Microsemi®

In-Circuit FPGA Debug – Challenges and Solutions

March 2014

Abstract

FPGAs are powerful design elements in embedded systems with many design advantages, but these devices can have complex designs with complex design issues that need to be debugged. Tracking down design issues such as definition errors, system interaction problems, and system timing errors can be a challenge. The inclusion of in-circuit debug capabilities in an FPGA can dramatically improve hardware debug, and avoid countess hours of frustration. This paper describes several different approaches to in-circuit debug for FPGAs, identifies key trade-offs, and through an example design, targeted for a Microsemi SmartFusion®2 SoC FPGA device, will show how new capabilities can be used to speed debug and test.

Introduction

FPGAs are pervasive and powerful design elements and are now found in virtually every embedded system. With increasing capacity, inclusion of complex on-chip functional blocks and advanced serial interfaces these devices can also have complex design problems that need to be debugged. Tracking down issues such as functional definition errors (at the FPGA or system level), functional system interaction problems, system timing issues, and signal fidelity issues between ICs (like noise, crosstalk, or reflections) all become much more complex when using advanced FPGAs. Simulation is certainly a big help in identifying many design problems, but many real world interactions won't show up until the design is implemented in hardware. Several different techniques for debugging complex design issues have been developed to simplify the process. A careful understanding of each of these key techniques, including the various advantages and disadvantages, is useful when considering which technique or combination of techniques is suitable for a particular design.

An example FPGA design, targeted for a Microsemi SmartFusion2 SoC FPGA device, can be used to demonstrate some of the advantages and disadvantages of these standard techniques as well as the newest in-circuit debug capabilities. This illustrative example will show how these various techniques can be used to speed the identification and elimination of hardware problems during hardware debug.

Why is FPGA Debugging a Critical Aspect of System Design and Development?

FPGAs have a variety of use models that differentiate them from other design elements. FPGAs can be used in the production product or can be used as a development vehicle to prove out or prototype a production design concept. When used as the production vehicle, FPGAs can be a much more flexible target than ASIC or CPU-based production vehicles. This is particularly important for a new design, one without much previous design experience. Designs with different architectural options can be easily created and tested so the optimal design is identified. FPGAs with on-chip processors (SoC FPGAs) make it also possible to trade-off CPU-based processing with hardware assisted FPGA-based acceleration functions. These advantages can dramatically reduce design, validation, testing, and failure analysis for new product developments.

When used for prototyping a design, perhaps for a production ASIC, FPGA flexibility is a key benefit. An actual hardware platform, even one that doesn't run at full speed, makes it much easier to obtain detailed system performance metrics, throughput analysis data and architecture proof-of-concept results. FPGA support for hardened implementations of industry standard busses (like PCIe®, Gb Enet, XAUI, USB, CAN, and others) simplifies the testing associated with these interfaces. The newest families of FPGAs with on-chip ARM processors (SoC FPGAs), makes it easy to prototype implementations with embedded processors to. Previously developed processor code can be ported to the prototype and new code created in parallel with the hardware design effort.

This combination of a standard processor with standard interface busses makes it possible to leverage the large ecosystem of available code libraries, drivers, functional APIs, Real Time Operating Systems, and even full Operating Systems to much more quickly create a working prototype. Additionally, once the design is solidified, the FPGA prototype can be used to capture extensive simulation test sets (for both stimulus and response) that reflect actual system data. These data sets can be invaluable in creating the final simulations for an ASIC or other production implementation. The advantages of using an FPGA as a design prototype can dramatically reduce design, validation, testing, and failure analysis for the final product implementation.

In both of these common FPGA use models the flexibility of the FPGA as a design target is a key advantage. This means that many design changes and iterations would be the norm, and thus the ability to rapidly debug design errors would be critical to enabling as many design options as possible. Without an efficient debug capability much of the advantage of FPGA design flexibility will be scarified for the additional debugging time required. Luckily, FPGAs can also provide additional hardware features that dramatically simplify real-time debugging. Prior to looking at these capabilities, let's first look at the most common types of issues an FPGA design might be faced with so we have the proper background to evaluate the efficiency and associated trade-offs of various debugging tools.

Common Issues When Debugging FPGA Designs

Along with the expanded capabilities that modern FPGAs bring, the associated increased complexity makes it more difficult to create error-free designs. In fact, it has been estimated that debugging can take over 50% of the embedded system design cycle. With time-to-market pressures continuing to squeeze the development cycle, hardware debugging of the initial system is relegated to an afterthought—all to often assuming that verification (itself a large percentage of the development schedule), will catch all the bugs prior to initial system bring-up. Let's look at just a few common types of system issues to better understand the challenges a typical design will face during initial system bring-up.

Functional definition errors can be doubly difficult to find since the designer has misunderstood a particular requirement, so the error can be overlooked even when looking carefully at the details of the design. An example of a common functional definition error would be where a state machine transition doesn't end up in the right state. Errors can also show up in system interfaces as an interaction problem. Interface latency, for example, might be incorrectly specified resulting in an unexpected buffer overflow or underflow condition.

System level timing issues are another very common source of design errors. Asynchronous events, in particular, are a common source of errors when synchronization or crossing timing domain effects are not carefully considered. When operating at speed these types of errors can be very problematic and may show up very infrequently, perhaps only when specific data patterns manifest themselves. Many common timing violations fall into this category and are usually very difficult, if not impossible to simulate.

Timing violations can also be the result of low signal fidelity between integrated circuits, in particular in systems with multiple power rails for each circuit. Low signal fidelity can result in signal noise, crosstalk, reflections, excess loading and Electro-Magnetic Interference (EMI) issues that often show up as timing violations. Power supply issues, like transients (in particular during system start-up or shut-down), load variations and high power dissipation stresses can also result in mysterious errors, often not easily traced back to a power supply source. Even when the design is completely correct board fabrication issues can result in errors. Faulty solder joints and improperly attached connectors, for example, can be the source of errors and may even be temperature or board location dependent. The use of advanced FPGA packaging techniques can make it difficult to probe signals on the printed circuit board to, so just getting access to a desired signal can often be problematic. Often many design issues don't create an immediate error and must ripple through the design until the error actually manifests itself. Tracing the starting error back to the root cause can often be a frustrating, difficult and time consuming task.

For example, a single bit wrong in a translation table may not result in an error until many cycles later. Some of the tools we will discuss later in this paper, that use dedicated in-circuit debug hardware, are specifically targeted at making these ‘bug hunts’ quicker and easier. Prior to getting into the details of these tools, let’s first look a popular software-based debugging technique simulation in order to better understand the advantages and disadvantages of using simulation for debugging.

