A GRAPHICAL APPROACH TO FPGA PROGRAMMING

Christian Amadasun (National Instruments, Austin, Texas, United States;
christian.amadasun@ni.com)

ABSTRACT

Programming of Field Programmable Gate Arrays (FPGAs) has long been the domain of engineers with VHDL or Verilog expertise. FPGA's have moved from being simple glue logic chips to replacing application specific integrated circuits (ASICs) and processors for signal processing and control applications. FPGA's have caught the attention of algorithm developers and communication researchers, who want to use FPGA's to instantiate systems or implement DSP algorithms. These efforts however, are often stifled by the complexities of programming FPGA's. RTL programming in either VHDL or Verilog is generally not a high enough level of abstraction needed to represent the world of signal flow graphs and complex signal processing algorithms. This paper describes the features of the LabVIEW FPGA environment and how graphical programming makes for a robust FPGA programming platform. This paper assumes a general familiarity with FPGA's and LabVIEW or other programming languages such as C, C++ or Java.

1. INTRODUCTION

NI LabVIEW employs Graphical "G" programming. Graphical programming allows the user to describe a program with a dataflow representation. Dataflow is well suited for signal processing algorithms. Such algorithms use arithmetic derived from linear systems theory to process data streams. Dataflow semantics are natural for expressing the block diagrams used to describe signal processing algorithms. As an example let's take a basic difference equation:

$$y[i]-b1y[i-1]-b2y[i-2]-b3y[i-3]=a0x[i]+a1x[i-1]+a2x[i-2]+a3x[i-3]$$

Difference equations like the one above help in understanding and manipulating LTI systems such as FIR and IIR filters. Such filters are not easily implemented using text based codes such as VHDL and Verilog. Graphically in LabVIEW the above sequence can be implemented as shown in Figure 1.

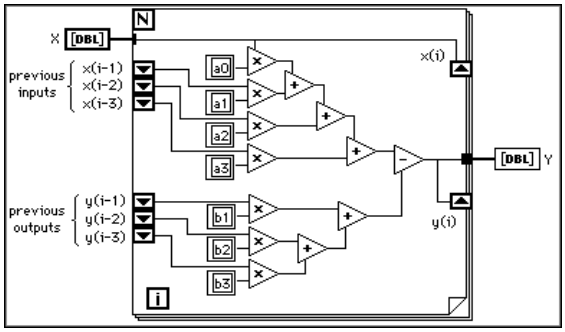


Figure 1. LabVIEW Block diagram for a Difference Equation

In the diagram above the transformation for a unit time delay is done using a For Loop with shift registers. DSP systems are also often described by signal flow diagrams like the one below:

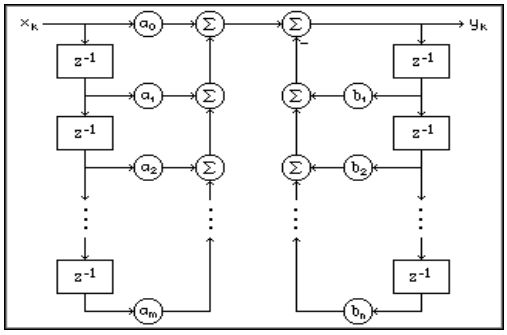
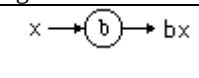
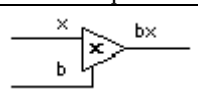
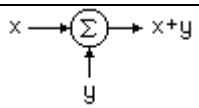
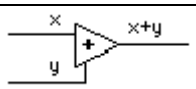
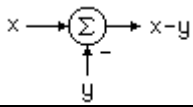
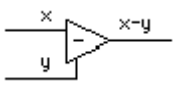


Figure 2. Signal Flow graph of a Difference Equation

One could convert signal flow diagrams into LabVIEW block diagrams by applying some simple transformations as show below

Table 1. Signal Flow to LabVIEW Diagram Conversion

Signal Flow Element	LabVIEW Equivalent
	
	
	

This graphical approach to DSP is far more intuitive than any text based alternative. The LabVIEW FPGA module is an add-on package to NI LabVIEW and extends G programming to FPGA's. LabVIEW FPGA provides an abstraction from RTL and employs a graphical block diagram approach to programming. Graphical programming is well suited for expressing the parallelism inherent to FPGA's and the timing explicit to DSP algorithms. Under the hood, LabVIEW FPGA uses code generation techniques to synthesize the graphical development environment to FPGA hardware. Tight integration between LabVIEW FPGA and NI FPGA hardware means that algorithm developers can focus squarely on developing their algorithms and not get bogged down by the complexities of digital hardware development.

