Not Just for Hardware Debug: Prototype Debuggers for System Validation and Optimization

Michael Sachtjen, Mentor Graphics, 613-963-1069, Michael_Sachtjen@mentor.com
Joe Gaubatz, Mentor Graphics, 503-685-4811, Joseph_Gaubatz@mentor.com

Simulation and emulation are great for finding and fixing hardware bugs. Software debuggers are great for finding and fixing general purpose software bugs. But traditional debugging tools often fall short in finding and fixing system-level bugs in complex SOC designs. In a series of short case studies, this presentation will explore how the use of embedded debug instrumentation in modern FPGA prototype debuggers can meet the emerging need for system-level debug.

The adoption a commercial silicon debug solution transformed the ways our teams approached debug and hardware/software co-verification. We hope that the lessons learned from these case studies can be of use to other design and verification teams. We will be presenting the reasons why our teams initially adopted the debug solution we did, how we deployed that solution to maximize its usefulness and why we found our solutions to be useful far beyond the original bugs we were tasked to find. The bugs and use models presented are intentionally kept a high level of abstraction. This is not a paper about how to use a silicon debugger to track down a specific class of bugs or improve the performance of a particular type of system. The strategies employed and lessons learned should be valuable to a wide variety of users and industries.

System-Level Debug Challenges are a Silicon Visibility Challenge

Conceptually, system-level debug challenges are shared whether the hardware implementation is ASIC or FPGA technology. In practice, differences arise in the commercial availability of silicon debug solutions. A handful of commercial options are available to the FPGA design and validation teams whereas ASIC silicon debug solutions remain the province of homegrown solutions. Therefore, we will begin by addressing the limitations of commercial FPGA solutions with respect to the ability to debug system-level issues.

The key challenges in system-level debug and debug productivity fall broadly into three categories:

- Trace breadth
- Trace depth
- Turn-around time to debug the appropriate set of signals relevant to the problem

Trace breadth means being able to trace all the signals relevant to debugging a particular issue. Since system-level issues involve functionality in multiple hardware blocks and, frequently, application and/or OS software functionality and its interaction with those blocks, the ability to trace signals from one or more peripheral or accelerator block plus the set of software memory map registers used by the software to interact with those blocks define a minimal trace set size. However, that set is typically

insufficient. The interaction of the hardware blocks and software occurs across the SoC/chip interconnect resulting in additional signals to trace. Lastly, when software is involved, the need to trace processor interactions with memory may also be relevant. To debug system-level interactions, the need to trace hundreds or thousands of signals and registers across multiple clock domains is typical, especially during the initial phases of debug identification and isolation. The introduction of multiple clock domains derives a further requirement that the trace data must be correlated in time in order to comprehend true cause-and-effect relationships.

Due to instrument design and resource efficiency in implementation, most commercial solutions offer limited trace breadth. Practical trace breadth is governed by three factors: Number of clock domains to be instrumented, the number of signals per clock domain to be traced and the resource efficiency of the embedded logic analyzer. The more clock domains containing signals to be traced, typically the fewer signals that can be instrumented per clock domain. With common commercial solutions, it is difficult to instrument more than a few hundred signals across more than 2-3 clock domains due to the relative resource inefficiency of the embedded logic analyzers.

Trace depth determines the ability to capture sufficient activity to recognize system-level cause-and-effect relationships. The interaction of software and multiple hardware blocks occurs via transactions communicated across the chip interconnect. Since the effect of a bug may surface many transactions later from the cause, the ability to trace seconds of trace data can determine whether the validation engineer can debug an issue from one trace set or if multiple runs are required to capture multiple snapshots of trace data which the engineer must cobble together manually to root cause the issue. Of course, the process of manually cobbling together multiple trace sets opens the door for human error in aligning trace data creating the real possibility that a false cause will be identified due to errors in manual alignment of trace data. Secondary, yet relevant, considerations include the intrusiveness of the instrumentation. If streaming of trace data off-chip is required to capture sufficiently deep traces, then two less attractive side-effects are possible. First, streaming creates a bottleneck impacting the rate of trace sampling and/or the trace breadth. Since trace-sampling rates typically cannot be sacrificed due to the impact to debug accuracy, the compromise to trace breadth may significantly impact debug productivity as covered above. Secondly, streaming can and usually is intrusive to the design functionality. Streaming requires the use of chip I/Os. Most designs and prototypes of the designs make full use of the I/Os available. It may or may not be acceptable to “de-feature” the design to free I/O resources for streaming depending on whether the functionality is related to the issue being debugged. The ability to de-feature a design and re-purpose I/Os requires foresight in design or, potentially, a non-trivial amount of design re-work to utilize.

