UBM TechWeb

EE Times

ELECTRONICALLY REPRINTED FROM FEBRUARY 15, 2012

Bridging software and hardware to accelerate SoC validation

Brad Quinton, Tektronix

As the complexity of systems-on-chip (SoCs) continues to increase, it is no longer possible to ignore the challenges caused by the convergence of software and hardware development. These highly functional systems now include a complex mix of software, firmware, embedded processors, GPUs, memory controllers, and other high-speed peripherals. This increased functional integration, combined with faster internal clock speeds and complex, high-speed I/O, means that delivering a functional and fully validated system is harder than ever.

Traditionally, software validation and debug and hardware validation and debug have been separate worlds. Often, software and hardware teams work in isolation, with the former concentrating on software execution within the context of the programming model, and the latter debugging within the hardware development framework, where clock-cycle accuracy, parallel operation and the relationship of debug data back to the original design is key. In theory, fully debugged software and hardware should work flawlessly together. But in the real world that is rarely the case, a fact that often leads to critical cost increases and time-to-market delays.

To deliver increased integration within a reasonable cost and time, the industry must transition to a new approach – design for visibility. Said another way, engineers must design, upfront, the ability to deliver a full system view if we are going to be able to continue to validate and debug these systems effectively. The key is to be able to understand causal relationships between behaviors that span hardware and software domains. This article describes an approach to debugging an SoC using embedded instruments, and shows how the integration of hardware and software debug views can lead to faster and more efficient debug of the entire system.

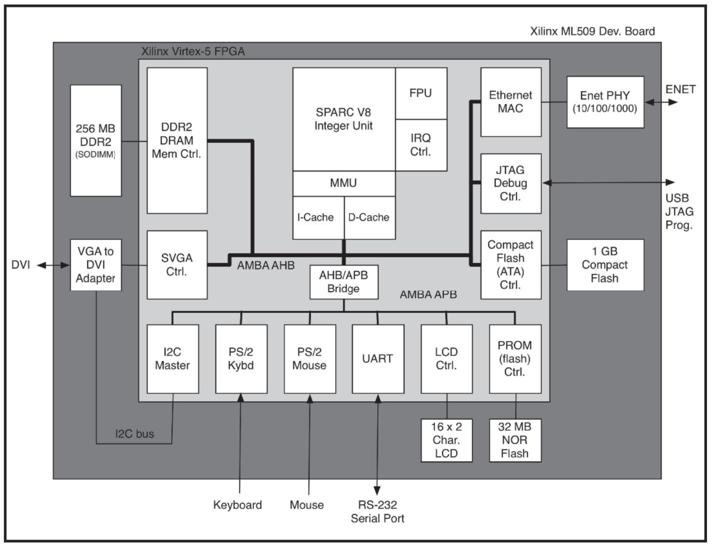


Figure 1: Baseline test bed SoC

Building the Test Bed

The test bed SoC shown in Figure 1 is composed of a 32-bit RISC instruction set processor, connected to an AMBA AHB system bus, and AMBA APB peripheral bus. The SoC also contains a DDR2 memory controller, a gigabit Ethernet network adapter, a Compact Flash controller, VGA controller and number of low-speed peripheral interfaces. The SoC runs Debian GNU Linux operating system version 4 running kernel v2.6.21. The processor core operates at 60MHz, the DDR memory controller at 100MHz, and the other I/O peripherals operate at their native frequencies between 33MHz and 12MHz. The entire SoC is implemented on a Virtex-5 development board.

Together, this system is a fully-functional computer able to provide terminal-based user access, connect to the Internet, run applications, mount file systems, etc. The SoC is characteristic of those that create complex debug scenarios, and stress the capabilities of both hardware and software debug infrastructures. In most cases, key operations span hardware and software.

