5

Debugging Multi-Core Systems-on-Chip

Bart Vermeulen

Distributed System Architectures Group
Advanced Applications Lab / Central R&D
NXP Semiconductors
Eindhoven, The Netherlands
bart.vermeulen@nxp.com

Kees Goossens

Electronic Systems Group
Electrical Engineering Faculty
Eindhoven University of Technology
Eindhoven, The Netherlands
k.g.w.goossens@tue.nl

CONTENTS

5.1 Introduction

Over the past decades the number of transistors that can be integrated on a single silicon die has continued to grow according to Moore's law [5]. Higher customer expectations, with respect to the functionality that is offered by a single mobile or home appliance, have led to an exponential increase in system complexity. However, the expected life cycle of these appliances has decreased significantly as well. These trends put pressure on design teams to reduce the time from first concept to market release for these products, the so-called *time-to-market*.

To quickly design a complex system on chip (SoC), design teams have therefore adopted intellectual property block *re-use methods*. Based on customer requirements, pre-designed and pre-verified intellectual property (IP) blocks, or a closely-related set of IP blocks (e.g., a central processing unit (CPU) with its L1 cache), are integrated on a single silicon die according to an application domain-specific *platform template* [15]. Not having to design

these IP blocks from scratch and leveraging a platform template significantly reduces the amount of time required to design an system on chip (SoC), and thereby its time-to-market.

Furthermore, during the design of an SoC a structural, temporal, behavioral and data *refinement* process is used to effectively tackle its complexity and efficiently explore its design space within the consumer and technology constraints. During this process, details are iteratively added to a design implementation until it is ready for fabrication. This process is illustrated in Figure 5.1, which is adapted from [38].

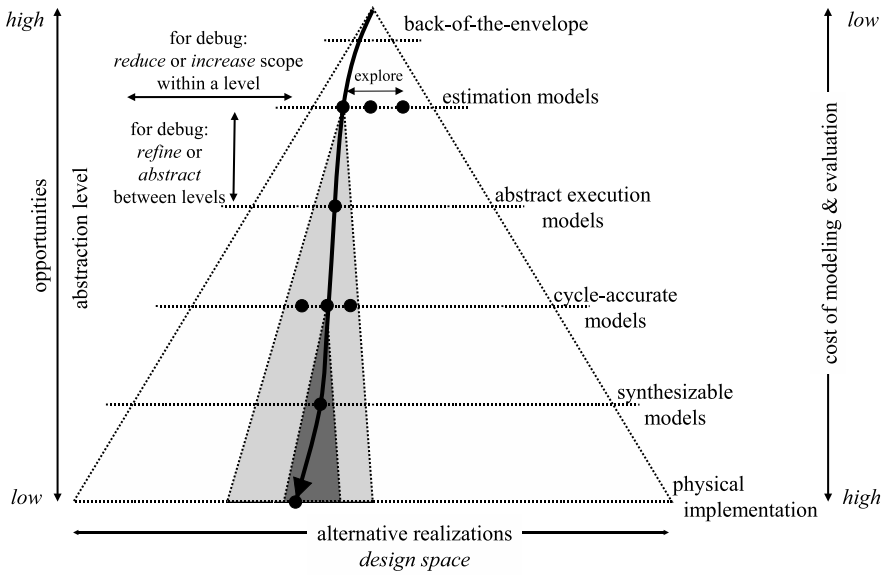


FIGURE 5.1: Design refinement process. (Adapted from A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools.* Ph.D. thesis, Delft University of Technology, 1999.)

The correctness of each refinement step, from one level of design abstraction to a lower level, has to be verified. Techniques such as formal verification, simulation, and emulation provide confidence that no errors were introduced and the resulting design should behave according to its original specification.

The ability to exhaustively verify a design before it is manufactured is severely restricted by the aforementioned increased system complexity. To both timely prepare a design and have sufficient confidence for its release to the market, verification engineers have to make trade-offs between the levels of design abstraction and the number of use cases to verify at each level. Functional problems may go undetected as it is impossible to cover all use cases at the level of the physical implementation before manufacturing. Problems may only manifest themselves after manufacturing test of an SoC, 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 discovered during the initial functional validation of the silicon 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 [55] show that this validation and debug process consumes over 50 percent of the total project time while the number of designs that are right first time is less than 40 percent.

The focus of this chapter is to describe the *debugging of a silicon implementation of an SoC*, which does not behave as specified in its product environment. During debugging, we need to find the root cause that explains the difference in the implementation's behavior from its specified behavior during a system run. We use the term "run" to mean a single execution of the system. For this we propose to use an iterative refinement and reduction process to zoom in on the location where and the point in time when an error in the system first manifests itself. This debug process requires both observation and control of the system in the environment where it fails.

The remainder of this chapter is organized as follows. Section 5.2 first provides a more in-depth analysis of the fundamental problems that need to be solved to debug an SoC. In particular, it is not easy to observe and control the system to be debugged. Section 5.3 describes how these fundamental problems affect the ideal debug process, and it subsequently defines the debug process used in practice. Section 5.4 presents an overview and comparison of existing debug methods. We introduce our debug method in Section 5.5. Section 5.6 defines the on-chip infrastructure to support our debug method, followed by the off-chip debug infrastructure in Section 5.7. We apply our method on a small example in Section 5.8, and conclude with Section 5.9.

5.2 Why Debugging Is Difficult

In this section, we identify three problems that make debugging intrinsically difficult: (1) limited internal observability, (2) asynchronicity, and (3) nondeterminism.

5.2.1 Limited Internal Observability

One of the biggest problems while debugging a system is the volume of data that potentially needs to be examined to find the root cause. Worst case: this volume is equal to the amount of time from start-up of the system to the first manifestation of incorrect behavior on the device pins multiplied by the product of the number of electrical signals inside the chip and their operating frequencies. This data volume is huge for multimillion transistor designs run-

ning at hundreds of megahertz. Consider for example a 10 million transistor design running at 100 megahertz. If we sample one signal per transistor per clock, then this design produces 10^{15} bits of data per second.

The exponential increase in the number of transistors on a single chip [5] 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 during its execution. If the same design has 1,000 pins, then even if we could use all these pins to output the data this design produces per second, we would have to operate these pins at speeds of 10^{12} bits per second per pin to output all data, which is clearly beyond current technological capabilities. Typically the number of device pins available for observation is much less as the chip still has to function in its environment and a large number of pins are reserved for power and ground signals.

5.2.2 Asynchronicity and Consistent Global States

In the remainder of this chapter we assume that each IP block in the system operates on a single clock, i.e., is synchronous. However, the clocks of different IP blocks can be *multi-synchronous* or *asynchronous* with respect to each other.

Multi-synchronous clocks are derived from a single base clock by using frequency multipliers and dividers or clock phase shifters. Data transfers between IP blocks take place on common clock edges, where explicit knowledge of the clock frequencies and phase relations of the IP blocks is used to correctly transfer data. Source-synchronous communication that tolerates limited clock jitter also falls in this category.

In contrast, *asynchronous* clocks have no fixed phase or frequency relation. Many embedded systems today use the globally-asynchronous locally-synchronous (GALS) [47] design style. As a consequence, all modern on-chip communication protocols use a so-called valid-accept handshake to safely transfer data between IP blocks, e.g., in the Advanced eXtensible Interface (AXI) [4] protocol, the Open Core protocol (OCP) [50] and the device transaction level (DTL) [54] protocol. As illustrated in Figure 5.2, the initiator prepares the "data" signals and activates its "valid" signal, thereby indicating to the target that the data can be safely sampled. The target samples the data using its own clock and signals the completion of this operation to the initiator by activating its "accept" signal. This handshake sequence ensures that the data are correctly communicated from the initiator to the target, irrespective of their functional clock frequencies and phase. The handshake sequence is part of the communication function of the IP block, and is usually implemented with stall states in an internal finite state machine (FSM). For ease of explanation, we assume the initiator and target stall while transferring data.

Debug requires the sampling of the system state for subsequent analysis. The state of an individual IP block can be safely sampled because it is in

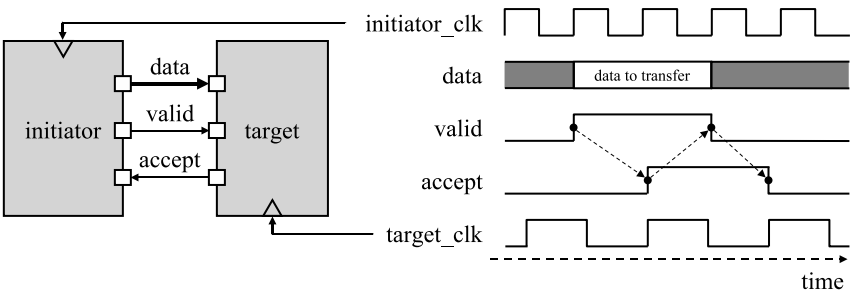


FIGURE 5.2: Safe asynchronous communication using a handshake.

a single clock domain, and an external observer simply has to use the same clock as used by the IP block. Sampling requires synchronicity to the clock of the IP block to prevent capturing a signal while it is making a transition. Proper digital design requires that IP signals are stable around the functional clock edges for an interval defined by the setup and hold times of the flip-flops used. The active edges of the functional clock therefore make good sampling points for external observation.

However, for debugging a system, we may need to inspect the *global state*, i.e., the combined local states of all IP blocks in the system. For multiple IP blocks, their safe sampling points are determined by the greatest common divisor of their frequencies. Only at these points, a *consistent global state* can be sampled, as the state of each IP block can be safely sampled at these points and the combination of all IP states also reflect the global state at these points. At all other points in times, it is not guaranteed safe to sample the state of all IP blocks. One or more local states are therefore unknown at those points preventing debug analysis. With two multi-synchronous clock domains, sampling on the slower clock may lead to missing some possible state transitions in the IP block with the faster clock. Conversely, sampling the state of the IP block running on the slower clock, with the faster clock is unsafe as we may sample in the middle of a state transition.

If two IP blocks are *asynchronous* with respect to each other, then there is no guarantee that their safe sampling points will ever coincide, and no points in time at which the global state can be consistently sampled may exist.

Consider as an example two IP blocks A and B. Block A has a clock period T_A of 2 ns, block B has a clock period T_B of 3 ns. We define the clock phase ϕ_{A-B} between these two clocks as the time between the rising edge of clock A and the rising edge of clock B. If $\phi_{A-B} = 0.5$ ns at a certain point in time $t = t_0$, then there is no point in time where the rising edges of clocks A and B coincide. For this, Equation 5.1 must hold for integer values of m and n. However the left-hand side of Equation 5.3 is always even for integer values of m, while the right-hand side of Equation 5.3 is always odd for integer values of n. Therefore there are no points in time where the rising clock edges for

clocks A and B coincide.

$$T_A \times m = \phi_{A-B} + T_B \times n \tag{5.1}$$
$$2 \times m = 0.5 + 3 \times n \tag{5.2}$$
$$4 \times m = 1 + 6 \times n \tag{5.3}$$

This is also illustrated in Figure 5.3.

