Communication-centric SoC Debug using Transactions

Bart Vermeulen*, Kees Goossens*†, Remco van Steeden‡, and Martijn Bennebroek§
*NXP Semiconductors Research / SOC Architectures and Infrastructure
5656 AE Eindhoven, The Netherlands, Email: {Bart.Vermeulen,Kees.Goossens}@nxp.com
†Computer Engineering, Technical University Delft, The Netherlands
‡Testable Design and Test of Integrated Systems, Technical University of Twente, The Netherlands
§Philips Research, IC Design Group, Eindhoven, The Netherlands

***Abstract*— The growth in System-on-Chip complexity puts pressure on system verification. Due to limitations in the pre-silicon verification process, errors in hardware and software slip through to the stage when silicon and the complete software stack are first brought together. Finding the remaining errors at this stage is becoming increasing difficult. We propose that debugging should be communication-centric at first and based on transactions. We combine run-time, on-chip abstraction of system data to the transaction level, with system-level debug control over the communication infrastructure. We prove our concepts and architecture with a gate-level implementation that includes a Network-on-Chip, breakpoint monitors, clock and reset control (all programmable through an IEEE 1149.1 TAP), and give a quantification of the associated hardware cost.**

I. Introduction

The functional requirements for high-volume, electronic appliances have been and continue to be the main drive behind the introduction and use of process technologies with ever decreasing feature sizes. Because of the consumer demand for more features in a single product, and for cost benefits, the number of transistors that are integrated on a single die doubles every 18 months. Over three decades the semiconductor industry has sustained this increase in transistor density. This large number of transistors is used to implement both programmable processor cores, which can execute embedded software, and dedicated peripheral functions, which either implement interfaces for standard communication protocols (e.g. USB, I^2C, and PCI Express), or hardware accelerators for common and configurable processing tasks (e.g. MPEG2 video and MP3 audio encoding and decoding, audio up- and down-sampling, and video scaling).

The industry has however not only seen an exponential increase in number of transistors per die, but also a similar exponential increase in the number of source code lines in the embedded software stack, required to access and control all device features in a user-friendly manner. These exponential trends are putting a significant burden on system verification [1]. On the one hand because the number of system use cases is rapidly growing, and on the other hand because the Time-to-Market is under continuous pressure to ensure newer products are delivered to the market on time and ahead of the competition.

During pre-silicon verification, e.g. using simulation or emulation, models have to be used that may be inaccurate with respect to the physical characteristics of the final silicon. Verification resources always lag by one processor generation and/or process technology, forcing a trade-off between modeling accuracy and number of use cases that can be extensively verified. As a result, errors in hardware and software slip through to the stage when silicon and the complete software stack are first brought together. Finding the remaining errors at this stage is becoming increasingly difficult. The high level of integration causes a significant reduction in internal observability, hampering debug methods in observing and determining the root cause of any undesired behavior. A comprehensive system debug methodology is required to effectively and efficiently find these root causes. As embedded systems consist of embedded software and hardware, the debug requirements for both software and hardware need to be considered.

Support for debugging software on a single processor core has been in use for a very long time. With increasing integration, solutions are now becoming commercially available that support the debugging of software applications, distributed over several, possibly heterogeneous processor cores [2], [3]. These solutions provide rudimentary multi-processor debug features, such as cross-breakpoints (i.e. routing a breakpoint from one processor core to one, a subset of, or all other processor cores), and real-time processor trace (to output performance statistics on for example the number of cache misses, CPU stall cycles, and conditional jumps that change the software execution flow). The amount of system-level debug support required to facilitate debug of software that runs on multiple processor cores is however still a topic of research. To determine how much an application developer really benefits from these debug support functions still requires further study, as systems that incorporate them are only now appearing in the market.