Use of Simulation for Debugging

Many times, the beginning of the design debug process begins with simulation. Applying a wide range of stimulus to the design and checking the expected output against the simulated designs output, is an easy way to catch most obvious design errors. A window showing a typical simulation run is given in Figure 1 below. The clear advantage of simulation verses hardware-based debugging, is that simulation can be done in the software—no actual hardware-based design is needed. Simulation can thus catch many design errors, in particular those associated with incorrect specifications, misunderstanding of interface requirements, function errors, and many other ‘gross’ types of errors that are readily detected through simple tests.

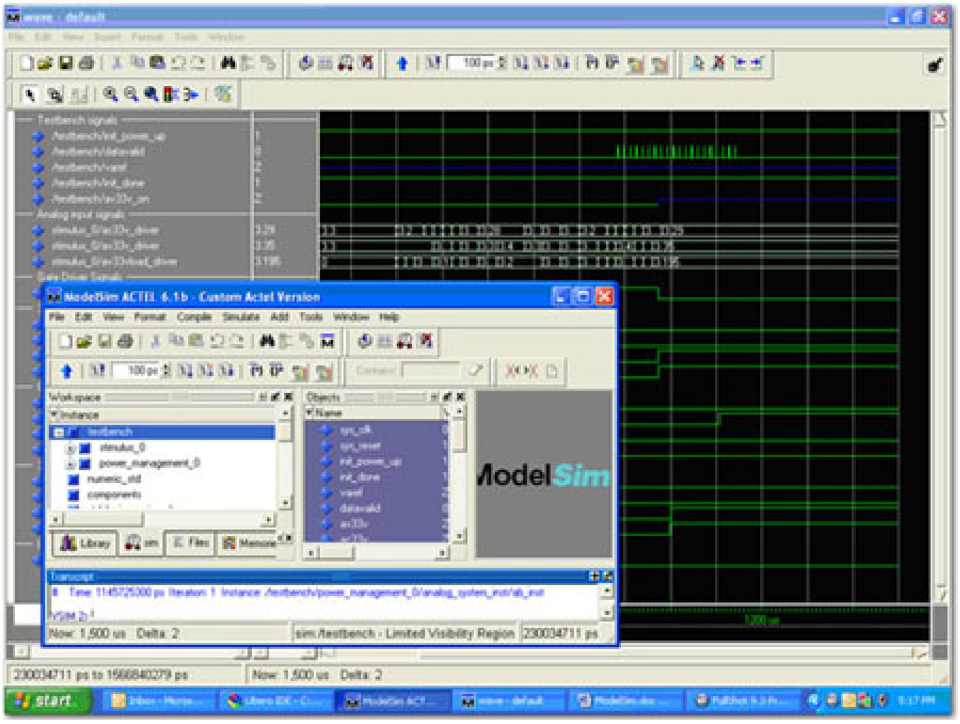

ModelSim ACTEL 6.1b - Custom Actel Version
File Edit View Format Compile Simulate Add Tools Window Help
Workspace
Objects
ModelSim
Transcript
Now: 1,500 us Delta: 2
start

Figure 1: **Typical Simulation Window**

Simulation is particularly effective when extensive stimulus combinations are available to the designer and the resulting outputs are well known. In these cases, simulation can do an almost exhaustive test of a design. Unfortunately, most designs don't have easy access to extensive test suites and the process of creating them can be very time consuming. Creating a test suite that covers 100% of the design is virtually impossible for large FPGA-based designs and short cuts must be used to try and cover the key elements of the design. Another difficulty with simulation, is that it isn't a ‘real world’ implementation and can’t catch asynchronous events, at-speed system interactions, or timing violations. Finally, the simulation process can be very slow and if many iterations are required simulation quickly becomes the most time consuming, and often the most costly portion of the development process.

As an alternative (or perhaps better stated, as an addition to simulation) FPGA designers found that they could add debug hardware into the FPGA design in order to observe and control key signals within the device. These techniques originally developed as ad-hoc approaches, but have gradually developed into a standard hardware debug strategy. This use of in-circuit debug capabilities offers significant advantages for FPGA-based designs and the next section will explore the three most common strategies and their various advantages and disadvantages.

Common In-Circuit Debug Approaches for FPGAs

The most common techniques to implementing in-circuit debug capabilities in FPGAs use either an embedded logic analyzer, external test equipment, or dedicated signal probe hardware embedded within the FPGA fabric. The embedded logic analyzer is typically implemented using FPGA fabric and is inserted into the design. The JTAG port is used to access the analyzer and the captured data can be displayed on a PC. When external test equipment is used, the FPGA design under test is modified so that selected internal FPGA signals are routed to output pins. These pins can then be observed through the external test equipment. When dedicated signal probe hardware is used, a wide selection of internal signals can be read in real time. Some probe implementations can even be used to write to register or memory locations further enhancing debug capabilities. Let’s look in more detail at the advantages and disadvantages of each of these techniques and then look at an example design to see how these different approaches can impact overall debugging time.

In-Circuit FPGA Debug-Embedded Logic Analyzer

The concept of the embedded logic analyzer was a direct result of the ad-hoc in-circuit debugging capabilities that designers implemented when FPGAs were first used. Embedded logic analyzers added new capabilities and eliminated the requirement for the designer to develop their own analyzer. Most FPGAs offer these capabilities and third parties offer standard analyzers (Identify®, from Synopsys, is one popular example) that can easily interface with higher level tools to further improve productivity.

The logic analyzer functionality is inserted into the design, using FPGA fabric and embedded memory blocks as trace buffers, as illustrated in Figure 2. Triggering resources are also created so that complex signal interactions can easily be selected and captured. Access to the analyzer for control and data transfer is typically done through the standard JTAG port to simplify interface requirements. Captured data can be displayed on a PC using common viewing software and typically mirrors a logic simulator waveform output viewing style.

