

## Exercise 2.1: Creating a LabVIEW FPGA VI

---

### Exercise Objective

This exercise will introduce you to basic concepts and methods of programming LabVIEW FPGA. You will create an application from scratch that shows many of the basic features of LabVIEW FPGA and R Series. This application will also show the parallelism of a LabVIEW FPGA system.

### Hands-On Summary

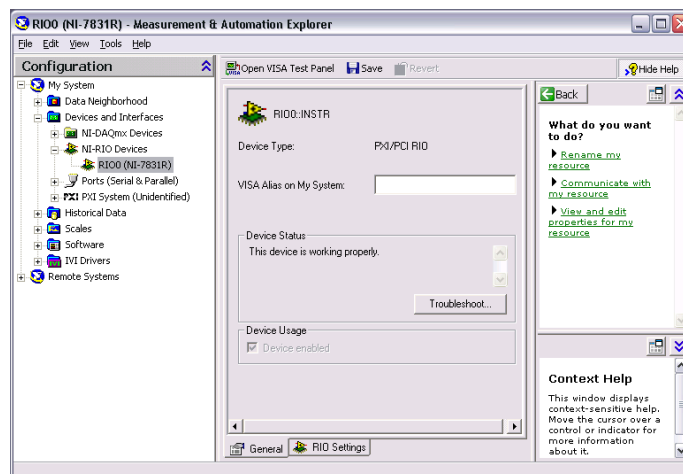
- Introduce LabVIEW FPGA
- Learn how to setup and configure a R Series device using the LabVIEW Project
- Create an FPGA Application that adds two numbers and benchmarks the execution.

### Hardware Setup

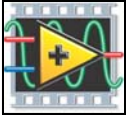
#### Part 0: Configuring the R Series Device in MAX

In this section you will learn the following:

- Discovering a R Series in Measurement and Automation Explorer (MAX)



1. Confirm that your RIO device shows up in MAX. Since we will be running the R Series on the windows machine, no further configuration is required to access the device from LabVIEW.

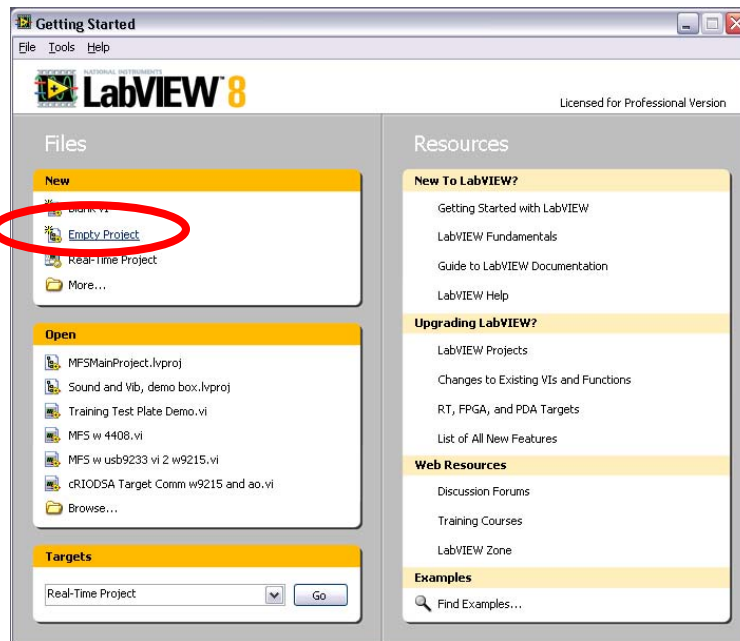



## Part 1: Developing the FPGA Application

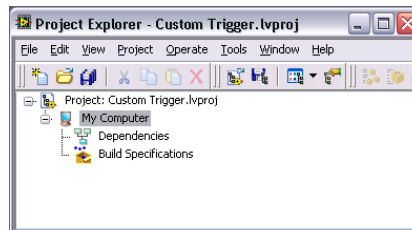
In this section you will learn:

- Develop, target, download, and run an application on an FPGA.

