FPGA Design Verification: Techniques for Creating a Fully Functional Design

Synplicity®
Simply Better Results

Synplicity, Inc.

Introduction

Verifying designs for FPGA implementation requires a set of tasks and tools unique to the technology. Unlike designs destined for an ASIC, standard cell or custom implementation of an FPGA design does not generally have extensive resources allocated and preparation devoted to design verification. In many cases verification is more of an afterthought due to the ability to reprogram the device.

In reality the failure to plan and budget for FPGA design verification can cause product development disruptions and delays that are just as serious as those caused by masked technology re-spins. Assuming that verification risk is low, because errors can be corrected by reprogramming, fails to consider the effort required to track and correct the errors especially in complex, multiple-clock FPGA designs. The effect may be magnified considering the inter-dependence of device, system, and software development for example, any delay by the hardware team can hamstring the efforts of others.

There exists a hierarchy of verification techniques and tools available for use during the course of FPGA development that can greatly reduce the risks of using the devices. Within the hierarchy the initial verification steps tend to be performed at a high level to detect gross functional errors. As the verification process proceeds through to the ultimate goal of a fully functional design operating at speed, the design problems become increasingly esoteric. Often the problems are data or timing dependant. Some problems may be so rare as to require hours or even days of run time at full speed before a single occurrence can be detected. Such errors tend to be highly dependant on obscure events. When they do happen, the test environment must be prepared to capture them and provide data that will allow for analysis.

This paper will consider the tools and techniques that may be used in the course of a verification cycle and will examine the benefits and shortcomings of each.

Simulation

Simulation compiles a design into a form where it can be driven with logic vectors representing the signals driving the inputs. The design behavior can then be viewed in a waveform. Designers use logic simulators to find and correct errors in designs.

Simulation operates on RTL and so provides the designer with a familiar view of the design. It offers the most complete view of and control over a design in that any node can be viewed or forced to a logic level. Simulation of large designs with many vectors is extremely slow, however, relative to the speed of the actual device operation.

Most simulation is performed using a testbench that typically includes the design itself and logic to drive all the design inputs such as data, clocks, reset, and control pins that operate the design in the system. The data pattern and timing of the inputs are determined by the logic.

If the design includes complex modules such as for communicating with external Ethernet or PCI ports or an external memory controller, then the testbench may also contain simulation modules of the external devices that model their interaction with the FPGA during system operation. Simulation modules may be created by the same designer who designed the FPGA logic and are based upon their understanding of how the interface works. Interface operation is inferred from the timing diagrams of a device data sheet.

When third party IP products are used in the design, very often an interface model is included with the IP that was used to verify it. The model can be incorporated into the design testbench design to drive the IP interface.

Testbench simulation uses fixed signal timing assignments and logic models of external devices to the FPGA as stimulus for the simulation. The results of the simulation are displayed on a waveform viewer. The designer debugs the logic by comparing the desired behavior with the waveforms and iterating the design source files.

RTL simulation shows only the logic behavior, however. If a signal's timing or a model performance does not behave just as the actual device, then the simulation results will not match what the device is exposed to on the system board. While it is possible to back-annotate the target device timing after design implementation so that the simulated behavior reflects the actual implemented timing, that does not correct for any differences in the timing between the testbench and the actual system operating environment.

Clocks are a particular source of concern especially in a multiple clock system where there is no practical way to model the phase and frequency relationship between the clocks in the testbench. Yet such timing relationships may well determine whether the design works in the system.

When a design is simulated, the level of verification is a function of how accurately the simulation vectors mimic the operating environment. Designs that are simulated with vectors that accurately reflect board conditions can detect most bugs. Modeling such conditions is not easy, however, and most vectors offer only an approximation of the actual stimulus. In most cases, simulation offers incomplete verification.

Hardware Verification

While simulation offers an important level of design verification, it does not provide sufficient understanding of the design performance in the system because the stimulus can never be more than an approximation of the actual physical environment. Hardware verification offers the only means of understanding what is happening inside the FPGA as it operates in the system at clock speed.

Hardware verification of FPGA designs may be divided into external verification with logic analyzers, and internal verification with hardware debuggers.

External Verification Using Logic analyzers

Originally used for board-level debug, logic analyzers have also been used to monitor internal FPGA signals via I/O pins connected to the internal signals. The logic analyzer offers the benefit of capturing data from the board and from within the FPGA at the same time. This ability offers analysis of the entire system.

