Interactive debugging of systems on chip with multiple clocks

Bart Vermeulen, Kees Goossens

Abstract

Many systems on chip (SOCs) use a globally-asynchronous locally-synchronous (GALS) design style to solve timing and scalability issues. This design style however also introduces two fundamental problems for debug; (1) how to obtain a consistent state, and (2) how to force the SOC to arrive in an erroneous state. The CSAR debug approach presented here addresses these problems. It uses temporal abstraction from clock cycles to communication handshakes and transactions as its key ingredient, complemented by scan-based state access, and guided replay. Experimental results show that the CSAR approach improves stability of the state bits captured via scan and the repeatability of the execution trace in a GALS SOC.

1 Introduction

Present-day systems on chip (SOCs) contain multiple programmable processor cores, hardware accelerators, and dedicated peripherals. Besides hardware functionality, they contain a growing amount of embedded software that runs on these SOCs. Both the hardware and software complexity of SOCs increases rapidly. Many SOCs are built using a globally-asynchronous locally-synchronous (GALS) design style to support tens of different clock domains, either for layout or power management reasons, or to interface to the outside world.

Before silicon is manufactured the correctness of a SOC is verified using e.g. formal verification, simulation, and emulation. These techniques are used to gain confidence that no design errors were introduced and that the resulting chip should behave according to its specification. However, the number of use cases to verify must be traded off against the amount of design detail to include, i.e. the level of abstraction during verification. Hence functional and electrical problems may go undetected at this stage as it is impossible to verify all use cases at the level of detail of a physical implementation. In particular for GALS SOCs, it is not feasible to verify the behavior of a SOC design for all combinations of clock frequencies and phases.

Prototype silicon may therefore still contain errors that only manifest themselves in the product, outside of a controlled test and verification environment. Any remaining error has to be found and removed as quickly as possible using post-silicon validation and debug. We aim to improve this process, since industry benchmarks show that on average it consumes over 50% of the total project time [1].

2 Why debugging a system on chip is difficult

Each of the following parts of debugging a SOC is non trivial: *observing* its state, obtaining a *consistent* state, and *directing* the SOC to the erroneous trace and state. We discuss each in turn.

2.1 Observability

The first problem with debugging a SOC lies in the limited observability into what happens inside the chip when it executes in its target environment, and why it does not exhibit its specified behavior (i.e. finding the root cause of the problem). Ideally one would like to use simulator-like functionality to inspect the state and operation of each intellectual property (IP) block in the chip, in as much detail as required to analyze the erroneous behavior. Unfortunately this observability is constrained for silicon implementations of a SOC due to (1) the limited amount of debug information that can be streamed out through the device output pins in real-time and (2) the limited amount of on-chip memory that can be dedicated to capture debug information, without affecting the functionality of the system or increasing the final product cost too much.

One popular debug approach is to use a so-called *interactive* (or run-stop) debugging technique, where the execution of the SOC is first stopped before its state is inspected in detail. One advantage of this technique is that the full state of the SOC can be inspected, without running into the speed limitations of the device pins. It also requires only a small amount of additional debug logic in the SOC [2]. The main disadvantage of interactive debugging is that it is intrusive: the SOC must be stopped prior to observing its state.

2.2 Sampling and Consistency

Regardless of the specific debug technique used, finding the error requires analysis of the SOC state. Since 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, where obtaining a consistent snapshot of the state of a distributed system is a known problem [3]. A consistent state of a SOC 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 the correlation and interpretation of the locally-consistent states across IP blocks. This is not trivial since taking a global snapshot is not instantaneous. IP block states can evolve during this time, and data has to be captured in more than one place. For example, a read transaction could be captured first in a central processing unit (CPU), and then in the interconnect on its way to the memory.

Sampling the state of an IP block with its own clock signal ensures local consistency. However in a GALS SOC no single moment in time exists at which the clocks of all IP blocks coincide. This is illustrated in Figure 1; we show the time line of two asynchronous IP blocks A and B with a circle to indicate a state change in a block. When sampling the state of Block B using the clock signal of Block A as sample signal, either one of the two states B1 or B2 for Block B, indicated by the black circles, or an invalid intermediate state, is observed. The same occurs when sampling the state of Block A (state A1 or A2) using the clock signal of Block B.