The rest of the paper is organized as follow; FPGA Programming, debugging techniques, compilation and summary.

2. FPGA PROGRAMMING

In this section programming in the LabVIEW FPGA environment is introduced. The starting point for this is the LabVIEW Project Explorer Window. You must use a project to build FPGA applications. The LabVIEW Project Explorer window is used to manage the components of an FPGA application. Figure 3 shows the Project Explorer window and its various components including the FPGA VI, host VI, FPGA target, FPGA I/O, FPGA FIFOs and FPGA target clocks.

2.1. Project Explorer components

The FPGA Target in the figure above is the NI PXIe-7965R which uses a Virtex 5 sx95t FPGA. The VI shown underneath this target runs on the FPGA. The 40 MHz Onboard Clock is the base clock. A base clock is a digital signal existing in the hardware and is used to clock the FPGA application. LabVIEW uses the base clock properties when setting timing constraints on circuits generated from the FPGA VI during compilation. The DAC Clock is a derived clock created from the 40 MHz clock.

2.2. FPGA VI

The function palette of a LabVIEW FPGA VI contains a subset of the functions found in standard LabVIEW along with functions specific to FPGA hardware like FPGA I/O nodes, single-cycle timed loop, fixed-point arithmetic and integration nodes for third party IP.

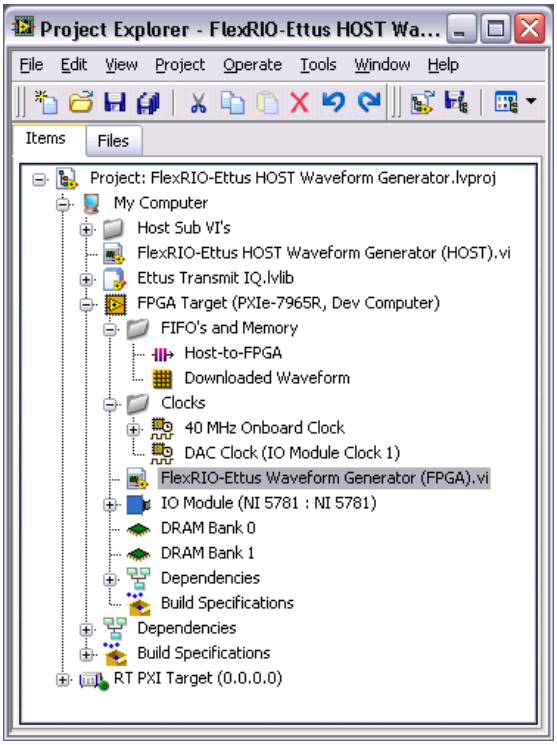


Figure 3. LabVIEW Project Explorer Window

Building a LabVIEW FPGA VI is akin to building one in standard LabVIEW; you will need to connect different function-nodes together by drawing wires. It is important to note that unlike a PC, FPGA have a limited set of resource, very function or VI you add to the block diagram of an FPGA VI uses a certain number of logic cells on the FPGA. If the FPGA VI design exceeds the number of available logic cells, the code will produce compilation errors.

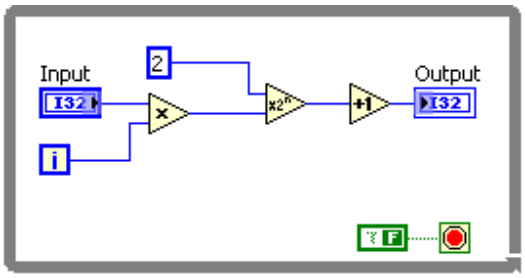


Figure 3. Simple LabVIEW FPGA VI (block diagram view)

The figure 3 above shows a simple VI that performs a simple mathematical operation. While the FPGA is running the input (a Control) and the Output (an Indicator) can be monitored on the front panel to ensure correct operation. Controls and Indicators are mapped to registers on the FPGA hardware and will only display the current values in that register. Connecting FPGA I/O to FPGA logic is a simple as connecting a wire from the FPGA I/O node to some FPGA logic as showed below.

