

Fast, Efficient RTL Debug for Programmable Logic Designs



Current Development and Debug Methodologies

In a typical FPGA design flow, most designers work from a written specification that contains architectural level drawings defining the major logic blocks, interfaces, and busses. The design manager begins to partition functionality based on the diagrams and to assign development based on the block's functional descriptions. Each block is coded individually and may be simulated in a block-specific testbench. The team assembles the blocks into a device-level file where the ports are pins on the target device. The design is then ready to be compiled for simulation initiating the debug phase of development: simulation followed by hardware debug.

Most simulation is performed using a testbench that is written by the designer. The testbench typically has clocks, reset, and control pins that operate on the design, and the signal assertion timing is specified in the testbench.

Testbenches also usually contain models of the interfaces to other devices that the FPGA will communicate within system operation. Such interfaces could be a memory bus, PCI, or Ethernet port. Some models are created by the same person 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, the interface model can be cut from the testbench provided with an IP product and incorporated into the FPGA testbench design to drive the IP interface.

Simulation for Debugging

Designers use logic simulators to find and correct errors in designs. Frequently the device-level design is embedded in a testbench file. The testbench drives the design inputs to operate it. Generally, simulation is the first phase of debug and must be followed by debug of the design running on hardware.

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, however, shows only the logic behavior. 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, it 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 phase or frequency relationship between the clocks. Designing a testbench that models all the possible phase relationships between the different clocks is difficult and time consuming. Yet many such relationships may drive the device in-system and it must perform correctly in any case. Clearly in-system analysis of physical design implementations is a requirement. The question is: What is the easiest and most efficient means to gather information as to what is actually happening inside a device in the laboratory environment?

Hardware Debugging with Logic Analyzers

Developers perform hardware verification of programmable logic designs running at-speed in the target system by viewing logic behavior under actual conditions. The classic hardware debug methodology involves using a logic analyzer to probe and display signals.

Probing is accomplished by using a set of pins on the PCB that connect to unused pins on the FPGA. The board pins are 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.

This method provides a window into the device in operation, but has a number of shortcomings. First, the designer must instrument the design manually and iterate the debug by manual editing each time. Any nodes they seek to view that are not at the device level of the design must be routed to the top-level. 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 may interfere with device operation or timing.

Debugging with a logic analyzer is time consuming, limited, and awkward.

On-chip Logic Analyzer Debugging Tools

A number of programmable logic vendors and others offer tools for hardware debugging using the programming channel to access device operation information. The tools use the programming port on the device to connect to the internal nodes. After the device has been programmed and it is operating in the system, the components sample and store behavior of nodes using logic and memory resources on the device. When the components are instantiated, the designer will synthesize the design and use the vendor's tools to place and route the design on a device. These tools include a viewer that accesses the information over the programming port and displays it in the signal waveforms.

This debug process involves iterations of the connection and implementation flow. The process begins with synthesizing cores that are used to connect the design nodes to sample data and the programming port used to convey it back to the PC for viewing.

The vendor tools overcome the limitations imposed by using device and board pins for probing with logic analyzers. The tools also probe with less interference to device logic implementation and timing. However they share some of the same drawbacks as debugging with logic analyzers.

Methods of Probing Designs

This process begins with synthesizing cores used to connect to the design nodes to sample data and the programming port used to convey it back to the PC for viewing.

At least two methods are available for making the connections between the cores and the nodes. In one method, the designer adds cores after synthesizing the design and connects the cores to the nodes. This method is relatively easy, but requires remaking the connections during every iteration because synthesis may eliminate them. This method also requires running synthesis with an option of retaining net names. Otherwise, while mapping, the names of signals and modules can be eliminated or altered. Even forcing the names to be retained does not guarantee that the functionality of the nodes will not be altered by logic compression or other implementation algorithms.

