Embedded Systems

Bart Vermeulen
Kees Goossens

Debugging Systems-on-Chip

Communication-centric and Abstraction-based Techniques

Springer

Embedded Systems

This Series addresses current and future challenges pertaining to embedded hardware, software, specifications and techniques. Titles in the Series cover a focused set of embedded topics relating to traditional computing devices as well as high-tech appliances used in newer, personal devices, and related topics. The material will vary by topic but in general most volumes will include fundamental material (when appropriate), methods, designs and techniques.

More information about this series at http://www.springer.com/series/8563

Bart Vermeulen • Kees Goossens

Debugging Systems-on-Chip

Communication-centric and Abstraction-based Techniques

Springer

Bart Vermeulen
NXP Semiconductors
Eindhoven
The Netherlands

Kees Goossens
Faculty of Electrical Engineering
Eindhoven University of Technology
Eindhoven
The Netherlands

ISSN 2193-0155 ISSN 2193-0163 (electronic)
ISBN 978-3-319-06241-9 ISBN 978-3-319-06242-6 (eBook)
DOI 10.1007/978-3-319-06242-6
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014942659

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

We can trace the origin of this book back to the year 2003, when we first started working more closely together. On the one hand, Bart had been working on system on chip (SOC) debugging since joining Philips Research in 1997. He had architected with other colleagues a low-cost, core-based scan architecture to provide bit-level access to the state of a digital SOC via an IEEE 1149.1 test access port.

Two problems that still needed to be addressed however were (1) how to set appropriate execution breakpoints in a generic SOC, and (2) how to extract and interpret a consistent state from a globally-asynchronous locally-synchronous (GALS) SOC. Earlier analyses had shown that state data might become inconsistent on clock domain boundaries, when a GALS SOC is stopped at the clock cycle level.

Kees, on the other hand, had been working on real-time networks on chip (NoCs) for use in SOCs for Philips's advanced set-top boxes. Apart from designing the NoC itself, he worked on debugging NoC-based systems, together with other colleagues, by monitoring packets inside the NoC and reconstructing transactions.

We realized at this point that these two debugging problems may be solved by utilizing the communication protocol agreements between pairs of SOC building blocks. Through the use of protocols, we may raise the level of abstraction of both the on-chip breakpoints and of the interpretation of the SOC state. In addition, we envisioned that manipulating the transactions at the NoC boundary for debugging would be more intuitive, simpler, and precise.

For the next six years, we worked together with students to further explore and validate these concepts. A sizable number of publications were written to document our progress. In 2009, Bart started to write his PhD thesis on this subject matter, to bring together these two fields in a consistent manner and to document all our research results so far. The book in front of you is based on this thesis, which Bart successfully defended at the Eindhoven University of Technology in December 2013.

In this book, we first identify the key requirements for debugging a modern, silicon SOC implementation, and the technical factors that complicate this debugging task. We then propose our novel communication-centric, scan-based, abstraction based, run/stop-based (CSAR) debug approach with associated on-chip debug architecture, electronic design automation (EDA) tools, and off-chip debugger software that addresses these requirements and complicating factors. This debug infrastructure allows a debug engineer to localize faults in silicon SOC implementations, by

controlling its execution and observing the internal state of an SOC at different levels of abstraction. We subsequently evaluate the efficiency and effectiveness of aspects of our approach and its supporting infrastructure using six industrial SOCs and an illustrative, example SOC model. We also quantify the hardware cost and design effort to support our approach.

This book is generally intended for readers interested in SOCs. We have written it for senior architects and engineers, as well as academics (both teachers and students), who look to better understand the problems involved in debugging digital integrated circuits (ICs). New concepts are gradually introduced and illustrated. This book is especially well-suited for readers that intend to implement debug support in their own digital ICs. In particular:

Novice/student & teachers Learns/teaches about SOC design, verification, and validation trends and challenges, the need to address the debug problem for silicon SOC implementations and the fundamental problems in meeting the debug requirements in practice, existing options for supporting software and hardware debug by design, and illustrating and evaluating an approach, an on-chip hardware architecture, EDA tools, and an off-chip debugger software architecture to address these problems.

Practitioner Learns about concrete concepts, architectures, and implementations for supporting the debugging of SOC software and hardware by design. This includes detailed descriptions of the on-chip debug architecture, the required EDA tools, and the off-chip debugger software.

Expert Learns about a comprehensive debug approach for modern, silicon SOC implementations with its associated debug infrastructure, which is directly usable with different SOC implementations, and provides comprehensive details on hardware and software architecture implementation details and choices therein that usually are not published.

We could not have performed the work described in this book without the support of the management of Philips Research Laboratories and NXP Semiconductors. We have collaborated with many colleagues at Philips Research Laboratories and NXP Semiconductors, and thank them all for the support they have given over the years. We also gratefully acknowledge the standardization efforts by colleagues in the Philips Debug Action Group and Core Test Action Group, and the efforts on the proprietary InCiDE software. Many thanks also go to the design teams of the SOCs described in Chap. 8. They allowed the validation of our debug theories, described in this book, in practice on their SOCs. We also had the pleasure to work together with university employees and students on debug research as well. We thank them for their debug contributions.

Last but certainly not least, we express our deepest thanks to our families and friends for their many years of support and for making the journey that led to this book both possible and so much more enjoyable.

Eindhoven, March 2014

Bart Vermeulen
Kees Goossens

Contents

Acronyms

AHB	ARM high performance bus
APB	Advanced peripheral bus
API	Application programmer's interface
ATE	Automated test equipment
AXI	Advanced extensible interface
BPS	Bug Positioning System
BLoG	Bug Localization Graph
CCS	Clock control slice
CDC	Clock domain crossing
CDR	Capture-DR
CCB	Clock control block
CGU	Clock generation unit
CMOS	Complementary metal oxide semiconductor
COTS	Commercial, off-the-shelf
CPA	Co-Processor Array
CPU	Central processing unit
CRGU	Clock and reset generation unit
CSAR	Communication-centric, scan-based, abstraction-based, run/stop-based
CSARDE	CSAR debug environment
CTAG	Core test action group
DAG	Debug action group
DAP	Debug access port
DCD	Debug chain database
DCSI	Debug control and status interconnect
DEF	Design exchange format
DfD	Design-for-debug
DfT	Design-for-test
DLL	Delay-locked loop
DSL	domain specific language
DSP	Digital signal processor
DTD	Document type definition
DTL	Device transaction level

DVFS	Dynamic voltage and frequency scaling
ECU	Electronic control unit
EDA	Electronic design automation
EDI	Event distribution interconnect
EMMI	Emission microscopy
FIFO	First-in first-out
FPGA	Field-programmable gate array
FSM	Finite state machine
GALS	Globally-asynchronous locally-synchronous
GCD	Greatest common divisor
GCP	Global control processor
GPU	Graphics processing unit
GSP	General Stream Processor
GUI	Graphical user interface
HDL	Hardware description language
IC	Integrated circuit
I2C	Inter-integrated circuit
IFRA	Instruction footprint recording and analysis
InCiDE	Integrated circuit debug environment
I/O	Input/output
IP	Internet protocol
IRC	International Roadmap Committee
ITRS	International Technology Roadmap for Semiconductors
LCM	Least common multiple
LED	Light emitting diode
LFSR	Linear feedback shift register
LSB	Least significant bit
LVP	Laser voltage probing
MCD	Multi-core debug
MCP	Multi-channel port
MED	Multi-core embedded debug
MMIO	Memory mapped input/output
MPSOC	Multi-processor system on chip
MSB	Most significant bit
NI	Network interface
NoC	Network on chip
OCP	Open core protocol
OMAP	Open multimedia application platform
OSGi	Open services gateway initiative
OS	Operating system
PAR	Project authorization request
PICA	Picosecond imaging circuit analysis
PIO	Primary input/output
PLL	Phase-locked loop
PSC	Protocol-specific controller

PSI	Protocol-specific instrument
RAM	Random accessible memory
RFD	Reconfigurable Fabric Die
RGU	Reset generation unit
RISC	Reduced instruction set computer
ROM	Read-only memory
RTL	Register-transfer-level
SDRAM	Synchronous, dynamic random accessible memory
SDR	Shift-DR
SIM	Scan input multiplexer
SIMD	Single-instruction, multiple-data
SIR	Shift-IR
SOC	System on chip
SRAM	Synchronous, random accessible memory
STG	State transition graph
TAP	Test access port
TCB	Test control block
TCK	Test clock
Tcl	Tool control language
TCP	Transmission control protocol
TDI	Test data input
TDO	Test data output
TMS	Test mode select
TPR	Test point register
TRSTN	Test reset
TTM	Time-to-market
UART	Universal asynchronous receiver/transmitter
UDR	Update-DR
UIR	Update-IR
UI	User interface
URL	Uniform resource locator
USB	Universal serial bus
VGA	Video graphics array
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
XML	Extensible markup language

Part I
Introduction

Chapter 1
Introduction

Abstract We start this chapter in Sect. 1.2 by reviewing the trends, challenges, and methodologies used in the implementation, verification, and validation of SOCs. We describe a generic and commonly-used post-silicon debug process in Sect. 1.3, which helps debug engineers with the localization of the root cause of problems that occur post-silicon. In Sect. 1.4, we show however that this generic process has requirements that cannot easily be met post-silicon. We outline our proposed solution and the organization of this book in Sect. 1.5. In Sect. 1.6 we provide an overview of the contributions of this book with respect to the state of the art in post-silicon debug. We conclude this chapter with a summary in Sect. 1.7.

1.1 Overview

In our daily lives we are surrounded by a large number of electronic consumer products. Examples are desktop computers, game consoles, mobile phones, tablet computers, and televisions. Each product uses ICs to provide the required functionality to its user. These users have come to expect increasingly more functionality and higher performance from each product with each new generation. For example, a mobile phone forty years ago only allowed its user to make a phone call, while today's mobile phone is a true personal assistant. Its owner can use it to stay in touch with family, friends, and colleagues 24 h a day, to download and listen to music and watch movies from on-line stores, and to be guided to nearby shops and restaurants based on his or her spoken query and current location. This increased functionality typically needs to be provided by a small number of ICs or even a single IC, due to the size restrictions of these products. The ICs used in many of these products therefore integrate a complete system on a single piece of silicon. We call the resulting IC a *system chip* or system on chip (SOC).

We start this chapter in Sect. 1.2 by reviewing the trends, challenges, and methodologies used in the implementation, verification, and validation of SOCs. We also assess their impact on the time-to-market (TTM)[1] of a new consumer product. Industrial benchmarks show that despite the many advances in these methodologies over the past years, some faults still slip through the pre-silicon verification and validation

[1] The time-to-market is the time from the first concept of a new product to its market release.

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6_1,

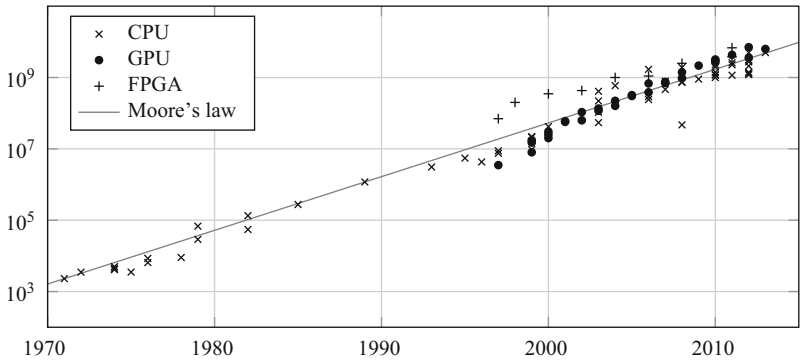


Fig. 1.1 Semiconductor process technology trends, based on [36]

processes and are only found after an SOC has been manufactured. At this stage, the root cause of any erroneous behavior has to be localized and removed from the silicon implementation of an SOC as quickly as possible, because it delays the introduction of the related product(s) into the market. This localization and removal process is called *post-silicon debug*. We describe a generic and commonly-used post-silicon debug process in Sect. 1.3, which helps debug engineers with the localization of these root causes. In Sect. 1.4, we show however that this generic process has requirements that cannot easily be met post-silicon. This is the problem that we address in this book. We outline our proposed solution and the organization of this book in Sect. 1.5. In Sect. 1.6 we provide an overview of the contributions of this book with respect to the state of the art in post-silicon debug. We conclude this chapter with a summary in Sect. 1.7.

1.2 System Chip Trends

1.2.1 Process Technologies

The high integration level of functions in SOCs has been made possible by the continued research and development in the semiconductor industry over the past four decades. During this period, this industry has been able to address the increasing consumer expectations in essence through the continued reduction in transistor feature size with each new semiconductor process generation. An SOC design team can integrate more transistors and hence implement more functionality on a single *die* [2] when the transistor feature size is smaller. To illustrate this trend, Fig. 1.1 shows the number of transistors that have been integrated on single dies to implement different

[2] A *die* is a single, unpackaged piece of silicon containing integrated circuits.

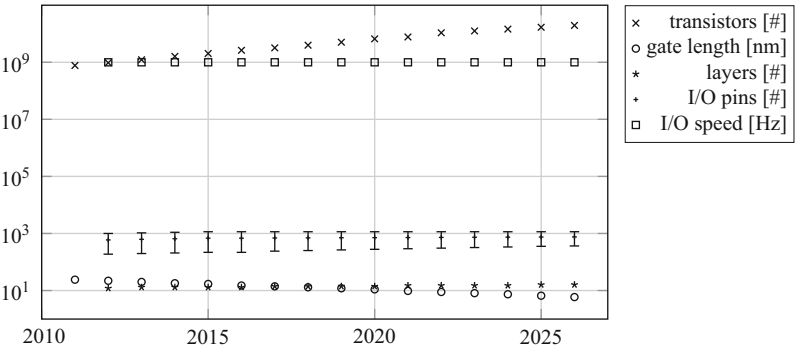


Fig. 1.2 ITRS roadmap predictions, based on [17]

ICs in the last four decades, to create central processing unit (CPUs), graphics processing unit (GPUs), and field-programmable gate arrays (FPGAs) [36]. The trend line shown in Fig. 1.1 depicts what is known in the industry as *Moore's law* [21]. This "law" states that the number of transistors that can be integrated on a die of the same size doubles roughly every 2 years. As can be seen, the semiconductor industry has managed to follow this trend successfully over the last four decades.

In addition to enabling higher integration levels, a smaller transistor also requires less power and time to change its state. These properties allow circuits in newer semiconductor processes to consume less dynamic power per transistor and to provide higher overall performance. When properly utilized, these technological benefits result in new consumer products with more functionality and enhanced performance, and where applicable, a longer battery life.

The International Roadmap Committee (IRC) predicts on its International Technology Roadmap for Semiconductors (ITRS) that the semiconductor industry will be able to continue to integrate an exponentially increasing number of transistors on a single die in the coming years [17]. A number of their predictions are shown in Fig. 1.2, including their predictions for the gate length of a transistor, the number of metal layers, and the number and maximum operating speed of the input/output (I/O) pins on mobile (SOCs) in the future. We discuss the consequences of these predictions later in this chapter.

The need to integrate more functionality on a single die makes the design of an (SOC) a complex task. Without countermeasures, the resulting increase in SOC complexity will in turn make it more difficult for SOC design teams to meet their TTM goal. A delay in bringing a fully-functional product to the market can however cause a substantial loss of revenue, market share, and customers, as shown in Fig. 1.3 [16]. Another recent survey [15] shows that the TTM is considered a key business differentiator by 68 % of the service providers interviewed. In the next section, we therefore investigate what methodologies SOC design teams are using to manage the increasing SOC complexity.