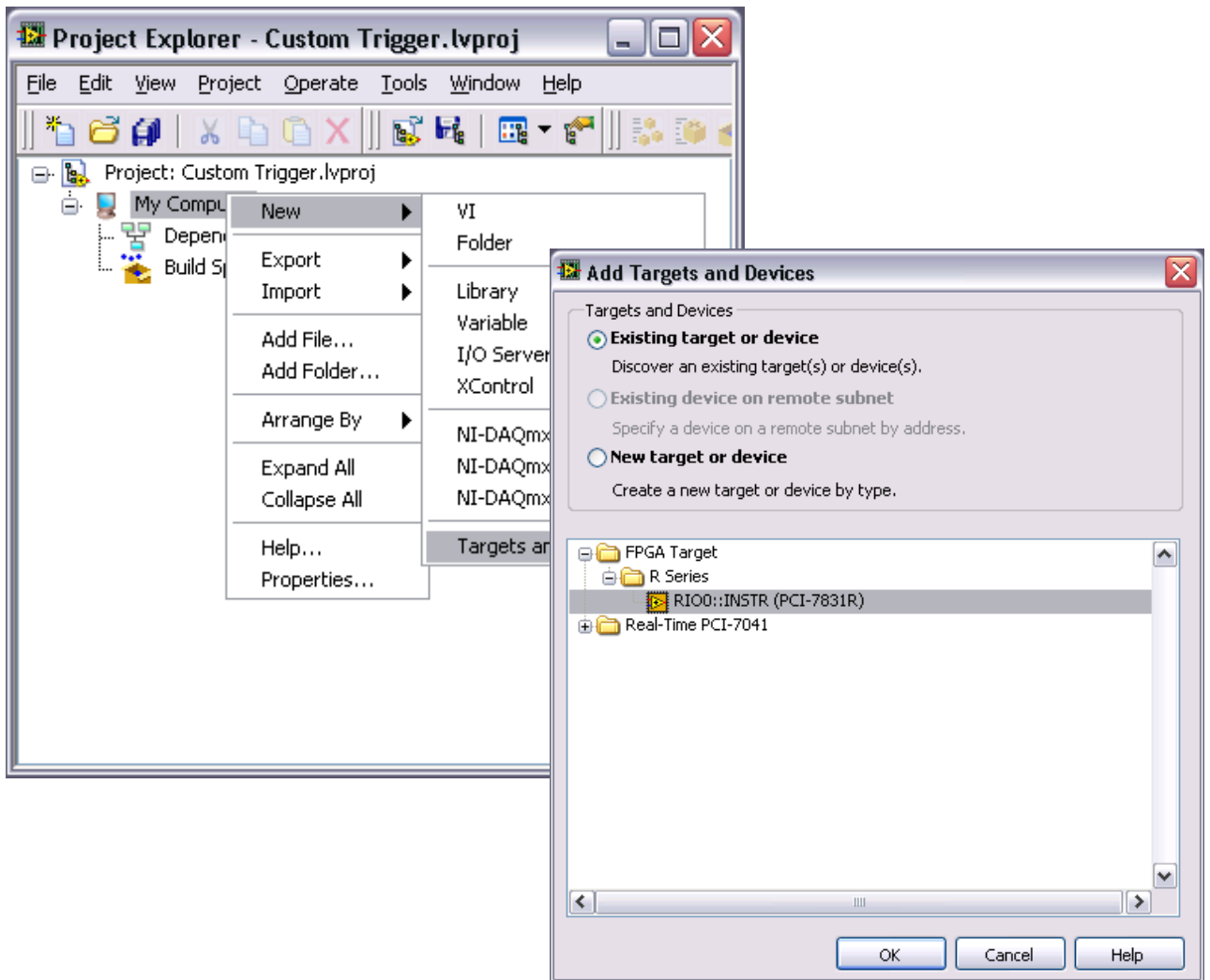
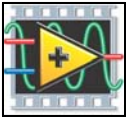
1. Launch **LabVIEW 8** and open an Empty Project.



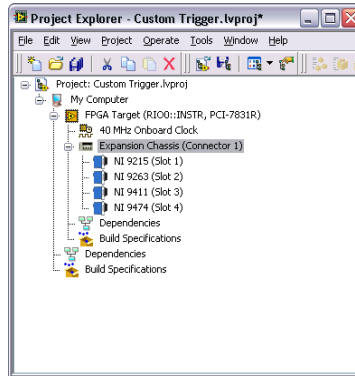
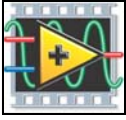
2. Click the **Save All** button  and save your project as Custom Trigger to the “Intro Demo” folder located on the desktop. Your project should look like this:



3. **Right-click on My Computer** and select **New>>Targets and Devices**. This will allow us to add an FPGA target to our project. Expand the FPGA Target folder, highlight the PCI-7831R device and select OK.



4. The LabVIEW Project should look like this:



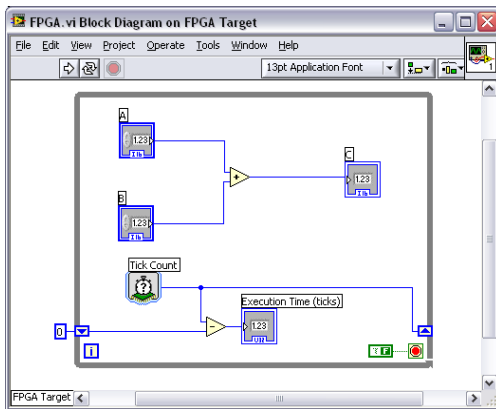
Replace this with new figure

## 5. Developing the LabVIEW FPGA Application

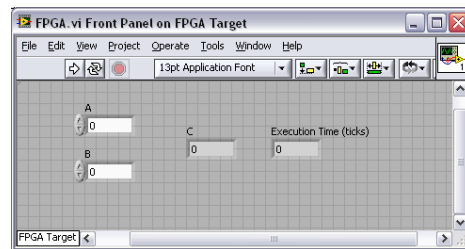
In this section you will:

- Explore the function palettes while targeted for FPGA.
- Design an application that adds two numbers and benchmarks the code

The completed application should look like this:

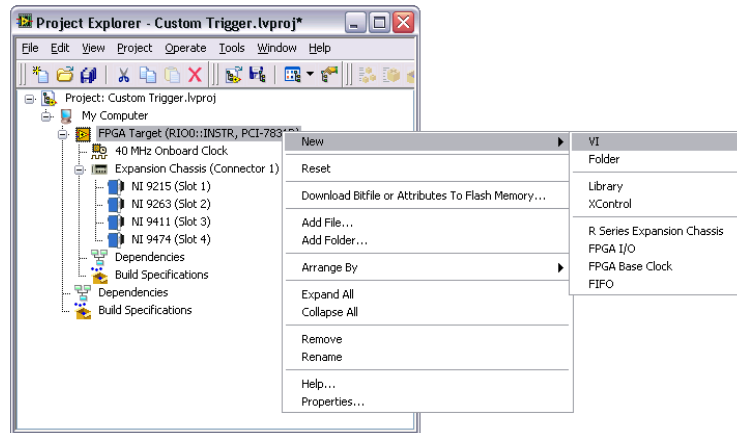
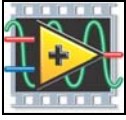


Block Diagram



Front Panel

6. In the LabVIEW Project Window, right-click on the FPGA target (PCI-7831R) and select New>>VI to start a new LabVIEW FPGA VI (**LabVIEW programs are called Virtual Instruments or VI's**). Save this VI as **AddFPGA.vi** by (File>>Save As) to signify that this program will be run on the FPGA.

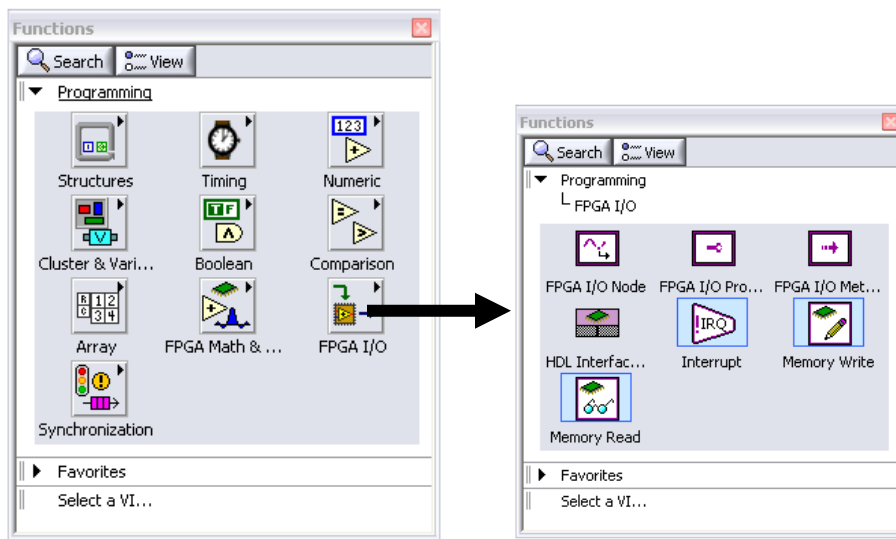


New VI

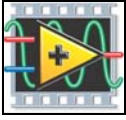
7. When the LabVIEW front panel appears, go to the block diagram by selecting **Window>>Show block diagram**.
8. Right-click in the white area on the block diagram to display the **Functions** palette. Click on the thumb tack icon in the top left corner of the **Functions** palette to tack it down.