Most commercial silicon debug solutions do not support off-chip streaming limiting the user to make-do with the trace depth that can be captured in an FPGA’s on-chip memory (e.g., block RAM). One advantage to the simplicity of the solution is the ease in calculating the trace depth available. An embedded logic analyzer tracing signals associated with a 100 MHz clock and a trace buffer 8k trace-width-words in depth can capture 0.00008 seconds of trace. That simply is inadequate to debug system-level interactions. For that reason, commercial solutions offer the ability to conditionally capture trace data allowing the user to define when data is valid and should be recorded to the trace buffer. Conditional capture works well when enough information is known to focus on activity relevant for

debugging. Applying compression technology to the trace can greatly increase effective trace depth and can work with conditional capture to achieve deep traces without the intrusiveness of off-chip streaming.

Turn-around time when debugging. Debugging is an iterative process of tracing back from an observable effect back to the root cause. Trace breadth and depth directly impact the debug turn-around time. Since the root cause is not yet known, all the signals relevant to debug are not known upfront. Yet, all debug instrumentation is finite in the number of signals that can be instrumented and traced. Whenever the design must be re-instrumented to include additional or different signals in the trace set, the turn-around time can be many hours, often overnight if re-synthesis and place and route of the design is required (and it usually is). Thus, productivity in debugging issues found in silicon is typically measured in days or weeks.

Most commercial solutions link the instrumentation of signals to be traced with the act of tracing. This exacerbates the debug productivity challenges. Some commercial solutions attempt to ease the problem by providing 8:1 multiplexors (muxes) on the inputs to the embedded logic analyzer. Muxing the input, in theory, should significantly improve debug productivity. In practice, the benefits have been marginal as muxes define 8 distinct trace sets. Either a significant amount of redundancy is required to include "high value" signals in all trace sets effectively reducing the theoretical 8x breadth improvement by about half or it has no impact on turnaround productivity as signals required to debug an issue are placed in mutually exclusive trace sets. However, there is one solution that separates the instrumentation of signals for potential tracing from configuring the instruments to trace a specific subset of the signals instrumented.

Other considerations for turnaround time and debug productivity include the ease-of-use and power of the software bundled with the instruments that make it easy to instrument the design and configure things such as the trigger and conditional capture conditions at run-time. However, this paper is not a review of the differences between commercial offerings. Therefore, we assume the software, more or less, adequately supports the capabilities of the instrumentation in its application to debugging system-level issues and that debugging is based on waveform views of signals.

In this presentation we will look at three case studies of debugging system-level issues in silicon. Collectively, the examples show silicon debug solutions fill an important gap in validating and supporting complex chips and SoCs. Silicon debug solutions can do what traditional pre-silicon hardware and software debugging tools cannot. A quick overview of the examples:

1. Software and hardware designers on a project labored for weeks attempting to reproduce in simulation system-level bugs exposed in silicon. Applying an FPGA debug solution resolved the issue in a few days.
2. Silicon debug isn't always about fixing binary correct/incorrect functionality. The second example shows how it can help improve the overall quality of a system.
3. The third example illustrates how a commercial silicon debug solution provided a second level of support in identifying and resolving a bug that escaped to silicon that incorporated a proprietary silicon debug capability.

Root Causing System-Level Bugs that Escape Simulation Detection

The first case study involves a team working on an FPGA prototype. This was a complete product prototype using two FPGAs which would later be converted to pin compatible ASICs. This scenario allowed co-design of hardware and software components within the complete system context. As is typical of chip and system prototypes, the early and concurrent development and testing of embedded software was a key objective of the prototyping stage. As hardware designers completed and verified features of the chips, software designers could enable those features for testing.

The fun began when software testing exposed two features behaving incorrectly. Both hardware and software designers searched for the cause of the issues. The hardware designers verified in simulation that properly configured, the hardware blocks work as expected. Meanwhile, the software designers, via register reads in the hardware, were able to confirm that the hardware blocks were configured correctly. This resulted in the all too common situation where software and hardware engineers pointed their fingers at the other side as responsible for the bug. The back-and-forth continued for approximately 2 weeks including extensive code reviews and various attempts to reproduce the situation in simulation.