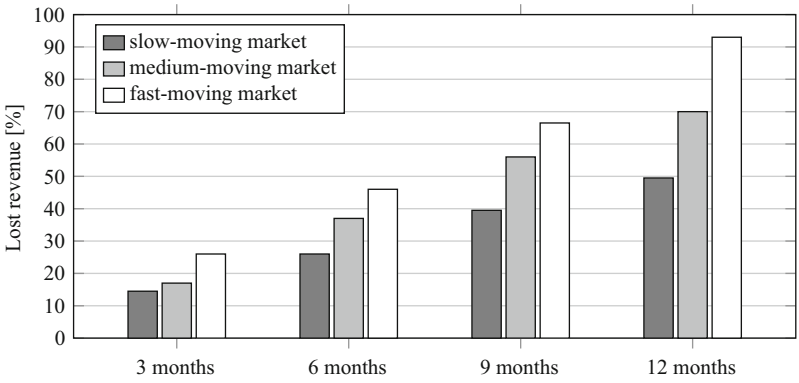


Fig. 1.3 Revenue lost by being late to the market, based on [16]

1.2.2 SOC Design Process

SOC design teams use a combination of four methodologies to be able to integrate the required number of customer functions in a new SOC within its required TTM. These four methodologies include using (1) an implementation refinement process, (2) a divide and conquer approach, (3) a reuse approach, and (4) increased programmability. We discuss these four methodologies in more detail below.

1.2.2.1 Implementation Refinement Process

The process to get to a new SOC silicon implementation starts from its *customer requirements specification*. This specification is an agreed, high-level description of the behavior of the SOC under specified operating conditions. SOC design teams subsequently use an *implementation refinement process* to obtain an implementation in silicon with the specified behavior. This process is illustrated in Fig. 1.4. In this book, we use the term *implementation* to refer to an executable instance of an SOC at a certain level of abstraction and the term *design* to refer to the activity of the SOC design team that leads to such an implementation. The SOC design team gradually adds more details, i.e., refines, the SOC implementation to effectively manage its complexity, while allowing its alternative implementations, i.e., its *design space*, to be efficiently explored within the customer and technology constraints.

An SOC design team uses *back-of-the-envelope calculations* and *estimation models* to efficiently explore the properties of different implementations. The team creates an *abstract executable implementation* of the SOC once its design space has been sufficiently explored and the key architectural decisions have been made. This executable implementation is refined in subsequent steps to *cycle-accurate implementations* at register-transfer-level (RTL) and gate-level. The creation of the *IC*

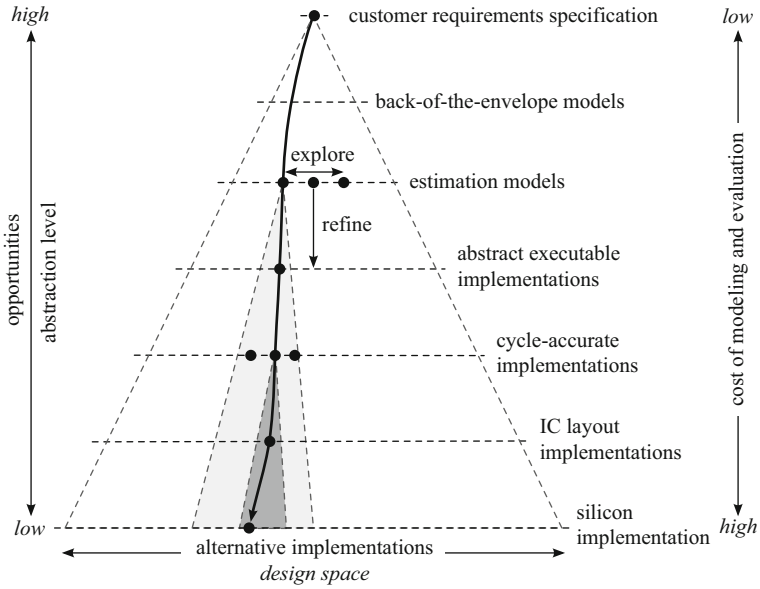


Fig. 1.4 Implementation refinement process, adapted from [19, 28]

layout implementation is the last step in the design process. This IC layout implementation closely resembles the actual silicon implementation of the SOC, both in behavior (e.g., in functionality, timing, power consumption, and signal interaction) as well as in appearance (e.g. in physical layout of the structures on a silicon die). The manufacturing process subsequently refines the IC layout implementation to a silicon implementation. Over the past decades, many steps in this implementation refinement process have been successfully automated with commercially-available EDA tools, thereby reducing the effort required from the SOC design team.

1.2.2.2 Divide And Conquer Approach

SOC design teams also use a *divide and conquer methodology*. They do this not only to manage the complexity of an SOC implementation, but also for scalability and performance reasons. In this methodology, the SOC implementation is built from smaller *building blocks* with standardized interfaces between them. These interfaces may allow for a loose coupling between the execution of the different building blocks of the same SOC. They may also allow the clock frequency and voltage supply of each block in the SOC to be individually regulated, to trade off its performance against its power consumption. This technique is called dynamic voltage and frequency scaling (DVFS), and the resulting hardware architecture is called GALS. A GALS hardware architecture relaxes the number of global constraints on the SOC implementation, thereby reducing its design effort.

In this book, we use *synchronous building blocks*, i.e., each building block has its own clock signal. Each building block furthermore operates at its own supply voltage. We refer to a set of closely-related building blocks that implement a specific set of functions as a *module*. An example of a commonly-used SOC module is a CPU with its L1 cache. Consequently, a module may use multiple clock signals and supply voltages. We come back to this in Chap. 3.

Modern on-chip interconnects, such as the advanced extensible interface (AXI) buses [3] and NoCs [8, 18, 23] support GALS operation by offering asynchronous interfaces between the modules and the interconnect. These interconnects are multi-threaded and support *pipelined transactions* between pairs of modules and *concurrent transactions* between different pairs of modules [13]. The SOC applications are implemented on top of the SOC hardware modules, as a number of concurrent computation threads. These threads communicate through the multi-threaded, on-chip interconnect.

1.2.2.3 Reuse Approach

SOC design teams reduce the design effort of a new SOC further by implementing the required SOC functionality through the integration of *pre-designed modules* according to a *pre-defined architecture template*. Example pre-designed modules are programmable processors, hardware accelerators, dedicated I/O peripherals, memory modules for on-chip data storage, and on-chip communication interconnects. Through this approach, the SOC teams leverage the experiences gained with these modules on earlier products that are based on the same, pre-defined architecture template, or *platform*. As described in [6], "a *platform* is a set of rules and guidelines of the hardware and software architecture, together with a suite of building blocks that fit into that architecture." Examples include the Ax (Apple), Exynos (Samsung), Nexperia (NXP), NovaThor (ST-Ericsson), open multimedia application platform (OMAP) (TI), SnapDragon (Qualcomm), and Tegra (Nvidia) platforms. Figure 1.5 shows the hardware details of NXP's PNX8550 SOC, which is based on the Nexperia platform [12]. The PNX8550 SOC hardware architecture integrates a MIPS32 CPU, two TriMedia digital signal processors (DSPs), a large number of I/O and co-processing units, and high-speed data and control buses to connect these units to the processors, to each other, and to the off-chip memory via the memory controller.

The software applications that run on the SOC hardware typically are also designed in a modular way to manage their complexity. Each application has a layered software architecture. Figure 1.6 shows the Linux-based software architecture that is used in the STB810 set-up box product as an example of this modularity [20]. This software architecture runs on the MIPS32 CPU that is shown in the top-left corner of Fig. 1.5 and controls the many audio and video hardware peripherals in the SOC.

The application programmer's interfaces (APIs) between the software components are standardized to decouple their implementation from their interface(s). This allows each component to be designed and tested in isolation. Standardized APIs furthermore allow reusing pre-designed software components in new SOC applications, thereby decreases their design effort.

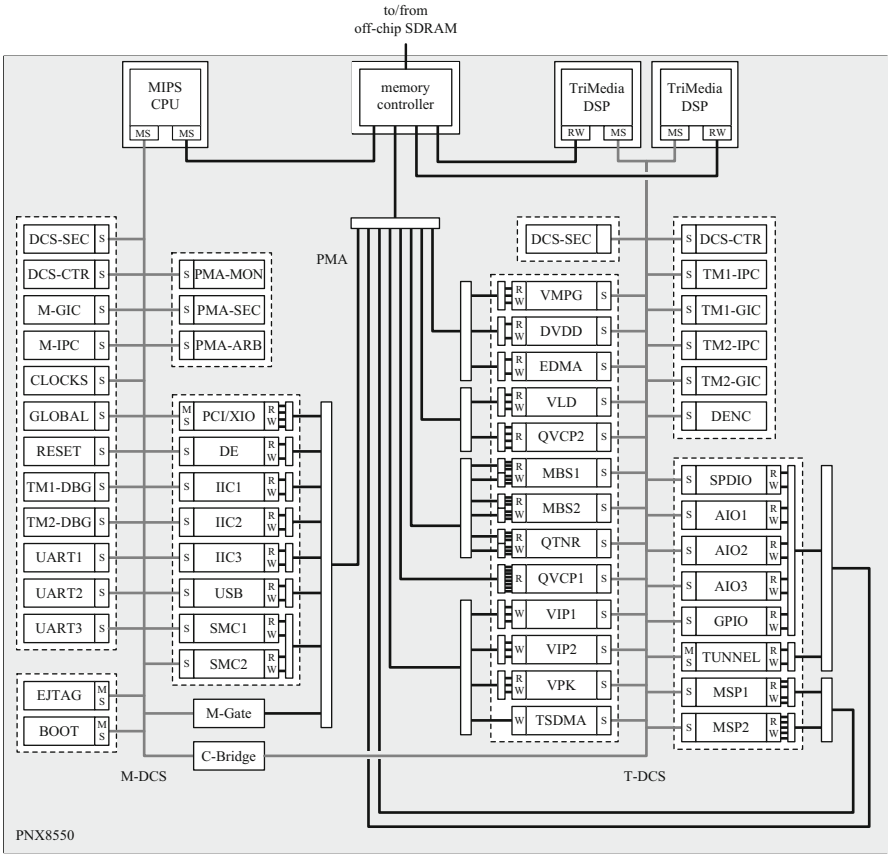


Fig. 1.5 PNX8550 hardware architecture, based on [12]

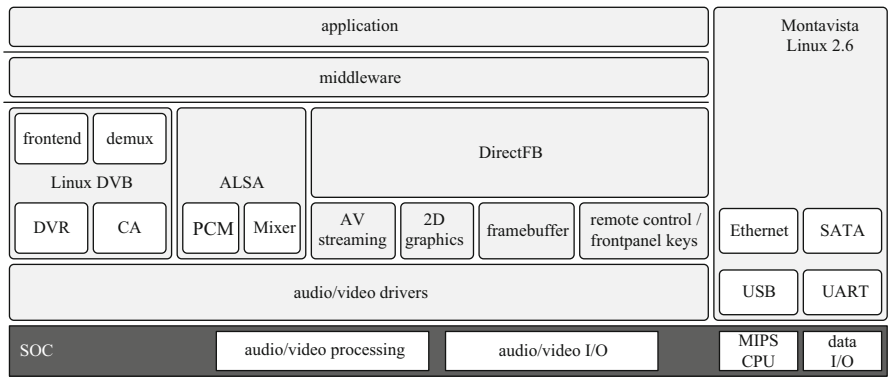


Fig. 1.6 STB810 software architecture for Linux, based on [20]

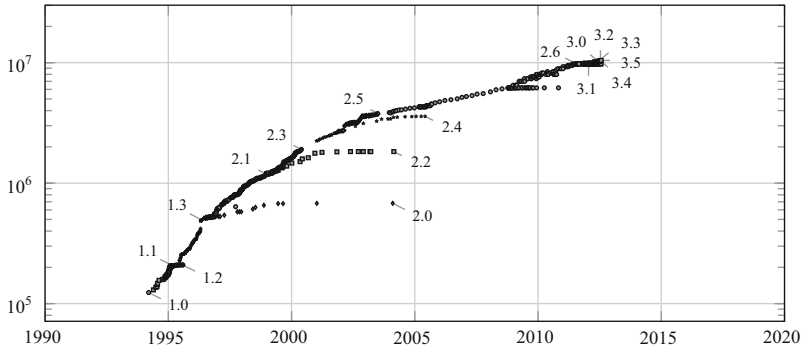


Fig. 1.7 Lines of source code in the Linux kernel over time

1.2.2.4 Increased Programmability

SOC teams have also increased the programmability of their SOC to further manage its hardware design effort. A recent survey shows that, of all ICs considered in 2012, 79 % of them had at least one embedded processor [9]. It furthermore showed that the average number of programmable processors in an SOC has increased over the last decade to 2.25 in 2012. This increased programmability allows software developers to (1) offer more functionality to the consumer by running increasingly more software on a single SOC, (2) target multiple products in the same domain using a single SOC, (3) defer design decisions, and (4) make late updates and bug fixes to a product, even after its release to the market. Figure 1.7 shows the increase over time in the number of source code lines[3] in the Linux kernel [4], which is just one of the basic components in the platform. Among others, the Android operating system (OS) from Google is based on this kernel and is used in the majority of mobile phones sold today. Figure 1.7 shows that the software complexity of the Linux kernel has significant increased in the past two decades.

1.2.3 SOC Verification and Validation

The IC layout implementation of every new SOC has to be checked for faults before it can be sent off to manufacturing. These checks provide the SOC design team with a certain level of confidence that the implementation is fault-free and that the resulting silicon implementation will consequently behave according to its customer requirements specification. We say that a *failure* occurs when the observed SOC behavior

[3] Figure 1.7 was obtained by counting the number of source code lines in the source code distributions of the Linux kernel, available at http://www.kernel.org, using the program `cloc` [7].

deviates from its specification. When such a failure occurs, the implementation contains a *fault*. A *fault* in an implementation is said to be initially *dormant*, until the point where it is *activated* and causes an *error*. A failure is the manifestation of an error to the customer.

Because of the SOC complexity, it is possible for the SOC design team or an EDA tool they use, to introduce a fault at any stage in the implementation refinement process. They therefore check the implementation after each refinement step for two properties. First, they check whether the (more refined) implementation still exhibits the same behavior as the implementation at the previous abstraction level. We refer to this process as *verification*. Commonly-used verification techniques include *formal verification*, *equivalency checking*, *(bi)simulation*, and *emulation*.

Second, they check whether the newly-added detail at the current abstraction level causes the implementation to behave as specified. As this detail is not present in the implementation at the previous abstraction level, they have to check the behavior of the refined implementation against its specification using other means, such as additional simulation and emulation, and static timing, IR-drop, and cross-talk analyses. We refer to this process as *validation*. These verification and validation processes are illustrated in Fig. 1.8.

The ability to exhaustively verify and validate the behavior of an SOC implementation before it is manufactured is however restricted by four factors. A first factor is the increased number of use cases that need to be verified for modern SOCs. This number increases as a consequence of the increased programmability, which allows an SOC module to perform many different functions, depending on the applications that run on the SOC.