Figure 4. Using a Digital FPGA I/O Node

In the figure above the value of the digital input 0 is assigned to the Mod1/DIO indicator. The FPGA I/O available on the LabVIEW FPGA VI is dependent on the FPGA Target being used and can support digital, digital port and analog I/O. Analog I/O supports unsigned, signed and fixed-point data types.

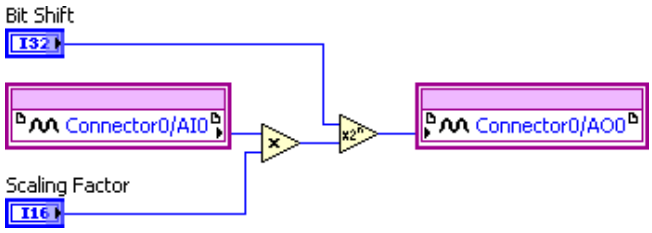


Figure 5. Using a Analog FPGA I/O Nodes

In the Figure above the value from the Analog Input channel AI0 is read and multiplied by a scaling factor, then bit shifted and the result is outputted on the Analog Output channel AO0.

2.3 Parallel Loops Execution

FPGA's allow for true parallel code execution. Figure 6 shows how two separate section of code might get mapped on FPGA hardware. Graphical programming promotes parallel code architectures because they are inherently described in the block diagram. The loops shown in Figure 7 run in parallel because there are no shared resources between the two loops. Each loop is then free to run at the rates determined by the Loop Timer parameter. A shared resource is any LabVIEW node that is accessed by multiple objects in the FPGA VI. Both the analog input and the analog output in Figure 8 are shared between the two loops. Sharing resources between two different tasks or loops can affect the deterministic execution of the tasks, even when they are in parallel.

2.4 Data Transfer on an FPGA VI

LabVIEW FPGA provides resources that can be shared by multiple processes. These resource may allow for lossy and lossless data transfer. Table 1 lists the data transfer methods available using LabVIEW FPGA, for brevity only FIFOs are discussed in detail.

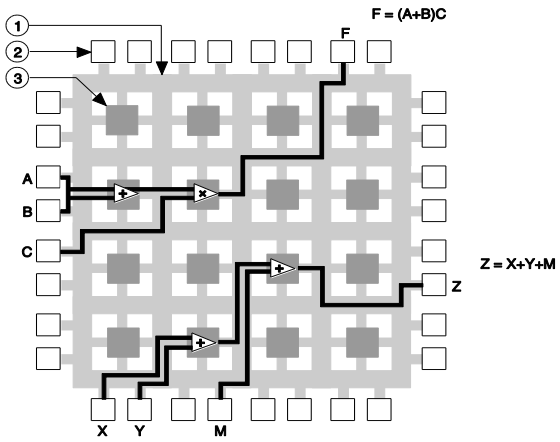


Figure 6. Function mapping on FPGA Hardware

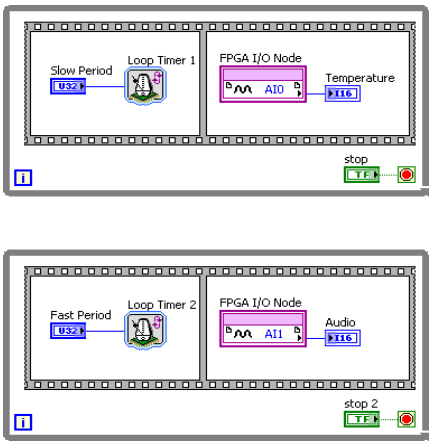


Figure 7. Parallel loops

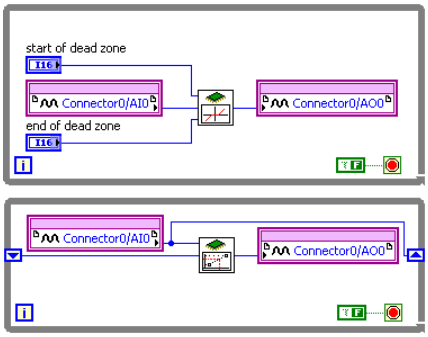


Figure 8. Shared Resources in Parallel loops

2.4.1 FPGA FIFO's

FIFO's are by far the most frequently used method for transferring data between parallel loops. Like Memory items, write and read methods exists, however unlike the later FIFOs do not have an address parameter. The Write and Read functions are shown in Figure 9.