After tiring of finger pointing without resolving the issue, the project team employed a silicon debugging solution in the hopes of finding new and materially different information by observing the actual silicon behavior as it interacted with the software instead of the simulated behavior. This is when one of the authors was brought into the team to assist with the use of the silicon debug solution. With no specific knowledge of any of the bugs, we identified a set of signals that would provide broad, general purpose debug visibility. Taking advantage of capabilities in this silicon debug solution used, we were able to instrument a large number of signals and then easily configure the instruments at run-time to trace specific subsets that seemed relevant to debugging each issue. This capability provided very quick turnaround as we reconfigured the active trace set to follow the causation as we traced back from first observation to root cause. Having experience using the FPGA vendor's debug solution, this run-time trace configuration and broad instrumentation saved us many days resulting from re-instrumentation and re-build spins that would have limited debug sessions to one or two a day instead of the near simulation-type multiple debug sessions daily.

Our method for selecting a broad, general-purpose debug instrumentation set can be applied to virtually any design. We selected all of the interface signals (signals between major blocks of the design) as well as some other key signals such as the register interface bus (the software register-memory map). Depending on the design and complexity of IP blocks, the addition of key state registers would result in a highly useful, default debug instrumentation for any design. Using the initial general-purpose instrumentation the hardware designer was able to find the source of the bug in one day. The designer started by viewing a few high level status signals. That brought his attention to the right area. He was able to select a few more signals in that area and quickly determined that the software configured the block with the correct values. However, the configuration involved out of order transactions, which caused the hardware to start performing its tasks before being fully configured. The designer selected a few more signals to verify the behavior and was then able to show the software designer the timing and

order of the configuration. This information directed the software designer exactly to how he needed to modify the code to correct the error.

Of course, the hardware engineer could boast that the problem was in software, not his hardware. In reality, this situation is far more common than the industry would like. How often has ambiguity in a system or interface specification resulted in bugs that escape pre-silicon verification? How many escaped silicon and system validation? How many have delayed product shipment due to the time to identify and fix the problem? These are exactly the types of system-level issues that require the clarifying visibility into hardware in order to quickly and accurately resolve the issue and keep projects on schedule. But, it also requires a silicon debug solution that enables a productive debug use model. The ability to dynamically change the trigger condition and signal selection without recompiling accelerated the debug cycle from days to hours even with silicon visibility.

Silicon Debug Improves Chip/System Quality

The success of a chip or system does not depend solely on correct functionality. If the performance or power consumption of the product does not meet market requirements or beat competitive offerings, the chip or system is highly unlikely to be successful. The second "bug" this company uncovered using the silicon debug solution wasn't a functional correctness issue. Systems performance was far lower than expected, so poor that the system would be unviable in the market unless it was fixed. While chasing down the first bug, the hardware designer noticed the hardware blocks were idle for far longer than expected. He noticed that the same software addressable registers were set several times with the same value before the block was finally allowed to run. By showing the software designer which registers were being set and when, the software designer was able to identify the sources of redundant register settings and optimize the code. Although the performance issue may have eventually been found using traditional software profiling tools, the visual waveform provided by the silicon debug solution allowed the team to quickly identify and then address the problem. When it comes to these qualitative issues, it is better to have multiple views offering different perspectives into the behavior of the system to both identify and rectify quality issues.

Silicon debug solutions are complementary to embedded software debug and analysis solutions. The hardware designers and software designers found the waveform viewer an invaluable tool for facilitating communication between the two groups. The software designers would not be able to interpret the waveforms by themselves, but with the hardware designers showing the cause and effect of the software transactions the software designers were able to gain valuable insight into the behavior of the hardware. On the surface this might sound like a very inefficient way for a software designer to do debugging if a hardware designer is required to assist in the debug, but what this company found was that by facilitating communication between the two, the two teams worked more closely as collaboration is facilitated by silicon observability. There were a lot less "I can't find the problem on my side, it must be on your side" conversations and a lot more "let's see what's going on" conversations. Each team felt more invested in the other team's success because they saw how it impacted the product and project's success. This resulted in further, broader use of the silicon debug solution which found more opportunities for enhancing system performance that would have otherwise been missed. Since

its use was in the FPGA prototype, the enhancements impacted both software and hardware optimizations.

