Functional Debug Techniques for Embedded Systems

Bart Vermeulen
NXP Semiconductors

Editor's note:
In the past few years, functional debug has made significant progress. This article describes the most common structured approaches available for silicon debug of embedded systems.

—Rob Aitken, ARM

Some problems in a new chip design or its embedded software show up only when a silicon prototype of the chip is placed in its intended target environment and the embedded software is executed. Traditionally, embedded-system debug is very difficult and time-consuming because of the intrinsic lack of internal system observability in the target environment.

Design for debug (DFD) is the act of adding debug support to a chip's design in the realization that not every silicon chip or embedded-software application is right the first time. For more than a decade, DFD has proven to be essential in the fast bring-up of prototype silicon and embedded software and in the analysis of field returns. In several examples throughout the industry, the inclusion and subsequent use of DFD features have contributed significantly to a reduction in both time to market and development cost.

DFD gives debug engineers increased observability of an embedded system's internal operation. There are severe constraints, however, on the amount of debug observability that DFD can provide for error localization. An embedded SoC contains tens of millions of gates. The outputs of these gates can switch with frequencies greater than 1 GHz, resulting in debug data volumes in the order of terabytes per second. This data volume cannot be output in its entirety in real time. There are not enough device pins available, and the pins that are available are limited in speed. Also, to ensure that debug data transport introduces no additional problems, it must intrude as little as possible on the original application's execution. Finally on-chip debug support must be implemented with minimal silicon area cost, so as not to overburden high-volume products. Therefore, chip manufacturers must always weigh the DFD implementation's cost against its potential to reduce time to market and development cost.

In practice, to bridge the gap between the amount of on-chip data and the limited off-chip bandwidth, SoC designers use two rather pragmatic and complementary approaches to transporting valuable internal debug information: run-stop debug and real-time trace debug.[1]

Run-stop debug

Run-stop debug uses execution control to start a system and then stop it on a breakpoint at a point of interest to allow inspection of the system's state. Figure 1 shows a software-controlled run-stop debugger. After inspection, the debugger can single-step the system's execution at some particular granularity, restart it from a reset, or resume it. This debug functionality is controlled via a low-speed serial debug interface, such as an IEEE 1149.1 test access port (TAP).

To support run-stop debug, designers add three functions to an embedded SoC's design: breakpoints, execution control, and internal-state access.

Breakpoints. Breakpoints determine the points in time at which system execution is interrupted. On-chip breakpoints enable a fast response to the occurrence of an internal event. Breakpoints are program-

0740-7475/08/$25.00 © 2008 IEEE Copublished by the IEEE CS and the IEEE CASS **IEEE Design & Test of Computers**

mable because designers do not know at design time when to stop execution to successfully expose a bug. For example, breakpoints can be based on the program counter in a processor core.

Execution control. Debug engineers use on-chip execution control to react to a breakpoint and interrupt system execution. There are two existing control approaches: stopping and halting.

Stopping involves disabling the functional clocks, thereby effectively freezing the content of functional flip-flops and embedded memories. Stopping execution allows the debug engineers to observe the system state very close to the breakpoint and in great detail because the content of all pipeline stages of internal processes are also frozen. Sometimes this full state observability is the only way to locate the root cause of a problem.

