the number of cycles may vary between readers. This means you must take appropriate steps, in the form of additional synchronization or tagging, if you need multiple readers to share the same variable value at any given time. Additionally, you may not be able to tell how long values take to transfer between loops and, as a result, you need to instrument your code or provide additional synchronization mechanisms to detect when data is written or read.

Because variables do not offer built-in buffering or synchronization mechanisms, they are usually not employed for data streaming. As a result, their throughput characteristics are not as relevant as their latency.

REGISTER ITEMS

Register items are similar to variables in functionality, resource use, and performance. In contrast to variables, you can obtain a reference to a register item by using the VI-Defined Register Configuration node. Register item references are used to write reusable code by passing reference wires to subVIs. These references to register items are resolved at compile time.

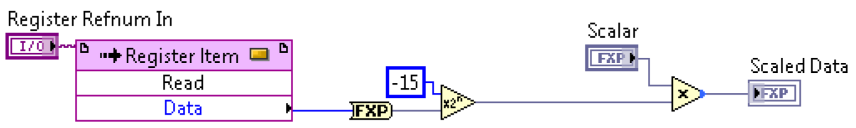


Figure 88. You can write reusable subVIs that process shared data using register reference wires. The same subVI can be used throughout the design but may read data from different registers depending on the value of the reference connected at the caller VI.

FIFO ITEMS

FIFOs are used for high-throughput, lossless transfers between loops using a first-in, first-out access policy. FIFOs buffer data and are useful when the source or the sink might generate or accept data in bursts.

FIFOs can be project- or VI-defined, and have similar throughput and latency characteristics. Project-defined FIFOs transfer data between loops in different VIs, but they depend on the LabVIEW project to provide the global FIFO definition and configuration parameters at compile time.

VI-defined FIFOs are defined directly on the diagram using the VI-Defined FIFO Configuration node. The node instantiates a FIFO for every time the parent VI is instantiated on the diagram. The configuration node also provides a reference that you can use to write reusable code and transfer data across VIs.

FIFOs provide a **Time Out?** terminal to indicate **Read** or **Write** method success. A timeout when reading indicates that the FIFO is empty and the value of **Element** is undefined, so the value should not be used by downstream nodes. This signal is the equivalent of a negated **output valid** in the four-wire protocol, explained in the Integrating High-Throughput IP chapter.

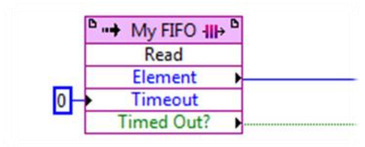


*Figure 89. FIFO Read operations inside the SCTL must use a timeout of zero. The **Time Out?** output indicates an empty FIFO.*

A timeout when writing indicates that the FIFO is full and the element could not be written. If the application cannot tolerate data loss, you should add logic to cache the attempted value and retry the write operation.

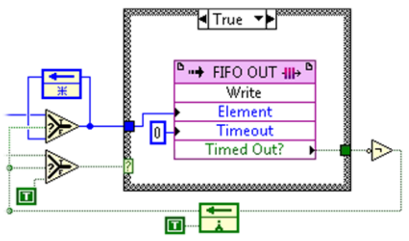


Figure 90. Additional logic must be used to handle the case where writing to the FIFO results in a timeout condition. The attempted value is cached using a Feedback Node and upstream code is prevented from producing additional values until there is room in the FIFO and the write operation succeeds.

FIFOs also expose a way to query how many elements can be read or written at a given time. These methods add circuitry that keep track of and expose the FIFO element count. The additional circuitry has a negative effect on the clock rate that the SCTL can achieve. Still, these methods are useful when FIFOs interact with IP that generates data in bursts. If the size of the expected data bursts is not known, it is typically better to simply use the **Timed Out?** Terminal and allow the handshaking to occur as needed.

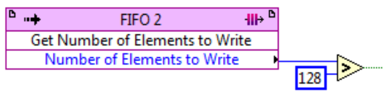


*Figure 91. The **Get Number of Elements to Write** method is typically used to keep bursty IP from executing until the FIFO can accept the expected output data burst.*

As discussed in the Resource Optimization Techniques, you can specify the implementation resource type for project- and VI-defined FIFOs.

When passing data between SCTLs that share the same clock domain, the flip-flop implementation provides the best performance but uses a significant number of slices, so it should be reserved for small FIFOs (up to 100 bytes or so). LUT-based FIFOs provide an intermediate option for FIFOs in the 100 bytes to 300 bytes range.

Block memory is the only FIFO implementation that supports transferring data across clock domains. In other words, block memory FIFOs support transferring data across diagrams using different rates or clocks that derive from different bases. You can write data to a block memory FIFO from only one clock domain and read from another.

When the FIFO is written and read from SCTLs in the same clock domain, LabVIEW uses a more lightweight synchronization implementation that results in lower latency and potentially higher clock rates.

Use block memory FIFOs when implementing relatively deep FIFOs (greater than 300 bytes). Block Memory FIFOs can have up to six clock cycles of latency before data written to it reaches the reader.

HANDSHAKE ITEMS

Handshake items are unbuffered mechanisms that use handshaking to achieve lossless transfer between a single writer and a single reader. Handshake items are like FIFOs, with a depth of one element, but exist as their own mechanism to get around implementation details that currently prevent FIFOs from being configured this way. Handshake items can only be used inside the SCTL, can be used to communicate across clock domains.

Like FIFOs, handshake items offer an option to clear their contents. They also provide a way to perform nondestructive reads in which the reader reads the contents of the handshake item without emptying it or signaling to the writer that the data has been read, and then later issues an explicit acknowledge so that the writer can provide a new value. This presents a better alternative to variables when the reader and writer must be synchronized.

Handshake items require several clock cycles to transfer a value across clock domains. This directly translates into communication latency. Like variables and register items, handshake items offer no buffering, so they are not designed for data streaming and their throughput performance is not as relevant as their latency.