A second method involves editing the source files to add the cores prior to synthesis. The designer edits the design's top-level source file to add a two-component hierarchy. The top-level component connects to the port and to the lower level component, which also connects to the nodes in the design being monitored. The components allow specification of the data display format and setup of trigger conditions using the connections to the

nodes. This method retains probe connections between iterations because they are made prior to synthesis. But because the tools do not support hierarchical insertion of the components, in order to view lower level nodes, the design must be edited manually and signals routed between logic levels so that they can be connected to the probes. This routing is a time consuming process that also distorts the design with changes that are not functionally used.

In a state machine, for example, a state that has no output could not be probed except by adding a dummy output signal as a state output and routing it through the hierarchy to the top-level. The addition of such signals and ports on components or modules merely for connection to probes is a major undertaking that delays debug and does not contribute to the design functionality.

Another problem with the method is that the design cannot be simulated with the instantiated components because they contain library elements that cannot be compiled for simulation.

Under either of the methods the designer still has to manually edit files to make the probe connections.

Maintaining Design Coherency

Multiple instrumentations may be required on designs where different engineers may be debugging separate modules at the same time. In order to keep these conflicting instrumentations from interfering with each other, different copies of the design for each instrumentation or set of probes may be needed. If, for example, each designer is routing signals up the hierarchy to connect to the probe, the routings of different signals may have to pass through the same blocks.

When managing multiple copies design coherency becomes an issue because fixes to one version must be added to all.

Preserving Logic Integrity

When using vendor debuggers to probe or trigger on nodes the functionality or polarity of the node can change during synthesis optimization even when the node name is retained. The result is false triggering or data. The change stems from the addition of probes at the gate level rather than at the RTL level.

Interpreting Results

Logic analyzers and on-chip logic analyzer tools both display waveforms showing logic level changes. The waveforms must be related back to the original source code to understand device behavior in the context of the original text-based design.

The waveforms only show a snapshot in time of behavior and must be organized in order to see signals whose behavior are antecedents to a resulting signal change. Preparing the probing and waveform display to provide a meaningful result requires careful preparation. Unless all the signals in the code that could affect another are shown in the waveform, you may not know which caused the change.

Iteration Time

Both logic analyzers and on-chip logic analyzer tools require lengthy iterations when moving the probes between nodes because the debug methodology is not tied to the implementation tools. With each pass synthesis and place and route must be run before returning to the lab. While these tools are running, the debug team is idle until the device is ready for reprogramming.

What designers need is a suite of tools that work with a synthesizer in an environment where nodes can be quickly located within a large design and appropriately tagged for monitoring. They need results that are easily viewed and easily related back to the source code, and design iterations swiftly implemented so that the debug team can fully concentrate on the design.

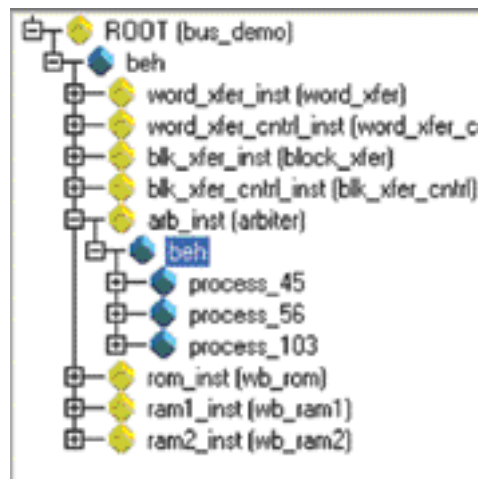
The Identify[®]RTL Debugger Flow

The Identify RTL Debugger from Synopsys' Synplicity Business Group is a hardware debug solution that addresses many of the shortcomings of existing solutions through the introduction of new and unique features.

The Identify product consists of two tools: the Identify Instrumentor and the Identify Debugger. The Identify Instrumentor is used to insert probes to gather data and control when it is gathered and stored. After a design has been instrumented, it is implemented on a device by synthesizing and place and route. When that is complete, the Debugger is used to examine the results.

Automated Probing

The Identify software uses a graphical UI rather than a text editor to navigate to any level of the design hierarchy and add probes to any node directly in the HDL source code.



