Interactive Debug of SoCs with Multiple Clocks

Bart Vermeulen
NXP Semiconductors

Kees Goossens
Eindhoven University of Technology

Editor's note:
Systems with elaborate multiple clock distributions are a necessity, and the authors address the postfabrication debug of such multiclock systems. Solutions, based on the authors' communication-centric debug approach, are presented that achieve a consistent snapshot of the system state and force the erroneous state in the face of nondeterminism.
—Zeljko Zilic, McGill University

Today's SoCs contain multiple programmable processor cores, hardware accelerators, and dedicated peripherals. In addition, a growing amount of embedded software runs on SoCs, causing both hardware and software complexity to increase rapidly. Many SoCs are built using a globally asynchronous, locally synchronous (GALS) design style to support tens of different clock domains and to facilitate layout, power management, and interfacing to the outside world.

Before silicon is manufactured, a SoC's correctness must be confirmed through formal verification, simulation, and emulation. These techniques provide confidence that no design errors were introduced and that the resulting chip will behave according to its specification. However, the number of use cases verified must be traded off with the amount of design detail—that is, the level of abstraction—included in the verification. Therefore, functional and electrical problems might go undetected at this stage because it is impossible to verify all use cases at the detail level of a physical implementation. For GALS SoCs in particular, verifying a SoC design's behavior for all combinations of clock frequencies and phases is not feasible. Prototype silicon, therefore, can still contain errors that manifest themselves only in the product, outside the controlled test and verification environment. Any remaining error must be found and removed as quickly as possible in postsilicon validation and debug. Industry benchmarks show that, on average, postsilicon debug consumes more than 50% of total project time.[1] This article introduces an approach we have developed to improve this process.

Why SoC debug is difficult

Debugging a SoC involves three nontrivial tasks: observing its state, obtaining a consistent state, and directing the SoC to the erroneous trace and state.

Observability

The first difficulty lies in the limited observability that a SoC provides of what happens inside it when it executes in its target environment and of why it doesn't exhibit its specified behavior (i.e., the problem's root cause). Ideally, we would use simulator-like functionality to inspect the state and operation of each intellectual property (IP) block in the chip, in as much detail as needed to analyze the erroneous behavior. Unfortunately, two factors constrain observability for silicon SoC implementations: the limited amount of debug information that we can stream through the device output pins in real time and the limited amount of on-chip memory that we can dedicate to capturing debug information without affecting system functionality or adding too much to final product cost.

One popular debug approach that overcomes the observability problem is the so-called interactive (or run/stop) technique, which stops an execution of the SoC before its state is inspected in detail. An advantage of this technique is that we can inspect the SoC's full state without running into the device pins' speed limitations. It also requires only a small amount of additional debug logic in the SoC.[2] The main disadvantage of interactive debug is that because the SoC must be stopped prior to observing its state, the technique is intrusive.

0740-7475/11/$26.00 © 2011 IEEE Copublished by the IEEE CS and the IEEE CASS

Sampling and consistency

Regardless of the specific debug technique used, finding an error requires analysis of the SoC state. Because on-chip communication delays are not negligible in large SoCs, it is not possible to instantaneously stop and observe the entire state of GALS SoCs. SoCs are therefore similar to distributed systems, for which obtaining a consistent state snapshot is a difficult problem.[3] A consistent SoC state consists of IP block states that are both locally and globally consistent. Local consistency allows the combination of single-bit values, as stored in the flip-flops in the IP block, to be interpreted as a valid functional IP state (counter values, instructions, etc.). Global consistency enables correlation and interpretation of locally consistent states across IP blocks. Achieving global consistency is difficult because taking a global snapshot is not instantaneous. IP block states can evolve during this time, and data must be captured in more than one place. For example, a write transaction could be captured first in a CPU and then, on its way to memory, in the interconnect.

