

Chapter 2

LabVIEW™ FPGA

2.1 Field-Programmable Gate Array (FPGA)

An field-programmable gate array (FPGA) [1] is a device that contains a matrix of reconfigurable gate array logic circuitry. When an FPGA is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. Unlike processors, FPGAs use dedicated hardware for processing logic and do not have an operating system. FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. As a result, the performance of one part of the application is not affected when additional processing is added. Also, multiple control loops can run on a single FPGA device at different rates. FPGA-based control systems can enforce critical interlock logic and can be designed to prevent I/O forcing by an operator. However, unlike hard-wired printed circuit board (PCB) designs which have fixed hardware resources, FPGA-based systems can literally rewire their internal circuitry to allow reconfiguration after the control system is deployed to the field. FPGA devices deliver the performance and reliability of dedicated hardware circuitry.

A single FPGA can replace thousands of discrete components by incorporating millions of logic gates in a single integrated circuit (IC) chip. The internal resources of an FPGA chip consist of a matrix of configurable logic blocks (CLBs) surrounded by a periphery of I/O blocks. Signals are routed within the FPGA matrix by programmable interconnect switches and wire routes (see Fig. 2.1).

FPGA technology provides the reliability of dedicated hardware circuitry, true parallel execution, and lightning fast closed-loop control performance. This application note provides answers to frequently asked questions (FAQs) regarding the use of reconfigurable FPGA-based hardware targets for closed-loop control applications. A compactRIO is shown in Fig. 2.2.

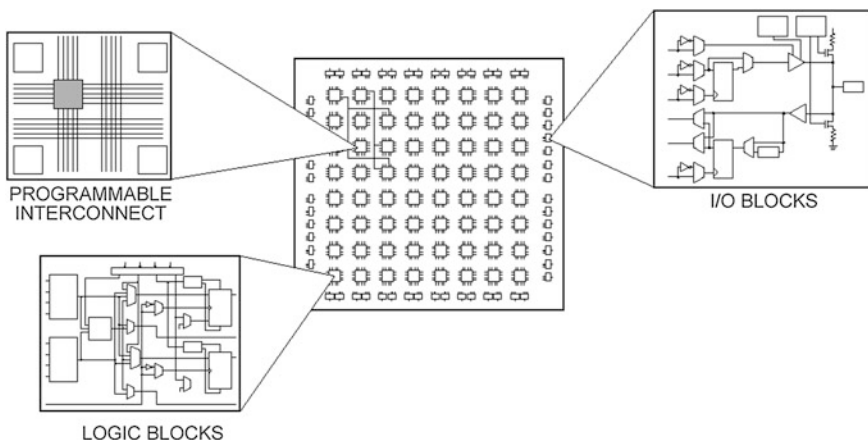


Fig. 2.1 Looking inside an FPGA chip

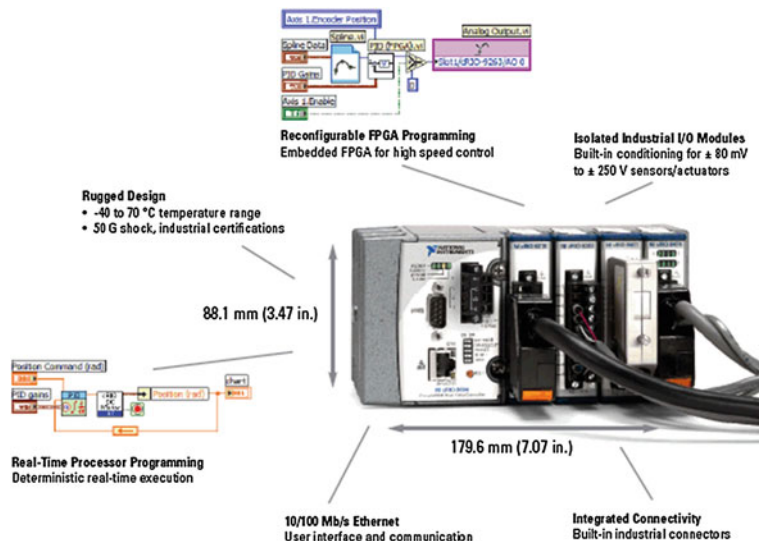
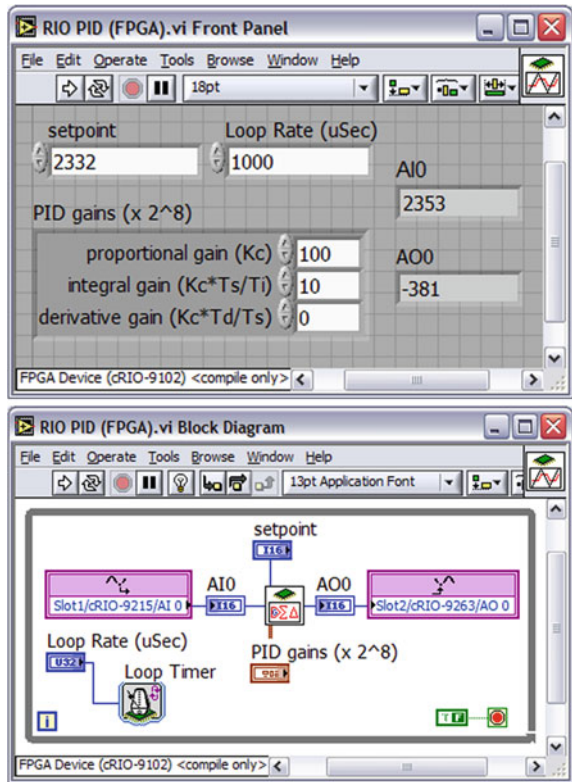


Fig. 2.2 NI CompactRIO is a small, rugged FPGA-based control system

2.1.1 How Do FPGA-Based Control Systems Compare to Processor-Based Systems?

Like processor-based control systems, FPGAs have been used to implement all types of industrial control systems (see Fig. 2.3), including analog process control, discrete logic, and batch or state machine-based control systems. However, FPGA-based control systems differ from processor-based systems in significant ways.

Fig. 2.3 Performing PID control in LabVIEW FPGA



When you compile [2] your control application for an FPGA device, the result is a highly optimized silicon implementation that provides true parallel processing with the performance and reliability benefits of dedicated hardware circuitry. Since there is no operating system on the FPGA chip, the code is implemented in a way that ensures maximum performance and reliability.

In addition to offering high reliability, FPGA devices can perform deterministic closed-loop control at extremely fast loop rates. In most FPGA-based control applications, speed is limited by the sensors, actuators, and I/O modules rather than the processing performance of the FPGA. For example, the proportional-integral-derivative (PID) control algorithm that is included with the LabVIEW FPGA Module executes in just 300 ns (0.000000300 s). PID control is commonly used for regulating analog process values such as pressure, temperature, force, displacement, fluid flow, or electrical current.

FPGA-based control systems offer deterministic closed control performance at rates exceeding 1 MHz. In fact, many algorithms can be executed in a single cycle of the FPGA clock (40 MHz). Processing is done in parallel, so multi-rate control

Performance limited to 1 kHz	→	Closed loop performance beyond 1 MHz
Serial execution, single rate control	→	Parallel execution, multi-rate control
Performance slows as app. grows	→	No slow down as application grows
Operating system runs control logic	→	Control logic in dedicated hardware
I/O modules have fixed functionality	→	I/O functionality is reconfigurable
Custom circuitry requires board layout	→	Software defined gate array
Separate motion control system	→	Motion integrated with other control logic

Fig. 2.4 Processor-based control (*left*) compared to FPGA-based control (*right*)

systems are easy to implement. Since control logic runs in dedicated hardware subsystems on the FPGA, applications do not slow down when additional processing is added. In many cases, a software-defined gate array in FPGA hardware can be used to replace a costly and time-consuming custom PCB layout.

FPGAs can digitally process signals at very high speeds and are often used to reconfigure the I/O module functionality. For example, a digital input module can be used to simply read the true/false state of each digital line. Alternately, the same FPGA can be reconfigured to perform processing on the digital signals and measure pulse width, perform digital filtering, or even measure position and velocity from a quadrature encoder sensor.

FPGA-based systems often incorporate motion control and motor drive commutation in a single FPGA-based control application. By contrast, processor-based systems typically offload the motor drive commutation to separate hardware since motor current or torque control requires fast loop rates (commonly 20 kHz) and precise timing of the gate drive commutation signals. A comparison between processors controller and LabVIEW FPGA is presented in Fig. 2.4.

2.1.2 How Do I Program My Control Application Using the LabVIEW FPGA Module?

The LabVIEW FPGA Module enables you to use high-level graphical dataflow programming to create a highly optimized gate array implementation of your analog or digital control logic. You can use normal LabVIEW programming techniques to develop your FPGA application. When you target FPGA hardware such as a CompactRIO chassis or R Series intelligent data acquisition (DAQ) device, the LabVIEW programming palette is simplified to contain only the functions that are designed to work on FPGAs [3]. The primary programming difference compared to traditional LabVIEW is that FPGA devices use integer math rather than floating-point math. Also, there is no notion of multithreading or priorities since each loop executes in independent dedicated hardware and does not have share resources—in effect, each loop executes in parallel at “time critical” priority.

The LabVIEW FPGA palette contains extensive intellectual property (IP) libraries [4]. Table 2.1 shows a list of some of the key function blocks for developing FPGA-based control systems. For more details, see the LabVIEW FPGA Module user manual in NI website.

Table 2.1 Lists of some of the key function blocks for developing FPGA-based control systems

Category	Key functions for control	Common control applications
Programming structures	For Loop, While Loop, Case Structure, Feedback Node, Sequence Structure, Single Cycle Timed Loop, Shift Register, HDL Interface Node	Analog process control loops, state machines, batch control, sequential function charts, event response, repeated execution, signal latching, subroutines, sequencing, system state control (power up, shut down, watchdog, fault, ...)
Input/Output	Analog Input, Analog Output, Digital Input, Digital Output, Digital Port Input, Digital Port Output	Interfacing to digital I/O, voltage, current, temperature, load, pressure, strain, relay, 4–20 mA, H-bridge, CAN communication, wireless networking, and other signals
Analog control	Discrete PID, Discrete Control Filter, Discrete Delay, Discrete Normalized Integrator, Initial Condition, Unit Delay, Zero-Order Hold, Backlash, Dead Zone, Friction, Memory Element, Quantizer, Rate Limiter, Relay, Saturation, Switch, Trigger, Linear Interpolation, Sine Generator, Look-Up Table 1D	Analog control algorithms, filtering of noisy signals, limiting input/output signals, scaling nonlinear sensor signals to engineering unit proportional values, function generation, sine, cosine, log, exponential, gain scheduling, ramp/soak
Discrete logic	And, And Array Elements, Boolean Array To Number, Boolean To (0,1), Compound Arithmetic, Exclusive Or, Implies, Not, Not And, Not Exclusive Or, Not Or, Number To Boolean Array, Or, Or Array Elements, Boolean Crossing	Digital control, digital logic, Boolean logic, relay ladder logic, sequence of events, state transitions, control of 2-state and 3-state discrete devices, edge detection
Comparison functions	Equal?, Equal To 0?, Greater?, Greater Or Equal?, Greater Or Equal To 0?, Greater Than 0?, Less?, Less Or Equal?, Less Or Equal To 0?, Less Than 0?, Not Equal?, Not Equal To 0?, Select, Max and Min, In Range and Coerce, Zero Crossing	Alarming, triggering, event detection, peak detection, signal comparison, thresholding, change of state detection, signal selection (high, min, max), limit testing, selector/multiplexer, heating/cooling split range control

(continued)

Table 2.1 (continued)

Category	Key functions for control	Common control applications
Math	Absolute Value, Add, Compound Arithmetic, Decrement, Increment, Multiply, Negate, Quotient and Remainder, Scale By Power Of 2, Sign, Subtract, Saturation Add, Saturation Multiply, Saturation Subtract, Join Numbers, Logical Shift, Rotate, Rotate Left With Carry, Rotate Right With Carry, Split Number, Swap Bytes, Swap Words	Analog signal manipulation, summing, counter/timers, rate of change detection, electronic gearing/camming, accumulator, averaging, totalizer, digital signal processing
Data transfer, timing, triggering and synchronization	Global Variable, Local Variable, FIFO Read, FIFO Write, Memory Read, Memory Write, Interrupt, Loop Timer, Tick Count, Wait, Generate Occurrence, Set Occurrence, Wait On Occurrence, First Call?	Watchdogs, timers, accumulators, pulse width measurement/generation, timer on/off delay
NI SoftMotion module	Motion Control Loop PID (32-bit), Spline Engine (Interpolation)	Multiaxis coordinated motion control, trajectory generation, straight line moves, jogging, arc move, contouring, interpolation
Digital filter design toolkit	Filter Design, Fixed-Point Tools, Code Generation	Digital filter design, convert floating-point to fixed-point, generate LabVIEW FPGA code

2.1.3 How Does the LabVIEW Compiler Translate My Graphical Code into FPGA Circuitry?

The LabVIEW FPGA module compiles your LabVIEW application to FPGA hardware using an automatic multistep process [2]. Behind the scenes, your graphical code is translated to text-based VHDL code. Then industry standard Xilinx ISE compiler tools are invoked and the VHDL code is optimized, reduced, and synthesized into a hardware circuit realization of your LabVIEW design. This process also applies timing constraints to the design and tries to achieve an efficient use of FPGA resources (sometimes called “fabric”).

A great deal of optimization is performed during the FPGA compilation process to reduce digital logic and create an optimal implementation of the LabVIEW application (see Fig. 2.5). Then the design is synthesized into a highly optimized silicon implementation that provides true parallel processing capabilities with the performance and reliability of dedicated hardware [5].

The end result is a bit stream file that contains the gate array configuration information. When you run the application, the bit stream is loaded into the FPGA

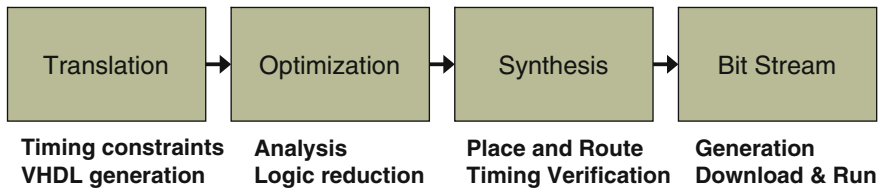


Fig. 2.5 LabVIEW FPGA compilation process

chip and used to reconfigure the gate array logic. The bit stream can also be loaded into nonvolatile flash memory and loaded instantaneously when power is applied to the target. There is no operating system on the FPGA chip; however, execution can be started and stopped using enable-chain logic that is built into the FPGA application.

2.1.4 *FPGAs Are Fast, but How Do Faster Loop Rates Improve Control System Performance?*

In general, the speed of the control system impacts its performance, stability, robustness, and disturbance rejection characteristics (see Fig. 2.6). Faster control systems are typically more stable, easier to tune, and less susceptible to changing conditions and disturbances.

To provide stable and robust control, a control system must be able to measure the process variable and set an actuator output command within a fixed period of time. Systems (plants) that can change quickly require fast control systems to guarantee reliable performance within acceptable limits. As a rule, the control loop rate should be at least ten times faster than the time constant of the system (plant). The time constant is a measure of the speed of the system.

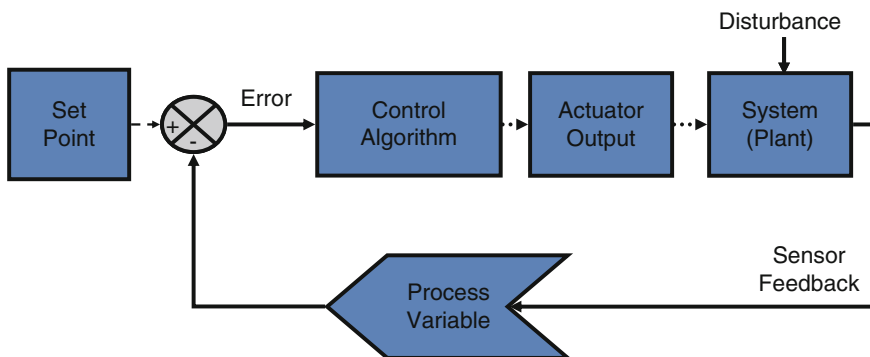


Fig. 2.6 Typical closed-loop control system

For example, the current in a DC motor may change as fast as 1 A per millisecond in response to a 24 V output from an H-bridge driver. To precisely control the motor current, the control system must sample the current quickly and make frequent adjustments to the actuator output.

2.1.5 *What FPGA Hardware Targets Are Available from NI?*

The CompactRIO reconfigurable embedded system (see Fig. 2.7) is a small modular system for industrial applications that require the highest level of ruggedness and reliability. CompactRIO is designed for harsh environments and offers a wide temperature range, high shock and vibration ratings, and an array of industrial certifications and ratings. CompactRIO is rated for marine environments, Class I, Division 2 rating for hazardous locations, and offers up to 2300 V of isolation. Like all FPGA targets from NI, CompactRIO uses the C Series industrial I/O modules for low-cost connectivity directly to industrial control sensors and actuators. In addition, there are many third-party vendors around the world that offer C Series I/O and communication modules.

The NI R Series intelligent DAQ devices are plug-in boards for PCI and PXI/CompactPCI buses with onboard FPGA hardware for user-defined signal processing and control. Up to 8 analog inputs, 8 analog outputs and 160 digital I/O channels are built into the intelligent DAQ devices. You can also connect an expansion chassis to any digital port and add C series industrial I/O modules. The NI intelligent DAQ devices enable you to define your own hardware functionality and offer limitless possibilities for timing, triggering, synchronization, digital signal processing, and control.

The PXI R Series intelligent DAQ system offers FPGA performance and reliability in the industry standard PXI form factor (see Fig. 2.8). In addition to the intelligent DAQ devices from NI, hundreds of non-reconfigurable plug-in boards are available from NI and other vendors around the world. The PXI system can be booted into Windows or the LabVIEW Real-Time operating system. C Series I/O modules provide signal conditioning and combine instrumentation grade accuracy

Fig. 2.7 NI CompactRIO reconfigurable embedded system





Fig. 2.8 PXI R series intelligent DAQ system



Fig. 2.9 PCI R series intelligent DAQ system

with industrial features such as isolation or high current drive capability. The R Series Expansion Chassis is used to connect C Series modules to intelligent DAQ devices. For more information, see the online application notes explaining the R Series Intelligent DAQ devices.

The PCI R Series Intelligent DAQ System enables you to add FPGA-based control capabilities to any desktop, industrial PC, or single-board computer (SBC) containing a PCI slot. Like all NI FPGA targets, the intelligent DAQ devices can load their bit stream instantly at power up from nonvolatile flash storage located on the plug-in board (see Fig. 2.9).

The National Instruments Compact Vision System (see Fig. 2.10) is a rugged standalone platform for industrial machine vision and I/O applications such as robotics, automated test, and automated inspection. All Compact Vision Systems

Fig. 2.10 NI compact vision system



contain a user-programmable FPGA for implementing custom triggers, counters, pulse width modulation (PWM), motion, and other digital control operations. NI Compact Vision systems use IEEE 1394 (FireWire) technology for interfacing to more than 300 compatible cameras.

2.1.6 What Closed-Loop Control Performance Can I Achieve?

In most cases, the computational performance of the FPGA is so fast that the control loop rate is limited only by the sensors, actuators, and I/O modules (see Fig. 2.11). This is a stark contrast to traditional control systems, where the processing performance was typically the limiting factor.

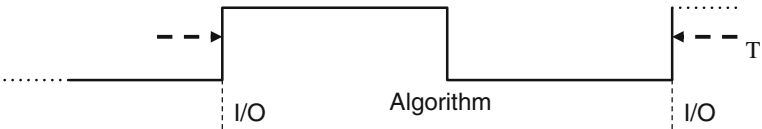


Fig. 2.11 The loop cycle time (T) is the time taken to execute one cycle of a control loop

For example, using R Series intelligent DAQ devices, the input/output and control logic calculations for discrete control applications can all be implemented at a 20 MHz control loop rate using the 5 V TTL digital I/O lines on the boards. These digital lines can be accessed from within a LabVIEW single-cycle timed loop (SCTL) executing at a 25 ns rate. Significant amounts of control logic can usually be included in a SCTL.

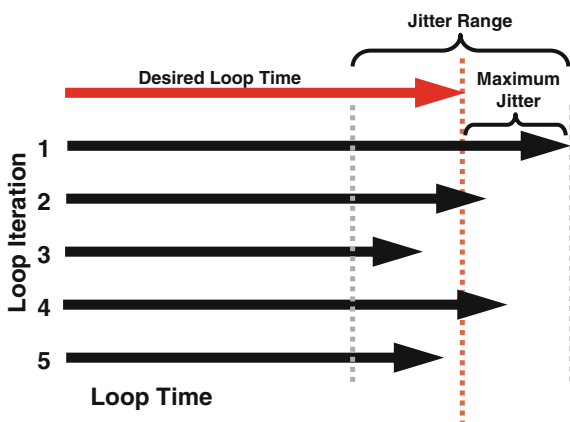
For 24 V discrete logic control applications using high current C Series digital I/O modules, the loop rate is limited to the update rate of the modules. For example, the NI 9423 digital input and NI 9474 digital output modules both have 1 μ s update rates, resulting in a maximum 24 V discrete control performance of 500 kHz.