A second factor is the increased parallelism in the SOC, resulting in multiple, possible execution traces even for a single use case. For example the advanced peripheral bus (APB) and ARM high performance bus (AHB) [2] are both single-threaded, and therefore only process one transaction at any point in time. All computation threads are effectively sequenced, resulting in one *global execution order* for the SOC. This execution order can be checked during verification and validation. With the loose coupling of the execution between building blocks, the communication and synchronization between the modules across the interconnect grow in complexity. Furthermore, modern on-chip interconnects are multi-threaded. The consequence is that the transactions between the different computation threads are no longer constrained to one particular global order. At best there is a *partial execution order* between the steps of the threads and transactions, imposed by the required interactions between the modules. We come back to this problem in Chap. 3. A recent survey shows that the run-time complexity in the communication and in the on-chip interconnect is becoming a cause of concern for SOC teams [5].

A third factor is the speed with which the behavior of an SOC implementation can be evaluated for one specific trace and one specific use case. This evaluation speed becomes lower as the SOC implementation becomes more refined, since it contains more details. Consequently it takes more time to evaluate the behavior of more refined implementations for a particular use case. The verification and validation techniques applied at the gate-level and IC layout level can for example take many hours, days,

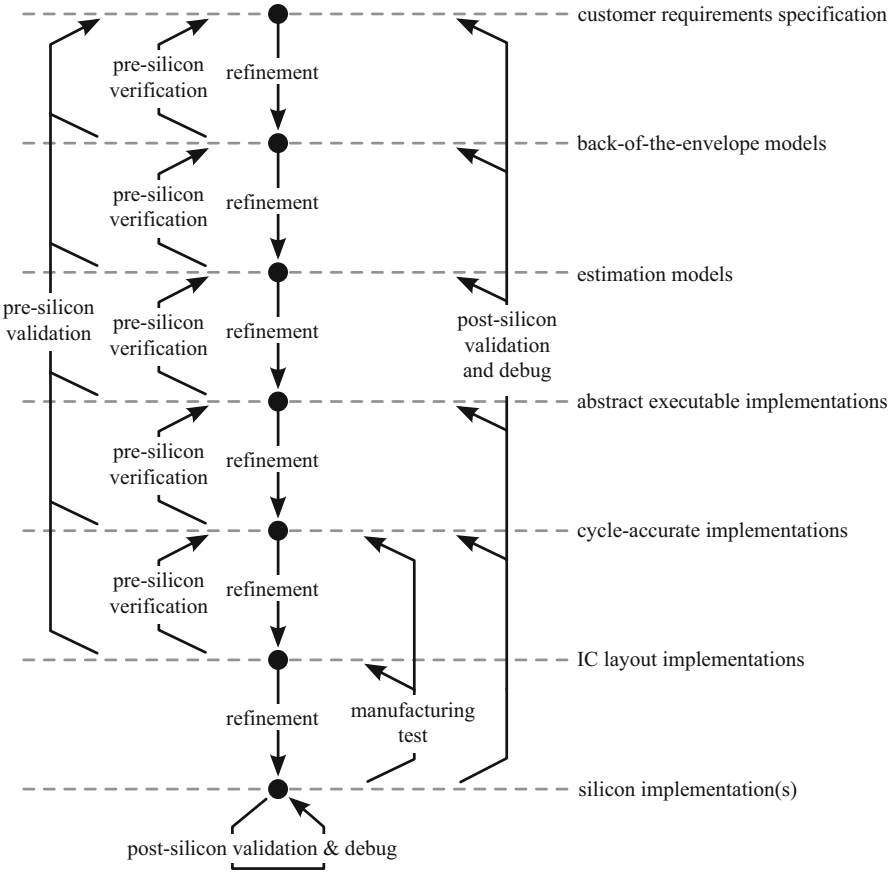


Fig. 1.8 Implementation refinement, pre-silicon verification, pre-silicon validation, manufacturing test, and post-silicon validation and debug processes

or even weeks to complete. An SOC design team uses *emulation techniques* to speed up the evaluation of an implementation by imitating its behavior using other SOCs. Although faster than simulation techniques, these techniques still tend to run one to several orders of magnitude slower than the final silicon implementation. To evaluate the behavior of an SOC implementation for all its use cases at the level of detail of a physical implementation can therefore take a very long time. The resulting TTM is unacceptable in a competitive market. Instead, an SOC design team has to select the most important use cases and only validate those.

A fourth factor are physical details that should be included and considered during verification and validation but are not, because they are not known in advance. Example details include specific manufacturing rules for a state-of-the-art process technology and any unpredictable environmental conditions under which the silicon implementation will be used.

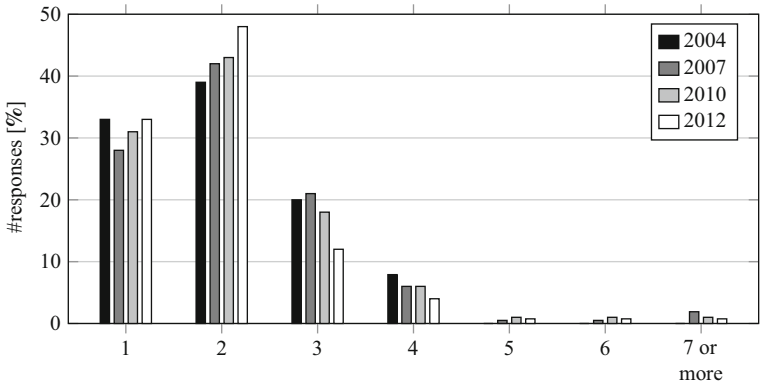


Fig. 1.9 Number of required silicon spins, based on [9]. Data used with permission of H. Foster (Mentor Graphics)

Because of these four factors, some implementation faults escape pre-silicon verification and validation and are only observed after manufacturing. A silicon implementation therefore has to be submitted to additional verification and validation. It is first tested for manufacturing faults. The *manufacturing test* process compares the behavior of the manufactured implementation with its IC layout implementation(s) and gate-level implementations. Advances in manufacturing test methodologies over the past decades allow complex silicon implementations to be screened effectively and efficiently for manufacturing faults. Based on a *fault diagnosis* of those silicon implementations that do not pass the manufacturing test, the set of manufacturing rules that is used in the pre-silicon validation can subsequently be extended to reduce the occurrence of manufacturing defects in future silicon implementations.

Manufacturing tests are effective at detecting structural faults in a silicon implementation. They do however not target functional implementation faults. A recent industry survey shows that only 33 % of the ICs considered in 2012 were determined to be *right-first-time*, i.e., contained no functional implementation faults, and that 19 % of the ICs required three or more spins[4] before their product(s) could be released to the market (refer to Fig. 1.9 [9]). Respins should be avoided as much as possible because of their associated cost. Some semiconductor companies however deliberately choose a first-time-functional/second-time-right strategy based on past experiences with escaped faults. These companies use the first silicon implementation to demonstrate key new functionality to customers, which these customers can subsequently use to build and test their own first product prototypes. Meanwhile, these first-time-functional implementations are used by the semiconductor company to find and remove any remaining faults that could not be (easily) found pre-silicon. The actual silicon implementation allows many use cases to be evaluated in a short

[4] A *silicon (re)spin* refers to a(nother) pass through (part of) the design process and the manufacturing process to create a new silicon implementation (based on a new design).

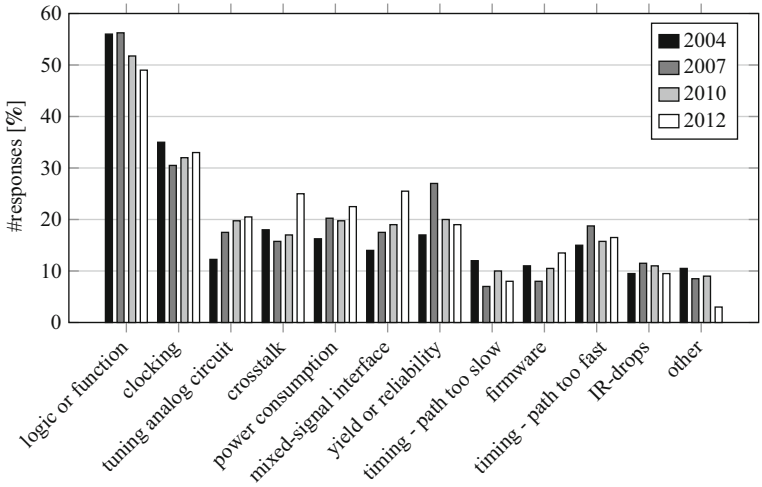


Fig. 1.10 Causes for needing multiple silicon spins, based on [9]. Data used with permission of H. Foster (Mentor Graphics)

amount of time. Its availability thereby significantly increases the chance that the silicon implementations from the second spin are fault-free.

The error categories that caused the respins in Fig. 1.9 are shown in Fig. 1.10 [9]. Note how the largest category is still for "logic and function errors". The size of this category has decreased over the past decade due to the advances made in SOC verification and validation techniques. However, 49 % of silicon spins are still caused, among others, by logic or functional flaws. In addition, we see increased contributions of clocking, timing analog circuit, crosstalk, power consumption, mixed-signal interfaces and firmware flaws. These results clearly show that the advances in process technology and design methodology enable SOC teams to implement SOCs that are more complex than we can currently fully verify and validate before manufacturing.

1.3 Post-Silicon SOC Debug

The post-silicon validation process may cause the SOC design team to observe a functional implementation fault in a silicon implementation. When this happens, the root cause of this fault has to be found and corrected as quickly as possible to allow the associated product(s) to be released to their markets. The additional process that is then used is called *post-silicon debug*. Industry benchmarks show that the post-silicon validation and debug processes combined can consume more than 35 % of the total project design time or cost [1, 22], due to, among others, the causes shown in Fig. 1.10. An SOC design team therefore needs effective and efficient post-silicon debug methods to find these escaped faults in a silicon implementation.

Table 1.1 A high-level overview of trace-based and run/stop-based debug approaches

Characteristic	Trace-based approach	Run/stop-based approach
Spatial scope	Subset of internal signals	All internal signals
Temporal scope	At multiple points during the execution	At the end of the execution
Hardware support	Trigger mechanism	Trigger mechanism
	On-chip memory	Execution stop mechanism
	Data output port	State access mechanism
	Configuration mechanism	Configuration mechanism
Main advantages	Real-time, internal observability	Full internal observability
		Full internal controllability
Disadvantages	Subset relatively very small	Non-real-time
	Silicon area cost	Silicon area cost

The post-silicon debug process involves observing and analyzing the erroneous behavior of a silicon implementation in the environment in which it fails. We refer to an execution of a silicon implementation in such an environment as a *debug experiment*. Its behavior is compared to the correct behavior of a *reference* under the same circumstances. The reference implementation may be another silicon implementation or may be taken from another abstraction level in the implementation refinement process. The goal is to *spatially* and *temporally* locate the root cause of the failure(s).

In the *ideal debug experiment*, we can observe the behavior of an implementation in every detail and at every point during its execution. We use this observability to quickly find the location inside the implementation and the earliest point in time at which the behavior of the implementation deviates from the behavior of the reference. However, in an *actual debug experiment* the means by which we can observe the behavior of a silicon implementation are both spatially and temporally severely restricted. This lack of observability is caused by the factors shown in Fig. 1.2.

A first factor are the six to seven orders of magnitude difference between the number of transistors that are integrated on a die and the maximum number of I/O pins, and their maximum operating speed that we can use to observe the behavior of these transistors. A second factor is the number of metal layers that shield these transistors from our observation through physical or optical probing techniques. We discuss these techniques in Sect. 9.1.1. A third factor is the fact that the transistors themselves are becoming too small to probe physically or optically (indicated by their physical gate length in Fig. 1.2). A silicon implementation is therefore intrinsically almost a black box for debugging.

SOC design teams use DfD techniques to help overcome these debug observation restrictions. These techniques increase the internal observability, and where necessary the controllability, of a silicon implementation. DfD techniques offer support for two common debug approaches: (1) *trace-based debug*, and (2) *run/stop-based debug*. Table 1.1 provides a high-level overview of both approaches.

Table 1.1 refers to the *scope* of a debug experiment. This scope includes two components: (1) a *spatial* component, which determines the set of internal signals that we observe, and (2) a *temporal* component, which determines when we observe

these internal signals. The scope determines the data volume that is observed and that has to be analyzed in a debug experiment. This debug data volume is larger when we choose to observe more signals and/or observe them at more points in time. Consider as an example an SOC implementation with 10 million transistors that runs at 100 MHz. This SOC produces 1 Pbits, i.e., 10^{15} bits, of data per second when we sample one signal for each transistor in each clock cycle.

It is therefore important to judiciously decide on the scope for each debug experiment we conduct, because our means to observe this debug data volume are restricted as previously explained. Keeping this data volume small also helps reduce the effort required to interpret the debug data after the experiment. The two debug approaches in Table 1.1 make a different trade-off between how much data to observe and how often to observe this data.

A *trace-based debug approach* involves either recording the values of a small subset of internal signals in an internal trace memory or streaming their values off-chip via a set of I/O pins, at interesting points during the execution of the SOC. Support for a trace-based debug approach is added to an implementation at design time by (1) selecting the (configurable) set of internal *trace signals*, (2) adding a (configurable) trigger mechanism that determines when these signals are sampled, and recorded or streamed, (3) adding on-chip *trace memory* to record and/or adding *trace output pins* to output signal values, and (4) adding a mechanism to configure this trace functionality. The main advantage of a trace-based debug approach is its ability to provide real-time information on the selected set of internal signals. A first drawback of this approach is that the subset of internal signals that can be observed in real-time is limited by the number of trace output pins that can be dedicated to stream data off-chip and their I/O speed. The amount of information we can stream off-chip is consequently relatively small compared to the total amount of information that is generated on-chip. A second drawback is the silicon area cost associated with its on-chip hardware components.

A *run/stop-based debug approach* involves stopping the execution of the SOC at an interesting point during its execution and extracting the value of all internal signals via a set of I/O pins afterwards. Support for a run/stop-based debug approach is added to an implementation at design time by (1) adding a (configurable) trigger mechanism that determines when the execution of the SOC is stopped, (2) adding a mechanism to stop the execution of the SOC, (3) adding a mechanism to access the state of the SOC, and (4) adding a mechanism to configure this debug functionality. The main advantage of this approach is its ability to provide observability of and control over all internal signals. A first drawback of this approach is however that this full observability and control is only available after the execution of the SOC has been stopped. A second drawback is the silicon area cost associated with its on-chip hardware components.

Figure 1.11 illustrates a post-silicon SOC debug setup with three different SOC environments, which can be used for both approaches. In a post-silicon debug setup, the silicon implementation of the SOC is embedded in a consumer product or in a product prototype (shown on the left-hand side). The silicon implementation was

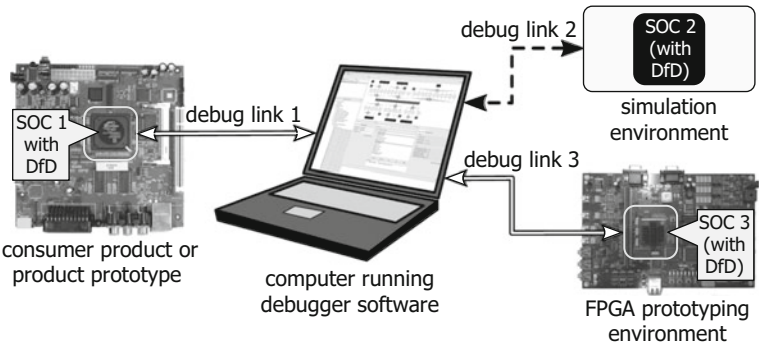


Fig. 1.11 Possible post-silicon SOC debug setups

extended at design time with DfD components to support either or both debug approaches. A computer is connected with a debug link 1 to a debug configuration and data port on this SOC. Debugger tools that run on the computer have to use the debug functionality inside the silicon implementation to provide the *debug engineer* that operates the computer with trace-based observability and/or run/stop-based observability and execution control over SOC 1.