As with registers and FIFOs, handshake items are VI- or project-defined. The performance characteristics of both types are identical with respect to throughput and latency. Handshake items consume fewer FPGA logic resources than FIFOs, and they do not consume block memory.

MEMORY ITEMS

Memory items represent the primary FPGA storage mechanism and transfer data within the FPGA. They provide a generic addressable space that you can access from different parts of your application.

The implementation resource type for memory items can be configured to any of the following:

Block memory—Memory items using block memory compile at higher clock rates relative to other types of memory items. You can configure block memory for read-write access or dual-port read access. You also can use a memory item implemented using block memory to write data in one clock domain and read the data from a different clock domain. In this implementation, you can use only one writer node and one reader node for each memory item. Block memory does not consume FPGA logic resources for storage purposes.

When you use memory items implemented using block memory in multiple clock domains, it is possible to read from and write to the same address simultaneously; however, doing so can result in reading incorrect data. Memory items do not have built-in handshaking, so you need additional handshaking code if you want to avoid collisions between a reader and a writer addressing the same item, guarantee data coherence across the memory, or transfer data in a lossless manner.

In general, FIFOs are a preferable mechanism if lossless transfers are important. Memory items are useful if random access to the data is important and it is acceptable that readers know only about the most recent coherent memory state as opposed to every coherent memory state over time.

The block memory implementation requires an entire clock cycle to read the requested address. As a result, you must place a Feedback Node directly after its data output and read the **Data** value during the following clock cycle. Note that this adds a cycle to the overall latency of the read operation.

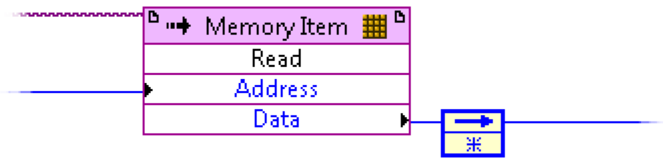


Figure 92. Memory items using block memory require a whole cycle to retrieve a value, so you must place a Feedback Node immediately downstream from the read operation.

Lookup tables (LUTs)—LUTs, also known as distributed RAM, consist of logic gates hard-wired on the FPGA. LUTs consume logic resources because they can function either as logic resources or as memory.

You do not need to wire the **Read** output directly to an uninitialized shift register or Feedback Node when the memory item is implemented with LUTs. You also can read from the memory item during the same cycle in which you provide the **Address** parameter.

Use LUTs for memory items when accessing the memory in an SCTL, when you need to read data in one cycle, or when you have limited remaining block memory.

Dynamic RAM (DRAM)—DRAM is a form of memory external to the FPGA and available on some NI RIO devices. DRAM provides a large amount of storage space, from hundreds to thousands of megabytes. As a result, you can use it to store data that would not fit elsewhere on the FPGA. However, because DRAM is external to the FPGA, the application cannot receive data from DRAM in a single clock cycle. You must use the **Request Data** and **Retrieve Data** methods to receive data. Sequential access to memory can prevent deterministic timing when the multiple read requests collide.

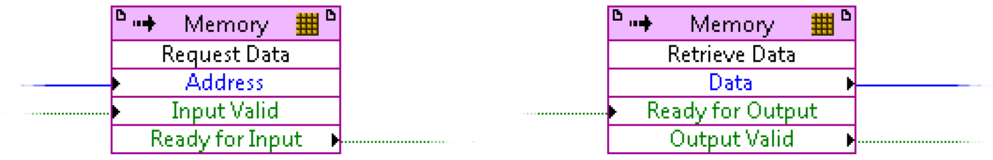


*Figure 93. DRAM reads are performed by requesting data from a given address and then retrieving it some cycles later when the **Output Valid** signal of the **Retrieve Data** method is **TRUE**.*

Sequential access to memory can prevent deterministic timing when the multiple read requests collide. Dynamic RAM must also be periodically refreshed and this causes additional indeterminism when the refreshed memory address collides with a read or write request.

To optimize DRAM performance, send requests for data or write data to the DRAM in bursts, such as on each clock cycle.

INTER-SCTL COMMUNICATION MECHANISM SUMMARY

The following table summarizes the characteristics of each for inter-SCTL communication mechanism.

Transfer Method	Lossy	Number of SCTL Accessors	FPGA Resource	Clock Domain Crossing[1]	Primary Use
Global Variables	Yes	1 Writer N Readers	Logic	Yes	Share the current variable value across one or more VIs.
Local Variables					Share the current variable value across the one VI with a front-panel item.
Register Items	Yes	N Writers[2,3] N Readers	Logic	Yes[4]	Write reusable IP that shares the current register value across one or more VIs.
Memory Items	Yes	N Writers[2,3,8] N Readers[2,8]	LUTs	No	Fast, random read/write access to small amounts of memory (hundred of bytes) from one clock domain.
			BRAM	Yes[4,5,6,7,9]	Random read/write access to moderate amounts of memory (kilobytes) from up from two clock domains.
			DRAM	Yes[5,6,7]	Longer latency, random read/write access to large amounts of memory (megabytes) from up to two clock domains.
Handshake Items	No	1 Writer 1 Reader	Logic	Yes	Write reusable IP that shares a value across up to two clock domains in synchronized fashion.
FIFOs	No[10]	N Writers[2,3] N Readers[2]	BRAM	Yes[3,4,5,7]	Lossless, large data transfer and storage buffer across loops in up to two clock domains.
			LUTs	No	Lossless, moderate data transfer and storage buffer within one clock domain.
			Flip-Flops		Fast, lossless, small data transfer and storage buffer within one clock domain.