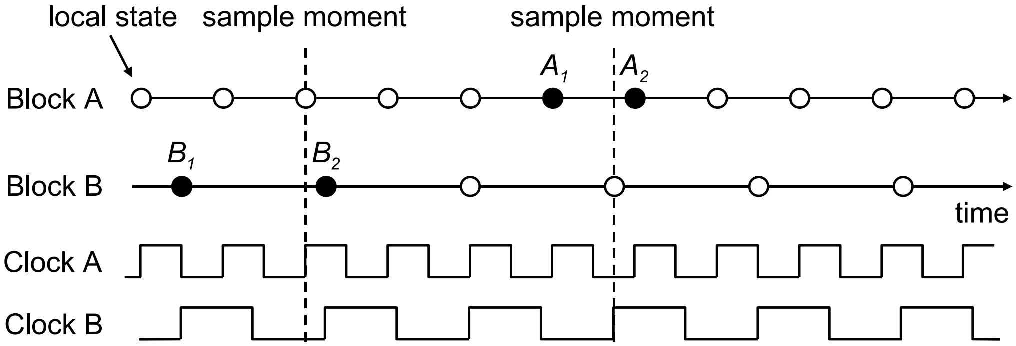


Fig. 1. Sampling problems with multiple clocks.

The captured state of a GALS SOC is therefore 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 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 may not be correlated in a globally consistent state.

2.3 Non-determinism and Erroneous State

Modern SOCs circumvent the sampling problem described earlier by using *handshake*-based IP communication protocols, such as the advanced extensible interface (AXI) [4]. During a handshaked data exchange, the data on the output of an initiator is held stable until the target has explicitly indicated that it has sampled it. An example of a four-phase handshake protocol is shown in Figure 2.

Data is prepared by the initiator on its data outputs before its valid output signal is asserted. This signal is then synchronized inside the target. The target samples the data inputs when it sees an activated valid signal, and asserts its ready output signal. This signal is synchronized in the initiator. The initiator de-asserts its valid output when it sees the activated ready signal, after which the target de-asserts its ready signal. Handshake-based communication ensures that the initiator data outputs are held functionally stable for the duration of the handshake, ensuring that the target can sample the data correctly. This process can take multiple clock cycles in each of the initiator and target clock domains, as the time it takes the initiator (target) to decide whether this signal is asserted are not, depends on the amount of time between the assertion of this signal and the active edge of the initiator (target) clock. The shorter this interval, the longer it can take the initiator (target) to reach a decision due to possible metastability [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 *non-deterministic* since it depends on the relative clock frequencies, clock phases, and fluctuations in settling to a stable value.

Since 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

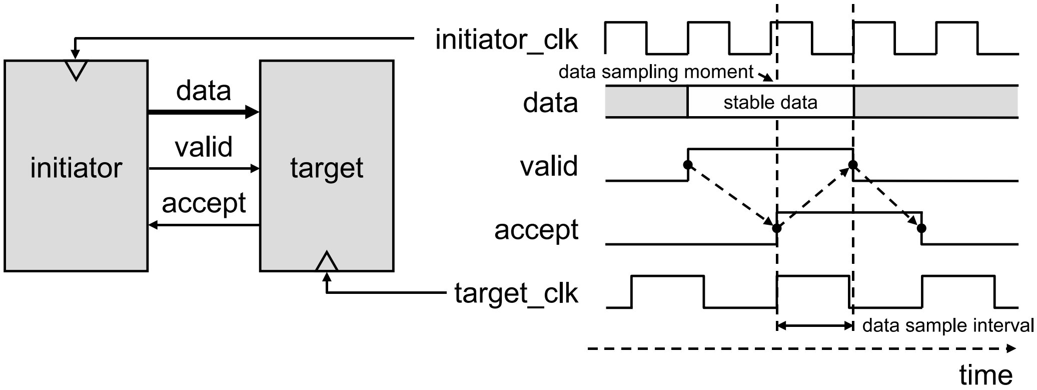


Fig. 2. Handshake-based communication between clock domains.

result from run to run [6]. It may thus be necessary to rerun the SOC many times before an *error is reproduced*, as it may depend on unlikely timings of data transfers.

Furthermore, with more than two IP blocks, the non-deterministic behavior at the clock-cycle level propagates to higher levels of abstraction, such as the transaction level. When asynchronous IP blocks share a resource, an arbiter is required to decide the order in which the requests from multiple IP blocks are processed. As detailed above, the requests from different IP blocks may arrive in different clock cycles at the inputs of this arbiter over multiple executions of the SOC, leading to different execution interleavings. (Using a handshaking asynchronous arbiter instead of a sampling synchronous arbiter suffers from the same problem [5]). An example of this is shown in Figure 3.

