Obtaining consistent global state dumps to interactively debug systems on chip with multiple clocks

(Invited Paper)

Bart Vermeulen
Central R&D, NXP Semiconductors
Email: bart.vermeulen@nxp.com

Kees Goossens
Eindhoven University of Technology
Email: k.g.w.goossens@tue.nl

Abstract

Post-silicon debugging of a system on chip (SOC) is complex due to (1) the intrinsic limits on the internal observability, (2) the absence of a single global clock, and (3) the need for asynchronous intellectual property (IP) blocks to interact with each other. These aspects prevent the instantaneous capture of a complete and consistent state of the SOC, and make the SOC non-deterministic at both the clock cycle level and the behavioral level. To debug an embedded system when the states that are extracted are irreproducible and inconsistent is nearly impossible. In this paper, we therefore introduce a method to capture a consistent, complete state of a multiple-clock SOC for interactive debugging. We reuse the same functionality that is used to ensure correct functional communication between asynchronous IP blocks, namely the handshake signals common in on-chip communication protocols. We merge the required on-chip hardware to support this debug functionality with the traditional debug architecture that reuses the manufacturing scan chains for debug. Our experimental results show that it is possible to ensure a globally consistent state is observed when the system is stopped on a breakpoint event.

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 the software complexity of SOCs increases rapidly. The reuse of intellectual property (IP) blocks significantly reduces the time required to design a SOC. IP blocks however commonly use different clock signals, either for layout or power management reasons, or because they have to interface to the outside world via standardized, fixed frequencies. Many embedded systems today have tens of clock domains, and have to use a globally-asynchronous locally-synchronous (GALS) design style [1]. For simplicity we split the different GALS styles into synchronous (including mesochronous) and asynchronous (including plesiochronous and heterochronous) [2].

The correctness of a design from high-level specification, via register transfer level (RTL) and gate-level implementations, to physical chip layout, has to be verified before silicon is manufactured, using e.g. formal verification, simulation, and emulation. These techniques provide confidence that no errors were introduced and the resulting design should behave according to its specification. During this verification process, trade-offs have to be made between the amount of design detail to include and the number of use cases to verify. Functional and electrical problems may go undetected as it is impossible to verify all use cases at the level of the physical implementation. In particular for SOCs with asynchronously operating IP blocks, it is not feasible to verify the behavior for all combinations of internal clock frequencies and phases.

Structural test programs will find *manufacturing defects* after an SOC is manufactured, but seldom catch functional timing errors, for example in synchronization protocols. Certain problems may therefore only manifest themselves after this manufacturing test, and, even worse, outside of controlled test and verification environments such as automated test equipment, simulators, and emulators. The root cause of any remaining problem that is discovered during the post-silicon validation of a chip, has to be found and removed as quickly as possible to ensure that the product can be sold to the customer on time and for a competitive price. Industry benchmarks [3] show that validation and debug consumes over 50% of the project time.

Post-silicon debugging of a GALS SOC is a complex task [4], [5]. In this paper we analyze three reasons for this complexity: (1) intrinsic limits on the internal observability, (2) the absence of a single global clock, and (3) the need for asynchronous IP blocks to interact with each other. These aspects prevent *the instantaneous capture of the entire state* of the SOC for analysis. They also make the SOC *non-deterministic* at both the *clock cycle* level and the *behavioral* level.

To debug an embedded system when the extracted states are irreproducible and inconsistent is nearly impossible. In this paper, we therefore introduce a method and architecture to capture a consistent, complete state of a multiple-clock SOC for interactive debugging. We use the same functionality that is used to ensure correct functional communication between asynchronous IP blocks. The hardware to support this also reuses the manufacturing scan chains, as for scan-based silicon debug [8]. Our experimental results show that, using e.g. the valid/accept handshake signals common in most on-chip communication protocols as the basis for stopping the execution of the SOC, it is possible to ensure that a consistent global state is observed, when the system is stopped on a breakpoint event.

This paper is organized as follows. In Section 2, we analyze the complexity of debugging asynchronous, multiple-clock SOCs, and present prior work. Section 3 details our approach to debugging a GALS SOC, the required on-chip infrastructure, and proof that our approach leads to a consistent global state when the SOC is stopped on an event. Section 5 presents our experiments, and Section 6 concludes this paper.