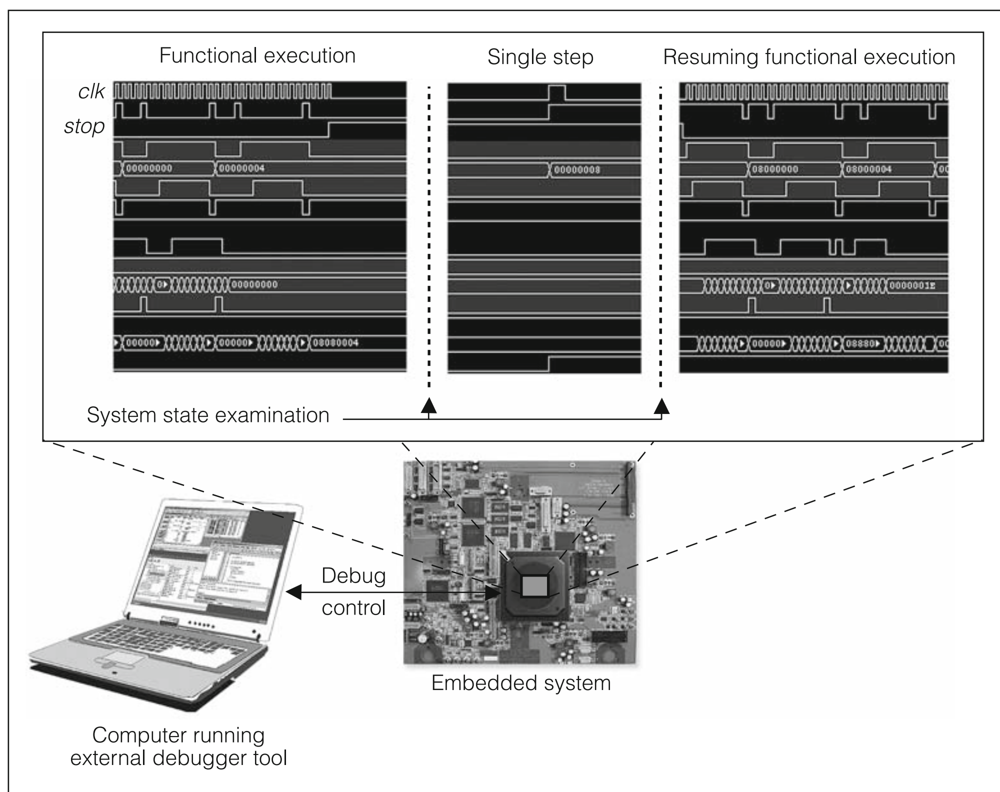

Figure 1. External debugger software controlling and inspecting a system's execution during run-stop debug.

A disadvantage of stopping a system, especially one with multiple, uncorrelated clocks, is that the system state after stopping might not be 100% identical in repeated runs with the same breakpoint. This state noise is a known inconvenience for which no solution exists, but it can be handled to some degree by design analysis or statistical postprocessing techniques.[2,3] A second disadvantage of stopping the clocks is that the exact phase relationship between functional clocks is lost, complicating a resume operation.

Halting a system's execution involves putting a subset of system components into a functionally idle mode in which no functional control or data processing takes place. The functional clocks continue to run in this approach.

Halting is categorized according to which system components are halted when a breakpoint occurs: computational components (such as processors and peripherals), communication components (such as system buses and on-chip networks), or both.

Computation-centered debug has the advantage of being closely related to the programmer's view of the system, as coded in the embedded software. At a breakpoint in one of the processors, dedicated on-chip hardware sends an interrupt to all the other processors, which then enter their own debug interrupt handler routines. The processors can then receive and execute instructions from external debug software. In this mode, the debug software has unrestricted access to the on-chip communication architecture for state access in the peripheral cores. However, because the interrupt starts in one processor core and is then distributed to the other parts of the chip, it is difficult to accurately correlate the operations in one (processor) core with those in other (processor or peripheral) cores.

Communication-centered debug is complementary to the computation-centered approach.[4] It relies on the communication infrastructure's central place in the system. At a breakpoint in the on-chip communication infrastructure, the infrastructure finishes the current transport of commands and data (at a granularity of choice) and afterward discontinues the handshakes involved in the communication protocols. On the

input side, the communication infrastructure no longer accepts commands and data from the computation cores. On the output side, it no longer delivers any commands or data. The net result is a functionally idle system that allows external debugger software to take control and inspect the system's state.[4] This approach offers better support than the computation-centered approach for synchronized debug of operations between cores.

An advantage of halting over stopping is that a subsequent resume operation is far easier. When a system halts, all clocks remain running, eliminating any clock phase misalignment when execution resumes. In contrast, such misalignment is very likely when a stopped multiple-clock system resumes. A halted system can usually resume operation when the external debugger software instructs the halted processors to exit their interrupt handlers and return to execution of the embedded software. Similarly, the debugger software instructs the communication infrastructure to resume acceptance and delivery of commands and data.

Functional execution can also continue in a single-step fashion, taking small steps through system computation or communication. The granularity of these steps is programmable so that the debug engineer can use the right granularity to debug hard-to-find system problems. For example, the steps can be taken at function calls, source code lines, assembly instructions, transactions, data words, or clock cycles.