1. Clock domain crossing denotes whether the mechanism can be used to pass data between SCTLs in different clock domains. All of these mechanisms can be used to pass data between SCTLs using the same clock domain.
2. You must explicitly disable arbitration if you need multiple accessors.
3. For multiple writers, you must implement your own synchronization mechanism to prevent data corruption or incorrect addressing due to writer collisions.
4. All writers must reside in the same clock domain.
5. All readers must reside in the same clock domain.
6. You must enable the **Dual-clock** option to cross clock domains.
7. For multiple readers, you must implement your own synchronization mechanism to prevent data corruption or incorrect addressing due to reader collisions.
8. Memory items using block memory support multiple reader SCTLs. Use dual-ported BRAM interface with only one writer and one reader to minimize access jitter
9. When you use memory items implemented using block memory in multiple clock domains, it is possible to read from and write to the same address simultaneously. However, doing so can result in reading incorrect data.
10. The **Timed Out?** terminal is used to determine **Read** or **Write** method success.

TRANSFERRING DATA BETWEEN THE FPGA AND THE HOST SYSTEM

The FPGA host interface aggregates the primary mechanisms for transferring data between the host and the FPGA. Host-based processing can play a key role in LabVIEW RIO applications, so it is important to understand the latency and throughput characteristics of the different FPGA interface mechanisms. This section deals with the performance aspects of those mechanisms and their recommended uses.

You may want to use the FPGA host interface to transfer data between the host computer and the FPGA to:

- Implement more data processing than you can fit on the FPGA
- Perform operations not available on the FPGA target, such as double-precision or extended-precision floating-point arithmetic
- Create a multitiered application, with the FPGA target as a component of a larger system
- Log data to disk

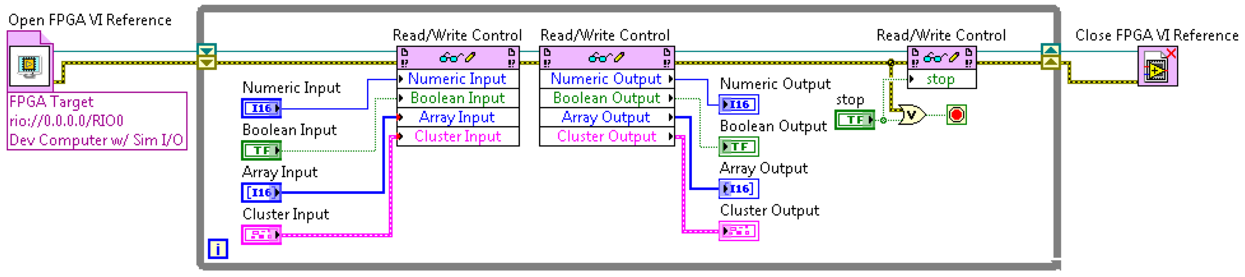


Figure 94. The FPGA host interface is the primary mechanism for transferring data between the FPGA and the host. It uses an object-oriented paradigm where top-level FPGA VI controls and indicators are accessed through property nodes, and methods can be invoked on the interface to perform different operations such as DMA FIFO access.

TOP-LEVEL VI CONTROLS AND INDICATORS

The LabVIEW FPGA Module creates a register map with a set of hardware registers for every control and indicator on the top-level FPGA VI. The FPGA host interface provides read and write access to these registers. Those familiar with driver and low-level I/O programming can think of this as memory-mapped register access (MMRA) to the FPGA.

The Read/Write Control function supports scalar data, such as numeric and Boolean controls, and structured data, such as arrays and clusters. As covered in the Resource Optimization Techniques chapter, do not overuse arrays on the front panel of your FPGA VIs because they consume significant FPGA resources.

Top-level FPGA VI controls and indicators are the lowest latency option for communicating with the FPGA. Although machine dependent, reads and writes to the FPGA are typically in the microsecond range. This provides an efficient way to exchange status and set configuration parameters. Use a multicore host and dedicate a CPU core to reading from the specific registers to achieve the lowest latency and tighter synchronization. Using a LabVIEW Real-Time host offers good mean latency and adds determinism, which bounds the maximum access latency to the FPGA.

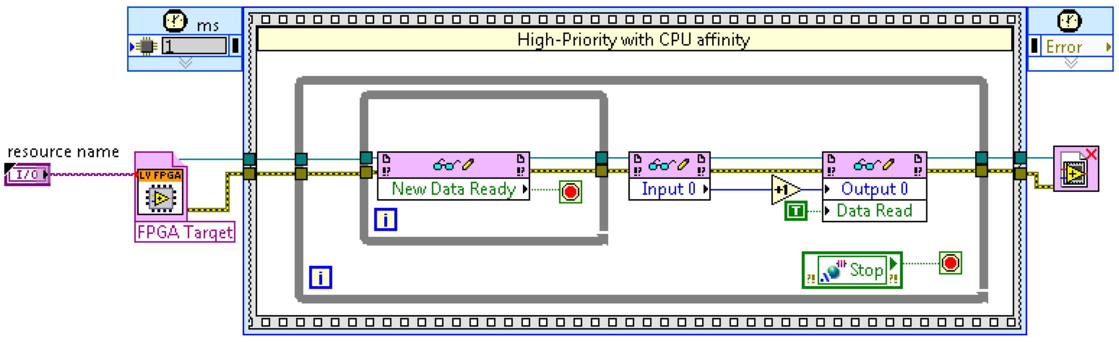


Figure 95. A timed sequence structure assigns CPU affinity and high priority to a host-side loop that polls on an FPGA VI control. This approach provides the lowest latency for transferring time-sensitive data between the FPGA and the host.

Keep in mind that the FPGA can update host interface registers much faster than the host can read them. As a result, it is easy for the host application to miss values. Additional handshaking or different synchronization mechanisms are required if you want data coherence across multiple reads or want to transfer data between the FPGA and the host in a lossless fashion. FPGA host interface FIFOs access host memory directly and are a recommended way to transfer large amounts of information between the FPGA and the host.