Connecting or probing the FPGA with a logic analyzer is accomplished by using pins on the PCB that are connected to unused pins on the FPGA. The board pins are then connected to a logic analyzer pod. The FPGA pins are entered into the design as ports that are connected to nodes in the design. The design is then routed using the programmable logic vendor tools. The signal values are captured and displayed on the logic analyzer.

The above method provides a window into the device in system operation as the logic analyzer captures data from internal nodes in the device, but has a number of shortcomings. First, the designer must instrument the design manually by connecting internal nodes up through the hierarchy to the device pins connecting external board pins. The process must be repeated for each iteration. Nodes that are not at the device level of the design must be routed to the top-level. The process can be error prone and time consuming,

Second, the probing capacity is limited by the number of free pins available on the device and the number of pins placed on the board. Third, signal names have to be entered into the logic analyzer viewer in order to track which node in the design is displayed on which line. The names have to be re-entered every time the probes are moved. Finally, routing nodes in the design to a pin in addition to the design destination may interfere with device operation or timing.

There exists a tool that eases the connection of design nodes from the flat netlist version of the design. This technique eliminates the burden of threading signals from nodes buried in the design hierarchy to the top level. Its

deficiency comes from the use of the netlist itself where the name and functionality may change from that of the RTL version without warning and providing false information.

Logic analyzers offer sophisticated triggering options, but triggers are not designed for trapping FPGA internal events and can only operate on nodes that are connected to the external pins. Debugging FPGA designs with a logic analyzer is time consuming, limited, and awkward.

Hardware Debuggers

Unburdened by the artificiality of simulation or the awkwardness of logic analyzers, hardware debuggers represent the ultimate system verification tool. Unlike simulators, debuggers display actual device logic data to show designers what the logic is actually doing inside the device running in the system at full speed. Using device resources to exchange data and commands frees debuggers from the requirement for device pins.

Among the available debuggers are to be found different features and methods of operation. Debuggers can be roughly divided into two classes as either based on using the design netlist or source code. Debuggers also offer different trigger mechanisms and features.

Netlist Debuggers

A netlist debugger is defined as one that operates on the flat, netlist version of the design that is produced by synthesis. Probes and triggers are added directly to the netlist and the data is displayed on waveform viewers.

RTL Debuggers

An RTL debugger is one that displays the design hierarchy and that adds probes and triggers directly in the source code. The results are annotated to the code and are seen over time by advancing or retarding the display for each clock. A waveform view of the results is also available.

Debugger Requirements

A number of requirements are elemental to a tool to be useful in verifying designs. Each of these requirements, in turn, necessitates a range of capabilities. The degree to which a tool offers these capabilities determines to a large extent its utility as a verification tool.

Clearly the data must accurately reflect the RTL functionality and designers must be able to relate the data back to the original design. The debugger must offer a way to view and interpret the results.

All debuggers probe design nodes and display results graphically. To maximize efficiency, however, a debugger should probe and display results at the RT level because that is the version of the design that is familiar and where the logic must function as intended.

When using a hardware debugger it is crucial to capture the precise data needed to discover bugs and verify system behavior. Not only must the tool locate the logic transitions around a certain event, but track bugs that may be rare events and ensure they are trapped for closer examination.

The debugger must offer triggering mechanisms that can capture any logic event of any duration or frequency and gather data based on that event. Triggering must be powerful enough to traverse a series of events to arrive at that which is the trigger. Finally, the debugger must be able to detect events across multiple clock domains such as metastability that result in transitions within a period.

Each of these requirements is discussed in the following sections which contrast netlist tools with those that operate on RTL.

Adding Probes and Triggers

A debugger should support fast and easy addition of probes and triggers and should allow the probe data to be easily referenced to the original design. Most importantly, it must ensure the integrity of the data at all times. It should offer the ability to select individual nodes, entire busses and fields of busses for probing or triggering.

Netlist debuggers display the entire list of design nodes for selecting individual nodes to be probed or used as triggers. The list can be reduced by filters to show only a subset. Nodes are selected for designation as probes or triggers.

Names of nodes may be unfamiliar as they may be generated during compilation. Their functionality may have been changed during compilation.

An RTL debugger adds probes and triggers directly in the source code. Figure 1, below, displays a compiled design in Synplicity's Identify® RTL Debugger software with all nodes that may be probed marked with an icon. Selecting a node opens a menu to set the node as a probe, trigger or both. Because the design hierarchy is maintained, navigating the design to locate probe points is straightforward.