Internal-state access. Once the system's functional execution has been interrupted, the system can provide access to the internal state through either functional or structural means. A functional method is to allow the execution of read and write operations to the processors and peripherals via a functional or test device interface under debugger software control, which works only for a halted system.[5,6] A structural method is to access the manufacturing-test scan chains from a TAP, which works only for a stopped system.[7]

Real-time trace debug

Real-time trace debug involves bringing internal signals out of the chip on a set of dedicated or shared device pins to increase a system's internal observability. This method complements run-stop debugging by providing additional information about the timing behavior of internal signals. An advantage of this method is that it makes the state of internal signals observable for a long time, helping the debug engineer detect timing-related problems between signals. Typical hardware signals selected are internally generated clocks, resets, handshake signals (such as valid and accept), and state registers of deeply embedded finite-state machines.[8]

A disadvantage is that this method restricts the number of internal signals observable in real time. Only a few chip pins can be reused and dedicated to debug while the application is running. There are far more internal signals to observe than debug device pins available, so these signals must be multiplexed. The outputs of the multiplexers are observable off chip. Multiplexing can occur at different hierarchical levels in the design to reduce the amount of trace wiring and avoid routing congestion.

The chip can output trace data directly on device pins, using either a parallel or a high-speed serial interface. Alternatively, the chip can capture the data in an internal trace buffer, which can later be read out through a low-speed debug interface, such as a TAP. The buffer's size limits the time during which debug data can be captured.

The selection of signals to capture and output can be hardwired in the design, weakly programmable, or fully programmable, depending on the trade-off between the amount of internal observability required and the associated hardware cost and software execution overhead. Fixing the selection in hardware severely restricts internal observability but imposes little hardware cost and no software execution penalty. Making the selection weakly programmable—for example, by using reconfigurable multiplexers in the trace path—increases internal observability as well as hardware cost. Adding dedicated instructions to the embedded software to explicitly write debug data to the trace architecture provides the most flexibility, although the additional load on the processor can become too intrusive for the original application. Also, because the processor doesn't have access to every internal signal, some hardware trace observability is unavoidable.

Figure 2 shows an example of a real-time trace architecture with three debug trace sources.

There are two general types of real-time trace architectures: nonelastic and elastic. Nonelastic architectures are characterized by having only a constant clock cycle delay between the trace source and the trace destination.[9] These architectures can be unclocked—that is, having a pure combinatorial delay—

or clocked by a single clock source. Pipeline stages can bridge long distances on the chip layout and align signals from multicycle paths.

Elastic trace architectures do not have a fixed clock cycle delay between trace source and destination, due to the introduction of varying amounts of buffering in the trace architecture.[6,8] This might be required when trace signals must cross multiple clock or power domains, or when the speeds on the trace sender and trace receiver sides are not the same or change dynamically over time. Elastic architectures use FIFO-like bridges to safely cross these domains, at the cost of introducing a variable delay and some hardware cost. For this type of trace architecture, it is often necessary to include time stamp information along with the debug data, so that debugger tools can correlate trace events from different trace sources.

The trace architecture in Figure 2 is elastic, due to the FIFO-like bridges it uses to transfer trace signals across clock and power domain boundaries.