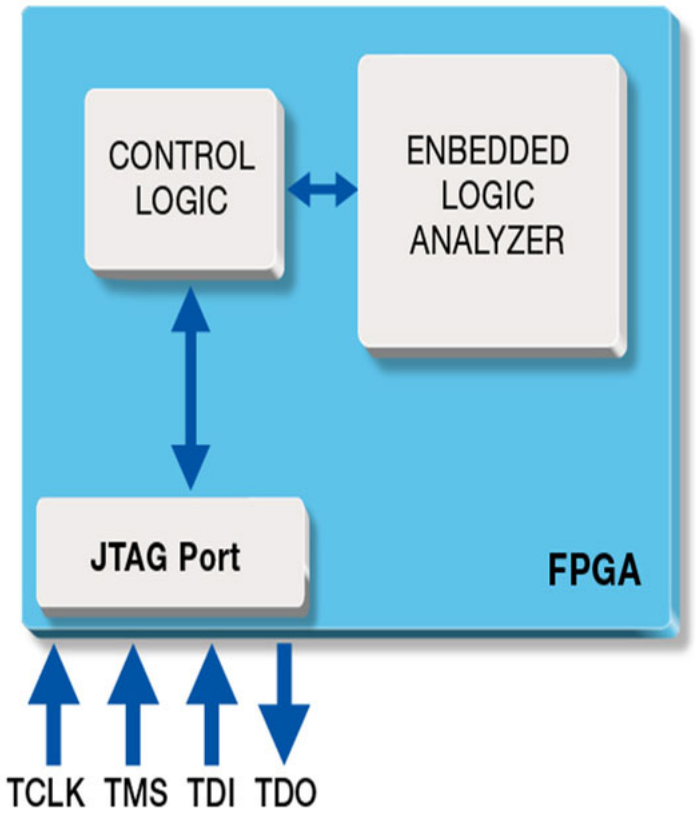


Figure 2: **In-Circuit FPGA Debug-Embedded Logic Analyzer Block Diagram**

The advantages of this approach are that no additional FPGA output pins are used, just the standard JTAG signals. The embedded logic analyzer IP cores are usually relatively inexpensive and in some cases can be an option to existing FPGA synthesis, or simulation tools. In some cases, the embedded logic analyzer can also provide additional outputs on unused I/Os, if it is more convenient. One of the disadvantages to this approach is that a large amount of FPGA resources are required. In particular, if trace buffers are used this will reduce the number of block memories available. If a wide buffer is needed this will also be a trade-off against memory depth (since the use of a wider memory results in shallower memory depth)—a big disadvantage when using smaller devices. Perhaps the largest drawback to this technique is that every time an adjustment to probe placement is made, it is necessary to recompile and reprogram the design. When using a large device this process can take a significant amount of time. Due to the way the signal probes are placed in the design it can be difficult to correlate signal timing relationships. Additionally, the delays between signal probes are not consistent and thus timing relationships are difficult to compare. This is a particular difficulty when comparing asynchronous signals or signals from different time domains.

In-Circuit FPGA Debug – External Test Equipment

The use of in-circuit debug code in conjunction with external test equipment was a natural development when an external logic analyzer was already available for system testing. By creating some simple debug code to identify and select internal test signals and apply them to FPGA I/Os, as shown in Figure 3, it was possible to leverage the analyzers advanced capabilities (such as large trace buffers, complex triggering sequences, and multiple viewing options) to create simple yet powerful debug environments. More complex in-circuit capabilities for advanced triggering options can minimize the number of outputs needed. For example, selecting specific addresses on a wide bus might be prohibitive if external pins were required.

Using internal FPGA logic dramatically reduces I/O requirements and can even look for specific address patterns (perhaps a call and return sequence) for debugging more complex problems. If a common user interface is available, this can simplify the learning curve and improves productivity.

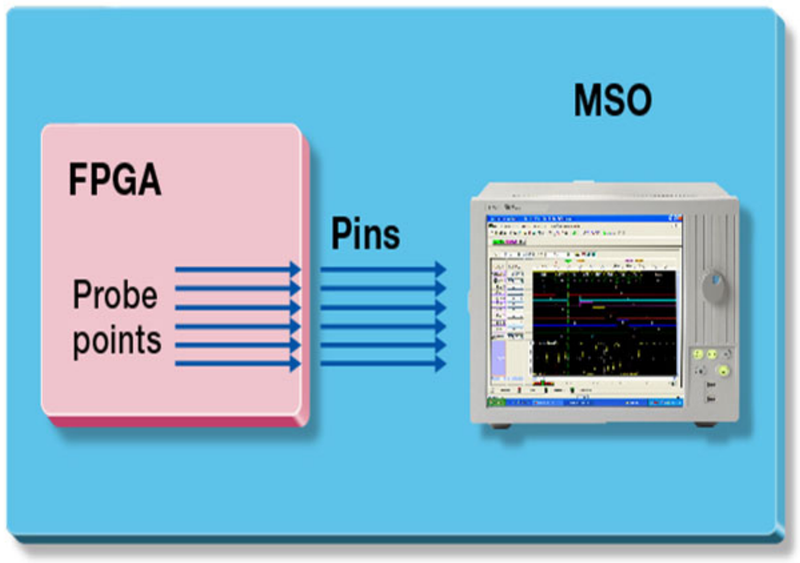


Figure 3: **In-Circuit FPGA Debug-External Test Equipment Block Diagram**

The advantages of this approach is that it leverages the cost of the external test equipment and thus there is no added tool cost. Some debug circuit IP cores are available from equipment manufacturers or FPGA manufacturers, and can be very low cost or even free. The amount of FPGA resources required to implement the signal selection logic is very small, and since the trace function is done using the external logic analyzer, no block memories are needed. Since selection logic is inexpensive, a large number of channels with wide triggering can be supported too. The logic analyzer can operate in both a Timing mode and a State mode which helps isolate some timing issues.

The disadvantages of this approach can include the need to purchase a logic analyzer, if one isn't already allocated to the project. This disadvantage may be enough to discourage this approach in many instances. Note however, that some low-cost logic analyzer options are becoming available that use the PC or a tablet for display, making this option much more cost-effective for simple debug requirements.

The number of FPGA pins consumed can be another disadvantage and if wide busses need to be observed, significant planning for board layout and the addition of debug connectors is needed. This requirement is most times difficult to predict early in the design phase and another unwanted complexity. Similar to the embedded logic analyzer approach the external test strategy requires recompiling and reprogramming of a design, when each new experiment is needed.

The common disadvantages of these two techniques—the use of on-chip resources (which can also impact the design's timing performance and create additional debugging requirements) the need to recompile and reprogram the design (which can add hours or even days to the debug schedule) the up-front planning required for identifying likely test scenarios, and the use of additional chip I/O resources created a need for an approach without these drawbacks. One response was the addition of dedicated debug logic into the FPGA fabric on some devices. In-circuit debug using hardware probes was the result.

In-Circuit FPGA Debug – Hardware Probes

The use of hardware probes dramatically simplifies in-circuit debug techniques for FPGAs. This technique implemented as a Live Probe feature on SmartFusion2®SoC FPGA and IGLOO®2 FPGA devices, adds dedicated probe lines to the FPGA fabric to observe the output of any logic element register bit. As shown in the block diagram in Figure 4, hardware probes are available in two probe channels A and B.