In analog process control applications, the control loop rate is also limited by the update rate of the I/O modules. The NI 9215 analog input and NI 9263 analog output modules offer 16-bit resolution and simultaneous sampling capabilities at 10 μ s update rates. This results in a closed-loop analog process control performance of 50 kHz.

2.1.7 How Much Jitter Can I Expect in My FPGA-Based Control Loops?

A common gauge of control system performance and robustness is jitter (see Fig. 2.12), which is a measure of the variation of the actual loop cycle time from the desired loop cycle time. In general, purpose operating systems such as Windows, the jitter is unbounded so closed-loop control system stability cannot be guaranteed. Processor-based control systems with real-time operating systems are commonly able to guarantee control loop jitter of less than 100 μ s. In FPGA-based applications, the control loop does not need to share hardware resources with other tasks and control loops can be precisely timed using the FPGA clock. The jitter for FPGA-based control loops depends on the accuracy of the FPGA clock source. In

Fig. 2.12 To guarantee stability, control loop jitter must be bounded



the case of the CompactRIO cRIO-910x reconfigurable chassis, the FPGA clock jitter is only 250 ps (0.000000000250 s) when using a 40 MHz FPGA clock rate.

2.1.8 Creating a New LabVIEW Real-Time Project and Adding I/O

Now it is presented how the real-time system is included in a project [6]. This step allows to configure inputs and outputs (I/O) in real time.

1. Launch **NI LabVIEW** by clicking on the desktop icon. Then click on the **Real-Time Project** (see Fig. 2.13) link to start a new LabVIEW project for your NI CompactRIO system.

LabVIEW 8.20 has a Real-Time Project Wizard that makes creating and configuring real-time applications easy. To help you get started, the wizard enables you to choose an appropriate programming architecture and automatically generates a software template application.

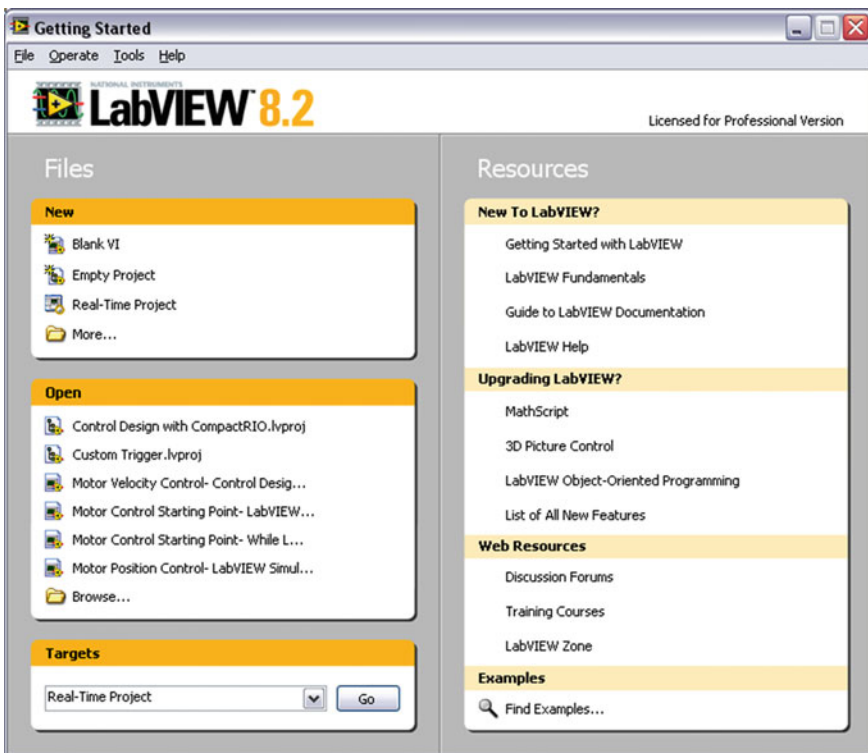


Fig. 2.13 Real-time project

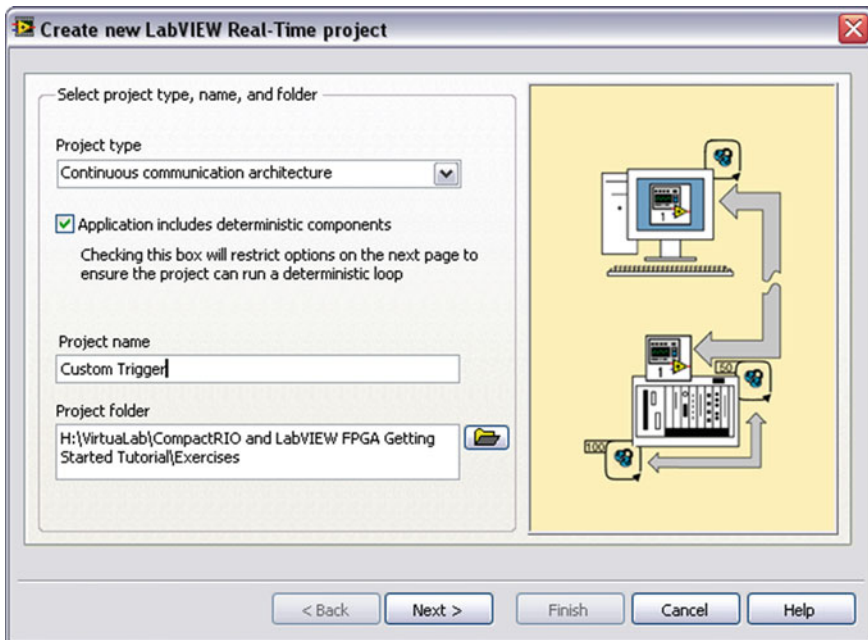



Fig. 2.14 Real-time folder

2. To select the working folder for your project, click the folder , navigate to **H:\VirtualLab\CompactRIO and LabVIEW FPGA Getting Started Tutorial\Exercises**, and then click the **Current Folder** button. Name your project **Custom Trigger**. Keep all of the project defaults as shown below and click the **Next** button (see Fig. 2.14).
3. Change your **Target Configuration** to **Two loops**. Under the **Host Configuration** section (see Fig. 2.15), check the **Include user interface** box. Then click **Next**.

The LabVIEW 8.20 Real-Time Project Wizard makes it easy to create a complete CompactRIO embedded system that includes an FPGA application (see Fig. 2.16), real-time processor application, and networked Windows host computer application. After this exercise is complete, you could use the template applications created by the wizard to create a complete networked system, including a deterministic loop running on the real-time controller to communicate with the FPGA and a lower priority loop to performed network communication, file logging, or additional analysis.

4. Click the **Browse** button to find the networked target you configured in MAX. Expand **Real-Time CompactRIO** folder and wait until your CompactRIO system is detected (see Fig. 2.17). Highlight your CompactRIO system and click **OK**. Then click **Next** to continue creating the real-time project.

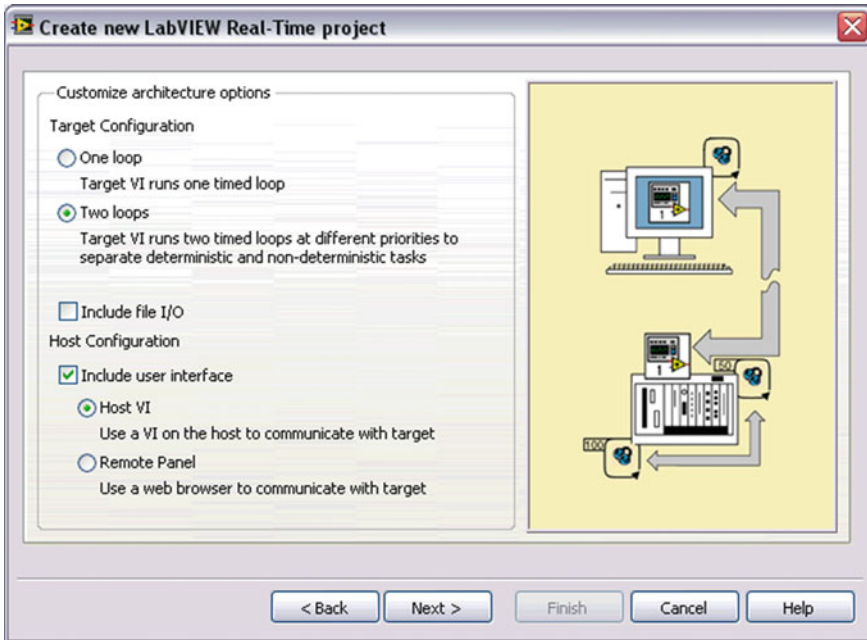


Fig. 2.15 Real-time target configuration

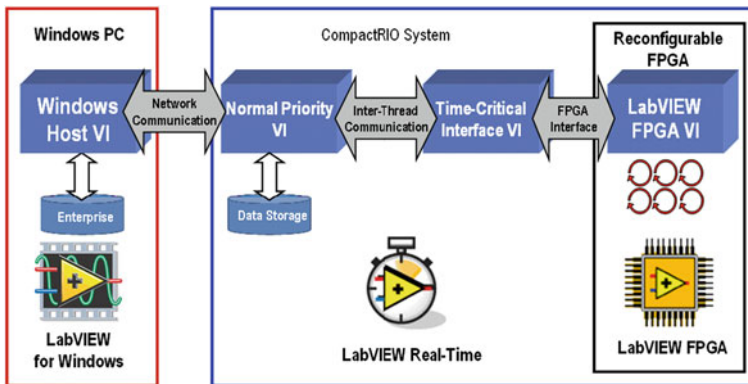


Fig. 2.16 Real-time application

5. Notice that the project wizard displays a preview of the project you configured. Click **Finish** to finalize the creation of the new real-time project and generate the application template code (see Fig. 2.18).

*When code generation is complete, two pre-built template applications will automatically open. The Windows host application (**host-network-RT (separate).vi**) includes a chart to plot the data sent by the CompactRIO system over*

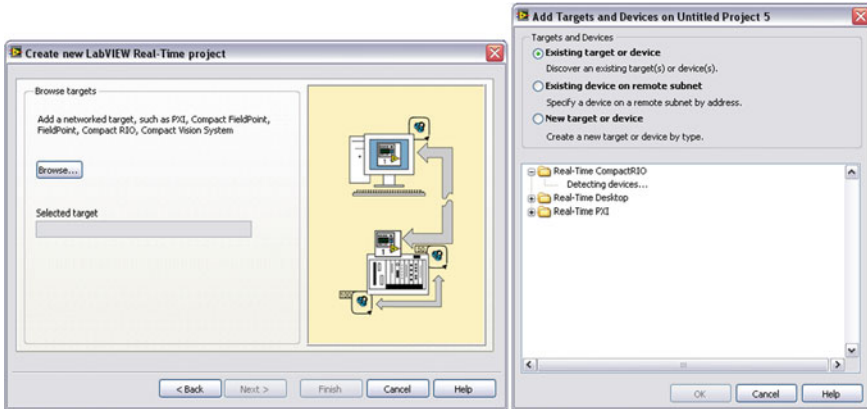


Fig. 2.17 CompactRIO detection

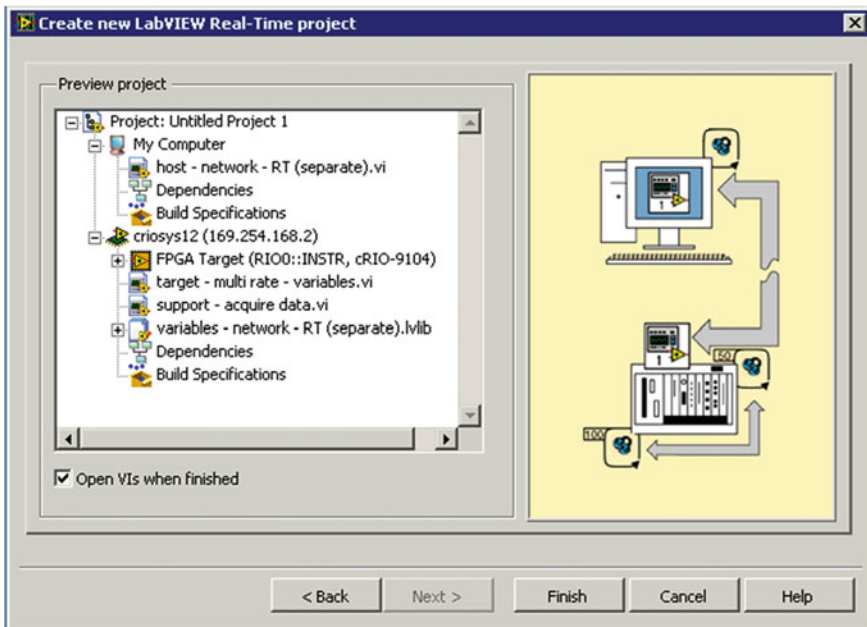


Fig. 2.18 Creating a new real-time project

the network, and a **stop-network** shared variable that is used to halt execution of the real-time embedded application running on the CompactRIO system (see Fig. 2.19).

6. In the **target-multi-rate-variables.vi** (see Fig. 2.20) real-time processor application, navigate to **Window>>Show Block Diagram**.

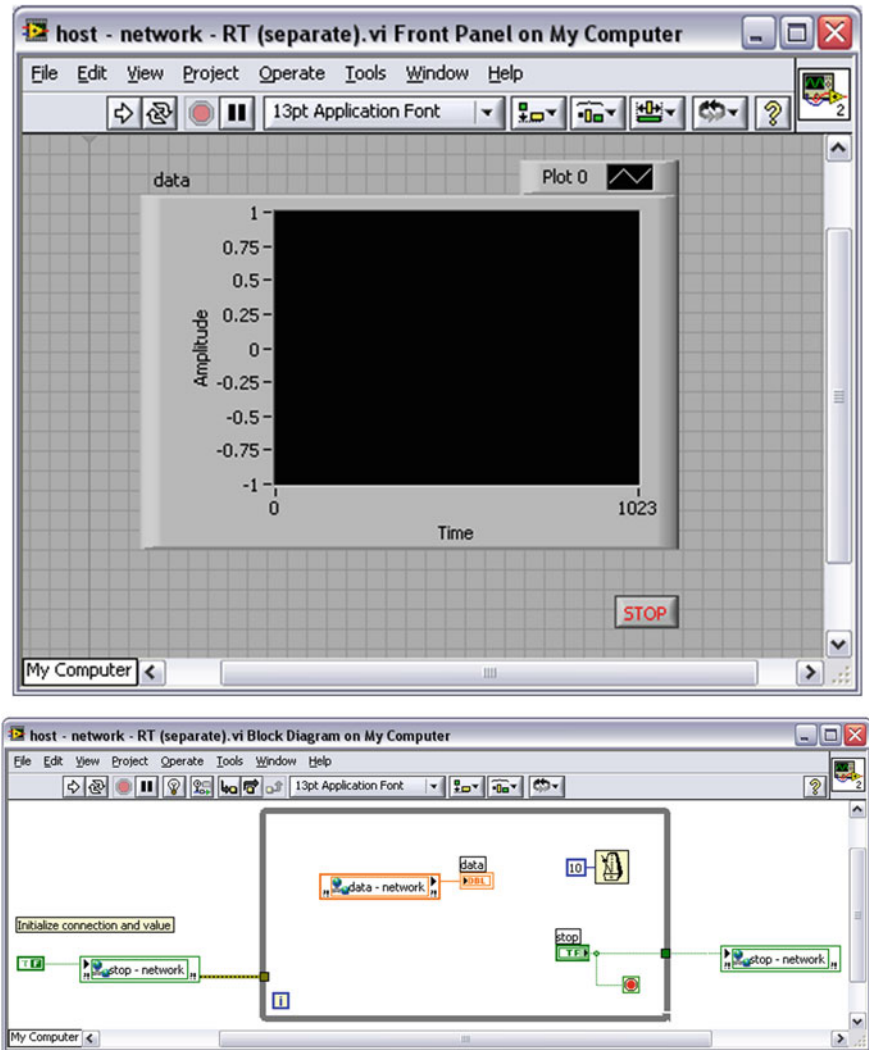


Fig. 2.19 Host–network–Real Time.vi

This embedded processor application produces a simulated I/O signal and sends the data to the Windows host computer using network-published shared variables. You would place any time critical routines, such as code to interface with your FPGA application within the top deterministic loop. Any lower priority non-deterministic tasks such as data logging or additional analysis would be placed in the bottom lower priority loop.

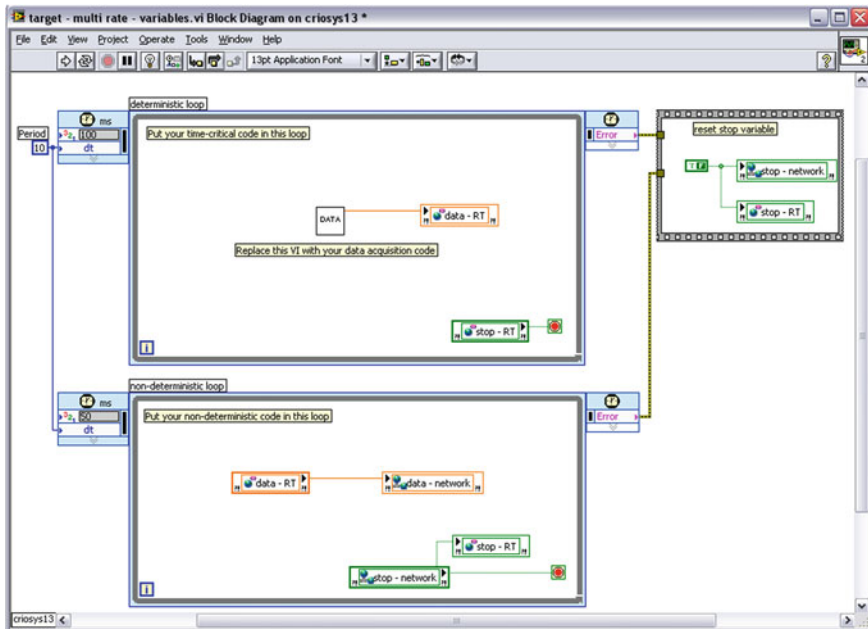


Fig. 2.20 Target-multi-rate-variables

7. Click the **Run** button on the real-time processor application (**target-multi-rate-variables.vi**). While the embedded application is being deployed, click the box next to **Close on successful completion** if it is not already checked (see Fig. 2.21).
8. Click the **Run** button on the Windows host application (**host-network-RT (separate).vi**). View the sinusoidal waveform displayed on host application chart (see Fig. 2.22). Click the **STOP** button on the host application and notice that the application stops running on both the host and real-time target.
9. In the **Project Explorer** window (see Fig. 2.23), right-click on the **FPGA Target** and select **New>>C Series Modules** to add your I/O modules to the project.
10. To automatically detect the I/O modules installed in your chassis (see Fig. 2.24), expand the **C Series Module** tab by clicking on the + symbol. Click **Continue** when the warning dialog window appears. A pre-built FPGA bit-stream will be downloaded to auto-detect the installed modules.