Hierarchy Navigator

Direct instrumentation allows changes to the design signal names to be ignored because the probe connections remain. Even if the name should change during implementation the connection remains and is viewed under the original name.

Controlling debug from the source code is one of the key advantages of using the Identify tool, and is the source of many of its powerful features and benefits. The Identify Debugger allows the designer to control and display the results of debug at the level where he originally entered the design. Power features allow complex triggering schemes to be performed.

Resulting data is viewed directly in the RTL source. The annotation can be logic levels or an enumerated type value. Results may also be displayed in any of the numerous supported waveform viewers.

Rather than viewing a single snapshot of the results, the designer may step forward and backward through the operation clock-by-clock and view the state of any node instrumented previously.

The Identify software supports multiple IICE[™] (Intelligent In-Circuit Emulator) instances implemented using logic resources that are triggered by different and asynchronous clocks to monitor nodes in different clock domains.

The Identify RTL Instrumentor compiles the entire design for probing automatically. In the compile process the design is copied to a directory and all probe points in the design are detected and displayed in a window. The signals are marked graphically as potential probe points.

A side text window displays the entire design hierarchy, which is navigated on the basis of components, modules, processes, and IF statements that are numbered by their line in the source files, making it easy to find and display any part of the design. Selecting a point in the hierarchy viewer displays the text in the adjacent window.

From the HDL view of the design, watchpoints or breakpoints may be added to the instrumentation. The mouse menu is used to select each node as a probe, trigger, both or neither. A signal can also be selected as the sample clock. The design probe type is displayed within the text of the code itself using color markers.

```
50   elsif @clk'event and @clk = '1' then
51     @rval1 <= @rval;
52     @rval2 <= @rval;
53   end if;
54 end process;
55
56 process ( @@next_state, @rval1, @rval2 )
57 begin
58   @rval1 <= '0';
59   @rval2 <= '0';
60
61   case (@@next_state) is
62     when st_idle1 =>
63       if ( @rval1 = '1' ) and ( @rval2 = '1' ) then
64         @next_state <= st_state2;
65       elsif ( @rval1 = '1' ) then
```

Instrumentation of Clock, Nodes, Triggers and Breakpoints

For those points specified, the Identify Instrumentor inserts the probe logic directly from the source code by means of its own compiler. As signals are assigned for probing, an estimation of the resources required to implement the probes is displayed in a transcript window.

In addition to the probes, the Identify Instrumentor inserts a communication module into the design to communicate sampled data over the JTAG port to the waveform viewer on the PC. Together the probing and communication logic make up an IICE, the logic of which is implemented from device resources such as combinatorial, sequential, clock buffers, and memory elements. Probe points may provide data for display or for sample triggering. The designer selects one signal as the clock for the IICE. Multiple IICE instances are supported for multiple clock designs.



Multiple IICE Controller Tabs

After the design is instrumented it is ready to be synthesized and compiled for place and route on the FPGA tools. Following that, the debugger is run to view the activity on the probes in waveform.

Design instrumentation has absolutely no impact on the functional behavior of the design. No part of the design is ever driven by logic inserted through instrumentation, and therefore functional integrity of the instrumented design is guaranteed. The resources required to implement the instrumentation are very nominal, typically ranging from 1-4% of the logic resources.

By probing from the RTL source code instead of from a flattened netlist, the designer receives feedback from the device operation at the same level where he entered the design. There is no need to interpolate from a waveform with a software-assigned net name back to the RTL.

Debugging the Design

Once the design is instrumented and implemented, the Identify Debugger tool is used to examine the design node behavior at speed in system operation. It is in the Identify Debugger tool that the power of the Identify software to extract relevant information from the most complex designs becomes evident.