As an example of hardware optimizations for system performance, it was possible to recognize that certain register were accessed more frequently than other registers. Since a typical register read operation took about 25 clocks to complete, this represented a ripe area of optimization. The hardware team created a special low latency bus that could access a register in about 2 clock cycles – an order of magnitude improvement for this aspect of the system performance.

On the software side it was recognized that certain hardware accelerators were not running at peak efficiency. The design included two buffers of raw data. As one buffer filled, the data in the other buffer was processed. Software controlled the ping-pong between the two buffers. They noticed that after the "ping" buffer was processed, it took valuable clock cycles to enable the "pong" buffer. By analyzing the control path delays the designers were able to safely send a "start" command to the pong buffer before the processing of the ping buffer data completed. As is often the case, optimizing performance by exploiting certain delays in operations can result in the creation of bugs as timing is pushed to the limit. Such a bug occurred when the ping-pong timing was incorrect resulting in software overwriting a memory location at the same time that hardware was processing the data. From the software perspective, they could see the memory value was corrupted but they could not determine the cause. Combining the silicon debug capabilities with the embedded software debug capabilities allowed both teams to recognize the simultaneous access to the same memory location. This turned out to be another example of the value of a good, broad default debug instrumentation. No change in instrumentation was required to debug this issue.

Due to the unprecedented visibility now available to the project team, optimization became a key metric of product development. The project manager kept a daily white board update of today's system performance encouraging even broader, deeper exploitation of the system visibility now available to the teams. Figure 1 shows the type of insight a silicon debug solution gave the team when they were looking for areas that could be optimized for performance. Long periods of idle time become glaringly obvious with a waveform view of system level interactions.

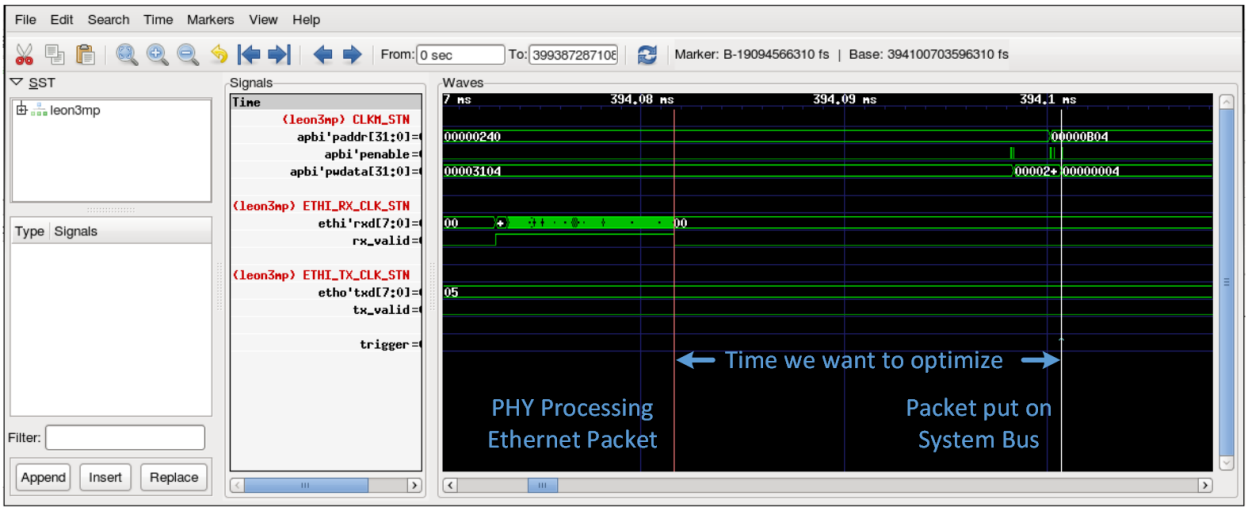

Figure 1: Example of System Optimization

Silicon debug solutions work best when they are non-intrusive to the design functionality as mentioned earlier in this paper. The silicon debug solution used was non-intrusive. However, hardware designers are often ignorant of how intrusive embedded software debuggers are! If you are an experienced software engineer, you've surely experienced at least once in your life the pleasure of observing a bug in the operation of your software that magically disappears as soon as debug visibility or a printf statement is added. The fact is, a software breakpoint results in the execution of hundreds of instructions to implement. A lot can happen and data can dissipate during that time. The software team recognized and appreciated the non-intrusive nature of the silicon debug solution. In addition to the latency and intrusive nature of software breakpoints, they are ineffective at halting hardware. The interaction between hardware and software are where many system bugs lurk and a software breakpoint changes the timing and interaction between the two. The team was able to observe things such as cache misses in a non-intrusive way which resulted in resizing the cache and rearranging code so that it would fit in the instruction cache with less calls to main memory.