FPGA HOST INTERFACE FIFOS (DIRECT MEMORY ACCESS)

FPGA host interface FIFOs use DMA to buffer and transfer data to host system memory at high speeds, with little processor involvement. This is an efficient mechanism when sending large blocks of data compared to front panel controls and indicators.

The API for FPGA host interface FIFOs is similar to that of standard FPGA FIFOs. Host interface FIFOs are a unidirectional transfer mechanism, and you can configure them to transfer host-to-FPGA or FPGA-to-host. The API for the host interface FIFOs varies depending on if you are accessing them from the FPGA or host side.

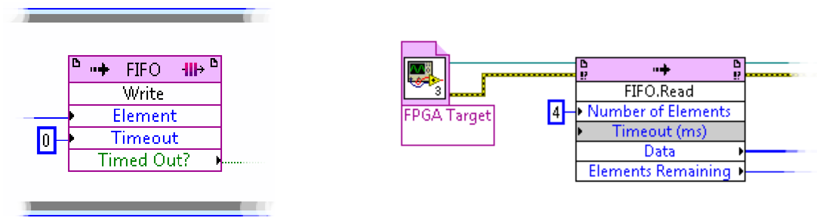


Figure 96. Host interface FIFOs abstract away the complexity of DMA communication between the FPGA and the host system. This example shows how you can write one element at a time from the FPGA side and read multiple elements at a time on the host.

CONFIGURING FPGA HOST INTERFACE FIFOS FOR BEST PERFORMANCE

Your code must handle and protect against transient conditions to achieve the best possible throughput with host interface FIFOs. As discussed later in this section, FIFOs can be "primed" with data, and the initialization order of the DMA engine, the source, and the sink can be controlled to minimize the likelihood of errors during the early stages of the transfer process.

A DMA channel consists of two FIFO buffers: one on the host computer and one on the FPGA target. Each side operates on its respective buffer and the DMA engine transfers data from one to the other once certain conditions are met.

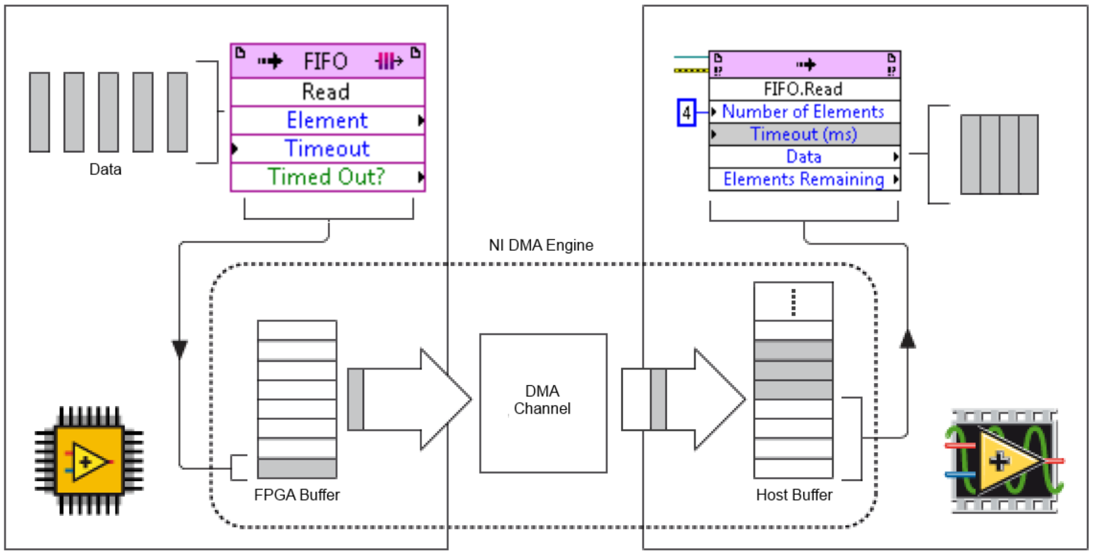


Figure 97. The DMA engine transfers data over the local bus to directly access host memory for transfer speeds that approach the bus bandwidth with little host CPU load.

The host-side buffer can be configured by the host application at run time before the transfer process starts. The current default size for the host-side buffer is the greater of 10,000 elements and twice the FPGA-side buffer size. NI recommends that you increase this buffer to a size multiple of 4,096 elements if you run into overflow or underflow errors. As a guideline, set the host-side buffer to be at least five times the number of elements you intend to read or write at a time. Increase the host-side buffer to hold several seconds of data if latency is not a concern.

The FPGA-side buffer is implemented using logic resources on the FPGA. You can configure the size of the FPGA-side buffer before compilation using the FIFO Properties dialog box accessible through the FIFO item in the LabVIEW project. The FPGA-side buffer defaults to 1,023 elements of depth. Because the FPGA can service this buffer much faster than the host can, and because the FPGA is usually resource constrained, NI recommends keeping this buffer size unchanged unless there is strong evidence that increasing it will improve performance. Once your design is close to final, and if latency is not a major concern, you may want to consider increasing the FPGA-side buffer as much as the compilation process allows.

The FPGA is often much faster than the host system with respect to servicing and providing data to the DMA engine, so the host CPU works harder to keep up with the FPGA when performing DMA FIFO transfers. The FPGA is also more resource constrained than the host in terms of temporary buffer space. As a result, the CPU should read as often as possible to keep the FPGA buffer small.

When the FPGA transfers data to the host, the CPU attempts to keep FPGA buffers as empty as possible so that the FPGA has a place to store data in case a transient condition temporarily affects the CPU or local bus. When the host transfers data to the FPGA, the host aims to keep the FPGA buffers as full as possible so that the FPGA has enough data to process in case a transient condition occurs.