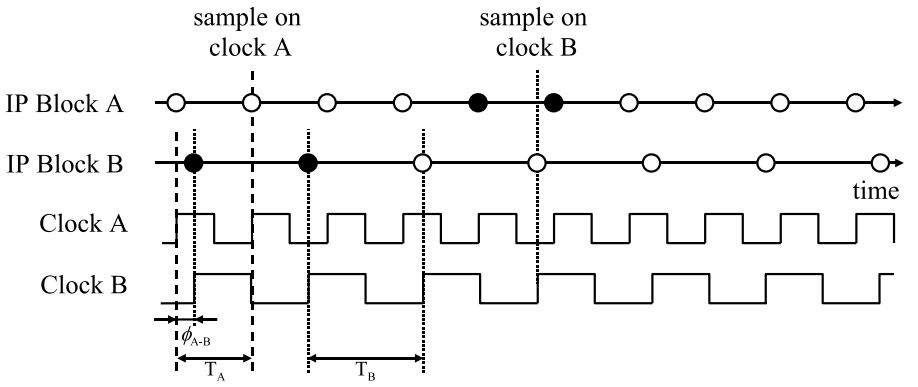


FIGURE 5.3: Lack of consistent global state with multiple, asynchronous clocks.

In general for a GALS system, it may therefore not be possible to correctly sample a globally consistent state at all (or even any) points in time at the clock cycle level. The only points at which the state of multiple IP blocks can potentially be safely captured is during synchronization operations, in which the state of both IP blocks has to be functionally defined and therefore has to be stable. It may therefore be possible to capture a consistent global state at these functional synchronization points. Synchronisation may however take place at different levels of abstraction, and require behavioral knowledge of the design to implement. Examples of using behavioral information to improve the ability to capture a globally consistent state will be introduced in Section 5.5.

5.2.3 Non-Determinism and Multiple Traces

Clock-domain crossings not only complicate the definition of a globally consistent state, but also cause variation in the exact duration of the communication between clock domains. When the initiator and target clocks have different (or even variable) frequencies or phases, then a valid-accept handshake can take a variable number of initiator and/or target clock cycles due to metastability [53, 64] (see A in Figure 5.4).

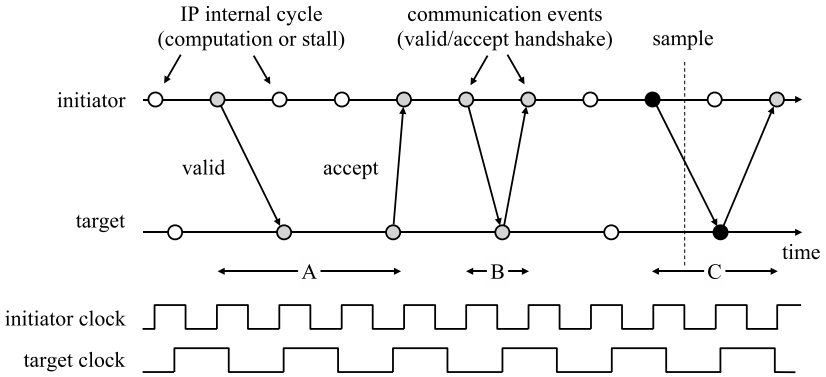
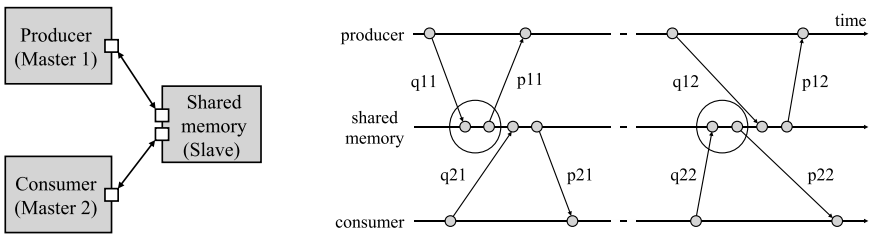


FIGURE 5.4: Non-determinism in communication between clock domains.

Essentially, in a GALS system it is not possible to safely sample a signal from another clock domain using a constant number of local clock cycles, due to metastability [65]. Although statistically it is very likely that the sampled signal is stable quickly, e.g., after one target clock cycle, it is possible that it takes (much) longer. This is illustrated in Figure 5.4 with the two handshakes, labeled B and C, respectively. B takes one initiator clock cycle, and C two cycles, even though in both cases the target responds within a single target clock cycle. This behavior occurs between asynchronous IP blocks in an SoC, but also for communication on the chip pins, for data transfers to and from the chip environment.

Critically, this local (inter-IP) non-determinism in communication behavior propagates to the system level, where it manifests itself in multiple communication traces [31, 60]. With the term "trace" we refer to a unique sequence of observed system states during a run. Figures 5.5a and 5.5b illustrate this phenomenon.



(a) System under Debug.

(b) Transaction ordering and multiple traces.

FIGURE 5.5: Example of system communication via shared memory.

As an example, Figure 5.5a shows two masters, called Producer and Consumer, communicating directly with a shared memory on different ports using

transactions, each transaction comprising a request and an optional response message. Examples of transaction requests include read commands with read addresses, and write commands with write addresses and data. Corresponding responses are read data and write acknowledgments, respectively. All modern on-chip communication protocols fit this model [19].

The shared memory in our example only has one execution thread, and therefore can only accept and execute a single request at a time. We will further assume for illustration purposes that a read by the Consumer is only correct if the Producer writes to the shared memory before the Consumer reads from it. Figure 5.5b shows Master 1 initiates a write request "q11," soon followed by a read request from Master 2 "q21." Master 1's request is executed first by the slave, resulting in a response "p11." Afterwards the request of Master 2 is executed by the Slave, resulting in a correct response "p21." Another sequence with a different, incorrect outcome is however also possible and is shown with the subsequent requests ("q12" and "q22"). This time, due to a different non-deterministic delay on the communication path between the masters and the slave, write request "q21" from the Producer is executed after read request "q22" from the Consumer. This response "p22" returned to the Consumer will be incorrect because the Consumer read the response before the Producer could write it.

Executing transactions in different orders can have an impact on the functional behavior of the IP blocks. For example, consider that Master 1 produces data in a first-in first-out (FIFO) data structure for Master 2, and signals that new data is ready by updating a FIFO counter or semaphore in the shared memory [49]. If Master 2 reads the counter from memory using polling, then both sequences are functionally correct. However, in the scenario shown on the right-hand side of Figure 5.5b Master 2 reads the old counter value, and it would require another polling read to observe the new counter value, resulting in a delayed data transfer. Whether this is a problem or not depends on the required data rates. It would definitely be erroneous, however, if the requests of the masters were write operations with different data to the same address. In this case, the functional behavior of the system would be non-deterministic, and possibly incorrect, from this point onward.

5.3 Debugging an SoC

In this section we define errors and explain how the analysis in Section 5.2 of what makes debugging intrinsically difficult affects the ideal debug process. We subsequently describe the debug process that has to be used in practice.

5.3.1 Errors

We assume that the observed global states are consistent in some sense, which is justified in Section 5.5. As shown in Subsection 5.2.3 multiple runs result in the same or different traces due to non-determinism. An error is said to have occurred when a state in a trace is considered incorrect with respect to either the specification or an (executable) reference model. Such a state is called an "erroneous state." Note that we consider errors, i.e., the manifestations of faults, and we consider the objective of debugging to be to find and remove the root cause of these errors (i.e., the faults causing them). Fault classifications and discussions on the relation between faults and errors can be found in [6, 9, 39, 44].

Error observations can be classified in three orthogonal ways: within a trace, between traces, and between systems.

- *Within a trace.* When all states following an erroneous state are erroneous states as well, the error is *permanent*, otherwise the error is *transient*. Transient errors may happen, for example, when erroneous data is overwritten by correct data, before it propagates to other parts of the system.

- *Between traces.* An error is *constant* when it occurs in every run (and hence in every trace). This is always the case when the system is deterministic as deterministic systems have only a single trace. An error is *intermittent* when it occurs in some but not all runs. For a system to exhibit intermittent errors, it has to be non-deterministic, as discussed in Section 5.2.3. It therefore produces different traces over multiple runs.

- *Between systems.* Finally, until now we assumed that the system does not change between runs. This is not necessarily the case. The debug observation or control of the system is often *intrusive*, i.e., it changes the behavior of the system. This phenomenon is also known as the "probe effect." As a result, often the error disappears and/or other errors appear when monitoring or controlling the system. In these cases, we basically generate traces for two different systems, so the resulting traces may be very different and hard to correlate. We call these *uncertain* errors, after the uncertainty principle[1], as opposed to *certain* errors.

For simplicity, we will assume in the remainder of this chapter that all errors are permanent and certain, though they may be intermittent. We use a small example to see how these differences in error types can manifest themselves during debugging of an embedded system.

[1] Gray [25] introduced "Bohrbugs" and "Heisenbugs." However, these terms are not used consistently in the literature, and we will therefore not use them.

5.3.2 Example Erroneous System

We illustrate constant, certain and intermittent errors by re-using the simple example system of Figure 5.5a and focus on the states of the individual IP blocks. The possible system traces are illustrated in Figure 5.6.

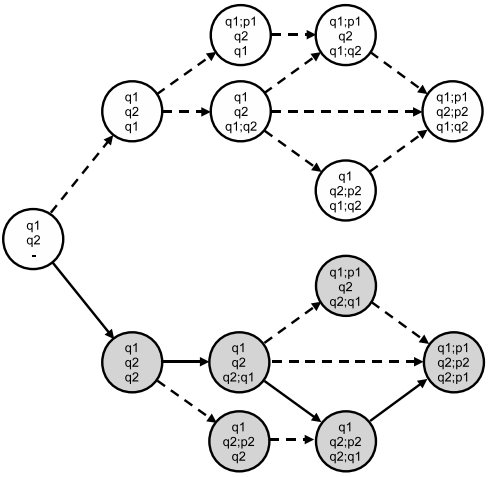


FIGURE 5.6: System traces and permanent intermittent errors.

Each circle corresponds to a consistent global state. The text inside the label indicates from top to bottom the state of Master 1, Master 2, and the Slave respectively. A shaded state indicates that the error has propagated into the global system state. Figure 5.6 also shows the largest scope, i.e., when the consistent global state comprises the local states of both Master 1 and Master 2 and the Slave. "qi" refers to the sending or receiving of a request of Master i, and "pi" for the corresponding response.

We can now illustrate how intermittent errors occur using Figure 5.6. A run proceeds along a certain trace, such as the one that is highlighted by the solid line. In the first state "(q1 q2 -)" both masters generate their request to the slave memory at the same time. As a result of non-deterministic communication between the masters and the shared slave, our example system can have multiple execution traces. Figure 5.5b illustrates this by focusing on the interleaving of transactions. In Figure 5.6 we concentrate on the divergence of the global states and resulting multiple traces instead. As shown in Figure 5.5b, the memory may accept and execute the request of Master 2 first (with global state "(q1 q2 q2)"), and offer an *erroneous* response "p2." Before this response is accepted by Master 2, the memory accepts and executes request "q1," causing the global state "(q1 q2 q2; q1)." Master 2 then accepts "p2" (with global state "(q1 q2; p2 q2; q1)"), followed by Master 1's acceptance of response "p1." The global end state is where both masters have received the response to their request ("(q1; p1 q2; p2 q2; q1)").

In an alternative trace the slave executes request "q1" before request "q2." Master 1 subsequently receives a correct response "p1," followed by a correct response "p2" for master 2. The global end state for this trace is "(q1; p1 q2; p2 q1; q2)," which differs from the end state of the previous trace by the order in which the slave handled the incoming requests ("q1" before or after "q2").

Hence, when executing the system a number of times it can generate different traces. Even with non-intrusive observation (i.e., with certain errors), the error may only be triggered and consequently visible in a subset of the traces and is therefore intermittent. Moreover, the error, i.e. the return of the incorrect response "p2," can become visible at Master 2 at different points in time in the different traces. This makes intermittent errors particularly hard to find [16, 25].