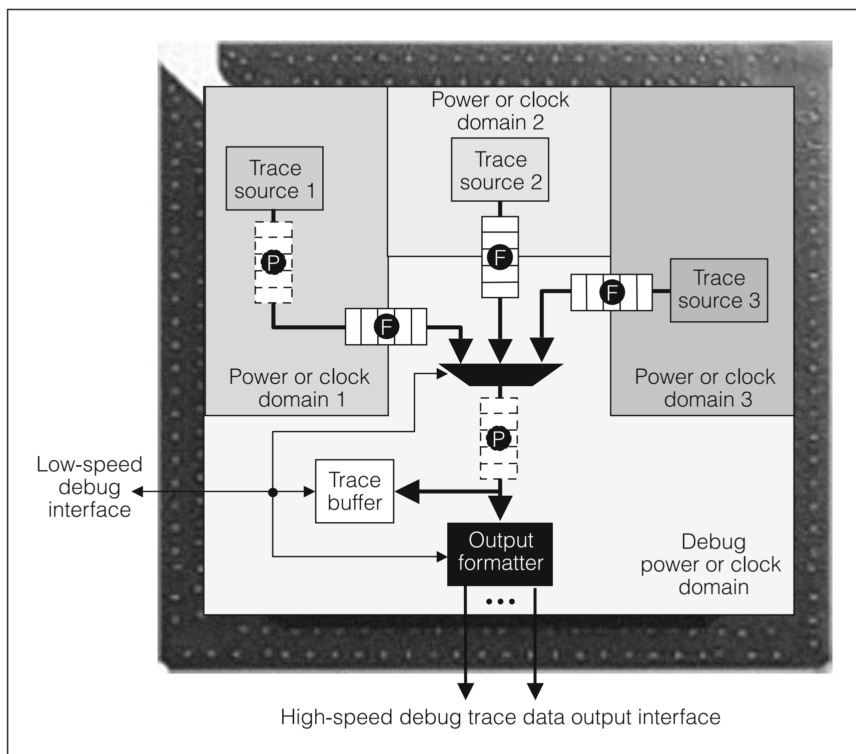

Figure 2. Example of a real-time trace architecture with three debug trace sources. (P: pipeline stage; F: FIFO-like bridge.)

Debug setup

Figure 3 shows an example of a typical functional debug setup. It includes a combination of debugger tools to control and analyze the behavior of on-chip CPUs, DSPs, and hardwired peripherals. These tools provide the following debug functions:

- *Embedded-software upload*. The debugger tools of the corresponding processors upload software programs to main memory or each processor's dedicated memory.
- *DFD-hardware programming*. Breakpoints are programmed to stop the chip at certain points in time. These points can be defined in terms of a source code line in the application code, a specific transaction in the communication infrastructure, or a programmable number of clock cycles in a peripheral core.
- *Board reset*. The application board is functionally reset, leaving the debug programming intact. This function provides that only a reset of, for example, the IEEE 1149.1 TAP resets all debug hardware.
- *Breakpoint hit*. The debugger tools continuously check the breakpoint mechanism's state to determine whether chip execution has been interrupted.
- *System state access*. Once execution is interrupted, the debugger tools use either functional means such as a bus access or structural means such as scan chains to access the system state for inspection and modification by the debug engineer.
- *Continue or stop*. After inspecting and possibly modifying the content of the relevant registers and memories, the debug engineer can decide either to reprogram the breakpoint and resume functional execution or stop the debug session. By repeatedly programming a breakpoint and accessing registers of interest, the engineer can determine the time at which the SoC's state starts to differ significantly from that of a reference chip, emulation, or simulation. This provides the first insight into a problem's root cause.
- *Real-time trace capture and analysis*. Whenever the system is functionally running, the debug tools can capture and analyze data on the real-time trace data output or in the internal trace