FunctionsPalette here

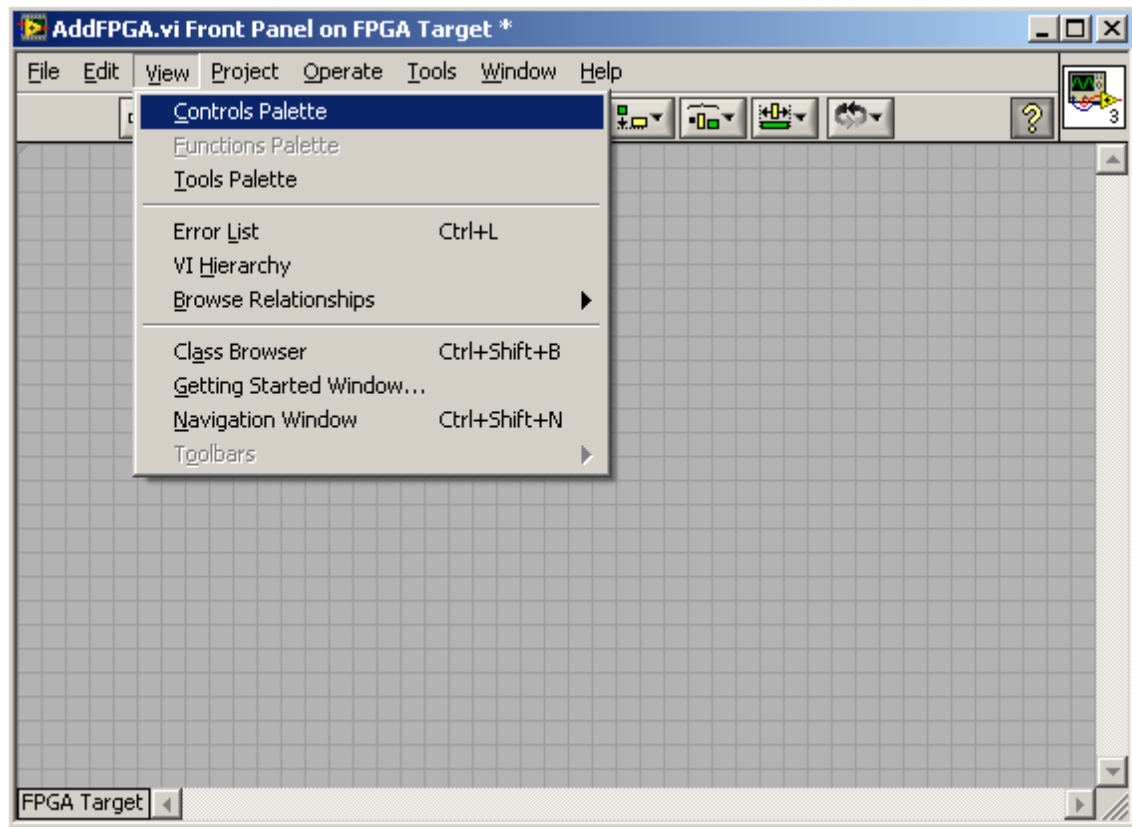
9. Navigate to the **Help** menu and select **Show context help**. Then browse through the **Functions** palette to familiarize yourself with the many math, digital logic, comparison, input/output, math and analysis functions and IP libraries that are available.

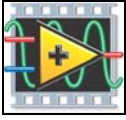


Browsing the LabVIEW FPGA functions palette

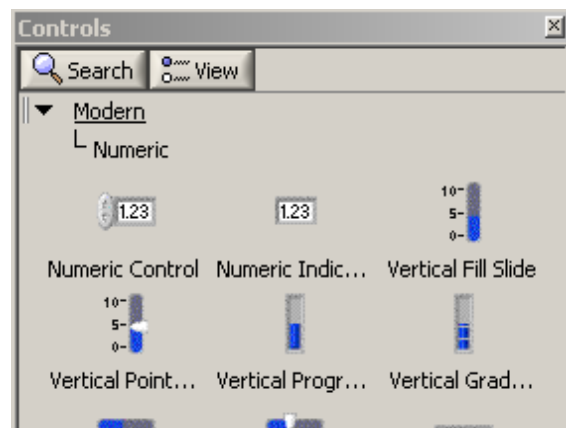


10. Go back to the LabVIEW front panel (gray screen), go to View >> Controls Palette. This will turn the controls Palette ON

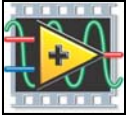




11. From Controls Palette press “Numeric” button which will invoke Numeric Palette. Now place two numeric controls and two numeric indicator in the front panel. You can always go back a level by clicking the upper level in the top left corner (Modern).



12. In “Front Panel” rename the controls **A** and **B**, and the indicators as **C** and **Execution Time (ticks)**. It is easier to name the component while placing. To rename right click in the middle of the controls and then click “Properties” and change the name in properties (it is faster to Delete and place them again and name while placing). The Front Panel and Block Diagrams should look like this now.

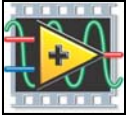


The image displays two windows from the LabVIEW FPGA environment, both titled "AddFPGA.vi on FPGA Target \*".