5.3.3 Debug Process

The process of debugging relies on the observation of the system, i.e., its states, for a certain duration of time, and at discrete points in time. This observation results in a state trace. The state can be observed at various levels of *abstraction*, which determines in how much detail we look at the system. We can consider for instance only which applications are running, which transactions are active, which signal transitions occur, or what the voltage levels are on the physical wires.

At a given level of abstraction, the *scope* of the observation determines how much of the system we observe and for how long we observe it. This scope may be varied between runs. For example, Figures 5.7, 5.8, and 5.6 illustrate observations with increasing (spatial) scope.

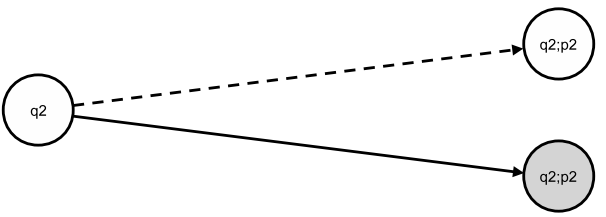


FIGURE 5.7: Scope reduced to include Master 2 only.

Figure 5.7 includes only Master 2 in the scope. We see two distinct end states as the order in which the requests from Master 1 and Master 2 are executed by the slave can still cause the response for Master 2 ("p2") to be different between runs. Figure 5.8 includes both Master 1 and Master 2 where both the request execution ordering by the slave and the order of acceptance of the responses by the individual masters splits the traces in six different traces. Figure 5.6 provides the most detail by including the state of all master and slave IP blocks.

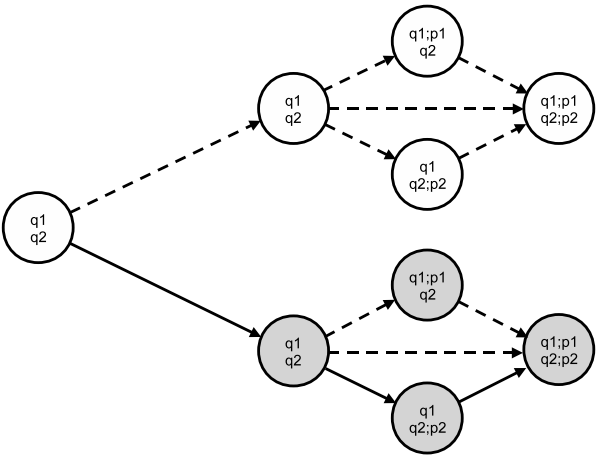


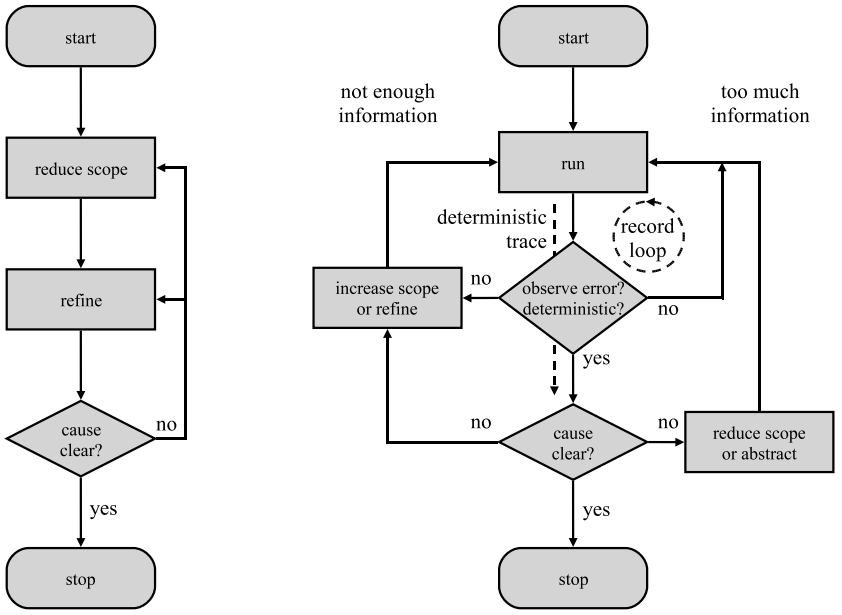
FIGURE 5.8: Scope reduced to include Master 1 and Master 2 only.

The observation and control of the system takes place in the same scope and at the same abstraction level. The debug process essentially involves *iteratively either increasing or decreasing the scope and abstraction level* of observation and control until the root cause of the error is found. In the ideal debug process, we observe only the relevant state to find the root cause for a particular error and for a minimal duration. This process is shown in Figure 5.9a.

First, we reduce the scope, i.e., zoom in on the part of the system where and when the error occurred. Preferably, we "just" walk back in time to when the error first occurred [43, 63], and observe only the state of the relevant IP blocks. Then we refine (lower the level of abstraction) to observe those IP blocks in more detail. For example, we refine the state of an IP block to look at its implementation at register transfer level (RTL) to logic gates or from source code to assembler, or we refine communication events to their individual data handshakes or clock edges. In Figure 5.1 the path from the highest abstraction level down to the physical implementation level can also be interpreted as an instance of the debug process, whereby the reduction of the debug scope takes places within one abstraction level, and the refinement takes place between abstraction levels.

However, in practice, debugging is more challenging due to the lack of internal observability and control, the difficulty involved in reproducing errors, and the problems in deducing their root cause. The effect of these three factors on the debug process is shown in Figure 5.9b.

1. Lack of *observability*. We can inspect given traces, but we need to restart every time we want to observe the trace of a new run. Each trace may take a long time (hours or even days), to trigger the error, resulting in a huge data volume to analyse.



(a) Ideal (b) In practise

FIGURE 5.9: Debug flow charts.

2. Lack of error *reproducibility*. Non-determinism causes multiple traces and intermittent errors, as discussed in Section 5.3.2. Finding the first state that exhibits the error may take a long time because every run of the system proceeds (non-deterministically) along one of many potential traces, with possibly very different probabilities. For example, the highlighted trace in Figure 5.8 may only be taken in 0.001 percent of the runs. Consequently the time between two runs that both exhibit the error may be very long.

3. *Deduction of root cause*. At some point during the debug process we arrive at Figure 5.7, where we have a minimal scope that exhibits the error. To deduce why either a good or bad trace is taken, we need to either increase the scope and observing the state of more IP blocks or refine the state of the IP blocks we are already looking at and observe their state in more detail. We need to intelligently guess that adding the state of the slave to the observed state is a good idea. A larger observed state will however usually result in a larger number of possible traces, as illustrated in Figure 5.6. In subsequent runs, the scope will have to be reduced to the relevant parts again. The decision when to increase the scope and when to refine the state is not trivial. Even without non-determinism, the cause of the error is often not evident when a good to bad state transition occurs, as we see an effect but cannot automatically

deduce the cause. We then either increase the information to investigate by increasing the scope or by refining the state. This is illustrated in Figure 5.6, where the state of the slave is added. In a subsequent run it is then possible to observe that executing "q2" before "q1" is the cause (at this abstraction level) of the error.

With this general debug process in mind, we describe in the following section various existing debug methods that have been proposed in literature.

5.4 Debug Methods

To simplify or automate the debug process, several methods have been proposed in the literature. They all assume that it is possible to find a consistent global state. Observing this global state at certain points in time over multiple runs results in a set of traces. Essentially, the existing debug methods differ in how often they observe what state while the system is running, and whether this is intrusive or not. We first define several properties we use to classify common debug methods.

5.4.1 Properties

We compare different, existing debug methods using three important debug properties: their use of abstraction techniques, their scope, and their intrusiveness.

Choosing the right abstraction level helps reduce the volume of data to observe. This reduces the bandwidth requirements for the observation infrastructure as well as the demands on the human debugger. We consider four basic abstractions [45]: (1) structural, (2) temporal, (3) behavioral, and (4) data.

- *Structural abstraction* determines what part of the system we observe within one abstraction level (e.g., all IP blocks, or only the masters) and at what granularity (e.g., subsystem, single IP block, logic gates, or transistors).

- *Temporal abstraction* determines what and how often we observe. For example, traditional trace methods observe the state at every cycle in an interval, or sample the state periodically. Alternatively, only "interesting" relevant state may be observed at or around relevant communication or synchronization events. Examples include the abstraction from clock cycles to handshakes (illustrated by the removal of internal clock cycles in Figure 5.4), moving to transactions, or to software synchronizations using semaphores and barriers.

- *Behavioral abstraction* determines what logical function is executed by a (hardware) module. For example, in a given use case, a processor may be programmed to perform a discrete cosine transform (DCT), and a network on chip (NoC) may be programmed to implement a number of "virtual wires" or connections. In another use case, they may have different logical functions.[2]

- *Data abstraction* determines how we interpret data. At the lowest level we observe voltage levels in a hardware module. We abstract from this voltages first to the bit level and subsequently use knowledge of the module's logical function at that moment in time to interpret the values of these bits. For example, a hardware module that implements a FIFO contains logical read and write pointers defining the valid data. Only with this knowledge can we display the collection of bits as a FIFO. Similarly, a processor's state can be abstracted to its pipeline registers [37], a memory content, for example, to a DCT block, and registers in a NoC to a connection with FIFOs, credit counters, etc.

Existing debug methods also vary in their *scope*, which was introduced in the previous section. Scope uses structural and temporal abstraction, but considers only one abstraction level.

Increased abstraction (and reduced scope) serve to reduce the volume of data that is observed. The system state can either be observed when the system is running, called *real-time trace*, or when it is stopped, called *run/stop debug*, or both.

During real-time trace debugging, the data is either stored on-chip in buffers, streamed off the chip, or both. This is only possible when the volume of data is not too large and hence may require the use of abstraction techniques. This trace process may be intrusive or not.

During run/stop debugging, the system is stopped for observation, which is by definition intrusive. However, in return, it usually allows access to much more system state because ample time and bandwidth are available for inspection, as the system execution has been stopped.

Every debug process relies on the observation of the system, i.e., accessing its state. *Intrusive* observation affects the behavior of the system under observation, and may lead to uncertain errors. *Non-intrusive* observation does not affect the behavior of the system (aside from consuming some additional power), but does require a dedicated and independent debug infrastructure, making it more expensive to implement on-chip than the infrastructure to support intrusive observation.

[2]This is a different slant on behavioral abstraction from [45], where it is defined as partial specification. In any case, the distinction of behavioral abstraction and temporal and data abstraction is to some extent arbitrary.

5.4.2 Comparing Existing Debug Methods

Without making changes to the design of a chip, a debug engineer has the classic *physical* and *optical* debug methods at his disposal, such as wafer probing [7], time-resolved photo-emission [48] (also known as picosecond imaging circuit analysis (PICA) [34]), laser voltage probing (LVP) [51], emission microscopy (EMMI) [30], and laser assisted device alteration (LADA) [59]. These physical and optical techniques are *non-intrusive*, provided that removing the package and preparing the sample cause no behavioral side effects. They provide observability at the lowest level of abstraction only, i.e., voltage levels on wires between transistors in real time.

Unfortunately these methods can only access the wires that are close to the surface. Access to other, deeply embedded transistors and wires is often blocked by the many metal layers used today to provide the connectivity inside the chip, and to aid in planarization. Back-side probing techniques help somewhat to reduce the problems of the increasing number of metal layers. In nanometer CMOS processes, these methods still suffer from a number of drawbacks. First, the number of transistors and wires to be probed is too large without upfront guidance. Moreover, the transistors and wires may be hard to access because they are very small. Finally, device preparation for each observation is often slow and expensive.

Hence these methods can only efficiently localize root causes of failures if the error is first narrowed down to the physical domain (such as crosstalk, or supply voltage noise). To reach this point, and walk the debug path in Figure 5.1 all the way down to the level of the physical implementation, we need to reduce the scope and lower the level of system abstraction.

