Employing Multi-FPGA Debug Techniques

Traditional FPGA Debugging Methods

Debugging in FPGAs has been difficult since day one. Unlike simulation where designers can see any signal at any time, signals when mapped to a FPGA may be difficult to locate or even worse optimized away. Even after you identify where the signal is, it may be difficult to capture the time period in which you would like to observe that signal as the FPGA runs at real speed and you cannot continuously capture and store the waveform of that signal. Therefore, some sort of triggering and waveform storage circuit is needed to perform debugging in an FPGA. Let us take a look at the two popular approaches today: external logic analyzer and internal logic analyzer.

External Logic Analyzers

Let's first take a look at the use of external logic analyzers that have been in use for years. Popular external logic analyzers today are from Agilent and Tektronix and can sample at GHz frequency and store GBs of waveforms. External logic analyzers have the ability to store large amounts of trace data but for the data to be useable, the data needs to be taken off the chip, which can be a difficult task. The signals, or probes, that a designer wants to observe need to be sent to FPGA I/O pins to connect to a logic analyzer. Since some probes may be buried deep inside design hierarchy, it may be time-consuming to get the right probes to the top of the design.

Physically, you also need some kind of adapter card that connects the FPGA I/O pins to the logic analyzer header. For example, Agilent logic analyzers use a 38-pin Mictor connector. Most off-the-shelf FPGA boards do provide optional daughter cards that can connect the FPGA I/O pins to the 38-pin Mictor connector. If you are building your own (RYO) board, then you should reserve a set of pins to connect to the Mictor connectors if you choose to have the ability to observe through a Logic Analyzer.

The biggest drawback for the use of external logic analyzers is actually the limited number of probes you can observe at a time since there are only a limited number of FPGA I/O pins you can use for debug. In most designs, the majority of FPGA I/O pins are used for external target interfaces or used as interconnects to other FPGAs if more than one FPGA is used. Therefore, reserving a large amount of pins for debugging through an external logic analyzer may not be feasible. Multiplexing the probes to I/O pins can solve the limited pin issue but is almost never used since

external logic analyzers need to capture data at real speed and also need to support de-multiplexing on the logic analyzer side.

Once connected, the external logic analyzer is used to set up triggering and data capture conditions. Triggering is typically done using a state-machine technique whereby values are specified for a signal and then either the data is captured or a different condition is sought after on another state. The signals remain static while the conditions can be altered at any time. Trace memory using an external logic analyzer is rather large therefore memory can afford to be wasted trying to find trigger conditions that are close to desired observation points. The advantage of an external logic analyzer is that it can sample at high frequency (in the GHz range), at high accuracy, and support very complex triggering conditions. Today, some designers still prefer to use an external logic analyzer because of these advantages as well as the feeling that debugging needs to be seen on real equipment, not just through a software tool.

Internal Logic Analyzers

Internal logic analyzers such as Xilinx's Vivado or Altera's SignalTap utilize cores that are embedded into the design whereby the trigger conditions are set using a GUI in software on a PC through a JTAG interface. The captured data is transferred to the PC where it can be viewed and analyzed. The internal logic analyzers provided by the FPGA vendors are tightly integrated with their FPGA place and route tool making them easy to learn and use.

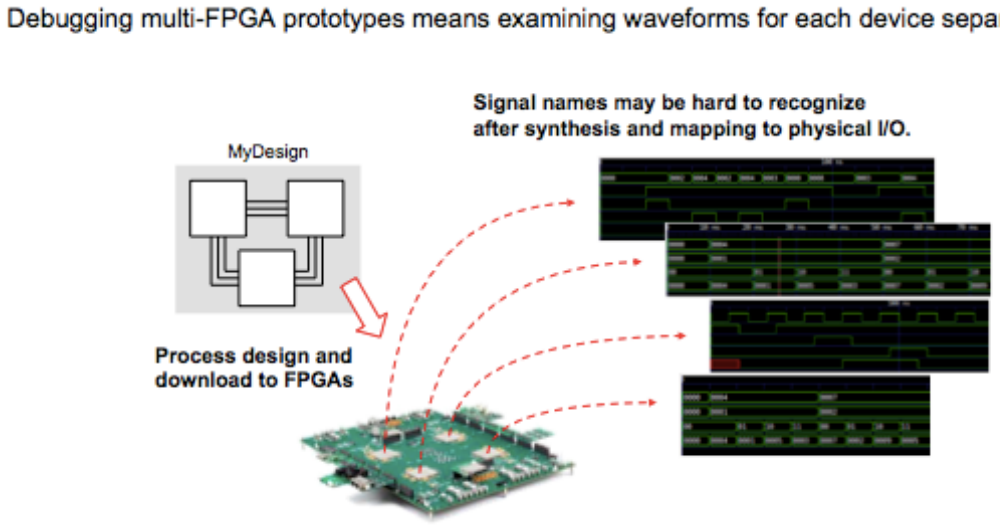
However, trace data needs to be stored in the FPGA internal block memory before a triggering condition is met and therefore they can only achieve very limited width and depth. Often, you have to choose between limiting the amount of memory you can have for your design versus allocating some memory for debugging. When a triggering condition is met, the logic analyzer stops storing new waveforms in the memory and shifts out current memory content through JTAG. The process can be slow if trace data is large. The probes must be statically defined and trigger conditions can be dynamically changed during debug just like with external logic analyzers. Most internal logic analyzers only support probing at the gate level so signal names may have changed or may have even been optimized away. Probes are static, meaning that to change probes you usually need to re-compile the design.