*Note: If you are working offline without a network connection to your CompactRIO system, you can still develop your code by selecting **New target or device** and manually adding the I/O modules.*

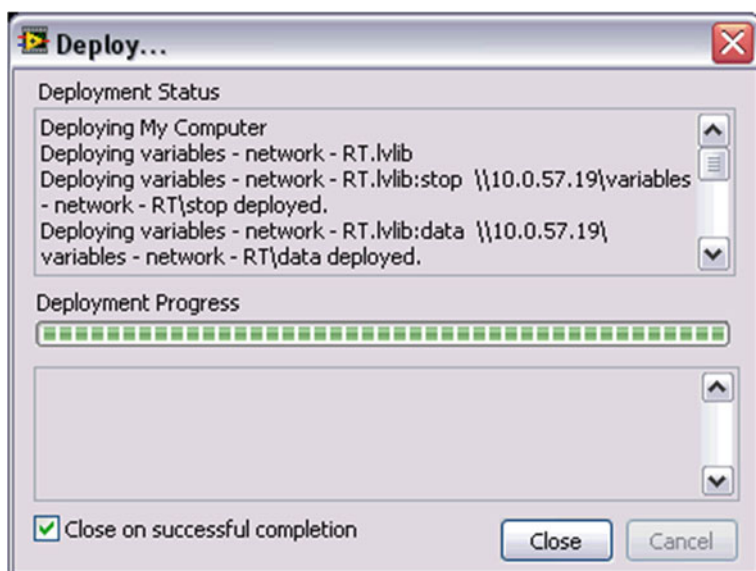


Fig. 2.21 Real-time program deployment

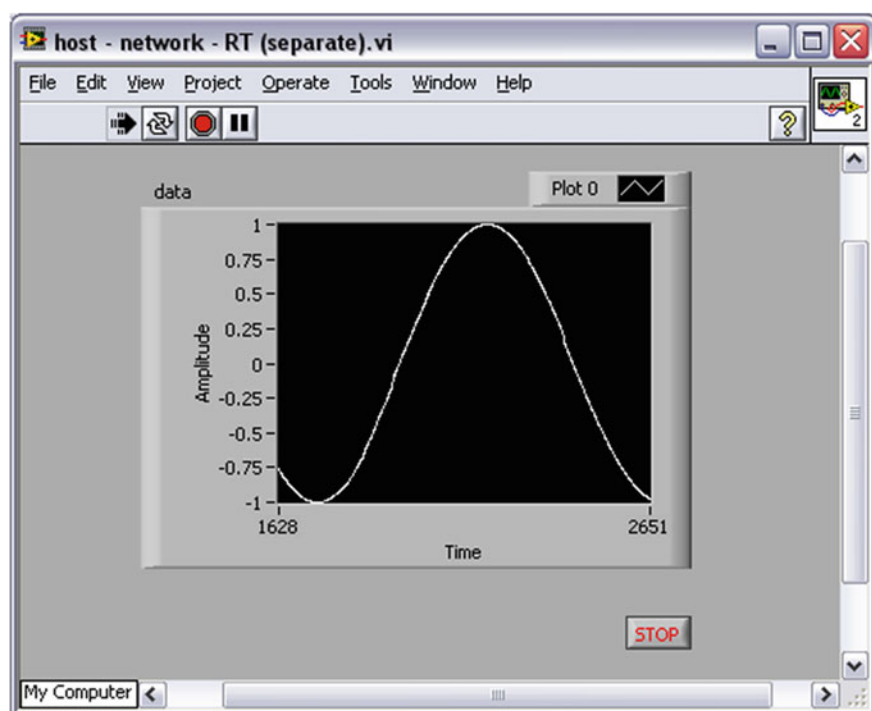


Fig. 2.22 Sine waveform displayed

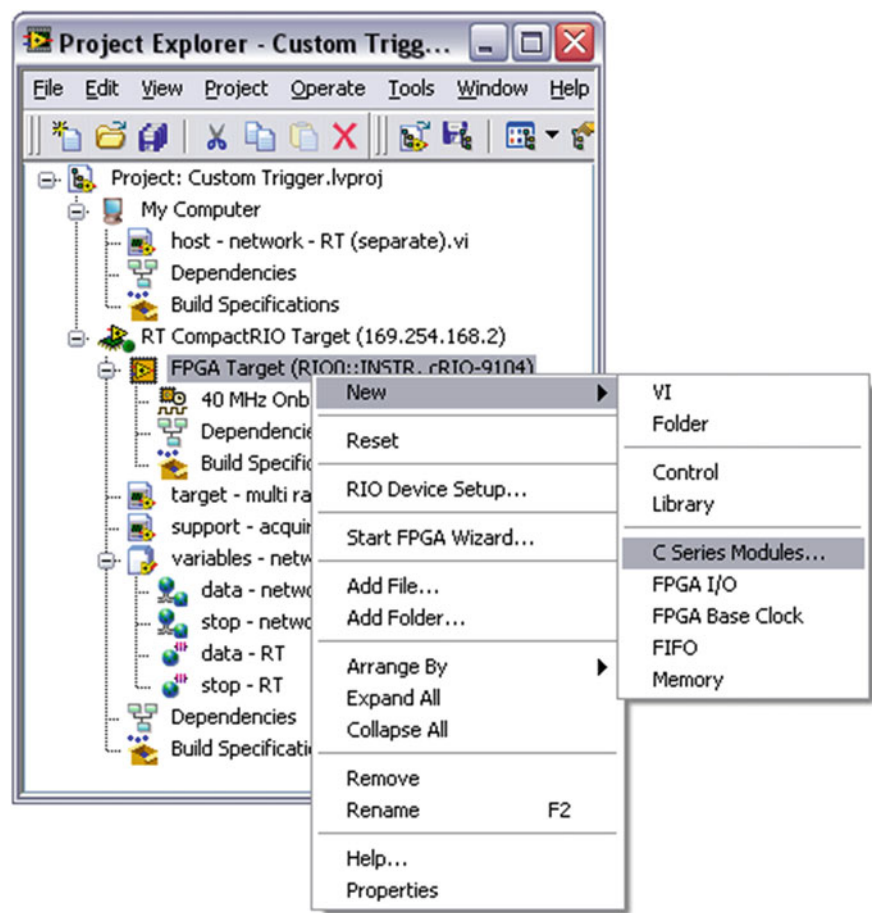


Fig. 2.23 Project explorer

11. After the modules are detected, select the modules that will be used in this exercise (see Fig. 2.25). To do this, first click on the **NI 9215** module, then hold down the **Ctrl** key and click on the **NI 9263** and **NI 9401** modules. Click **OK** to add all modules to your project.
12. In the **Project Explorer** window (see Fig. 2.26), right-click on the **FPGA Target** and select **New>>FPGA I/O** to add your I/O channels to the project. The **Analog Input** section is highlighted. To highlight all sections, hold down the **Shift** key and click on the **Digital Port Input and Output** section.
13. Next click the **Add** button to add all of the I/O channels. Then click the **OK** button to finish adding the channels to your project (see Fig. 2.27).

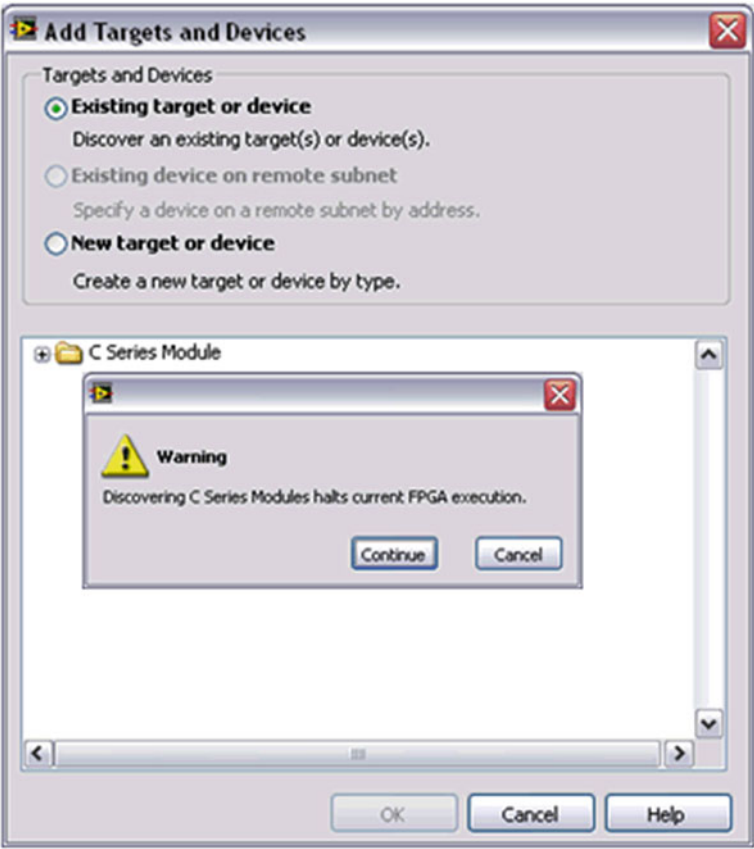


Fig. 2.24 Adding targets and devices

14. In the **Project Explorer** (see Fig. 2.28) window, right-click on the **FPGA Target** and select **Collapse All**. If you click the + symbol next to the **FPGA Target**, your LabVIEW Project should appear similar to what is shown below. Click the **Save All** button to save the project and all subVIs.

2.2 Developing the LabVIEW FPGA Application

One of the most important steps to build a fuzzy controller is to design an FPGA application, so the next section describes how an application is designed. When you finish a FPGA application, the front panel and block diagram of your completed FPGA application will look like Fig. 2.29.

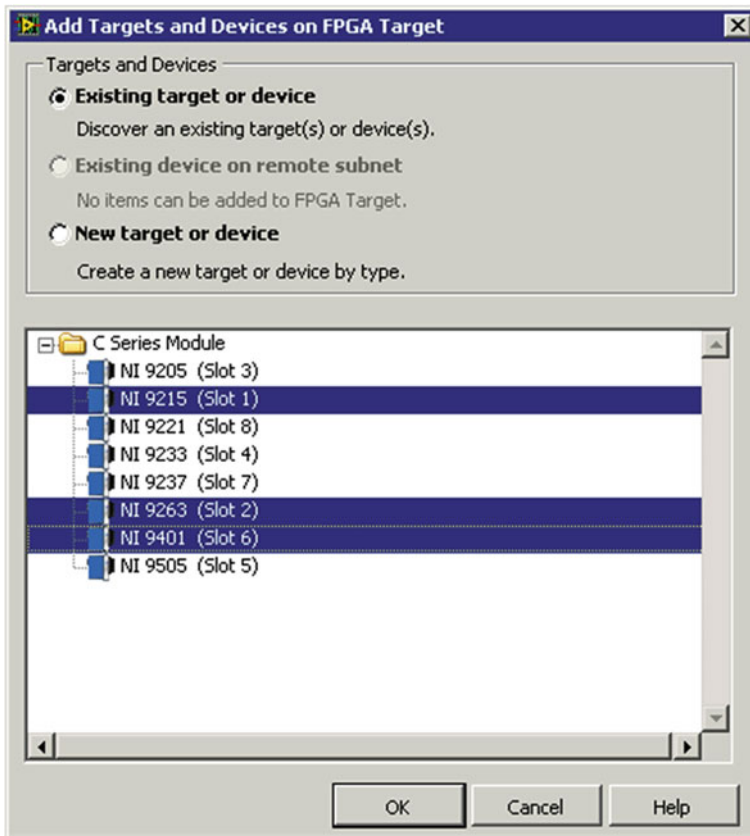


Fig. 2.25 Modules detected

The next step shows how an application is developed. *LabVIEW applications are called “Virtual Instruments” or “VIs.”*

1. In the LabVIEW **Project Explorer**, right-click on the **FPGA Target** and select **New>>VI** to start a new LabVIEW FPGA application. When the VI opens, navigate to **File>>Save**. Then browse to the **H:\VirtualLab\CompactRIO and LabVIEW FPGA Getting Started Tutorial\Exercises** folder and save the application as “**Simple AIAO (FPGA)**” (see Fig. 2.30).

To make it easier to distinguish the intended execution target, it is recommended that you include the words “FPGA” in the filename of your FPGA applications.

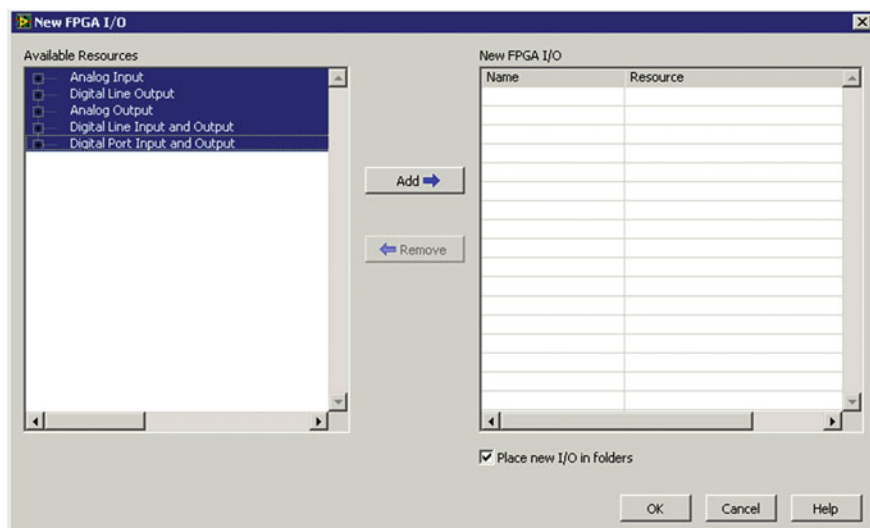


Fig. 2.26 FPGA I/O modules

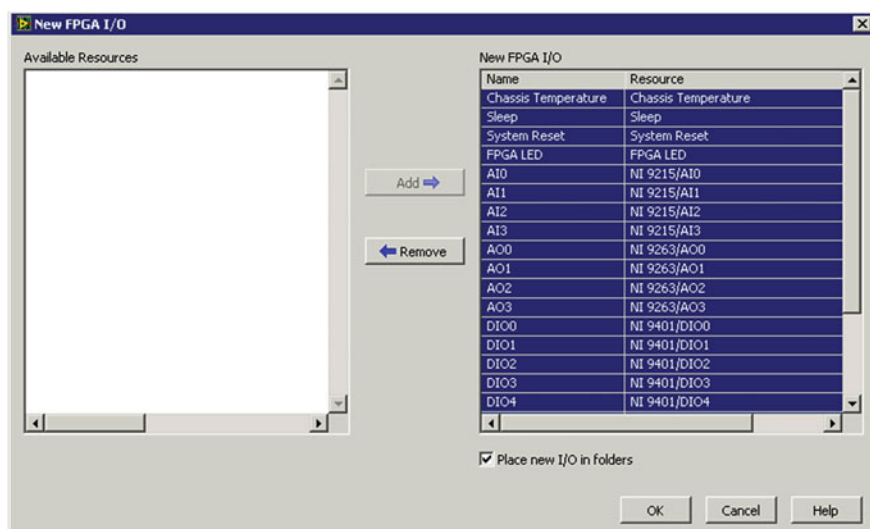
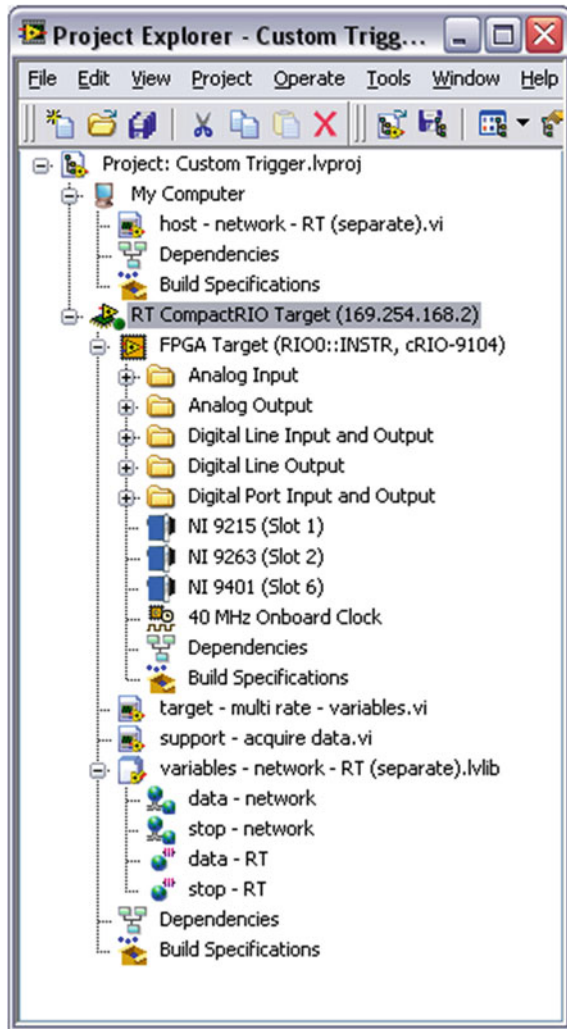


Fig. 2.27 Modules selected in the FPGA

Fig. 2.28 Project explorer



2. Navigate to the block diagram window for your **Simple AIAO (FPGA)** application. (You can also make the block diagram appear by selecting **Window>>Show block diagram** while viewing the front panel.) Right-click in the white area of the block diagram to display the **Functions** palette. Click on the thumb tack icon in the top left corner of the **Functions** palette to tack it down. Then navigate to the **Help** menu on your VI and select **Show context help**.

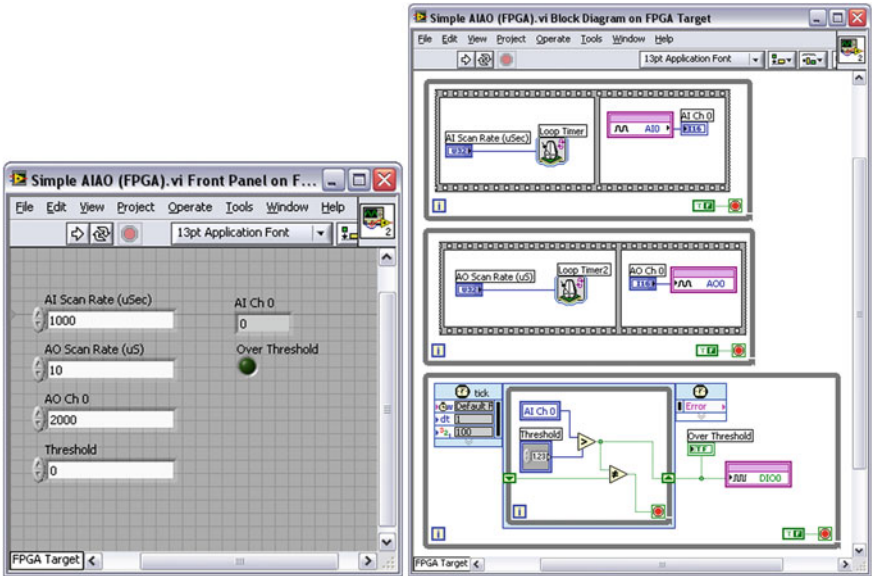


Fig. 2.29 Font panel and block diagram in FPGA application

3. Browse through the **Functions** palette to familiarize yourself with the many IP blocks that ship with LabVIEW FPGA. Be sure to browse through the **Programming Structures**, **Timing**, **Numeric**, **Boolean** logic, **Comparison**, **FPGA Math and Analysis**, **FPGA I/O**, and **Synchronization** palettes (see Fig. 2.31).

If you have the NI SoftMotion Development Module installed, you will see an additional **Spline Engine** function under the **VisionMotion** category that is used for high-performance multiaxis coordinated motion control. Other motion control IP blocks, such as a 32-bit motion PID controller function, are located in the LabVIEW examples directory. To locate these examples, navigate to **Help>>Find Examples**. The NI Digital Filter Design Toolkit is another powerful add-on that provides the ability to generate your own custom signal processing IP blocks for LabVIEW FPGA. For fixed-point filter design, users can model quantization effects, optimize numeric representation/topology, and finally deploy the design using automatically generated LabVIEW FPGA code. A common use for LabVIEW FPGA and RIO hardware is the development of custom triggering logic. In this exercise, you will program the FPGA to read in data from an analog channel, compare it to a threshold, and write a TRUE/FALSE value to a digital channel. If the analog input value exceeds the threshold then a NI 9401 digital output channel will turn on. You will use the indicator LED on the module and software front panel indicators on the FPGA application to view the status.