Logical debug methods have been introduced for this purpose. Logical debug methods use built-in support called design for debug (DfD) to increase the internal observability and controllability, and act as a precursor to the physical and optical debug methods by helping to quickly reduce the scope containing the first manifestation of the root cause.

These logical debug methods reduce the data volume by making a trade-off between focusing on the real-time behavior of the system and maximizing the amount of state that can be inspected. Only a small subset of the entire internal state can be chosen for observation when the real-time behavior of the system is to be studied due to the aforementioned I/O bandwidth constraints. Whether this is intrusive or not depends on the infrastructure that is used to transport and/or store the data. ARM's CoreSight Trace [2] and FS2's PDTrace [46] architectures are examples of *non-intrusive*, real-time trace. Sample on the Fly [37] is a real-time trace method used for central processing units (CPUs) that periodically copies part of the CPU state in dedicated scan chains that can then be read out non-intrusively. Memory-mapped I/O can be used to read and write addressable state over the functional/inter-IP interconnect while the system is running, for example, with ARM's debug access port (DAP) [2], or FS2's Multi-Core Embedded Debug (MED) system [41].

This will however be more intrusive than a dedicated observation and control architecture.

By stopping the system at an interesting point in time, a much larger volume of data can be inspected. This run/stop-type approach however is *intrusive*. The infrastructure used to access the state and its implementation cost are then the limiting factors. For example, the manufacturing test scan chains provide a low-cost infrastructure, which can be used to read out the entire digital state when the system is stopped [71].

The majority of published, logical debug methods do not address the problems caused by asynchronicity, inconsistency of global states, non-determinism or multiple traces. However, there are several notable exceptions that we discuss next: latch divergence analysis, deterministic (re)play, and the use of abstraction for debug.

5.4.2.1 Latch Divergence Analysis

Latch divergence analysis [13] aims to automatically pinpoint erroneous states. It does so by running a CPU many times, and recording its state at every clock cycle. The traces that are obtained from runs with a correct end result are then compared with each other. The unstable part of each state, called latch divergence noise, is filtered out. This step yields the stable substate across all good traces. Similarly, the stable substate across traces with an incorrect end result is computed. This substate is then compared with the stable substate of the good traces.

The inference is that the unstable parts are caused by noise, e.g., through interaction with an analog block or uninitialized memory, and can be safely filtered out, as they are not caused by the error. An advantage of this method is that it can be easily automated. However, this method does not distinguish noise in substates due to intermittent errors, i.e., those that only occur in some traces, and correct but only partially specified system behavior. Filtering out the noise caused by the partial specification of the behavior may obscure the root cause of an error.

5.4.2.2 Deterministic (Re)play

Instant replay [42], and deterministic replay [18, 56] aim to reduce the time between runs that exhibit an error. When an error is observed, the system is subsequently placed in "record" mode and restarted. The system is repeatedly run until the error is observed again. This step corresponds to the dashed "record loop" in Figure 5.9b. At this point, the debug process can start by replaying the same run and observing the recorded trace as highlighted in Figure 5.7, provided that the recording contains enough information to deterministically replay the trace containing the error. The key idea is that a previously intermittent error appears in every replayed run ("deterministic trace" in Figure 5.9b). Deterministic replay requires all sources of non-determinism to be recorded at the granularity at which they cause divergence in a trace.

It also requires an additional on-chip infrastructure to force the single trace that triggers the error once it has been recorded.

Deterministic replay has been used successfully for software systems, where the non-determinism is limited to the explicit synchronization of threads or processes. The number of divergence points is relatively small, and the frequency of synchronization is low in these cases [42]. However, for embedded systems with multiple asynchronous clock domains, we have seen in Section 5.2.3 that a clock domain crossing between asynchronous clock domains gives rise to non-determinism. Therefore the delay across this interface needs to be recorded. Since an SoC easily contains more than a hundred IP ports connecting asynchronously to an interconnect [22], running at hundreds of megahertz, the data rate to be recorded quickly reaches gigabits per second. It is expensive in silicon areas to non-intrusively record this data on-chip and expensive in device pins to stream it non-intrusively off-chip. However, an intermediate means of communication, namely source-synchronous embedded systems, has been successfully used for a limited number of processors [60].

Pervasive debugging [29] has been proposed with the same goal as deterministic replay. It proposes to model the entire system in sufficient detail such that non-deterministic effects become deterministic. This may be possible for (source)-synchronous systems. However, it is infeasible for systems that contain asynchronous clock domains, or contain errors relating to physical properties (e.g., crosstalk, or supply voltage noise) and environmental effects (ambient temperature, chip I/O, etc.). Relative debugging [1], where an alternative (usually sequential) version of the system is used as a reference to check observed states against, suffers from the same limitations.

Finally, synchro-tokens [31] may be interpreted as deterministic *play*. All synchronizations of a GALS system are made deterministic in every run (and not only during debug), from the view of the communicating parties. Hence, there is a unique global trace (the "deterministic trace" namely the (software) synchronization points, in Figure 5.9b), and all errors are constant. The main drawback of this method is that it reduces performance by essentially statically scheduling the entire system.

5.4.2.3 Use of Abstraction for Debug

System simulations for debug tend to focus on only one or two abstraction levels at a time. For example, traditional software debug allows observation and control (e.g., single-stepping) per function, per line in the source code, and can show the corresponding assembly code. It is difficult to debug multi-threaded or parallel software programs using conventional software debuggers because the parallel nature of programs is not supported well. However, specialized debuggers make the distinction between inter-process communication and intra-process computation. By abstracting to synchronization events [8] they allow the user to focus on less but more relevant information.

Hardware descriptions define parallel hardware, but traditional hardware simulation does not make a distinction between inter-IP communication (e.g., VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) or Verilog signals) and intra-IP computation (e.g., VHDL variables). Traditional hardware simulation is more limited because it simulates either the RTL or the gate-level description, and does not show any relation between them. In recent years, transaction-level modelling and related visualisation techniques have been introduced to abstract away from the signal level IP interfaces and allow a user to focus on the transaction attributes instead [61] or correlate gate level with RTL descriptions [33].

Traditionally, when debugging real hardware that executes software, either functional accesses, real-time trace, or state-dump methods are used to retrieve the system state, as described earlier. Once the state has been collected, it can be interpreted at a higher level, e.g., by re-presenting it at the gate level or RTL level [68]. Recently, DfD hardware has been added to observe and control the system at higher levels of abstraction. Examples include transaction-based debug [24], programmable run-time monitors [11, 73], and observation based on signatures [72].

Overall, we observe that the existing software debug methods are quite mature, especially for sequential software, but less so for parallel software. Existing hardware debug methods are even more limited. Abstraction is currently only applied in a limited fashion, and then almost exclusively for software debug.

5.5 CSAR Debug Approach

In this section we define a debug approach called CSAR and discuss its characteristics. Following this, Sections 5.6 and 5.7 describe how this approach is supported, both on-chip and off-chip. Section 5.8 illustrates how our approach works for a small example.

The CSAR debug method can be characterized as:

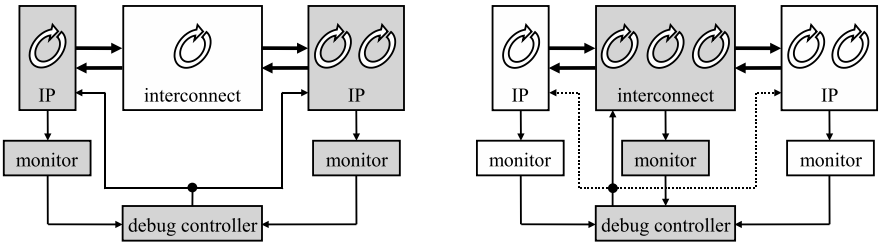
- Centered on *C*ommunication
- Using *S*can chains
- Based on *A*bstraction
- Implementing *R*un/stop control

Each characteristic is described in more detail below.

5.5.1 Communication-Centric Debug

Figure 5.10a illustrates traditional computation-centric debug, in which the computation inside IP blocks, especially embedded processors, is observed. When something of interest happens, this is signaled to the debug controller that can take action, 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, we focus on debugging the communication between IP blocks, as shown in Figure 5.10b.



(a) Computation-centric (b) Communication-centric

FIGURE 5.10: Run/stop debug methods.

Older on-chip interconnects, such as the advanced peripheral bus (APB) and ARM high performance bus (AHB) [3], are single-threaded. This means that only one transaction is processed by the interconnect at any point in time. As a result, the interconnect forces a unique trace for all IP blocks attached to these buses even when using a GALS design style. For scalability and performance reasons, recent interconnects, such as multi-layer AHB and AXI buses [4], and NoCs [14, 36, 52], are multi-threaded. In other words, they allow multiple transactions between a master and a slave (pipelining), and concurrent transactions between different masters and slaves. Moreover, support for GALS operation where the IP-interconnect interface is asynchronous is common. Hence no unique trace exists anymore, as we have seen in Section 5.2.

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.

5.5.2 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 IEEE Standard 1149.1-2001, Test Access Port (TAP) [71]. This helps minimize the hardware cost.

5.5.3 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.

Ideally we deterministically follow the erroneous trace. Rather than collecting and storing information for replay (recall Figure 5.9b), 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 using off-chip debugger software.

5.5.4 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, we can observe only those clock cycles where information is transferred, i.e., by abstracting to handshakes. In Figure 5.4 this would correspond to observing only the communication behavior at the gray and black clock cycles, and ignoring the internal behavior at the white clock cycles. Conventional computation-centric debug can be used to observe the internal behavior of the IP blocks in isolation.

As an example, a DTL transaction request consists of a command and a number of data words (indicated by the command). Each of these can be individually abstracted to a handshake, called *element*. 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 5.11 shows several temporal abstraction levels: clock cycles, handshakes, messages, transactions, etc. Each time we combine a number of events to a coarser event that is meaningful and consistent by itself.

We also use structural and behavioral abstraction (refer to the left-hand side of Figure 5.11). Our debug observability involves retrieving the functional state (i.e., the bits in registers and memories) from the chip. We re-use the scan chains (the lowest level in Figure 5.11) that are inserted for manufactur-

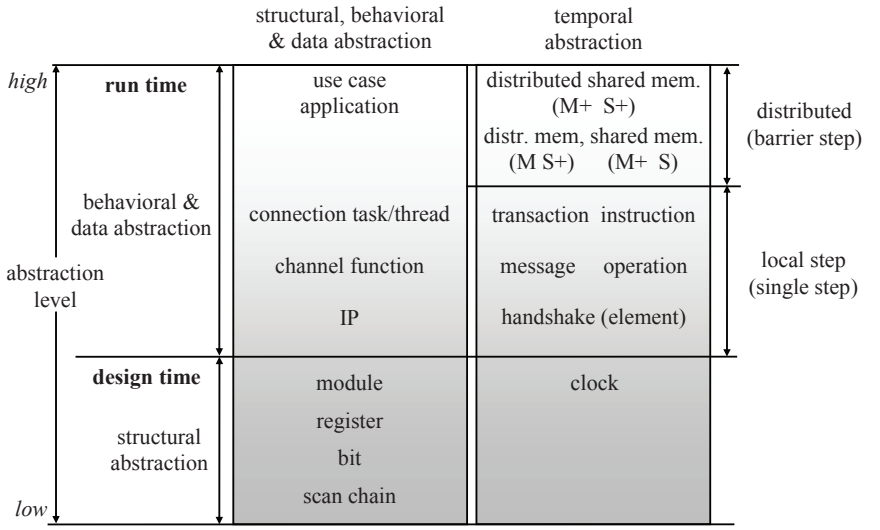


FIGURE 5.11: Debug abstractions.

ing test of the chip, when the system has stopped. This provides an intrusive means to "scan out" all or part of the state from the chip. The resulting state dump is a sequence of bits that still has to be mapped to registers and memories in gate-level and 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. In other words, we make use of behavioral information that allows us to interpret a set of registers. For example, a simple IP block, which implements a FIFO contains data registers, and read and write pointers. Without an abstraction from structure to behavior, they are all simply registers. At the functional IP level however, 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 how it has been programmed. Because we focus on communication, we move from structural interconnect components such as network interfaces (NIs) and routers to logical communication channels and connections that are used by applications. Processors execute functions, which are part of threads and tasks, which themselves in turn are part of the complete application. The application that runs on the

system depends on the use case. The implementation of these abstractions is described in Section 5.7.2.