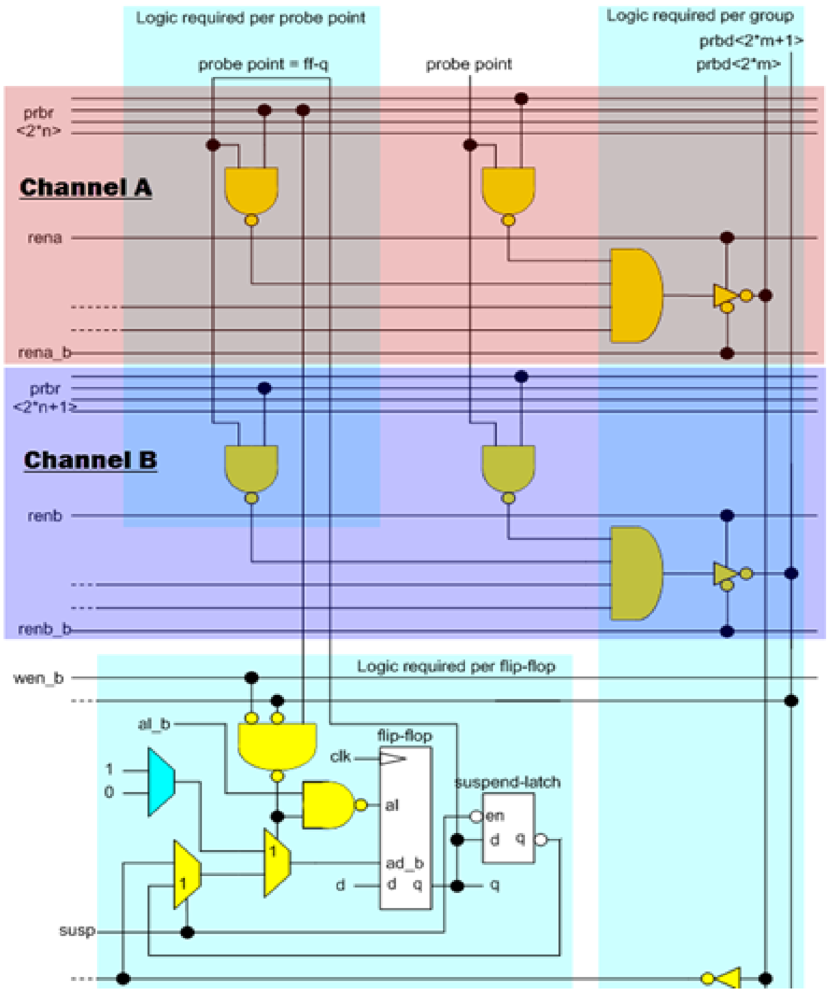


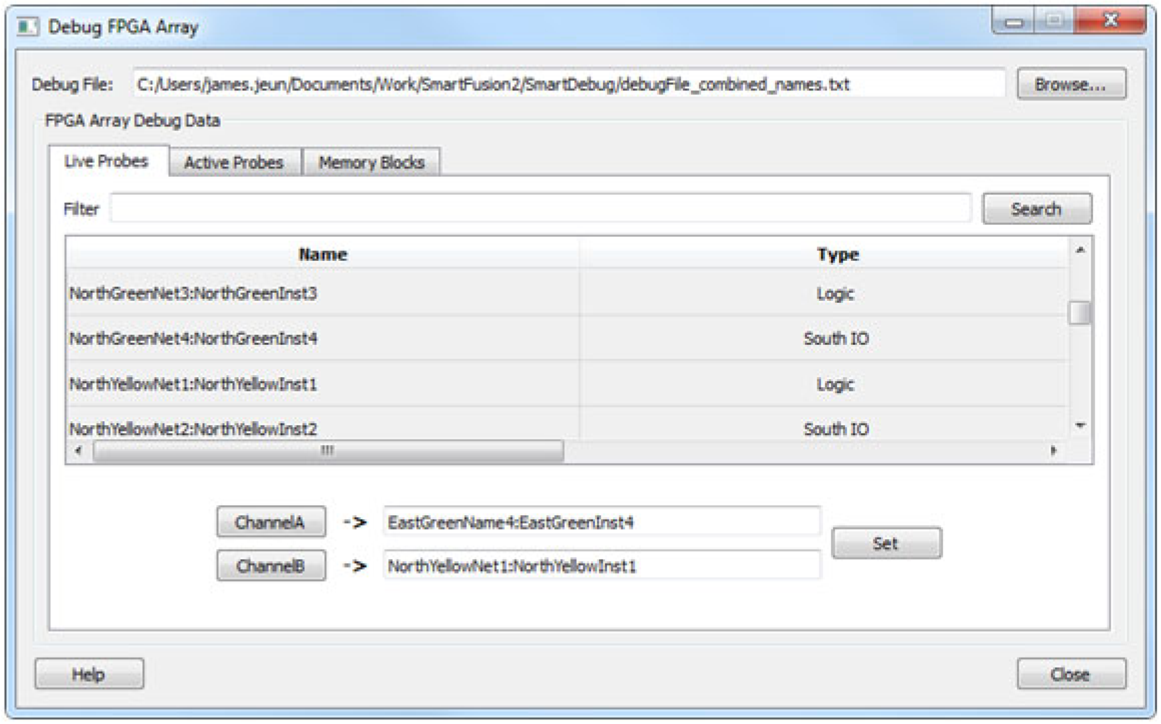
Figure 4: **Diagram of Probe Circuits**

Selected register outputs (probe points), like the one sourced at the bottom of the figure, are routed above the two probe channels and if selected can be applied to either the A or B channel. These real-time channel signals can then be sent to dedicated Probe A and Probe B pins on the device. The Probe A and Probe B signals can also be internally routed to an embedded logic analyzer.

Note that the timing characteristics of the probe pins are regular and have negligible deviation from one probe point to another, making it much easier to compare timing characteristics of the real-time signals. Data can be captured at up to 100MHz making it appropriate for the majority of target designs.

Perhaps most importantly the probe point locations, since they aren't selected as part of the implemented design (they're selected through dedicated hardware) can be quickly changed by simply sending the selection data to the device. No design recompile and reprogramming is needed

To simplify the use of the Live Probe capability even more, the associated debug software tool has access to all the probe signal locations through an automatically generated debug file. As shown in Figure 5, the signal name can be selected from the signal list and applied to the desired channel. This can be done even while the design is running so that probing activity within the design is seamless and very efficient.



Debug File: C:/Users/james.jeun/Documents/Work/SmartFusion2/SmartDebug/debugFile_combined_names.txt

Name	Type
NorthGreenNet3:NorthGreenInst3	Logic
NorthGreenNet4:NorthGreenInst4	South IO
NorthYellowNet1:NorthYellowInst1	Logic
NorthYellowNet2:NorthYellowInst2	South IO

ChannelA -> EastGreenName4:EastGreenInst4

ChannelB -> NorthYellowNet1:NorthYellowInst1

Figure 5: **Debug Set-Up Screen Shot**

In many cases, the hardware probe capability, like Live Probe, can be used in conjunction with the previously described embedded logic analyzer and the external test techniques.