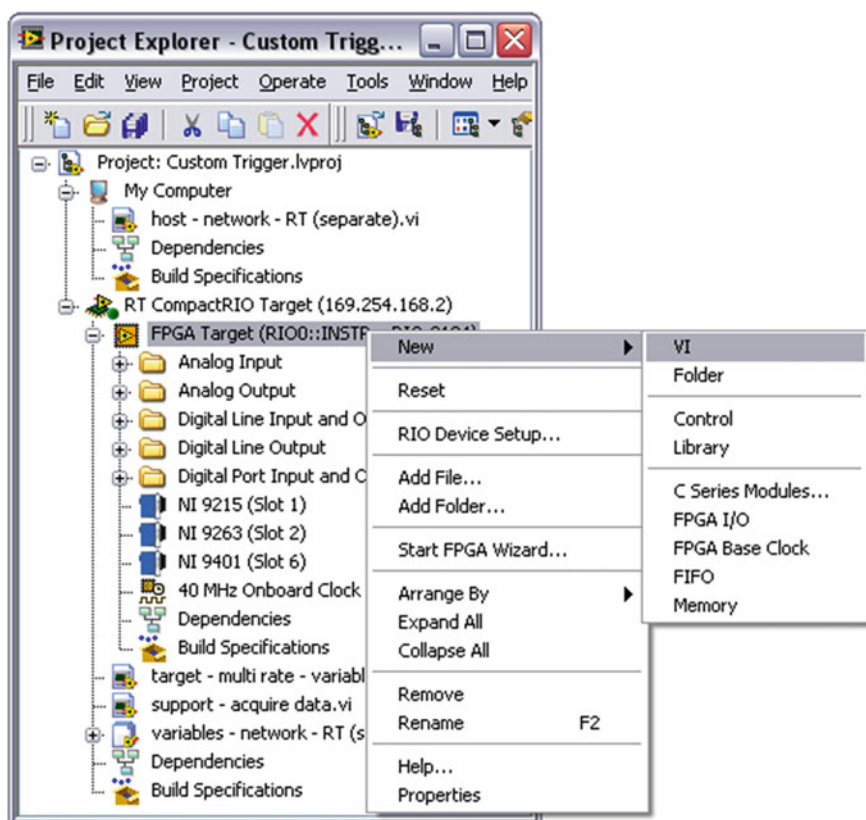


Fig. 2.30 Project explorer

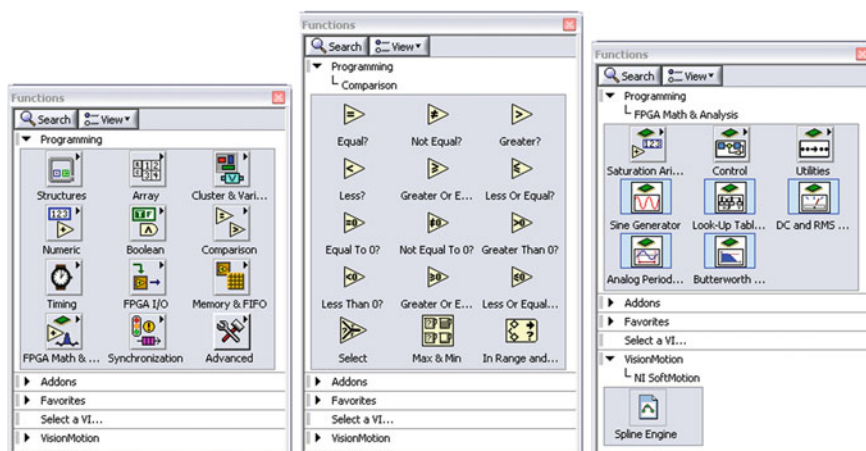


Fig. 2.31 Functions programming

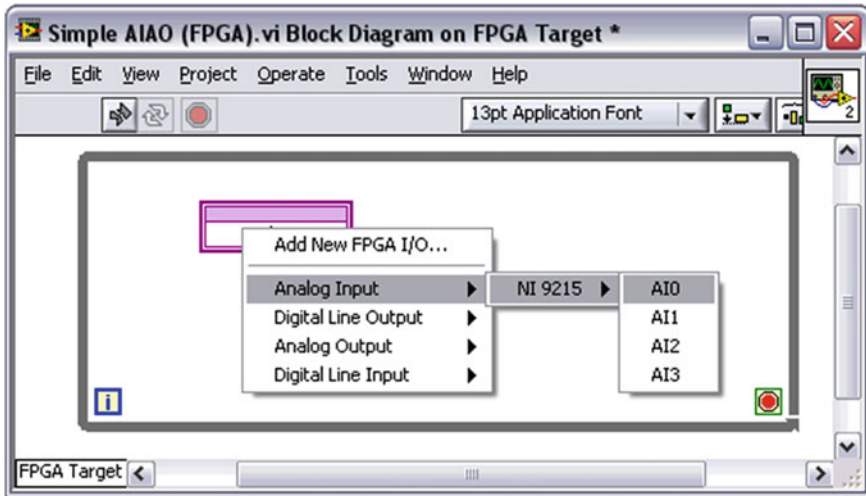


Fig. 2.32 Analog Input

4. We will begin building the application by reading in analog input channel 0 at a timed interval. First, place a **While Loop** from the **Functions>>Structures** palette on the block diagram.
5. From the **Functions>>FPGA I/O** palette, place a **FPGA I/O Node** function inside the **While Loop**.
6. Left-click on the **I/O Name** terminal and select **Analog Input>>NI 9215>>AI0** (see Fig. 2.32).
7. Right-click on the **AI0** terminal and select **Create>>Indicator**. Label the indicator “**AI Ch 0**”.
8. From the **Functions>>Structures** palette enclose the **FPGA I/O Node** and the **AI Ch 0** indicator in a **Flat Sequence Structure**. Then right-click on the border of the sequence structure and select **Add Frame Before**.
9. Expand the left frame to make more room. Then place a **Loop Timer** function (**Functions>>Time and Dialog**) inside the left frame. Select **μsec** as the counter units and **32 Bit** as the size of the internal counter.
10. Right-click on the left input terminal of the **Loop Timer** function and select **Create>>Control**. Label the control “**AI Scan Rate (uS)**”. By timing the loop, this will set the sampling rate of the simultaneous sampling NI 9215 analog input module. Using the sequence structure, you insure that the timing interval between samples is correct even on the first few iterations of the loop.
11. Right-click on the conditional terminal of the while loop (see Fig. 2.33) and select **Create>>Constant**. Make sure the constant is set to the default value of **FALSE**. The FPGA application should appear as shown below.

By placing the loop timing function in the first frame of the flat sequence structure, we ensure that the correct loop timing occurs on the first iteration of

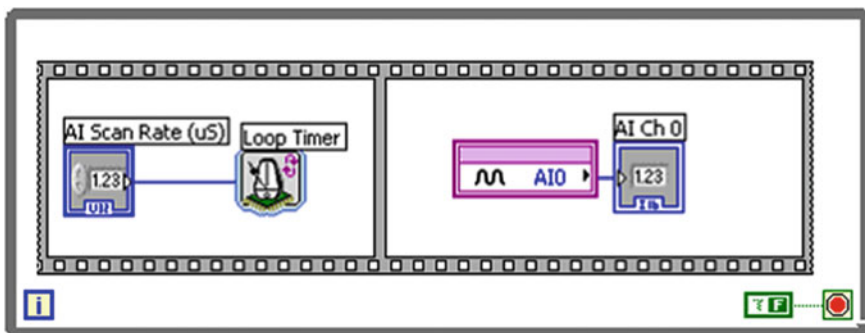


Fig. 2.33 While loop

the loop. If no sequence structure was used, the second acquisition would occur immediately after the first since no delay would be added. That's because on first execution, the **Loop Timer** function sets its internal timing register but does not add a delay to the loop. In general, for any functions placed in parallel on the block diagram with no data dependencies, LabVIEW FPGA will synchronize the start of each parallel function.

12. Place another **While Loop** structure (**Functions>>Structures**) on the block diagram below the analog input loop you just created.
13. Drop down an **FPGA I/O Node** function (**Functions>>FPGA I/O**). Left-click the I/O node and select **Analog Output>>NI 9263>>AO0** to access the analog output channel 0.
14. Right-click on the **AO0** terminal and select **Create>>Control**. Label the control "AO Ch 0".
15. Following the same process as you did before, enclose the **FPGA I/O Node** in a **Flat Sequence Structure**, add a frame to the left, and drop in a **Loop Timer** function. Select **μsec** as the counter units and **32 Bit** as the size of the internal counter.
16. Right-click on the input terminal to the **Loop Timer2** function and select **Create>>Control**. Label the control "AO Scan Rate (uS)." This control will set the update rate of the NI 9263 analog output module.
17. Right-click on the conditional terminal of the while loop and select **Create>>Constant**. Make sure the constant is set to the default value of **FALSE**. The FPGA application should appear as shown below in Fig. 2.34.

Note that in LabVIEW FPGA, each loop will execute in parallel. Unlike processors, FPGAs use dedicated hardware for processing logic and do not have an operating system. FPGAs are truly parallel in nature so different processing operations do not have to compete for the same resources. As a result, the performance of one part of the application is not affected when additional processing is added. Also, multiple loops can run on a single FPGA device at different rates. To learn more, view the FPGA-based Control FAQ.

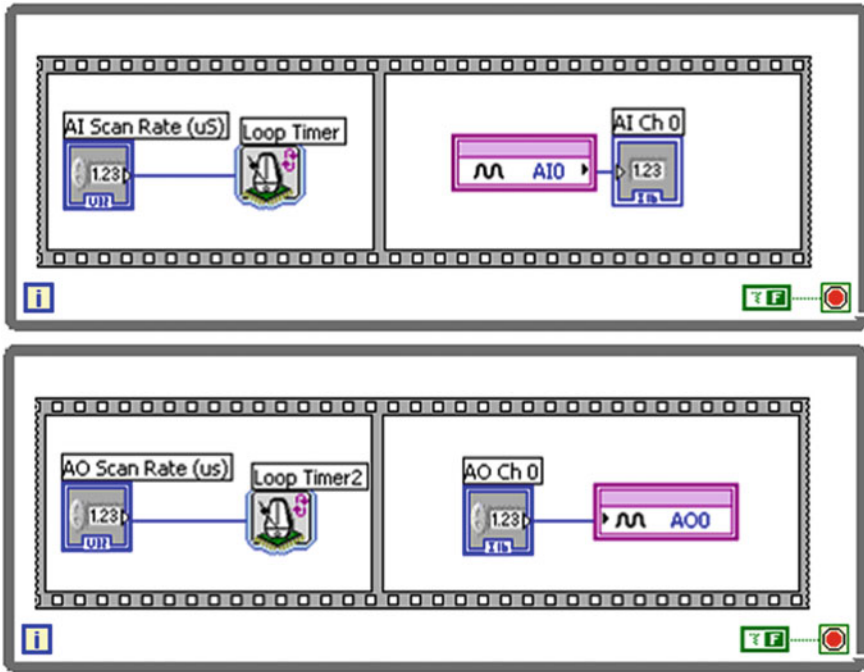


Fig. 2.34 Condition terminal while loop

18. Next we will add another loop to perform custom triggering and digital output. Drop down an additional **While Loop** below the first two loops.
In this application, we will look for a trigger condition to see if the analog input is above or below the threshold. This custom trigger logic is implemented inside a SCTL, which will execute at a 40 MHz rate. If a change is detected, we will set a digital output on the NI 9401 module to the appropriate value. The NI 9474 module has a worst-case output propagation delay of 1 μ s.
19. Place a **Timed Loop (Functions>>Structures>>Timed Structures)** inside the while loop.
20. Right-click on the **AI Ch 0** indicator and select **Create>>Local Variable**. Place the local variable inside the **Timed Loop** structure. Right-click on the local variable and select **Change To Read**.
21. Drop a **Greater?** function (**Functions>>Comparison**) on the block diagram and wire the **AI Ch 0** local variable to the top input terminal. Then right-click on the lower terminal and select **Create>>Control**. Label the control "**Threshold**."
22. Right-click on the border of the **Timed Loop** and select **Add Shift Register**. This register will store the value of the comparison function and pass it from one iteration of the loop to the next. Connect the output of the **Greater?** function to the input of the shift register on the right border of the timed loop.

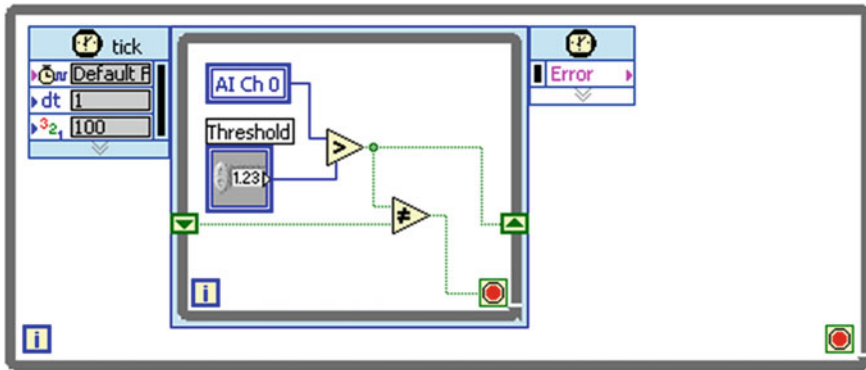


Fig. 2.35 Time loop

23. Place a **Not Equal?** function (**Functions>>Comparison**) to the right of the **Greater?** function and connect the green signal wire to the top terminal of the **Not Equal?** function. Then wire the Shift Register value from the left side of the timed loop to the bottom terminal the **Not Equal?** function. Finally, wire the output of the **Not Equal?** function to the conditional terminal of the timed loop. If the analog in value has risen above or dropped below the threshold value then we will stop the timed loop and update the digital output (see Fig. 2.35).

*The SCTL structure instructs the LabVIEW compiler to execute the code inside of it within a single 25 ns clock cycle of the FPGA (40 MHz). Code inside of a SCTL not only executes faster, but also uses fewer FPGA resources or “slices.” However, certain functions are not supported in the SCTL such as analog input and analog output I/O nodes and the **Quotient and Remainder** function (an integer math divide function). If you use an unsupported function inside of a SCTL, you will get a compile error early in the compilation process.*

24. Right-click the output of the **Shift Register** on the right border of the timed loop and select **Create>>Indicator**. Name this indicator “**Over Threshold**.”
25. Drop down an **FPGA I/O Node (Functions>>FPGA Device I/O)** next to your **Over Threshold** indicator (outside of the timed loop). Left-click the **FPGA I/O Node** and select **Digital Line Input and Output>>NI 9401>>DIO0** to access the digital output channel 0. Right-click on the **FPGA I/O Node** and select **Change to Write**. Then wire the output signal from the right shift register to the **Digital Output** node.
26. Wire a **FALSE Boolean Constant** to the Loop Condition terminal of the outer while loop. The FPGA application should appear similar to that shown below in Fig. 2.36.

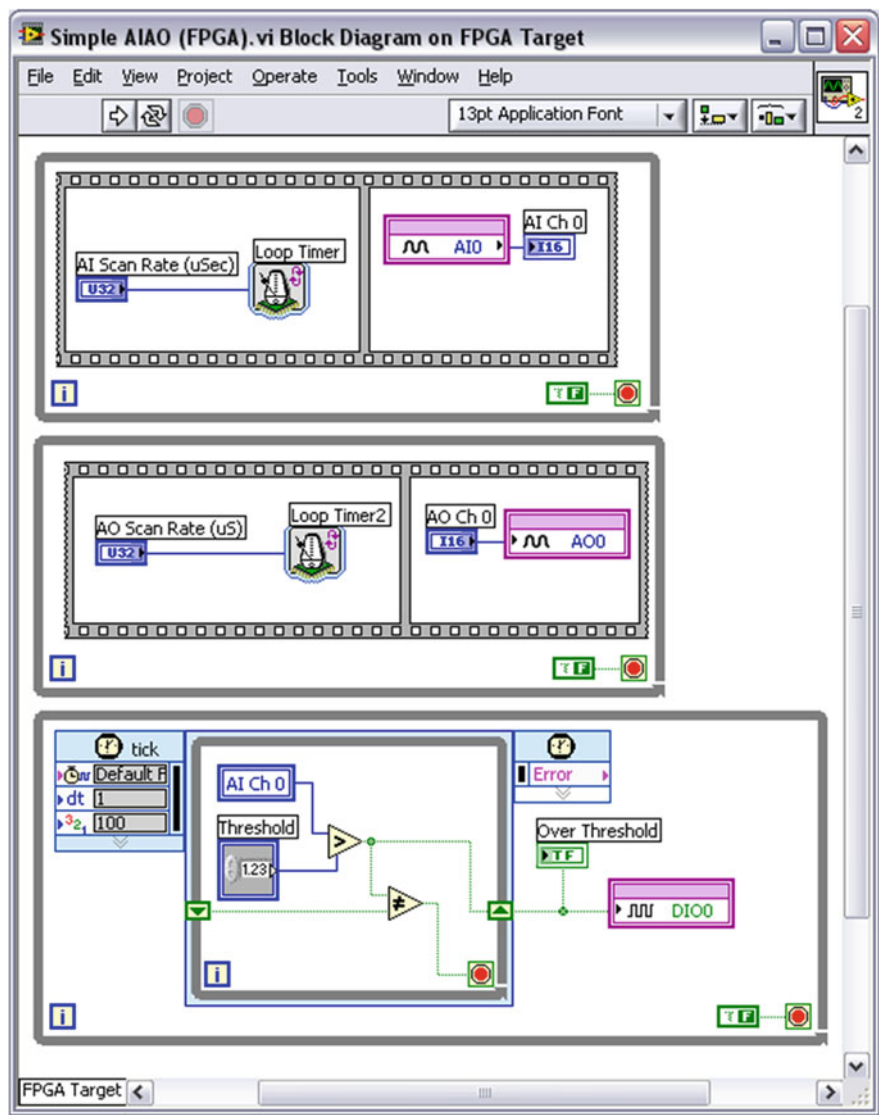
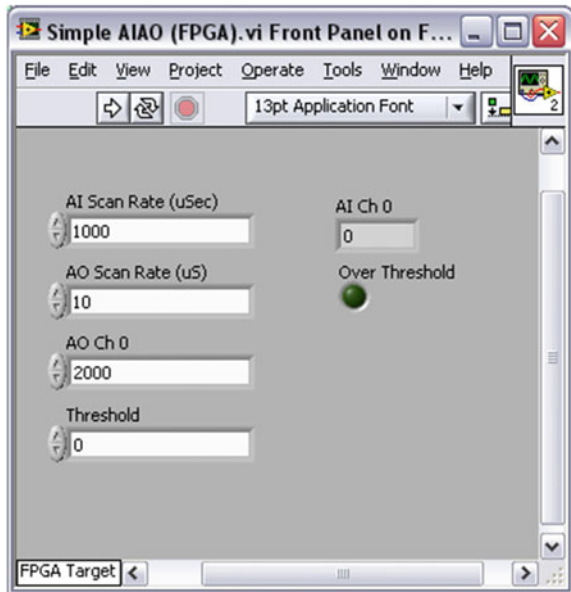


Fig. 2.36 Loop conditional terminal

27. Navigate to the front panel of your FPGA application and arrange the controls and indicators as shown in Fig. 2.37. Set the **AI Scan Rate (uS)** to **1000**. Set the **AO Scan Rate (us)** to **10**. Set **AO Ch 0** to **2000**. Then navigate to the **Edit** menu and select **Make current values default**.

This will set the FPGA application so that at startup the analog output loop runs at 100 kS/s (10 μ s) and the analog input loop runs at 1 kS/s (1000 μ s).

Fig. 2.37 FPGA applications front panel



Setting the **Threshold** to 0 and the NI 9263 analog output value (**AO Ch 0**) to 2000 (about 0.653 V) will cause the digital output to go high due to an over threshold condition when the application starts. For more information about the scaling from integer values to voltage outputs on the NI 9263, refer to the NI 9263 Operating Instructions manual.

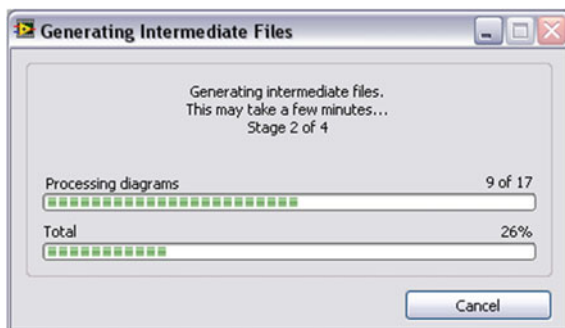
28. When you are ready to compile navigate to **File>>Save All** to save all open applications and the project.

2.3 Compiling the FPGA Application

In this section you will compile the LabVIEW FPGA application and learn more about the compilation process and view the compilation report.

1. Click the **Run** button to start the compile process.
2. Sit back and enjoy 5–10 min of relaxation while your LabVIEW FPGA application compiles. To better understand the LabVIEW FPGA compilation process, review the information below.

Fig. 2.38 FPGA compilation process



2.3.1 Understanding the LabVIEW FPGA Compilation Process

The LabVIEW FPGA Module [7] uses an industry standard Xilinx ISE compiler. First, your graphical LabVIEW FPGA code is translated to text-based VHDL code. At this time, the **Generating Intermediate Files** dialogue is displayed (see Fig. 2.38).

Then the Xilinx ISE compiler tools are invoked and the VHDL code is optimized, reduced, and synthesized into a hardware circuit realization of your LabVIEW design. This process also applies timing constraints on the circuit design that ensure an efficient use of FPGA resources (sometimes called “fabric”).

A great deal of optimization is performed during the compilation process to reduce digital logic and create an optimal implementation of the LabVIEW application. The end result is a highly optimized silicon implementation that provides true parallel processing with the performance and reliability benefits of dedicated hardware circuitry. Since there is no operating system on the FPGA chip, the code is implemented in a way that ensures maximum performance and reliability (see Fig. 2.39).

The end result is a bit stream file that is loaded into your LabVIEW FPGA.VI file. When you run the application, the bitstream is loaded into the FPGA chip to configure the gate array logic. While the application is running on the FPGA, data for the front panel controls and indicators is passed over the network several times per second to enable **Interactive Mode** testing of the application. The update rate in interactive mode communication is typically limited to about 10 S/s. Later you could build a real-time host interface to the FPGA application that enables high-speed data transfer and interrupt synchronization between the floating-point host processor and integer-based FPGA chipset.