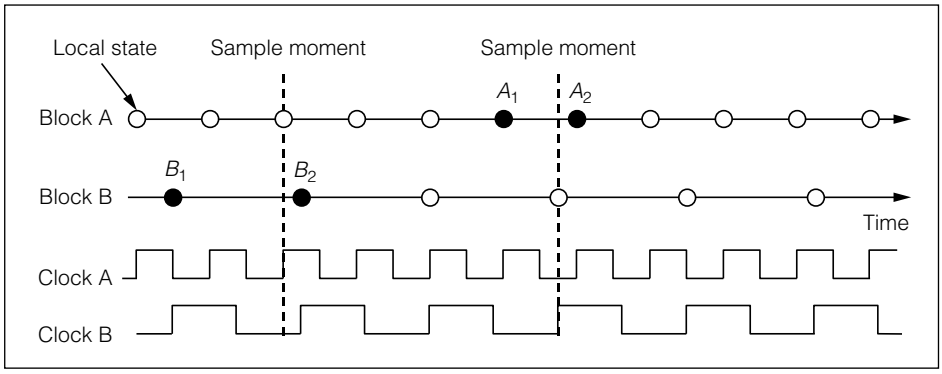

Figure 1. The sampling problem in a globally asynchronous, locally synchronous (GALS) SoC with multiple clocks.

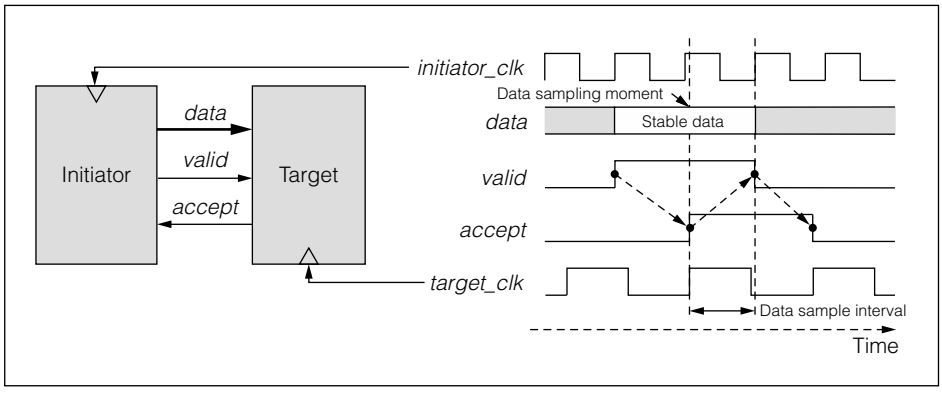

Figure 2. Handshake-based communication between clock domains.

Sampling an IP block's state with its own clock signal ensures local consistency. However, in a GALS SoC, there is no single moment at which the clocks of all IP blocks coincide. Figure 1 illustrates this problem; it shows the timeline of two asynchronous IP blocks, A and B. Each circle indicates a state change in a block. When sampling the state of Block B using the clock signal of Block A as a sample signal, we observe either one of the two states—B_1 or B_2—for Block B (indicated by the black circles) or an invalid intermediate state. The same is true in sampling the state of Block A (state A_1 or A_2) using the clock signal of Block B.

The captured state of a GALS SoC, therefore, is not necessarily consistent. Synchronizing the moment of taking a snapshot with any of the clock signals creates the risk of sampling the state of an unrelated clock domain while it is still changing. This leads to metastability in (some of) the flip-flops in the observed clock domain and possibly to capturing a locally inconsistent state. Alternatively, sampling each IP block independently on its own clock at a different point in time results in locally consistent states that may have progressed to different extents. Hence, they might not be correlated in a globally consistent state.

Nondeterminism and erroneous state

Modern SoCs circumvent the sampling problem by using handshake-based IP communication protocols, such as the advanced extensible interface (AXI).[4] During a handshaked data exchange, the data on the initiator's output remains stable until the target has explicitly indicated that it has sampled the data. Figure 2 shows an example of a four-phase handshake protocol. The initiator prepares data on its data outputs before asserting its *valid* output signal. This signal is then synchronized inside the target. The target samples the data inputs when it sees an activated