As shown in Figure 6, the Live Probe capability to select signals 'on the fly' makes it possible to quickly and easily change the signals under observation without needing to recompile the design. An external logic analyzer or scope can easily observe the probed signals, as illustrated in the top right part of the figure on the dedicated probe output pins. Alternatively (or maybe even in addition to) the internal logic analyzer (the ILA Identify block, shown in the figure) can be used to observe the probe pins. The probe signals can be captured by the ILA and observed on the waveform window. Probe locations can be changed without the need to recompile the target design.

Note that the additional capabilities for triggering and trace can be used to enhance probe functionality, making it easy to spot even complex design issues.

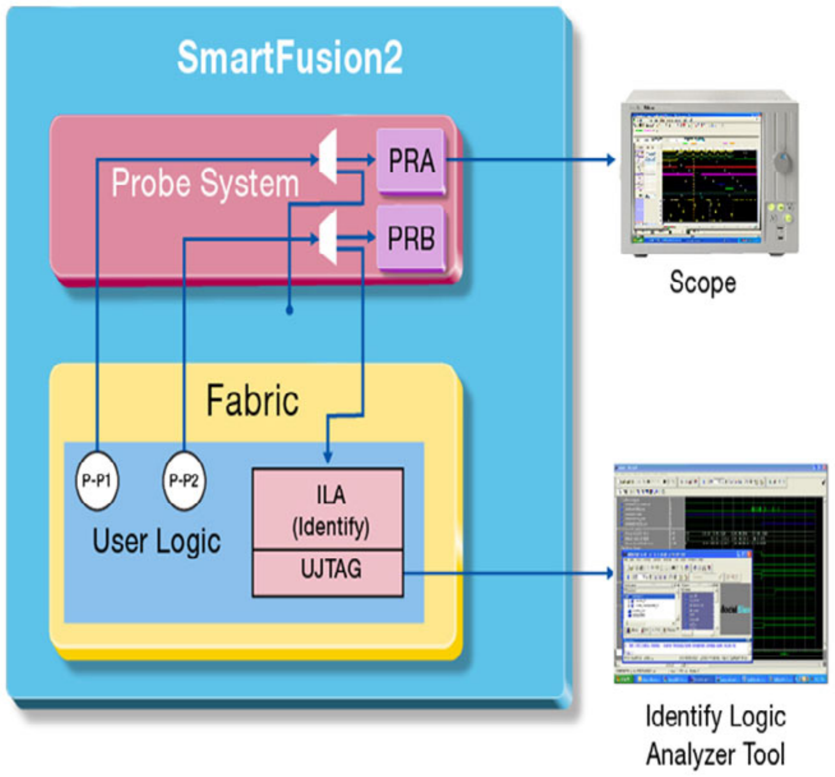
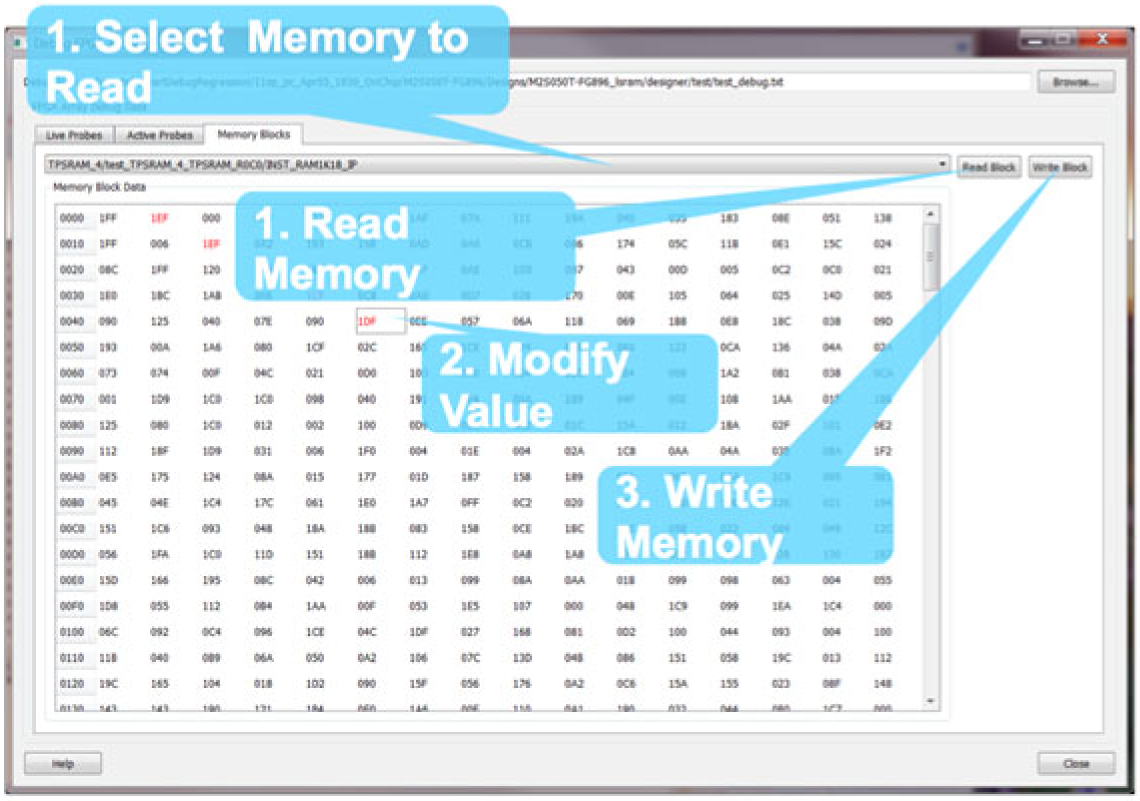


Figure 6: **Live Probe Data Output Options Diagram**

Additional hardware debug capabilities are also available on the SmartFusion2 SoC FPGA and IGLOO2 FPGA devices. One of these capabilities, called Active Probe, can dynamically and asynchronously read or write to any logic element register bit. A written value persists for a single clock cycle so normal operation can continue, making it a very valuable debugging tool. Active Probe is of particular interest if a quick observation of an internal signal is desired (perhaps simply to check that it is active or in the desired state, like a reset signal), or if there is a need to quickly test a logic function by writing to a probe point (perhaps to initiate a state machine transition by quickly setting an input value to isolate a control flow problem).

Another debug capability provided by Microsemi is Memory Debug. This feature allows the designer to dynamically and asynchronously read or write to a selected FPGA fabric SRAM block. As illustrated in the screen shot of the Debug Tool (Figure 7), when the Memory Blocks tab is selected the user can select the desired memory to read, execute a snapshot capture of the memory, modify memory values, and then write the values back to the device. This can be particularly useful for checking or setting data buffers used in communications ports for computation oriented scratch-pad or even for code executed by an embedded CPU. Debugging complex data dependent errors is significantly quicker and easier when memories can be observed and controlled so quickly.