Many hardware debug features have been reported on in the past [4], [5]. The two most commonly used features are (a) non-intrusive hardware trace, and (b) run-stop control. In the non-intrusive, hardware trace method, key signals inside the hard-wired IP cores are selected at design time, and brought out onto dedicated chip pins via a dedicated interconnect. Alternatively these signals can be internally stored in an embedded trace buffer, for read-out at a later point in time. The advantages of this method is that the behavior of key internal signals can be observed in real-time, preventing especially timing bugs from escaping detection. A clear drawback of this method is that these signals need to be selected up-front, at SoC design time, limiting the flexibility in observability once silicon has been manufactured. Research is currently on-going to help determine the best signals to select for observation using this real-time trace method. It is to be expected that this hardware trace method will in future SoCs be merged with the real-time processor trace method described above, to reduce the amount of dedicated, on-chip routing resources required for debug.

Figure 1 shows a high-level overview of traditional run-stop debug methods. Shown are two IP cores, with a communication interconnect. A breakpoint is programmed in one of the monitors observing the IP to determine the point in time at which the execution of that IP core has to be stopped. The rest of the system is stopped in response using a debug control interconnect. Once the system has

stopped its execution, intrusive access methods can be applied to query and, if required, modify the system state.

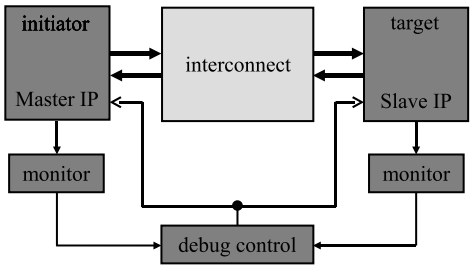

Fig. 1. Traditional Computation-centric Debug Control

One popular method for intrusive access is the re-use of the manufacturing test scan chains [6]. The reason for the popularity of this method lies in the fact that these scan chains (1) are already required for high-quality manufacturing test, and (2) can be easily accessed in a system environment via a standard IEEE 1149.1 Test Access Port (TAP) [7] with only a small amount of additional, on-chip hardware. After state examination, the system execution can be resumed, or restarted. Obtaining state dumps at several points during the execution of a failing scenario allows engineers to more quickly zoom in on the time and location of the failure's root cause. This method however suffers from drawbacks related to the low-level at which the debug information becomes available.

Firstly, when the scan chains are used as the access mechanism, the complete chip state is returned as a large number of individual bits. Debug tools are then required to back-annotate and help correlate this data to the design database and abstraction levels designers are familiar with. Advances have been made to bring this information back to the RTL and system transaction level [8], [9], [10], which makes the interpretation of this data and the subsequent explanation of any undesired system behavior easier. Secondly, this information is often extracted at the level of individual clock cycles. Thirdly, modern SoCs typically contain multiple clocks, running at different frequencies and phases. One problem that has to be addressed at this low level is the non-determinism and divergence in state data between multiple runs using the same breakpoint setting [11], [12].

In this paper, we contribute a new, run-control-based debug methodology that remedies this non-determinism, by combining run-time, on-chip abstraction of system data to the transaction level with appropriate, system-level debug control over the communication infrastructure. This methodology forms a natural complement to the existing, state-of-the-art methods that debug software and hardware in isolation, and when combined provide the foundation for a consistent and complete SoC software and hardware debug framework.

The remainder of this paper is organized as follows. Section II present the two key concepts of our SoC debug approach. In Section III we introduce the key components of a Network-on-Chip, which in Section IV is used to explain how our approach improves the debugability of SoCs. Finally, this paper concludes with Section V.

II. Trans-action-level Communication-centric Debug

Existing software and hardware debug methods cannot be efficiently combined in the same debug framework due to the large distance between the abstraction level used for debugging by the application software programmer and the one used by the hardware designer. The application software programmer analyzes and debugs erroneous behavior by examining an application's source code in a software debug tool that provides (1) a view of the state of the programmer's model of the processor (e.g. including register file content, and condition flags), (2) processor execution control (e.g. start, stop, and single step at the level of source code lines), and (3) a limited view on the state of the hard-wired peripherals, restricted to those peripheral registers accessible from the processor through, for example, MMIO reads and writes. The hardware designer debugs erroneous behavior of a hard-wired peripheral preferably by examining waveforms traces of internal signals, states of internal state machines, and, when scan chains are reused [6], even individual flipflop bits. A hardware debug tool can correlate these bits to appropriate flipflops and back-annotate this data to either gate-level or RTL descriptions of the design [10].