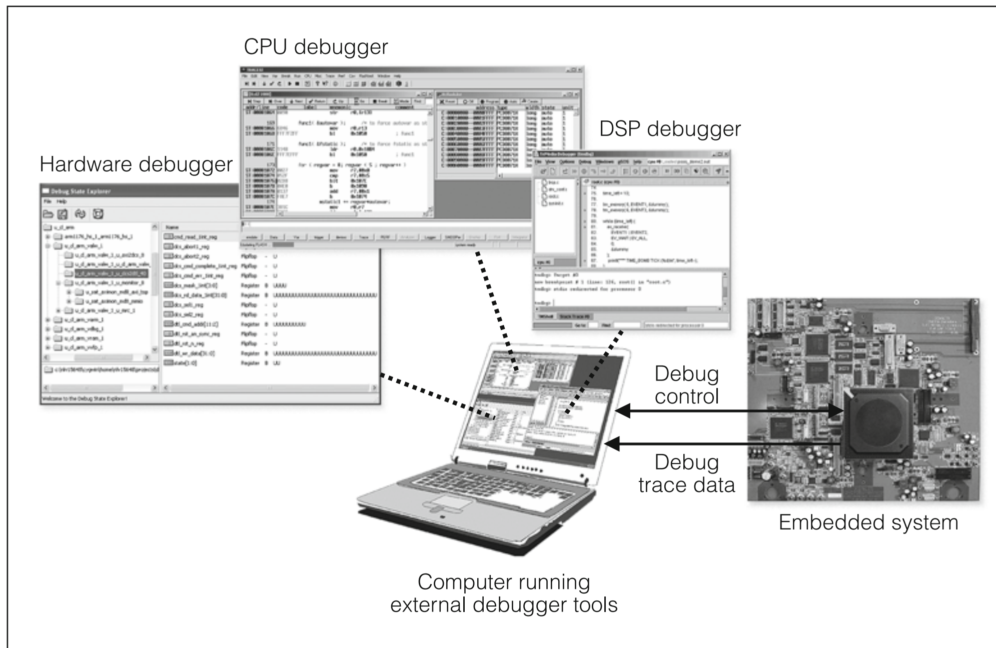

Figure 3. A typical in-situ debug setup.

buffer and present the result of this analysis to the debug engineer for interpretation.

A server application synchronizes the operations of the various debugger tools, using an interface open to all the tools for accessing the IEEE 1149.1 TAP.

Using DFD in industrial SoCs

Run-stop and real-time trace debug have been successful in industrial SoCs. Table 1 lists the characteristics and debug functionality of several examples. Debug engineers used the debug support in three of these chips to help locate the root cause of erroneous behavior in early prototype silicon and embedded-software applications.

Co-Processor Array

Designers used the functional-access path to the external SDRAM extensively during silicon bring-up of the Co-Processor Array SoC. The video processors in this SoC normally communicated data directly to one another but could be programmed to store intermediate video frames in the external SDRAM. By extracting these video frames using the functional-access path and visualizing them on screen, designers could determine more quickly whether a video processor was functioning correctly.

The scan-based silicon debug feature helped the designers diagnose a video synchronization problem. This problem occurred only after approximately 50 to 100 input video frames—1 to 2 seconds of real-time

Table 1. Characteristics and debug functionality of several industrial SoCs.

	SoC				
Characteristic	**Co-Processor Array**	**PNX8525**	**Codec**	**Xetal-II**	**En-II**
Introduction year	1999	2000	2002	2005	2007
Application domain	Audio and video	Audio and video	Audio and video	Video	General-purpose
Process technology (nm)	350	180	180	90	65
No. of clock domains	30	92	32	4	32
Flip-flop count	100,000	193,000	163,000	50,000	245,000
Die size (mm^2)	169	108	102	74	45
Debug functionality	Scan, hardware breakpoints, background SDRAM access	Scan, software and hardware breakpoints, nonelastic trace	Scan, software and hardware breakpoints, nonelastic trace	Scan, hardware breakpoints	Scan, software and hardware cross-breakpoints, elastic trace, environment monitors
Debug area (%)	4	0.3	0.3	0.1	2

video processing. For no obvious reason, the image would disappear from a TV monitor connected to the chip's video output. Before tape-out, the SoC's entire timing-back-annotated netlist had been simulated, but only for the first five video frames, because of the very long simulation runtimes. None of these video frames showed any error, leading to the conclusion that the design was error free.

Nevertheless, the silicon failed at bring-up. After repeated state dumps during the SoC's initialization sequence, and after examining the on-chip bus transactions in detail, the designers discovered that the internal ROM from which the SoC was booting contained a programming error. An incorrect value in one of the video output processor's configuration registers caused the output processor to eventually lose synchronization with the input stream.

Once they found the cause of this problem, the designers could use the debug functionality to stop the chip during its boot sequence and upload a corrected boot program in the external SDRAM. When the SoC was subsequently booted from the external SDRAM, the output image remained stable and correct. With the capability of circumventing this initial boot problem, the designers could verify and demonstrate all of the chip's major video-processing capabilities and show that the remainder of the chip was error free.

PNX8525 and Codec

With enhanced hardware and software breakpoint capabilities and a nonelastic, real-time trace architecture, the PNX8525 Nexperia and Codec SoCs contain more debug functionality than the Co-Processor Array.[9] Designers put the debug features to good use, particularly during first silicon bring-up and early development of several embedded-software applications.