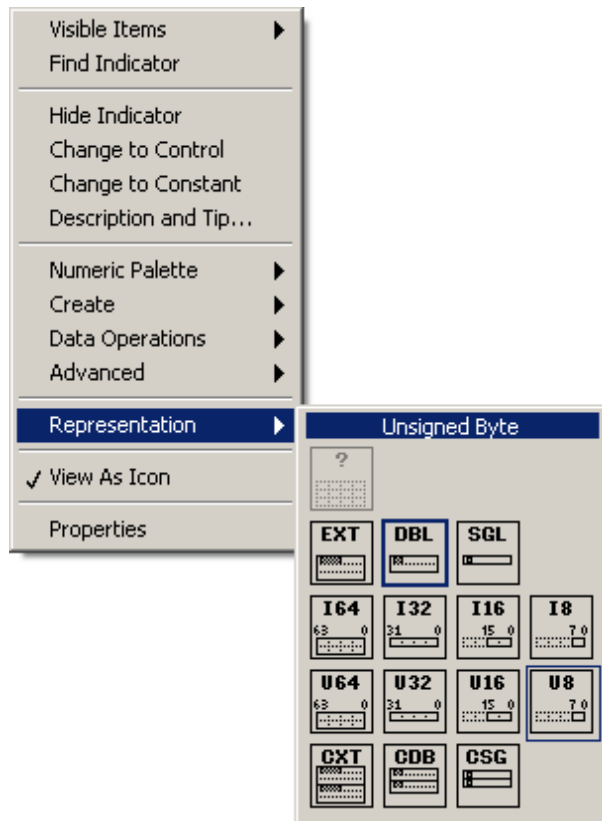
The top window is the "Front Panel". It features a menu bar with "File", "Edit", "View", "Project", "Operate", "Tools", "Window", and "Help". Below the menu bar is a toolbar with icons for navigation and execution. The main area is a grid with four numeric display indicators: "A" (value 0), "B" (value 0), "C" (value 0), and "Execution Time (ticks)" (value 0). A status bar at the bottom shows "FPGA Target".

The bottom window is the "Block Diagram". It also has a menu bar and toolbar. The main area contains four numeric display indicators, each with a value of 1.23. The indicators are labeled "A", "B", "C", and "Execution Time (ticks)". Each indicator is highlighted with a blue border, indicating they are selected or active. A status bar at the bottom shows "FPGA Target".

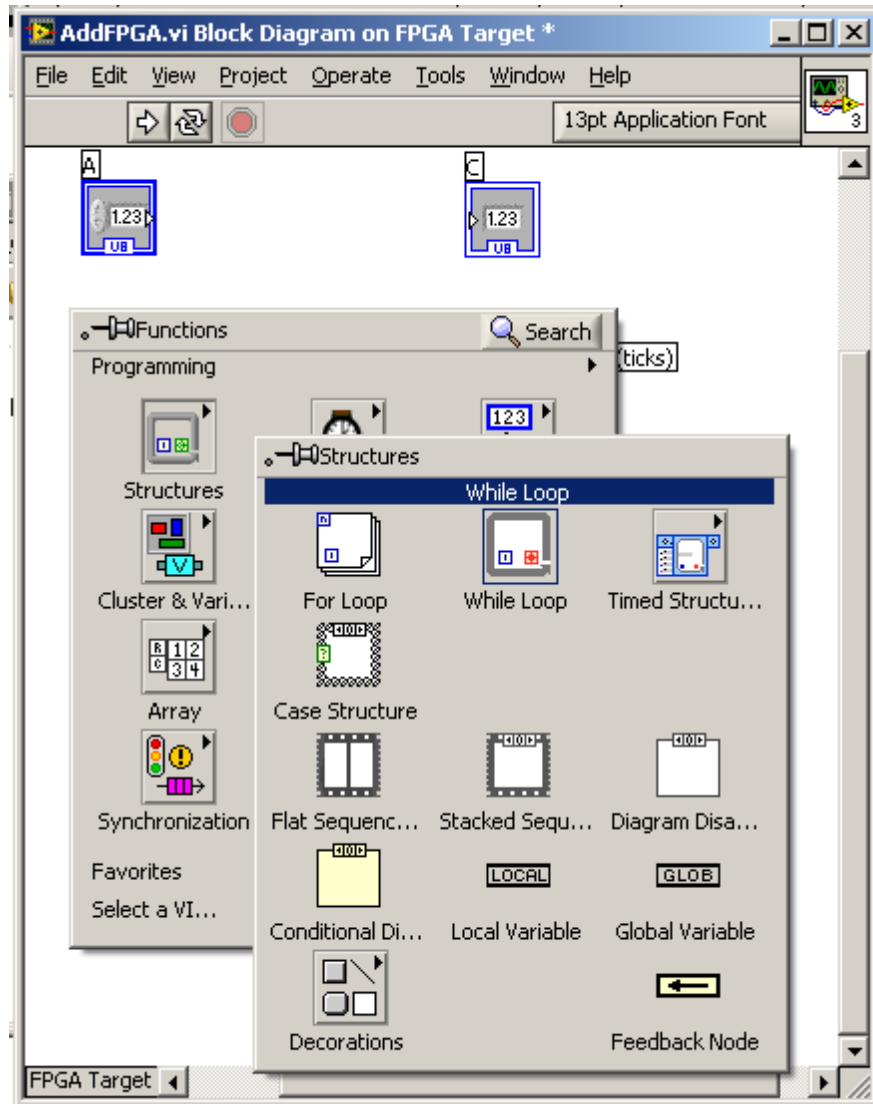
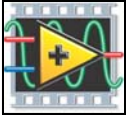




13. Change their data type to be an Unsigned Byte integer, by right-clicking on each of them and selecting **Representation** » **Unsigned Byte** from the shortcut menu. Notice U8 in at the bottom of each icon in Block Diagram (it was I16 initially).