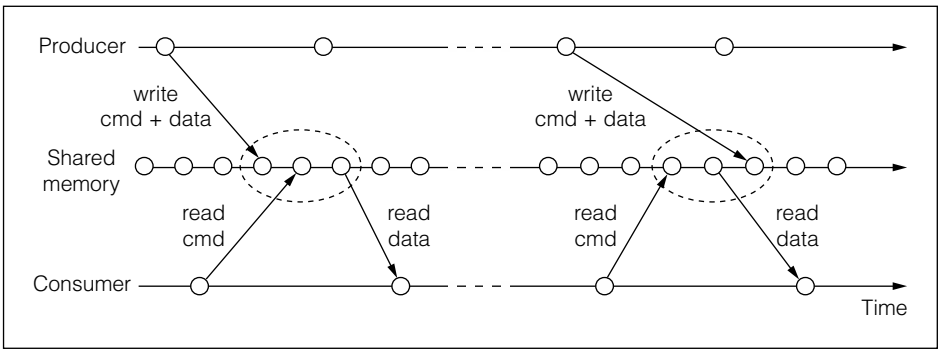

Figure 3. Multiple interleavings caused by arbitration. (cmd: command.)

valid signal, and then it asserts its *accept* output signal. This signal is synchronized in the initiator. The initiator deasserts its *valid* output when it sees the activated *accept* signal; then the target deasserts its *accept* signal.

Handshake-based communication ensures that initiator data outputs are held functionally stable for the handshake's duration, ensuring that the target can sample the data correctly. This process can take multiple clock cycles in each initiator and target clock domain because the time it takes the initiator (or target) to decide whether this signal is asserted depends on the amount of time between signal assertion and the initiator (or target) clock's active edge. The shorter this interval, the longer it can take the initiator (or target) to reach a decision.[5] As an unavoidable consequence of asynchronous chip I/O and the GALS design style, the clock cycle in which the target sees the valid data is nondeterministic—it depends on relative clock frequencies, clock phases, and fluctuations in the signal level's settling to a stable value.

Because communication between IP blocks takes a variable amount of time, their behaviors can vary from one execution run to the next. Even sampling each IP block i on the same local clock cycle c_i does not yield a constant result from run to run.[6] It may thus be necessary to rerun the SoC many times before an error is reproduced because the error might depend on unlikely timings of data transfers.

Furthermore, with more than two IP blocks, the nondeterministic behavior at the clock-cycle level propagates to higher abstraction levels, such as the transaction level. When asynchronous IP blocks share a resource, an arbiter must decide the order in which requests from multiple IP blocks are processed. Requests from different IP blocks may arrive in different clock cycles at the arbiter's inputs over multiple executions of the SoC, leading to different execution interleavings. (Using a handshaking asynchronous arbiter instead of a sampling synchronous arbiter incurs the same problem.[5])

Figure 3 shows an example of this problem. Here, the requests from the producer and consumer to the shared memory are very close in time. Small differences in clock frequency and/or phase between execution runs can change the order in which the arbiter receives and processes these requests. This can affect system state at the transaction level. For example, the result of a read operation by the consumer can depend on whether a write operation by the producer preceded it or not. By introducing nondeterminism at the transaction level, the GALS design style further complicates error reproducibility.

An error can therefore be intermittent—that is, it might not occur in all traces, which would make it permanent. For simplification, we assume that errors are constant—that they persist after they occur and are not transient. They are also *certain*—there is no probing effect making the debug observation itself intrusive and changing the observed SoC's behavior from run to run.[7]

Problem statement

In summary, although the GALS design style solves many timing and scalability issues, it introduces two fundamental problems for debug:

1. *how to obtain a consistent global state*, when no instantaneous distribution of a sample clock is possible, and when there is no sample clock aligned with all IP clocks to avoid metastability or inconsistent data; and
2. *how to force the SoC to arrive in an erroneous state*, when in each SoC execution run, each asynchronous communication may use a different number of clock cycles to synchronize, leading to different traces at both the clock-cycle and transaction levels.

To address these problems, we have developed the CSAR (for communication-centric, scan-based, abstraction-based, run/stop-based) debug approach.

CSAR debug approach

The CSAR approach abstracts from absolute time by raising the moment of state sampling from inconsistent local clock cycles to the level of handshakes of globally consistent IP communication protocols. The on-chip architecture supports the distribution of events that safely (but potentially at a nondeterministic time) stop the communication handshakes and hence communication between IP blocks. This ensures that the states of individual (single-threaded) IP blocks are stable and hence can be sampled deterministically and that the states of different IP blocks are consistent with each other; that is, every data element (message) is found either in the state of the sender or receiver IP block or in the state of the channel between them. In addition, through guided replay of selected transactions, we force execution to continue along a subset of traces toward a trace that exhibits the error.