2. Problem description and related work

2.1. Limited intrinsic observability

Post-silicon debugging is intrinsically limited by the lack of internal observability. A chip package, many metal layers, very small transistor dimensions, and a thick substrate all cause physical debug methods, such as mechanical probing, laser voltage probing, and laser assisted device alteration to be very time-consuming, error-prone, and expensive [6]. Hence, electrical debug methods, where functional or dedicated access paths are embedded in the chip to facilitate debugging, have been adopted. Still the amount of debug data that a SOC produces in real-time is huge. Consider a 10-million transistor design running at 100 MHz; sampling one signal per transistor per clock cycle produces 10^{15} bits per second. The exponential increase in the number of transistors on a single chip [7] compared to the (linearly increasing) number of input/output (I/O) pins makes it impossible to observe all electrical signals inside the chip at every moment.

Hence, people resort to a combination of two, complementary approaches; (1) *real-time trace*, and (2) *interactive debugging*. Real-time trace debug approaches store a subset of the system state in real-time into an on-chip or off-chip memory. Given silicon area and I/O speed constraints, this subset is necessarily small. In contrast, in the interactive debugging techniques, the execution of the system is stopped at a point of interest. Once stopped, the state of the execution can be inspected in detail without running into any real-time I/O limitations. Afterwards, the execution can e.g. be restarted, resumed, or stepped. Manufacturing test scan chains are often reused to interactively debug silicon [8]–[10]. They are used to extract the chip state, contained in the flip-flops and embedded memories, and store this *state dump* off-chip for subsequent analysis. This approach is (1) low cost, as the scan chains are already needed inside the chip to facilitate manufacturing test, and (2) comprehensive, as the full state can be extracted. Additional state restoration techniques can expand the scope of a state dump to include the combinational signals between the flip-flops [11], [12].

The execution of the chip first has to be stopped to activate the scan chains. Stopping occurs by programming a breakpoint before or during the execution of the chip, for a particular condition of interest at a given location. At run-time, the functional clock is then gated when this breakpoint is reached. Once stopped, the circuit can be switched into scan test mode, and the content of the scan chains and embedded memories extracted via the scan chains. Dahlgren et al. [5] automate this process of creating state dumps at multiple clock cycles using multiple execution runs for the UltraSPARC micro-processor. They show that it is possible to spatially and temporally localize design issues based on the early detection of significant deviations between state dumps obtained from known good and from known bad samples. Their method works well for fully-synchronous systems, or at the level of single IPs, where determinism at the clock-cycle level still exists. However as we will show in the next two subsections, there is often very little clock-cycle determinism between IP blocks at the system level in a GALS SOC.

2.2. Absence of a single global clock

To analyze the behavior of a malfunctioning SOC we need to examine its state. However, a GALS SOC lacks a single global clock that clocks all state elements.

This causes problems when trying to instantaneously sample multiple signals. The state of an IP block in one clock domain is not guaranteed to be stable when sampled with the clock of another, asynchronous clock domain. As a result, the sampled state in the former clock domain can be *inconsistent*, either in itself or with respect to the state in the latter clock domain. In general, it is not possible to sample consistent states of any two, asynchronous IP blocks at single points in time. Figure 1 illustrates this problem; the circles on each IP block's time line indicate a state change in the IP block. When sampling the state of the target using the initiator clock as the sample clock, either one of the two states indicated by the black circles, or an invalid intermediate state, is observed.

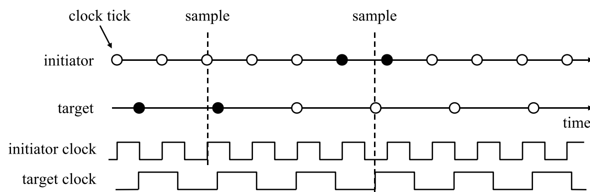

Figure 1. Sampling problems with multiple clocks.