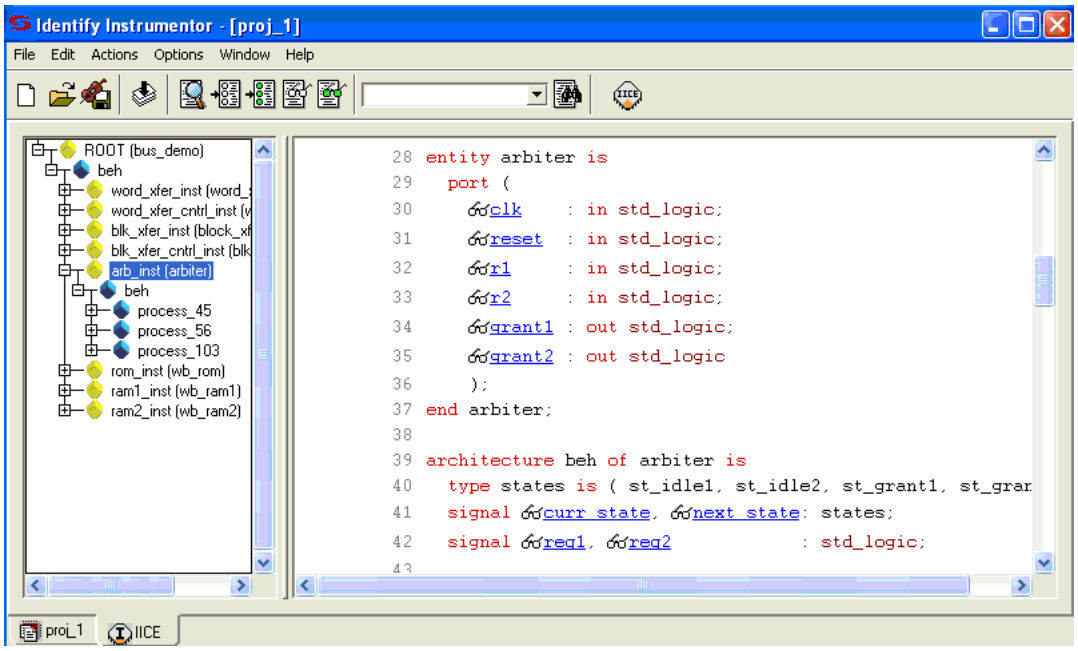

Figure 1: Synplicity's Identify Instrumentor

In addition to probing individual nodes, debuggers should also be able to capture data from busses and display bus data. Busses and fields of busses should also be available to serve as triggers just as individual nodes. Fields should be programmed to check for specific values and ranges of values.

Busses are flattened in the netlist, but a bus can be reassembled in a netlist debugger by selecting each of its bits individually. There is no way to probe or trigger on parts of a bus as such since the concept of a bus has little meaning in the flat version of the design.

The Identify RTL Debugger offers individual nodes, busses, and partial bus instrumentation. Partial bus segments are defined using the menu, Figure 2, which assigns fields to be sampled or act as a trigger.

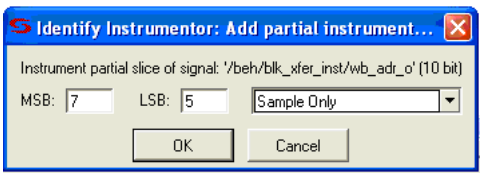

Figure 2: Identify Instrumentor, Bus Trigger Menu

Each partial bus segment can be instrumented using the bus trigger menu displayed in Figure 2. After specifying the field, the choices are to probe (sample) the field, trigger from one of several patterns in it, or both.

Moving Probes

It is difficult to anticipate all the nodes that must be probed in order to diagnose all the errors that are discovered in a design during debug. In most cases both the number and locations of probes will be iterated during verification. In the worst case these changes require a complete recompilation including synthesis and P&R of the entire design. Ideally the probes can be moved with minimal effort by the user and without recompiling.

The Identify RTL Debugger tool and some netlist debuggers offer incremental compilation where probes or triggers can be moved from one point to another by invoking the FPGA vendor editor to modify the existing routing.

Triggering Across Clock Domains

Multiple clocks are frequently used in FPGA devices that feature numerous dedicated clock buffers. In multi-clock systems it is common to encounter timing problems related to clocking data between domains. Such problems include metastability, failure to meet setup or hold times, and dropped data. Detecting these often subtle problems is usually difficult. The problem may not appear in logic simulation at all and be detected in debug only by over-sampling within a domain or by triggering from one domain and sampling in the other. Hardware debuggers should offer a means to detect the type of problems that stem from the transfer of data between clock domains.

Netlist debuggers do not offer any sort of mechanism to detect inter-clock timing problems.