The Identify Debugger is used to activate breakpoints and watchpoints that control when data is sampled. Watchpoint probes can be moved between nodes. After set up of the debug sample and control signals, the Identify Debugger is used to collect and buffer sample data. Samples are clocked by the source selected during instrumentation.

```
46  #@real1(not sampled) <= '0';  
47  
48  case (#@curr_state#st_idle2) is  
49    when st_idle1 =>  
50      if ( #@real1_1 = '1' ) and ( #@real2_0 = '1' ) then  
51        #@next_state#st_grant1 <= st_grant2;
```

Annotated Binary and Enumerated Type Values

Data samples are displayed in the waveform viewer or directly in the RTL itself. The unique cycle feature is used to advance back or forward in time and the code is re-annotated with the sample values each time the cycle pointer is moved.

The waveform data can be exported to standard VCD files or VHDL models with enumerated type information preserved.

Various Complex Triggering Mechanisms

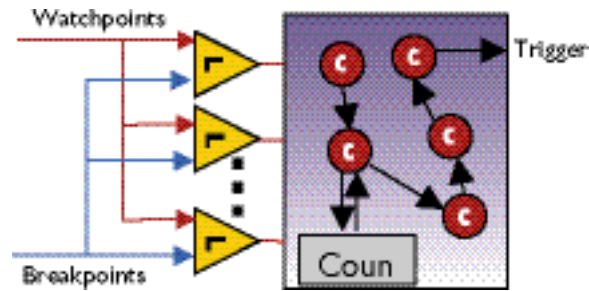
One Identify Debugger trigger mechanism operates on RTL control flow statements such as IF, WHEN, CASE, and others. The statements are specified by the Identify Instrumentor as breakpoints that allow isolation and monitoring of nodes based on events as the code is executed. The breakpoints are displayed in the RTL as buttons immediately below each statement.

The designer can enable or disable the breakpoints dynamically during debug and within a design iteration, allowing isolation of minute sections of code for sampling rapidly even in highly complex designs.

Another trigger mechanism is to use design signals or busses as triggers and/or data samples. If a node is used as a trigger, then the designer has the option of setting a value or transition over which the trigger will cause the values of all the sampled nodes to be buffered. When the logic value of the condition becomes true on the device, the trigger enables sampling. Breakpoints and watchpoint triggers can be used together.

There is also a trigger counter that may be used with the above point triggers in several different ways. The counter may store samples before or after single or multiple trigger events. It can also allow triggering on repeated occurrences of a logic event. It may trigger on a single clock width of the event or multiple widths.

Perhaps the most powerful and flexible form of triggering comes from state machine. The designer can enter the number of states required for state machine triggering during instrumentation, then enter the state inputs and transition conditions during debug using the state machine editor. States may be used or not, and the inputs and conditions adjust dynamically during debug. Sample triggering or the trigger counter can be enabled to increment from any state. The state machine trigger offers complete control over triggering.



Trigger Using Watchpoints, Breakpoints, Counter and State Machine

Rapid Design Iterations

The Identify Debugger automates and vastly accelerates iterative debug by allowing probes to be moved between nodes and then invokes the vendor place and route tools in incremental mode so that only those changes required to route the new probe connections are made. The swift iteration of the design is performed under the control of Identify Debugger and does not require invoking the vendor interface. Incremental compilation allows the designer to return to debugging it right away.

Summary

Simulation is only the first step towards design verification. What is required to ensure design integrity and robustness under actual system conditions is a flexible and powerful hardware debug tool.

Logic analyzers offer one solution, but they are difficult to use and require external pins and time consuming debug iterations. Programmable logic vendor tools are another solution that are easier than a logic analyzer, but these are not optimized to understand device operation back to text-based (HDL) designs

Identify RTL Debugger stands alone as the tool providing the fastest design iterations and the most powerful features available for the debug of programmable logic designs.

SYNOPSYS[®]
Predictable Success

Synopsys, Inc.
Synplicity Business Group
600 West California Avenue
Sunnyvale, CA 94086 USA
www.synplicity.com

Copyright © 2008 Synopsys, Inc. All rights reserved. Specifications subject to change without notice. Synopsys, Synplicity, the Synplicity logo, Identify, and "Simply Better Results" are registered trademarks of Synplicity, Inc. All other names mentioned herein are trademarks or registered trademarks of their respective companies.