When transferring data from the FPGA to the host, you need to understand that the DMA engine does not start until the host performs a **Read** or **Start** method call. The FPGA quickly overflows its FIFO buffer if the DMA engine is not running yet; therefore, you should start the DMA by calling the **Start** or **Read** method on the host before writing to the FIFO on the FPGA side.

Once the transfer has started, the host reads from the host-side buffer by calling the **Read** method. If the host-side buffer fills up, the DMA engine stops transferring data and the FPGA-side FIFO reports the overflow as a timeout condition.

The host-side **Read** method uses a polling mechanism to check for the requested amount of data before timing out. Dedicating a CPU core to this process by placing the **Read** method inside a Timed Loop or a Timed Sequence structure on the host VI provides the best performance at the expense of high CPU load. If the host side is keeping up with the DMA stream, you might want to use a low timeout value and explicitly call a **Wait Microsecond** function when the read call returns no data, so that you can share CPU time with other processes on the system.

When transferring data from the host to the FPGA or if it is important that the FPGA always has data to process or output, write to the FIFO before starting the process that reads from it on the FPGA side. In this case, larger host-side buffers are better for surviving transient conditions, but they increase transfer latency.

HOST INTERFACE FIFO LATENCY AND THROUGHPUT

As previously discussed, latency is directly proportional to the amount of data in flight plus some additional mechanism overhead. The majority of the data in flight is in either FPGA- or host-side buffers.

When transferring data from the FPGA to the host, the host-side buffer should be nearly empty in the steady state. Additionally, the FPGA-side buffer should be relatively small, so the latency is determined by how often the DMA engine transfers blocks of data to the host. The DMA engine transfers data to the host whenever any of the following conditions are met:

- The FPGA-side buffer is one quarter full
- The FPGA-side buffer has at least 512 bytes (a full PCI Express packet)
- The eviction timer of the DMA controller fires—this timer has a period of approximately one microsecond

When transferring data from the host to the FPGA, the host-side buffer should be nearly full in the steady state. Additionally, the FPGA-side buffer should still be relatively empty, so the average latency is mostly determined by the size of the host-side buffer divided by the average throughput. Average throughput depends on how often the FPGA reads from the DMA FIFO and is limited to the maximum system bus bandwidth.

FPGA DMA system bandwidth is dependent on the bus technology and platform that you choose. Bus bandwidth increases quickly as NI adopts the latest bus technologies, so be sure to review the product documentation for the most up-to-date specifications. Examples of the current bandwidth capabilities of NI hardware products include the following:

- NI FlexRIO devices currently offer the highest FPGA-based DMA throughput by leveraging PCI Express technology in a PXI Express chassis.
    - The NI PXIe-797xR for NI FlexRIO series uses four PCI Express 2.0 lanes to achieve 1.7 GB/s of unidirectional throughput and 1.2 GB/s per direction when transferring data in both directions.
    - The NI PXIe-796xR for NI FlexRIO series can achieve 800 MB/s to the FPGA, 838 MB/s from the FPGA, and 800 MB/s per direction when transferring data in both directions.
- The NI cRIO-9068 uses a high-speed AXI bus to deliver 300 MB/s for a single DMA channel or 300 MB/s aggregated over 16 DMA channels.
- The multicore NI cRIO-9082 uses a PCI bus to connect the FPGA to the processor and deliver a typical bandwidth of 90 MB/s to 100 MB/s.

REDUCING BUFFER COPIES IN HOST INTERFACE FIFOS

When reading or writing to host interface FIFOs, data is copied between LabVIEW and NI-RIO driver memory buffers. These additional data copies consume CPU cycles and limit the size of applications that you can build using FIFOs. The LabVIEW FPGA Module expands the DMA FIFO API to provide direct access to the DMA buffer on the NI-RIO driver side by combining the **Acquire Data Region** and **Acquire Write Region FIFO** methods with the Data Value Reference concept in LabVIEW [3].

Using the region API, you can obtain a data value reference to the NI-RIO buffer and directly operate on the data within the boundaries of the In Place Element structure. Reading the referenced data within the structure does not copy it, and any write operations go directly to the driver-side buffer. You must delete the data value reference after use to allow the driver to reuse the memory for subsequent transfer operations.

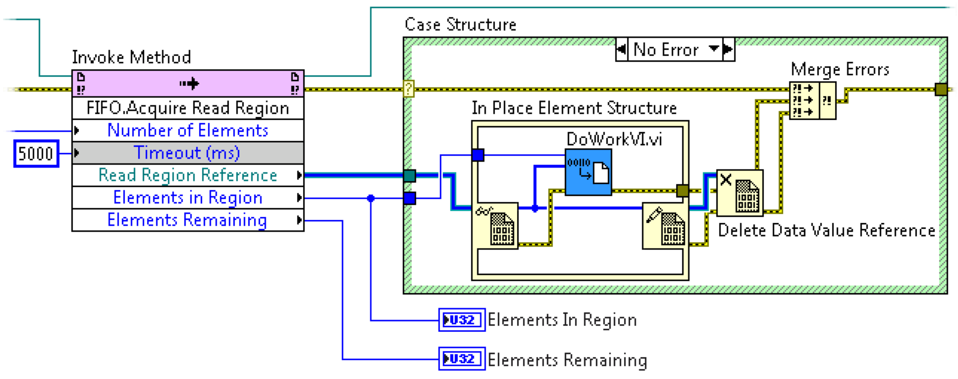


Figure 98. This block diagram uses the Invoke Method function configured with the Acquire Read Region method to acquire a region to read from the buffer. It then passes the reference to the read region through an In Place Element structure, where further processing occurs without making buffer copies.

INTERRUPTS

LabVIEW FPGA devices can synchronize with the host using hardware interrupts. Interrupts are useful in applications where the host waits for a signal from the FPGA, such as when you poll the status of an indicator to see if an event has occurred. This requires some host processing power to continually read from the indicator. Interrupts provide a polling-free method where the host does not need to query the device for its status, and, in turn, frees up CPU and bus resources.