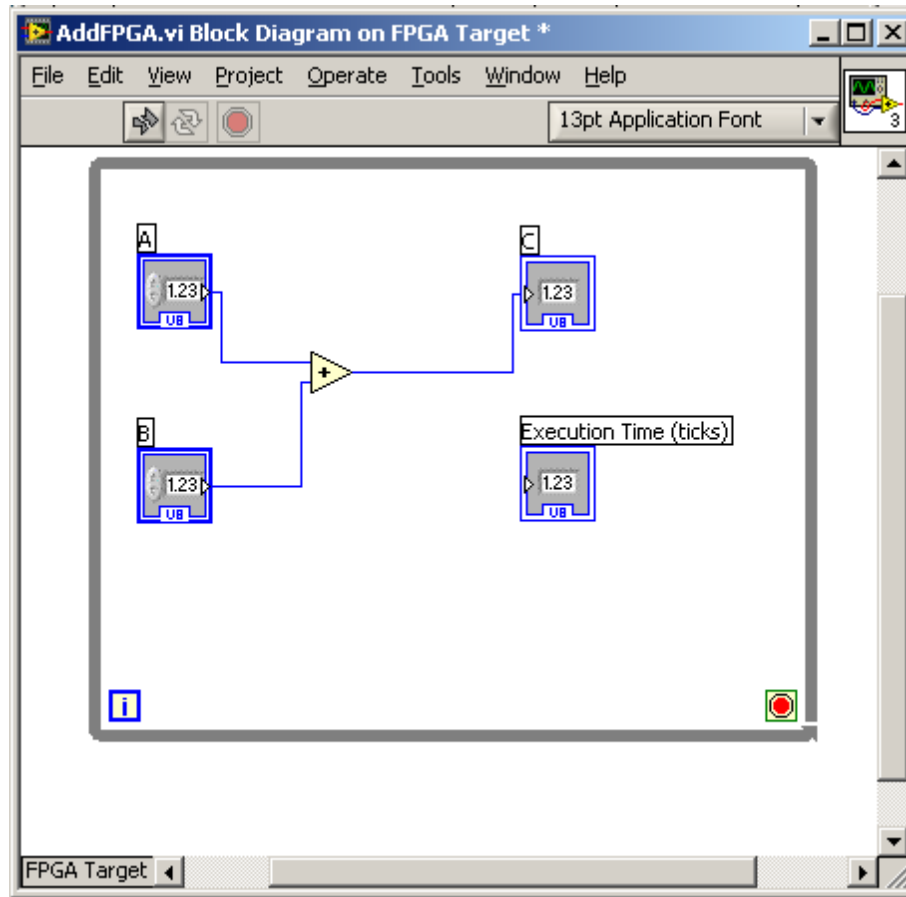
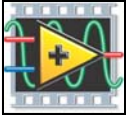


14. In the block diagram, add a while loop around the four controls. This is done by right-clicking and then in Functions >> Structures >> While Loop (as shown in screen shot below), make sure the terminals for the controls and indicator are inside the While loop.