5.6 On-Chip Debug Infrastructure

5.6.1 Overview

Dedicated debug IP modules have to be added to an SoC at design time to provide the debug functionality described in the previous sections. These modules include (refer to Figure 5.12):

- Monitors to observe the computation and/or communication and generate events
- Computation-specific instruments (CSIs) to act on these events and control the *computation inside* the IP blocks
- Protocol-specific instruments (PSIs) to act on these events and control the *communication between* the IP blocks
- An event distribution interconnect (EDI) to distribute the events from the monitors to the computation-specific instruments (CSIs) and protocol-specific instruments (PSIs)
- A debug control interconnect (DCI) to allow the programming of all debug blocks and querying of their status by off-chip debug equipment (see Section 5.7)
- A debug data interconnect (DDI) to allow access to the manufacturing-test scan chains to read out the complete state of the chip

The following subsections describe the functionality of each of these modules in more detail.

5.6.2 Monitors

Monitors observe the behavior of (part of) a chip while the chip is executing. They can be programmed to generate one or more events when a particular point in the overall execution of the system is reached [58], the system completes an execution step at a certain level of behavioral or temporal abstraction [24], or an internal system property becomes invalid [17]. These events can be distributed to subsequently influence either the system execution or the start or stop of real-time trace.

Monitors can also derive new data from the observed execution data of a system component by, for example, filtering [12] or compressing the information into a signature value using a multiple-input signature register

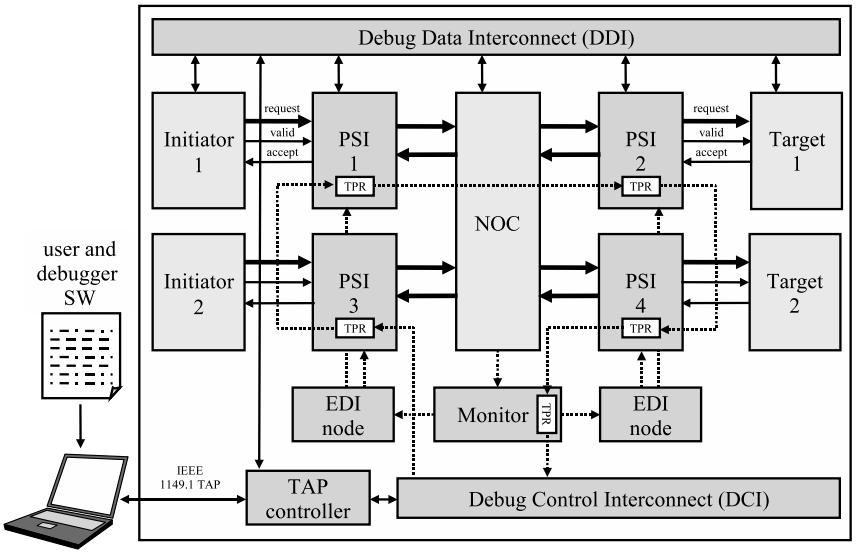


FIGURE 5.12: Debug hardware architecture.

(MISR) [66, 72]. As we focus on run/stop debugging, this type of monitor functionality falls outside the scope of this chapter.

Monitors are specialized to observe either the execution behavior of the *computation* (i.e., intra-IP) or the *communication* (i.e., inter-IP).

- *Computation* monitors can be added to the producers, the consumers, and the communication processing elements inside the communication architecture. CPUs traditionally include on-chip debug support [40], which enables an event to be generated when the program counter (PC) of the CPU reaches a certain memory address. This ability allows the event to be generated on reaching a certain function call, a single source code line, or an assembly instruction. When so required, events can also be generated at the level of clock cycles [28], by counting the number of clock cycles since the last CPU reset. For hardware accelerator IP blocks, custom event logic may be designed [70] that serves the purpose of partitioning the execution interval of an IP block into regular sections at possibly multiple levels of temporal abstraction.

- *Communication* monitors [11, 73] can be added on the interfaces of the producers, the consumers and the communication architecture, or within the communication architecture itself (i.e., in a NoC also on the interfaces between the routers and NIs). They observe the traffic and can generate events when either a transaction with a specific set of attributes is observed, and/or when a certain number of specific transactions have been communicated from a particular producer and/or to a particu-

lar consumer. As the communication protocols used in different chips may implement safe communication differently, a communication monitor may utilize a protocol-specific front end (PSFE) to abstract away these differences and provide the transaction data and attributes to a generic back end, which processes this data and determines whether the event condition has occurred. For a bus monitor, the filter criteria typically include an address range, a reference data value, an associated mask value, and optionally a transaction ID identifying the source of the transaction. A network monitor observes the packetized data stream on a link between two routers or between a router and a NI. Filter criteria may include whether the data on the link belongs to a packet header, a packet body, or the end of a packet, information on the quality of service (QoS) of the data (best effort (BE) or guaranteed throughput (GT)), whether a higher-level message has ended, and/or the sequence number of a data element in a packet.

Upon instantiation, the monitor is connected to a specific communication link, at which time the appropriate PSFE can be instantiated, based on the protocol agreed upon between the sender and the receiver [66]. The monitors are programmed and queried via the Debug Control Interconnect (DCI) (see Section 5.6.6 for details).

5.6.3 Computation-Specific Instrument

CSIs are instantiated inside or close to an IP block. Their purpose is to stop the execution of the component at a certain level of behavioral or temporal granularity when an event arrives. CPUs traditionally support interrupt handling, whereby the CPU's program flow is redirected to an interrupt vector look-up table on the arrival of an event. This table contains an entry for each type of interrupt (event) that can occur together with an address from which to continue execution. Debug events can be handled by an IP block as if it is an interrupt. Interrupts on the other hand can also be seen as signals that indicate the IP block's progression and can also be monitored.

Most CPUs support stalling the processor pipeline to halt execution in those cases where data first has to arrive from the communication architecture before its execution can continue. This stalling mechanism can be implemented either in the data path of the pipeline or in the control path (i.e. in the clock signal). In the latter option, special gating logic is added to the clock generation unit (CGU) [28] that prevents the pipeline from being clocked. These functional stalling mechanisms can be re-used for run/stop debugging to halt the execution of the processor at very low additional hardware cost.

Computation-specific Instruments (CSIs) are programmed and queried through the DCI to perform a specific action, such as starting, stopping, or single stepping, at a certain granularity (function entry/exit, source code line,

assembly instruction, clock cycle), when an event is received through the Event Distribution Interconnect (EDI).

5.6.4 Protocol-Specific Instrument

Section 5.2.2 described how we cannot always stop multiple IP blocks with asynchronous clocks such that their states are consistent. However, they can communicate safely with each other at different levels of abstraction, e.g., by using a valid-accept handshake as illustrated in Figure 5.2. By using the functional synchronization mechanisms, we can recover a consistent global state for debugging [24]. In Figure 5.2 the initiator raises its valid signal to indicate that the data it wishes to send is valid. The initiator stalls until the target signals that it consumed the data by raising the accept signal. The white circles in Figure 5.4 indicate these stall cycles of an IP block.

Essentially, because the internal state of the IP does not change while it is stalled, it can be safely sampled on any clock. In Figure 5.4 this is illustrated by the two black clock cycles. If the target does not accept the request handshake of the initiator then the dashed synchronization will not occur. The initiator will instead stall, allowing its state to be safely sampled.

We assume that all IP blocks communicate via an interconnect, such as a NoC [21], as shown in Figure 5.13.

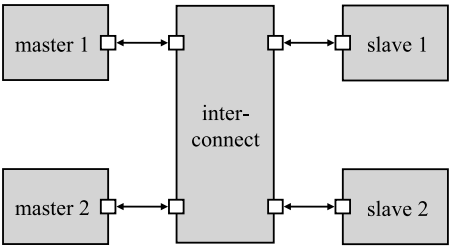


FIGURE 5.13: Example system under debug.

Every IP block will communicate at some point using the interconnect, possibly after some internal computation. If we control the handshakes between the IP blocks and the interconnect, it is possible to stall the IP blocks and the NoC when they offer a request or wait for a response. When all IP blocks are stalled, their states can be safely sampled, and a consistent global state is available.

However, note that the states are consistent in the sense that each IP block is in a stall state, waiting for a request or response. The global state may be inconsistent at a higher level of abstraction. For example, consider inter-IP communication based on synchronized tokens in a FIFO [49], described in Sections 5.2.3 and 5.3.2. Stopping at the level of transactions, many of which constitute the transfer of a single token, does not guarantee that a token is

either at the producer or the consumer. It may be partially produced, fully produced but not yet synchronized, etc. This can only be resolved by lifting the abstraction level yet again. In general, the Chandy-Lamport's "snapshot" algorithm [10] or derivatives thereof can be used to ensure that a collection of local states is globally consistent. Sarangi et al. [60] demonstrate this for source-synchronous multiprocessor debugs.

Protocol-specific instruments (PSIs) are instantiated on the communication interfaces of producers and consumers or inside the communication architecture where they control the data communication. A protocol-specific Instrument (PSI) is protocol-specific because it requires knowledge of the communication protocol to determine when a request or response is in progress, and when there are pending responses (for pipelined transactions). Based on this information and its program, a PSI can determine when it should stop the communication on a link after an event arrives from the EDI.

The communication on a bus is stopped by gating the handshake signals, thereby preventing the completion of the communication of the request or response. Communication requests are no longer accepted from the producers and no longer offered to the consumers. Responses are no longer accepted from the consumers nor offered to the producers.

Stopping the communication may take place at various levels of granularity, e.g., individual data elements, data messages, or entire transactions. PSIs are programmed through the DCI to perform a specific action, such as starting, stopping, or single stepping, at a certain behavioral or temporal granularity when an event is received through the EDI.

5.6.5 Event Distribution Interconnect

The EDI connects the event sources (the monitors) with the sinks (the CSIs and PSIs). The EDI acts as a high-speed broadcast mechanism that propagates events to all event sinks. Ideally, when an event is generated anywhere in the SoC, all on-going computation and communication execution steps are stopped as soon as possible, at their specified level of behavioral or temporal abstraction.

There are several possible ways to distribute a debug event:

1. Packet-level event distribution [62] uses the functional interconnect as an EDI. Re-using the functional interconnect does increase the demands on the communication infrastructure as the additional data volume has to be taken into account. This is undesirable because events are only generated during debugging and not during normal operation. Permanent bandwidth reservations can be made if the communication architecture supports this to avoid the "probing" effect the debug data has on the timing of the functional data. However, permanently reserving this bandwidth may be expensive.