Interrupts do have a higher cost per use than polling because of OS overhead. When an interrupt occurs, the OS often queries several devices to find the source of the interrupt, making interrupt latency dependent on your choice of bus, CPU, or OS.

Average interrupt latency is lowest on PXI systems with a high-performance embedded controller, where it typically is in the range of a few microseconds. CompactRIO controllers usually feature interrupt latency in the tens of microseconds, up to a hundred microseconds. Using LabVIEW Real-Time on the controller provides consistent interrupt latency, thanks to the deterministic OS response.

TRANSFERRING DATA BETWEEN DEVICES

PEER-TO-PEER STREAMING

On PXI, you can use peer-to-peer streaming to transfer data from an FPGA target directly to another FPGA target at high rates, bypassing the host processor. You also can use peer-to-peer streaming to send data between FPGA and non-FPGA devices, such as digitizers, arbitrary signal generators, and other modular instruments from NI.

Peer-to-peer streaming works by setting up a direct, unidirectional communication path between devices in a PXI chassis, leveraging PCI Express technology. The direct connection is useful for distributing high-performance LabVIEW FPGA applications across multiple NI FlexRIO devices in a chassis. If your application runs out of space on an NI FlexRIO device, you can simply add another board to the chassis and stream the data to it for further processing.

Because multiple modular instruments from NI support peer-to-peer streaming, you can use these high-precision instruments to acquire and stream data directly to NI FlexRIO devices for processing. For example, a high-throughput PXI Express-based digitizer can transfer data to an FPGA device to perform spectral analysis, which frees up the host processor to perform other tasks. PXI Express uses a tree topology. As a result, by carefully selecting slots for your devices, you can use peer-to-peer streaming to send data from the digitizer directly to the FPGA device without affecting other peer-to-peer transfers in the system.

Peer-to-peer streaming presents a FIFO interface, similar to other FIFOs in LabVIEW FPGA. The interface also includes the ability to determine how many samples or available slots are in the FIFO, as well as the state of the peer-to-peer stream. You can use this API to write data from one device and read it a few microseconds later from another device.

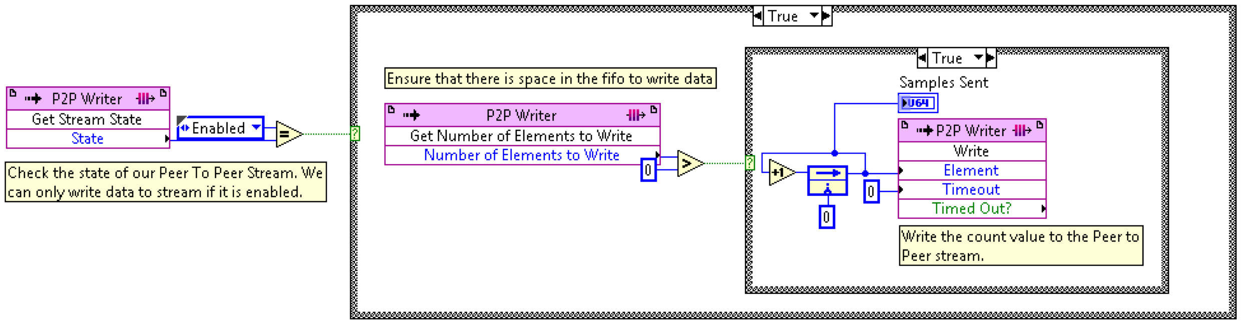

True
True
P2P Writer
Get Stream State
State
Enabled
Check the state of our Peer To Peer Stream. We can only write data to stream if it is enabled.
Ensure that there is space in the fifo to write data
P2P Writer
Get Number of Elements to Write
Number of Elements to Write
0
Samples Sent
U64
+1
0
0
P2P Writer
Write
Element
Timeout
Timed Out?
Write the count value to the Peer to Peer stream.

Figure 99. The peer-to-peer streaming API on the FPGA side presents a FIFO interface similar to the host interface and FPGA FIFOs including the ability to query the space or number of elements available on the peer's buffer, as well as stream state.

You configure peer-to-peer connections on the host system using an API that links peer-to-peer endpoints together and then enables the stream. Once configured, there is no further host involvement as the FPGAs send data from one to another.

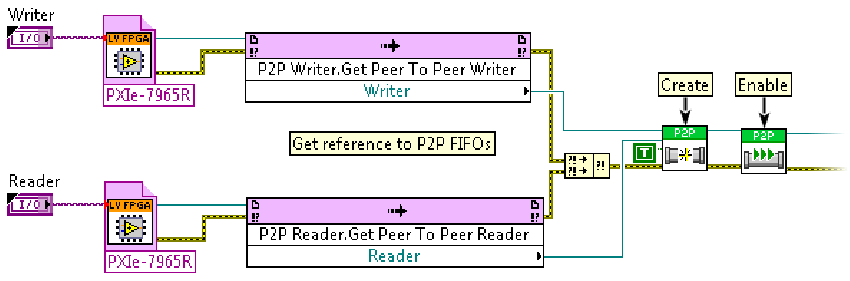

Writer
I/O
LV FPGA
PXIe-7965R
P2P Writer.Get Peer To Peer Writer
Writer
Get reference to P2P FIFOs
Create
Enable
P2P
P2P
Reader
I/O
LV FPGA
PXIe-7965R
P2P Reader.Get Peer To Peer Reader
Reader

Figure 100. Peer-to-peer streaming is configured and monitored through a host-side API.

Peer-to-peer throughput depends on the device, chassis, slot configuration, and controller you use. Consider the examples and factors discussed in the following section when selecting and configuring hardware for peer-to-peer streaming applications.