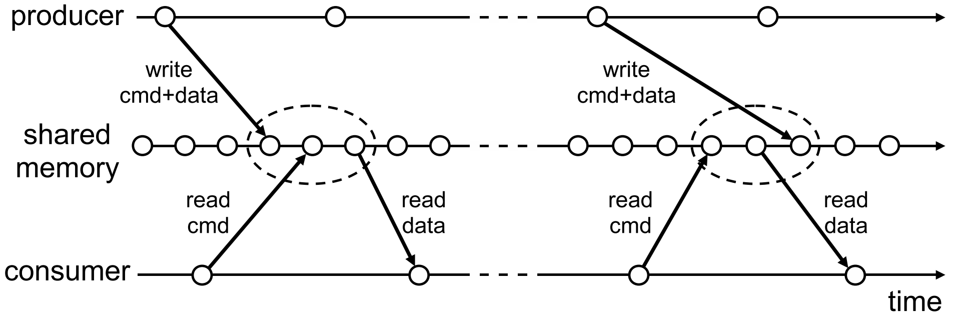


Fig. 3. Multiple interleavings due to arbitration.

The requests from the producer and consumer to the shared memory in Figure 3 occur very close in time. Small fluctuations may change the order in which the arbiter receives and processes these requests. This may affect the state of the system at the level of transactions, as the result of e.g. a read operation by the consumer may be different, depending on whether a write operation by the producer was processed before or after it. Hence GALS also introduces *non-determinism at the transaction level*, further exacerbating error reproducibility.

An error can therefore be *intermittent*, i.e. not occur in all traces (which would make it *permanent*). For simplification, we assume that errors are *constant*, i.e. they persist after they occur and are not *transient*. They are also *certain*, i.e. there is no "probing effect" whereby observation is intrusive and changes the observed SOC's behavior [7].

2.4 Debugging 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 exists no sample clock that is aligned with all IP clocks to avoid metastability or inconsistent data.
2) *How to force the SOC to arrive in an erroneous state*, when in each execution run of the SOC each asynchronous communication may use a different number of clock cycles to synchronize, leading to different traces at both clock-cycle and transaction levels.

The next sections describe the CSAR approach that we are developing to address these problems.

3 The CSAR debug approach

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

1) Observe that almost all IP blocks stall when they cannot send or receive data. Disabling the valid and/or ready signals of IP ports, makes the internal state stable, which then *can be sampled on any clock*. By disabling communication handshakes, *locally consistent* states can be observed, which alleviates Problem 1.
2) Using *temporal abstraction* of clock cycles to handshakes, the communication between two IP blocks is made *deterministic*. The variations in the non-deterministic number of clock cycles for a single communication action are simplified to a deterministic handshake (data was either transferred or not), which partially addresses Problem 2. Note that the abstraction to handshakes does not completely eliminate the non-determinism, as the decision to stop may be located very close in time to the handshake, potentially causing non-determinism in the decision to stop before or after the particular handshake. As a consequence however, the resulting non-determinism in state will affect a larger set of correlated data (e.g. an entire transaction), allowing it to be more easily interpreted, opposed to individual bits.
3) The previous step also contributes to solving Problem 1, by guaranteeing that communicating asynchronous IP blocks are consistent with respect to each other, since data is either in the sender (if the handshake did not take place) or in the receiver (if the handshake did take place), and is never duplicated (e.g. due to oversampling the slower clock) or lost (e.g. due to undersampling the faster clock). Since this holds for *all* communicating IP blocks, this in turn guarantees *global consistency*. The event distribution interconnect (EDI), described below, ensures that IP blocks are stopped as soon as possible, such that IP states have minimally progressed beyond the moment at which the system was informed to stop.
4) Temporal abstraction removes the variation in time (the number of clock cycles per handshake), but not the non-deterministic interleaving of transactions for more than two IP blocks. The latter still results in the multiple traces of Figure 3. Although it can be solved by enforcing a static order (interleaving) for all arbiters in the SOC [8], this is quite restrictive and often wasteful in performance. For this reason, we allow non-deterministic interleavings in normal execution runs. For debug, an erroneous trace has to be found, which, ideally, can then be replayed at will. *Deterministic replay* [9] requires recording and replaying one particular order of arbiter decisions (at particular local clock cycles), which can be expensive.