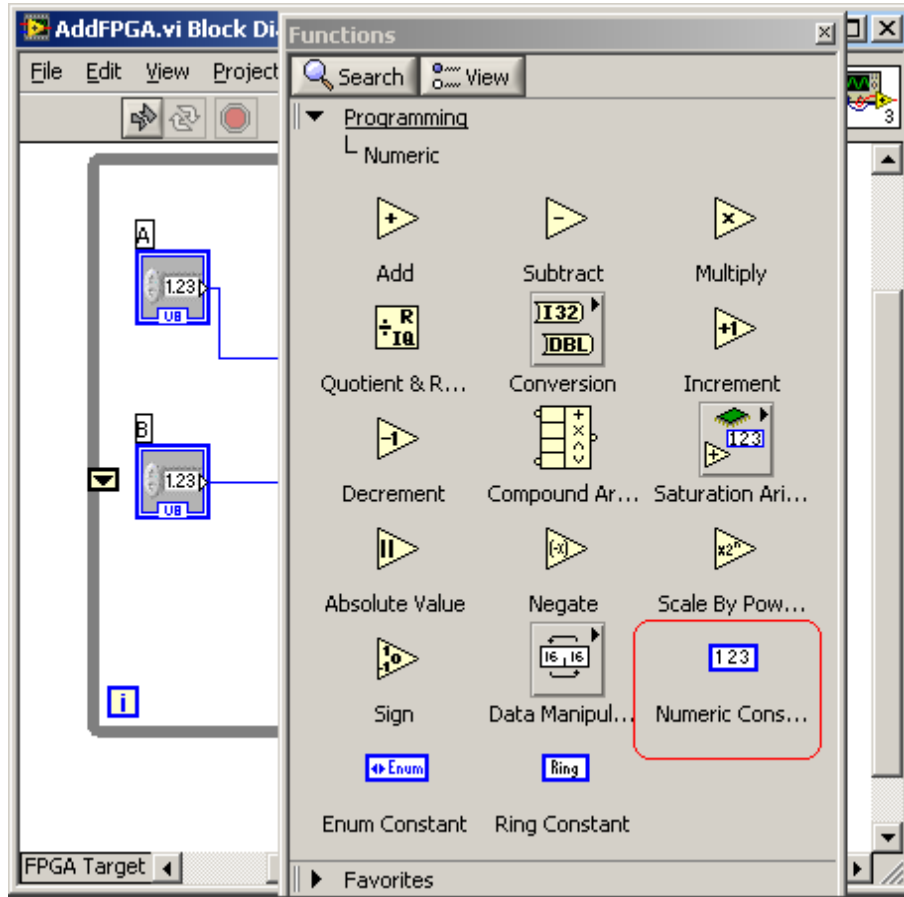
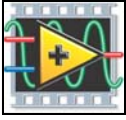
15. Place the **Add** function (Functions>>Numeric) inside the while loop.
16. If you do not have the Tools Palette visible, turn ON Tools Palette from View>>Tools Palette. In Tools click on “Wire Reel” button and wire **A** and **B** to the inputs of the add function and wire **C** to the add function output. Wiring may feel tricky and may require some practice.



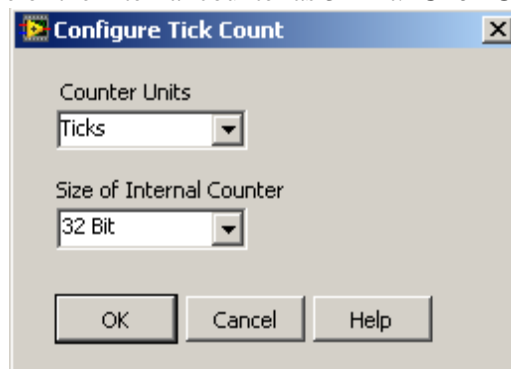


*A common use of parallelism in LabVIEW FPGA is for benchmarking purposes. Since LabVIEW FPGA implements independent tasks in parallel, we can easily add benchmarking code in parallel to our main application without effecting the performance.*

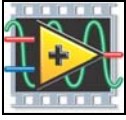
17. Right click on the left edge of the While Loop and select **Add Shift Register**.
18. Initialize the shift register by dropping down a numeric constant of zero from the Numeric Palette (which is accessible from the Functions Palette **Functions>>Numeric>>Numeric Constant**).



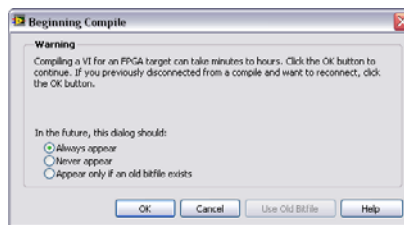
19. Now, drop down a **Tick Count** function (**Functions>>Timing**), select the counter units to be **Ticks** and keep the size of the internal counter as 32 Bit. **Click OK.**



Configure Tick Count Dialog



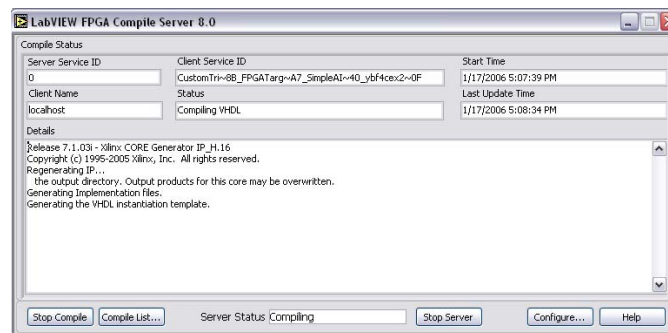
20. Wire the output of the Tick Count function to the shift register on the right side of the While Loop
21. Drop down a **Subtract** function on to the block diagram.
22. Wire the output of the shift register on the left side of the While Loop and the value of the **Tick Count** function to one inputs of the Subtract function. Wire the output of the Subtract function to the indicator labeled **Execution Time (ticks)**.
23. **Save the VI and project.** Click the **Run** button to start the compile process. The following window will appear:



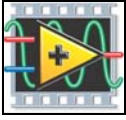
**Figure 20.** FPGA Build Warning

This is just telling you that compiling the FPGA code can take a long time. Click **OK** and the compiling process will begin. After the build process completes, you may run the VI in FPGA interactive mode. In this mode, the embedded VISA server passes data from the FPGA over the PCI bus to your PC. The update rate in interactive mode is typically limited to about 10 S/s.