.../M2S050T-FG896_lsram/designer/test/test_debug.txt

TPSRAM_4/test_TPSRAM_4_TPSRAM_R0C0/INST_RAM1K18_IP

Memory Block Data

0000	1FF	1EF	000	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	183	08E	051	138
0010	1FF	006	1EF	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	174	05C	118	0E1	15C	024
0020	08C	1FF	120	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	043	00D	005	0C2	0C8	021
0030	1E0	1BC	1A8	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	170	00E	105	064	025	14D	005
0040	090	125	040	07E	090	1DF	0EE	057	06A	118	069	188	0E8	18C	038	09D
0050	193	00A	1A6	080	1CF	02C	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0CA	136	04A	[illegible]
0060	073	074	00F	04C	021	0D0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1A2	081	038	[illegible]
0070	001	1D9	1C0	1C0	098	040	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	108	1AA	[illegible]	[illegible]
0080	125	080	1C0	012	002	100	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	18A	02F	[illegible]	0E2
0090	112	18F	1D9	031	006	1F0	004	01E	004	02A	1C8	0AA	04A	[illegible]	[illegible]	1F2
00A0	0E5	175	124	08A	015	177	01D	187	158	189	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
00B0	045	04E	1C4	17C	061	1E0	1A7	0FF	0C2	020	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
00C0	151	1C6	093	048	18A	188	083	158	0CE	18C	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
00D0	056	1FA	1C0	11D	151	188	112	1E8	0A8	1A8	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
00E0	15D	166	195	08C	042	006	013	099	08A	0AA	018	099	098	063	004	055
00F0	1D8	055	112	084	1AA	00F	053	1E5	107	000	048	1C9	099	1EA	1C4	000
0100	06C	092	0C4	096	1CE	04C	1DF	027	168	081	002	100	044	093	004	100
0110	118	040	089	06A	050	0A2	106	07C	130	048	086	151	058	19C	013	112
0120	19C	165	104	018	102	090	15F	056	176	0A2	0C6	15A	155	023	08F	148
0130	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

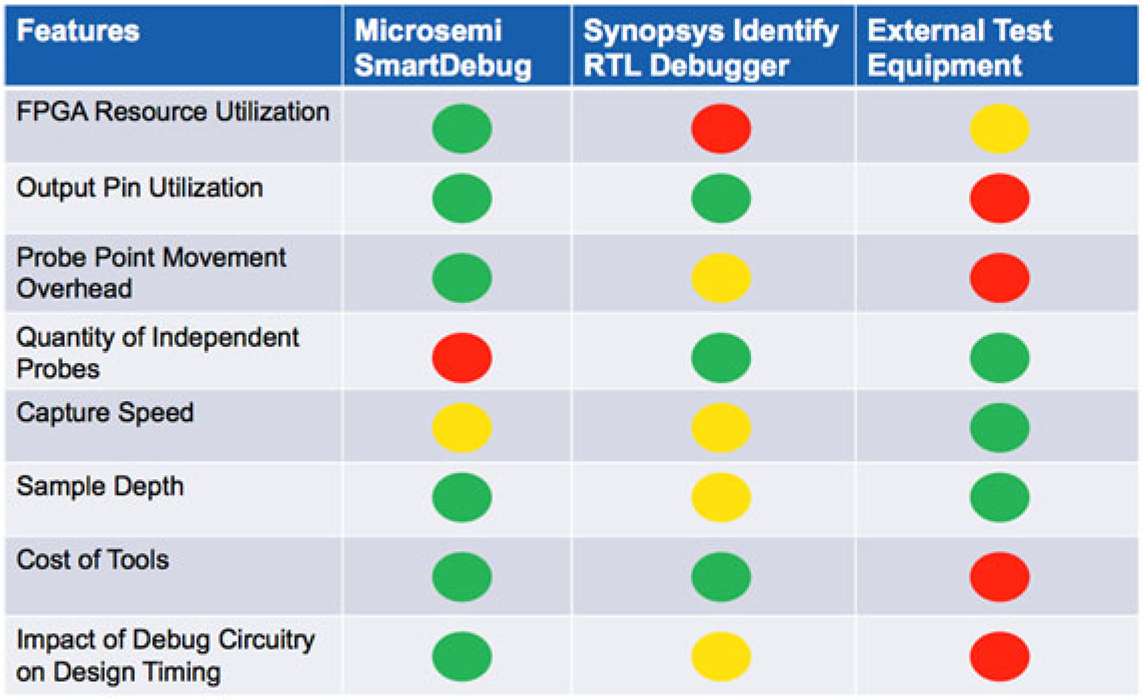
Figure 7: **Annoted Memory Debug Tool Screen Shot**

Once a design is debugged it may be desirable to turn off the hardware debug capabilities to protect sensitive information. An attacker could use these same facilities to read-out critical information or change system settings that could allow easy access to sensitive portions of the system. Microsemi has added features to allow the designer to secure the device after debugging is completed. For example, access to Live Probe and Active Probe can be locked to completely disable the function as a possible means of attack (it even eliminates the possibility of probe activity creating any patterns in the supply current which could be used to try and observe probe data indirectly). Alternatively, access to selected portions of the design can be locked out to prevent access to just those sections. This can be convenient if only a portion of the design needs to be secure making the rest of the design still accessible for in field testing or error analysis.

In-Circuit Debug Comparison Chart

Now that a detailed review of the three main in-circuit hardware debug techniques have been described a summary chart, as shown in Figure 8, has been created that details the various advantages and disadvantages of each method. Remembering that some techniques can be used in conjunction (Live Probe and Internal Logic Analyzer (ILA), like Synopsys Identify, for example), we can see the key strengths and weaknesses of each technique. The collection of in-circuit hardware debug capabilities (Live Probe, Active Probe, and Memory Debug—collectively called SmartDebug), are weakest in comparison to the other techniques when it come to the number of total probes available (a red circle) and are weaker than the best (yellow circle) when the capture speed is considered (external test equipment can be faster).

ILA-based techniques, like Synopsys Identify, are weakest when compared to the other techniques and when FPGA resource requirements are considered. External test equipment-based techniques are weakest over a number of considerations with cost, design timing impact, and probe movement overhead (due to the need to recompile the design) the most onerous. Perhaps the optimal solution is a combination of SmartDebug and one of the other techniques, so that the number of channels weakness of SmartDebug can be mitigated and the probe point movement disadvantages of the other techniques reduced as well.