A. Transaction-level Debug

The recent introduction of the concept of transactions for pre-silicon system verification [13] provides an intermediate abstraction level between those traditionally used by the application programmer and hardware designer. Using the transaction level, software engineers and hardware designers can share a common abstraction level at which they can both contribute to the process of locating the root-cause of a system error for SoC debug.

For software engineers, the transaction level is the lowest level at which the embedded processors can be programmed by issuing read and/or write instructions. These read and write instructions cause transactions on the on-chip communication infrastructure, through translation to transaction commands using the appropriate communication protocol. The communication architecture transports these commands to one or more targets, which implement the actual write and/or read operation. As such, there is a natural correspondence between read and write instructions in software, and transactions within the system's communication infrastructure.

A hard-wired target is designed to respond to read and write commands on its communication interface that is connected to the system's communication infrastructure. When a read or write command is delivered to the target, the hardware designer knows how this target should react to this command. For example, when the target in question is a memory core, and the command is a write command, then the appropriate reaction of the target to the delivery of this write command is to store the command's data at the command's address.

The system can be viewed at the level of transactions, with the processors initiating read and write transactions, peripherals reacting to these transactions when they are delivered, and the communication infrastructure linking the initiators and targets together and transporting these transactions. Each transaction has an associated initiator and intended (set of) target(s). Inspecting transactions and detecting either missing transactions or transactions with incorrect attributes (such as address or data values), enables a quick identification of a suspect initiator and suspect target(s). By extending the debug scope further to include the communication infrastructure, the transaction level does not only allow the identification of the suspect initiator and target(s), but also a suspect path through the communication infrastructure. This identification allows for a large set of on-chip IP cores to be quickly discarded as the potential source of the problem, thereby greatly speeding up the debug process.

What remains is the required control over the communication path between the suspect initiator, through the communication infrastructure on the suspect communication path, to the suspect target(s). In

case of read instructions, also the complete return path has to be considered. To reach this point in the debug process, our approach relies on appropriate execution control of the communication infrastructure itself.

B. Communication-centric Debug

When the abstraction level for software and hardware debug is raised to the transaction level, it turns out to be extremely useful to extend the on-chip debug execution control to include not only the programmable processors, but also the communication infrastructure (see Figure 2).

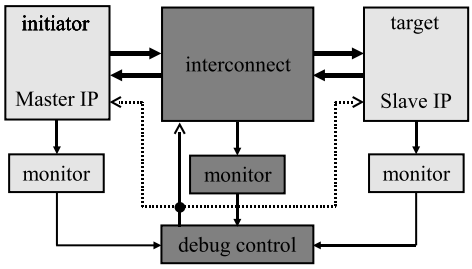

Fig. 2. Communication-centric SoC Debug

Control over the communication infrastructure allows finer-grain control over the generation, transportation, and delivery of transactions, thereby helping in the localization and isolation of suspect components. Figure 2 shows a (set of) monitor(s) to specifically observe the transactions that occur inside or at the edge of the communication infrastructure. Upon the detection of a transaction with specific characteristics (e.g. with a specific destination target, data value, address value, or frequency of occurrence), the monitor can signal to the debug control unit that the interconnect has to stop the transportation of transactions. When this breakpoint occurs, the communication infrastructure no longer accepts any read or write commands from the initiators (i.e. it prevents transaction generation), and it no longer delivers any read and write transactions to the targeted peripherals. It is up to the implementation of the communication infrastructure whether it still transports on-going transactions within the interconnect or whether this is also stopped.

In a properly designed system, i.e. where the SoC communication infrastructure uses communication protocols based on handshakes to interface with initiators and targets (as the commonly used AXI [14], OCP [15], and DTL [16] protocols do), each individual IP core will automatically stop its execution as well after the communication infrastructure has stopped. On the next interaction with the communication infrastructure, the IP core no longer is granted permission to communicate. As such all IP cores, and in fact the entire system reaches a functionally idle mode, where initiators and targets are waiting for the acceptance, respectively delivery of commands and data by the communication infrastructure. This acceptance can subsequently be controlled from, for example, external SoC debugger software, allowing very fine-grain, transaction-level control over the communication that takes place inside the SoC.