The CSAR approach's most important ingredient is the temporal abstraction of clock cycles to handshakes and transactions, which has several consequences.

First, almost all IP blocks stall when they cannot send or receive data. Disabling the *valid* and/or *accept* signals of IP ports stabilizes the internal state, which then can be sampled on any clock. Disabling communication handshakes allows observation of locally consistent states, alleviating problem 1.

Second, using temporal abstraction of clock cycles to handshakes makes the communication between two IP blocks deterministic. The variations in the nondeterministic number of clock cycles for a single communication action are simplified to a deterministic handshake (data was either transferred or not), partially addressing problem 2. The abstraction to handshakes does not completely eliminate nondeterminism because the decision to stop can be located very close in time to the handshake, potentially causing nondeterminism in the decision to stop before or after the particular handshake. As a consequence, however, the resulting nondeterminism in state will affect a larger set of correlated data (e.g., an entire transaction), allowing that data to be more easily interpreted, unlike individual bits.

Third, the previous item also contributes to solving problem 1 by guaranteeing that communicating asynchronous IP blocks are consistent with each other, since data is either in the sender (if the handshake did not occur) or in the receiver (if the handshake did occur). Moreover, data is never duplicated (e.g., due to oversampling the slower clock) or lost (e.g., due to undersampling the faster clock). Since this consistency holds for all communicating IP blocks, it in turn guarantees global consistency. The event distribution interconnect (EDI), described later, ensures that IP blocks stop as soon as possible, such that IP states have minimally progressed beyond the moment at which the system was commanded to stop.

Finally, temporal abstraction removes the variation in time (the number of clock cycles per handshake) but not the nondeterministic interleaving of transactions for more than two IP blocks. The latter still causes the multiple traces of Figure 3. Although we can solve this problem by enforcing a static order (interleaving) for all arbiters in the SoC,[8] this is quite restrictive and often wasteful in performance. Therefore, we allow nondeterministic interleavings in normal execution runs. For debugging, we must find an erroneous trace, which, ideally, we can then replay at will. Deterministic replay, the recording and replaying of a particular order of arbiter decisions (at particular local clock cycles), can be expensive.[9] For this reason, the CSAR approach incorporates monitors for nonintrusive observation of the SoC until a particular event of interest occurs. On this event, the distributed protocol-specific instrument (PSI) components enforce a particular local order of handshakes and, hence, arbitration interleavings. The SoC is thereby guided to the erroneous state, alleviating problem 2. However, the PSIs are currently directed from off-chip debugger software via the IEEE Std. 1149.1-2001 test access port (TAP), which is onerous, and the guiding process remains challenging.

Communication-centric debug

In traditional, computation-centric debug approaches, we observe computation inside IP blocks, especially embedded processors. When an important internal event occurs, we can take specific debug actions, such as stopping the computation in some or all IP blocks.

With an increasing number of processors, communication and synchronization between IP blocks grow in complexity and become a major source of errors. To complement mature computation-centric processor debug methods, the CSAR approach also allows debugging the communication and synchronization between IP blocks.

Older on-chip interconnects, such as the advanced peripheral bus (APB) and ARM high-performance bus (AHB), are single-threaded and process only one transaction at a time. As a result, the interconnect forces a unique trace for all IP blocks even when a GALS design style is used. For scalability and performance, newer interconnects, such as multilayer AHB and AXI buses and networks on chip (NoCs),[10] are multithreaded. In other words, they allow both multiple transactions between a master and a slave (pipelining) and concurrent transactions between different masters and slaves. Therefore, no unique trace exists for these newer interconnects.

The aim of communication-centric debug in the CSAR approach is to observe and control the traces that the interconnect, and hence the IP blocks attached to it, follow. This gives insight into the communication and synchronization between IP blocks and allows (partially) deterministic replay.

Scan-based debug

Because only a limited amount of trace data can be stored on-chip or sent off-chip, in the CSAR approach we allow the user to observe state only when the system has been stopped. We reuse the scan chains that embedded systems use for manufacturing test to create access to all state in the chip's flip-flops and memories via the TAP.[11] This helps minimize silicon area cost.