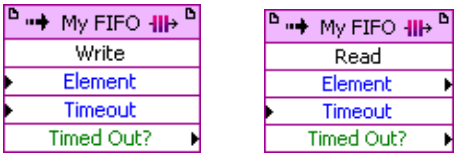


Figure 9. FIFO Write and Read

Table 1. Data Transfer Methods

Transfer Method	FPAG Resource	Lossy?
Variables	Logic	Yes
Memory Items	Memory	Yes
FIFOs (Flip-Flop)	Logic	No
FIFOs (Look Up Table)	Logic	No
FIFOs (Block Memory)	Logic and Memory	No

The inputs Element and Timeout are respectively the data element to written or read and the number of ticks the function waits for space if the FIFO is full. The output Timed Out? Is True if attempt to write failed. Does not overwrite or add new element. FIFO transfer may be lossy if write times out. If you set the Timeout value to -1, then the node will wait indefinitely. The default Timeout value is 0, resulting in no wait.

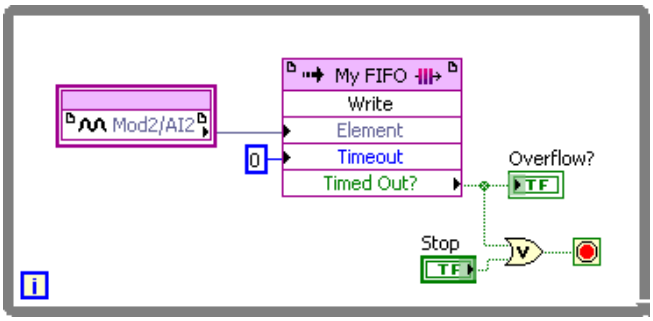


Figure 10. FIFO Write Overflow Handling

In the figure above the analog input node writes in current value into the FIFO, the loop will stop executing if the FIFO times outs. Overflow and Underflow can occur when using FIFOs it is necessary to be able to detect and handle each appropriately.

When a Write loop executes faster than the Read loop the FIFO is filled and the FIFO Write method times out. Data can no longer be written to the FIFO until space is available. Space can be created by reading the data or resetting the FIFO. Hence Data is lost until space is made available. When the Write loop executes slower than read loop, underflow occurs. The FIFO is empty and the FIFO Read method times out.

LabVIEW FPGA allows for the use of Direct Memory Access FIFO's for data transfer between a host (running Windows or a Real Time OS) and the FPGA VI. This will be discussed in more detail in the basic host integration section.

2.5 Enforcing Dataflow in FPGA

LabVIEW FPGA uses three components to maintain this dataflow paradigm. First, the node has logic corresponding to its function. In the figure below, notice the Boolean Not function and its associated logic. The next component needed for dataflow is synchronization. This component registers the outputs of the function in order to isolate the logic from timing uncertainties. Finally additional logic referred to as the enable chain coordinates the dataflow by validating the inputs and outputs.