When asynchronous IP blocks need to functionally communicate (or are shared), *synchronization* and *arbitration* have to take place. Both operations cause the SOC to become non-deterministic at the clock-cycle level, as explained below. To permit functional communication between asynchronous IP blocks, modern SOCs circumvent the sampling problem by using handshake-based communication protocols. All modern on-chip communication protocols, e.g. the advanced extensible interface (AXI) [13], the open core protocol (OCP) [14] and the device transaction level (DTL) [15] protocols, use a handshake mechanism to reliably communicate data between any two, asynchronous IP blocks. During the handshake, 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 to the target clock domain inside the target. Once the target sees the activated valid signal, it samples its data inputs, and asserts its accept output signal. This accept signal is then synchronized to the initiator clock domain inside the initiator. Once the initiator sees the activated accept signal, it deasserts its valid output, upon which the target subsequently deasserts its accept signal. Using a handshake-based communication protocol ensures that the data on the output of the initiator is held functionally stable for the duration of the handshake, and therefore ensures that the data is correctly sampled by the target.

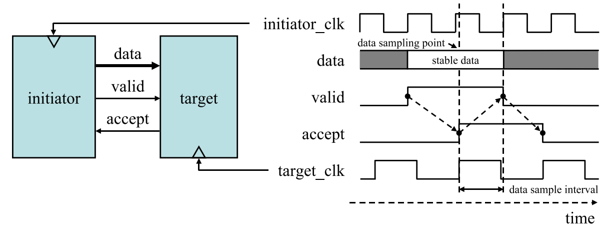

Figure 2. Handshake-based communication between clock domains.

2.3. Non-deterministic IP block interactions

For a target to observe that an initiator is offering data, the target first has to sample the valid control signal. The time it takes the 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 target clock. The shorter this interval, the longer it can take the target to reach a decision [16]. As the valid signal is asynchronous to the target clock, it is not possible to bound this duration. Hence the actual transfer of a single data element across an asynchronous clock domain boundary can take one or more target clock cycles, and makes the asynchronous IP interactions, and consequently the SOC *non-deterministic at the clock cycle level*.

A similar situation occurs when the local clock signals need to be gated, to allow inspection of the system state at a breakpoint via the scan chains. Inside a clock generation unit (CGU), an asynchronous stop request signal is sampled to decide whether or not to stop the functional clock this cycle. Figure 3 shows an example circuit and associated timing diagram to gate a functional clock, clk_in_x, when the stop request signal stop_req is asserted.

The output signal clk_out_x drives the clock inputs of the flip-flops and memories inside an IP block. The falling-edge triggered Flip-flop Y ensures that the output clock signal is never gated when it is active, as this may cause a glitch to appear on the output clock signal clk_out_x. Flip-flop X is needed to reduce the chance of metastability on Signal B. When the request signal is asserted, the output clock signal should be gated after one rising edge and one subsequent falling edge on the functional input clock. However, the closer the assertion of the request signal occurs to the active

edge of the functional clock signal, the longer the actual stopping of the clock domain can take. When the state of the IP block is not guaranteed to be stable during this synchronization process, the duration of the synchronization will determine the exact clock cycle at which the state is sampled, making the state, subsequently extracted via the scan chains, also *non-deterministic at the clock cycle level*.

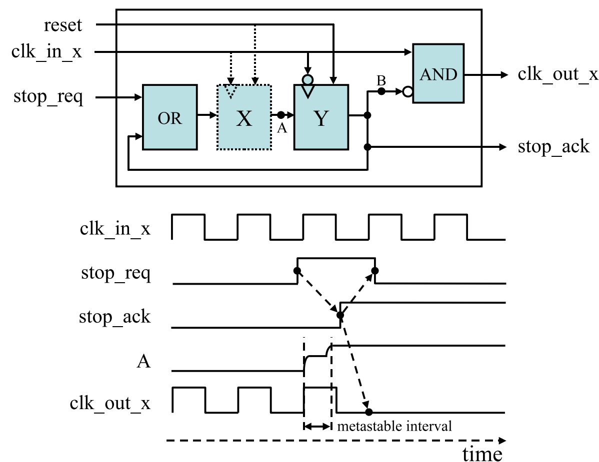

Figure 3. Example clock gate circuitry.