Run/stop-based debug

Because state can be observed only via the scan chains when the system has stopped, we use non-intrusive monitoring and run/stop control to stop the system at interesting points in time. We implement these functions by monitoring a subset of the system state and generating events on programmable conditions. These assertions can be distributed—that is, they can involve multiple monitors at different locations—and sequential—that is, they can consider the state at different points in time. The EDI broadcasts events at high speed to the monitors and the PSIs.

Ideally, we would deterministically follow the erroneous trace. However, rather than collecting and storing information for replay, we first monitor (unintrusively) for a happening of interest. Then, we iteratively guide the system toward the error trace by disallowing particular communications, thereby forcing execution to continue along a subset of system traces. Thus, we can iteratively refine the set of system traces to a unique trace that exhibits an error. This process can be interpreted as partially deterministic replay, or guided replay, although errors can become uncertain because the debug process is intrusive, occurring only after the system has stopped and using off-chip debugger software.

Abstraction-based debug

In the CSAR approach, we use temporal abstraction to limit the frequency and number of observations to those of interest. Rather than observing a port between an IP block and the interconnect at every clock cycle, the monitors, by abstracting to handshakes, consider only the clock cycles in which information is transferred. Conventional computation-centric debug can be used in combination with this approach to observe the internal behavior of IP blocks.

For example, an AXI transaction request consists of a command and a number of data words. Each of these can be individually abstracted to a handshake. Similarly, a response consists of a number of data words. A message is a request or a response, and a transaction is the request together with the (optional) response. Figure 4 shows temporal abstraction from clock cycles, via handshakes and transactions, to distributed shared memory. At each step, we combine a number of events into a coarser event that is meaningful and consistent in itself.

As Figure 4 shows, we also use structural and behavioral abstractions. Our debug observability involves reusing the scan chains to retrieve the functional state (the bits in registers and memories) from the chip when the system has stopped. This provides intrusive access to state from the chip. The resulting state dump is a sequence of bits that is still mapped to logical registers and memories in gate- and register-transfer-level descriptions. Modules are one level higher, corresponding to the structural design hierarchy. These abstraction levels only describe structure—that is, how gates and registers are (hierarchically) interconnected.

The next level makes a significant step in abstraction by interpreting structural modules as functional IP blocks. Information about an IP block's intended behavior lets us interpret sets of registers. For example, a simple IP block that implements a FIFO buffer contains data registers and read and write pointers. At the functional IP level, we can interpret the values

in the read and write registers and, for example, display only the valid entries in the data registers.

The higher abstraction levels, from channel to use case, go one step further. They abstract from hardware to software, or from the static design-time view to the dynamic runtime view—in other words, from structural components of the system to its logical view, or how it is programmed. Because we focus on communication, we move from structural interconnect components such as routers and network interfaces to logical communication channels and connections used by applications. Processors execute functions, which are part of threads and tasks that synchronize, which in turn are part of the complete application. Finally, use cases define the combinations of applications that run on the system, as required by the user.