DEVICE CONSIDERATIONS FOR PEER-TO-PEER STREAMING

Peer-to-peer devices must be able to supply or accept data at a desired rate, which in turn puts a requirement on their bus interfaces and processing logic speeds. The slowest peer limits the overall achievable rate.

- NI PXIe-797xR for NI FlexRIO devices support PCI Express 2.0 communication, which offers the highest throughput for peer-to-peer communication. These devices make use of four PCI Express lanes for up to 1.5 GB/s of throughput for unidirectional transfers.
- NI PXIe-796x for NI FlexRIO devices use PCI Express 1.0 lanes and stream data at more than 800 MB/s into or out of the module. When streaming in both directions simultaneously, the FPGA modules can achieve rates of more than 700 MB/s per direction.

CHASSIS CONSIDERATIONS FOR PEER-TO-PEER STREAMING

The PCI Express backplane switches route data through the chassis and provide the high-bandwidth point-to-point connections that enable peer-to-peer data streaming. The bandwidth is dependent on the switch for modules in chassis slots that are directly connected to the same PCI Express switch.

The NI PXIe-1085 chassis provides a maximum of 4 GB/s through eight PCI Express 2.0 lanes available on any slot. Devices may not be able to saturate the bus; however, bandwidth is available if you decide to upgrade the devices in the future. Similarly, PXI Express 2.0 devices can work in PXI Express 1.0 chassis, but they are limited to PXI Express 1.0 bandwidth.

Chassis design also makes bandwidth dependent on the slots you select for your devices. When modules in a peer-to-peer streaming system are not connected to the same PCI Express switch, data must pass through other switches on the backplane or through the host controller switch, making throughput dependent on the capabilities of these other switches.

The NI PXIe-1085 chassis has two PCI Express switches connected by eight PCI Express 2.0 lanes for an interswitch bandwidth of 4 GB/s. This interswitch connection is shared by peer-to-peer connections when one peer is inserted into any slot between slots 2 and 9 and the other is in any slot between 11 and 18. Adding enough peers in this manner eventually saturates the interswitch connection, which affects throughput. As a result, it is recommended that you place peer-to-peer streaming pairs on the same switch when possible.

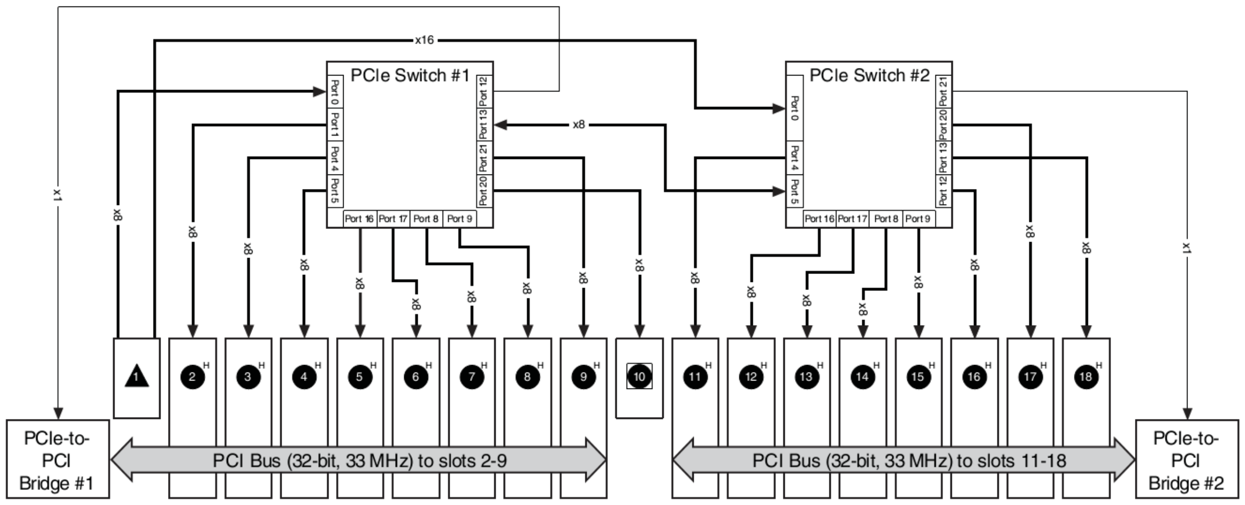


Figure 101. The NI PXIe-1085 chassis backplane has two PCI Express 2.0 switches interconnecting 18 chassis slots.

In the case of the NI PXIe-1065 chassis, there are two chassis segments. Devices in slots 9 through 14 communicate with each other directly through the PCI Express switch on the backplane. Devices in slots 7 and 8 must go through the host controller's onboard chipset switch, which makes bandwidth controller dependent.

Older controllers, such as an NI 8130, for example, limit the bandwidth to about 640 MB/s when the peer-to-peer stream crosses from one chassis segment to the other. Some controllers, such as an NI 8108, do not support peer-to-peer streaming, but streaming can still occur when devices are directly connected to a chassis switch.