Supplementing Proprietary ASIC Debug Capabilities

The final case study comes from a company which prototyped their ASICs before committing to silicon. Although this company has a homegrown ASIC design for debug (DFD) architecture, they found the use of a commercial silicon debug solution worked better in their prototypes. For prototype debug, they employed a very systematic method of debugging. Like the company of the first two case studies, they instrumented all of their interface signals. When they first observed a bug this default instrumentation was sufficient to isolate the functional block likely containing the root cause. If the initial instrumentation was insufficient to root cause the bug, it typically required just one re-instrumentation to root cause and fix the bug. They found the ability to instrument large structures, deep traces, the ability to dynamically select different signals and triggers at run time, and the ability to debug across multiple clock domains to be among the most valuable turnaround time, debug productivity capabilities. Despite their success using the commercial silicon debug solution in prototyping, they made the conservative decision to continue using their proprietary debug solution in ASICs due to its past success in that context.

Based on our discussions with several ASIC design organizations, it is common that proprietary DFD solutions utilize a tree of muxes that route key probe points to dedicated debug pins which can then be connected to a traditional logic analyzer. Probe points are carefully chosen and mux pin position carefully chosen. If every probe point is available in combination of every other probe point using a standard muxing scheme the result is a complete cross bar switch, which consumes tremendous resources. If you minimize your mux network then you end up with blocking scenarios as discussed at the beginning of the paper in reference to commercial solutions that place 8:1 muxes on the inputs to the embedded logic analyzer. You might be able to view signals A and B, or signals A and C together, but you cannot view signals B and C together.

This company experienced a bug that escaped to ASIC silicon. It is never good for customers to find bugs in your product. Fortunately, they found the bug before the customer, but they were still under tremendous pressure to issue a software fix before the customer encountered it. The ASIC had insufficient probe points and the probe points that existed could not be viewed in the combination required to debug the problem. The scenario which triggered the bug was also known and repeatable. The prototype team was called upon to recreate the bug in the prototype. Once the bug was recreated, two re-instrumentations and 2 days were all that was required to find the root cause of the bug. On the third day, a software fix was delivered and validated. The software fix was in the customer's hands on the fourth day. The commercial silicon debug solution saved the company from a potentially bad situation and ensured their customer was able to stay on their system development and integration schedule. The value of having a more powerful fallback debug capability that could be used effectively in the prototype also reduced the risks inherent in the homegrown ASIC DFD capabilities. Regardless of relative strengths and weaknesses of the commercial solution versus the homegrown solution, resource constraints in final ASIC silicon and the inability to re-instrument ASICs (at least not in a reasonable amount of time and cost) will create challenges for any solution. Thus, the value of prototyping ASICs extends beyond receipt of first silicon.

All of these case studies illustrate the advantages of choosing an advanced silicon debug solution. To be efficient for system level debug all of the challenges presented at the beginning of this paper must be addressed by your debug solution: trace breadth, trace depth and debug turnaround time. Additionally the solution must be resource efficient and easy to use. If any of these elements are missing, silicon debug is often viewed as a tool of last resort. But if all of these challenges are met, then the user is empowered to track down the root cause of the bug where the bug was discovered, in hardware. For the case studies presented, the debug solution used was Mentor Graphics' Certus.

Regardless of which silicon debug solution chosen (commercial solution or internally developed), the authors hope these case studies provide the rational to encourage everyone to adopt one for their ASIC prototypes, FPGA designs and ASIC silicon requirements. The benefits in improving the quality of your future chips and improving your silicon and system-level debug productivity are well worth the investment. In addition to the benefits prior to shipping product to your customers, silicon debug solutions, if integrated into the full chip and system functionality, can help provide better support of your products once in the field.