When the entire system is held in a functionally idle mode, it is safe to stop the functional clocks without danger of upsetting any functional communication. After the SoC has been completely stopped, i.e. when all transaction traffic is functionally halted, and the functional clocks are switched off, a core-based scan method [17] can be applied to efficiently inspect the complete internal state of the system to facilitate debugging. Such an architecture reuses the available IEEE 1149.1 TAP and associated controller [7] to configure all on-chip scan chains into a single, serial shift register. The content of this serial register can be scanned out through the TAP's TDO pin. The SoC state data that is obtained in this manner can be back-annotated to the SoC's design database (at gate-level or RTL) using, for example, the method described in [10].

III. Network-on-Chip

To validate our concepts on transaction-level, communication-centric debug, we applied our methodology to an SoC with a Network-on-Chip [18] as the communication infrastructure. A NoC was chosen for the following reasons:

- NoCs are commonly considered to be the most promising solution for the scalability issues in the SoC interconnect for deep sub-micron process technologies.
- Choosing a NoC as the SoC communication infrastructure helps magnify any problems related to methods that want to control the on-chip communication infrastructure. As such, a NoC more clearly exposes any problems with parallelism, latency, and scheduling, that might not become apparent in a single or multi-layered bus system.
- A NoC-based solution for efficient and effective debug communication control can more readily be ported to a single or multi-layered bus system, than the other way around.

A generic block diagram of the NoC architecture we used is given in Figure 3.

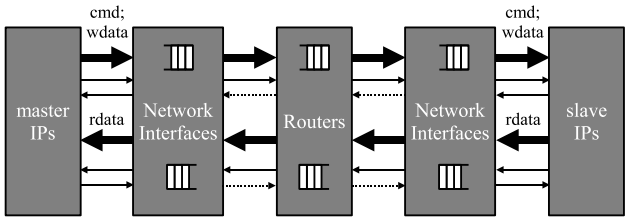

Fig. 3. Generic block diagram of an Æthereal NOC.

In Figure 3, a master IP core can initiate a transaction, containing command (cmd) and write data (wdata) and communicate it to a Network Interface (NI) using a particular interface protocol. The NI packs the transaction information in one or more network packets for transport through the network. These network packets are subsequently communicated from the NI to a set of routers. The routers are responsible for delivering the network packets to the NI, connected to the transaction's target IP core. The NI at the destination collects these network packets and reconstructs the original request made by the master IP core. This reconstructed request is applied to the communication interface of the target IP core using an appropriate interface protocol. The same process is followed for possible responses (rdata) from the target slave IP core. These responses are communicated via a slave-side NI and result in network packets that are sent to the master IP core, through the routers and master NI. The propagation of packets through our network occurs at the granularity of flits. Each flit contains three 32-bit words with 2-bits of sideband information. When available, a flit is transported from one NoC component to another in three clock cycles. This three clock cycle latency is relevant in relation to the distribution of key debug signals, as we discuss below.

Figure 4 shows the hardware modifications and extensions required to apply our transaction-level, communication-centric debug methodology to an SoC with this generic NoC architecture.

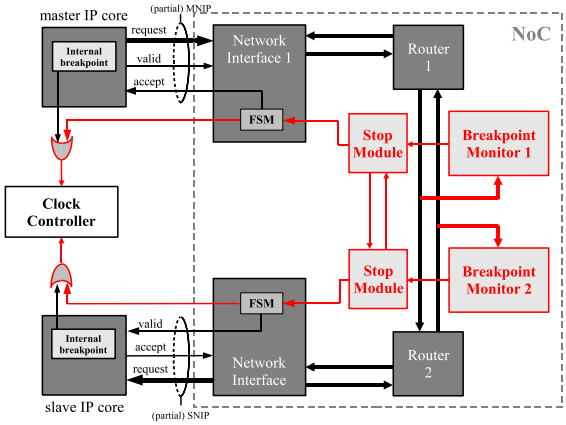

Fig. 4. Example NoC used in experiments.