The Identify RTL Debugger tool offers a cross triggering feature, shown in Figure 3 below, that allows the trigger logic of one domain to drive and enable the trigger in another. Cross triggering can be used to view the timing of events that cross domains. It can also be used to see events that occur within a clock period by over-sampling the period with a faster clock.

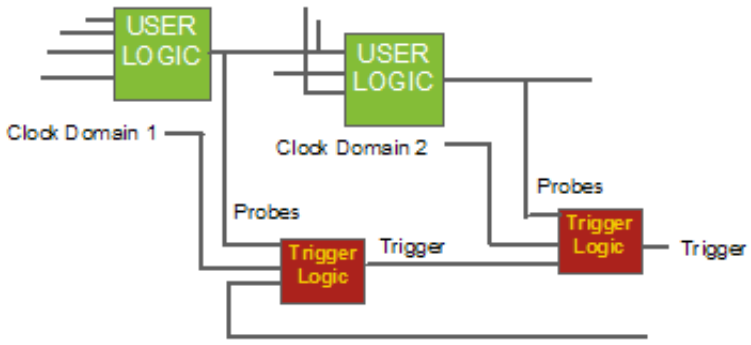

Figure 3: Cross Trigger Example

Sampling and Trigger Modes

Sampling modes control the way data is added to the buffer when a trigger condition is reached. These modes allow users to sort data inflows according to the mode and increase the efficiency of the buffer by storing only relevant data. Examples include filling the entire buffer, storing a single line, or storing until interrupted.

Trigger modes described the timing of data storage relative to a trigger condition. Examples include the number of trigger cycles, the width of cycles, and the delay after a cycle.

Netlist debuggers offer only very simple buffer controls, the number of samples per trigger or buffer configuration, but no real data sorting capability.

The Identify RTL Debugger has four sampling modes and four triggering modes. The sampling modes control how the buffer captures data. The triggering modes control the timing of captures.

State Machine Triggering

The most precise and powerful means of detecting a unique condition is through the use of a state machine as a trigger. A state machine can traverse between states on any condition as it tracks sequences of events. State machines can create a sequence of steps and conditions that must be completed by the device in operation to arrive at one or more trigger conditions. Trigger conditions can be associated with any state so that the machine can gather data on obscure events.

Netlist debuggers do not offer a state machine trigger that is integrated into the tool, but do allow the use of a manually implemented state machine or counter to trigger on an event. The state machine is added to the design by editing the code. The manual solution would require that the logic be manually adjusted and new trigger nodes specified during instrumentation for each trigger adjustment and then re-synthesized. Debugger triggers would be attached to the nodes of the state machine.

Netlist debuggers do include a simple form of serial state triggering in that a succession of trigger conditions must be met before advancing to the last condition that buffers the data. There is no provision for alternate paths or counting the occurrences of a state or time to remain in a state.

The Identify RTL Debugger offers a state machine editor to automate state machine triggering by providing a menu-based editor that includes macro trigger functions. Adjustments such as whether to trigger on a state, under what conditions, and how the counter will be used to trigger, are made in the debugger and may be dynamically changed during debug without tampering directly with the design.

The editor, shown in Figure 4, has macros using any trigger or conditional modes or one of two sample modes similar to those in the state machine. The macro editor contains fields that determine which condition will be used for the state and the number of events or samples that will be counted. Each state has conditions under which it will transition to another state.

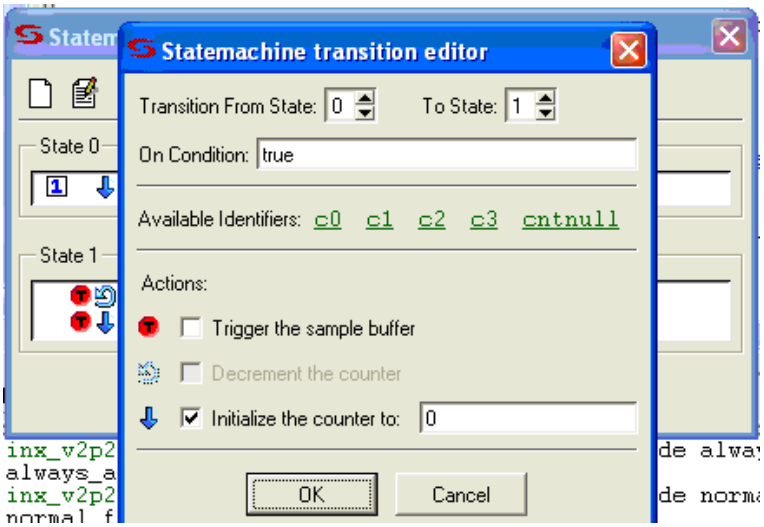

Figure 4: The Identify Statemachine Transition Editor Automates State Machine Triggering

Breakpoint Triggering

Triggering on RTL branching statements (e.g. WHEN or IF statements) allows designers to infer triggers directly from the code itself rather than logic values. The benefit is to reference design debug data closely to the code.