When asynchronous IP blocks need to share a (scarce) 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 at different clock cycles at this arbiter over multiple executions of the SOC. The time it subsequently takes the arbiter to reach a decision on which request to process first, depends on the amount of time between the arrival times of the requests at the arbiter. The shorter this interval, the longer the decision process can take in the arbiter [16]. An example of this is shown in Figure 4.

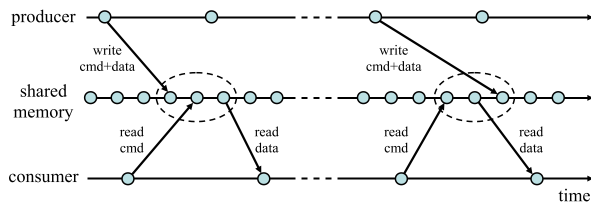

Figure 4. Multiple interleavings due to arbitration.

The requests from the producer and consumer to the shared memory in Figure 4 occur very close in time. Small fluctuations may change the order in which the arbiter actually receives and processes these requests. This may affect the state of the system at the behavioral level, 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 behavioral level*.

In conclusion, obtaining stable state dumps is problematic in GALS SOCs. It is not possible to sample an entire *consistent* SOC state, or stop the entire SOC at *a single point in time*. Furthermore, each time the SOC has to be (re)run to create a state dump, it is not possible to guarantee exactly the same clock frequencies and phase relations between the clocks, causing the internal synchronization and arbitration processes to take less or more time to complete. SOC state dumps that are extracted on a particular clock cycle may therefore vary significantly between runs, without that difference being indicative of an error.

3. Consistent SOC states

We address the problems described in the previous section by raising the abstraction level of the SOC execution above the clock cycle level. To achieve this, we reuse the functional mechanisms that ensure correct data communication between asynchronous IP blocks. For this, we also leverage work done by Chandy and Lamport [17] and Miller and Choi [18] in the domain of distributed systems, in particular on snapshot algorithms for distributed computations. Subsection 3.1 summarizes their basic distributed system model and algorithms to obtain *consistent global states* by using in-band communication between processes. Subsection 3.2 presents our adaptation of their theory to post-silicon, interactive debugging using the scan chains. Using additional design for debug (DfD) hardware, we can obtain consistent global states for asynchronous, multiple-clock SOCs.

3.1. Distributed snapshot algorithms

A distributed system can be modeled as a set of asynchronous processes, $p_1, p_2, ..., p_i, ..., p_n$ [17], [18], in which processes communicate with each other by sending and/or receiving messages via channels. We denote the communication channel from process p_i to process p_j by C_{ij}. Channels are assumed to be error-free, and to deliver the messages in the order sent. The delay across a channel is arbitrary but finite. The notation m_{ij} denotes a message from process p_i to process p_j.

A process is defined as a set of states, an initial state, and a set of events. Each event e_i^x in process p_i is an atomic action, which changes the state of process

p_i and possibly the state of an output channel C_{ij} or input channel C_{ki}. The *channel state* is the sequence of messages that were sent on it, but excluding those messages that were received from it. The state of channel C_{ij} can be changed by process p_i by sending a message m_{ij} along C_{ij}. The state of channel C_{ki} may be changed by process p_i by receiving a message m_{ki} along C_{ki}.

The *global state* of a distributed system is defined as the collection of states of all processes and communication channels. Because the system is distributed and it is not possible to instantaneously capture the states of all processes and channels, two conditions are specified for a *consistent global state* [19]: (1) Every message m_{ij} that is recorded in the state of process p_i as sent, must either be recorded in the state of channel C_{ij}, or in the state of process p_j, and (2) every message m_{ij} that is recorded as received in a process p_j, must be recorded as sent in the state of process p_i. Condition (1) ensures that no messages are lost in the record of the global state, and Condition (2) ensures that no effect is recorded without also its cause being recorded. The distributed snapshot algorithms of [17], [18] yield a consistent global state, despite the fact that the states are not recorded at the same physical time.