The diagram shown in Fig. 2.40 is a summary of the LabVIEW FPGA compilation process. To learn more, click to view an application note on FPGA-based control.

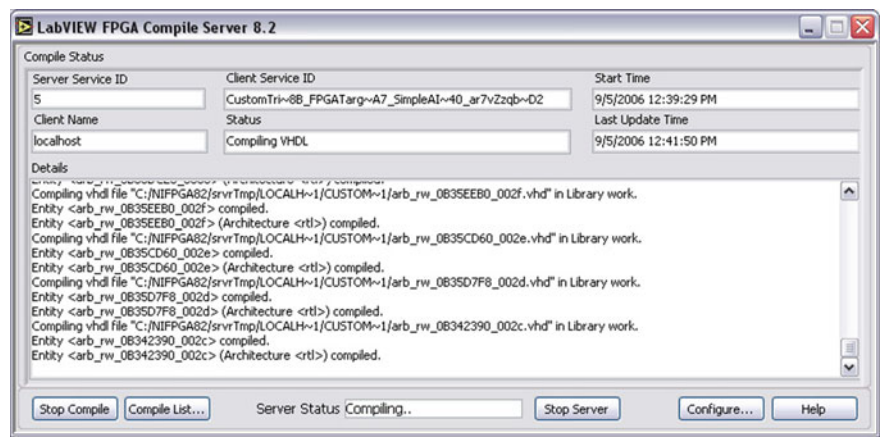


Fig. 2.39 FPGA compilation server

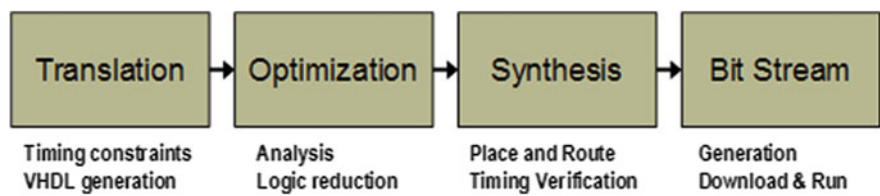


Fig. 2.40 Summary of the LabVIEW FPGA compilation process

2.3.2 FPGA Clock Speed

By default, the FPGA clock runs at 40 MHz. This means that one **Tick** of the FPGA clock is equal to 25 ns. By changing the compile options, you can increase the FPGA clock speed up to 200 MHz (5 ns). There are some drawbacks to using higher clock speeds that you should be aware of before changing the compile option. For more information, refer to the CompactRIO Technical Developers Library by visiting [8] or by right-clicking on the **40 MHz Onboard Clock** item in the project and selecting **Help**.

2.3.3 The Compilation Report

When the compilation is complete, the compile report will be generated. This report shows the start and end compilation time, the number of SLICES used, a compiled clock rate (40 MHz), and an estimated maximum clock rate (see Fig. 2.41).

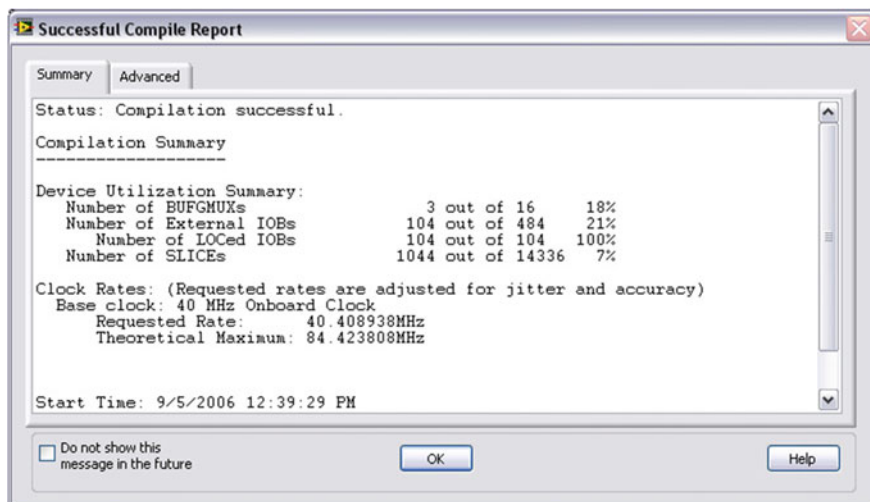


Fig. 2.41 Compilation report

A SLICE is a collection of logic components on the FPGA. The percentage shown is the percentage of the FPGA used. In most cases, you can actually fit more onto the FPGA and run the code faster than this report would lead you to believe. For small applications, the compiler does not “try as hard” to optimize, as long as the timing and other design constraints are met. As the FPGA reaches greater than 90 % usage, the compiler performs heavy optimization to make the most efficient use of resources.

2.4 Advanced Methods for LABVIEW FPGA

This section covers a number of advanced tips and tricks to cut your development time when creating high-performance control systems with LabVIEW FPGA and CompactRIO. It will be introduced the debugging techniques such as simulation that will make you confident before you hit the compile button. You will also learn a number of recommended programming practices, how to avoid common mistakes, and numerous methods to create fast, efficient, and reliable LabVIEW FPGA applications. Throughout this section, we will be walking you a number of examples that were developed to create a high performance control system for a brushed DC motor. We will be showing you a variety of the programming techniques that were used in the creation of LabVIEW FPGA subVIs for generating the PWM Drive signal, decoding the digital pulses from the Quadrature Encoder sensor, and performing PID control to close the motor position loop. The end result is a high performance control system with sub-nanosecond timing jitter, multiple 40 MHz processing loops, and that consumes only 17 % of a 3 million gate FPGA.

2.4.1 Introduction

To help you understand the role of the FPGA in a typical CompactRIO control system application, first we will review a typical software architecture (see Fig. 2.42).

First, the FPGA provides an interface to the I/O modules using an elemental I/O node interface. In some cases, the I/O is as simple as reading the voltage from an analog input module. For more complicated I/O types like the Quadrature encoder sensors that are common in motor control, the FPGA performs additional processing to convert raw digital signals into a meaningful measurement, such as the position and speed of a motor. In addition to I/O, the FPGA is commonly used for analog PID control, digital true/false logic, and event response.

Sending data from the FPGA to the real-time processor is as simple as creating a control or indicator on the front panel of the LabVIEW FPGA application. For high-speed buffered data, you can use DMA to stream data from FPGA memory to processor memory. The FPGA can also generate interrupts, which cause lower priority tasks to be interrupted on the host processor. This provides a way for the FPGA to synchronize the execution of code on the host processor, which can then perform calculations and respond in a deterministic fashion. In general, the real-time processor is slower than the FPGA but offers an extensive palette of floating-point control, math, and signal processing functions.

The CompactRIO processor executes a multithreaded, hard real-time operating system and is programmed using LabVIEW Real-Time. By multithreaded, we mean that it can execute multiple pieces of code, or loops, at different priorities. That

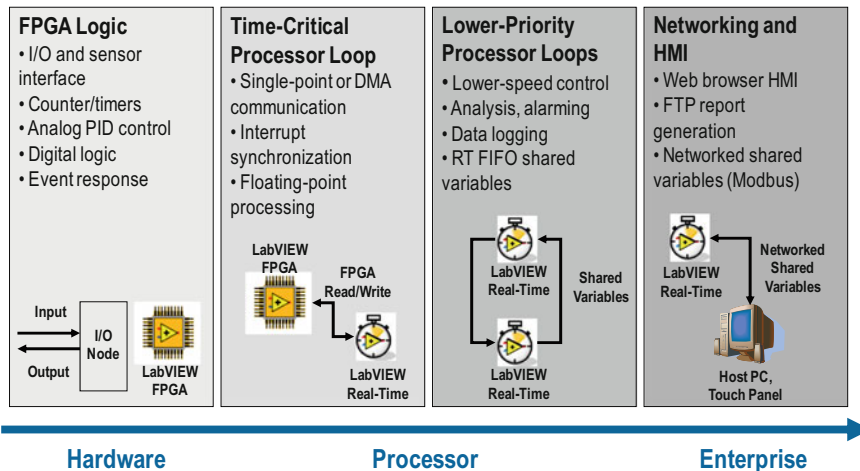


Fig. 2.42 Typical software architecture

means that you can add functionality to your application, such as data logging, lower speed control loops, or alarming without interfering with higher priority tasks that are more time sensitive. You can share data between loops of different priorities without causing interference using RT FIFO shared variables.

While the CompactRIO system is capable of standalone operation, many applications involve networking and a human-machine interface, or HMI. The two easiest ways to communicate with your CompactRIO system are through the web browser HMI or through file transfer protocol, or FTP. CompactRIO features a built-in web server that can host the front panel user interface of the lower priority loops in the embedded application. Alternately, you can use the LabVIEW Touch Panel module to create a low-cost HMI for your system. CompactRIO also has a built-in Modbus server that can publish or receive data from networked devices such as PLCs. Modbus/TCP is one of the most commonly used industrial networking protocols over Ethernet.

Let us take a look at five key development techniques that will help you create reliable and high-performance LabVIEW FPGA applications.

2.4.2 Technique 1: Use Single-Cycle Timed Loops (SCTLs)

The first development technique we will introduce is the use of SCTLs, in LabVIEW FPGA.

SCTLs work by telling the LabVIEW FPGA compiler to optimize the code inside, and add the special timing constraint that the code must execute in a single tick of the FPGA clock. Code compiled inside a SCTL is more optimized and takes up less space on the FPGA compared to the same code inside of a regular while loop. Code inside a SCTL also executes extremely fast. At the default clock rate of 40 MHz, one cycle is equal to just 25 ns.

Below are two identical LabVIEW FPGA applications (see Fig. 2.43)—the one on the left uses normal while loops, while the one on the right uses SCTLs in its subVIs. This example shows off the power of parallel processing. The upper loop is reading and processing the digital signals from a quadrature encoder sensor on a motor and the lower loop is performing PWM, or PWM, to control the amount of power being sent to the motor. This application is written for the NI 9505 motor drive module which can control up to 8 A, 30 V brushed DC motors. This code runs extremely fast—in the application on the right we are running two different loops at a 40 MHz clock rate.

The results from our compile report are also shown. The application built with SCTLs uses fewer SLICES, but it takes longer to compile because the compiler has to work harder to meeting the timing constraints applied by the SCTL.

Now let us take a look at how the SCTL works in more depth.

When code is compiled in a normal while loop, LabVIEW FPGA inserts flip-flops to clock data from one function to the next, thereby enforcing the synchronous dataflow nature of LabVIEW and preventing race conditions. The

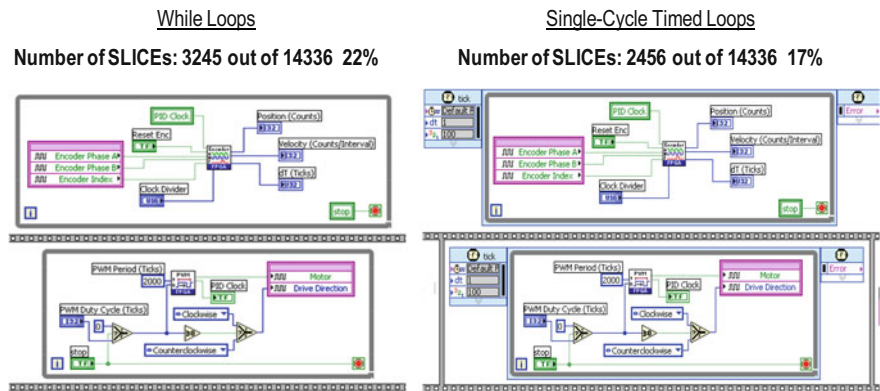


Fig. 2.43 Two loop cycles

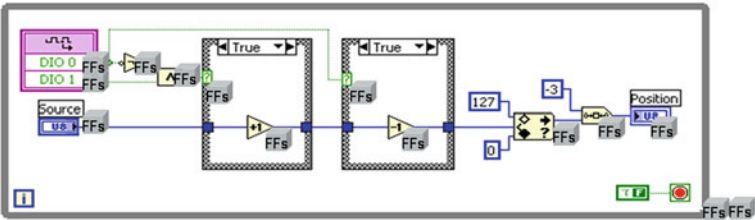


Fig. 2.44 Single-cycle time loop

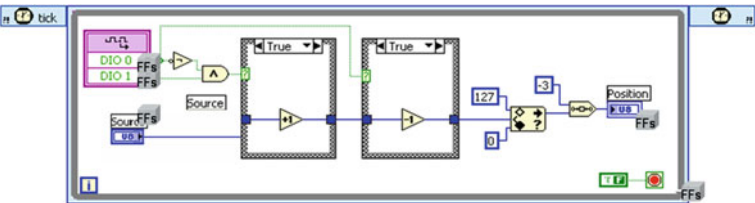


Fig. 2.45 Single-cycle time loop only I/O have flip-flops


flip-flops are marked here with the FF boxes drawn at the output of each function (see Fig. 2.44).

Below is the same code compiled into a SCTL. Here you see that only the inputs and outputs of the loop have flip-flops (see Fig. 2.45). The internal code is implemented in a more parallel fashion and more logic reduction is done to optimize the code in between the inputs and outputs of the loop.

As you can see, SCTLs are a simple way to optimize your LabVIEW FPGA code. So what is the catch? Why would not you always use the SCTL? There are some limitations to the use of SCTLs as it is shown in Table 2.2.

Table 2.2 Single cycle time loop limitations

Items not allowed in SCTL	Suggested alternative
Long sequences of serial code	Make the code more parallel. Insert feedback nodes in the wires to add pipelining
Quotient and remainder	Use a scale by power of 2 to do integer division, or use the fixed-point math library
Loop timer, wait functions	Use a tick count function to trigger an event instead
Analog input, analog output	Place in a separate while loop and use local variables to send data
While loops	For nested subVIs, use feedback nodes to hold state

To use the SCTL all operations inside the SCTL must fit within one cycle of the FPGA clock. In the beginning of the compile process, the code generator will give an error message if the SCTL cannot generate the proper code for the compiler. That means that long sequences of serial code may not be able to fit in a SCTL. By serial code, we mean code where the results of one calculation are needed by the next operation, preventing the calculations from being executed in parallel. To fix this you can rewrite the code to make it more parallel. For example, you can insert a Feedback Node () to pass the results from one calculation to the next on the following iteration of the loop—this also known as pipelining. You can use the pipelining technique to reduce the length of each run through the SCTL by breaking up the code among multiple iterations of the SCTL.

The **Quotient and Remainder** function is another one that cannot be used in a SCTL. If you need to divide a number by an integer value, you can use the **Scale by Power of 2** function instead. This function lets you multiply or divide by powers of two, i.e., 2, 4, 8, 16, 32, etc. For a fixed-point result, you can use the Fixed-Point Math Library for LabVIEW FPGA. The fixed-point divide subVI and configuration panel is shown in Fig. 2.46, including the Execution Mode control which enables the function to be used within a SCTL.

The Fixed-Point math Library contains LabVIEW FPGA IP blocks that implement a variety of elementary and transcendental math functions. These functions use the Fixed-point data type introduced in LabVIEW 8.5 extending the current offering of functions to include Divide, Sine, Cosine, and many more important math operations. All functions are verified for usage inside and outside a SCTL as well as in Windows and FPGA simulation on the Development computer. The toolkit comes with help documentation that includes details for each function to learn more about individual usage. To download the free toolkit, follow the link below [9].

If you are trying to make a subVI for use inside of a SCTL, you can use a feedback node to hold state information in the subVI. This eliminates the need to use a while loop inside of a SCTL. The LabVIEW FPGA example below calculates one of the differential equations for a DC motor using functions from the

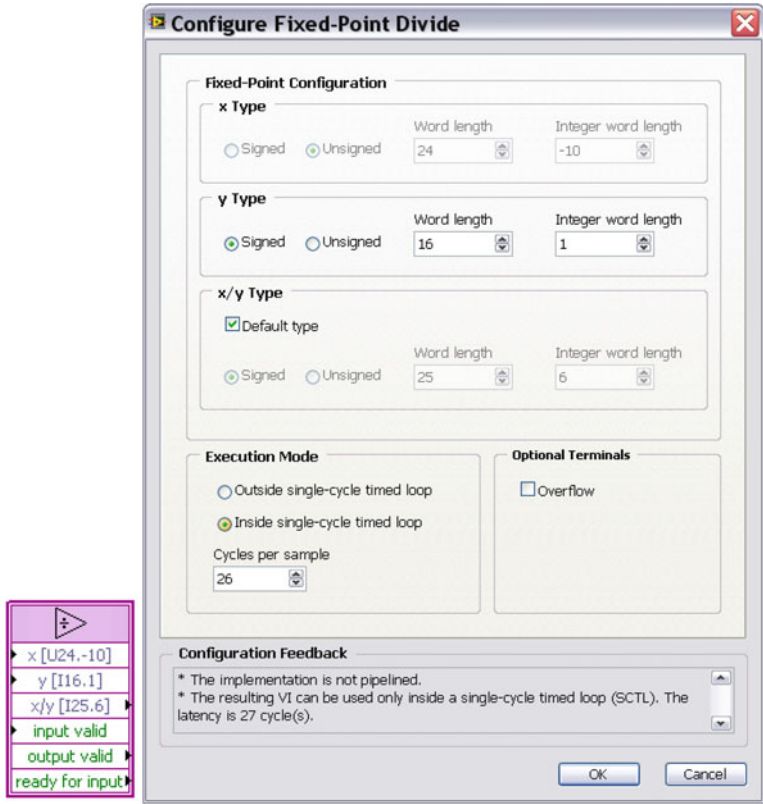


Fig. 2.46 Fixed-point divide

fixed-point math library (see Fig. 2.47). After each fixed-point math function, a feedback node is used to pipeline the result and thereby pass the value from one iteration to the next. In the upper right corner, a **Tick Count** function is also used in combination with a feedback node to calculate the loop rate of the subVI execution.

In Fig. 2.48 you can see the top-level SCTL in the FPGA application, which calls the motor simulation subVI. Since the subVI is nested within a SCTL, the **Loop Rate (Ticks)** value returned is always equal to 1. However, due to pipelining there is a six-tick latency from the **voltage (V)** input to the **i (A)** current output of the subVI.

In addition to pipelining, you can use a State machine within the SCTL to better organize your code and run through a sequence of steps. The basic component of the State machine is a Case structure with each containing one state and using a shift register to determine the next state after each iteration of the loop. Of course each state must be able to run in one clock cycle if the subVI is to be placed in a SCTL. In addition, you can use shift registers and a counter value to implement the functionality of a **For Loop** or add a specific number of Wait states to your program execution.

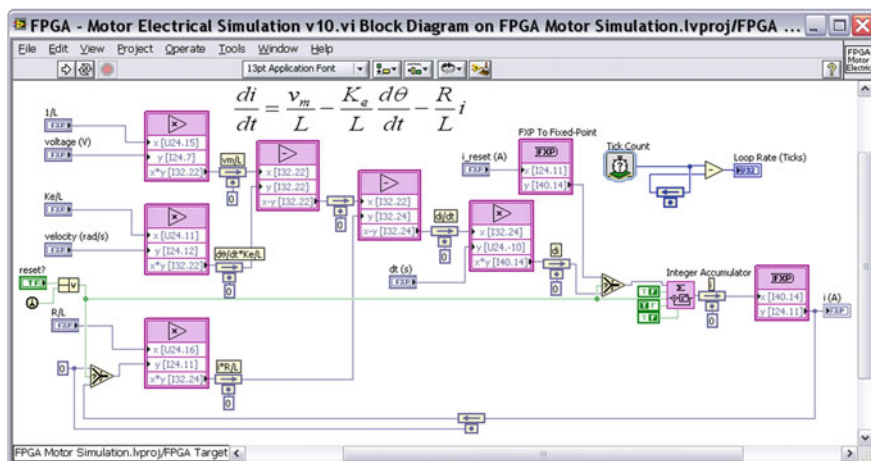


Fig. 2.47 DC motor differential equation

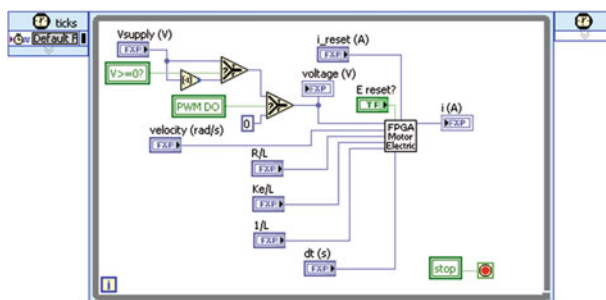


Fig. 2.48 SCTL in the FPGA application

Note: Adding a loop timer or wait function will cause the code to execute slower than one tick, and therefore cannot be used within a SCTL. Analog input and analog output functions also take more than one clock tick to execute and cannot be used in a SCTL. However, you can put them a normal while loops and use local variables to share data with the SCTLs.