The post-silicon debug infrastructure may additionally contain a reference SOC implementation inside a simulation environment or mapped on an FPGA prototyping environment (shown on the right-hand side). The same computer is connected respectively via a debug link 2 or 3 to these environments, to interact with the corresponding SOC implementation. Note that debug link 2 does not have to be a physical link, because the computer may host the simulation environment and may therefore have direct access. For the simulation and FPGA prototyping environments, the debugger tools that run on the computer may use the debug functionality of the environment, of any DfD embedded in the SOC, or both, to provide the debug engineer with internal observability of and execution control over respectively SOC 2 and 3.

The debug engineer subsequently uses the post-silicon debug process shown in Fig. 1.12 to debug the silicon implementation in this debug setup.

This process uses a silicon implementation, embedded in a setup as shown in Fig. 1.11, together with a reference, and comprises the following steps:

- Step 1: Start by applying power to the silicon implementation. Depending on whether the reference is a simulation model or a silicon implementation, respectively start its simulation environment or apply power to it.
- Step 2: Choose an initial abstraction level for the reference. For example, we can choose an abstract executable implementation, a cycle-accurate implementation, or an IC layout implementation. This has to be a judicious and conservative choice because the more detailed implementation we choose, the more internal detail we can observe, but the lower its evaluation speed will consequently be. The abstraction level of the reference may therefore be different than for the silicon implementation. We discuss the consequences of a possible difference in abstraction level in Sect. 2.2.3.

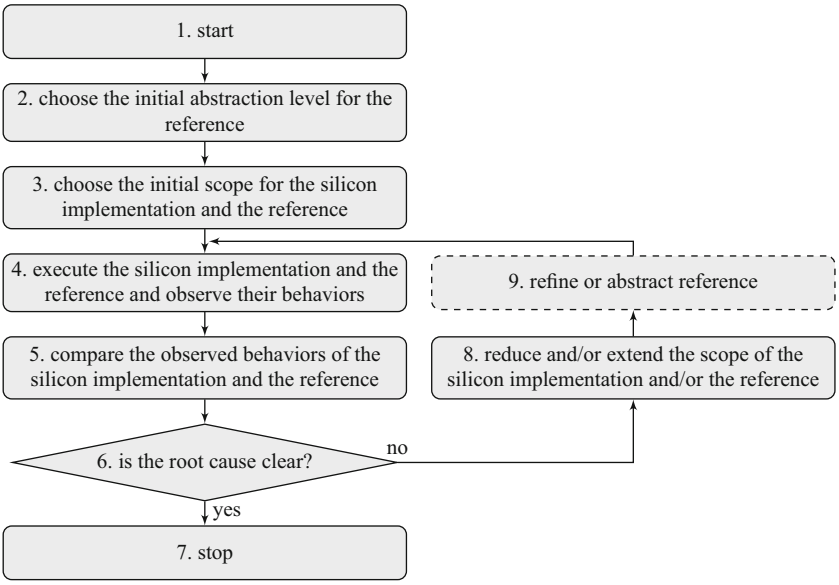


Fig. 1.12 Overview of the post-silicon debug process, adapted from [28]

- Step 3: Choose an initial temporal scope for the silicon implementation and the reference, i.e., decide from which point in the execution of the SOC the observation should start, at which point it should stop, and at which intermediate points its internal signals are observed. Also choose an initial spatial scope for the reference, because observing the full spatial scope of the reference for the selected temporal scope may take a very long time, if this observability is obtained using a simulation environment. The temporal scope is the same for the silicon implementation and the reference. The spatial scope however may be different and depends on the abstraction level of the reference chosen in Step 2. We discuss the consequences of a potential difference in spatial scope in Sect. 2.2.3.
- Step 4: Execute both implementations and observe their respective behaviors.
- Step 5: Compare the observed behavior of the silicon implementation with the behavior of the reference.
- Step 6: Decide whether the comparison performed in Step 5 clarifies the root cause of the failure. Proceed to Step 7 if it did, and to Step 8 if it did not.
- Step 7: Power-down the silicon implementation. Depending on whether the reference is a simulation model or a silicon implementation, respectively stop its simulation environment or power it down.
- Step 8: Based on the results of the comparison in Step 5, choose to reduce the temporal scope of both the silicon implementation and the reference, to extend the spatial scope of the reference, or a combination of both.
- Step 9: Based on the results of the comparison in Step 5, choose to optionally refine the reference to a lower level of abstraction, which may better explain the

behavior observed in the silicon implementation, or abstract the reference, which allows for a more efficient exploration of another time interval. Afterwards, go to Step 4.

This iterative process is repeated until the root cause of the error is clear, i.e., spatially and temporally localized. Overall, we see that this post-silicon debug process imposes the following five requirements on the debug infrastructure.

DR-1: Internal Observability *The debug engineer has to be able to observe the internal signals of the silicon implementation and of the reference implementation.*

DR-2: Execution Control *The debug engineer has to be able to control the execution of the silicon implementation and of the reference implementation to reproduce the faulty behavior.*

DR-3: Non-intrusive Debug *The functional behavior of the silicon implementation and of the reference implementation should not be changed by the fact that they are being debugged.*

DR-4: Efficient Implementation and Use *Debug support has to be implemented efficiently and easy to use to keep the SOC cost low.*

DR-5: Reference Abstraction Level *The debug engineer has to be able to change the abstraction level of the reference.*

Debug requirement DR-1 states both the need for the debug engineer to be able to select which internal signals to observe during a debug experiment (i.e., to set the spatial scope) and the need to make their values observable from the outside of the SOC. Debug requirement DR-2 states the need to be able to reproduce the faulty behavior to localize its root cause. Spatial localization is enabled by satisfying debug requirement DR-1 and temporal localization is enabled by satisfying debug requirement DR-2. Debug requirement DR-3 states the need that the application of the debug process should not remove the original failure or introduce additional failures. Debug requirement DR-4 states that the cost of providing the debug functionality should not cause the SOC to become non-competitive. The implementation cost covers the non-recurring engineering cost of the implementation, the silicon area cost per SOC, as well as the cost for (adapting) the debugger software that runs on the computer connected to the SOC environments. Additionally, this requirement states that the debug support should be easy to use by debug engineers. Debug requirement DR-5 states the need to be able to evaluate the behavior of the reference at different abstraction levels, to allow a judicious trade-off between the level of detail in the reference and its evaluation time. The pre-silicon verification and validation infrastructure, together with the pre-silicon implementations we obtained as part of the implementation refinement process described in Sect. 1.2.2, can typically be used to meet these debug requirements for as far as the (pre-silicon) references are concerned. We therefore do not address these debug requirements for the reference further in this book.

1.4 Problem Statement

In this book, we address the problem of debugging a faulty silicon implementation using the post-silicon debug process in Fig. 1.12. As described in the previous section, this process imposes debug requirements DR-1 to DR-4 on the infrastructure to debug a silicon implementation. These requirements cannot be easily met due to the observability restrictions described in Sect. 1.3. Therefore, we specifically address in this book the following five problems:

1. How do we select the internal signals that are to be observed during a debug experiment?
2. How do we observe the large number of internal signals using the limited number of I/O pins?
3. How do we reproduce faulty behavior of a silicon implementation?
4. How do we use this debug functionality without removing the original failure or introducing new failures?
5. How do we implement all required debug functionality efficiently?
6. How do we combine the solutions to these related problems into one coherent debug approach and infrastructure?

We limit the scope of these problems to the debugging of digital SOC implementation, because modern SOCs are predominantly digital [9].

1.5 Proposed Approach and Book Organization

Figure 1.13 shows a high-level overview of the proposed approach and the organization of this book. We perform an in-depth analysis on why debugging silicon implementations is difficult in Part II. We first perform this analysis for debugging a single building block in Chap. 2, and subsequently for debugging multiple, interacting building blocks in Chap. 3. In particular, we revisit debug requirements DR-1 to DR-4 for the post-silicon debug process, and identify nine factors that complicate meeting these four requirements.

In Part III we propose our CSAR debug approach and associated on-chip and off-chip infrastructure. In Chap. 4, we present a high-level overview of our approach. In this approach, we extend the post-silicon debug process in Fig. 1.12 and define 25 debug infrastructure requirements to mitigate or remove the effects of the identified complicating factors. The required debug infrastructure consists of (1) a generic, on-chip debug architecture, (2) a configurable, automated DfD flow to be used during the design process of an SOC, and (3) customizable, off-chip debugger software. We present our generic on-chip debug architecture and details on its components in Chap. 5. We describe how these components can be automatically instantiated in an SOC implementation at design time under user control in Chap. 6. Information on the instantiated debug components and their functionality is stored in a DfD configuration file. Our off-chip debugger software, called CSAR debug environment

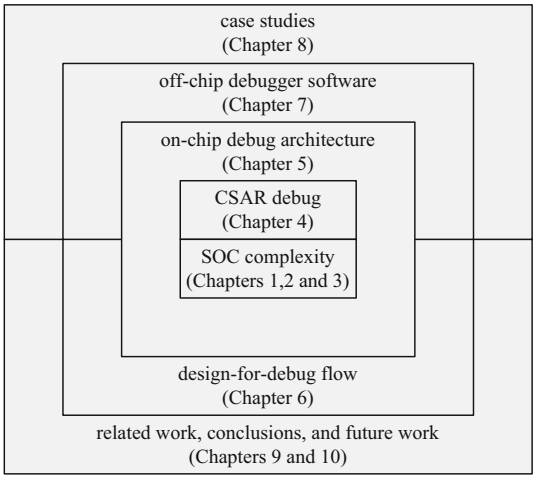


Fig. 1.13 Overview of the organization of this book

(CSARDE), is described in Chap. 7. This software uses this information to interact with the on-chip debug architecture in different environments. This software helps to address the identified nine factors that complicate debugging, by allowing the silicon implementation to be debugged using reference implementations at multiple abstraction levels.

In Part IV, we evaluate in Chap. 8 the effectiveness and efficiency of our CSAR debug approach and its supporting infrastructure. For this, we provide an overview of, and evaluate, six industrial SOCs that include in their silicon implementation parts of our proposed approach and infrastructure. We also use an illustrative, example SOC implementation and different types of errors. We furthermore quantify the hardware cost and design effort to support our approach.

In Part V, we discuss related work in Chap. 9 and present conclusions and future work in Chap. 10.

1.6 Book Contributions

In this book we make the following contributions to the state-of-the-art in post-silicon SOC debug:

- We capture the commonly-used post-silicon debug practices in one generic post-silicon debug process and derive its four key debug requirements [28].
- We identify nine factors that complicate the debugging of a silicon SOC implementation using the debug process in Fig. 1.12 [11, 28, 29, 32].

- We propose a *communication-centric, scan-based, abstraction-based, run/stop-based (CSAR) debug approach* to address the requirements and the complicating factors of this debug process [13, 28, 29, 35]. In this approach,
  - we use protocol-specific instruments (PSIs) to allow us to control the handshake signals used in on-chip communication protocols,
  - we use this communication-centric control to create a partial or a total transaction order in the SOC,
  - we use the stall states in the SOC building blocks to create locally-consistent states.
  - we combine the use of the control over the handshake signals and the stall states to create globally-consistent states,
  - we use a fast and scalable event distribution interconnect (EDI) to approximate the instantaneous stopping of the SOC,
  - we use the scan chains to extract the locally and globally consistent SOC state via an IEEE Std. 1149.1 test access port (TAP),
  - we introduce the guided replay process at the communication level between building blocks, and
  - we use structural, data, behavioral, and temporal abstraction techniques to the scan chain contents to abstract all the way up to the level of the application with multiple distributed masters and slaves.
- We provide a customizable, scalable, and layout-friendly, on-chip debug architecture to support our CSAR debug approach. This debug architecture provides control over the on-chip communication and allows the extraction of globally-consistent state information from the SOC via an IEEE Std. 1149.1 test access port (TAP) in different SOC environments [10, 24–26, 31, 33].
- We automate the inclusion of a customized CSAR debug architecture in an SOC implementation at design time, using a configurable DfD flow [14].
- We provide configurable, off-chip debugger software to analyze and compare the state information extracted from the SOC with one or more of its references, in order to help localize the root cause of a failure [14, 34].
- We demonstrate the applicability of this work by evaluating our CSAR debug approach using six industrial SOC designs and an illustrative, example SOC implementation [13, 14, 27–30, 35].

1.7 Summary

In this chapter, we observed that the SOCs that are used at the heart of consumer products are becoming increasingly complex over time. We subsequently reviewed the trends, challenges, and methodologies used in SOC design, verification, and validation. We observed that it is difficult for SOC design teams to ensure that the first silicon implementation of an SOC contains no faults and behaves correctly in the product. Industry benchmarks provide evidence that implementation faults unfortunately do slip through pre-silicon verification and validation and end up in the

silicon implementation. As a consequence, these faults have to be localized using, and removed from, the actual silicon with an effective and efficient debug process. We have also shown that debugging a silicon implementation is very difficult due to the intrinsic restrictions on its internal observability and controllability. We therefore propose in this book a communication-centric, scan-based, abstraction-based, run/stop-based (CSAR) debug approach, and support it by providing a customizable, scalable, and layout-friendly, on-chip debug architecture with debug components, a configurable and automated DfD flow, and configurable off-chip debugger software.

References

1. Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. Design Automation Conference*, pages 7–12, New York, NY, USA, 2006. Association for Computing Machinery, Inc.
2. ARM. AMBA specification. rev. 2. 0, 1999.
3. ARM Limited. *AMBA AXI Protocol Specification*, June 2003.
4. Daniel P. Bovet and Marco Cesati. *Understanding the linux kernel*. O'Reilly, November 2005.
5. Jack Browne. 318 engineers surveyed on top core frequencies and noc use issues, November 2012.
6. Theo A. C. M. Claasen. System on a chip: Changing ic design today and in the future. *IEEE* Micro, 23(3):20–26, May 2003.
7. A. Danial. Cloc - count lines of code, 2012.
8. Giovanni De Micheli and Luca Benini, editors. *Networks on Chips: Technology and Tools*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann Publishers Inc., July 2006.
9. Harry Foster. Wilson research group and mentor graphics, 2012 functional verification study. blog, 2013.
10. Jeroen Geuzebroek and Bart Vermeulen. Integration of Hardware Assertions in Systems-on-Chip. In *Proc. IEEE International Test Conference*, 2008.
11. Sandeep Kumar Goel and Bart Vermeulen. Data Invalidation Analysis for Scan-Based Debug on Multiple-Clock System Chips. *Journal of Electronic Testing: Theory and Applications*, 19(4):407–416, 2003.
12. Sandeep Kumar Goel, Kuoshu Chiu, Erik Jan Marinissen, Toan Nguyen, and Steven Oostdijk. Test infrastructure design for the nexperia home platform pnx8550 system chip. In *Proc. Design, Automation, and Test in Europe conference*, 2004.
13. K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek.. In *Proc. International Symposium on Networks on Chip*, pages 95–106, 5 2007.
14. 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*, 2009.
15. Hadas Haran. What's stopping you from improving your time to market?, April 2011.
16. IBM. Revenue lost by being late to market, 2006.
17. International Roadmap Committee. The international technology roadmap for semiconductors (itrs), 2012.
18. Axel Jantsch and Hannu Tenhunen, editors. *Networks on Chip*. Kluwer Academic Publishers, 2003.
19. A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, 1999.
20. Jerry Kuo. *IP-STB introduction*. NXP Semiconductors, 2007.