Debug Infrastructures

Processor core developers generally provide debug infrastructures, either as a fixed set of features for a given core or as a configurable add-on to a family of cores. In either case, the debug infrastructure becomes a part of the manufactured core. Debug software then uses this infrastructure to provide debug features to software developers.

The processor core highlighted here supports a basic set of debug capabilities similar to those available on most modern processors, including those from Intel, AMD, IBM, Oracle and ARM. In this case, a “back-door” accessible via JTAG allows a software debugger, for example GDB, to read and write memory in the system and detect the operational state of the processor. Through these mechanisms, along with access to the original software source code, GDB and other software debuggers can provide software breakpoints, single-step operation, examination of variable values, stack tracing, configuration of initial conditions, alternation of memory values and resume functionality.

In most cases, hardware debug infrastructures are not delivered with the hardware IP cores that make-up a SoC. Instead the hardware debug infrastructure is often overlaid onto an existing SoC design. There are a number of reasons for this difference. First, unlike software debug, the underlying functionality required of hardware debug is diverse and often not completely understood until the SoC is assembled. In addition, each new SoC often requires a different debug infrastructure. Finally, as an emerging area, there is less standardization and less of an ecosystem around hardware debug. The development of a hardware debug infrastructure thus is often left to individual designers who create ad-hoc debug features targeting different functional areas. In larger organizations, internally supported tools and architectures are often developed. However, as the complexity of SoCs continues to rise, so does the complexity of creating an efficient hardware debug infrastructure, and internal development efforts are difficult to sustain.

As an alternative, test and measurement vendors can provide complete design tools, IP library, and work flow to create a hardware debug infrastructure. The set up shown in Figure 2, called the Tektronix Clarus Post-Silicon Validation Suite, is composed of re-configurable embedded instruments that can be connected together and distributed throughout the SoC to create a debug infrastructure specific to the functional requirements. The Implementer tool allows the instrumentation of any signal, at any level of hierarchy, at the RTL-level (Verilog, System Verilog, and VHDL) in the hardware design. The Analyzer configures and controls the embedded instruments via JTAG or an Ethernet connection. Lastly, the Investigator maps the data collected by the embedded instruments back to the original RTL (in a simulation environment) to enable more complex debug.

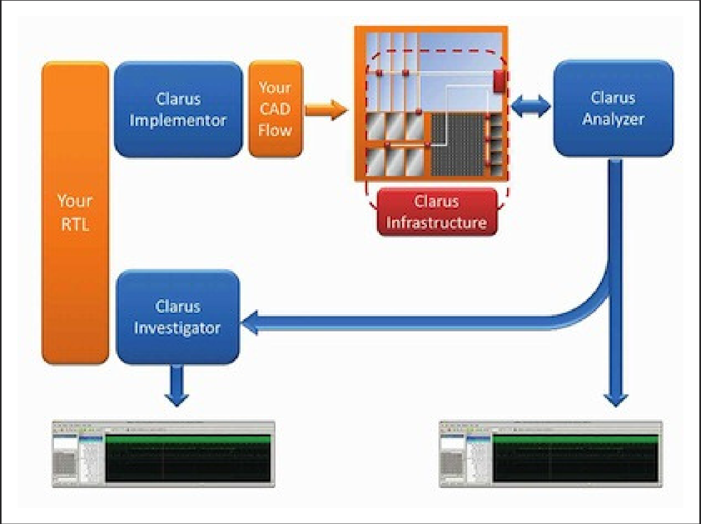


Figure 2: Architecture of the Clarus Post-Silicon Validation Suite

The embedded instruments are applied to an SOC to provide a debug infrastructure as shown in Figure 3. An important aspect is the ability to reconfigure the instrument to target various signals and scenarios in different areas of the SoC while debug is in progress. The base instruments are called capture stations, which independently manage the selection,