2.4.3 Creating Counters and Timers

If you need to trigger an event after a period of time, use the Tick Count function to measure elapsed time as shown in Fig. 2.49. Do not use the iteration terminal that is built into while loops and SCTLs because it will eventually saturate at its maximum value. This happens after 2,147,483,647 iterations of the loop. At a 40 MHz clock

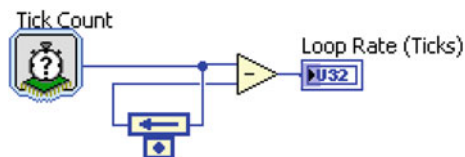


Fig. 2.49 Tick count function to measure elapsed time

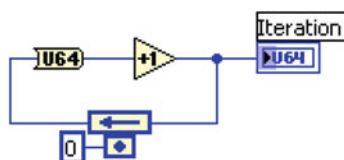


Fig. 2.50 An iteration counter

rate, this takes only 53.687 s. Instead, make your own counter using an unsigned integer and a feedback node. The tick count function is to provide time based on the 40 MHz FPGA clock.

By using an unsigned integer for the counter value, elapsed time calculations will still be correct when the counter rolls over. This is because if you subtract one count value from another using unsigned integers, you still get the correct answer even if the counters overflows.

Another common type of counter is an iteration counter that measures the number of times a loop has executed (see Fig. 2.50). Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over. The unsigned 64-bit integer data type we are using for the counter provides a huge counting range—equivalent to about 18 billion-billion. Even with the FPGA clock running at 40 MHz, this counter will not overflow for more than 14,000 years.

Now let us talk about another technique that will help you create well-written and efficient LabVIEW FPGA code.

2.4.4 Write Your FPGA Code as Modular, Reusable SubVIs

The next major development technique we will suggest is modular development—break your application into independent functions that can each be individually specified, designed, and tested. It seems like a simple concept, but for FPGA development it can have some especially nice benefits. Here is a simple example—a function designed to measure the rate of the loop in which it is placed and count the number of time it executes. Inside the loop we have a **Tick Count** function that reads the current FPGA clock and subtracts it from the previous value, which is stored in a shift register. In addition, we have a 64-bit counter that increments each

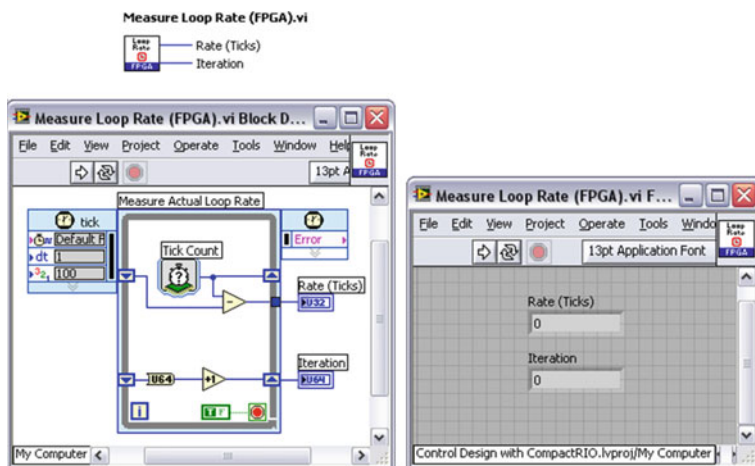


Fig. 2.51 FPGA subVI

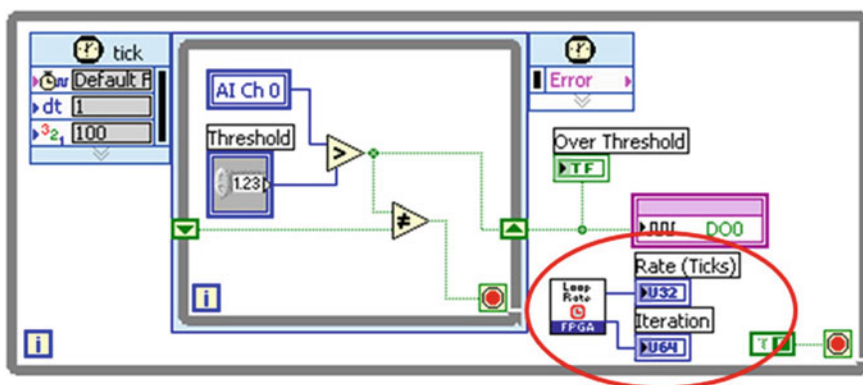


Fig. 2.52 The subVI placed inside another loop

time the function is called. This function uses a SCTL so it only takes a single 25 ns clock tick to execute—therefore, this subVI is designed to be placed inside of a normal while loop without affecting its execution speed (see Fig. 2.51).

Here is the front panel. The indicators have been assigned to the two right terminals of the subVI so data can be passed to the upper level LabVIEW FPGA application in which it is placed.

Fig. 2.52 shows an application example where the function is used. The subVI is placed inside another loop to measure the execution rate of the top-level code.

Some of the top benefits of writing the code this way are presented in Table 2.3.

Writing modular code is almost always a good idea, but when you are designing FPGA logic it has extra advantages.

Table 2.3 Top benefits of writing the code

Benefit	Explanation
Easier to debug and troubleshoot	Code can be tested on Windows before compiling
Easier to document and track changes	Help information can be included in the VI documentation
Creates a cleaner, more easily understood top-level diagram	Code is more intuitive to other programmers

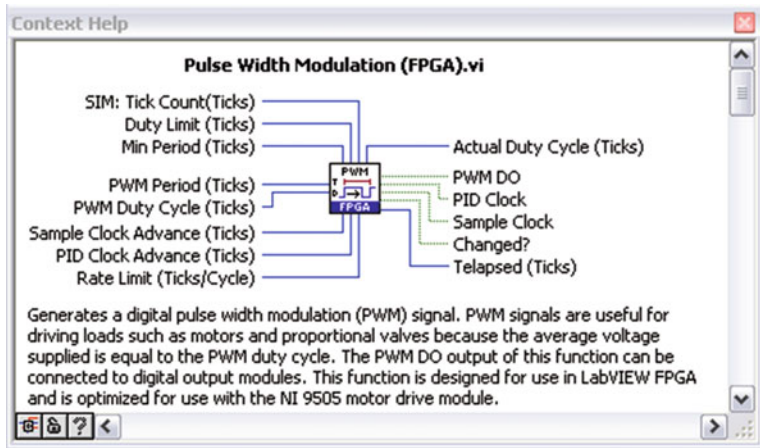


Fig. 2.53 PWM example

First, the code is easier to debug and troubleshoot. One big benefit is that the subVI can be tested on Windows before you compile it for the FPGA. We will show some examples of that later.

Second, it is easier to document and track changes because the code is modular and you can include help information in the VI documentation.

Third, the intended functionality of the code is typically cleaner, easier to understand, and more reusable. The options you want to offer the programmer are typically made available as terminals on the subVI. Most often the user will not need to modify the underlying code—they can just use the parameters you provide, such as in this **Pulse Width Modulation (FPGA)** example (see Fig. 2.53).

Now let us take a look at some tips for how to create modular, reusable subVIs for LabVIEW FPGA.

2.4.5 Separate Logic from I/O

The first tip is to keep I/O nodes out of your subVIs. This makes them more modular and portable and makes the top-level diagram more readable. Particularly for control applications, it is nice to have all of the I/O operations made clearly visible when viewing the top-level diagram for the application, like we have shown here in this PWM loop written for the NI 9505 motor drive module (see Fig. 2.54).

Rather than embedding the I/O node into the subVI, a terminal is used to pass the data from the subVI to the top-level diagram. This makes the FPGA code easier to debug, since the subVI can be tested individually in Windows using simulated I/O. This will be explained in more detail in a subsequent section (see Fig. 2.55).

Taking this approach also tends to reduce unnecessary I/O node instances that might otherwise be included multiple times in the subVI, resulting in unnecessary gate usage due to the need for the compiler to add the extra arbitration logic necessary to handle multiple callers accessing the same shared resource.

Finally, this approach makes the top-level application more readable—all I/O read and write operations are explicitly shown on the top-level diagram and not hidden from view.

Often when you are writing function blocks like this, the subVI will need some local memory capability so it can hold state values, such as elapsed time, and pass that information from one iteration to the next.

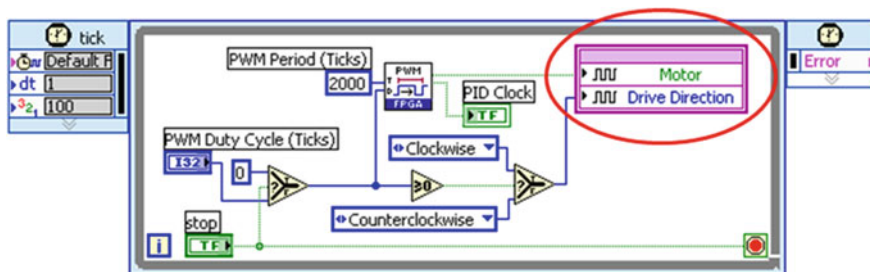


Fig. 2.54 PWM example for the NI 9505

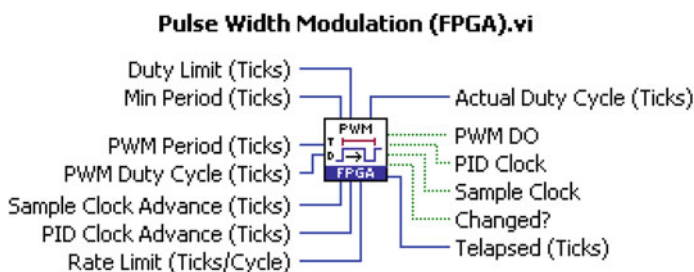


Fig. 2.55 PWM SubVI

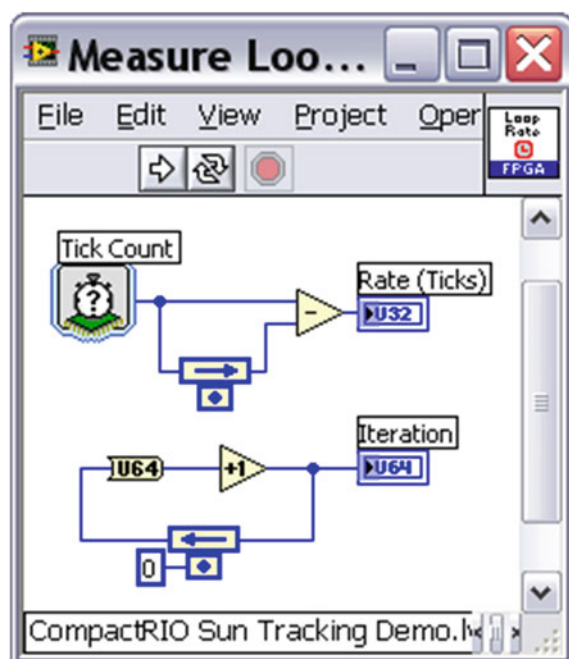


Fig. 2.57 Measure loop

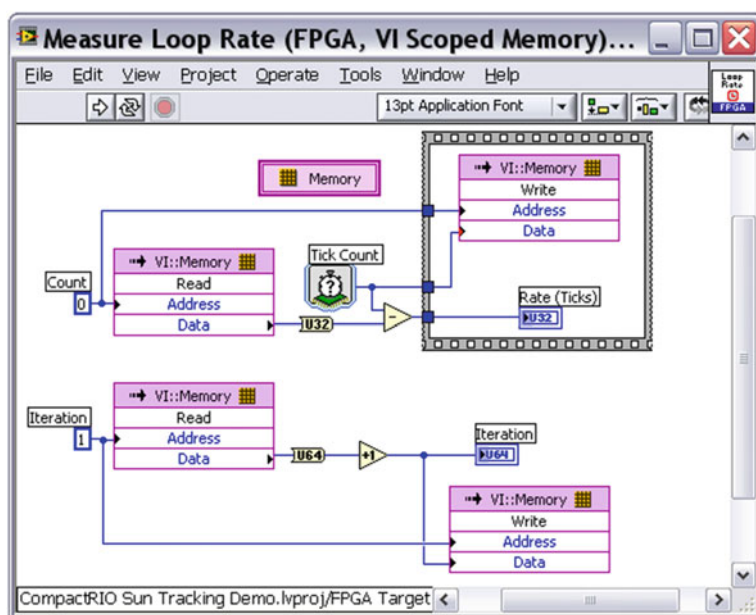


Fig. 2.58 VI memory

added to the project. This makes the code more modular and portable when moving it between projects.

In this simple example, using VI-scoped memory is probably overkill for the application. We only have two memory locations and we are only storing one data point in each memory location. However, VI-scoped memory is a powerful tool for applications that need to store arrays of data. In general, you should always avoid using large front panel arrays as a data storage mechanism—use VI-scoped memory instead.

2.4.7 Run-Time Updateable Look-up Table (LUT)

A common use for local memory in FPGA-based control applications is to store table data, such as the calibration table for a nonlinear sensor, a pre-calculated math formula (such as log or exponential), or an arbitrary waveform that can be replayed by indexing through the table addresses. Below is an FPGA-based look-up table (LUT) configured to store 10,000 fixed-point values and perform linear interpolation between stored values. Because VI-scoped memory is used, the LUT values can be changed while the application is running and without the need to recompile the FPGA (see Figs. 2.59 and 2.60).

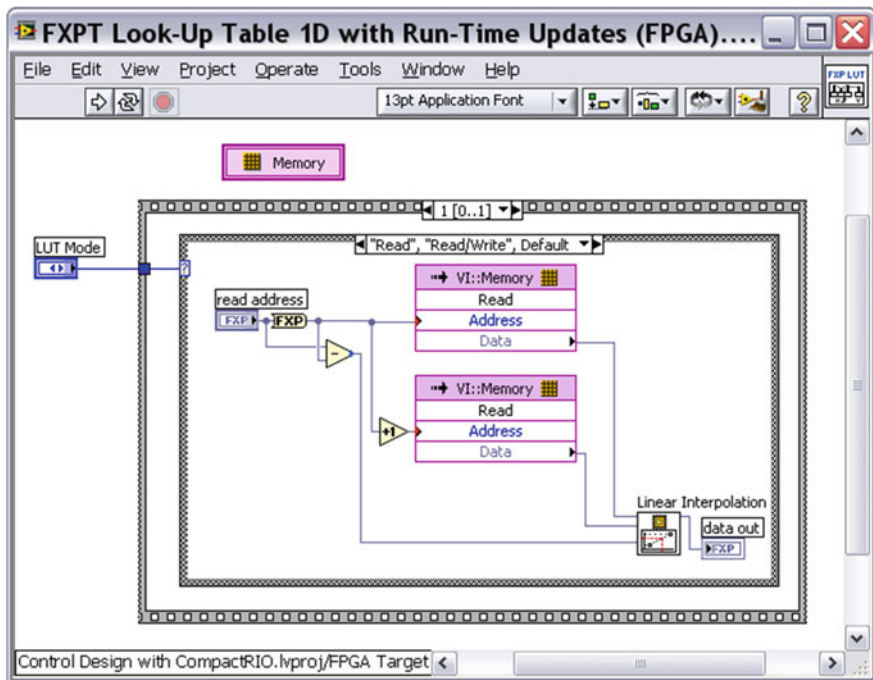


Fig. 2.59 Look-up table

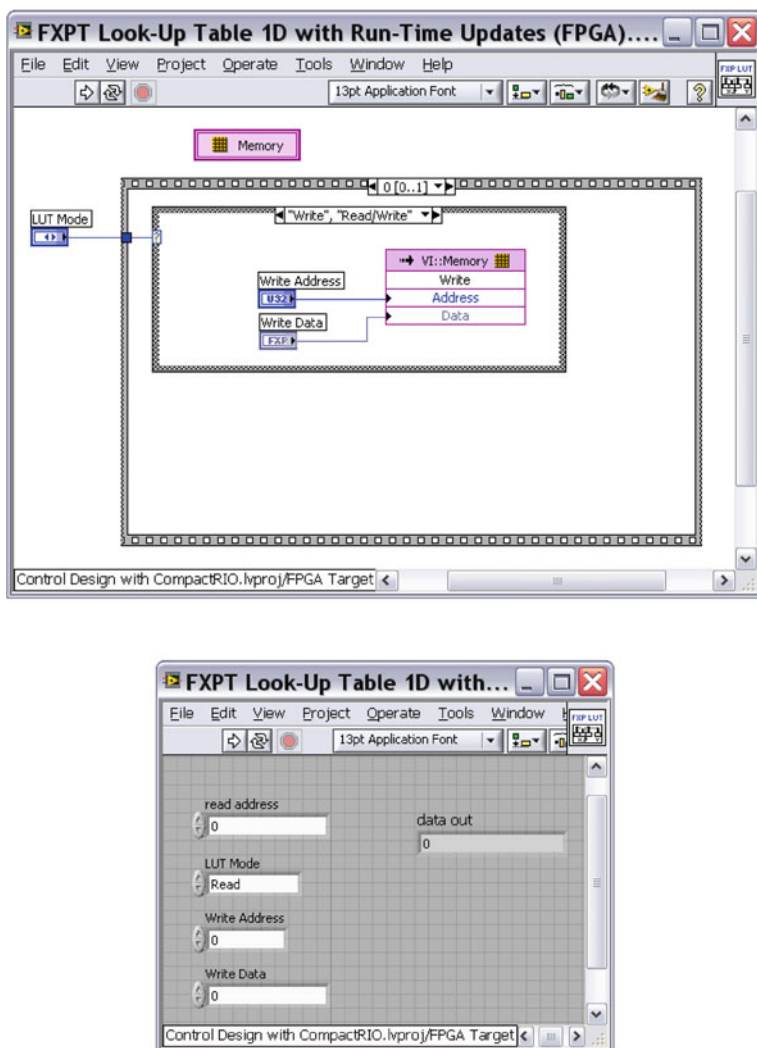


Fig. 2.60 Fix point frontal panel and block diagram

Let us take a look at the configuration pages for the VI-Scoped Memory block in this example. You can configure the depth, data type, and even define initial values for the memory elements (see Fig. 2.61).

Now let us look at another tip for creating modular FPGA subVIs that have to do with the timing of how the code runs.

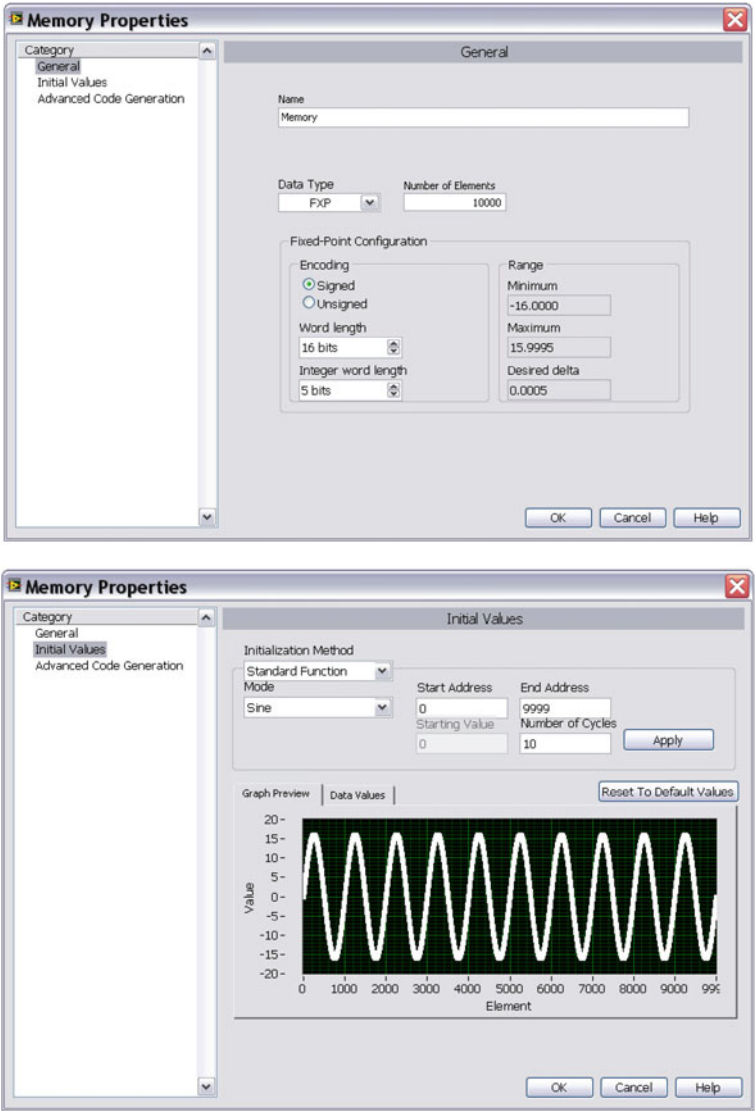


Fig. 2.61 Memory properties

2.4.8 Do not Place Delay Timers in the SubVI

In general, it is a good idea to avoid using Loop Timer or Wait functions within your modular subVIs. If the subVI has no delays, it will execute “as fast as possible” and thereby inherit the timing properties of the calling VI, rather than

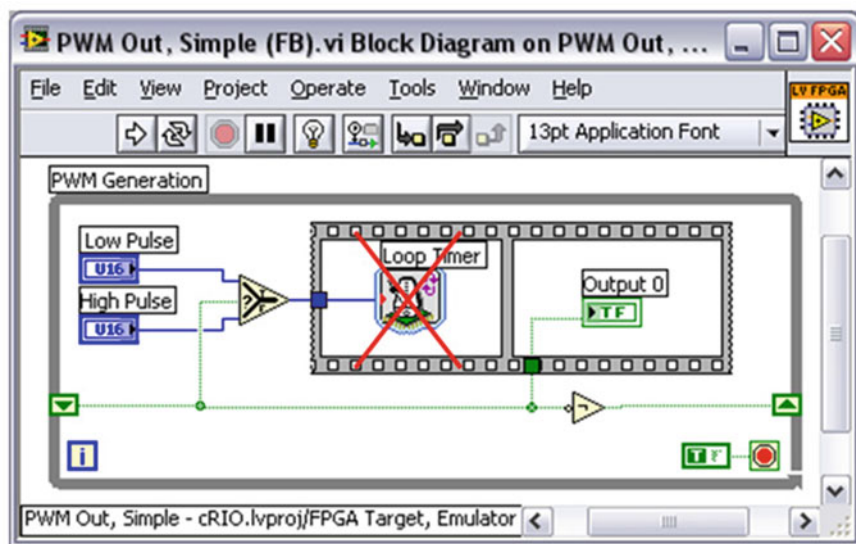


Fig. 2.62 Do not place delay times

slowing down the caller. Also, code can typically be more easily adapted for use in a SCTL if it has no internal functions that cause delays (see Fig. 2.62).