21. G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8): 114–117, April 1965.
22. Amir Nahir, Avi Ziv, Rajesh Galivanche, Alan Hu, Miron Abramovici, Albert Camilleri, Bob Bentley, Harry Foster, Valeria Bertacco, and Shakti Kapoor. Bridging pre-silicon verification and post-silicon validation. In *Proc. Design Automation Conference*, 2010.
23. Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures*. Systems on Silicon. Morgan Kaufmann Publishers Inc., 2008.
24. G. J. Van Rootselaar and B. Vermeulen. Silicon debug: scan chains alone are not enough. In *Proc. IEEE International Test Conference*, pages 892–902, 1999.
25. B. Vermeulen and G.J. van Rootselaar. Silicon debug of a co-processor array for video applications. In *Proc. High-Level Design Validation and Test Workshop*, pages 47–52, 2000.
26. B. Vermeulen and S.K. Goel. Design for debug: catching design errors in digital chips. *Design Test of Computers, IEEE*, 19(3):35–43, 5 2002.
27. Bart Vermeulen and Kees Goossens. A Network-on-Chip Monitoring Infrastructure for Communication-centric Debug of Embedded Multi-Processor SoCs. In *Proc. International Symposium on VLSI Design, Automation and Test*, 2009.
28. Bart Vermeulen and Kees Goossens. Debugging Multi-Core Systems on Chip. In George Kornaros, editor, *Multi-Core Embedded Systems*, chapter 5, pages 153–198. CRC Press/Taylor & Francis Group, 2010.
29. Bart Vermeulen and Kees Goossens. Obtaining consistent global state dumps to interactively debug systems on chip with multiple clocks. In *Proc. High-Level Design Validation and Test Workshop*, 6 2010.
30. Bart Vermeulen and Kees Goossens. Interactive debugging of systems on chip with multiple clocks. *IEEE* Design and Test of Computers, 5 2011. Special issue on Transaction-Level Validation of Multicore Architectures.
31. B. Vermeulen, T. Waayers, and S.K. Goel. Core-based scan architecture for silicon debug. In *Proc. IEEE International Test Conference*, pages 638–647, 2002.
32. Bart Vermeulen, John Dielissen, Kees Goossens, and Călin Ciordaş. Bringing Communication Networks On Chip: Test and Verification Implications. *IEEE Communications Magazine*, 41(9):74–81, 9 2003.
33. Bart Vermeulen, Zalfany Urfianto, and Sandeep Kumar Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proc. Design Automation Conference*, pages 514–517, San Diego, CA, USA, 6 2004.
34. Bart Vermeulen, Yu-Chin Hsu, and Robert Ruiz. Silicon Debug. *Test and Measurement World Magazine*, pages 41–45, 10 2006.
35. B. Vermeulen, K. Goossens, and S. Umrani. Debugging distributed-shared-memory communication at multiple granularities in networks on chip. In *Proc. International Symposium on Networks on Chip*, pages 3–12, 4 2008.
36. Wikipedia. Transistor count, 2012.

Part II
The Complexity of Debugging System Chips

Chapter 2
Post-silicon Debugging of a Single Building Block

Abstract In this chapter, we analyze the factors that complicate the post-silicon debugging of a single SOC building block. In Sect. 2.1, we first introduce a formal finite state machine (FSM) description, to capture the cycle-accurate behavior of a single building block. To debug the silicon implementation of the corresponding building block, we can subsequently use the debug process described in Sect. 1.3 to compare its behavior to the behavior that is captured by this description. In doing so, we identify however six factors in Sect. 2.2 that complicate (the application of) this process. We conclude this chapter with a summary in Sect. 2.3.

2.1 Behavior of a Single Building Block

2.1.1 Formal Definitions

Design teams commonly use an FSM description to specify the cycle-accurate behavior of an SOC building block [2, 3]. A block diagram of an FSM is shown in Fig. 2.1. The FSM in Fig. 2.1 has an input I, an internal state s, and an output O. The input I takes a value from the set $\mathcal{I}$ with a possible input symbols. The state takes its value from the set $\mathcal{S}$ with b possible states. The output O takes its value from the set $\mathcal{O}$ with c possible output symbols. The current state s of the FSM is stored in a register, which is triggered by the input clock signal "clk". The content of this register is set to the output of the combinational logic block δ, whenever a rising edge occurs on this clock signal. We indicate this *synchronous behavior* with a triangular symbol on the clock input of all registers in this book. We refer to the rising edges on the clock signal that cause this update of the FSM state as the *active edges* for these registers. A silicon implementation of this FSM stores its state in *state elements*, such as *flip-flops* and *random accessible memory (RAM)*. The combinational logic block λ determines the new value for the output O, based on the current state s and optionally the momentary value of the input I. We formally define our FSM as a 6-tuple $\mathcal{M} = \{\mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \delta, \lambda\}$, in which:

- $\mathcal{S}$ is the *set of possible states*; $\mathcal{S} = \{s^0, \ldots, s^{b-1}\}$.
- s^0 is the *initial state*; $s^0 \in \mathcal{S}$.
- $\mathcal{I}$ is the *input alphabet*, i.e., the set of input symbols; $\mathcal{I} = \{i^0, \ldots, i^{a-1}\}$.
- $\mathcal{O}$ is the *output alphabet*, i.e., the set of output symbols; $\mathcal{O} = \{o^0, \ldots, o^{c-1}\}$.

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip*, Embedded Systems, DOI 10.1007/978-3-319-06242-6_2,

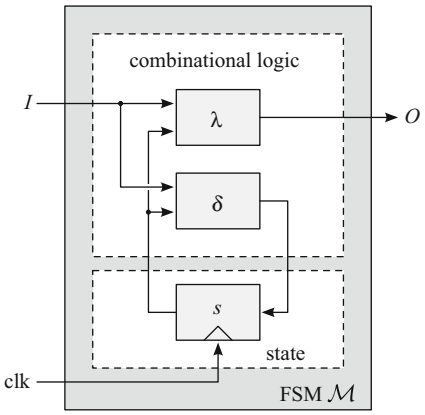


Fig. 2.1 Example reference for a building block

Table 2.1 Finite state machine specification

s	I	$\delta(s, I)$	$\lambda(s, I)$
s^0	0	s^0	0
s^0	1	s^1	0
s^1	0	s^0	0
s^1	1	s^2	0
s^2	0	s^0	0
s^2	1	s^3	1
s^3	0	s^0	1
s^3	1	s^0	1

- Function δ is the *FSM next-state function* that specifies the next state of the FSM, based on the current state s and the current value of the input I; $\delta : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S}$. Note that the definition of this function makes our FSM *deterministic*, as this function is a *total function*, i.e., defined for all its arguments, and maps every possible input combination to exactly one next state value.
- Function λ is the *FSM output function* that specifies the new value for the output O of the FSM, based on the current state s and optionally the current value of the input I; $\lambda : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{O}$. Note that this function can represent the output behavior of both Mealy [2] and Moore FSMs [3], depending on whether the input signals respectively directly influence the outputs or only through a change in the internal state.

Table 2.1 shows the details of an example FSM. This FSM has four internal states, i.e., $\mathcal{S} = \{s^0, s^1, s^2, s^3\}$, an input alphabet $\mathcal{I} = \{0, 1\}$, and an output alphabet $\mathcal{O} = \{0, 1\}$. The FSM next-state function δ and FSM output function λ are specified in Table 2.1. While the input symbol "1" is applied to the FSM's input, this FSM transitions through these four states on successive active clock edges. The FSM transitions from any state to its initial state on the next active clock edge when the input symbol "0" is applied.

An FSM has an associated *state transition graph (STG)* $\mathcal{G}$ (refer to Fig. 2.2 for our example FSM). An STG is a labeled and directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where

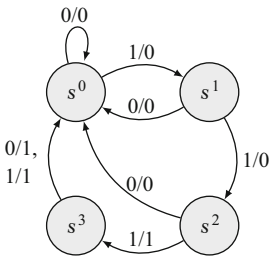


Fig. 2.2 Corresponding STG of the FSM in Table 2.1

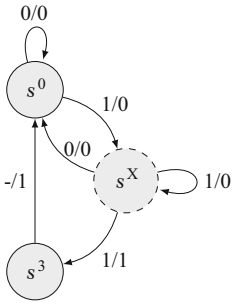


Fig. 2.3 Example FSM with a super state s^X

each vertex $v \in \mathcal{V}$ corresponds to a state $s = f_s(v)$, $f_s : \mathcal{V} \rightarrow \mathcal{S}$, and vice versa, $v = f_v(s)$, $f_v : \mathcal{S} \rightarrow \mathcal{V}$. Each edge $e_{ij} = (v, w)$, $e_{ij} \in \mathcal{E}$ and $v, w \in \mathcal{V}$, corresponds to a transition from state $s^i = f_s(v)$ to state $s^j = f_s(w)$. This edge e_{ij} has a set of labels $L_{ij} = L(e_{ij})$, $L : \mathcal{V}^2 \rightarrow 2^{\mathcal{I} \times \mathcal{O}}$. Each label $l \in L_{ij}$ is a pair of symbols (x, y), $x \in \mathcal{I}$ and $y \in \mathcal{O}$. The input symbol x causes a transition from state s^i to state s^j, i.e., $s^j = \delta(s^i, x)$. The output symbol y corresponds to the resulting new value for the output O, i.e., $y = \lambda(s^i, x)$ [1].

In this book, we use a *super state* s^X as a compact notation for a set of FSM states. We use this notation later in this book to prevent having to draw very large and complex STGs. When we say that an FSM is in a super state s^X, we mean that its current state s is a member of the super state s^X. Our notation for a super state in an STG is a dashed circle. We only show the transitions between the super state and the states of the FSM that are not included in the super state. We do not draw the transitions between the states in the super state. As an example, the super state s^X shown in Fig. 2.3 is a compact notation for the states s^1 and s^2 that are shown in Fig. 2.2. Figure 2.3 is a behaviorally-equivalent abstraction of Fig. 2.2. Please note further that although multiple edges in the STG in Fig. 2.3 share the same input symbol, this does not indicate that the corresponding FSM is *non-deterministic*. For these transitions, both the input symbol of the FSM and the specific state inside each super-state determine which edge is taken.

We furthermore define a *stall state* using Eq. 2.1.

$$s \in \mathcal{S} \text{ is a stall state} \Leftrightarrow \exists x \in \mathcal{I}.\delta(s, x) = s \tag{2.1}$$

A stall state is a state of the FSMs, where the execution of the FSM can be temporarily suspended under external control. A stall state has a *self edge* in the associated STG. For example, s^0 in Fig. 2.2 is a stall state. *Stalling* an FSM means setting its inputs to the condition on one of its self edges, thereby forcing it to stay in the associated state once it has reached this state. We discuss stall states and their use in the communication between building blocks in Chap. 3.

We furthermore define a *reset input symbol* using Eq. 2.2.

$$x \in \mathcal{I} \text{ is a reset input symbol} \Leftrightarrow \forall s \in \mathcal{S}.\delta(s, x) = s^0 \tag{2.2}$$

A reset input symbol is an input symbol of the FSM, which, when it is applied to the input of the FSM, causes the state of the FSM to be unconditionally set to the initial state s^0, on the next active edge of the clock. The input symbol "0" in our example FSM is a reset input symbol, as the FSM transitions from every state to the initial state s^0 on the next active edge of the clock, whenever this symbol is applied to the input of the FSM.

2.1.2 Execution Behavior

We can obtain the expected behavior for a building block from its reference FSM $\mathcal{M}$, when we are given an input function $x(t) : \mathbb{R}_0^+ \rightarrow \mathcal{I}$, and a clock period $T \in \mathbb{R}^+$ and a clock phase $\phi \in \mathbb{R}_0^+$ for the input clock signal. The *clock period* is defined as the amount of time between two adjacent active edges on the clock signal. The *clock phase* is defined as the amount of time between a reference moment in time $t = 0$ and the first active edge on the clock signal. The n^{th} active edge on a clock signal is placed at a time $t_{ae}(n), n \in \mathbb{N}^+$, which is given by Eq. 2.3.

$$t_{ae}(n) = (n-1)T + \phi, n \in \mathbb{N}^+ \tag{2.3}$$

Note that this function places the first active edge at $t = \phi$. We define the *current state function* $s(t)$ of an FSM $\mathcal{M}$ in Eq. 2.4, $s(t) : \mathbb{R}_0^+ \rightarrow \mathcal{S}$.

$$s(t) = \begin{cases} s^0 & \text{for } 0 \leq t \leq \phi \\ \delta(s(u(t)), x(u(t))) & \text{for } t > \phi \end{cases} \tag{2.4}$$

The initial state of the FSM is assumed to be s^0 up to and including the moment in time of the first active edge ($t = \phi$). We examine the validity of this assumption in Sect. 2.2.2. To define the state $s(t)$ for $t > \phi$, we use the time-quantization function $u(t) : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ to determine the location in time of the closest preceding active edge. The function $u(t)$ is defined in Eq. 2.5 and shown graphically in Fig. 2.4.

$$u(t) = \begin{cases} 0 & \text{for } 0 \leq t \leq \phi \\ \left(\lceil \frac{t-\phi}{T} \rceil - 1\right) T + \phi & \text{for } t > \phi \end{cases} \tag{2.5}$$

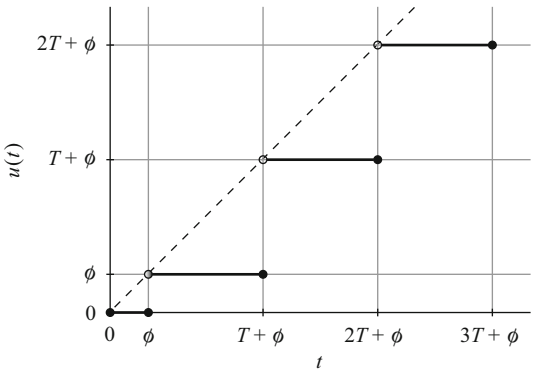


Fig. 2.4 The time-quantization function $u(t)$

The dashed line in Fig. 2.4 shows the function $v(t) = t$. Note how the function $u(t)$ stays below the function $v(t)$ and yields the time of the closest preceding active edge for each time t. To calculate the next state value at each active edge, the current state of the FSM is combined with the momentary values on the input I of the FSM at the time of the last active edge, given by $x(u(t))$. These two values are subsequently used as the input parameters for the FSM next-state function δ to obtain the new state. This new state is valid for the period $(u(t) + T, u(t) + 2\ T]$. We define the *output function* $y(t) : \mathbb{R}_0^+ \rightarrow \mathcal{O}$ of FSM $\mathcal{M}$ using Eq. 2.6.

$$y(t) = \lambda(s(t), x(t)) \tag{2.6}$$

Equation 2.6 uses the FSM output function λ to specify the new output value based on the current state of the FSM at time t, defined by $s(t)$, and the momentary value on the input of the FSM I at time t, given by $x(t)$.