   For this reason, CSAR incorporates *monitors* for non-intrusive observation of the SOC until a particular happening of interest. On this event, the distributed *protocol-specific instrument (PSI)* components are used to enforce a particular local order of handshakes, and hence arbitration interleavings. In this manner, the SOC can be guided to the erroneous state, to alleviate Problem 2. However, the PSIs are 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.

The CSAR debug method can be characterized as a communication-centric, scan-based, abstraction-based, run/stop-based approach. Each characteristic is described in more detail below.

3.1 Communication-Centric Debug

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

With an increasing number of processors, the communication and synchronization between the IP blocks grow in complexity and become an important source of errors. To complement mature existing computation-centric processor debug methods, the CSAR approach also allows debugging the communication 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 any point in time. Hence the interconnect forces a unique trace for all IP blocks even when using a GALS design style. For scalability and performance reasons, recent interconnects, such as multi-layer AHB and AXI buses, and networks on chip (NOCs) [10], are multi-threaded. In other words, they allow both multiple transactions between a master and a slave (pipelining), and concurrent transactions between different masters and slaves. Hence no unique trace exists anymore, as we saw in Section 2.3.

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

3.2 Scan-Based Debug

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

3.3 Run/Stop-Based Debug

As the state can only be observed via the scan chains when the system has been stopped, *non-intrusive monitoring* and *run/stop control* are used to stop the system at interesting points in time. This is implemented by non-intrusively monitoring a subset of the system state, and generating events on programmable conditions. These assertions may be distributed, i.e. involve multiple monitors at different locations, and sequential, i.e. consider the state at different points in time. The EDI broadcast events at high speed to monitors and PSIs.

Ideally we deterministically follow the erroneous trace. However, rather than collecting and storing information for replay, we first monitor (non intrusively) for a happening of interest. Following this, we iteratively guide the system toward the error trace by disallowing particular communications and thereby forcing execution to continue along a subset of system traces. This allows the user to iteratively refine the set of system traces to a unique trace that exhibits an error. This may be interpreted as *partially* deterministic replay, or "guided replay," although errors may become uncertain, as this process is currently intrusive because the guidance of the system does not occur in real-time, but only after the system has been stopped, and is done using off-chip debugger software.

3.4 Abstraction-Based Debug

We use temporal abstraction to reduce the frequency and number of observations to those that are of interest. In particular, rather than observing a port between an IP and the interconnect at every clock cycle, the monitors only consider those clock cycles where information is transferred, i.e., by abstracting to handshakes. Conventional computation-centric debug can still be used in combination with this approach to observe the internal behavior of the IP blocks.

As an 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. Each time we combine a number of events to a coarser event that is meaningful and consistent in itself.

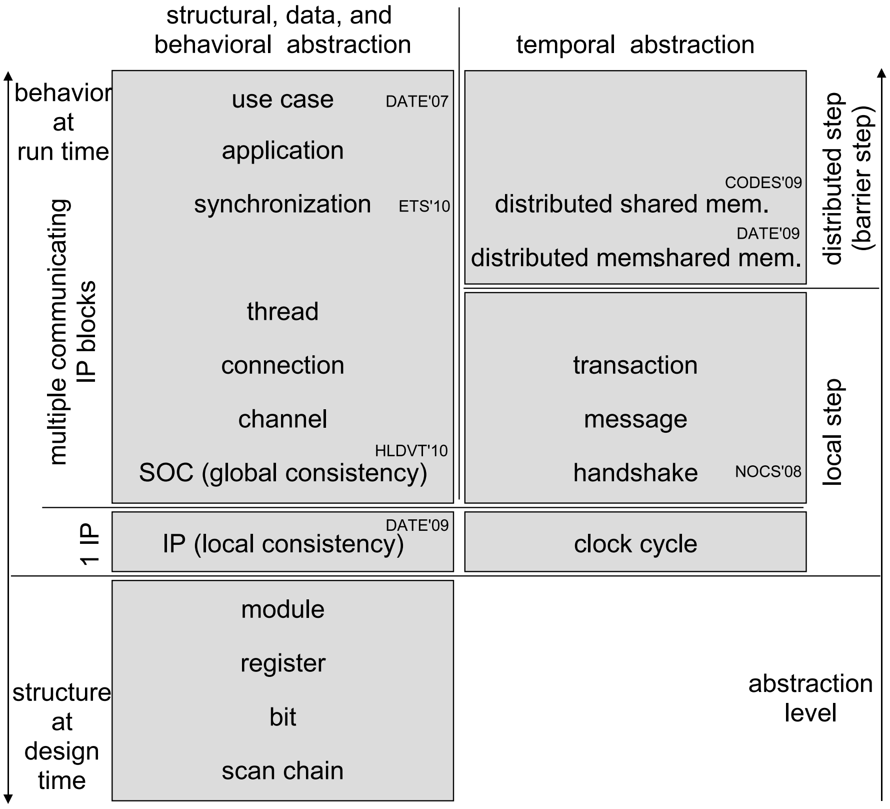


Fig. 4. Debug abstractions.