Below we have adapted the PWM code on the left to use a **Tick Count** function rather than a **Loop Timer** function. We use a feedback node to hold an elapsed time count value, and we turn the output on and off at the appropriate times and reset the elapsed time counter at the end of the PWM cycle. The code may look a bit more complicated, but it can be dropped inside of a top-level loop without affecting the overall timing of the loop—it is more portable (see Fig. 2.63).

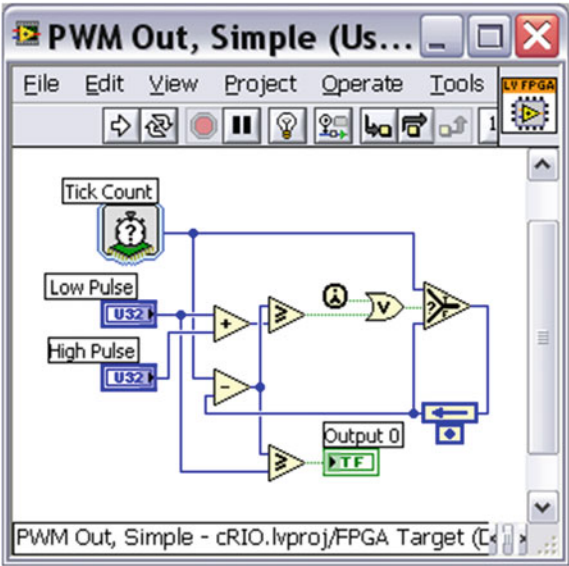
Now let me share one more tip before we move on to the next topic—how to make the code so that multiple copies of a subVI can be placed in the same application and each copy is independent of the others.

2.4.9 Reentrancy

Reentrancy is a setting in the subVI execution properties that enable multiple copies of the function block to be executed in parallel with distinct and separate data storage (see Fig. 2.64).

Figure 2.65 shows an example. In this case our subVI is set to reentrant, meaning all four of these loops will run simultaneously and any internal shift registers, local variables, or VI-scoped memory data will be unique to each instance.

Fig. 2.63 Tick count function



In the case of LabVIEW FPGA, it also means that each copy of the function uses its own FPGA slices—so reentrancy is great for code portability but it does use more gates.

Note: If you are really squeezed for FPGA gates, you can make your function multiplexed rather than reentrant. This is an advanced topic we will not cover here but it basically involves using local memory to store the register values for each of the calling loops, which identify themselves with an integer “ID tag” value. Since

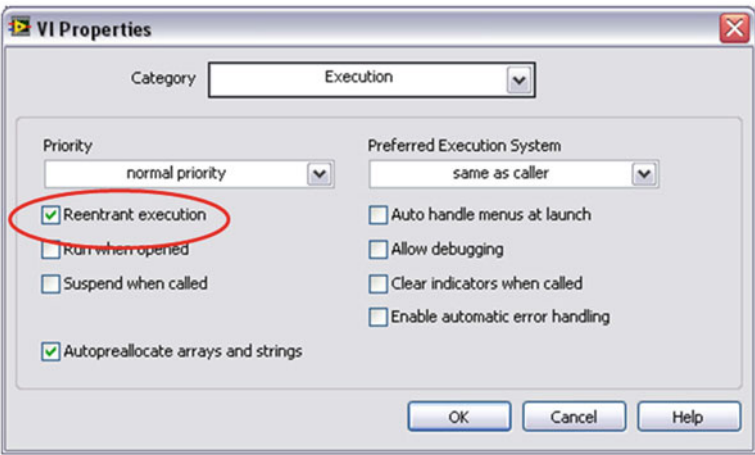


Fig. 2.64 Reentrancy

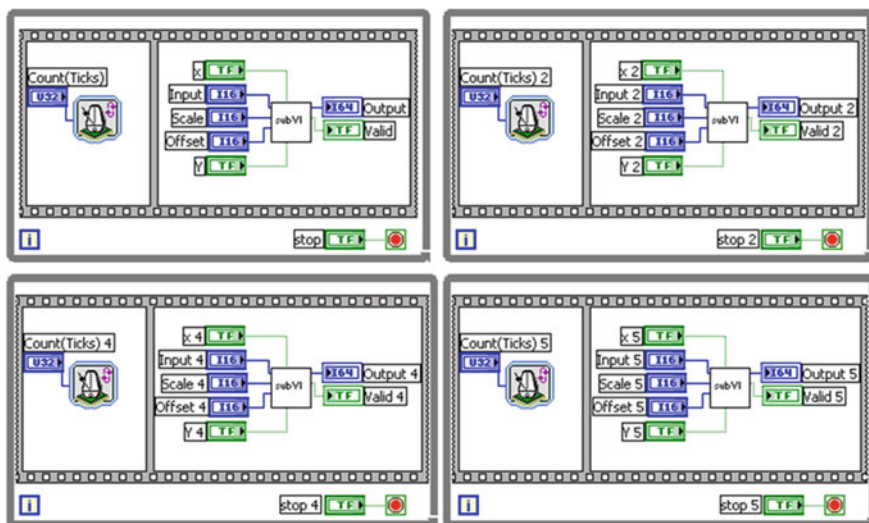


Fig. 2.65 SubVI is set to reentrant

the loops are all using the same underlying FPGA slices (with different memory addresses for the data), each caller will block the other callers resulting in slower execution. However, gate usage is much less since the same hardware SLICE logic is reused. For many control applications where the FPGA is already much faster than the I/O, this is a nice option for saving gates. Several functions on the LabVIEW FPGA palette use multiplexing techniques to enable high channel count operation with minimal FPGA gate usage. These include the **PID**, **Butterworth Filter**, **Notch Filter**, and **Rational Resampler** functions. To see how this works, drop one of these functions onto the block diagram and configure it for multiple channels. Then right-click on the function and select **Convert to SubVI** to reveal the underlying code.

Now let us take a look at a major development benefit you get from writing your LabVIEW FPGA code as described in the sections above.

2.5 Use Simulation Before You Compile

This third development technique is really powerful because it provides a way to get around the longer compilation time and more limited debugging capabilities of LabVIEW FPGA. One of the most powerful aspects of LabVIEW code for embedded developers is the portability of the code. Code written for LabVIEW FPGA is still just LabVIEW code—it can be run on Windows or other devices and operating systems. The main difference between these processing targets is the speed at which the code runs and whether they support true parallel processing like

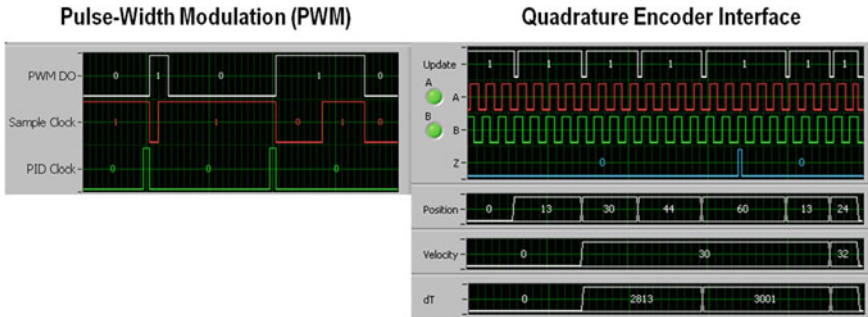


Fig. 2.66 FPGA simulation

an FPGA or simulated parallel processing like a multithreaded operating system for a microprocessor.

LabVIEW FPGA includes the ability to run the entire LabVIEW FPGA application in simulation mode, and this can be done in conjunction with the host processor application for testing purposes with either random data used for FPGA I/O read operations or using a custom VI to generate the simulated I/O signals. This is particularly useful for testing FPGA to host communication including DMA data transfers.

However, the disadvantage of this approach is that the entire FPGA application is simulated. For the development and testing of new LabVIEW functions, it can be advantageous to test the code one function at a time. This section will focus on a capability called functional simulation, which enables a “divide and conquer” approach to debugging which allows each function to be tested individually before compiling to the FPGA. Below are screens from two functional simulation examples running on Windows that were used for testing and debugging purposes (see Fig. 2.66).

The example below shows the front panel and block diagram of a test application used to debug a LabVIEW FPGA subVI for PWM. The test application is located in the **My Computer** section of the LabVIEW project, and when it is opened it runs in Windows (see Fig. 2.67).

2.5.1 Providing Tick Count Values for Simulation

The Conditional Disable Structure in LabVIEW lets you modify what underlying code is used when the subVI is compiled for different processing targets. In this case, I have got a Tick Count function that is executed when the code is compiled for the FPGA and a front panel control that is executed when the code is executed on Windows. This lets me use a “simulated” tick count value when I am testing the

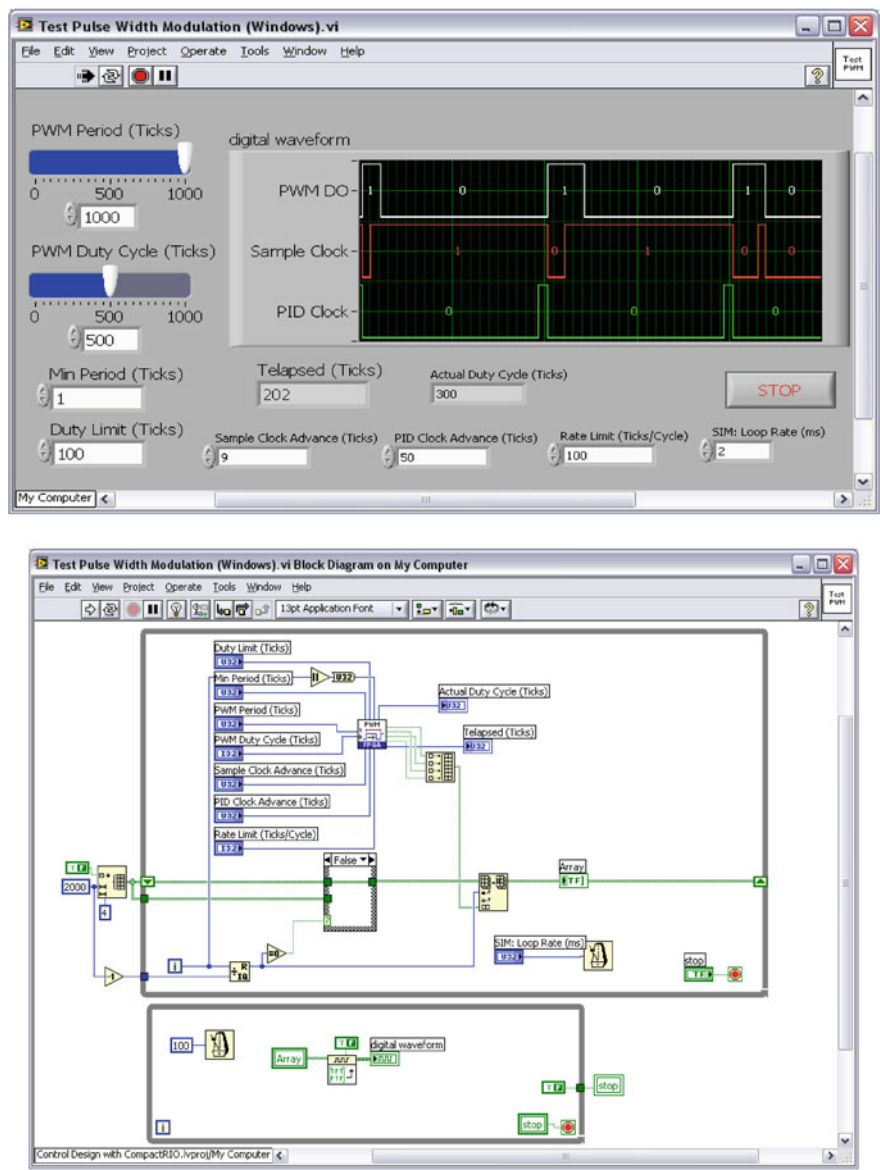


Fig. 2.67 Debugging a LabVIEW FPGA subVI for pulse width modulation

code in Windows, providing the ability to create both bit accurate and cycle accurate simulations (see Fig. 2.68).

This technique is used in the PWM test example above—when the subVI is executed in Windows a simulated FPGA clock is passed to the subVI using the Iteration terminal of the top-level while loop.

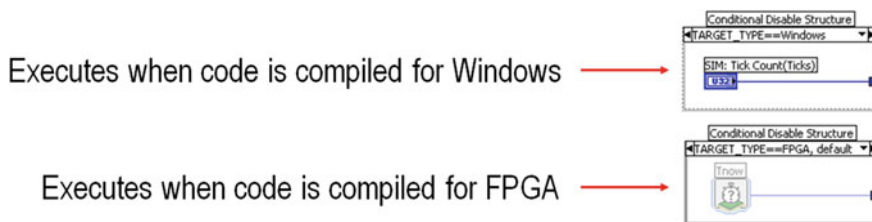


Fig. 2.68 FPGA code compiled by Windows and FPGA

As you have seen, functional simulation lets you test, iterate and be confident in your FPGA logic before you compile. It also enables you to use the full LabVIEW debugging toolset while the code is running, and you can create “test patterns” that enable you to verify the code under a variety of conditions that otherwise might be hard to test. Here are some of the top benefits of using simulation as a step in your development process.


- Quickly iterate and add features
- Be confident in your LabVIEW FPGA code before you compile
- Use full LabVIEW debugging capabilities (probes, highlight execution, etc.)
- Verify the code under a variety of conditions

Now let us take simulation a step farther and create a simulation that accurately mimics the dynamic closed-loop behavior of the physical system within which our LabVIEW FPGA code will be connected.

2.5.2 *Test the LabVIEW FPGA Code Using the LabVIEW Control Design & Simulation Module*

The LabVIEW Control Design & Simulation Module (CD&Sim) includes state-of-the-art technology for simulating mechatronic systems like the DC motor we will be controlling with our LabVIEW FPGA application. Figure 2.69 shows the theoretical model equations for a brushed DC motor driven by a PWM chopper circuit and connected to a simple inertial load with viscous friction.

This is implemented using a LabVIEW CD&Sim subsystem containing formula node. The two differential equations shown above are entered into the formula

nodes in text format as shown below. Integrator functions () are used to convert from higher order derivatives, such as from acceleration to velocity and from velocity to position (see Fig. 2.70).

The Brushed DC Motor.vi subsystem is placed within a top-level simulation loop and connected to the LabVIEW FPGA function to simulate the pulsed voltage signal used to drive the motor. The result is a high fidelity closed-loop simulation of

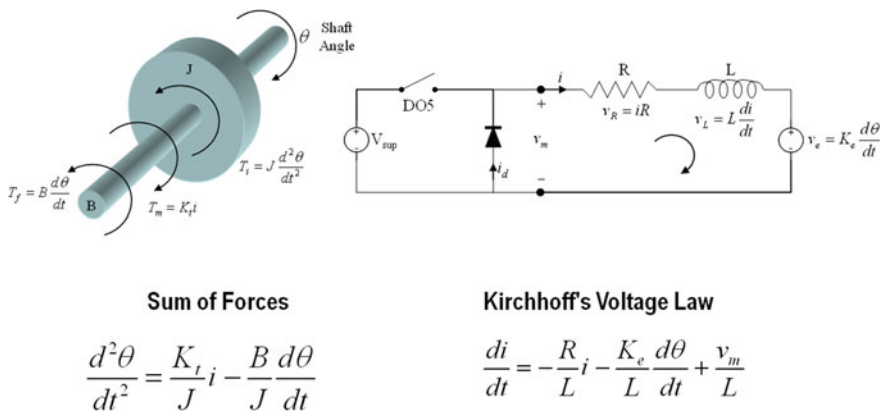


Fig. 2.69 DC drive motor model

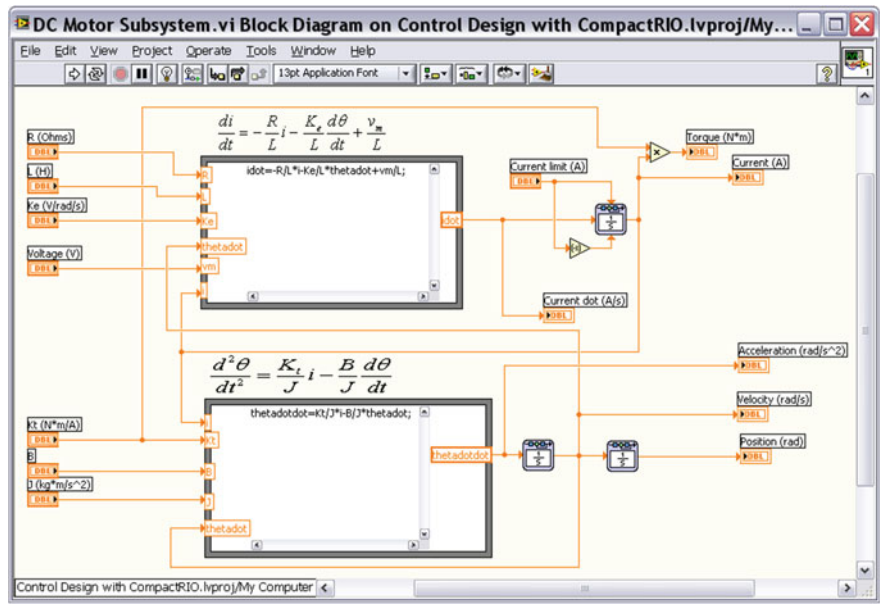


Fig. 2.70 DC drive model

how the LabVIEW FPGA code will behave when connected to the real-world electromechanical system (see Fig. 2.71).

The simulation results have been validated against actual measurements from the deployed LabVIEW FPGA application controlling a motor using the NI 9505 motor drive module, which showed a nearly identical match between the simulated and measured waveforms.