Both algorithms allow a process to either initiate a snapshot recording, or respond to a snapshot recording marker sent by another process. When a process initiates a snapshot recording, it first records its own state and subsequently sends out a marker on all its out-bound communication channels. As long as it does not receive a marker on each of its in-bound communication channels, it records any incoming messages as being part of the state of the corresponding channel. When a process receives a snapshot recording marker on one of its in-bound communication channels, and it hasn't already records its own state, it will do so. A key difference between the two algorithms is that in the algorithm by Miller and Choi, each process also immediately stops its execution after it has recorded its own state and has output snapshot recording markers. It does continue to record incoming messages as being part of the state of the corresponding channel though.

3.2. Our CSAR approach

We first describe our CSAR approach [20], followed by the required on-chip hardware support in Section 4.

Our CSAR approach resembles the algorithm by Miller and Choi the most. The mapping of the distributed system model to a GALS SOC is fairly straightforward, as each IP block can be modeled as an asynchronous process. The communication channels between the IP blocks are mapped to the the on-chip communication architecture, e.g. a multi-layer bus, or a network on chip (NOC). The non-zero delay and error-free channel characteristics apply to most communication architectures, and also the in-order delivery of data often applies with respect to a single IP block. We however do differ from the algorithms described above in two respects.

Firstly, we introduce a protocol-specific instrument (PSI) between the processes and their channels to gate the handshake control signals after reaching a breakpoint, instead of recording the process state, or stopping the execution of processes immediately. By gating the control signals, we effectively stop all communication between IPs. When single-threaded processes subsequently need to communicate with each other, they will functionally stall on the gated handshake control signals. The state of these processes will not change as long as they are stalled. During this stall interval, it is safe to gate the clocks to each individual process, as regardless of how long it takes to synchronize the clock stop request signal in the local clock domains, this clock-level non-determinism no longer effects the process state that is extracted later on via the scan chains.

Secondly, we use a simple, high-speed, and low-cost event distribution interconnect (EDI) to distribute the analogy of the markers to (the PSIs between) the other IP blocks in the system. The EDI has the exact same topology as the on-chip communication architecture, but is faster, as it requires only one clock cycle of the communication architecture for each communication hop. An event therefore reaches (the PSIs between) the other IP blocks connected to the communication architecture, ahead of any messages that were sent after it. It can then trigger any local PSI to prevent those messages from being passed to the target IP block.

For each handshaked data element (i.e. message), one of the PSIs controls its admission into a channel and another PSI its reception from that channel. Combined, these PSIs therefore unambiguously position a particular message either in a process state or a channel state, and prevent the inconsistent recording of the data element as being part of both the sender (receiver) and the channel state. As data elements are never lost in our SOC, and there is no ambiguity about their location, our approach satisfies the two conditions specified in [19] and therefore yields a consistent global state.

4. On-chip DfD hardware

We use the DfD architecture described in [21] that was subsequently extended for communication-centric

debug in [22] to extract a consistent global state from the manufacturing test scan chains inside the chip. Below we first briefly describe the overall architecture and then detail the implementation of the key components, the PSIs and the EDI.

4.1. Architecture overview

Figure 5 shows a simple SOC that contains the debug architecture required to capture consistent global states. Although this example is fairly simple, our approach easily scales to any number of IP blocks.

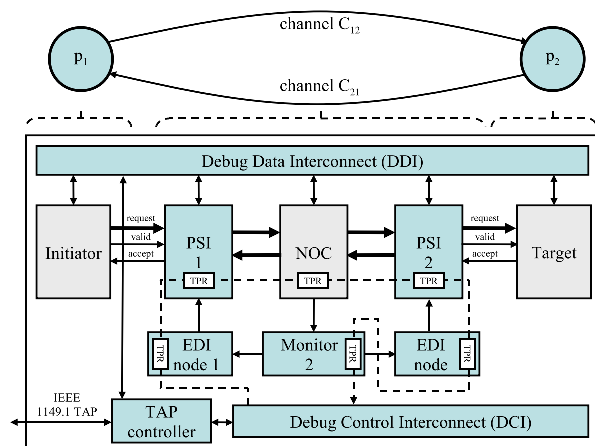

Figure 5. Debug hardware architecture.