Analyzing power-down modes. With 92 internally generated clocks in the PNX8525, real-time trace functionality was crucial for validating their frequencies and phases. In addition, because each clock domain could be turned off to save power, debug engineers could easily validate correct operation of power management functions by observing clock signals on an external logic analyzer.

Analyzing the GPIO state machine. Engineers also used the real-time trace infrastructure to monitor the state machine in the general-purpose I/O module, to learn why it was not correctly responding to external triggers. Observing that the state machine's state did not change even in the clear presence of external triggers, engineers looked elsewhere for the problem. They more closely examined the external-trigger source selection, leading to the discovery of a programming error. By fixing this error, they showed a correctly functioning state machine, which transitioned correctly when external triggers were applied.

Analyzing failing memories. At design time, designers made the output of the data comparators in the BIST hardware observable via the real-time trace architecture. This allowed engineers to thoroughly analyze the memories in several samples that had failed on the ATE. Whenever an incorrect read operation occurred during a BIST run, one of the trace output pins would be asserted. This signal subsequently triggered a breakpoint and stopped the on-chip clocks. The engineers then scanned the BIST engine to determine the address from which the faulty data was read. Repeatedly executing this procedure, each time increasing the counter value in the breakpoint, stopped the chip on the next failing address and let engineers create a complete memory bitmap through the TAP.

Analyzing an audio problem. During a functional system test of the Codec SoC, several device samples had problems playing correct audio at their nominal supply voltage. However, no problems occurred when the samples played at a higher voltage. These samples passed the manufacturing test on ATE and did not show this voltage-dependent behavior.

The debug engineers started the analysis on the system test board by first simplifying the audio test bench as much as possible. They defined a breakpoint sequence that allowed comparison of a good chip's state with that of a failing one. After dumping the content of several embedded memories over several runs on the audio test bench, each time refining the breakpoint sequence, the engineers found significant differences between the passing and failing chips' states. They traced these differences backward in time and through the design's logic cones. In the end, they isolated the problem to a single flip-flop containing an incorrect logic value at a particular clock cycle. Looking at the logic driving that flip-flop's input, they saw clearly that the input was driven by a minimal-

sized buffer using a long tristateable bus wire to which a bus keeper was connected.

Apparently, under some operating conditions, this buffer could not drive the input of this flip-flop high against the bus keeper before the start of the next clock cycle. These conditions did not show up during presilicon timing verification or the initial structural test on ATE. Once the designers discovered this, they replaced the buffer with a larger version in the planned silicon revision. The designers estimated that the debug support saved up to four times in time and up to eight times in effort compared to unsupported debug.

Xetal-II

The debug functionality of the Xetal-II SoC consists of scan-based silicon debug, combined with a hardware breakpoint on the instruction address of the internal global control processor, a specialized controller.[10] The external debugger software polls the internal breakpoint register's status until the breakpoint is hit, and scans out the complete system state afterward. Chip states can be stored for later analysis. Once a state dump is created, the circuit switches back to functional mode, and if more state dumps are needed, the debugger software repeats the task.

En-II

The debug infrastructure implemented on the En-II SoC consists of a mix of functionality from prior SoCs, with several improvements in debug capability.[11] Run-stop control is available over the embedded processors, including an ARM1176 CPU and a TriMedia VLIW DSP. Cross-breakpoint support is provided between the embedded processors, the on-chip trace buffer, and the trace output interface. Clock control disables all functional clocks on a breakpoint, after which debugger tools can access the chip's complete state via the TAP, using functional as well as scan chain accesses. The infrastructure provides elastic real-time trace observability for execution of embedded processors, and both an on-chip trace buffer and external data acquisition equipment can capture the real-time trace information. Special environment sensors monitor the maximum process frequency, voltage drop, and local on-die temperature variations.

Next-generation SoCs will contain more programmable processors, a scalable communication infrastructure (such as a network on chip), and many dedicated hardwired functions. At runtime, many software and hardware execution threads will be active simultaneously. Thus, these SoCs will require solutions for multithreading, multiprocessor, and communication debug that go beyond the capabilities offered by today's DFD. Existing DFD infrastructures will have to be extended to accommodate these new requirements.