Note how the functions $s(t)$ and $u(t)$ associate respectively a current state and an output value with the FSM for each absolute point in time $t \in \mathbb{R}_0^+$.

2.2 Complicating Factors for Debugging

The FSM description, formalized in Sect. 2.1.1, can be used as a reference while debugging its silicon implementation using the debug process described in Sect. 1.3. In this section we identify six factors that complicate (the application of) this process.

2.2.1 Limited Observability and Controllability

The high integration level of modern SOCs results in a very large state space. The state of the SOC also changes very rapidly over time. The amount of functional data that is therefore generated inside an SOC is multiple orders of magnitude larger than

the amount of data that can be stored on-chip or streamed off-chip in real-time. This severely limits the spatial scope of a debug experiment.

Furthermore, the state of a building block can only be controlled via its inputs. A debug engineer may apply a reset input symbol to start the execution of a building block from its initial state s^0. It may be difficult, if not impossible, to change the temporal scope between debug experiments and start the execution of a building block from any other state. It may therefore be difficult to control the clock signal and inputs of a silicon implementation of a building block and observe its state and outputs using only off-chip debug instruments.

The use of DfD techniques can improve this debug observability and controllability. The cost of the silicon area associated with these debug instruments has to be carefully weighed against their potential benefits to keep the resulting SOC competitive. As it is not clear in advance what specific debug functionality is the best investment to be able to find the root cause of post-silicon failures, it is best to utilize a generic DfD technique that maximizes the internal observability at an acceptable cost. Based on these observations, we define complicating factors CF-1, CF-2, and CF-3 for debugging.

CF-1: Limited Spatial Observability and Controllability *The intrinsic observability and controllability of a silicon implementation for debugging is limited by the number of input and output pins it has and their maximum operating speed, and is significantly smaller than the amount of data that is generated internally.*

CF-2: Limited Temporal Controllability and Observability *It may be difficult, if not impossible, to start the execution of a silicon implementation from a state other than its initial state s^0, and observe its internal state at every point in time during its execution.*

CF-3: Implementation Cost *An on-chip debug architecture to improve the debug controllability and observability of a silicon implementation costs implementation effort and silicon area.*

2.2.2 Undefined Substate and Outputs

The assumption that the start-up state of an implementation of a building block is the initial state s^0 (refer to Eq. 2.4) is not true for all implementations of a building block. In a silicon implementation of a building block, its state is stored in flip-flops and embedded RAMs. A single flip-flop in a silicon implementation settles to a random start-up state when its power supply is turned on. The same holds for the start-up state of individual bit-cells in a RAM. As such, when the same silicon implementation is powered up multiple times, this very likely results in different initial states. The reason why these implementations still function correctly despite their different initial states, is because they are designed to respond to the application of a reset input symbol to their input. Most digital ICs have a dedicated reset input

for this purpose. Asserting this reset input forces the state of the building block to its initial state s^0. Comparing the behavior of two implementations before the assertion of their reset inputs can lead to the incorrect attribution of a difference in their states or outputs to a fault in the silicon implementation, while this difference is in fact caused by a valid difference in start-up state. In order to accurately compare the states, it is therefore important to first assert the reset input of both implementations at the start of each debug experiment. This should force both implementations to their initial state s^0, after which their behaviors can be checked for errors.

Embedded RAM modules however do not have an input to reset their state. Instead, the designer of a building block that uses an internal RAM module has to guarantee by design that the assertion of the reset input of the building block, or the module this building block is a part of, initializes enough internal state and outputs to make it comparable to the state and outputs of another implementation of that building block. This means that any uninitialized substate, including the state of the embedded RAM module, is not allowed to affect the execution behavior of the implementation. Failure to do so may result in undefined behavior. Similarly, the application or non-application of a certain control sequence to the inputs of the building block may lead to the state of a building block to become undefined. For example, the state of a synchronous, dynamic random accessible memory (SDRAM) module becomes undefined when its content is not refreshed frequently enough. Based on these observations, we define complicating factor CF-4 for debugging.

CF-4: Undefined Substate and Outputs *(A part of) the state and outputs of a (silicon) implementation of a building block may be undefined in certain intervals during its execution.*

Whether a part of the state of an implementation is undefined or not can only be known to the debug engineer if another part of its state has been designed to indicate this. We can subsequently use this indicator to determine the undefined part of the state and outputs, and not use this part in the state comparison with another implementation. We discuss how to do this next.

2.2.3 State Comparison

The abstraction level of the silicon implementation and a reference for a building block may not be the same. In those cases, it is necessary to use *abstraction functions* to bridge this difference and be able to compare their states and outputs. These functions have to translate (semantically interpret) the state and output values at the abstraction level of the silicon implementation to the corresponding data values at the abstraction level of the reference. These functions depend on three aspects: (1) the structural transformations that have taken place in the implementation refinement process going from the abstraction level of the reference to the abstraction level of the silicon implementation, (2) the difference in data types used at the abstraction levels of the reference and silicon implementation, and (3) the behavior of the building block.

Consider, as an example, a building block with a reference FSM $\mathcal{M}$, which has eight states, i.e., $|\mathcal{S}_{\mathcal{M}}| = 8$. Its silicon implementation needs at least a three-bit state register to represent an element in this set. During debug, we need a state abstraction function, $f_{s,abstract} : \mathbb{B}^3 \rightarrow \mathcal{S}_{\mathcal{M}}$, that translates the state of the silicon implementation to a state that can be compared to the state of the reference FSM. We call an abstraction function *consistent*, when it is a *total function*. We subsequently call a state s of the silicon implementation a *consistent state*, when it is possible to translate this state to a corresponding state of the reference, i.e., when $f_{s,abstract}(s)$ is defined for this state. We call a state s of the silicon implementation *inconsistent* when this function is not defined for the state s.

Consider, for an example of an inconsistent state, another building block with a reference FSM $\mathcal{M}'$, which has five states, i.e., $|\mathcal{S}_{\mathcal{M}'}| = 5$. The silicon implementation of this building block still needs at least a three-bit state register to represents an element in this set. However, not all possible values of this three-bit register may represent a valid state in the reference FSM, i.e., the associated abstraction function $f'_{s,abstract} : \mathbb{B}^3 \rightarrow \mathcal{S}_{\mathcal{M}}$ may not be a *consistent function*.

Note that we similarly need an output abstraction function, $f_{o,abstract} : \mathbb{B}^x \rightarrow \mathcal{O}$, $x \in \mathbb{N}^+$ for the value on the x binary outputs of a silicon implementation, and can similarly define a *consistent output* and an *inconsistent output* of the silicon implementation.

Not being able to translate and compare the state and outputs of a silicon implementation with a reference complicates the debug process. We therefore investigated possible causes for inconsistent states during debug and found that certain SOC architectures may cause this. We discuss this in more detail in Chap. 3. Based on these observations, we define complicating factor CF-5 for debugging.

CF-5: State Comparison *The state and outputs of a (silicon) implementation of a building block may need to be translated to allow a comparison to the state and outputs of a reference.*

2.2.4 Transient Errors

Some errors may not cause a failure when they are activated, or the failure may not be visible long enough to be observed, because our temporal observability is limited (refer to complicating factor CF-2). We call an error *permanent* when, after its activation, its effects remain observable in the state or in the output values of the building block. Otherwise, the error is called *transient*. This is for example the case, when an erroneous state of a building block is corrected before the effects of the error can propagate to the output of the building block.

The problem with transient errors is that we cannot state with certainty whether a fault is present in the silicon implementation or not, when we observe that its state and outputs match those of the reference at a certain point in its execution. An error may become dormant after its activation, and before it or its effects are

observed in the debug process of Fig. 1.12. We therefore incorrectly change the temporal scope to search for the error later in time instead of earlier. We can only make a correct change in the scope when we actually observe that the state and/or the output(s) of the silicon implementation differ from those of the reference. In the presence of permanent faults, we can therefore apply more advanced and faster search algorithms, e.g., a binary search algorithm, to reduce the temporal scope than we can in the presence of transient faults. Based on these observations, we define complicating factor CF-6 for debugging.

CF-6: Transient Errors *Errors in a (silicon) implementation may be transient.*

2.3 Summary

In this chapter, we introduced a formal FSM description as a cycle-accurate reference for the behavior of a silicon implementation of a single SOC building block. We subsequently analyzed the difficulty in debugging the behavior of this silicon implementation using this reference. We identified six complicating factors that relate to the limited observability and controllability of a silicon implementation, possibly-undefined substate of an implementation after start-up and at other times during its execution, the possibility of inconsistent states, and the occurrence of transient errors.

A good silicon debug approach should (1) address the intrinsic limitations to the spatial and temporal observability and controllability of the SOC state, (2) address the spatial and temporal controllability of the SOC execution, (3) provide state and output abstraction functions to help bridge any differences in abstraction level between the SOC implementation and its reference, and filter out undefined (sub)state, and (4) do so at a low implementation cost. In particular, good temporal observability of the SOC state is required to debug transient errors.

We introduce a debug approach that meets these requirements in Chap. 4. Before then, we first investigate the factors that complicate the debugging of multiple, interacting building blocks in the next chapter, because SOC are composed of multiple building blocks.

References

1. John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Publishing Co., 1979.
2. George H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.
3. Edward F Moore. Gedanken-experiments on sequential machines. 34:129–153, 1956.

Chapter 3
Post-silicon Debugging of Multiple Building Blocks

Abstract In this chapter we extend the analysis that we started in Chap. 2, of the factors that complicate the debugging of a silicon SOC implementation, by looking at the post-silicon debugging of multiple, interacting building blocks. We first analyze in Sect. 3.1 the interaction between pairs of building blocks in detail. We describe the design steps that are taken by SOC design teams to ensure that data transfers between two interacting building blocks occur without problems. We look in Sect. 3.2 at the arbitration process needed to share the access to a building block between multiple building blocks, and evaluate its impact on the execution of the SOC. In Sect. 3.3, we subsequently identify three additional factors that complicate the application of the post-silicon debug process of Sect. 1.3 to an SOC with multiple, interacting building blocks. We conclude this chapter with a summary in Sect. 3.4.

3.1 Communication Between Two Building Blocks

3.1.1 Overview

An SOC building block may need to interact with other building blocks to provide a certain function. Figure 3.1 shows two building blocks *A* and *B* that interact with each other through a shared data communication link. Figure 3.1 only shows the clock inputs of, and the signals between, the two building blocks. The internal structure of each building block is no longer shown for clarity. The cycle-accurate behavior of each block can be captured in an FSM definition, as described in Sect. 2.1. For the correct operation of these two blocks together, it is important that the data transfers between these two blocks occur without problems. To find out what can go wrong, we first need to look at how data is actually sampled and stored in a building block. The data in a building block is stored as part of its state. In a silicon implementation of a building block, this state is stored in flip-flops and/or embedded RAM. Figure 3.2 shows the input and output signals of a flip-flop FF, comprising a data input "d", a clock input "clk", and a data output "q". Figure 3.3 shows a timing diagram for these signals. A RAM behaves similarly with respects to its inputs and outputs.

The flip-flop transfers the value on its data input to its data output at each *active edge* on its clock input signal. We refer to this action as *data sampling*. For a correct operation of the flip-flop, the signal on its data input should not be changed in the

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip*, Embedded Systems, DOI 10.1007/978-3-319-06242-6_3,

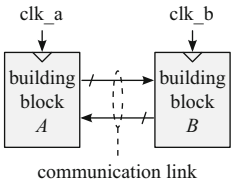


Fig. 3.1 Two interacting building blocks

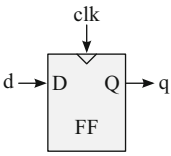


Fig. 3.2 Flip-flop block diagram

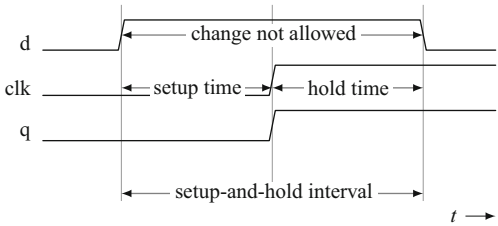


Fig. 3.3 Flip-flop timing diagram, based on [4]

setup-and-hold interval around each active edge of its clock signal. The duration Δt_{sh} of this interval is defined by the *setup time* t_s before, and the *hold time* t_h after, active edges on the clock signal, i.e., $\Delta t_{sh} = t_s + t_h$. These two values are properties of the flip-flop and vary with temperature, voltage, and process. These two values furthermore constrain the *minimum clock period* T_{min} for the flip-flop.

A designer has to make sure that there is always a minimum amount of time between a transition in the data signal and each active clock edge, as specified by the setup and hold times. Figure 3.4 shows an example circuit to explain what can happen when the signal on the data input of a flip-flop is changed too close to an active edge on its clock signal. Flip-flop FF_A in Fig. 3.4 is clocked by a clock signal "clk_a" and flip-flop FF_B by a clock signal "clk_b". In Fig. 3.5, the signal on the data input of flip-flop FF_B is inverted a time interval Δt_1 before an active edge on the clock signal "clk_b". We have reduced the steepness of the slopes on clock signal "clk_b" for illustration purposes. When the input signal is inverted inside the setup-and-hold interval of a flip-flop, the output signal "q_b" is not guaranteed to reach its inverted value by the end of the clock edge ($t = t_1$). As a result, the data output signal "q_b" of flip-flop FF_B is left at a value between logic-0 and logic-1. In this example, the value of the data output signal "q_b" is regenerated to its new value of logic-1 in the remainder of the clock period to a logic-1 value at $t = t_2$. Provided that this regeneration occurs quickly, this still gives the combinational logic behind

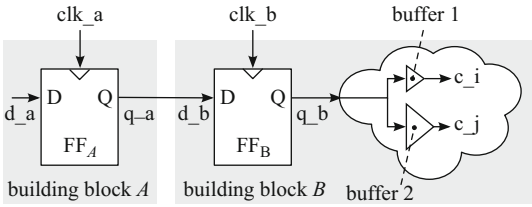


Fig. 3.4 Example circuit to illustrate meta-stability in a flip-flop

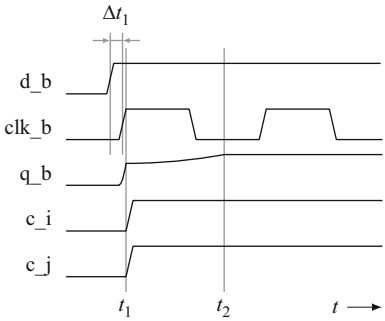


Fig. 3.5 First example of meta-stability in a flip-flop

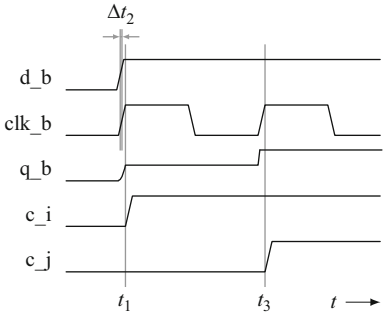


Fig. 3.6 Second example of meta-stability in a flip-flop

flip-flop FF$_B$ enough time to correctly evaluate this logic level before the next active edge on the clock signal "clk_b".