Some third party internal logic analyzers do support RTL probing which can improve the user experience. They also provide more advanced triggering and analytic features that allow you to get meaningful data from limited amount of waveform storage memories inside an FPGA.

Even though internal logic analyzers supplied by the FPGA vendors have quite a few limitations, they are still by far the most popular tools used for FPGA debugging today. This is due to their relative low-cost and tight integration with the FPGA vendors' own place-and-route tools.

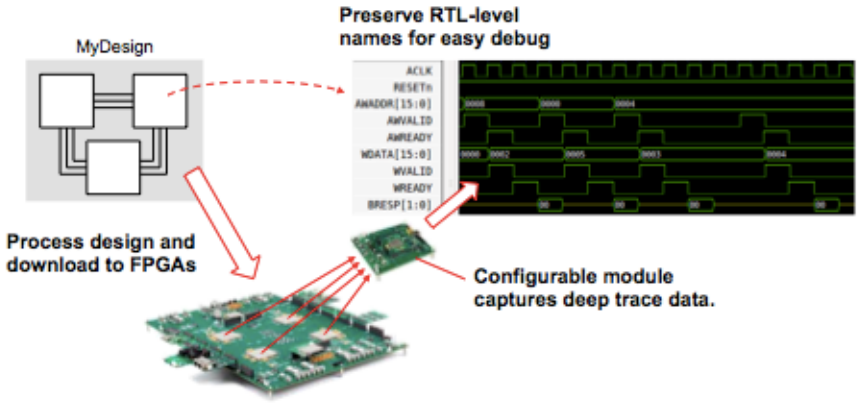
Multi-FPGA Debugging Methods

External and FPGA Internal Logic Analyzers are better suited for debugging a single FPGA. Although external logic analyzers can probe signals simultaneously from a multi-FPGA environment, the limited number of probes available makes debug inefficient. Debugging only one FPGA at a time in a multi-FPGA environment makes effective debug of the design significantly more difficult, time-consuming, and error-prone. These logic analyzers can only provide a subset of the picture in which to debug and don't have the trace depth for delving into the behavior of a multi-FPGA design. Debugging only a piece of the design at a time can lead to errors in other parts of the design as the bugs are fixed. The difficulties involved with this type of approach are illustrated in the diagram below.



What's needed is a holistic approach to debug for multi-FPGA platforms to ensure design behavior is not affected as bugs are corrected because RTL-level signals and module names are maintained throughout. With the use of a configurable external module, multi-FPGA debug will also allow for the detection of very hard to find corner case bugs because of the deep trace depth that can be achieved.