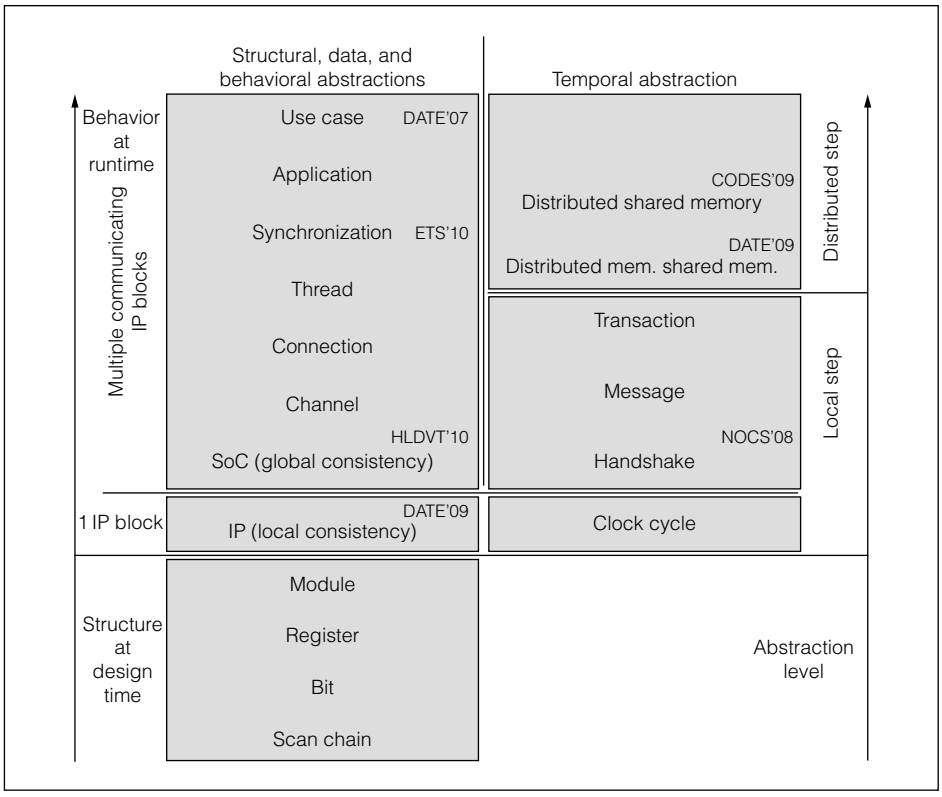

Figure 4. Overview of temporal, structural, and behavioral abstractions in the CSAR (communication-centric, scan-based, abstraction-based, run/stop-based) debug approach. (Abbreviations—e.g., ETS'10—next to a given item indicate conference proceedings that contain further information about the item.)

Experimental results

Figure 5 shows a multiple-clock SoC that illustrates our approach. CPU tiles 1 and 2, each with tightly coupled instruction and data memories, communicate via the NoC and two memory tiles, using the C-HEAP (CPU-controller heterogeneous embedded architectures for signal processing) software FIFO protocol.[12] The SoC uses clock domain crossing modules to communicate across clock domain boundaries. CPU tile 3 is responsible for initializing the NoC and is synchronous with it. The monitors, PSIs, and EDI (not shown) count handshakes on the communication interfaces and, if programmed, stop all communication on a particular handshake.

We defined two use cases. In case 1, all blocks operate with a clock period of 2,000,003 femtoseconds (approximately 500 MHz). For use case 2, we change the clock period of tile 1 to 3,000,016 fs (approximately 333 MHz), and of tile 2 to 5,000,011 fs (approximately 200 MHz). We chose these periods because

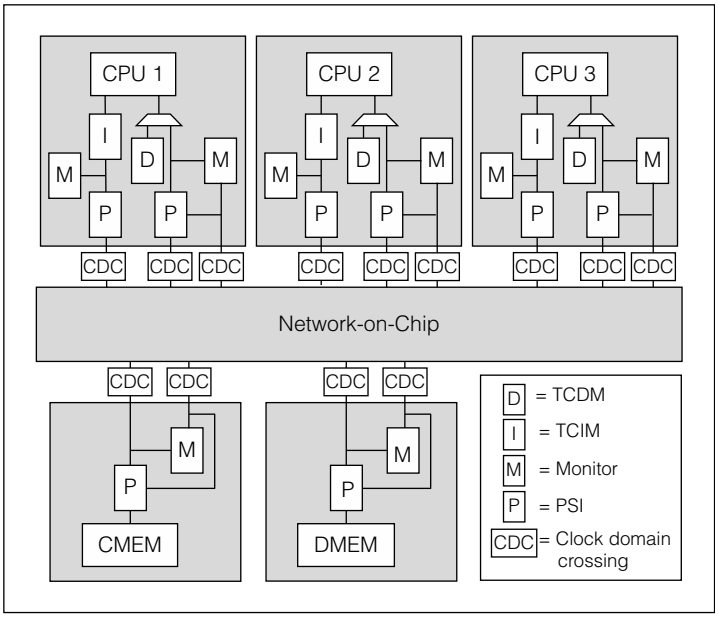

Figure 5. Example of a multiple-clock SoC. (CMEM: control memory; DMEM: data memory; PSI: protocol-specific instrument; TCDM: tightly coupled data memory; TCIM: tightly coupled instruction memory.)