The time interval Δt_2 in Fig. 3.6 is however much shorter than the time interval Δt_1. The output signal "q_b" now only just reaches an exactly balanced level between the logic-0 and logic-1 values when the clock edge finishes at $t = t_1$. This signal state is called a *meta-stable* state, because the signal will eventually settle to either a logic-0 or a logic-1 value under the influence of internal noise and other disturbances. The amount of time taken to do so is however indeterminate and can therefore be very long [4]. In the example in Fig. 3.6, this signal is not regenerated to a logic-1 value in the remainder of the clock period. Consequently, the two buffers in building

Table 3.1 Classification of the relation between two clock signals [9]

Classification	Period relation	Phase relation
Synchronous	Same	Same
Mesochronous	Same	Constant over time
Plesiochronous	Small difference	Slowly varying over time
Periodic	Different	Periodic variation
Asynchronous	Not applicable	Arbitrary

block B do not see a logic-1 value on their input. They instead see this meta-stable value. A meta-stable value can cause problems in a digital circuit, because either the logic gates may interpret the meta-stable value as the old value, which may be incorrect, or the logic gates may interpret this meta-stable value as two different values. A difference in interpretation causes a *logical inconsistency*, which may lead to an *inconsistent state* or *inconsistent output values*, and therefore undefined and potentially-harmful behavior for building block B and the SOC. As identified in Sect. 2.2.3, inconsistent states and/or output values also complicate debugging.

The meta-stability on signal "q_b" is eventually resolved in Fig. 3.6 at $t = t_3$, because the input value of the flip-flop is held stable long enough. This resolution may however take one or more clock cycles [4]. In the timing diagram of Fig. 3.6, it took one additional clock cycle for the value on the output of flip-flop FF_B to become a stable value.

Now that we know how a building block samples data on its inputs into its state elements, we can examine what happens when a building block receives data from another building block. To determine whether meta-stability can occur in the communication between these two building blocks, we need to examine the relation between their clock signals. Table 3.1 summarizes five possible relations between two clock signals [9].

These five relations are:

1. *Synchronous*: The two clock signals are identical, i.e., they have the exact same period and their phase difference is zero. An example is a design that uses two outputs of a balanced clock tree.
2. *Mesochronous*: The two clock signals have the same period, but a constant and known phase difference over time. An example is a design with two clocks that have been shifted in phase using a delay-locked loop (DLL).
3. *Plesiochronous*: The clock periods of the two clock signals are not identical, but are nominally the same. This is for example the case for clock signals derived from two crystal oscillators that are marked with the same clock frequency[1]. Because the two crystals are physically not identical, the generated clock signals are not identical either. The phase difference between these two clock signals varies slowly over time.
4. *Periodic*: The clock periods of the two clock signals are fractions of the period of a certain master clock. Their phase difference is a function of time. An example

[1] The clock frequency is the reciprocal of the clock period.

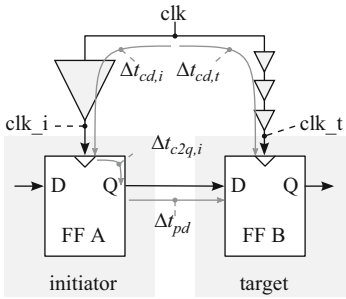


Fig. 3.7 Synchronous communication between two flip-flops in different building blocks

is a design that uses frequency multipliers and dividers, together with one clock source, to derive two or more clocks for its building blocks. Data transfers between these building blocks can take place on the active edges of another clock. This clock signal has a period that is an integer multiple of the least common multiple (LCM) of the clock periods of the two clock signals.

5. *Asynchronous*: The two clock signals have no fixed or known period or phase relation. This is for example the case for two clock signals from two unknown clock sources.

At design time, a designer has to select an appropriate *communication technique* between two building blocks based on the relation between their two clock signals. This technique ensures by design that the data transfers between the two building blocks occur without meta-stability problems. In our review of these techniques below, we examine unidirectional data communication links, for which we distinguish the *initiator* that sends the data and the *target* that receives this data. Bidirectional data communication uses two unidirectional data communication links. Below we also only focus on the synchronous and asynchronous relations.We show in Sect. 3.1.2 that we can treat the mesochronous relation as a synchronous relation by design. We can furthermore treat the plesiochronous and periodic relations as asynchronous relations.

3.1.2 Synchronous Communication

Figure 3.7 shows an example of *synchronous communication* between two flip-flops in different building blocks. On an active edge on its clock signal "clk_i", the initiator sets its output signal to the data value that needs to be transferred. This active edge arrives at the clock input of flip-flop A from its clock source after a clock distribution delay $\Delta t_{cd,i}$. This data subsequently arrives at the data input of the target after a propagation delay, which comprises the delay from the clock input of flip-flop A to its output ($\Delta t_{c2q,i}$) and the *data propagation delay* over the data wire from flip-flop A in the initiator to flip-flop B in the target (Δt_{pd}). The target samples the data on the

active edges of the clock signal "clk_t". These active edges arrive at the clock input of flip-flop B from its clock source after a clock distribution delay $\Delta t_{cd,t}$. The correct sampling of the data by the target is guaranteed by the careful design of the on-chip clock distribution and the data paths between the building blocks, thereby ensuring that for each data path Eqs. 3.1 and 3.2 hold under all temperature, voltage, and process variations (refer to Fig. 3.7).

$$\Delta t_{cd,i} + \Delta t_{c2q,i} + \Delta t_{pd} < \Delta t_{cd,t} + T - t_s \tag{3.1}$$

$$\Delta t_{cd,i} + \Delta t_{c2q,i} + \Delta t_{pd} > \Delta t_{cd,t} + t_h \tag{3.2}$$

Equation 3.1 constraints the propagation delays to prevent that a change in the data on the output of the initiator arrives at a flip-flop in the target within the setup time before an active edge on the target clock. Equation 3.2 prevents a change within the hold time after an active edge on the target clock. Combining these two equations, we obtain the constraints expressed in Eq. 3.3 between the phase difference $\Delta t_{cd,i} - \Delta t_{cd,t}$ between the two clock signals "clk_i" and "clk_t".

$$t_h - \Delta t_{c2q,i} - \Delta t_{pd} < \Delta t_{cd,i} - \Delta t_{cd,t} < T - t_s - \Delta t_{c2q,i} - \Delta t_{pd} \tag{3.3}$$

We refer to all sequential logic gates, whose clock and data signal delays have been designed to meet the constraint in Eq. 3.3, and all combinational logic gates between them, as a *clock domain*. A clock domain can be treated as a single building block for debugging (refer to Chap. 2). Note that Eq. 3.3 may not require that the phase difference is zero. Consequently, Eq. 3.3 can also be used to ensure that mesochronous building blocks can communicate with each other without meta-stability problems.

The progress in semiconductor technology in the last decades has however made the use of this *synchronous design style* more complicated. Long wires are now slower than logic, i.e., the *minimum clock period* T_{min} of flip-flops has been reduced faster than the on-chip *data propagation delay* Δt_{pd}. As a consequence, it is no longer possible in modern process technologies to cross the larger part of a silicon die within this minimum clock period. This problem can be solved by adding registers to the SOC design to partition long paths into a set of stages, where each stage satisfies Eq. 3.3. This so-called *pipelining* however adds a deterministic latency of a number of clock cycles to the communication of each data element. This reduces the performance and silicon area advantages of using a process technology with smaller geometries. Furthermore, guaranteeing that the constraint in Eq. 3.3 holds under all process variations and all operating conditions for all interacting flip-flop pairs becomes more difficult without increasing the clock period T. Increasing the clock period however again reduces the performance advantage of a process technology with smaller geometries. Consequently it is no longer possible nor desired to have an SOC that is 100 % synchronous, with all its modules in a single clock domain.

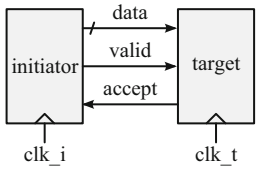


Fig. 3.8 Example circuit with a data handshake

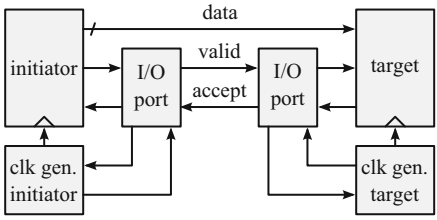


Fig. 3.9 Example circuit with pausible clocks

3.1.3 Asynchronous Communication

SOC designers have consequently adopted the *GALS design style* [11] to address the limitations of the synchronous design style described above. This design style assumes by default that the clock relation between interacting building blocks is *asynchronous*. A *handshake protocol* is used to guarantee the correct transfer of data between two building blocks. Two advantages of the GALS design style over the synchronous design style are that (1) it imposes fewer timing constraints on the overall SOC design, and hence is more scalable, and (2) each building block can run at its own best frequency and voltage operating point from a power consumption/performance trade-off point of view. Later in this book, we however see three disadvantages of this design style. These are (1) that the throughput of asynchronous transfers is lower than for synchronous transfers, (2) that the latency to communicate across a clock domain boundary is non-deterministic at the clock cycle level, and (3) that additional control logic and signal wires are needed. Nevertheless, the GALS design style offers better possibilities to take advantage of newer process technologies. Figures 3.8 and 3.9 shows two implementations that use a handshake protocol.

The technique in Fig. 3.8 uses a data handshake protocol to prevent a change in the data signal near the sampling edges of the target clock. The implementation shown in Fig. 3.9 uses *pausible clocks* to prevent an active edge on the target clock signal when the data signal on its input is changed [11]. In both implementations, metastability problems are avoided when the data signals are sampled. The data signals from the initiator are in both cases accompanied by a *valid* control signal from the initiator to the target and an *accept* control signal from the target to the initiator. This is the case for a *push handshake*. In a *pull handshake*, the data direction is reversed, and the valid and accept signals take the role of respectively a *request* signal and an *acknowledge* signal. In this book, we describe and use push handshakes, but pull handshakes work similarly.

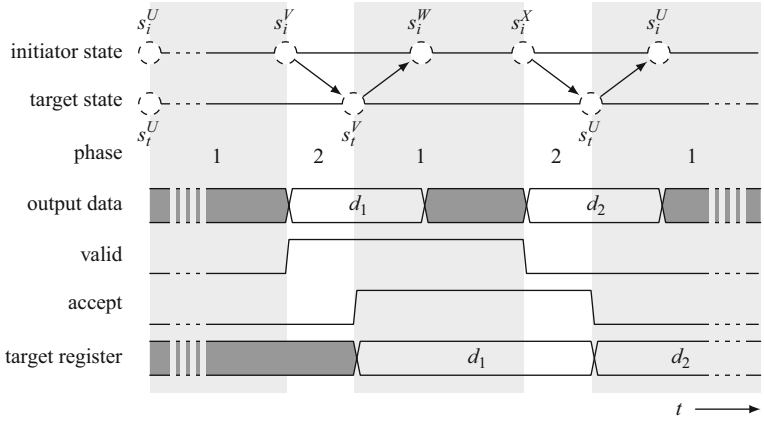


Fig. 3.10 Timing diagram of a two-phase handshake protocol

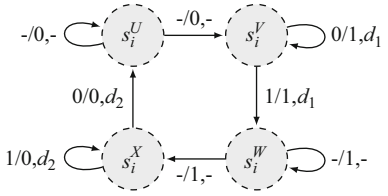


Fig. 3.11 Initiator STG for a two-phase handshake protocol

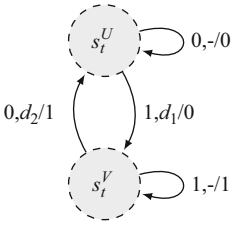


Fig. 3.12 Target STG for a two-phase handshake protocol

The *handshake protocol* is performed using the two control signals and has either two or four phases. For our explanation of these two handshake protocols below, we use the implementation in Fig. 3.8 and *Moore FSMs* [10] on both sides. The protocols works similar for the implementation in Fig. 3.9 or *Mealy FSMs* [7].

The *two-phase handshake protocol* is illustrated using Figs. 3.10, 3.11, and 3.12, with respectively a timing diagram, an STG for the initiator, and an STG for the target. The edge labels in the initiator's STG denote the "accept/valid,data" signal combination, while the edge labels in the target's STG denote the "valid,data/accept" signal combination. We use *super states* in these STGs to not restrict the implementation of the initiator or target unnecessarily to a single state. We refer to these super-states in our discussion in Sect. 4.1.1.

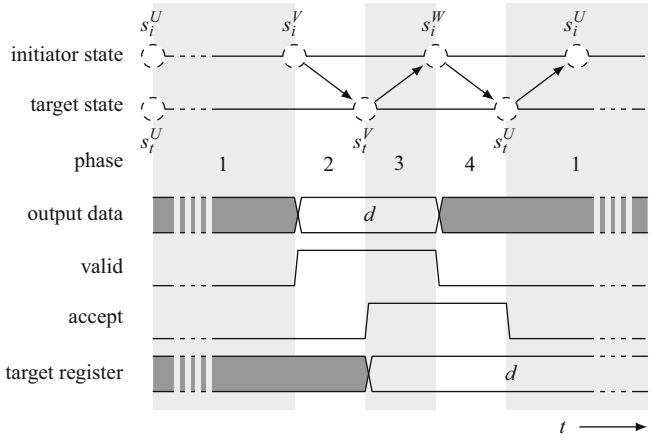


Fig. 3.13 Timing diagram

The initiator and target both start in a state that is part of respectively the *super state* s_i^U and s_t^U. The valid and accept control signals are set to logic-0 in these super-states. This marks the start of Phase 1 in the handshake protocol. The initiator subsequently starts Phase 2 of the protocol by transitioning to a state in the super-state s_i^V, where it applies the *data element* d_1 to its data output and toggles its valid output signal once. These actions are synchronous to the initiator clock "clk_i". The target continuously samples the valid signal of the initiator using the target clock "clk_t". When the target is in super-state s_t^U and observes that the initiator has toggled its valid signal, it responds by transitioning to a state in the super-state s_t^V, where it samples the data signals in a local register once using its local clock signal. In the super-state s_t^V, the target toggles its accept output signal once, synchronous to its local clock, to signal to the initiator that it has sampled the data. This ends Phase 2 of the protocol. The initiator continuously samples the accept signal of the target on its local clock. When the initiator is in the super-state s_i^V and observes that the target has toggled its accept signal as well, it responds by transitioning to a state in the super-state s_i^W. At this point the initiator is allowed to change the value on its data output.

The transfer of another data element d_2 occurs in the same way as the first data element d_1, except that the values of the valid and accept signals are inverted. During this second handshake, the initiator transitions from super-state s_i^W through super-state s_i^X back to a state in super-state s_i^U, while the target transitions from super-state s_t^V back to a state in super-state s_t^U, all under control of the valid and accept signals.