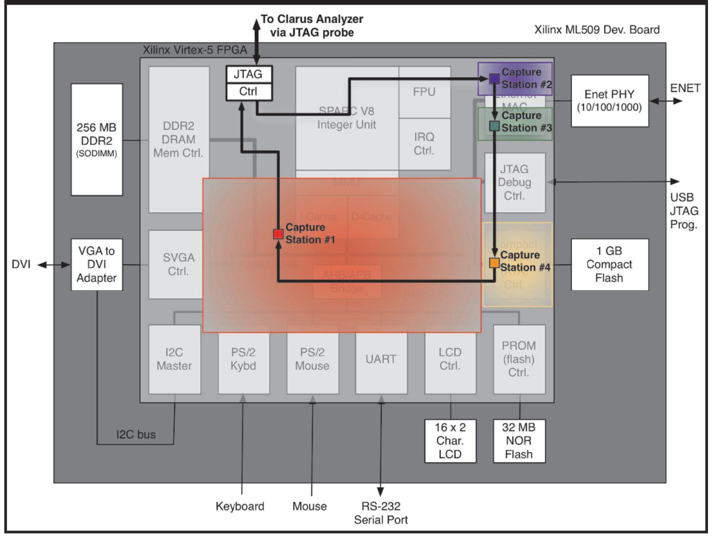


Figure 3: Hardware debug infrastructure

compression, processing and storage of observed data. Multiple stations are often used together to create a design-specific infrastructure for a given SoC. During insertion, the capture stations are configured with a list of potential signals of interest, a maximum number of simultaneous observations, and maximum RAM size. Capture stations are generally assigned to specific clock domains, and capture synchronously to observed data. The analyzer collects the data from each station, reverses the compression algorithms, and aligns the data captured in each station to produce a time-correlated view across all capture stations.

The SoC used in this example has four capture stations: one in the processor clock domain, labeled Capture Station #1 (60MHz) targeting 362 signals; one in the RX Ethernet domain, labeled Capture Station #2 (25 MHz) targeting 17 signals; one in the TX Ethernet domain, labeled Capture Station #3 (25 MHz) targeting 17 signals; and finally one in the compact flash clock domain, labeled Capture Station #4 (33 MHz) targeting 178 signals. Each of these stations operates in parallel, and is able to make selective observations of any combination of signals. The final output of the analyzer tool is a waveform representing the clock-cycle accurate signal transactions in the actual silicon device as shown in Figure 4.

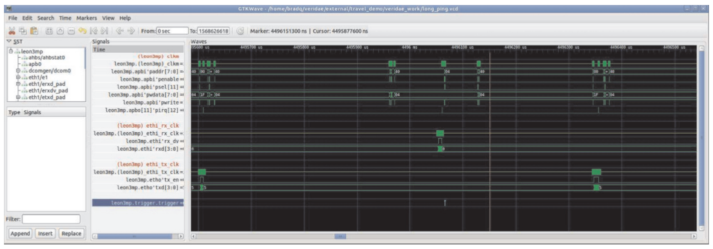
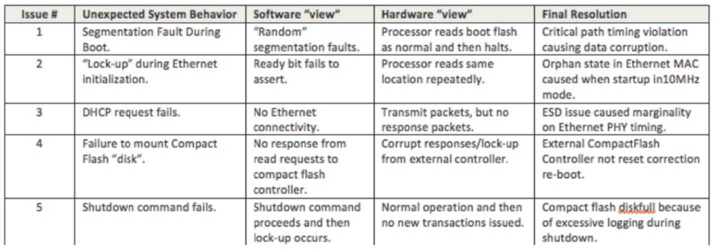


Figure 4: Example SoC waveform

While both the software and hardware debug infrastructures perform well on the target platform for issues that are confined to either software or hardware, it is a significant challenge to understand behavior that involves the interaction of software and hardware. Table 1 outlines a short list of some of the issues encountered during the development of our test bed, and which are representative of the issues we see across the industry.