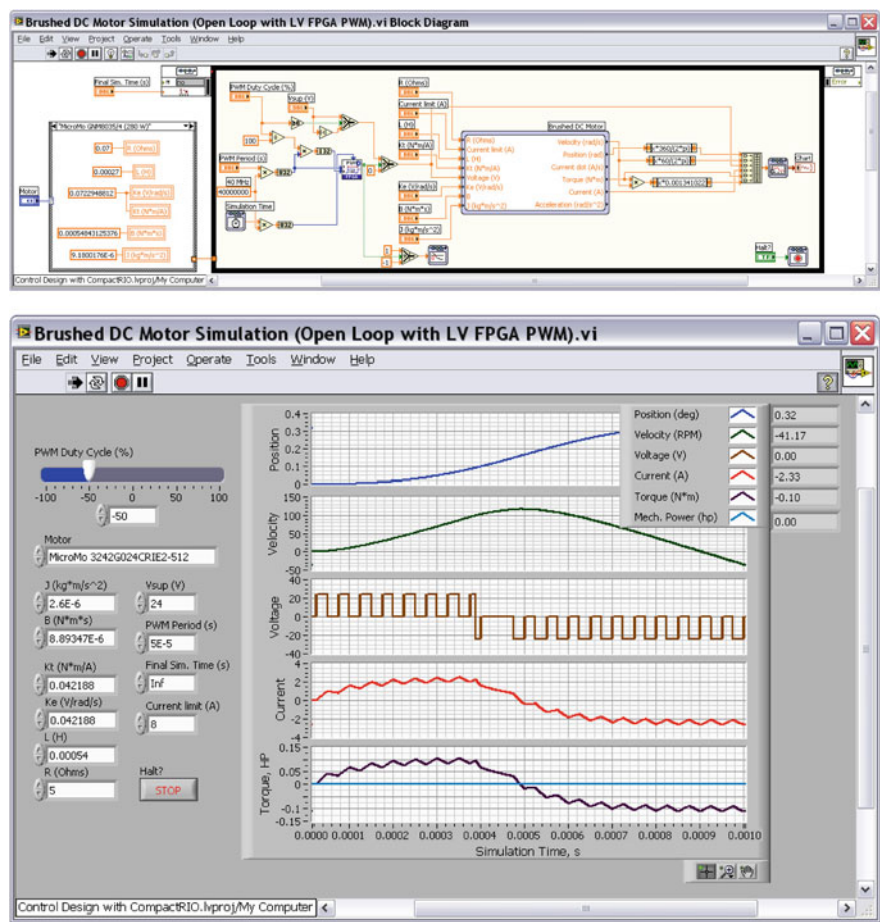


Fig. 2.71 Brushless motor drive

This approach lets you take code validation way beyond basic functional validation. Think of this like a virtual machine emulator that lets you anticipate how your code will perform in the real world. You can use simulation to help make design decisions, evaluate performance, select components, and test worst-case conditions. You can even tune the PID control loops for your control system in simulation and see how well that tuning works with different motors and load conditions. Simulation can also help you select the right physical components for your system, such as picking the right motor to meet your performance requirements.

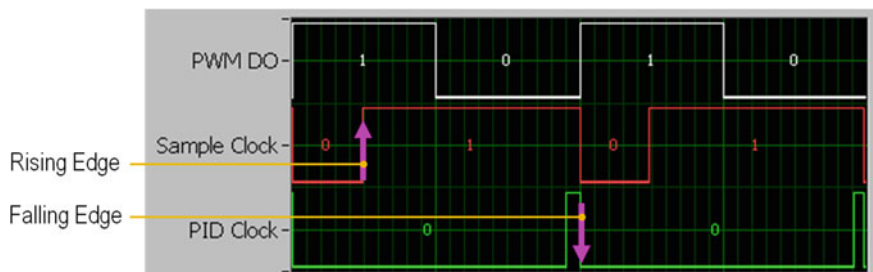


Fig. 2.72 Trigger on either the rising or falling edge

2.6 Synchronize Your Loops

Now for our fourth development technique—how to control the timing and synchronization of your LabVIEW FPGA code.

For most control applications, the timing of when the code executes is very important to the performance and reliability of the system. Fortunately, LabVIEW FPGA gives you both unprecedented speed and complete control over the timing of the code. Unlike a processor, an FPGA executes code in a truly parallel fashion rather than only being able to execute one instruction at a time. That makes programming easier because you do not have to worry about setting priorities and sharing the processor time among the different tasks. Each control loop is like a custom designed processor that is completely dedicated to its task. The result is high reliability and high-performance code. One of the benefits of this performance is that control loops are typically more stable, easier to tune, and more responsive to disturbance when they run at a fast rate.

In this motor control example, we have two different clock signals—a Sample Clock and a PID Clock. These are Boolean signals we generate in the application to provide synchronization among the loops. We can trigger on either the rising or falling edge of these clock signals (see Fig. 2.72).

Now let us take a look at the LabVIEW FPGA code used to monitor these signals and trigger on either the rising or falling edge.

Typically triggering a loop based on a Boolean clock signal works like this—first wait for the rising or falling edge to occur, and then execute the LabVIEW FPGA code that you want to run when the trigger condition occurs. A sequence structure is often used where the first frame of the sequence is used to wait for the trigger, and the second frame is used to execute the triggered code, as shown below.

Rising Edge Trigger: In this case we are looking for the trigger signal to transition from False (or 0) to True (or 1). This is done by holding the value in a shift register and using the **Greater Than?** Function (see Fig. 2.73). (Note: A True constant is wired to the iteration terminal to initialize the value and avoid an early trigger on the first iteration.)

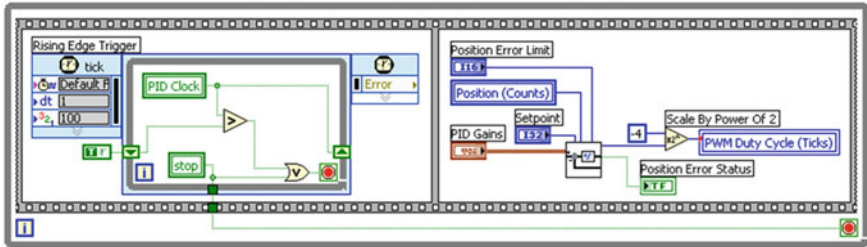


Fig. 2.73 Rising edge trigger

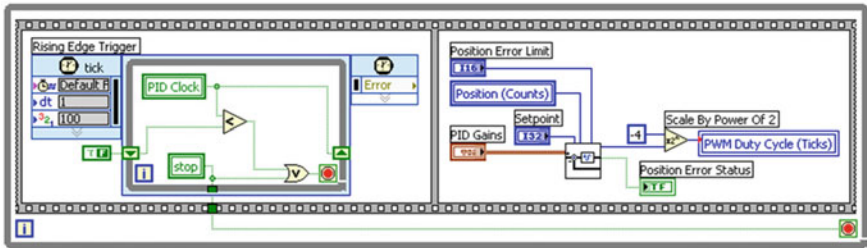


Fig. 2.74 Falling edge trigger

Falling Edge Trigger: In this case we use a **Less Than?** function to detect the transition from True (or 1) to False (or 0) (see Fig. 2.74). (Note: A False constant is wired to the iteration terminal to initialize the value.)

Analog Level Trigger: Here we use a **Greater Than?** function to detect when the analog signal is greater than our analog threshold level, and then use the Boolean output of the function as our trigger signal (see Fig. 2.75). This case actually a rising **or** falling edge detector since we are using the Not Equal? function to detect any transition.

Now let us take a look at another common triggering use case—this is where we want to latch the value of a signal when a trigger event occurs.

2.6.1 Latching Values

In this case we use a rising edge trigger to latch the **Analog Input** (see Fig. 2.76) value from another loop into the **Latched Analog Input** register. This value is held constant until the next trigger event occurs. In this example, the actual analog input operation is occurring in another loop and we are using a local variable for

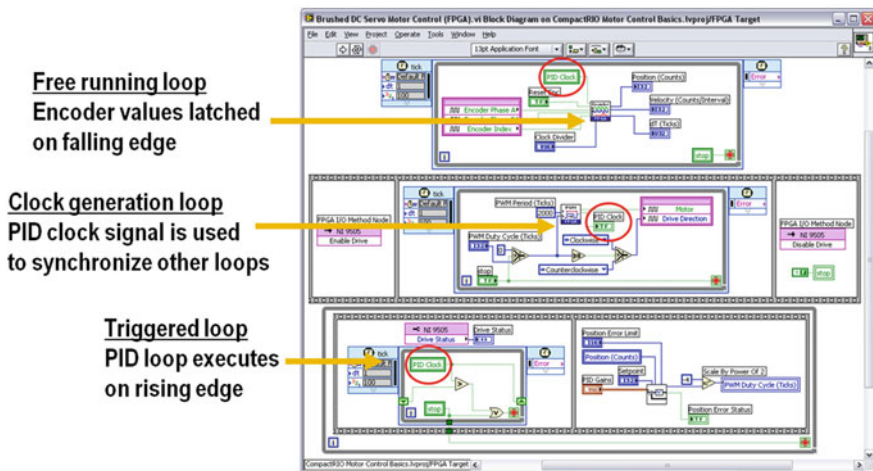
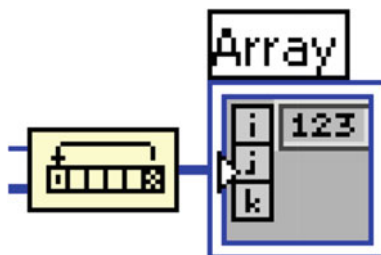


Fig. 2.77 Triggering and latching techniques

Observations:

- One loop is used to generate synchronization clocks used by other loops.
- The encoder function needs to run at full speed to avoid missing any digital pulses. This function runs at 40 MHz but latches the Position (Counts) and Velocity (Counts/Interval) signals to synchronize the data with the other loops.
- The PID function needs to run at a specific speed (20 kHz or 2000 ticks) and avoid any jitter in its timing. This is because the integral and derivate gains depend on the time interval, T_s . If the time interval was varying, or if the same old value was passed multiple times into the function, the integral and derivative gains would be incorrect.
- In the bottom loop, you can see that the execution is triggered by the rising edge of the PID clock signal. We read a local variable for the signal in this SCTL, and exit the loop when a rising edge is detected. Then we execute the 32-bit PID algorithm that is included with the NI SoftMotion Development module. This reads the commanded position, compares it to the position measured by the encoder, and then generates a command for the PWM loop. In this case, we are using a **Scale by Power of 2** function to divide the PID output signal by 2^{-4} , which is equivalent to dividing by 16. This scales the value to the ± 2000 ticks range needed by the PWM function. A value of 1000 ticks is equal to a 50 % duty cycle since the PWM period is 2000 ticks.
- Note that the upper two loops are running at a 40 MHz loop rate, where the lower loop is triggered to run at a 20 kHz loop rate by the PWM clock signal. (When triggered, the SoftMotion PID function takes 36 ticks to execute.)

Fig. 2.78 Rotate 1D array function



2.7 Technique 5: Avoid “Gate Hogs”

Now that you understand four key techniques that are useful for developing LabVIEW FPGA code, let us talk about one last technique—how to avoid “gate hogs.” These are often “innocent looking” code that eats up lots of your FPGA gates (also known as slices). Here are three of the most common offenders.

Large Arrays or Clusters: Creating a large array or cluster with a front panel indicator or control is one of the most common programming mistakes that eat up lots of FPGA gates. If you do not need a front panel indicator for communication with the host processor, then do not create one. If you need to transfer more than a dozen or so array elements, use DMA instead as a way to pass the data. Also, avoid using array manipulation functions like this **Rotate 1D Array** function whenever possible (see Fig. 2.78).

Quotient and Remainder: This function does integer division. (The quotient output, $\text{floor}(x/y)$, is x divided by y , rounded down to the closest integer. The remainder output, $x - y * \text{floor}(x/y)$, is whatever is left over. For example, 23 divide by 5 gives a quotient of 4 and a remainder of 3.) This function is gate intensive and takes multiple clock cycles to execute so it cannot be used in a SCTL. Be sure to wire up the minimum data type needed to the terminals when using this function and use constants rather than controls when possible (see Fig. 2.79).

Scale By Power of 2: If the n terminal is positive, this function multiplies the x input by 2 to the power of n (2^n). If n is negative, the function divides by 2^n . For example, setting n to $+4$ would multiply by 16, while setting it to -4 would divide by sixteen. This function is much more efficient than the **Quotient and Remainder** function. However, use a constant of the minimum data size needed for the n terminal whenever possible (see Fig. 2.80).

Note: DMA is a better way to send an array of data to the host than creating a front panel indicator for the array and using the **FPGA Read/Write** method. Arrays



Fig. 2.79 Quotient and remainder

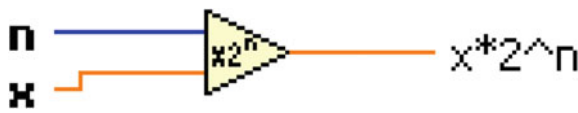
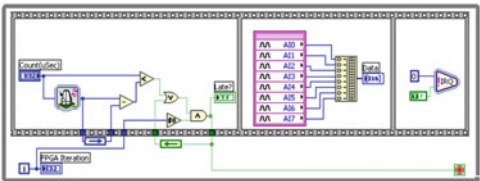


Fig. 2.80 Scale by power of 2

FPGA



Host

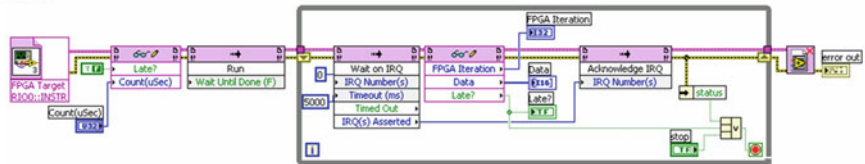


Fig. 2.81 Optimized FPGA code

are useful for collecting a set of simultaneously sampled data to be fed into a DMA buffer for transfer to the host computer. It is okay to use an array to collect the data points together for indexing into the DMA Write function as long as you do not create a front panel indicator for the array. Using auto-indexing on the for loop used to write the data into the DMA, buffer is fine as long as you do not create a front panel indicator for the array because the compiler does a good job of optimizing arrays passed into **For Loops** for indexing purposes.

2.7.1 Avoid Front Panel Arrays for Data Transfer

When optimizing your code for the amount of space it uses on the FPGA, you should consider the front panel controls and indicators you are using. Each front panel object and the data it represents take up a significant portion of the FPGA space. By reducing the number of these objects and reducing the size of any arrays used on the front panel, you can significantly reduce the FPGA space required by the VI (see Fig. 2.81).

Instead of creating large arrays to store data and transfer it to the host application (shown above), use DMA to transfer an array of analog input samples to the host processor as shown in Fig. 2.82.

Here are some programming instructions for implementing DMA:

- When setting the FPGA buffer size, you can use the default size (1023). Creating a larger FPGA memory buffer typically does **not** have benefits.
- You should set the host buffer size to a large value than the default size. By default, the host buffer size is 2 times bigger than the FPGA buffer. You should actually set it to at least two times the **Number of Elements** you plan to use.
- If you are passing an array of data, the **Number of Elements** input should always be an integer multiple of the array size. For example, if you are passing an array of 8 elements, the **Number of Elements** should be an integer multiple of 8 (such as 80, which would give 10 samples of 8 elements each.)
- Each DMA transaction has overhead, so reading larger blocks of data is typically better. The **DMA FIFO.Read function** automatically waits until the **Number of Elements** you requested become available, minimizing processor usage.
- Packing 16-bit channel data into a U32 (since DMA uses U32 data type) typically does **not** have benefits on CompactRIO, because the PCI bus has very high bandwidth for sending DMA data, so you most likely are nowhere near to using up all of the bus bandwidth. Instead, it is typically the processor that is the bottleneck in processing the data being streamed. Packing the data in the FPGA means it has to be unpacked on the processor, adding additional processor overhead. In general, you should send each channel as a U32 even if you are acquiring 16-bit data.
- The **Full** output on the DMA FIFO Write function is actually an error indicator. Under normal operation this should never occur so it is recommended that you stop the application if this error occurs and reset the FPGA before restarting.

2.7.3 *Use the Minimum Data Type Necessary*

Remember to use the minimum data type necessary when programming in LabVIEW FPGA. For example, using an 32-bit integer (I32) to index a Case Structure is probably overkill since it is unlikely that you will be writing code for **2 billion different cases**. Usually, an unsigned 8-bit integer (U8) does the trick, since it works for up to 256 different cases (see Fig. 2.84).

2.7.4 *Optimizing for Size*

The FPGA application shown in Fig. 2.85 is too large to compile, because it uses an array to store sine data.

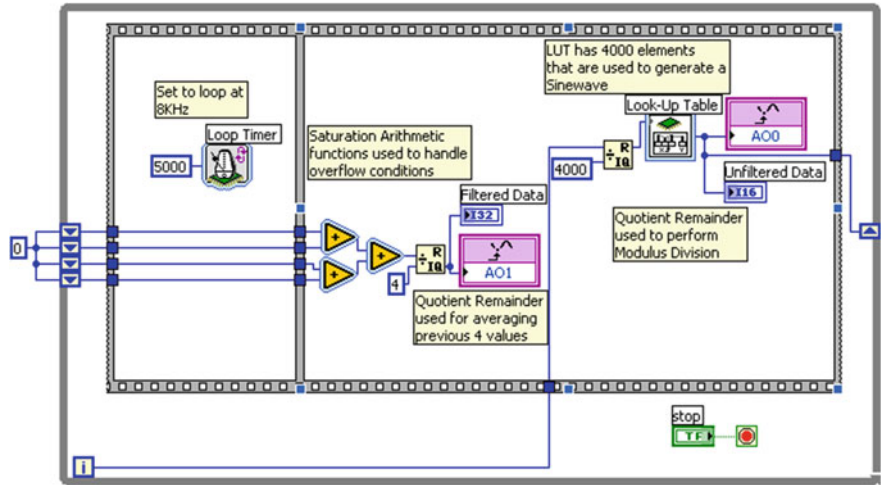


Fig. 2.86 Look-up table to store sine data

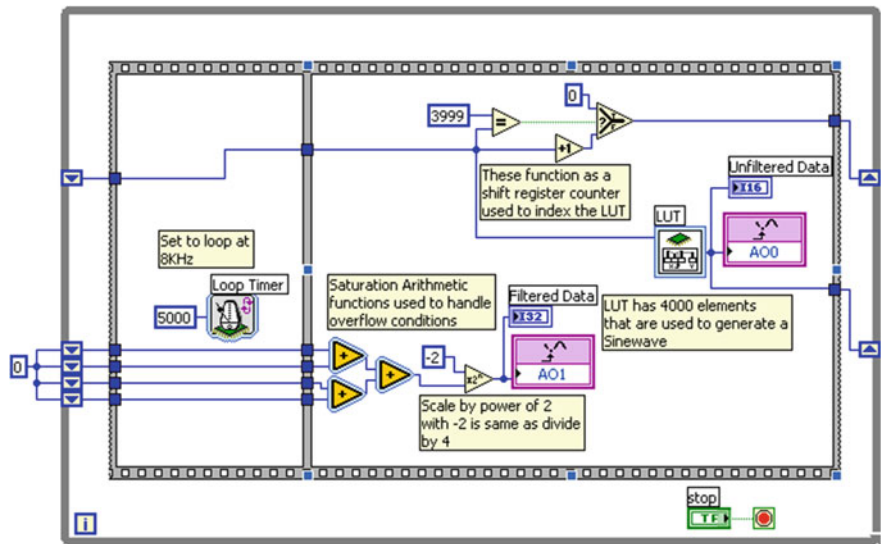


Fig. 2.87 Removed both quotient remainder functions

common technique in FPGA. The other QR function was replaced by a scale by 2 to the power of n. Because the scale by 2 has a constant input, it uses very little FPGA space (see Fig. 2.87). Note: Scale by 2^{-2} is equal to dividing by 4.

Now the application takes only 9 % of the FPGA gates.

Table 2.4 Optimization methods for LabVIEW FPGA

Optimization technique	FPGA speed	FPGA size
Reduce <i>combinatorial paths</i>	✓	
Use <i>pipelining</i> when appropriate	✓	
Use <i>single-cycle timed loops</i>	✓	✓
Use <i>parallel operations</i>	✓	
Select appropriate <i>arbitration options</i>	✓	✓
Use <i>non-reentrant subVIs</i>		✓
Use <i>reentrant subVIs</i>	✓	
Limit the number of <i>front panel objects</i> , such as arrays		✓
Use the <i>smallest data type</i> possible	✓	✓
Avoid <i>large VIs and functions</i> , if possible	✓	✓
Schedule timing <i>using handshaking signals</i>	✓	✓

2.7.5 Additional Techniques to Optimize Your FPGA Applications

For more information on this topic, see the “Optimizing FPGA VIs for Speed and Size” topic on the NI Developer Zone [10]. The document contains detailed information on more than ten techniques you can use to optimize your LabVIEW FPGA applications. Table 2.4 shows some optimization methods for LabVIEW FPGA that can be implemented in your code.

References

1. <http://www.ni.com/>
2. <http://sine.ni.com/nips/cds/view/p/lang/en/nid/210566>
3. <http://www.ni.com/example/7781/en/>
4. <http://www.ni.com/ipnet/>
5. <http://www.ni.com/fpga-hardware/whatsnew/>
6. <http://sine.ni.com/nips/cds/view/p/lang/en/nid/11766>
7. <http://sine.ni.com/nips/cds/view/p/lang/en/nid/11834>
8. <http://www.ni.com/compactrio>
9. <http://zone.ni.com/devzone/cda/tut/p/id/7781>
10. http://zone.ni.com/reference/en-XX/help/371599D-01/vfpgaconcepts/optimizing_fpga_vis/

Fuzzy Logic Type 1 and Type 2 Based on LabVIEW™
FPGA

Ponce-Cruz, P.; Molina, A.; MacCleery, B.

2016, XVI, 233 p. 203 illus., 52 illus. in color. With online
files/update., Hardcover

ISBN: 978-3-319-26655-8