Note that Figure 4 has intentionally been simplified, as it only shows the communication path from a single master IP core, through two network interface and two routers, to a single slave IP core. The return path naturally also exists, containing similar debug logic, and in practice, several more master and slave IP cores would typically be connected to the NoC, and more NIs and routers would be used inside the NoC.

NoC breakpoint signals are generated by a set of breakpoint monitors that monitor network connections [19], [20] and can be programmed either via an IEEE 1149.1 TAP or via the network itself. Monitoring inside the NoC helps reduce the number of monitors required when the number of NI ports is large, and also increases debugability of the NoC itself. Naturally monitors can also still be placed on the interface between the IP cores and the NoC. A stop module is added per NoC router. The connectivity of the stop modules uses the same topology as the router network, which allows the detection of a breakpoint condition anywhere in the network to be distributed to all stop modules as quickly and efficiently as possible. A fast distribution of the breakpoint signal is essential to minimize the chance of loosing, potentially crucial, debug information. In our implementation, the breakpoint signal travels from the breakpoint module, through the stop modules, to all network interfaces, at a rate of one clock cycle per network component. This is the highest possible transfer rate without imposing special constraints on the placement of pairs of network components in the design layout. Given that the network data itself is transfered with a latency of three clock cycles per network component, our implementation ensures us that we can always stop the NoC with the data that causes the breakpoint condition still present in the NoC itself. This is necessary when we want to validate and debug the delivery and subsequent processing of that data by its target slave IP core.

Inside the network interfaces, the breakpoint signal prevents transaction requests from any master IP core from being accepted (by keeping the accept signal inactive), and causes any data for any slave IP cores to be withheld (by keeping the valid signal inactive). This essentially freezes the functional communication between the IP cores and the NoC.

Some breakpoint monitors are traditionally added to the master, and occasionally also slave, IP cores to generate a breakpoint signal on a programmable, internal condition. Known examples are instruction and data address breakpoints in processor cores. In our approach these monitors are reused, and their breakpoint output signals combined with the breakpoint signals from the NoC. These breakpoint signals are then processed in a global Cross Trigger Module (CTM), which may either halt or stop one or more SoC components. Halting a component refers to keeping the components in a functionally idle mode, whereas stopping the component refers to gating its functional clock(s). Halting for example a microprocessor could involve forcing it to execute No-Operation (NOP) instructions. Halting a NoC could involve freezing the functional communication between the IP cores and the NoC, as we describe above.

Within the NoC we make a further distinction, on whether the Network Interfaces stop accepting data from the initiators only or whether they also stop providing data to their targets. We will refer to the former method as transaction-level stopping, and to the latter method as message-level stopping. Message-level stopping is more fine-grain than transaction-level stopping, and can be used during debug to determine which SoC component produces incorrect data or no data at all; the master IP core, the NoC, or the slave IP core.

Once all components are in a functionally idle mode, their clocks can be safely switched off. This is necessary when the scan chains are to be used to scan out the system state. Activating the scan chains while the functional clocks are still running, might cause glitches in the clock or data signals that corrupt the system state and render it useless. By first switching the clocks off, this condition is avoided. Figure 5 shows the scan-based debug architecture that is used to implement this system state access mechanism.

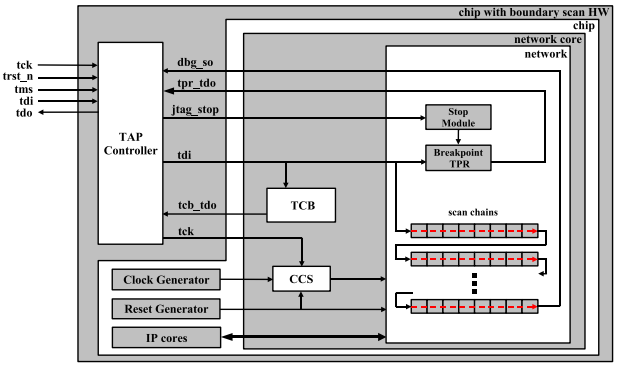

Fig. 5. Scan-based Debug Architecture.

