As a speaker of NIWeek, begin your session by welcoming attendees.

Convey these points:
• Your time is valuable, thank you for being at NIWeek 2017.
• We aim to make this as valuable as possible, from sharing best practices to helping connect you to fellow engineers and NI experts.
220 – Advanced Debugging Techniques for LabVIEW FPGA

John Gentile  
National Instruments  
Product Support Engineer – FlexRIO & High-Speed Serial | CLD | CTD  
john.gentile@ni.com

Jim Kneale  
Northrop Grumman Corporation  
Senior Advisory Engineer  
james.kneale@ngc.com
1. The FPGA Vis are not on the host PC

2. The controls and indicators are just ports by which the FPGA passes data back and forth to the PC – Usually at much higher rates than can be displayed

3. If I wanted to go directly to the hardware with an oscilloscope, most modern FPGAs are ball grid arrays. The signals might not be present on external pins.

4. Once you start running in the hundreds of MHz, everything is RF. You have to deal with reflections, skew, and edge rates. FPGA simulations can handle the internals, but external circuitry is tougher to model.

5. Complex logic and state machines

6. If the design was easy, I would be using a waveform generator, but we go to FPGAs when we need to push the envelope

7. A lot of us are coming at this from a LabVIEW environment running on a processor. FPGA takes a different mindset, and you are going to make mistakes along the way.

If you can manage the impact of these errors and correct them faster than your competitors, then you win.
So in this presentation we will be going over different kinds of debugging techniques and tools all with the goal of identifying a fault that’s causing an FPGA design to not function correctly. <<Ask audience>> So to start, when we say “not function correctly” what are some FPGA bugs or weirdness you’ve encountered before in your designs? (or is everyone perfect with no design bugs…) <<Our example answers>>:
- Trigger not arriving
- Timing or clocking issues
- Algorithm/logic producing errant data (or even none at all!)

The techniques and tools presented will basically go in order of simplest to most complex and its good to keep in mind that each is a different tool in your design toolbox; as such, when actually using these, pick the best one for the job or problem at hand. With that in mind, let’s start with some initial debugging techniques that you can start from when a design may have errors…
Let’s first start with simulation; simulation can be a very useful step for debugging and validation even before a design is compiled and built. LabVIEW FPGA actually provides interfaces for a couple different types of simulation methods:

- The easiest and most basic is to just use LabVIEW VIs (http://zone.ni.com/reference/en-XX/help/371599K-01/lvfpgahelp/running_fpga_vi_on_emulator/):
  - LabVIEW can run FPGA VIs on a host PC with simulated I/O without needing to compile a bitfile and run on a physical target. Since LabVIEW just runs the G code as if it was host code, the code runs almost instantly and therefore this is useful if you’re looking to quickly create or validate a design before compiling. This is especially helpful for even simple designs, as FPGA compilation can take a long amount of time and thus a guess-and-check development or debugging method eats up a large amount of time on real HW. If your design is majority or all G code, than a good practice I like to take is to develop the main parts of the algorithm and design in regular LabVIEW host VI code and check its operation. From here, most of the code is either drop-in or an easy port to LabVIEW FPGA code.
- <<Ask audience>> So does anyone have any familiarity with either of these tools?
- <<me>> So yeah, both these methods allow LabVIEW to export your design into a 3rd party application for further simulation using lower-level testbenches that can be defined with VHDL and timing characteristics can be analyzed as well as different stimulus and their responses can be plotted on digital waveform graphs. For large or complex designs, especially those designs written in HDL other than G, simulation is a very important step to help understand your code and its operation.

There are some cons to simulation though:
1) Simulation is only as good as the testbenches or VIs you write for them and some testbenches require a fairly deep understanding of HDL to create
2) Some parts of your design, such as “black box IP” or socketed CLIP, may not be able to be simulated by LabVIEW
3) Depending on the simulation method and design complexity/size, simulation may take a very long time or testbenches can be very high effort to implement
4) At the end of the day, simulation is an approximation of real operation and can completely miss some types of design faults

So let’s say after some simulation everything looks to be fine; what’s our next step? <<NEXT SLIDE>>
... well the next common issue that any FPGA designer will inevitably run into is to get a given design to actually compile into a bitfile. Now, there can be many reasons for a build to fail such as timing constraints, limited physical space, path routing, etc. but to reiterate, the focus of this presentation will be on debugging a design that can build but doesn’t operate as intended. For more info on topics of meeting timing and other build considerations, see online information such as LabVIEW High-Throughput FPGA Developer’s Guide (http://www.ni.com/tutorial/14600/en/), the LabVIEW FPGA Help (http://zone.ni.com/reference/en-XX/help/371599M-01/) or even Xilinx’s guides on meeting timing for advanced designs.