Issue #	Unexpected System Behavior	Software "view"	Hardware "view"	Final Resolution
1	Segmentation Fault During Boot.	"Random" segmentation faults.	Processor reads boot flash as normal and then halts.	Critical path timing violation causing data corruption.
2	"Lock-up" during Ethernet initialization.	Ready bit fails to assert.	Processor reads same location repeatedly.	Orphan state in Ethernet MAC caused when startup in10MHz mode.
3	DHCP request fails.	No Ethernet connectivity.	Transmit packets, but no response packets.	ESD issue caused marginality on Ethernet PHY timing.
4	Failure to mount Compact Flash "disk".	No response from read requests to compact flash controller.	Corrupt responses/lock-up from external controller.	External CompactFlash Controller not reset correction re-boot.
5	Shutdown command fails.	Shutdown command proceeds and then lock-up occurs.	Normal operation and then no new transactions issued.	Compact flash diskfull because of excessive logging during shutdown.

Table 1: Example SoC debug issues

A primary challenge is that while the effects of the unexpected behaviors are "visible" using either the software or hardware debug infrastructures, it is often very difficult to determine whether the observed incorrect behavior is the cause or the symptom. The question often becomes whether the unexpected behavior in the software is a reaction to incorrect hardware behavior, or the other way around. The key is to determine the causal relationship between events, which requires a common reference between the software and hardware debug views.

Event Management

The ability to re-construct a causal relationship between software and hardware debug views involves integration across the debug state and event processing from the two debug infrastructures, or integrated event management as shown in Figure 5.

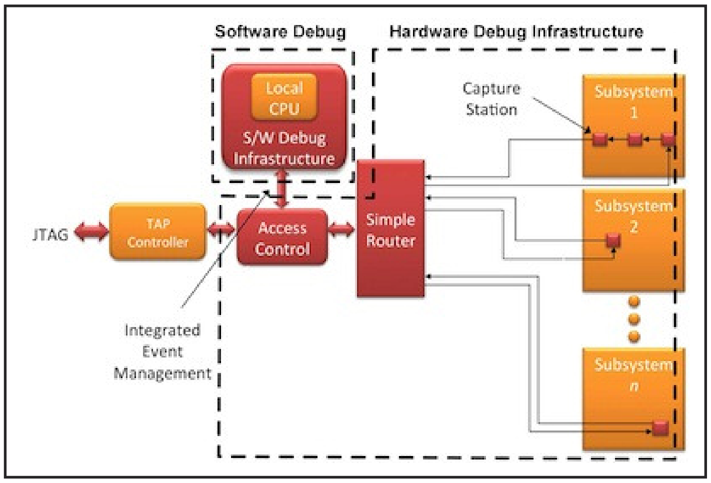


Figure 5: Integrated event management

In this example, distributed, asynchronous instruments provided by the Clarus Suite, make it possible for each capture station to be viewed as autonomous. To support "cross-triggering" between instruments there is a shared event bus and a centralized event processor. The centralized event processor, labeled Access Control in Figure 5, communicates the debug events and state to the Analyzer software that manages the overall debug infrastructure. This enables the effective hardware debug of many functional units and clock domains simultaneously. To create the integrated event management this information propagates into and collects data from the software debug infrastructure. With integrated event management in place, the infrastructure can detect software breakpoint events and the debug state of the processor. Likewise, the software debug infrastructure is able to detect hardware triggers and the debug state of the hardware debug infrastructure.