To access the scan chains, the circuit mode is switched from functional mode to a test mode using a Test Control Block (TCB), accessible from an IEEE 1149.1 TAP. Once the test mode has been activated, all internal scan chains are concatenated into one long shift register that behaves as a user-defined Data Register of the chip-level TAP controller. A Clock Control Slice (CCS) allows the application of the TAP's TCK clock signal to the functional flipflops, which causes this register to shift its system state out onto the TAP's TDO pin on subsequent TCK cycles.

Figure 5 also shows our method for programming the breakpoint monitors through the use of a Test Point Register (TPR), which is another user-defined Data Register of the chip-level TAP controller, and the possibility to force a system stop from the IEEE 1149.1 TAP by asserting the jtag_stop signal from the TAP controller, using a special TAP instruction.

IV. Experimental Results

We have integrated our transaction-level communication-centric debug methodology in the Æthereal design flow. When an Æthereal NoC instance is generated from its high-level specification, the flow now also automatically instantiates the required debug modules. Breakpoints can be programmed in the NoC breakpoint monitors through an IEEE 1149.1 TAP.

Figure 6 shows three sets of signal traces, each of which shows the debug architecture in action in different use cases. The master IP core communicates with a slave IP core through DTL ports, via two NIs and two routers, as is shown in Figure 4. The signal traces show the request and response signals at the master-side NI port (MNIP) and at the slave-side NI port (SNIP).

The signal trace at the top of Figure 6 shows *normal operation* where the MNIP accepts four commands from the master IP core, as is shown by the four marked pulses on the dtl_cmd_accept signal immediately below the clock signal. The first and third command are write commands (as dtl_read_cmd is low), the other two are read commands. Each command transfers eight data words (wdata, and rdata respectively). The write command and data are transported from the MNIP to the SNIP and offered to the slave (dtl_cmd_valid is high), as illustrated by the solid arrows. Similarly, as is illustrated with the dashed arrows, the read command is accepted by the MNIP, transported to the SNIP and then offered to the slave IP core (dtl_cmd_valid is high). The slave responds with read data ("rdata"), which is transported and offered to the master (dtl_rd_valid). The master IP core accepts the read data before offering another write and read command.

In a *transaction-level* debug scenario (refer to the middle set of signal traces in Figure 6), the monitor at the router connected to the slave NI triggers an event and generates a stop signal for all NIs. The event is raised immediately after the first read command. Now only the master NI reacts to the stop signal (stop_in, in dashed circle). Thus the write and read commands are still offered to the slave, which in turn reacts as it did during normal operation. The master also still accepts this read data, and thus finishes all outstanding transactions. The NI tracks the completion of messages and does not accept any new commands after the stop event, not even when the master offers a new write command (dtl_cmd_valid is high). This is illustrated by the absence of the second set of write and read commands.

Using the TAP, external debugger software can poll the state of the stop modules and the NI's blocked signal to determine whether there no longer is any activity in the NoC. The NoC clock is then gated and replaced by the TAP's TCK clock signal for subsequent scan out of the complete SoC state using the scan chains on the TAP's TDO pin. This breakpoint detection and scan out phase could unfortunately not be shown in 6 due to lack of space.

The bottom set of signal traces in Figure 6 shows a *message-level* debug scenario. The same event is raised but now results in a stop signal to both master and slave NIs. The stop signal originates from the router that is closer to the slave NI, and therefore reaches the slave NI earlier than the master NI. As explained, this signal reaches the slave NI before the write command and data do. As a result, the message handshake is not initiated, and the NI keeps the write command and data in its FIFOs and does not offer them to the slave. This is evident from the absence of a pulse on the dtl_cmd_valid.

Our debug architecture requires only small changes to the functional architecture, and mainly involves adding the monitors and the event distribution interconnect. We have synthesized our example SoC using a commercially-available synthesis tool and a production-quality 130 nm CMOS technology library. The additional hardware area cost turned out to be around 4.5% of the NoC area, and less than 0.2% of the complete SOC area.