An initiator and a target IP block are connected via PSIs to the on-chip interconnect (in this case a NOC). As the name suggests, the implementation of these PSIs is specific to the communication protocol used for this connection. The PSIs receive events from the EDI nodes, which originate from one or more monitors in the SOC. These monitors are programmed with the breakpoint condition. All debug components can be programmed and queried via the on-chip IEEE Std 1149.1-2001 test access port (TAP) controller [23] and the debug control interconnect (DCI). Once the PSIs have stopped all communication handshakes inside the SOC, the local clocks can be gated, and the SOC switched to debug scan mode. In this mode, all scan chains can be accessed via the debug data interconnect (DDI) to extract a state dump of the entire system. As detailed above, the state we extract from the scan chains is globally consistent. Below, we detail the specific implementations of the PSIs and EDI.

4.2. Protocol-specific instrument (PSI)

A PSI intervenes in the communication between two asynchronous IP blocks. It receives the handshake signals from both IP blocks and under normal circumstances passes these signals transparently on to the other IP block. However, the PSI gates all outgoing handshake control signals upon receiving a stop request. Effectively an initiator no longer sees the target accept the data, and the target no longer sees the initiator offering data. (Note that the initiator or target clock cycle at which the handshake is disabled may vary, due to intrinsic synchronization non-determinism.) If the initiator or the target requires the data to make progress, they will stall until this data becomes available. In this state, when the state of the processes no longer changes, we can safely gate the functional clocks, use the DDI and scan chains to create a state dump via the TAP, and return the system in the exact same state, without disrupting the functional behavior. As we also control the gating of the handshake control signals in all PSIs, we can then selectively re-enable communication in all or a subset of the channels.

Figure 6 shows an example PSI implementation, corresponding to the handshake protocol depicted in Figure 2, with an example waveform where an initiator attempts to communicate after a stop event. As described, this attempt fail because the PSI blocks all communication after a stop event.

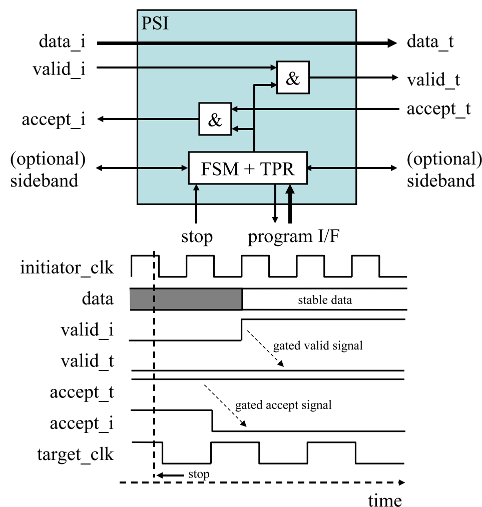

Figure 6. Example PSI implementation.

As shown in Figure 6, a PSI implementation only requires a minimum number of additional gates on each boundary between IP blocks. Often the size of a PSI implementation is dominated by its configuration register, i.e. its test point register (TPR), which determines when and how communication is stopped or resumed. This is particularly the case for the more

complex communication protocols, such as the AXI, OCP and DTL protocols.

4.3. Event distribution interconnect (EDI)

As described in Section 3.2, the EDI is used as a low-cost, simple, and high-speed infrastructure for events, e.g. to initiate stopping of all communication. Debug events may be generated by a variety of monitors [24]–[27]. Although the EDI has the exact same topology as the on-chip communication architecture, events are not sent as in-band or side-band, as opposed to in-band [18], [28]. Communication on the EDI is completely decoupled from the communication on the main communication architecture. Hence it is faster than the main communication architecture, and also faster than in [17], [18]. Messages can therefore be captured quicker, for a more precise breakpoint. Figure 7 shows the implementation of an EDI node and an EDI clock domain crossing (CDC).

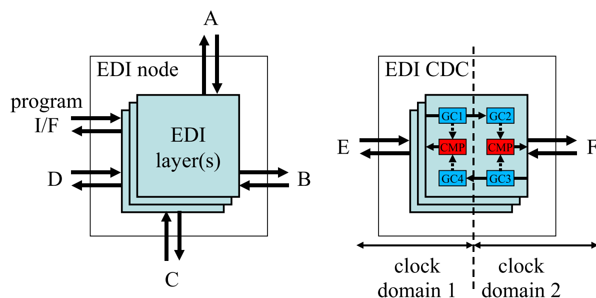

Figure 7. EDI node and clock domain crossing.