So your design builds and simulation/running on host seemed to work fine but you’re still getting errant behavior...<<NEXT SLIDE>>
...now we should start looking at debugging on the actual target. One method to employ as a starting point is to literally divide and conquer; isolate small chunks of your design into individual components that you can perform unit tests on without needing to worry about other parts of the design, for example:

- If your design is already designed as a finite state machine, try to test individual states
- Break algorithms into succinct black boxes that can operate alone
  (http://zone.ni.com/reference/en-XX/help/371599M-01/lvfpgaconcepts/fpga_debug_sim_intro/)
- And find what parts of your design are most likely at play for a given problem and start there

Once these areas of your design are found... <<NEXT SLIDE>>
... an initial first step could be to add LabVIEW FPGA controls & indicators to “probe” certain signals, Finite State Machine controls and/or critical data paths. <<Ask audience>> For instance, what could you do if you wanted to see if a clock was running? <<Possible answer>> simply add a counter and indicator in a SCTL driven by that clock to see if it increments. There’s even times when I’m creating a first-pass at a design that I’ll purposefully place an indicator down to show what the current state in my finite state machine is for inevitable debugging (see picture bottom left). I think of it similar to how board designers strategically place test points on a board layout for the ability of analog or V&V tests and debugging down the road.

The caveat to simple controls and indicators is if for instance you are trying to probe a fast toggling data path (as most FPGA logic is); there are limits to how fast a host computer can read/write the controls/indicators and therefore there needs to be more strategic thought and logic put into the LabVIEW FPGA side to effectively analyze what’s happening... [foreshadowing to Jim’s part]
1. In order to remain competitive as a software developer, engineer, or company – you need to be constantly innovating.

2. When doing so, you will come across issues that stop you in your tracks.

3. You go through the basic methods on the previous slide, and still the thing just won’t work.

4. Soon the questions will come down from customers and management, and you don’t have an answer or a path forward.

5. We’re going to show you a few additional things to try when that time comes. Yes, they take time to write, but they give you a set of powerful tools to see what is happening inside, solve the problem, and move on.

6. And it’s a concrete answer to give to your management, that it will take a fixed amount of time to create a tool with a high level of certainty to isolate the problem.
LabVIEW FPGA Logic Analyzer
We would love to be able to probe inside the FPGA with a logic analyzer.

What I’ll be describing in the next few slides isn’t any fancy programming.

A little extra effort to add a standardized monitor port to VIs that you think might give you problems can provide you visibility for troubleshooting or unit test.

The Digital WF control takes care of the formatting right out of the box.
Getting into the details

1. When creating a complex VI like a state machine, I add an extra output pin as a monitor port.
2. I’ve found a U32 works well because that’s about all the signals you can see on a digital WF control.
3. Just keep it a consistent size.
4. Populate the bits of the U32 by shifting and adding the bits you want to look at.
5. If I have a state machine, I try to use zero as the idle state, and put the state as the first 4 LSB bits.
1. Many Vis with monitor ports can’t all be viewed at once
2. Wire them to a case statement that you can select from the host
3. A single FPGA compile will let you debug multiple Vis that way
4. The case statement also determines what triggers a snapshot of data
5. I generally trigger when the state bits go from idle to something else
6. Easier to duplicate the cases when coding it up
7. You can also show data prior to the trigger
1. This VI waits for the trigger to be asserted in the first case
2. Then collects a fixed number of samples at the clock rate of the SCTL
3. A DMA FIFO then streams the data to the host
4. There are more elegant ways of doing this: circular buffer, variable samples, flow control – but this works (and fits on a slide)
Now we turn our attention to the host side.

1. Most of this slide consists of code you are doing anyways
2. We read the data off the DMA FIFO
3. When we get enough, it displays to the Digital WF control.
4. When you complete test and debug, you could disable most of the code that collects the data in the FPGA and displays on the host
5. However, I would leave the monitor ports on the Vis
6. Deleting a pin will impact your code, and if the data isn’t going anywhere, it will get optimized out in place-and-route
1. You can collect time aligned samples from multiple clock domains
2. Put your logic analyzer VI in the fastest clock rate SCTL
3. Pass data through a FIFO from other clock domains and keep the last result if it doesn’t produce new data with the next clock.
4. Note that when you try this in simulation, each SCTL will be operating in lock-step
1. The data stream can be used in other languages besides LV, by using the C API

2. In LabWindows CVI, it took some additional work to do the pan and zoom

3. This technique has resolved problems in my FPGA designs, but more importantly when I had problems with signal integrity when the product was being tested

4. It started out as throwaway code to fix one problem, but has evolved to an important component in my toolbox.
So what if we have logic (such as socketed CLIP, custom IP or non-LV diagram components) that cannot be probed by LabVIEW? Or what if we need to analyze many signals on an active device?