V. Conclusion

In this paper we addressed the debugging of complex SoCs with a novel, transaction-level, communication-centric debug methodology. Operating at the transaction level during debug, allows both application developers and hardware engineers to more easily contribute to the debug process. In addition, by extending existing, computation-centric debug architectures with debug control over the communication infrastructure, the difficulties of stopping and examining a multiple-clock SoC are avoided as functional clocks can now be safely stopped when the interconnecting communication infrastructure is functionally idle. We have proven these concepts and architecture with a gate-level implementation of a small SoC, which includes a NoC. Signal traces illustrate the capabilities to debug at both transaction and message level. We are currently extending this work by further validating these debug concepts on an FPGA NoC setup. In future we also intend to apply and evaluate this debug methodology to existing bus-architecture-based SoCs.

References

[1] B. Bailey, "A new vision of 'scalable' verification," *EETimes*, Mar. 2004.

[2] *CoreSight: V1.0 Architecture Specification*, ARM.

[3] R. Leatherman and N. Stollon, "An embedded debugging architecture for SoCs," *IEEE Potentials*, vol. 24, no. 1, pp. 12–16, Feb-Mar 2005.

[4] D. D. Josephson, S. Poehhnan, and V. Govan, "Debug methodology for the McKinley processor," in *Proceedings of the IEEE International Test Conference*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 451–460.

[5] A. Hopkins and K. McDonald-Maier, "Debug support for complex systems on-chip: A review," *IEE Proceedings Computers and Digital Techniques*, vol. 153, no. 4, pp. 197–207, July 2006.

[6] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, R. Raman, and M. Wong, "microSPARC: A case study of scan-based debug." in *Proceedings IEEE International Test Conference (ITC)*, 1994, pp. 70–75.

[7] IEEE Computer Society, *IEEE Standard Test Access Port and Boundary-Scan Architecture-IEEE Std 1149.1-2001*. IEEE Press, 2001.

[8] Y. Hsu, B. Tabbara, Y. Chen, and F. Tsai, "Advanced techniques for rtl debugging," in *Proceedings of the Design Automation Conference*, 2003, pp. 362–367.

[9] B. Tabbara and K. Hashmi, "Transaction-level modelling and debug of socs," in *Proceedings of the IP SOC Conference*, 2004.

[10] B. Vermeulen, Y.-C. Hsu, and R. Ruiz, "Silicon debug," *Test and Measurement World*, pp. 41–45, Oct. 2006.

[11] S. K. Goel and B. Vermeulen, "Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips," in *Proceedings IEEE International Test Conference (ITC)*, Oct. 2002, pp. 1103–1110.

[12] P. Dahlgren, P. Dickinson, and I. Parulkar, "Latch Divergency in Microprocessor Failure Analysis," in *Proceedings of the IEEE International Test Conference*, September/October 2003, pp. 755–763.

[13] B. Tabbara and K. Hashmi, "Transaction level modeling: Verification leaps ahead," *EDA Tech Forum*, pp. 14–17, Mar. 2005.

[14] *AMBA AXI Protocol Specification*, ARM, June 2003.

[15] OCP International Partnership, "Open core protocol specification," 2001.

[16] *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, Philips Semiconductors, July 2002.

[17] B. Vermeulen, T. Waayers, and S. Goel, "Core-based Scan Architecture for Silicon Debug," in *Proceedings IEEE International Test Conference (ITC)*, Baltimore, MD, USA, Oct. 2002, pp. 638–647.

[18] K. Goossens, J. Dielissen, and A. Rădulescu, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, Sept-Oct 2005.

[19] C. Ciordaş, T. Basten, A. Rădulescu, K. Goossens, and J. van Meerbergen, "An event-based monitoring service for networks on chip," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 4, pp. 702–723, Oct. 2005.

[20] C. Ciordaş, K. Goossens, A. Rădulescu, and T. Basten, "NoC monitoring: Impact on the design flow," in *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, May 2006, pp. 1981–1984.