An EDI node is parameterized in its number of layers and I/O ports. The latter corresponds to the number of neighboring EDI nodes, PSIs and monitors. Each layer can independently process an incoming event. The silicon cost of an EDI layer, and therefore also an EDI node, is dependent on the network topology. The total cost of the EDI is linear in the number of layers.

Each EDI node can be programmed with an output mask using its TPR. This mask determines to which outputs an incoming event is propagated, irrespective from the input the event came from. The EDI distributes events faster than the functional communication interconnect propagates messages. This permits the EDI to reduce the time to stop all on-chip communication as much as possible. However, stopping can never be instantaneous due to the communication delays involved and the asynchronous clocks.

An EDI CDC permits safe communication of events across an asynchronous clock domain boundary. Incoming events from one clock domain are accumulated using a gray counter, the value of which is tracked to generate an equal number of events in the other clock domain. The EDI CDC never loses events, even though the amount of time between events can be changed.

5. Experimental results

Figure 8 shows a simple, multiple-clock circuit to illustrate our approach.

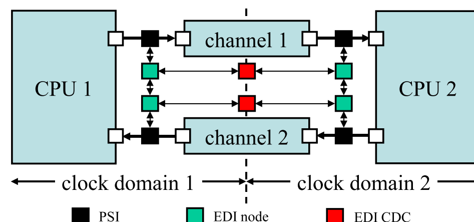

Figure 8. Example multiple-clock circuit.

Two central processing units (CPUs), located in separate, asynchronous clock domains, communicate via two channels. After reset, CPU 1 starts by writing eight data words into Channel 1. Both processors then repeatedly execute, in parallel, the same basic loop, in which CPU 1 (CPU 2) reads four words from input Channel 2 (Channel 1), modifies these words, and writes them into output Channel 1 (Channel 2). The circuit has been extended with our PSI and EDI components to permit state sampling after stopping the communication handshakes between the IP blocks. We conducted experiments with two clock periods, $p_1 = 2,000,003$ fs and $p_2 = 3,000,016$ fs, for each of the two clock signals. These clock periods were chosen because they are prime, and yield frequencies below 500 MHz. In four experiments, we used each of the four possible combinations of clock periods, (p_1,p_1), (p_1,p_2), (p_2,p_1), (p_2,p_2), and created state dumps at each clock cycle of the clock of CPU 1, respectively the clock of CPU 2. Figure 9 shows the results when calculating the stability of state bits across multiple state dumps, taken at the same absolute clock cycle count, but for the different combinations of clock periods. Note how on average around 10% of the state is unstable when sampled on an absolute clock cycle count, irrespective of whether the clock of CPU 1 or CPU 2 is used.

When we repeat the exact same experiment, but instead halt the execution of the circuit using PSIs on the communication handshakes between CPU 1 and Channel 1, respectively CPU 2 and Channel 2, the total number of unstable bits reduces to 0%. Rather than showing a flat line at 0%, Figure 10 illustrates

this observed stability for the program counters (PCs) and two processor registers (r[2], r[3]) of both CPUs, for clock period combinations (p_1,p_2) and (p_2,p_1). Although the circuit stops at a different *absolute cycle count* in time, it stops at the same *handshake count* in time (handshake #85 in this case). This results in stable and consistent register values for all registers in our example circuit, regardless of the specific clock period combination used.

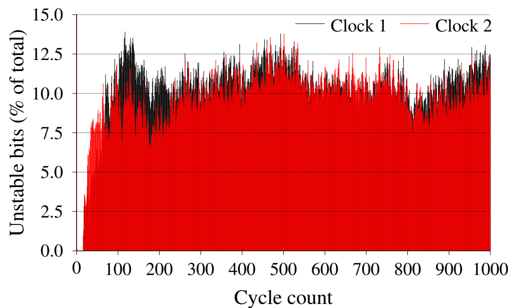

Figure 9. Unstable state bits at cycle level.