By their nature, netlist products cannot offer breakpoint triggers since triggers are only added to the design after netlist compilation where such constructs no longer exist.

The Identify RTL Debugger does offer breakpoint triggers which are placed adjacent to the branch statements and may be dynamically and individually activated during the debug cycle.

Resource Requirements

The resources required to implement probes and triggers are important in designs that are constrained by resources. In such cases, designers want to know whether the probing resources they add would make the design too large to be implemented on the device.

The probes of netlist debuggers are in octal increments because that is the size of the cores used for probing. Crossing an octal boundary entails the addition of an entire eight-bit core. The requirements are a function of the number of match units added to the trigger port and the complexity of the match type.

When using netlist tools, there is no way to know what resources are actually used to probe the design except by manual calculation because vendor tools do not provide this information. It can be calculated from tables in the user guide, but only for the simplest configuration used for the table.

With the Identify RTL Debugger tool, the resources are added on a bit basis so that only required resources are used. The resources used are displayed dynamically and updated each time a probe, trigger, or breakpoint is added or deleted. The display allows a designer to see the area cost.

Displaying Results

Designers must be able to view the debug results both over lengthy time periods and focus within even a single clock. Results consist of the logic values sampled in the probed nodes in the design usually stored using on-chip memory as a circular buffer. There must be a clear path to relate the results back to the original design. The logic functionality of nodes must match that of the original source code.

As has been described above, netlist nodes may change name or functionality without warning. Names may be designated to be retained intact at the beginning of compilation at the cost of optimization of the design. Those names not so designated cannot be relied upon to survive. Those that can be found in the netlist may function differently. These changes are inherent in netlist debuggers and explain why it can be difficult to relate debug results back to the source code.

In contrast, an RTL debugger retains both names and in probed nodes functionality because the probes themselves require it by being added directly to the source code and prior to compilation.

All debuggers offer a waveform viewer. The Identify RTL Debugger also offers a view of debug results directly in the RTL code for each clock as shown in Figure 5. By scrolling backward and forward in time, users view the logic or enumerated state values posted to the code for each clock.

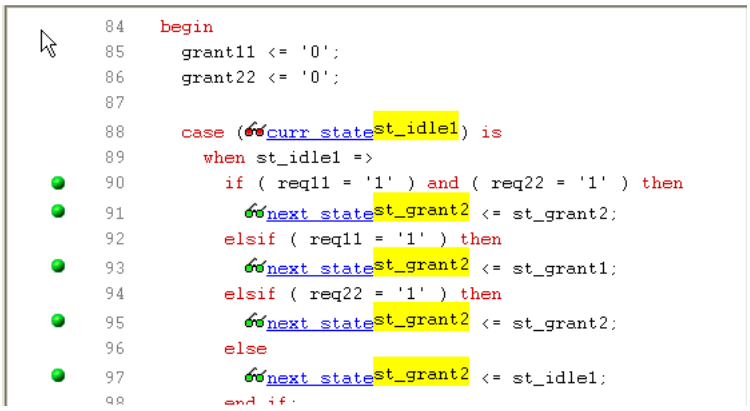

Figure 5: Identify Debugger View of RTL Code for each Clock

Summary

Design verification involves a series of steps—from uncovering basic functional errors down to trapping rare and obscure events in hardware. Different tools are used at each step in the verification process. Simulators find logic errors, logic analyzers find board-level errors, and debuggers capture internal FPGA errors. The most powerful debuggers can be set to trigger on any event or series of events.

Debuggers can be divided into two classes: netlist and RTL. The Identify RTL product from Synplicity brings uniquely powerful and comprehensive capabilities to FPGA debug. The multiple-clock triggering feature allows designers to see events that are likely to remain undetected using any other tool. The sampling modes maximize buffer efficiency. The advanced triggering capabilities are a means for highly sophisticated refinement of data search methods.

The Identify product is a dynamic, in-system debug environment that is unlike any other product available and offers huge productivity gains by allowing designers to debug at the same level they design – in the RTL code.

Synplicity®

Simply Better Results

Synplicity, Inc.
600 W. California Ave, Sunnyvale, CA 94086
Phone: (408) 215-6000, Fax: (408) 222-0264
www.synplicity.com