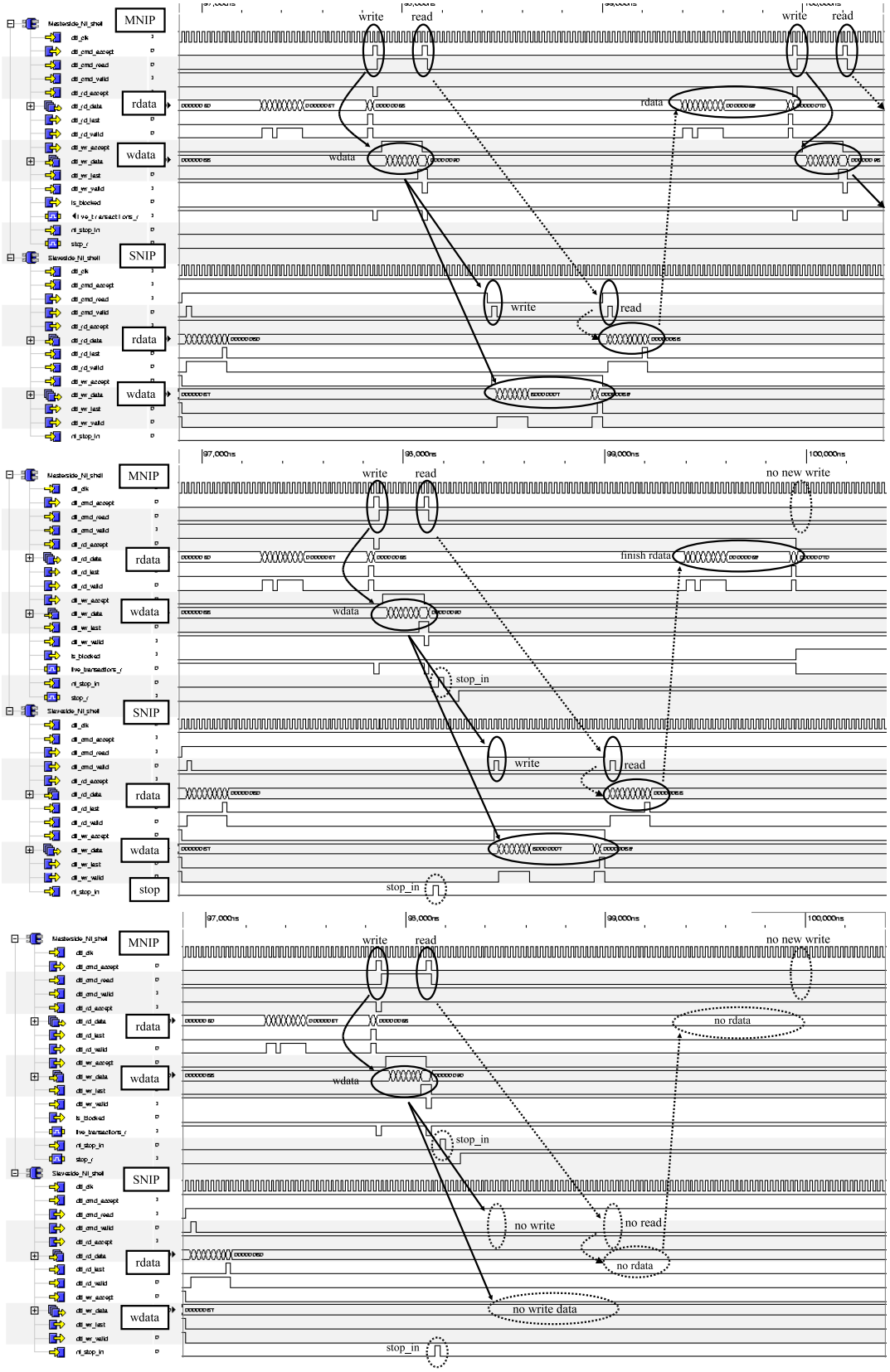

Fig. 6. Gate-level signal traces.