Limited on-chip debug data storage and off-chip debug data bandwidth drive the need for design time criteria as to what debug data should be observable in a running system. Establishing such criteria requires a better analysis of what system data provides the best debug support for a particular SoC in combination with an efficient DFD implementation.

Another promising avenue is active reuse in the postsilicon validation process of debug techniques, hardware modules, and tests from the presilicon verification process. Examples include reusing coverage-based random and targeted test benches and reusing assertions in hardware to check system properties and communication protocols in a running system.[12] Reusing presilicon verification IP reduces the effort and time needed to find the cause of a problem and helps identify possible gaps in the verification process. The semiconductor industry and several academic partners have started looking into this kind of reuse, and first results are now available. It will be an area of increased activity in coming years. ■

References

1. A. Hopkins and K. McDonald-Maier, "Debug Support for Complex Systems On-Chip: A Review," *IEE Proc. Computers and Digital Techniques*, vol. 153, no. 4, July 2006, pp. 197-207.
2. S.K. Goel and B. Vermeulen, "Data Invalidation Analysis for Scan-Based Debug on Multiple-Clock System Chips," *J. Electronic Testing: Theory and Applications*, vol. 19, no. 4, Aug. 2003, pp. 407-416.
3. P. Dahlgren, P. Dickinson, and I. Parulkar, "Latch Divergency in Microprocessor Failure Analysis," *Proc. Int'l Test Conf.* (ITC 03), IEEE CS Press, 2003, pp. 755-763.
4. K. Goossens et al., "Transaction-Based Communication-Centric Debug," *Proc. 1st Int'l Symp. Networks-on-Chip* (NOCs 07), IEEE Press, 2007, pp. 95-106.
5. R. Leatherman and N. Stollon, "An Embedded Debugging Architecture for SoCs," *IEEE Potentials*, vol. 24, no. 1, Feb.-Mar. 2005, pp. 12-16.
6. "CoreSight Technology System Design Guide," ARM, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dgi0012b.

7. K. Holdbrook et al., "MicroSPARC: A Case Study of Scan-Based Debug," *Proc. Int'l Test Conf.* (ITC 94), IEEE CS Press, 1994, pp. 70-75.
8. M. Abramovici et al., "A Reconfigurable Design-for-Debug Infrastructure for SoCs," *Proc. 43rd Design Automation Conf.* (DAC 06), ACM Press, 2006, pp. 7-12.
9. B. Vermeulen, S. Oostdijk, and F. Bouwman, "Test and Debug Strategy of the PNX8525 Nexperia Digital Video Platform System Chip," *Proc. Int'l Test Conf.* (ITC 01), IEEE CS Press, 2001, pp. 121-130.
10. A. Abbo et al., "Xetal-II: A 107 GOPS, 600 mW Massively Parallel Processor for Video Scene Analysis," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, Jan. 2008, pp. 192-201.
11. B. Vermeulen and S. Bakker, "Debug Architecture for the En-II System Chip," *Computers and Digital Techniques*, vol. 1, no. 6, Nov. 2007, pp. 678-684.
12. M. Boule, J.-S. Chenard, and Z. Zilic, "Debug Enhancements in Assertion-Checker Generation," *Computers and Digital Techniques*, vol. 1, no. 6, Nov. 2007, pp. 669-677.

Bart Vermeulen is a senior scientist at NXP Semiconductors, the Netherlands. His research interests include the design, validation, and debug of embedded-system chips. He has an MSc in electrical engineering from the Eindhoven University of Technology, the Netherlands. He is a member of the IEEE.

■ Direct questions and comments about this article to Bart Vermeulen, NXP Semiconductors, High Tech Campus 37, Room 2.52, 5656 AE Eindhoven, the Netherlands; bart.vermeulen@nxp.com.

For further information on this or any other computing topic, please visit our Digital Library at http://www.computer.org/csdl.

Giving You the Edge

IT Professional magazine gives builders and managers of enterprise systems the "how to" and "what for" articles at your fingertips, so you can delve into and fully understand issues surrounding:

- Enterprise architecture and standards
- Information systems
- Network management
- Programming languages
- Project management
- Training and education
- Web systems
- Wireless applications
- And much, much more …

IT Professional

www.computer.org/itpro