We also use *structural and behavioral abstractions*. Our debug observability involves re-using the scan chains to retrieve the functional state (i.e., the bits in registers and memories) from the chip when the system has stopped. This provides intrusive access to the state from the chip. The resulting state dump is a sequence of bits that still is mapped to logical registers and memories in gate-level and register transfer level (RTL) descriptions. One level higher are modules, which correspond to the structural design hierarchy. These abstraction levels only describe structure, i.e., 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 on the intended behavior of an IP block allows us to interpret sets of registers. For example, a simple IP block, which implements a first-in first-out (FIFO) 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 levels of abstraction, 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 run-time view, in other words, not from what components the system is constructed from, but to its logical view, i.e. how it has been programmed. Because we focus on communication, we move from structural interconnect components such as routers and network interfaces to logical communication channels and connections that are used by applications. Processors execute functions, which are part of threads and tasks that synchronize, which themselves in turn are part of the complete application. Finally, the use cases define which combinations of applications run on the system, as required by the user.

4 Experimental results

Figure 5 shows the multiple-clock SOC we will use to illustrate our approach. Processor tiles 1 and 2, each with tightly-couple instruction and data memories, communicate via a NOC [10] and two memory tiles, using the C-HEAP software FIFO protocol. Clock domain boundaries are crossed using clock domain crossing (CDC) modules. Tile 3 is responsible for initializing the NOC and is synchronous with the NOC. The monitors, PSIs, and EDI (not shown) are used to count handshakes on the communication interfaces, and, if programmed, stop all communication on a particular handshake.

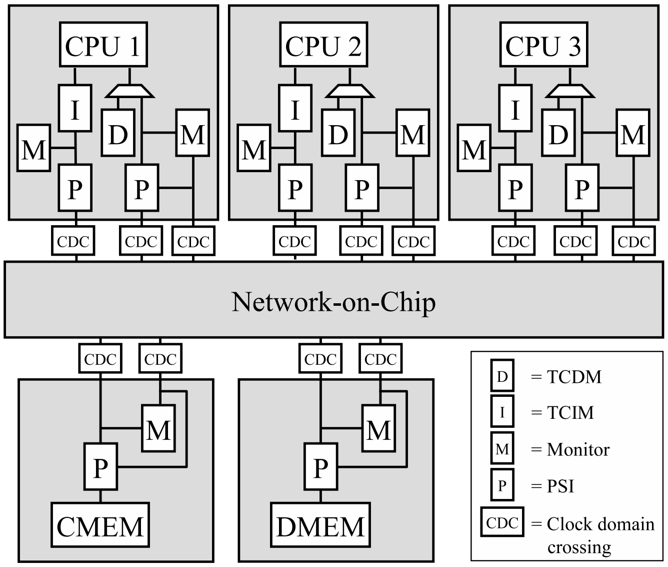


Fig. 5. Example multiple-clock SOC.