For awhile now, the industry standard is to use a JTAG interface to debug deeper by probing device signals and analyzing data and timing on PC. <<Ask Audience>> As a show of hands, how many of you are familiar with JTAG or have used one to debug signals? <<Me>> The JTAG standard came as a result of the increased use of ball-grid array (BGA) components and ASICs (i.e. microprocessors, memory, FPGAs, etc.); as Jim mentioned in the first slide, FPGAs and other BGA components, do not expose test points or pins to measure easily so JTAG was invented to test interconnects between ICs on a PCB without the use of physical test probes so, not only design issues like proper logic operation, but manufacturing issues like opens and shorts could be detected and caught.

This process is done by inserting a little additional logic that acts as a boundary scan register for important signal paths and then connecting those points via a scan path and some control logic. The scan register cells can capture data from connected logic signals (TDO) as well as force data onto pins (TDI). These captured signals are then serially shifted to boundary scan registers which are controlled by a Test Access Port (TAP) which is a 16-state finite state machine (FSM) controlling the JTAG pins. Multiple devices or chips with a JTAG interface can also be connected to form a JTAG chain. These devices are then usually...
controlled through a discrete set of pins on a JTAG header which can connect to a debugging computer with a wide variety of JTAG equipment (such as USB based JTAG debuggers) and software (such as ChipScope or other logic analyzers that show signal waveforms over time as well as features such as triggering and state control).

Because of this powerful interface, other non-debug operations are also commonly used with JTAG such as downloading data/bitfiles to flash, in system programming, in circuit emulation, and many more. From a debugging standpoint, though, this means a very large number of board-level test vectors can be interconnected, monitored and controlled from a relatively simple FSM interface.

For FPGA targets looking to utilize JTAG functionality, Xilinx has an Integrated Logic Analyzer (ILA) IP core that can be dropped into any design to probe and/or control specific parts of a design. The ILA core is created in Xilinx tools such as the Vivado IDE, through a simple wizard interface and after its creation, the ILA is then hooked up to specific parts of your design which are then connected to the JTAG interface. This is what allows FPGAs to have JTAG control and monitoring as well as opening up the debug options mentioned previously. As well, because the ILA core is synchronous to the design being monitored, all design constraints that are applied to your design are also applied to the components inside the ILA core creating a simple, non-intrusive debugging solution that can natively interface with host PC debugging applications like ChipScope (pictured) which provide graphing of digital waveforms and controls such as triggering and bit-banging (from https://www.xilinx.com/products/intellectual-property/ila.html). However, I want to state again though that this a low-level debugging tool as the ILA core has to be attached to signals of interest in VHDL-based designs, such as socketed or user component level IP, not designs made in LabVIEW VIs or G-code.

While this is a great and time-tested solution, it does require physical connections to be made to the DUT/FPGA board such as a formal JTAG header or unused device digital I/O as shown on this FlexRIO digital FAM. On a side note, speaking of the setup in the picture, we do have a partner that makes a FAM-to-DIO breakout accessory meant especially for JTAG
debugging. However, for some devices or scenarios, this may pose a problem; if your system doesn’t have a header or spare DIO to be connected, than you’re kinda out of luck 😞. So what if there was another way?...<<NEXT SLIDE>>

Surprise surprise, there is! Xilinx Virtual Cable (XVC) is a TCP/IP-based protocol layer that wraps the ILA/JTAG functionality. The operation is that a host controller that is bus connected to the FPGA interacts with the JTAG interface running on the FPGA and opens up a TCP/IP server to allow the JTAG interface to be interacted with by remote connections as if they had a direct JTAG debugger to the FPGA. This simply means we can have the same JTAG and debug features we just talked about without using a physical JTAG cable! This also means the same debug flow and application tools are used (again Vivado HW manager or ISE ChipScope) so this allows us to debug in areas we previously couldn’t such as when:
- The previously mentioned case where there’s no JTAG header or extra digital IO on the board or
- The FPGA device is deployed in a hard-to-access location (say enclosed in a hardened box, or on top of a windmill or in a secured lab or even deployed half-way around the world!) and finally
- If simply extra efficiency is needed when there are many systems to monitor or debug