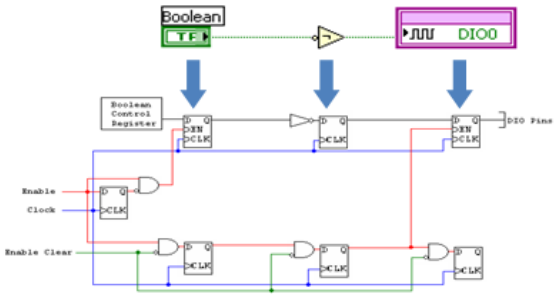


Figure 11. Dataflow enforcement using enable chain

The figure above shows a simple VI that has a Boolean control, a Not function and a Digital Output function. The Boolean control has some logic associated with the data register to retrieve data from a host application. A flip-flop links the enable chain. The Not function has the logic associated with the function itself, a synchronization flip-flop, and an enable chain flip-flop. The Digital Output function has a synchronization flip-flop and an enable chain flip-flop. When the program runs, the enable line goes high to enable the synchronization flip-flop associated with the Boolean control. Meanwhile, a rising edge of the clock pushes the data from the register through the flip-flop. Downstream, the previous values are held on the outputs of the flip-flops. During the next rising edge of the clock, the synchronization flip-flop associated with the Not function passes the new value through. On the third rising edge of the clock, the enable in of the digital output is high and the new value is pushed through the flip-flop to the I/O pins. The data is synchronized using a flip-flop that pushes the data through the flip-flop on the first rising edge of the clock.

Due to the enable chain overhead, each function or VI takes a minimum of one clock cycle. Some functions, such as analog input operations, can take hundreds of clock cycles depending upon the complexity of the operation and hardware limitations, While loops take 2 clock ticks.

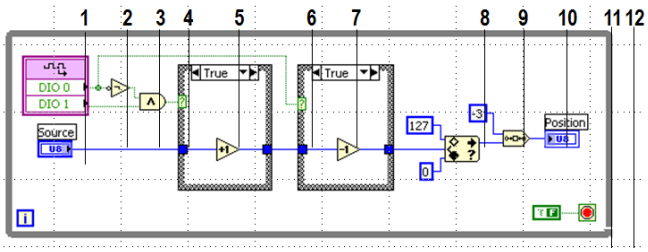


Figure 13. Dataflow Clock Tick Execution

Use a single-cycle Timed Loop to convert the above 12 clock-cycle While Loop Into this 1 clock-cycle Single Cycle Timed Loop. LabVIEW automatically optimizes code inside an SCTL.

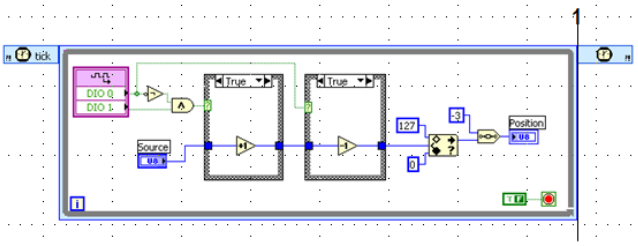


Figure 14. Single Cycle Timed Loop (SCTL)

The SCTL accomplishes this by removing the enable chain registers from code inside the SCTL. All code in the SCTL finishes executing within one tick of the specified FPGA clock and consumes less space on the FPGA.

Within a loop, you can split your code into different loop iterations to reduce the duration of each iteration, this process is called pipelining. The figure below illustrates two different ways to achieve pipelining

- Use shift registers to pass data to the next piece of code
- Use Feedback nodes to maintain the look and feel of the original application

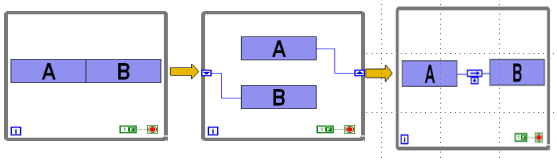


Figure 14. Two ways of Pipelining

Feedback Nodes like shift registers are implemented as registers and requires logic resources in proportion to the width of the data type. When you implement a pipeline, the output of the final step lags behind the input by the number of steps in the pipeline and the output is invalid for each clock cycle until the pipeline fills. The number of steps in a pipeline is called the pipeline depth, and the latency of a pipeline, measured in clock cycles, corresponds to its depth. For a pipeline of depth N, the result is invalid until the Nth loop iteration, and the output of each valid loop iteration lags behind the input by N-1 iterations.

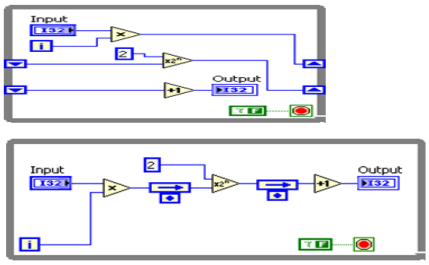


Figure 15. Pipeling the VI in figure 3

Pipelining can also be done in single-cycle timed loops.