Features	Microsemi SmartDebug	Synopsys Identify RTL Debugger	External Test Equipment
FPGA Resource Utilization	🟢	🔴	🟡
Output Pin Utilization	🟢	🟢	🔴
Probe Point Movement Overhead	🟢	🟡	🔴
Quantity of Independent Probes	🔴	🟢	🟢
Capture Speed	🟡	🟡	🟢
Sample Depth	🟢	🟡	🟢
Cost of Tools	🟢	🟢	🔴
Impact of Debug Circuitry on Design Timing	🟢	🟡	🔴

Figure 8: **In-Circuit FPGA Debug Methods Comparison Chart**

Simple Debug Use Case

Now that we have a better understanding of the various in-circuit debug options, let's look at a simple design example to see how these techniques perform. Figure 9, shows a simple FPGA design in a SmartFusion2 SoC FPGA device. The Microcontroller Subsystem (MSS) is reset by the CoreSF2Reset Soft IP block. The inputs to this block are the Power On Reset, a User Fabric Reset, and an External Reset. The outputs are a reset to the User Fabric, an MSS reset, and an M3 reset. The error symptoms are that there is no activity on the I/Os even though the device exits the POR state successfully. The three different options for debugging this error are illustrated in the figure as well: The blue box (labeled ETE) is for the External Test Equipment method; the green box (labeled ILA) is for the Internal Logic Analyzer method; and the orange box (labeled AP) is for the Active Probe method. We will assume the potential root causes of the error are improperly asserted reset inputs to the CoreSF2Reset Soft IP block.

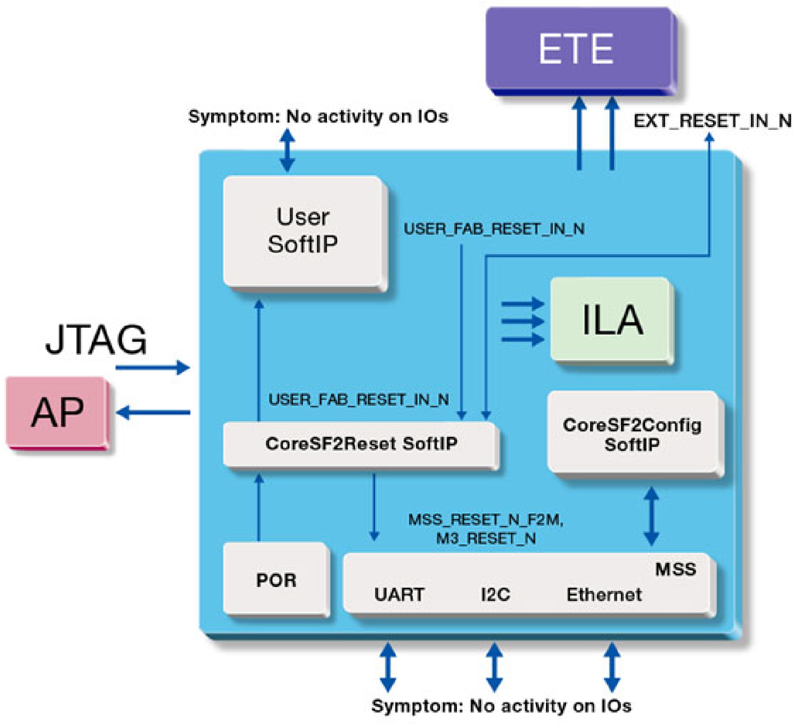


Figure 9: **Example Design with a Reset Error**

Let’s now look at the debug process for three of the previously described in-circuit methods.

External Test Equipment

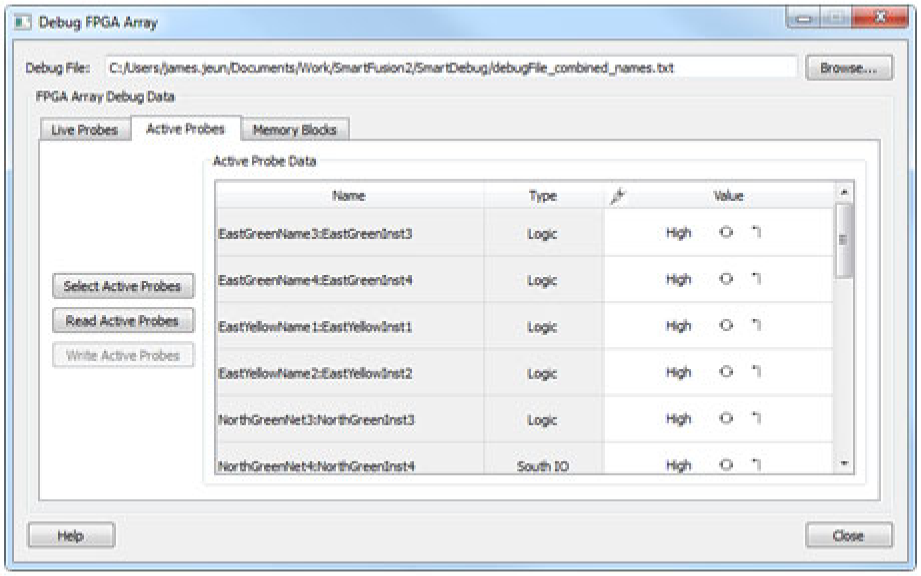
Using this method, it is assumed that the test equipment is available and not being used by a higher priority project. Additionally, it is important to have planned ahead so that some FPGA I/Os are available and can be easily connected to the test equipment. Having a header on the PCB for example, would be very helpful and minimize time spent trying to identify and connect to a ‘likely suspect’ or the potential shorting of pins during probing. The design will need to be recompiled to select the signals we want to investigate. Hopefully, we won’t be ‘peeling back the onion’ and need to select additional signals for further investigation, since often our initial investigation just results in more questions. In any event, the recompile and reprogramming process can take a significant amount of time, and if it results in timing violations a redesign is required (we are all familiar with how frustrating trying to solve timing closure issues can be, in particular, when you are making the design changes to find a design bug—the entire process can take from minutes to hours)!

Internal Logic Analyzer

Using this method the ILA must be inserted into the design using fabric resources, and then needs to be recompiled. Note that if the ILA has already been instantiated, the signals we want to investigate may not have been instrumented, which would also require a recompile. This process risks changing the original design and violating timing constraints. If timing is met, the design needs to be reprogrammed and reinitialized. This whole process can take several minutes or even hours if recompile times are long and multiple passes are needed.

Active Probe

Using this method the Active Probe can be pointed to the source of the various reset signals, all of which are sourced by register outputs (as is common in any good digital design practice). The signals are selected one at a time, from an Active Probe menu shown in Figure 10 below. The selected signal values can be read and are displayed on the Active Probe data window. Any mis-assertions easily identified. This test can be done immediately without the need to recompile and reprogram the device. The whole process takes just a few seconds.