2. Cycle-level event distribution [67]. A global, single-cycle event distribution is not scalable and difficult to implement independently from the final chip lay-out. In our solution, a network of EDI nodes is used that follows the NoC topology. The EDI node is parametrized in the number of neighboring nodes. Each node synchronously broadcasts at the NoC functional clock speed any events it receives from neighboring monitors or EDI nodes to the other EDI nodes in its neighborhood. This transport mechanism incurs one clock cycle delay for every hop that needs to be taken to reach the event sinks.

The latter method is the fastest, is scalable and re-uses the communication topology. Therefore it forms the basis of our EDI implementation. Event data travels as fast as or faster than the functional data that caused the event. This is quick enough to distribute an event to all CSIs and PSIs before the data on which the monitor triggered leaves the communication architecture. This is a very important property we can use for debug as it allows us to keep the data that caused the event within the boundaries of the communication architecture for a (potentially) infinite amount of time. The actual processing of this data by the targeted consumer can then be analysed at any required level of detail. This is achieved by subsequently controlling the delivery operation for this data at the required debug granularity by programming the PSI and CSI near the consumer from the debugger software (see Section 5.7).

5.6.6 Debug Control Interconnect

The purpose of the DCI is to allow the functionality of the debug components to be controlled and their status queried.

The DCI allows run-time access to the on-chip debug infrastructure from off-chip debug equipment independently and transparently from the functional operation of the SoC. Examples of debug status information include whether any of the programmed events inside the monitors have already occurred, and/or whether the computation or communication inside the system has been stopped in response.

The state of the monitors, the PSIs and the CSIs becomes observable and controllable via so-called test point registers (TPRs) that connect to a IEEE Standard 1149.1-2001, TAP Controller (TAPC) as user-defined data registers [35]. These TPRs can be accessed and therefore programmed and queried using one or more user-defined instructions in the TAPC.

5.6.7 Debug Data Interconnect

The purpose of the Debug Data Interconnect (DDI) is to allow the system state to be observed and controlled after an event has stopped the relevant computation and communication.

Once the execution of a chip has come to a complete stop, preventing

debug accesses from disturbing its execution is no longer a concern. The only concern is storage of the state inside the IP blocks.

We use the manufacturing-test scan chains to implement the DDI, as proposed by [32, 57, 71] and use a standard design flow with commercial, off-the-shelf (COTS) gate-level synthesis and scan-chain insertion. The IEEE 1149.1-compliant scan-based manufacturing test and debug infrastructure are made accessible from the TAP. Using the TAPC, data can be scanned out of the chip for use by the off-chip debug infrastructure described next.

5.7 Off-Chip Debug Infrastructure

5.7.1 Overview

This section presents the off-chip debug infrastructure and describes the techniques it can use to raise the debug abstraction level above the bit- and clock-cycle level, as depicted in Figure 5.11. We also present a generic debug application programmer's interface (API), which allows debug controllability and observability at the behavioral computation and communication level.

Figure 5.14 shows a generic, off-chip debug infrastructure. Our debugger software, called the integrated circuit debug environment (InCiDE) [69], connects to the debug port of the chip in potentially different user environments. Figure 5.14 shows a simulation environment, a field-programmable gate array (FPGA)-based prototyping environment, and a real product environment as three examples. The debugger software gains access the on-chip debug functionality through the debug interface, as described in Section 5.6. The debugger software allows the user to place (parts of) the SoC in functional or debug mode, and to inspect or modify the state of functional IP blocks or debug components.

5.7.2 Abstractions Used by Debugger Software

The InCiDE debugger software is layered and performs *structural*, *data*, *behavioral*, and *temporal* abstractions (refer to Figure 5.11) to provide the user with a high-level debug interface to the device-under-debug (DUD). Each abstraction function is described in more detail in the following subsections.

5.7.2.1 Structural Abstraction

Structural abstraction is achieved by applying the following three consecutive steps.

1. *Target Abstraction*

   Target-specific drivers are used to connect the debugger software using

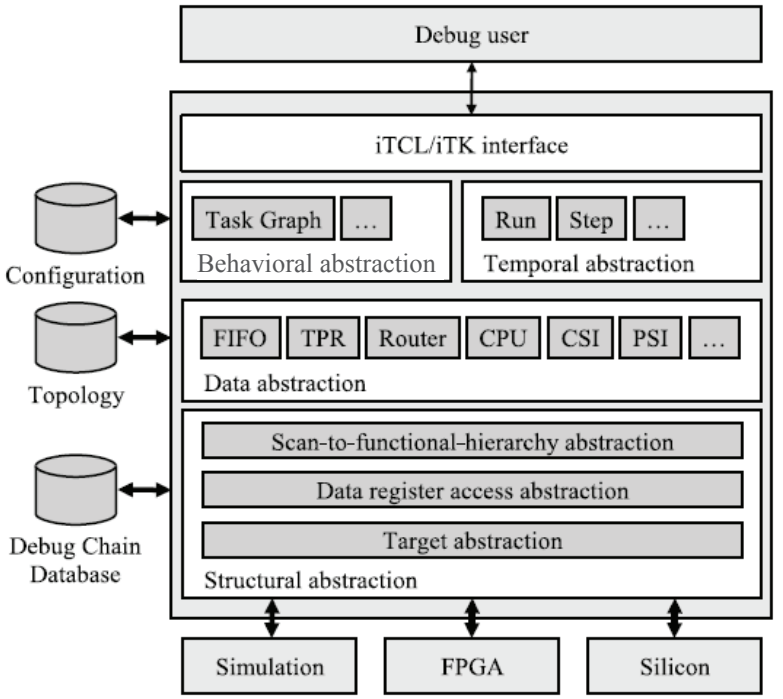


FIGURE 5.14: Off-chip debug infrastructure with software architecture.

the same software API to different implementation types of the DUD. Debug targets include simulation, FPGA prototyping, and product environments. A target driver enables access to the TAPC in its corresponding environment and allows performing capture, shift, and update operations on user data registers connected to the TAPC. An example tool control language (TCL) function call may look like Listing 5.1.

Listing 5.1: Writing and reading a user-defined data register.

```
1 set result [tap_write_read [list 0100 01011]]
```

which will shift the binary string "01011" (right-bit first) into the user-defined data register belonging to the TAPC binary instruction opcode "0100" via the test data input (TDI). The bit-string that is returned contains the values captured on the test data output (TDO) pin of the TAP on successive test clock (TCK) cycles during this shift operation. This layer also provides the `tap_reset` and `tap_nop` n commands to reset the TAPC and have no operation for n TCK cycles, respectively.

2. *Data Register Access Abstraction*

   The mechanisms to access the various user-defined data registers connected to the TAPC are not always identical. For example, access to the debug scan chain requires that other user data registers are programmed first. As described in Section 5.6, this scan chain is connected as a user data register to the TAPC. To access it, the circuit first has to be switched from functional mode to debug scan mode and its functional clock(s) switched to the clock on the TCK input. In our architecture [71], a test control block (TCB) is used for this. The TCB is also mapped as a user-defined data register under the TAPC but can be accessed directly, i.e. without having to program another user-defined data register first. To access the debug scan chain, this layer therefore takes care of first programming the TCB to subsequently enable operations on the debug scan chain. For instance, the previous access to the debug scan chain is "wrapped" by this layer into Listing 5.2, while binary instruction opcodes are also replaced by more understandable instruction names.

   Listing 5.2: Abstracting away from TAPC data register access details.

```
1 set result [tap_write_read [list \
2   PROGRAM_TCB <debug mode> \
3   DBG_SCAN    01011 \
4   PROGRAM_TCB <functional mode> \
5 ]]
6 set result [lindex $result 1]
```

   This layer hides the subtle differences in the exact bit strings that are needed to enable access to the debug scan chain in different SoCs.

3. *Scan-to-Functional Hierarchy Abstraction*

   This layer replaces the scan-oriented method of accessing flip-flops in user-defined data registers with a more design(er)-friendly method of accessing flip-flops and registers using their location in the RTL hierarchy. A multi-bit RTL variable or signal may be mapped to multiple flip-flops during synthesis. This layer utilizes this mapping information from the synthesis step to reconstructs the values of RTL variables and signals during debug from the values in their constituent flip-flops. In addition, it groups those signals and variables into hierarchical modules. A designer using this system can refer to signal and variable names using their RTL hierarchical identifiers and retrieve and set their values without needing to know the details about the TAPC, its user-defined instructions and data registers.

   For example, the purpose of the previous access, shown in Listing 5.2 may have been to set the value of a five-bit RTL signal "usoc.unoc.u1router.be_queue.wrptr" to 0x0B ("01011"). Using this layer, this can now be accomplished by executing the code in Listing 5.3.

Listing 5.3: Setting and querying a register.

```
1 dcd_set usoc.unoc.u1router.be_queue.wrptr 0x0B
2 dcd_synchronise
3 puts [dcd_get usoc.unoc.u1router.be_queue.wrptr HEX]
```

This layer takes care of mapping the individual bits of the value 0x0B into the correct bits inside the debug scan chain. The "dcd_synchronise" function is used to send the resulting chain to the chip and retrieve the previous content of the on-chip chain. The "puts" command prints the value of the register just retrieved from the chip.

These three structural abstraction steps are design-independent and are the consequences of our choice to access the state in the design using manufacturing-test scan chains mapped to the TAPC. They can therefore be applied to any digital design that utilizes the same on-chip debug architecture as presented in Section 5.6. They do however require structural information from various stages in the design and design for test (DfT) process, specifically the mapping information of RTL signals and variables to scannable flip-flops in the design, the location of these flip-flops in the resulting user-defined data registers, and specific TAPC instructions to subsequently enable access to these user-defined data registers. In Figure 5.14 all this information is stored in the debug chain database, which is automatically generated by our debugger software InCiDE.

5.7.2.2 Data Abstraction

The second abstraction technique employed by the debugger software is data abstraction. Based on the design's topology information, the debugger software can represent the state of known building blocks at a higher level than individual RTL signals or values.

For example, this layer can represent the state of a FIFO as its set of internal signals, including its memory, its read and its write pointers using the structural abstraction layers. If a design instance called "usoc.unoc.u1router.be_queue" is an 8-entry, 32-bit word FIFO, the user could use the command in Listing 5.4 to display its current state.

Listing 5.4: Querying individual registers of a FIFO.

```
1 dcd_synchronise
2 puts [dcd_get usoc.unoc.u1router.be_queue.mem    HEX ]
3 puts [dcd_get usoc.unoc.u1router.be_queue.wrptr HEX ]
4 puts [dcd_get usoc.unoc.u1router.be_queue.rdptr HEX ]
```

resulting in output such as

```
0x00000000
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007
0x3
0x5
```

However, the user can also use the data abstraction layer and use the command in Listing 5.5

Listing 5.5: Printing the state of a FIFO.

```
1 print_fifo usoc.noc.u1router.be_queue VALID_ONLY HEX
```

to get

```
 ----------------------------
| usoc.unoc.u1router.be_queue |
|----------------------------|
| Nr |        DATA           |
|----|-----------------------|
| 03 |             0x00000003 |
| 04 |             0x00000004 |
 ----------------------------
```

Note how the software has interpreted the values of the read and write pointer to only print the valid entries in the FIFO ("VALID_ONLY"). Similar data abstraction functions have been implemented for the other standardised design modules, such as the monitors, CSIs, PSIs, routers, NIs and CPUs. In addition, these abstraction functions can be nested, e.g. the data abstraction function for the router may call multiple FIFO data functions to display the state of all its BE queues. The design knowledge required for this is contained in the "topology" file shown in Figure 5.14, which is automatically generated by the NoC design flow [20, 26].