2.6 BASIC HOST INTEGRATION

LabVIEW FPGA provides an interface to the FPGA VI running on the FPGA.With Programmatic FPGA Interface Communication, you programmatically monitor and control an FPGA VI with a separate host VI running on the host computer. You might write a host VI to send information between the host computer and the FPGA target for the following reasons

- You want to do more data processing than you can fit on the FPGA.
- You need to perform operations not available on the FPGA target, such as floating-point arithmetic.
- You want to create a multi-tiered application with the FPGA target as a component of a larger system

When you use Programmatic FPGA Interface Communication, the FPGA VI runs on the FPGA target, and the host VI runs on the host computer, as shown in the following illustration.

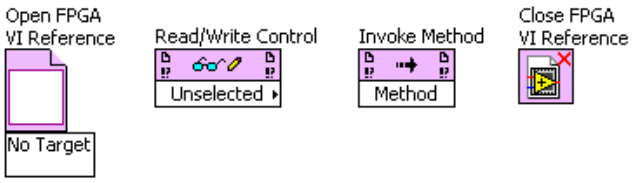


Figure 15. Host-side FPGA VI Interface Nodes

The Open FPGA VI Reference establishes communication with a FPGA VI from the host VI. The Read/Write Control nodes allows write and read access to the values in the

control and indicators respectively of the FPGA VI. The Invoke nodes allows you to configure, start, write and read from DMA FIFO's as well as toggle and read interrupt lines. The Close FPGA VI Reference terminates communication with the FPGA VI.

2.7 Third party IP Integration

LabVIEW FPGA provides an IP Integration Node that allows the user import pre-existing code. This node wraps Xilinx's OR customized VHDL modules based on .xco or .vhd files and supports both cycle accurate co-simulation and FPGA hardware execution. A detailed discussion of the IP Integration Node is above the scope of this paper.

3. DEBUGGING FPGA CODE

Development techniques for FPGA programming are significantly different than though for PC application development. The FPGA compile process can take a significant amount of time, also once the code is running in hardware there is no ability to probe, single-step, process highlight, set breakpoints, and more. For these reasons it is impossible to employ a "code and fix" method of programming, one technique is to do more simulation on the development computer to avoid unnecessary compiles due to programming errors.

Because you are just using LabVIEW code when making FPGA logic, it is always possible to execute your VIs on the host computer. This means you can use all the debugging features of LabVIEW and you do not have to wait for it to compile every time you need to test some logic. Additionally, you can create a Test-bench VI to assert the inputs that would normally be connected to the outside world via FPGA I/O and capture the outputs for analysis and verification. Finally, you can run the host program simultaneously with the FPGA code including simulated registers and DMA first-in-first-out (FIFO) memory buffers. You cannot test certain behavior, such as timing and determinism.

4. FPGA COMPILE

The LabVIEW FPGA module compiles your LabVIEW application to FPGA hardware using an automatic multi-step process.

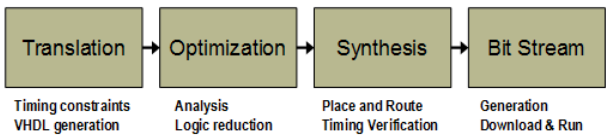


Figure 16. Compile stages

The first step in the compilation process is the generation of intermediate files. For this process, LabVIEW parses your block diagram and converts the code to text-based VHDL. The Xilinx ISE compiler tools are then 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.

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. 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.

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 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.

After code generation completes with no errors, the Compilation Status window appears. This is the main window that guides you through the compile. It features a progress bar and some basic timestamps and VI info. As the compile continues, alerts at the bottom of the window tell you when a new report is done. After the "Synthesis" step, you see "Estimated Device Utilization" and "Estimated Timing" reports. Both of these reports come early in the compile so you can cancel the compile in a timely fashion if the reports indicate with high confidence that the compile will over-map your FPGA hardware or not meet your timing constraints.

5. SUMMARY

LabVIEW FPGA side-steps the need for VHDL or Verilog knowledge and allows novices and experts alike take advantage of FPGA hardware. LabVIEW FPGA employs G programming and provides a high enough level of abstraction for translating signal processing algorithms to code that can run on hardware. The environment provides power debug and compilation features to helps ease FPGA application development