Debug File: C:/Users/james.jeun/Documents/Work/SmartFusion2/SmartDebug/debugFile_combined_names.txt

FPGA Array Debug Data

Active Probe Data

Name	Type	Value
EastGreenName3:EastGreenInst3	Logic	High
EastGreenName4:EastGreenInst4	Logic	High
EastYellowName1:EastYellowInst1	Logic	High
EastYellowName2:EastYellowInst2	Logic	High
NorthGreenNet3:NorthGreenInst3	Logic	High
NorthGreenNet4:NorthGreenInst4	South IO	High

Figure 10: **Use of Active Probe to Read Signal Values**

More Complex Debug Use Case

The above design was very simple and is useful as an introduction to using the described design techniques, but a more complex example might be even more illustrative. Many times the signal of interest isn't a static signal as it was in our simple example but is dynamic. A common dynamic signal is an intermediate clock, perhaps used for timing a handshake for a serial interface. Figure 11 shows such a design with the user Soft IP core, in this case, a custom serial interface connected to the system APB bus. The errors symptoms are that there is no activity on the users custom serial interface, and that when an APB bus master issues a transaction to access the serial interface it goes into an exception condition indicating an incorrect handshake. These conditions seem to rule out a static cause, like an incorrect reset signal, since the transaction state machine seems to not be operating at the rate expected and thus causes the exception. The root cause is thought to be the clock frequency generator within the user IP core. If it is not running at the correct frequency the described errors would result.

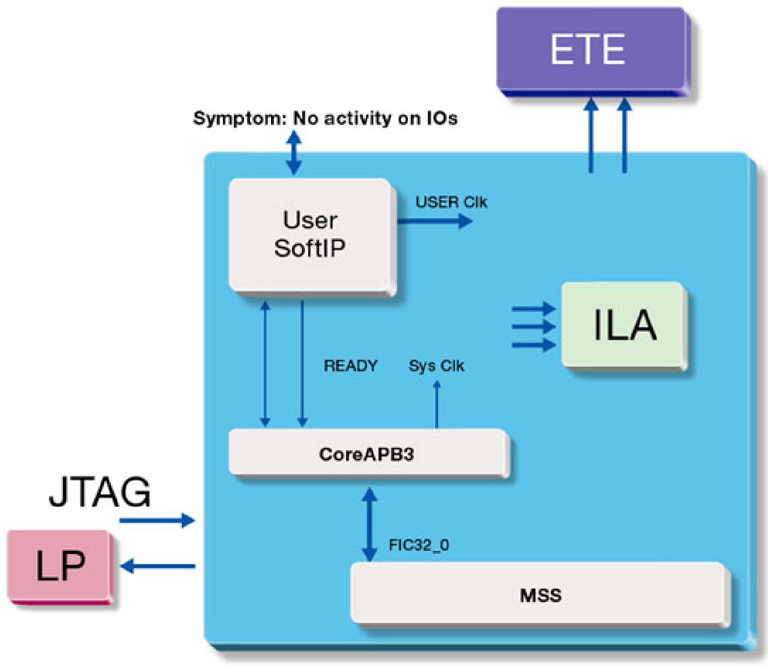


Figure 11: **Dynamic Signal Debug Example**

In this situation it is probably a better strategy to replace the Active Probe approach with the Live Probe. This is illustrated in the above figure by the orange colored LP box, using the JTAG signal for the probe source selection.

External Test Equipment

For this case, the methodology is very similar to the previously described simple example. The user clock signal is brought out to the test point (hopefully on a header) and a time consuming recompile is needed. It may also be helpful to bring out a reference signal, perhaps a system clock that is used to clock the users IP as a comparison signal. We will again be subjected to the need to recompile and reprogram so the entire process could take a significant amount of time.

Internal Logic Analyzer

This case is very similar to the simple example. The ILA must be inserted, or the desired signal defined, and a recompile and reprogram cycle executed. All the previously described issues still result in a significant debug cycle time. There is an additional complexity, however. The clock that drives the ILA needs to be synchronous, and ideally much faster with respect to the clock to be observed from the user Soft IP core. If these clocks are asynchronous, or don't have the correct timing relationships, data capture will be unpredictable and a possible source of confusion for the debug process.

Note that if the user Soft IP clock is not generated on-chip (perhaps it is recovered from the serial interface) the designer may need to add a clock module to generate a faster ILA clock using additional resources and possibly creating a timing violation.

Live Probe

Using this method, the Live Probe can be quickly pointed to the source of the user clock and any other clock source from a register to chase down the root cause of the error. The Live Probe will show the selected signal outputs in real time and any timing relationship between the signals is thus much easier to determine. The whole process takes just a few seconds.

Other Debug Features for Serial Interfaces

It is also important to point out that there are many additional debug capabilities in SmartFusion2 SoC FPGA and IGLOO2 FPGA devices that can be used on serial interfaces, like the one in the previous example design where errors are even more complicated. SERDES Debug, for example, provides specific debug capabilities for the dedicated high-speed serial interfaces. Some of the SERDES Debug features include PMA test support (like PRBS pattern generation and loopback testing) support for multiple SERDES test configurations with register-level reconfiguration to avoid the use of the full design flow to make configuration changes, and text reports showing configured protocols, SERDES configuration registers, and Lane configuration registers. These features make SERDES debug much easier and can be used in conjunction with Live Probe and Active Probe to further speed debugging of complex circuits.

The previously described Memory Debug tool can also be used in conjunction with SERDES Debug to speed testing. Since memory buffers can be quickly and easily inspected and changed with Memory Debug, it is possible to quickly create ’test packets’ and observe loopback or inter-system communications results. The designer can leverage these capabilities and thus minimize the need for specialized ‘test harnesses’ that consume additional FPGA fabric and that might impact chip timing.

Conclusion

This paper has described in detail several different approaches to implementing in-circuit debug for FPGAs and SoC FPGAs—the use of an Integrated Logic Analyzer, the use of external test equipment, and the use of dedicated probe circuits integrated into the FPGA fabric. The addition of specialized and dedicated probe circuits, like Active Probe and Live Probe offered by Microsemi on SmartFusion2 SoC FPGA and IGLOO2 FPGA devices, was shown to significantly speed and simplifies the debug process. The ability to quickly modify the selection of internal signals (without the need to execute a very time consuming recompile), and the ability to probe internal signals (without the need to use FPGA fabric and potentially introduce timing violations) were shown to be major advantages when debugging FPGA designs. Additionally, the use of multiple methodologies, which can work together to provide an even more comprehensive debug capability was described. Finally, two example debug use cases were given to illustrate the trade-offs between the described methods.arnesses’ that consume additional FPGA fabric and that might impact chip timing.

To Learn More

1. IGLOO2 FPGAs
2. SmartFusion2 SoC FPGAs

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo CA 92656 USA
Within the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996

55900178-0/3.14