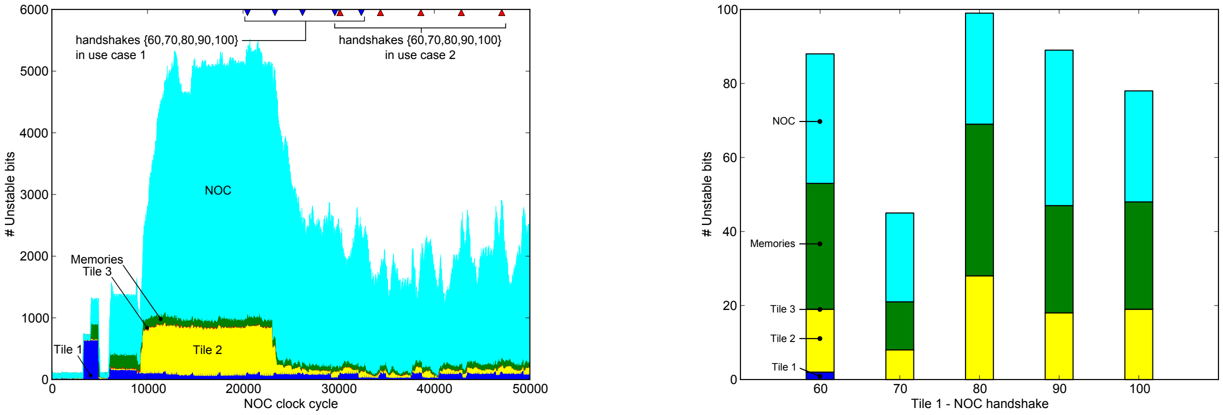
We defined two use cases. In use case 1, all blocks operate with a clock period of 2,000,003 fs (approx. 500 MHz). For use case 2, we change the clock period of tile 1 to 3,000,016 fs (approx. 333 MHz), and of tile 2 to 5,000,011 fs (approx. 200 MHz). These periods were chosen because they are prime and yield realistic operating frequencies. We conducted two experiments. In experiment 1, we sampled the entire system state in the interval 0-50,000 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 varies between 0 and 5,533 bits, with an average of 2,617 bits.

We subsequently 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, the monitor in tile 1 generates an event when the desired handshakes on the tile's interface takes place. The EDI sends the events to all the PSIs, which then inhibit communication on the local interface. We then wait until the system is quiescent. This may take some time, since IPs may have internal activity; e.g., the NOC continues to move packets until they have all arrived at their destinations. Some internal activity may never cease, such as the NOC arbiter's counters.

The absolute times of breakpoints for both use cases are shown at the top of Figure 6a. Note that in use case 2, where tile 1 runs at a lower frequency, the handshakes occur later than in use case 1. With the CSAR approach we correct for this difference in time, and prevent the unstableness this difference causes in the system state.

Figure 6b shows the remaining unstableness in the entire state between the two use cases when stopping on the indicated communication handshakes. The remaining unstableness is predominantly located in registers on the boundaries of the clock domains that may non-deterministically sample signals from neighboring clock domains. As the values in these register are never used without a valid handshake, their unstableness does not affect the functional operation of the clock domain.

Figure 6b clearly illustrates that the CSAR approach yields a significantly lower number of unstable state bits, which eases state interpretation, and thereby improves system debugability.



(a) Cycle level

(b) Communication handshake level

Fig. 6. State unstableness

5 Conclusion

In this paper we introduced the CSAR approach to interactively debug GALS SOCs using consistent global states. In essence, we abstract from absolute time by raising the moment of state sampling from inconsistent local clock cycles to the level of handshakes of IP communication protocols that are globally consistent. The on-chip architecture supports the distribution of events that safely (but potentially at a non-deterministic time) stop the communication handshakes and hence communication between IP blocks. This ensures that (1) the states of individual (single-threaded) IPs are stable, and can hence be sampled deterministically, and that (2) the states of different IPs are consistent with each other, i.e. every data element (message) is found either in the state of the sender or receiver IP, or in the state of the channel between them. In addition, we described guided replay of selected transactions to force the execution to continue along a subset of traces towards a trace that exhibits the error.

References

[1] B. Roberts, "The verities of verification," *Electronic Business*, Jan. 2003.
[2] B. Vermeulen, "Functional Debug Techniques for Embedded Systems," *IEEE Design & Test of Computers*, vol. 25, no. 3, pp. 208–215, 2008.
[3] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
[4] *AMBA AXI Protocol Specification*, ARM, Jun. 2003.
[5] D. J. Kinniment, *Synchronization and Arbitration in Digital Systems*. Wiley Publishing, 2008.
[6] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch divergency in microprocessor failure analysis. In *Proceedings IEEE International Test Conference (ITC)*, pages 755–763, 2003.
[7] B. Vermeulen and K. Goossens, "Debugging multi-core systems on chip," in *Multi-Core Embedded Systems*, G. Kornaros, Ed. CRC Press/Taylor & Francis Group, 2010, ch. 5, 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 Transactions on Computers*, vol. 54, no. 12, pp. 1532–1546, Dec. 2005.
[9] M. Ronsse and K. de Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," in *ACM Transactions on Compuer Systems*, vol. 17, no. 2, May 1999, pp. 133–152.
[10] K. Goossens and A. Hansson, "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proceedings ACM/IEEE Design Automation Conference (DAC)*, Jun. 2010.

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