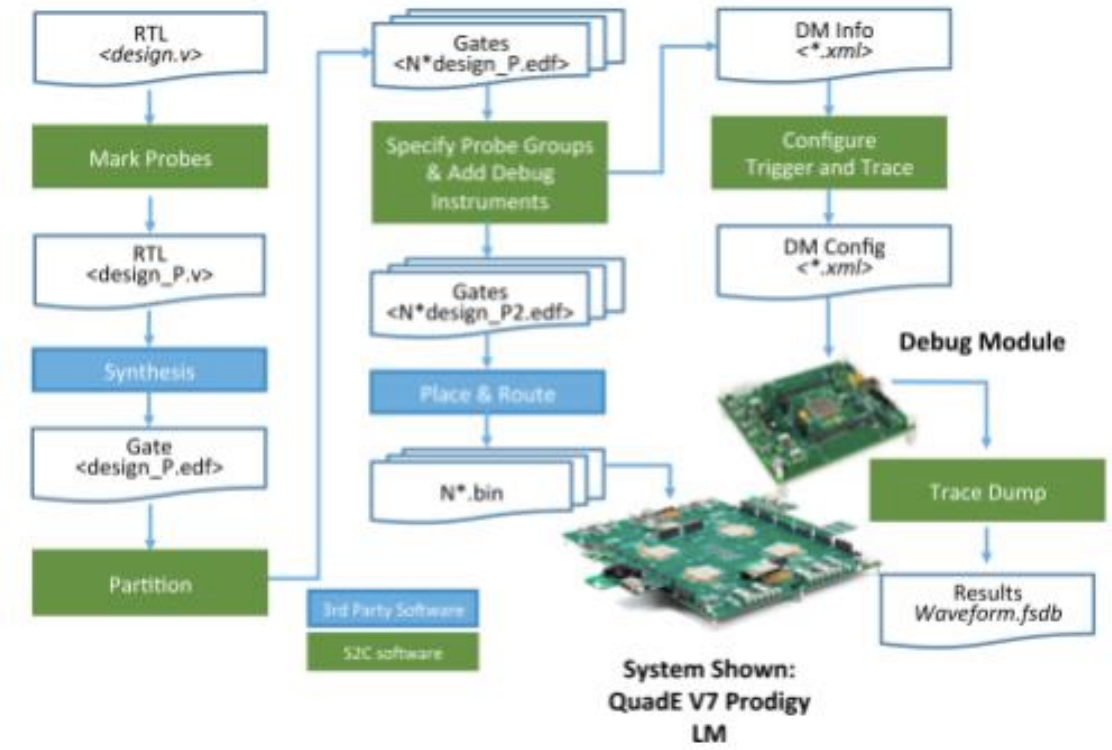
- Multi-device debug – maintain RTL-level signal & module names
- External module maintains a long/deep trace – making long runs with system data possible.



Benefits of using a multi-FPGA debug approach

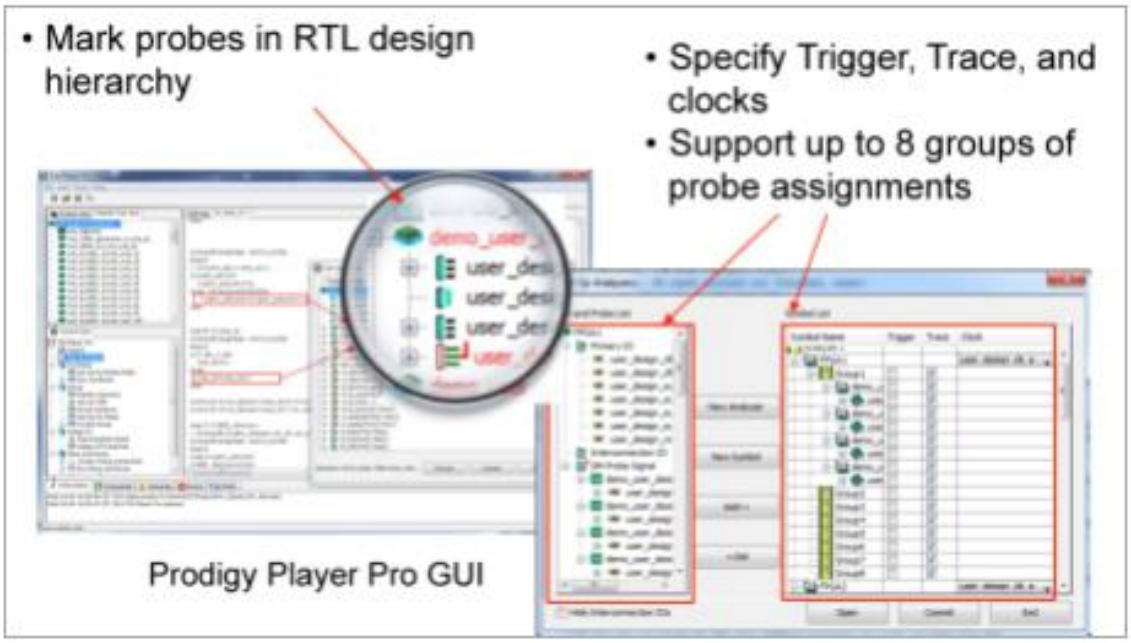
To understand this more, let's take a look at how multi-FPGA debug actually works. In the diagram below you can see that you must first mark probes at the RTL level so the probes are maintained throughout the compile flow. There should be no limit to how many probes you can mark as this simply tells the synthesis and partition tools to retain the RTL names for probing. After a design is partitioned to multiple FPGAs you can start selecting the signals you would like to probe in each FPGA. Multiple groups should be supported so you can see thousands of signals from any FPGA. Debug instrumentation is then added to each FPGA for FPGA place-and-route. Note that since the triggering logic and waveform storage are performed using an external module, the debug instrumentation in each FPGA consumes very little resources inside your design FPGA.