5.7.2.3 Behavioral Abstraction

The previous two abstraction techniques focused on providing an abstracted state view and structural interconnectivity of common IP blocks. Behavioral abstraction targets the abstraction of the programmable functionality of these blocks. For example, two IP blocks communicate via two NIs and several routers. A monitor observes the communication data in Router R3 (refer to Figure 5.15).

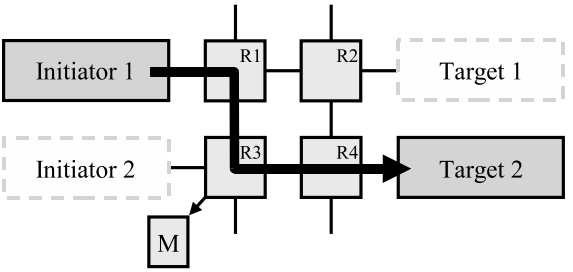


FIGURE 5.15: Physical and logical interconnectivity.

The exact IP modules that are involved depend not only on the physical interconnectivity but also on the programming of these IP blocks. For debugging a problem at the task graph level, we are first interested in the logical connection between these blocks. Only when there appears to be something physically wrong with this logical connection, do we refine the state view and look at their physical interconnectivity. A debug user can for instance issue a command as shown in Listing 5.6.

Listing 5.6: Querying the routers in the NoC.

```
1 set routers [get_router [get_conn {uc3 initiator1 target2}]]
```

This command provides a list of all routers that the logical connection between Initiator 1 and Target 2 uses in Use Case 3. With the data abstraction functions from the previous subsection, the user is able to display the states of these routers at the required level of detail.

Enabling debug at the behavioral level requires knowledge of the active use case, i.e., the programming of the NoC. This information is contained in the "configuration" file shown in Figure 5.14, which is automatically generated by the NoC design flow [20, 27].

5.7.2.4 Temporal Abstraction

A fourth debug abstraction technique is temporal abstraction. Traditionally debugging takes place at the clock cycle level of the CPU that is debugged. A disadvantage of this technique is that in a non-deterministic system the same event is unlikely to occur at the exact same clock cycle in multiple runs. Therefore temporal abstraction couples the debug execution control to events that are more meaningful to measure the progress made in the system's execution. Examples that are enabled using the hardware described in Section 5.6 include "Run until Initiator 1 or 2 initiates a transaction," and "Allow Target 2 to return 5 responses" before stopping the on-chip computation and/or communication [23].

Temporal abstraction first allows multiple clock cycles to be abstracted to one or more data element handshakes (refer to Figure 5.11). Protocol infor-

mation on the handshake signals is used for this. The steps to messages on channels and to transactions on connections move the temporal abstraction level to the logical communication level.

The two subsequent temporal abstraction steps in Figure 5.11 are more complex as they involve the synchronized stepping of multiple communication channels. For this a basic single step for a communication channel is defined as *all* PSIs involved leaving their stopped state and process one communication request. The TCL command "`step $L -n` S" performs S single steps in succession for all PSIs in List L. For multiple channels, all stopped PSIs of the channels involved will need to process one communication request.

Note that this *single step* method forces a unique transaction order that must be known in advance to accurately represent the original use case. Otherwise there can be unwanted dependencies between the channels that are single-stepped, which potentially can lead to a deadlock. For this reason we also introduce the *barrier stepping* method and a corresponding TCL command extension "`step $L -n` S `-some` N," where *at least* N out of all PSIs in List L must perform a single step [23]. Barrier stepping is equal to single stepping when N is equal to the size of List L.

5.8 Debug Example

In this section we describe the application of the on-chip and off-chip debug infrastructure of Sections 5.6 and 5.7 using the example in Figure 5.12 and the NoC topology shown in Figure 5.15. We run our debugger software InCiDE with its extended API to perform interactive debugging using a simulated target. The following listing and output demonstrate the use of the API to control the communication inside the SoC during debug.

Listing 5.7: Example debug use case.

```
1 tap_reset
2 tap_nop 1000
3 set my_conn [get_conn {uc3 initiator1 target2}]
4 set my_routers [get_router $my_conn]
5 set my_router [lindex $my_conn 1]
6 set my_mon [get_monitor $my_router]
7 set_mon_event $my_mon {-fw 2 -value 0x0E40}
```

Line 1 resets the TAPC and Line 2 provides enough time for the system boot code [27] to functionally program the NoC. Lines 3 and 4 find the connection ("$my_conn") between Initiator 1 and Target 2 for the active use case , and the routers ("$my_routers") involved in the connection between Initiator 1 and Target 2. Note that on Line 5 we select the second router (Router R3) from the list of routers, and retrieve the monitor connected to it (refer to

Figure 5.15). This monitor is programmed on Line 7 to generate an event when the third word in a flit ("-fw 2") is equal to 0x0E40.

```
8  set my_tpr [get_tpr [get_psi $my_conn M req]]
9  set_psi_action $my_tpr -gran e -cond edi
10 dcd_synchronise tpr
11 tap_nop 1000
12 dcd_synchronise tpr
13 print_tpr $my_tpr
```

Lines 8 and 9 find the TPR of the PSI on the master request side of the connection between Initiator 1 and Target 2. This PSI TPR is programmed to stop all communication at the granularity of elements ("-gran e") when an event comes in via the EDI ("-cond edi"). Lines 10 and 11 write the resulting TPR debug program into the chip, and wait 1000 TCK cycles. On Line 12 the chip content is read back and on Line 13 the content of the PSI TPR is printed. This results in the following output.

```
 ------------------------------------------------------------
|             {initiator1 pi} -> {core4 pt}                  |
|------------------------------------------------------------|
|Ch. Type | St.En. | St. Gran. | St. Cond. | St.St. | Left |
|---------|--------|-----------|-----------|--------|------|
|   Req   |  Yes   |  Element  |    EDI    |  Yes   | Yes  |
|   Resp  |   No   |  Message  |    EDI    |   No   |  No  |
 ------------------------------------------------------------
```

This table confirms that between Initiator 1 and its network interface ("core4"), the PSI was programmed to stop the communication on the request channel at the element level when an event comes in from the EDI. The PSI has entered the stop state ("St.St.") on the request channel.

```
14 continue $my_tpr
15 dcd_synchronise tpr
16 print_tpr $my_tpr
```

Line 14 continues the communication on the request channel, while Lines 15 and 16 query the TPR state, resulting in the following output.

```
 ------------------------------------------------------------
|             {initiator1 pi} -> {core4 pt}                  |
|------------------------------------------------------------|
|Ch. Type | St.En. | St. Gran. | St. Cond. | St.St. | Left |
|---------|--------|-----------|-----------|--------|------|
|   Req   |  Yes   |  Element  |    EDI    |   No   | Yes  |
|   Resp  |   No   |  Message  |    EDI    |   No   |  No  |
 ------------------------------------------------------------
```

We observe that the PSI has left the stop state and is currently running, waiting for another event from the EDI. We now retrieve all PSI TPRs on a master request side. We program these to stop at the element level when an

event comes in via the EDI. We subsequently generate an event on the EDI via the TAP using the "stop" command.

```
17 set my_tpr_all [get_tpr [get_psi * M req]]
18 set_psi_action $my_tpr_all -gran e -cond edi
19 stop
```

Once all transactions have stopped, we perform barrier stepping. We request that three execution steps are taken (at the granularity of data elements) by at least two PSIs ("-some 2") with verbose output ("-v").

```
20 step $my_tpr_all -n 3 -some 2 -v
```

This results in the following output.

```
- INFO: Checking if all Elements have stopped.....
- INFO: All Elements have stopped.
- INFO: Stepping starts.
- INFO: step 1 finished.
- INFO: step 2 finished.
- INFO: step 3 finished.
- INFO: All Elements are stopped.
```

The printed INFO lines show our barrier stepping algorithm at work. It first checks whether all selected PSIs ("$my_tpr_all") have entered their stopped state. If so, the software continues all PSIs. It subsequently polls whether at least two have since left and returned to their stopped state. When this has happened, the software will issue continue commands for those PSIs only and initiating the second step. This continues until for a third time, at least two PSIs have exited and re-entered their stopped state. Once barrier stepping is completed, we can read the content of the chip and print the content of the router.

```
21 dcd_synchronise
22 print_router $my_router HEX
```

This results for example in the following output.

```
 --------------------
| BE queue of R3_p1  |
|--------------------|
| Q.Nr |    DATA     |
|------|-------------|
|  18  | 0x200000123 |
|  19  | 0x300000124 |
 --------------------
- INFO: No valid data in GT queue of R3_p1.
```

In addition, we can print the state of the network interface.

```
23 print_ni [get_ni conn $my_conn M req ] HEX
```

This results in the following output.

```
 -------------------------
| INPUT queue of NI000_p2 |
|-------------------------|
| Q.Nr |       DATA       |
|------|------------------|
|  21  |       0x08000004 |
|  22  |       0x00000108 |
|  23  |       0x00000109 |
|  24  |       0x0000010A |
|  25  |       0x0000010B |
 -------------------------
- INFO: No valid data in OUTPUT queue of NI000_p2.
```

5.9 Conclusions

In this chapter, we introduced three fundamental reasons why debugging a multi-processor SoC is intrinsically difficult; (1) limited internal observability, (2) asynchronicity, and (3) non-determinism. The observation of the root cause of an error is limited by the available amount of bandwidth to off-chip analysis equipment. Capturing a globally consistent state in a GALS system may not be possible at the level of individual clock cycles. In addition, an error may manifest itself in some runs of the system but not in others.

We classified existing debug methods by the information (scope), the detail (data abstraction), and the information frequency (temporal abstraction) they provide about the system. Debug methods are either intrusive or not. We subsequently introduced our communication-centric, scan-based, run/stop-based, and abstraction-based debug method, and described in detail the required on-chip and off-chip infrastructure that allows users of our debug system to debug an SoC at several number of levels of abstraction. We also illustrated our debug approach using a simple example system.

The analysis and methods presented in this chapter are only the first steps toward addressing the problem of debugging an SoC using a scientific approach. The use of on-chip DfD components, and debug abstraction techniques implemented in off-chip debugger software are ingredients for an overall SoC debug system. This system should link hardware debug to software debug, for SoCs with distributed computation, and using deterministic or guided replay.

A significant amount of research still needs to be carried out to reach this goal. This includes, for example, understanding and determining what parts of a system need to be monitored, and what parts must be controlled during debug and in what manner. More generally, pre-silicon verification and post-silicon debug methods and tools need to be brought together for seamless verification and debug throughout the SoC design process, and to prevent gaps in the verification coverage, and duplication of debug functionality.

Review Questions

[Q_1] Explain why the internal observability is limited in modern embedded systems.

[Q_2] Using multiple, asynchronous clock domains complicates debugging more than a single clock domain. Explore why designers utilize multiple, asynchronous clock domains when this is the case.

[Q_3] Describe the effect multiple, asynchronous clock domains have on the observation of a consistent global state.

[Q_4] What is the difference between a system run and a system trace?

[Q_5] Which three orthogonal classes of error observation for embedded systems have been explained in this chapter, and what types of errors occur in each class?

[Q_6] Describe how a single, unmodified system can produce multiple traces.

[Q_7] Describe the steps of the ideal debug flow.

[Q_8] List the four abstraction techniques presented in this chapter, and explain their role in the debug process.

[Q_9] Name three optical or physical debug techniques.

[Q_10] Explain the differences between, on the one hand, optical and physical debug techniques, and on the other hand, logical debug techniques.

[Q_11] What is deterministic replay and what are its requirements?

[Q_12] Name the four key characteristics of the CSAR debug approach.

[Q_13] List the required on-chip functionality to support the CSAR debug approach

[Q_14] Describe the functionality of the off-chip debug software in relation to the four abstraction techniques described in this chapter.