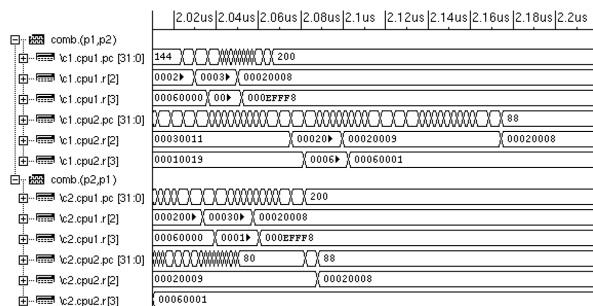

Figure 10. Example of state stability at handshake level.

6. Conclusion

In this paper we introduced a method and architecture to retrieve consistent global states from GALS SOCs. In essence, we abstract from absolute time by raising the moment of state sampling to the functional level of handshakes of IP communication protocols, such as AXI and OCP. The EDI distributes events (or state recording markers) that safely [but potentially at a non-deterministic time] stop the 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. To evaluate our CSAR approach further, we plan to apply this method to a multiple-clock SOC running more realistic user applications.

References

[1] J. Muttersbach et al., "Practical Design of Globally-Asynchronous Locally-Synchronous Systems," in *ASYNC*, 2000.

[2] D. Messerschmitt, "Synchronization in digital system design," *J. on Sel. Areas in Comm.*, vol. 8, no. 8, oct 1990.

[3] B. Roberts, "The verities of verification," *Electronic Business*, Jan. 2003.

[4] S. K. Goel et al., "Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips," in *ITC*, 2002.

[5] P. Dahlgren et al., "Latch Divergency in Microprocessor Failure Analysis," in *ITC*, 2003.

[6] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *DAC*, 2006.

[7] S. I. Association, "The International Technology Roadmap for Semiconductors," 2008.

[8] K. Holdbrook et al. "microSPARC: A Case Study of Scan-Based Debug." in *ITC*, 1994

[9] G. Rootselaar et al., "Silicon Debug: Scan Chains Alone Are Not Enough," in *ITC*, 1999.

[10] D. Josephson et al., "Debug Methodology for the McKinley Processor," in *ITC*, 2004.

[11] Y.-C. Hsu et al., "Visibility enhancement for silicon debug," in *DAC*, 2006.

[12] H. F. Ko et al., "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *DATE*, 2008.

[13] *AMBA AXI Protocol Specification*, ARM, Jun. 2003.

[14] OCP International Partnership, "Open Core Protocol Specification," 2001.

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

[16] D. J. Kinniment, *Synchronization and Arbitration in Digital Systems*. Wiley Publishing, 2008.

[17] K. M. Chandy et al., "Distributed snapshots: determining global states of distributed systems," *ACM Trans. on Comp. Sys.*, , vol. 3, no. 1, 1985.

[18] B. Miller et al., "Breakpoints and halting in distributed programs," in *ICDCS*, 1988.

[19] A. D. Kshemkalyani et al., *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[20] B. Vermeulen et al., "Debugging multi-core systems on chip," in *Multi-Core Embedded Systems*, G. Kornaros, Ed. CRC Press, Sep. 2010, ch. 5.

[21] B. Vermeulen et al., "Core-based Scan Architecture for Silicon Debug," in *ITC*, 2002.

[22] B. Vermeulen et al., "Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip," in *NOCS*, 2008.

[23] IEEE, *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society, 2001.

[24] ARM, "Embedded Trace Buffer," ARM Ltd., Tech. Rep., http://www.arm.com/.

[25] R. Leatherman et al., "An embedding debugging architecture for SOCs," *Potentials, IEEE*, vol. 24, no. 1, 2005.

[26] C. Ciordaş, et al., "A Monitoring-aware Network-On-Chip Design Flow," *J. of Sys. Arch.*, vol. 54, 2008.

[27] B. Vermeulen et al., "A Network-on-Chip Monitoring Infrastructure for Communication-centric Debug of Embedded Multi-Processor SoCs," in *VLSI-DAT*, 2009.

[28] S. Tang et al., "In-band cross-trigger event transmission for transaction-based debug," in *DATE*, 2008.