The two key benefits of integrated event management are the ability of software debug initiated events to control hardware triggers, and the ability of hardware debug initiated events to control software debug. More specifically, software breakpoints can be mapped to specific hardware behavior and hardware triggers can break software at a specific point. Ex-

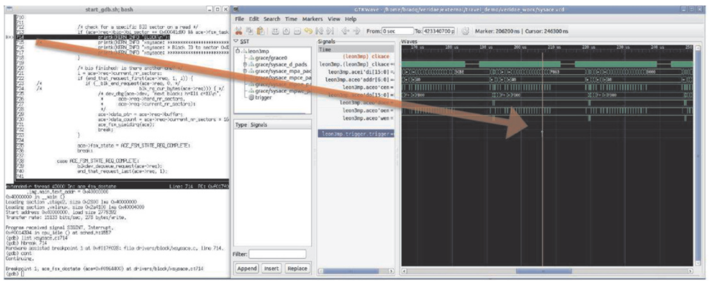


Figure 6: Example of a software-initiated event

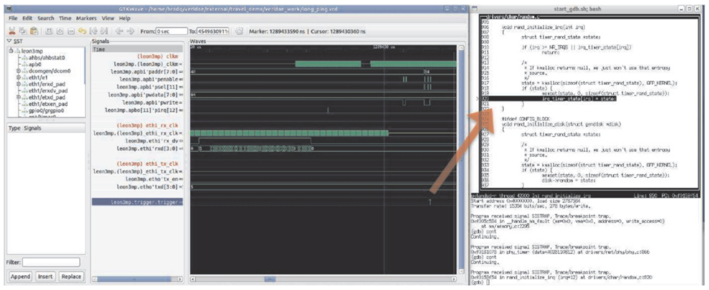


Figure 7: Example of a hardware-initiated event

amples of these two scenarios are shown in Figures 6 and 7.

To demonstrate the capabilities of the software-initiated breakpoints in an integrated debug system, the Linux kernel was modified to print the message "BLOCK" when a read occurs on disk sector 0x00041d90. Then, traces from the "sysace" Compact Flash controller with the debug infrastructure were targeted. Using GDB, a hardware breakpoint on line 714 of xsysace.c file (the line where the printk occurs) was set. Then the test infrastructure was configured to monitor the software debug infrastructure using integrated event management. Finally, the "find /" command forced the kernel to read the entire disk. As shown in Figure 6, the software break point halted kernel execution on the desired line and also triggered the hardware debug infrastructure. As a result the detailed behavior of the hardware is visible at the time of the software breakpoint.

The Ethernet adapter was used to demonstrate the capabilities of the hardware-initiated triggers in an integrated debug system. A hardware trigger was set to occur when the "RX Packet Ready Interrupt Bit" in the Ethernet adapter was cleared by the Linux Kernel. The integrated event management interface was configured to map hardware events to the software debug infrastructure. A ping to the IP address of the router in the system initiated a transmit packet from the SoC to which the router responded. When that response occurred, the packet arrived on the Ethernet PHY, and was processed by the Ethernet adapter. The processor was then interrupted and the Linux Kernel serviced the interrupt. When the interrupt servicing was complete, the interrupt was cleared. This caused a hardware trigger and the software halted, as shown in Figure 7. The resulting view shows the simultaneous or time-correlated behavior of hardware and software in a complex system from the PHY level all the way to the operating system.

Summary

By creating an integrated event management interface between software and hardware debug infrastructures, it is possible to achieve single event synchronization around both software and hardware debug events. This synchronization enables the meaningful presentation of simultaneous debug data from both infrastructures. Such a full system view opens a window into the causal relationship between SoC functionality that spans software and hardware, leading to faster and more efficient debug of increasingly complex SoC designs.

About the Author

Dr. Brad Quinton is the Chief Architect for the Embedded Instrumentation Group at Tektronix. He has over 15 years of engineering and research experience in the semiconductor industry. Quinton's doctoral research at the University of British Columbia exploring on-chip debug architectures, was the inspiration behind the technology being developed by Veridae Systems Inc. which is now part of Tektronix Embedded Instrumentation Group. Previously, Brad served as a consultant and senior design engineer at Teradici Corporation, and as a consultant at Altera, where he designed and debugged new devices. Prior to this he spent many years at PMC-Sierra, where, in his last role, he managed a multi-million dollar IC development from concept to market release. He can be reached a brad.quinton@tektronix.com.

Tektronix®