Bibliography

[1] D.A. Abramson and R. Sosic. Relative Debugging Using Multiple Program Versions. In *Int'l Symposium on Languages for Intensional Programming*, 1995.

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

[3] ARM. AMBA Specification. Rev. 2.0, 1999.

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

[5] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*. 2008.

[6] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. In *Building the Information Society,* ed. René Jacquart. Dependability And Its Threats: A Taxonomy, pages 91–120. Kluwer, 2004.

[7] C. Beddoe-Stephens. Semiconductor Wafer Probing. *Test and Measurement World*, pages 33–35, November 1982.

[8] Michael Bedy, Steve Carr, Xianlong Huang, and Ching-Kuang Shene. A Visualization System for Multithreaded Programming. *SIGCSE Bulletin*, 32(1):1–5, 2000.

[9] British Standards Institute. British Standard BS 5760 on Reliability of Systems, Equipment and Components.

[10] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[11] Călin Ciordaş, Kees Goossens, Twan Basten, Andrei Rădulescu, and Andre Boon. Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective. In *Proc. Symposium on Industrial Embedded Systems (IES)*, pages 1–10, Antibes, France, October 2006. IEEE.

[12] Călin Ciordaş, Andreas Hansson, Kees Goossens, and Twan Basten. A Monitoring-aware Network-On-Chip Design Flow. *Journal of Systems Architecture*, 54(3-4):397–410, March 2008.

[13] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 755–763, September/October 2003.

[14] Giovanni De Micheli and Luca Benini, editors. *Networks on Chips: Technology and Tools*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, July 2006.

[15] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design and Test of Computers*, pages 21–31, September/October 2001.

[16] Marc Eisenstadt. My Hairiest Bug War Stories. *Communications of the ACM*, 40(4):30–37, April 1997.

[17] Jeroen Geuzebroek and Bart Vermeulen. Integration of Hardware Assertions in Systems-on-Chip. In *Proc. IEEE Int'l Test Conference (ITC)*, 2008.

[18] Holger Giese and Stefan Henkler. Architecture-Driven Platform Independent Deterministic Replay for Distributed Hard Real-Time Systems. In *Proc. ISSTA Workshop on the Role Of Software Architecture for Testing and Analysis*, pages 28–39, 2006.

[19] Kees Goossens, Martijn Bennebroek, Jae Young Hur, and Muhammad Aqeel Wahlah. Hardwired Networks on Chip in FPGAs to Unify Data and Configuration Interconnects. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 45–54. IEEE Computer Society, April 2008.

[20] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema. A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1182–1187, Washington, DC, USA, March 2005. IEEE Computer Society.

[21] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, Sept-Oct 2005.

[22] Kees Goossens, Om Prakash Gangwal, Jens Röver, and A. P. Niranjan. Interconnect and Memory Organization in SOCs for Advanced Set-Top Boxes and TV — Evolution, Analysis, and Trends. In Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15, pages 399–423. Kluwer, 2004.

[23] Kees Goossens, Bart Vermeulen, and Ashkan Beyranvand Nejad. A High-Level Debug Environment for Communication-Centric Debug. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.

[24] Kees Goossens, Bart Vermeulen, Remco van Steeden, and Martijn Bennebroek. Transaction-Based Communication-Centric Debug. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 95–106, Washington, DC, USA, May 2007. IEEE Computer Society.

[25] Jim Gray. Why Do Computers Stop and What Can Be Done about It? In *Proc. Symposium on Reliablity in Distributed Software and Database Systems*, 1986.

[26] Andreas Hansson. A Composable and Predictable On-Chip Interconnect. PhD thesis, Eindhoven University of Technology, June 2009.

[27] Andreas Hansson and Kees Goossens. Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 233–242, Washington, DC, USA, May 2007. IEEE Computer Society.

[28] H. Hao and K. Bhabuthmal. Clock Controller Design in SuperSPARC II Microprocessor. In *Proc. Int'l Conference on Computer Design (ICCD)*, pages 124–129, Austin, TX, USA, October 2–4, 1995.

[29] Timothy L. Harris. Dependable Software Needs Pervasive Debugging. In *Proc. Workshop on ACM SIGOPS*, pages 38–43, New York, NY, USA, 2002. ACM.

[30] C.F. Hawkins, J.M. Soden, E.I. Cole Jr., and E.S. Snyder. The Use of Light Emission in Failure Analysis of CMOS ICs. In *Proc. Int'l Symposium for Testing and Failure Analysis (ISTFA)*, 1990.

[31] Matthew W. Heath, Wayne P. Burleson, and Ian G. Harris. Synchrotokens: A Deterministic GALS Methodology for Chip-level Debug and Test. *IEEE Transactions on Computers*, 54(12):1532–1546, December 2005.

[32] Kalon Holdbrook, Sunil Joshi, Samir Mitra, Joe Petolino, Renu Raman, and Michelle Wong. microSPARC: A Case Study of Scan-Based Debug. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 70–75, 1994.

[33] Yu-Chin Hsu, Furshing Tsai, Wells Jong, and Ying-Tsai Chang. Visibility Enhancement for Silicon Debug. In *Proc. Design Automation Conference (DAC)*, 2006.

[34] William Huott, Moyra McManus, Daniel Knebel, Steven Steen, Dennis Manzer, Pia Sanda, Steven Wilson, Yuen Chan, Antonio Pelella, and Stanislav Polonsky. The Attack of the "Holey Shmoos": A Case Study of Advanced DFD and Picosecond Imaging Circuit Analysis (PICA). In *Proc. IEEE Int'l Test Conference (ITC)*, page 883, Washington, DC, USA, 1999. IEEE Computer Society.

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

[36] Axel Jantsch and Hannu Tenhunen, editors. *Networks on Chip*. Kluwer, 2003.

[37] D.D. Josephson, S. Poehhnan, and V. Govan. Debug Methodology for the McKinley Processor. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 665–670, Oct 2004.

[38] A.C.J. Kienhuis. Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools. PhD thesis, Delft University of Technology, 1999.

[39] Herman Kopetz. The Fault Hypothesis for the Time-Triggered Architecture, In *Building the Information Society,* ed. René Jacquart, pages 221–234. Kluwer, 2004.

[40] Norbert Laengrich. Adapting Hardware-assisted Debug to Embedded Linux and Other Modern OS Environments. *PC/104 Embedded Solutions Journal of Small Embedded Form Factors*, 2006.

[41] Rick Leatherman and Neal Stollon. An Embedded Debugging Architecture for SoCs. *IEEE Potentials*, 24(1):12–16, Feb-Mar 2005.

[42] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[43] Bill Lewis. Debugging Backwards in Time. In *International Workshop on Automated Debugging*, October 2003.

[44] Michael R. Lyu, editor. *Handbook of Software Reliability and System Reliability.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[45] Thomas Frederick Melham. Formalising Abstraction Mechanisms for Hardware Verification in Higher Order Logic. PhD thesis, University of Cambridge, August 1990. Also available as Technical Report UCAM-CL-TR-201.

[46] MIPS Technologies. PDTrace Interface Specification., 2002.

[47] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical Design of Globally-Asynchronous Locally-Synchronous Systems. In *Proc. Int'l Symposium on Asynchronous Circuits and Systems (ASYNC)*, April 2000.

[48] N. Nataraj, T. Lundquist, and Ketan Shah. Fault Localization Using Time Resolved Photon Emission and Still Waveforms. In *Proc. IEEE Int'l Test Conference (ITC)*, volume 1, pages 254–263, September 30–October 2, 2003.

[49] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino Busá, Kees Goossens, Rafael Peset Llopis, and Paul Lippens. C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *ACM Tansactions on Design Automation for Embedded Systems*, 7(3):233–270, 2002.

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

[51] M. Paniccia, T. Eiles, V. R. M. Rao, and Wai Mun Yee. Novel Optical Probing Technique for Flip Chip Packaged Microprocessors. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 740–747, Washington, DC, USA, October 1998.

[52] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures.* Morgan Kaufmann, 2008.

[53] Stephen E. Paynter, Neil Henderson, and James M. Armstrong. Metastability in Asynchronous Wait-Free Protocols. *IEEE Trans. Comput.*, 55(3):292–303, 2006.

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

[55] Bill Roberts. The Verities of Verification. *Electronic Business*, January 2003.

[56] Michiel Ronsse and Koen de Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. In *ACM Transactions on Compuer Systems*, volume 17, pages 133–152, May 1999.

[57] G.J. Rootselaar and B. Vermeulen. Silicon Debug: Scan Chains Alone Are Not Enough. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 892–902, Atlantic City, NJ, USA, September 1999.

[58] G.J. van Rootselaar, F. Bouwman, E.J. Marinissen, and M. Verstraelen. Debugging of Systems on a Chip: Embedded Triggers. In *Proc. Workshop on High-Level Design Validation and Test (HLDVT)*, 1997.

[59] J. A. Rowlette and T. M. Eiles. Critical Timing Analysis in Microprocessors Using Near-IR Laser Assisted Device Alteration (LADA). In *Proc. IEEE Int'l Test Conference (ITC)*, volume 1, pages 264–273, September 30–October 2, 2003.

[60] Smruti R. Sarangi, Brian Greskamp, and Josep Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *Proc. IEEE Int'l Conference on Dependable Systems and Networks*, pages 301–312, Washington, DC, USA, 2006. IEEE Computer Society.

[61] B. Tabbara and K. Hashmi. Transaction-Level Modelling and Debug of SoCs. In *Proc. IP SOC Conference*, 2004.

[62] Shan Tang and Qiang Xu. In-band Cross-trigger Event Transmission for Transaction-based Debug. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 414–419, New York, NY, USA, 2008. ACM.

[63] Radu Teodorescu and Josep Torrellas. Empowering Software Debugging Through Architectural Support for Program Rollback. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[64] Stephen H. Unger. Hazards, Critical Races, and Metastability. *IEEE Trans. Comput.*, 44(6):754–768, 1995.

[65] H. J. M. Veendrick. The Behaviour of Flip-flops Used as Synchronizers and Prediction of Their Failure Rate. *IEEE Journal of Solid-State Circuits*, 15(2):169–176, April 1980.

[66] Bart Vermeulen and Kees Goossens. A Network-on-Chip Monitoring Infrastructure for Communication-centric Debug of Embedded Multi-Processor SoCs. In *Proc. Int'l Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2009.

[67] Bart Vermeulen, Kees Goossens, and Siddharth Umrani. Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 3–12. IEEE Computer Society, April 2008.

[68] Bart Vermeulen, Yu-Chin Hsu, and Robert Ruiz. Silicon Debug. *Test and Measurement World*, pages 41–45, October 2006.

[69] Bart Vermeulen and Gert Jan van Rootselaar. Silicon Debug of a Co-processor Array for Video Applications. In *Proc. Workshop on High-Level Design Validation and Test (HLDVT)*, pages 47–52, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[70] Bart Vermeulen, Mohammad Z. Urfianto, and Sandeep K. Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proc. Design Automation Conference (DAC)*, pages 514–517, New York, NY, USA, 2004. ACM.

[71] Bart Vermeulen, Tom Waayers, and Sandeep K. Goel. Core-based Scan Architecture for Silicon Debug. In *Proc. IEEE Int'l Test Conference (ITC)*, pages 638–647, Baltimore, MD, USA, October 2002.

[72] Joon-Sung Yang and N.A. Touba. Enhancing Silicon Debug via Periodic Monitoring. In *Proc. Int'l Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 125–133, October 2008.

[73] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. Int'l Symposium on Computer Architecture*, 2004.