After the multi-FPGA design is compiled and downloaded to FPGAs, you can now set your trigger conditions and the information is uploaded into the dedicated debug module hardware. When you start running your design, the debug module will capture and store the waveforms continuously from multiple FPGAs in external DDR memory. The communication bandwidth between the debug module to each FPGA needs to be high in order to trace wide waveform at high speed. Then, when a trigger condition is detected by the debug module, the DDR3 memory content is sent to the host computer for analysis via a high speed PC port such as Gigabit Ethernet. The waveforms in VCD or FSDB format can then be debugged using popular waveform debug tools such as Verdi. Signals from multiple FPGAs can be viewed in a single waveform window.



Multi-Debug Flow

The use of a separate debug module of this nature allows for deep trace with a large number of RTL-level probes, the use of minimal FPGA resources to avoid design impact, and system-level debugging across the entire SoC design. An example of this device is the Prodigy Multi-Debug Module from S2C. The Prodigy Multi-Debug Module supports up to 32 FPGAs at a time with 16GB of DDR3 trace buffer and can utilize up to four 5GHz transceivers to capture waveforms from each FPGA to the debug module. The use of Gigabit Transceivers allows large amounts of data to be transmitted at high frequency. General-purpose I/O pins are not occupied by debugging so they can be used for interconnecting between FPGAs and external interfaces. Deep trace is achieved with 16GB or trace memory with actual trace depth dependent on the number of signals that are probed. S2C also provides an easy-to-use GUI (as shown below) that allows you to mark probes in RTL before synthesis, quickly locate probes after design partitioning, and select probes before FPGA place-and-route.



Mark probes in Prodigy Player Pro

There are variations of the above multi-FPGA debug approach. There are solutions that use cascading instead of distributed topology to collect trace data from multiple FPGAs. Cascading topology means that trace data from multiple FPGAs needs to be collected to a single FPGA through potentially many FPGAs before transmitting to an external debug module for storage. Cascading topology is easier to implement in hardware but the large debugging data going from FPGA to FPGA can create a bottleneck that in turn reduces the amount of probes that can be seen at any one time and decreases the speed at which they can be captured. Distributed topology, on the other hand, sends debug data continuously from every FPGA directly to the external Debug Module. The hardware is more difficult to implement but this method can maximize the amount of probes that can be seen at the same time as well as produce faster capture speed.

Other advanced FPGA Debug Techniques

One technique to increase trace depth is to compress the waveform being temporarily stored in the memory. The compression needs to be lossless and meet the performance required to continuously store incoming waveforms from multiple FPGAs. This waveform compression technique has already been developed in some third party FPGA debug tools to address the trace depth issue.

Hardware assertions in FPGAs is another interesting area that can make debugging FPGAs easier. Instead of continuously capturing large amounts of data and looking for trigger conditions to shift out the waveform for analysis, you can embed the conditions

that you are looking for together with your design in the FPGA. When such conditions occur, you receive hig- level messages such as: a memory is full, a bus has a contention, the CPU is at a specific state, etc. Nevertheless, most tools today do not generate synthesizable assertions that can be mapped in FPGAs so designers will have to write and embed assertions in the FPGAs themselves.

Finally, some newer FPGA families now support register and memory readbacks and even allow you to set the register and memory content. The readback feature enables you to access all nodes inside an FPGA at a given time. However to access that information, you would need to stop the design clock to shift out the register data. Therefore this feature can only be used when the design is run in a controlled clock environment and not really useful when running FPGA prototypes in or close to real time speed. In addition, just by taking a snapshot of what's inside an FPGA cannot solve the issue/bug you are looking for. Are you taking the right snapshot and how many snap shots do you need to take? Readback data is often shifted out through a JTAG port which is also very slow when dataset is large. FPGA vendors do have plans to improve this feature by allowing shifting out the readback data without stopping the clock as well as using a faster protocol to shift out the data. We hope to see better support of this feature from Xilinx and Altera and also a complete environment that allows designers to quickly see what they are looking for.