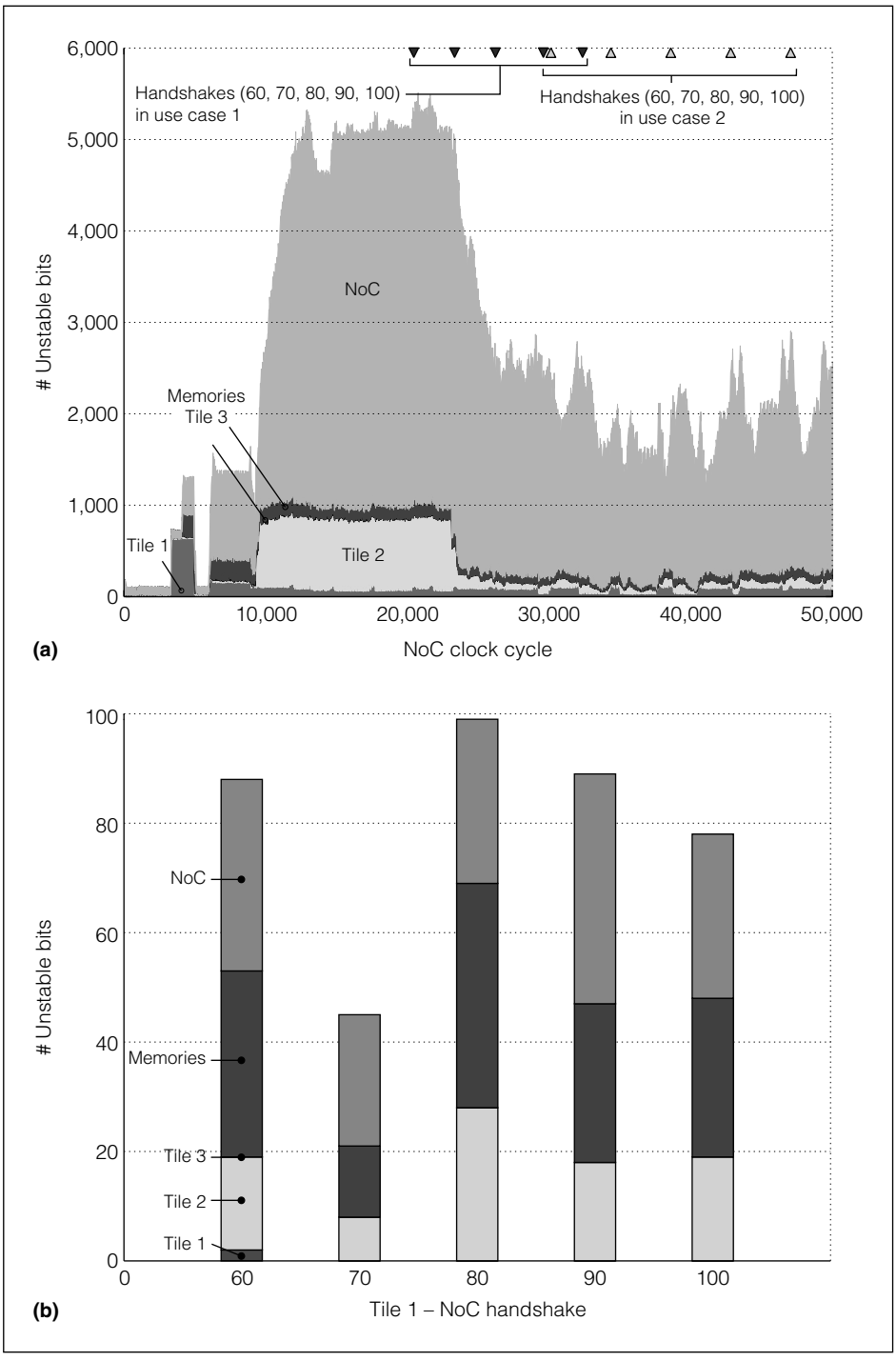

Figure 6. State unstableness: cycle level (a), and communication handshake level (b). By using the communication handshake level, the CSAR debug approach significantly lowers the number of unstable bits per observed state.

they are prime and they yield realistic operating frequencies.

We conducted two experiments. In the first experiment, we sampled the entire system state in the interval 0 to 50,000 NoC clock cycles and compared all state bits. (A bit is stable if it has the same value in both traces.) Figure 6a shows that the amount of state unstableness due to the use of multiple clocks varied between 0 and 5,533 bits, with an average of 2,617 bits.