While the application is compiling, ask one of the instructors if you have any questions or comments. You've worked hard—sit back and enjoy 5-10 minutes of well deserved rest and relaxation while your LabVIEW FPGA application compiles! If desired, you may read the comments below that explain the compilation process and other topics that may be of interest.



**Figure 21.** Snapshot during FPGA compilation



## ***Understand the LabVIEW FPGA Compilation Process***

*The LabVIEW FPGA Module 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. 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.*

*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. Later we will build a real-time host interface to the application that enables high speed data transfer and interrupt synchronization between the floating-point host processor and integer-based FPGA chipset.*

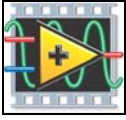
*NOTE: If you disconnect during the compilation, you will need to run the application once while targeted to the FPGA in order to load the bitstream into the FPGA VI file from the compile server.*

## ***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 nanoseconds. By changing the compile options, you can increase the FPGA clock speed up to 200 MHz (5 nanoseconds). 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 (<http://www.ni.com/compactrio>) or click the **Help** button on the **Target>Build Options** menu.*

## ***Understanding the Compilation Report***

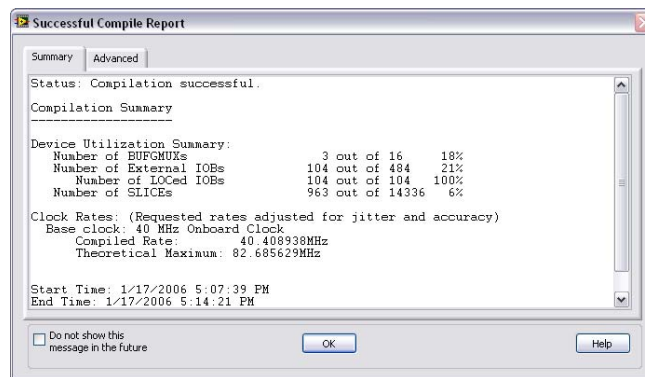
*A SLICE is a collection of logic components on the FPGA. The percentage shown is the percentage of the FPGA used. As the FPGA reaches greater than 90% usage, the compiler performs more optimization to make the most efficient use of resources. For simple applications, the compiler does not “try” very hard to optimize, as it does not need to make efficient use of FPGA resources. For this reason, you may be able to fit more onto the FPGA than this report would lead you to believe.*



## Debugging your FPGA Algorithms before Compilation

Because LabVIEW FPGA uses the same high level LabVIEW source code that is used on all LabVIEW targets, you can execute the same code on any processor in a functionally equivalent manner. This makes LabVIEW a powerful universal programming language for rapid development of embedded systems. Although the execution timing will be different when implemented on the FPGA, the control logic and integer math functionality is the same regardless of how the code is targeted. When running LabVIEW FPGA code in Windows, you can use breakpoints, probes, highlight execution mode and other standard debugging tools to validate the FPGA code. While working in Windows, you can enable the LabVIEW FPGA palette view by clicking the options button at the top-right corner of your functions palette and selecting **FPGA Hardware** in the **Palette View** menu.

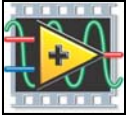
When the compiler is complete, the compile report will be generated. This report shows the start and end compilation time, the number of SLICES used, a compiled clock rate (40 MHz), and an estimated maximum clock rate.



**Figure 22.** Build 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 than this report would lead you to believe. For simple applications, the compiler does not “try” very hard to optimize, as it does not need to make efficient use of FPGA resources. The compiler will begin to optimize the compiled code when 95% of the FPGA resources are used.

24. Click **OK** to close the build report. The FPGA VI will begin to run automatically. Adjust the analog output (AO 0) voltage and observe the reaction of the system.
25. Click **Stop Server** to close the compile server.
26. Run the VI and watch how it adds the two numbers. You should also notice that the code takes 6 ticks (or 150 nanoseconds) to execute. Remember we selected an Unsigned Byte representation for the controls and indicators. Therefore each control has a range 0-255. Since the indicator also has the same 8 bit range notice what occurs when 2 large numbers are



added together. For example,  $240+120$  returns 104 as a result. This is due to overflow and occurs when the result does not have enough bits to store the result. There are special nodes found in the Functions>>Numeric>>Saturation Arithmetic palette that can be used to handle these types of situations. We will discuss these a little later in the lesson.

27. After you have run and tested the code close the application and the Project Manager.

### **ALL STAR (OPTIONAL)**

28. Now open the precompiled example found in the project named Exercise 21.pjt in the Solutions folder. The VI you should open is named Ex 2 1 optional.vi.

29. Looking at the block diagram you will notice that we added a subtraction task to the VI. Run this VI and you will notice that the execution time is still 6 ticks. The execution time remains the same because the addition and subtraction tasks are performed in parallel.