As the pictures show, as long as the FPGA target and controller can be network connected and run a basic TCP/IP server, which a lot of NI HW can, it can take advantage of XVC debugging. However, I wouldn’t suggest XVC as an initial debugging step as it requires more steps than JTAG interfacing and requires network connectivity. JTAG is also a better candidate when in development or V&V at your desk whereas XVC is better in remote...
debugging or prolonged monitoring of a deployed system. So, to go a little deeper into what
XVC and instantiating ILA looks like... """"NEXT SLIDE"

Ref: https://www.xilinx.com/products/intellectual-property/xvc.html,
https://github.com/Xilinx/XilinxVirtualCable
Here’s a block diagram overview of the XVC process. To make a design or system XVC capable is relatively simple and essentially the process goes (click through):

1) Create the ILA logic analyzer core step-by-step using Xilinx wizard and then hook the ILA into desired parts of your IP. Bring your IP/CLIP into LabVIEW using the LabVIEW FPGA IP Wizard and...

2) Attach controls and indicators to JTAG CLIP signals to allow for the host controller to read/write the JTAG interface. As well at this time, you can create any other necessary LabVIEW FPGA design logic in the VI, such as FIFOs, DRAM, etc. and compile & run your design on your intended target.

3) From here, you can create and run the XVC server host-code, which again can be just a simple TCP/IP server; this can be made in LabVIEW, C/C++ (using the NI FPGA C API) or any other language that can interact with the LabVIEW FPGA controls and indicators we exposed in the last step. This is what allows for remote connections to interact with the JTAG signals on our target’s HW. Finally...

4) Attach to XVC server with another network connected host PC using applications like Vivado HW Manager or ChipScope Pro. Here, we can do things like read data signals on specific trigger conditions, send data to the FPGA, etc.

For more information, see the hyperlinked tutorial on ni.com for a detailed walk-through on implementing the XVC protocol with NI FPGA targets as well as you can reference some
Xilinx debugging guides on the topic
XVC Demo

[if time permits...]
So we talked a little about Xilinx Vivado for XVC and JTAG debugging, but I also wanted to point out a new feature we are introducing in LabVIEW FPGA 2017 called Vivado Project Export! Vivado Project Export supports some NI FPGA-enabled hardware (right now FlexRIO and High-Speed Serial devices) to take the whole design of the project, as in user IP AND LabVIEW parts, into Vivado allowing for development and debugging of full designs using Xilinx Vivado (as opposed to just a designer’s HDL IP). Some results of this are:

- The benefits and redundancies of hardware/platform abstraction and other NI primitives are retained in the exported project, so the digital designer need only worry about their IP or socketed CLIP. To reiterate these useful parts of NI IP encompass:
  - Hardware-specific “Board Support IP”
  - System bus and I/O interfaces (like ADC/DAC, PCIe, DRAM, DMA FIFO, etc.)
  - Software platform and board driver support (i.e. NI-RIO)
- From a debugging standpoint, this opens up even more possibilities beyond logic analyzers and JTAG interfaces as the designer now has access to:
  - Vivado generated intermediate files such as design checkpoints and all reports
  - More accurate (and possibly easier) simulation and development if you have developers on your team used to the Vivado workflow
  - More Vivado tools (Timing Analyzer, Constraint Editor, Floorplanner, etc.)
  - And even 3rd party tools...

[From Rob Bauer’s Vivado HW Export Slides]
In summary, this is a simple table to layout what we went over in regards to debugging. Thank you all for listening... <<PAUSE AND THEN NEXT SLIDE>>

<table>
<thead>
<tr>
<th>Method</th>
<th>When to Use</th>
<th>When Not to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>Initial design, POC</td>
<td>Deployment</td>
</tr>
<tr>
<td>Unit Tests/Basic LV Probes</td>
<td>V&amp;V, simple designs</td>
<td>Large/HDL designs</td>
</tr>
<tr>
<td>LabVIEW Logic Analyzer</td>
<td>V&amp;V, complex debug</td>
<td>HDL-heavy designs</td>
</tr>
<tr>
<td>JTAG</td>
<td>HDL-heavy designs</td>
<td>Remote, All LabVIEW</td>
</tr>
<tr>
<td>Xilinx Virtual Cable</td>
<td>HDL, remote systems</td>
<td>Initial design, V&amp;V</td>
</tr>
</tbody>
</table>
Questions?
We want your feedback. Please take one minute to complete the session survey.
Convey these points:
• Thank you for attending NIWeek 2017.
• There’s many ways to stay connected.
• You can download the session materials through the mobile app or at ni.com/niweekcommunity.
• If you’re interested, you can reach me at [share your email]