In the second experiment (Figure 6b), we programmed our debug infrastructure to first stop the entire system on handshakes 60, 70, 80, 90, and 100 on the interface between tile 1 and the NoC. For each handshake, the monitor in tile 1 generated an event when the desired handshake on the tile's interface occurred. The EDI sent the events to all the PSIs, which then inhibited communication on the local interface. We then waited until all communication was stopped to extract the system state. This wait can take some time because IP blocks may have internal activity—for example, the NoC continues to move packets until they have all arrived at their destinations. Some internal activity might never cease, such as the NoC arbiter's counters.

We show the absolute times of breakpoints for both use cases at the top of Figure 6a. Note that in use case 2, where tile 1 ran at a lower frequency, the handshakes occurred later than in use case 1. With the CSAR approach, we corrected for this difference in time and prevented the unstableness it causes in system state.

Figure 6b shows the remaining unstableness in the entire state when stopping on the indicated communication handshakes. The remaining unstableness was predominantly located in registers on the boundaries of the clock domains that could nondeterministically sample signals from neighboring clock domains.

The values in these registers, however, are never used without a valid handshake, so their unstableness did not affect the clock domain's functional operation. Figure 6b shows that the CSAR approach yielded a significantly lower number of unstable state bits, easing state interpretation and thus improving system debuggability.

The CSAR approach effectively uses hardware- and software-based temporal, structural, and behavioral abstraction techniques in the debug process to obtain globally consistent states of a GALS SoC and guide its execution to an erroneous state. This guiding process remains challenging and is a subject of our ongoing research. ■

References

1. B. Roberts, "The Verities of Verification," *Electronic Business,* vol. 29, no. 8, 2003, pp. 26-32.
2. B. Vermeulen, "Functional Debug Techniques for Embedded Systems," *IEEE Design & Test,* vol. 25, no. 3, 2008, pp. 208-215.
3. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, 1985, pp. 63-75.
4. AMBA AXI Protocol Specification, ARM Ltd., June 2003.
5. J. Kinniment, *Synchronization and Arbitration in Digital Systems,* John Wiley & Sons, 2008.
6. 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.
7. B. Vermeulen and K. Goossens, "Debugging Multi-Core Systems on Chip," *Multi-Core Embedded Systems,* G. Kornaros, ed., CRC Press/Taylor & Francis Group, 2010, pp. 153-198.
8. M.W. Heath, W.P. Burleson, and I.G. Harris, "Synchro-Tokens: A Deterministic GALS Methodology for Chip-Level Debug and Test," *IEEE Trans. Computers,* vol. 54, no. 12, 2005, pp. 1532-1546.
9. M. Ronsse and K. de Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," *ACM Trans. Computer Systems,* vol. 17, no. 2, 1999, pp. 133-152.
10. K. Goossens and A. Hansson, "The Aethereal Network on Chip after Ten Years: Goals, Evolution, Lessons, and Future," *Proc. 47th Design Automation Conf.* (DAC 10), ACM Press, 2010, pp. 306-311.
11. B. Vermeulen, T. Waayers, and S.K. Goel, "Core-Based Scan Architecture for Silicon Debug," *Proc. Int'l Test Conf.* (ITC 02), IEEE Press, 2002, pp. 638-647.
12. A. Nieuwland et al., "C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems," *Design Automation for Embedded Systems,* vol. 7, no. 3, 2002, pp. 229-266.

Bart Vermeulen is a senior principal scientist at NXP Semiconductors, the Netherlands. His research interests include design and validation of robust distributed architectures for embedded automotive systems. He has an MSc in electrical engineering from Eindhoven University of Technology, the Netherlands. He is a member of IEEE.

Kees Goossens is a professor of real-time embedded systems in the Electrical Engineering Department of Eindhoven University of Technology. His research interests include composable, predictable, low-power embedded systems. He has a PhD in computer science from the University of Edinburgh. He is a member of IEEE.

■ Direct questions and comments about this article to Bart Vermeulen, NXP Semiconductors, High Tech Campus 32, Room 0.26, 5656 AE Eindhoven, The Netherlands; bart.vermeulen@nxp.com.