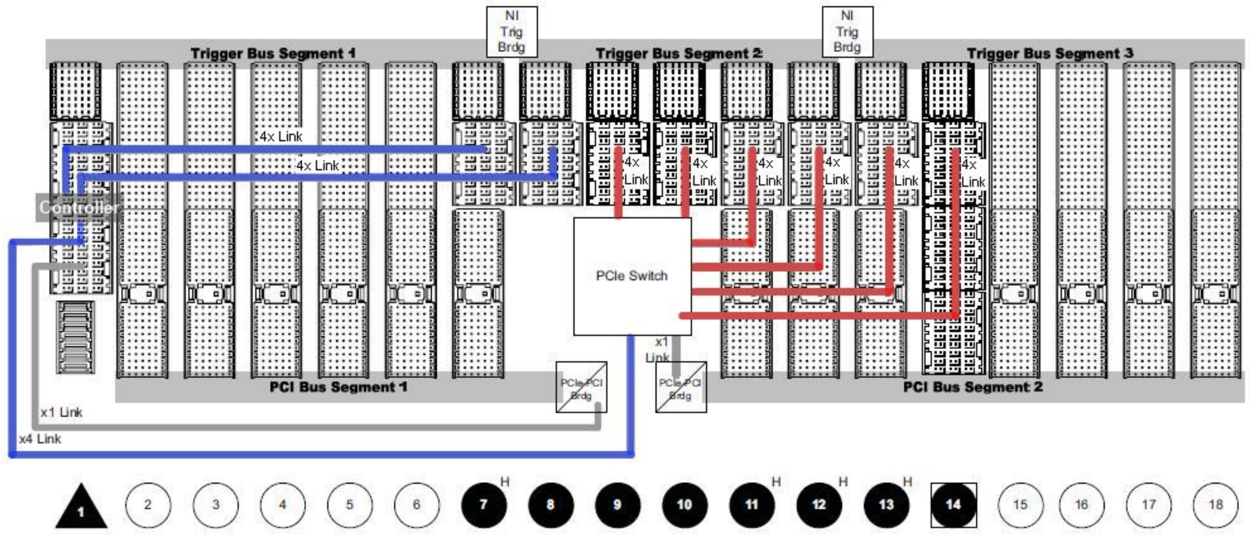


Figure 102. In the NI PXIe-1065 chassis backplane, devices connected in slots 7 and 8 must go through the host controller switch in slot 1 when performing peer-to-peer streaming, which makes throughput controller dependent.

When multiple peer-to-peer streams run simultaneously, bandwidth is evenly shared for streams in the same direction on a link. Simultaneous streams in opposite directions have a slight negative impact on the throughput of others due to acknowledge messages that travel back to the data source.

Finally, peer-to-peer latency is important when attempting to solve closed-loop control applications that require streaming data across multiple FPGAs. Peer-to-peer latency is typically 2 µs to 4 µs, but it can temporarily increase to 10 µs or more if the stream is competing with other traffic on the system. As a result, you should attempt to isolate peer-to-peer traffic to dedicated PCI Express switches if latency is important.

Additional Resources

[1]	DMA Best Practices http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_dma_best_practices
[2]	An Introduction to Peer-to-Peer Streaming http://www.ni.com/white-paper/10801/en
[3]	Improving Efficiency When Accessing DMA FIFOs (FPGA Module) http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgahelp/fpga_zerocopy_dma/

This page intentionally left blank.

NEXT STEPS

The LabVIEW product documentation provides detailed information on how to accomplish many of the tasks discussed in this guide. Hardware manuals also contain valuable information about the features and performance characteristics of NI RIO devices.

The main NI support page, ni.com/support, provides quick access to manuals, KnowledgeBase documents, tutorials, example code, community forums, technical support, and customer service.

FORMAL TRAINING

You can obtain high-throughput LabVIEW application instructor-led training from NI. The High-Throughput LabVIEW FPGA course covers many of the topics in this guide in more detail, with guided exercises to check for understanding.

This guide deals with the subject of high-performance LabVIEW FPGA applications and focuses on programming inside the single-cycle Timed Loop. Not every application, or every part of an application, requires SCTL programming. The standard LabVIEW FPGA course covers many of the basic concepts you need to program LabVIEW FPGA applications outside the SCTL. This course is available in an instructor-led format, but it is also included free of charge for Standard Service Program (SSP) subscribers as self-paced online training.

EVALUATING THE NI RIO PLATFORM

You can download an evaluation version of LabVIEW and the LabVIEW FPGA Module and create, simulate, and compile FPGA designs even if you do not have access to NI hardware. You also can obtain the LabVIEW RIO evaluation kit [4], a low-cost option for evaluating the NI RIO platform.

NI ALLIANCE PARTNERS AND SERVICES

NI partners can help you with the design, integration, and deployment of your application. The NI Alliance Partner Network is a program of more than 700 independent, third-party companies worldwide that provide engineers with complete solutions and high-quality products based on graphical system design.

Finally, you can count on NI field engineers to discuss how to solve your particular application using the NI platform.

Additional Resources

[1]	High-Throughput LabVIEW FPGA Training http://sine.ni.com/tacs/app/overview/p/ap/of/lang/en/ol/en/oc/us/pg/1/sn/n24:16770/id/2158/
[2]	LabVIEW FPGA Training 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/
[3]	Self-Paced Online Training http://www.ni.com/white-paper/14457/en/
[4]	NI LabVIEW RIO Architecture Evaluation Options http://www.ni.com/rioeval/
[5]	NI Alliance Partner Network http://www.ni.com/alliance/

REVISIONS AND FEEDBACK

NI strives to provide high-quality content and welcomes your comments. Feel free to send feedback for future revisions of the guide to hprioguide@ni.com .

If there are areas you believe need clarification, submit your question to the LabVIEW FPGA Discussion Forum, where the applications, support, and R&D engineers can answer it for the benefit of all readers. Begin your message with the word "fpga", post to "LabVIEW", and add the tag "hprioguide" to your message for faster routing.

Revision	Date	Change Summary
1.0	02/14/2014	Initial release
1.1	02/25/2014	Integrating High-Throughput IP chapter: ▪ Truncate rounding biases toward negative infinity. ▪ Overflow and rounding tweaks not needed unless overriding the output type.
		Resource Optimization chapter: ▪ Overflow only coerces, never rounds ▪ Overflow and rounding tweaks not needed unless overriding the output type. ▪ Round-Half-Up biased toward greater values and does not require comparison logic. ▪ Discrete Delay function now supports different data types. ▪ Discrete Delay function supports dynamic delays.
		Minor grammar corrections, clarifications, and figure fixes.