The *four-phase handshake protocol* is similarly illustrated in Figs. 3.13, 3.14, and 3.15. The initiator and target again start in a state that is part of respectively the *super state* s_i^U and s_t^U. The valid and accept control signals are set to logic-0 in these super-states. This marks the start of Phase 1 in this protocol. The initiator subsequently starts Phase 2 of this protocol by transitioning to a state in the super-state s_i^V, where it applies the *data element* d to its data output and asserts its valid output

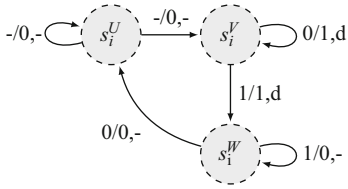


Fig. 3.14 Initiator STG

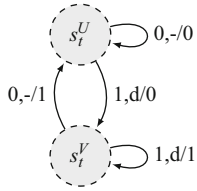


Fig. 3.15 Target STG

signal. These actions are synchronous to the initiator clock. The target continuously samples the valid signal of the initiator using the target clock. When the target is in super-state s_t^U and observes that the initiator has asserted its valid signal, it responds by transitioning to a state in the super-state s_t^V. In super-state s_t^V, it samples the data signals in a local register once using its local clock and asserts its accept output signal to signal to the initiator that it has sampled the data. This ends Phase 2 of this protocol. The initiator continuously samples the accept signal from the target on its local clock. When the initiator is in super-state s_i^V and observes that the target has asserted its accept signal, it responds by transitioning to a state in the super-state s_i^W, where it deasserts its valid output signal synchronous to its local clock. At this point the initiator is allowed to change the value on its data output. This ends Phase 3 of this protocol. When the target is in the super-state s_t^V and observes the deassertion of the valid signal by the initiator, it responds by transitioning to a state in the super-state s_t^U, where it deasserts its accept output signal synchronous to its local clock. This ends Phase 4 of this protocol. When the initiator is in the super-state s_i^W and observes that the target has deasserted its accept signal, it responds by transitioning to a state in the super-state s_i^U, where it can optionally start the transfer of another data element.

Please note that the target STGs in Figs. 3.12 and 3.15 are simplifications of real implementations to help focus on the operation of the handshake protocols. As both handshake protocols allow the target to delay its acceptance of the data, the target STG can be extended as shown in Fig. 3.16. The STG in Fig. 3.16 is the STG of a target implementing the two-phase handshake protocol, but with two additional super-states s_t^W and s_t^X. The target does not immediately acknowledge a request from the initiator when it is in the super-state s_t^W or s_t^X. The initiator stalls in respectively super-state s_i^V or s_i^X until the target returns to respectively super-state s_t^U or s_t^V and accepts the data. The super-states s_t^W and s_t^X can for example be used to implement the uninterrupted processing of previously-received data elements by the target.

These asynchronous handshake protocols do not prevent meta-stability on the sampled valid and accept control signals. Any meta-stability on these signals are

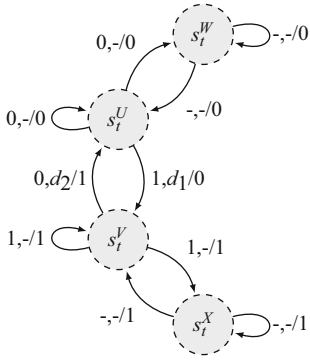


Fig. 3.16 Extended target STG

however guaranteed to be eventually resolved, because the initiator stalls and thereby keeps its valid and data signals stable until the target acknowledges that it has accepted the request from the initiator. Meta-stability on the handshake control signals therefore does not cause meta-stable data to be sampled. No meta-stability occurs when the data signals are sampled in a register in the target, because these data signals are held stable by the initiator when the target samples them. Consequently, handshake-based communication techniques ensure the correct transfer of data by design, even when the clocks are asynchronous.

3.1.4 SOC Communication Protocols

The example in Fig. 3.8 shows a unidirectional communication link between an initiator and a target. This link consists of one *signal group* that comprises a valid handshake signal, an accept handshake signal, and a set of associated data signals. The handshake signals ensure the correct transfer of the data signals without metastability problems. Modern communication protocols, such as the *AXI protocol* [1], the *device transaction level (DTL) protocol* [14] and the *open core protocol (OCP)* [15], use a bidirectional communication link between an initiator and a target that comprises a *request channel* and a *response channel* (refer to Fig. 3.17). These two communication channels transfer the write and read transactions between the initiator and the target. A *transaction* comprises a *request message* from the initiator to the target and an optional *response message* from the target to the initiator. Each message consists of one or more data elements that are individually transferred between the initiator and the target using a handshake. A *write transaction* consists of a *write request message* and an optional *write response message*. The write request message contains a *write command element* and one or more elements with the data to write. The optional *write response message* contains a *write acknowledge element*. A *read transaction* consists of a *read request message* and a *read response message*. The read request message contains a *read command element*, while the *read response message* contains one or more elements with the data that was read.

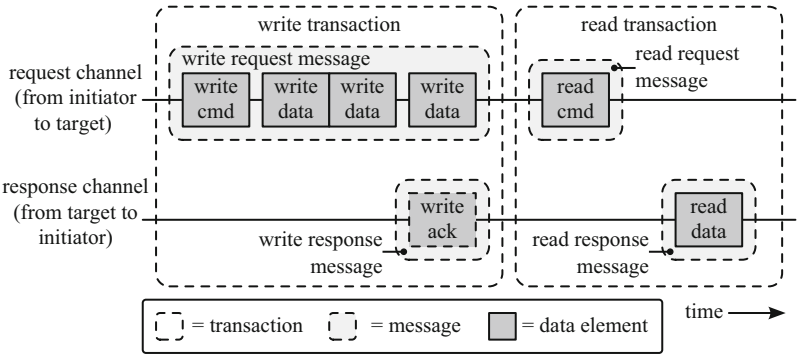


Fig. 3.17 Communication request and response channels, transactions, messages, and data elements

Table 3.2 Main signals and signal groups of the DTL communication protocol [14]

Name	Source[a]	Description
System group		
clk	S	DTL clock
rst_an	S	Asynchronous DTL reset
Command group		
cmd_read	I	Command read operation
cmd_addr	I	Command address
cmd_block_size	I	Command block size
cmd_rd_mask	I	Command read mask
cmd_valid	I	Command valid
cmd_accept	T	Command accept
Write group		
wr_data	I	Write data
wr_mask	I	Write data byte mask
wr_last	I	Write last
wr_valid	I	Write valid
wr_accept	T	Write accept
Read group		
rd_data	T	Read data
rd_last	T	Read last
rd_valid	T	Read valid
rd_accept	I	Read accept

[a] S system, I initiator, T target

The operating principles of these protocols are very similar. We therefore illustrate these principles using the DTL protocol below, because we also use this protocol in our case study in Chap. 8. Table 3.2 gives an overview and short description of the main signal groups and signals involved in the DTL communication between an initiator and a target [14]. Table 3.2 also lists the source of each signal.

The DTL protocol is a synchronous communication protocol, i.e., it requires the initiator and target to be part of the same clock domain. This allows the maximum

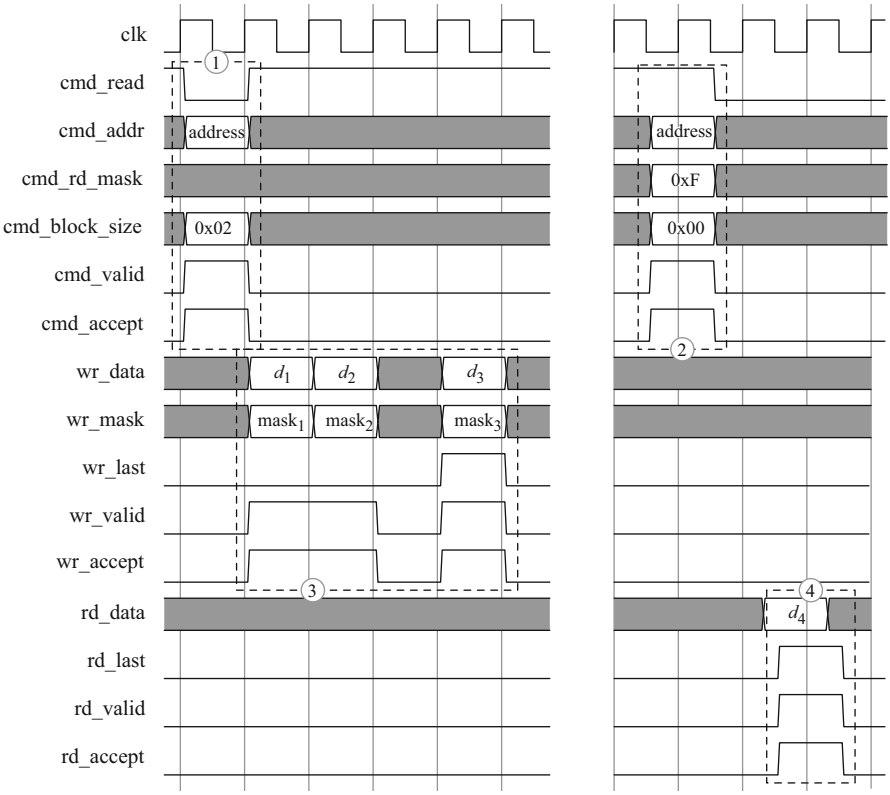


Fig. 3.18 Example DTL write and read transactions, based on [14]

throughput of the communication link, of one data element per clock cycle, to be utilized during data transfers. The protocol however still prescribes the use of handshake control signals to control the data transfer between the initiator and the target. The handshake control signals allow the initiator and target to execute independently from each other when they are not communicating with each other. The initiator will only stall when it has data to communicate to the target and the target is not ready to receive this data yet.

The DTL protocol uses three signal groups to transfer commands and data between the initiator and the target, even though the use of a single signal group for each channel is sufficient. SOC protocols typically use multiple signal groups to support *pipelined* and *concurrent transactions* and thereby obtain a higher system performance.

Figure 3.18 shows an example *write transaction* and an example *read transaction* on a DTL communication link, based on the DTL protocol specification in [14]. The initiator starts the transaction in both cases. It sends respectively a write command element or a read command element with command information to the target, using

the signals in the *command group* (indicated with ① and ② in Fig. 3.18). This information includes the type of command ("cmd_read"), the start address ("cmd_addr"), and the block size ("cmd_block_size"). The validation of this information by the initiator and its subsequent acceptance by the target is indicated by respectively the "cmd_valid" and the "cmd_accept" handshake signal. Write transfers take place similar to the transfer of the command, i.e., the initiator provides the data to write by means of the signals in the *write group* (indicated as ③ in Fig. 3.18). This information includes the data to write ("wr_data"), a possible byte mask ("wr_mask"), and a flag indicating whether the current data element is the last element in the write request message ("wr_last"). The validation of the write data by the initiator and its subsequent acceptance by the target is indicated by respectively the "wr_valid" and the "wr_accept" handshake signal. In the example write transaction in Fig. 3.18, the command element specifies a write operation ("cmd_read=0") and a block size of three elements ("cmd_block_size=2"). These three data elements are subsequently transferred from the initiator to the target. The transfer of the last write data element is indicated by the assertion of the "wr_last" signal.

A read response message takes place in the opposite direction compared to a read request message, e.g., the target provides the data that was read by means of the signals in the *read group* (indicated as ④). This information includes the data that was read ("rd_data") and a flag indicating whether the current data element is the last element in the read message ("rd_last"). The validation by the target of the data read and its subsequent acceptance by the initiator is controlled by respectively the "rd_valid" and "rd_accept" handshake signals.

To enable communication between asynchronous building blocks using the DTL communication protocol, an SOC design team has to use a so-called clock domain crossing (CDC) module [2]. A CDC module consists of two asynchronous building blocks that communicate with each other using an asynchronous protocol (refer to Fig. 3.19). An initiator block in clock domain a can use the DTL interface on the initiator side of the CDC module to write data into the CDC module. A target block in clock domain b can use the DTL interface on the target side of the CDC block to read this data. A memory inside the CDC module keeps track of the data elements that have been written but not yet read.

3.1.5 Variable Communication Duration

In exchange for a correct data transfer between an initiator and an target in different clock domains, asynchronous communication protocols introduce a variation in the duration of the handshakes. This is illustrated in Figs. 3.20 and 3.21 for the two-phase, asynchronous communication protocol. In Fig. 3.20, it takes two active edges on the initiator clock to complete Handshake I, measured from the active edge on which the initiator toggles its valid signal to the active edge on which the initiator samples a toggled accept signal from the target. Handshake II however takes three active edges on the initiator clock to complete. This difference in duration is caused

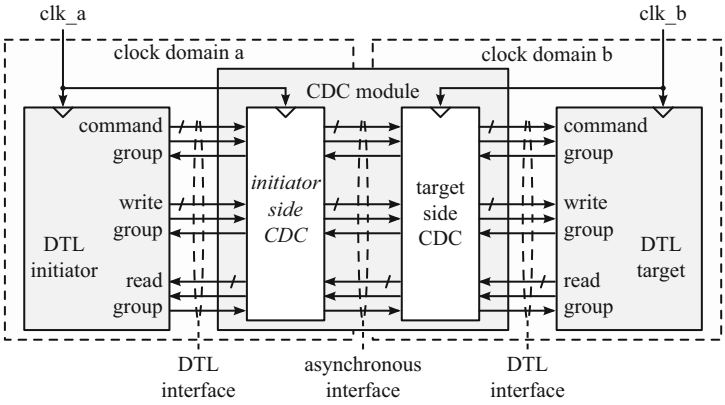


Fig. 3.19 Block diagram of a CDC module

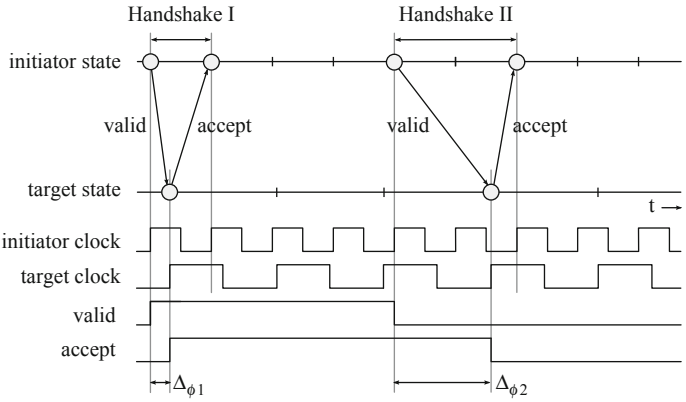


Fig. 3.20 Non-determinism at clock-cycle level due to clock phase differences

by a difference in the clock periods of the initiator and target, and by the phase difference between the two clocks at the start of the handshakes (refer to Δ_{ϕ_1} and Δ_{ϕ_2} in Fig. 3.20). The handshakes in Fig. 3.20 are examples in which the control signals from the initiator and the target are sampled without meta-stability.

Figure 3.21 shows another example handshake, where the target samples the valid control signal from the initiator with meta-stability, because the valid signal is asserted by the initiator in the setup-and-hold interval around an active edge on the target clock signal. In this example, it takes one additional clock cycle of the target clock to resolve this meta-stability. This causes a total duration of Handshake III of four active edges on the initiator clock. This difference in the duration of the handshake is visible as a variable communication latency between the two building blocks involved. This difference has to be taken into account by the SOC applications. We discuss the consequences of this variable delay next, in Sects. 3.2 and 3.3.

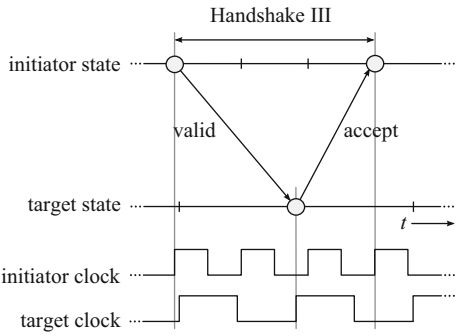


Fig. 3.21 Non-determinism at clock-cycle level due to meta-stability

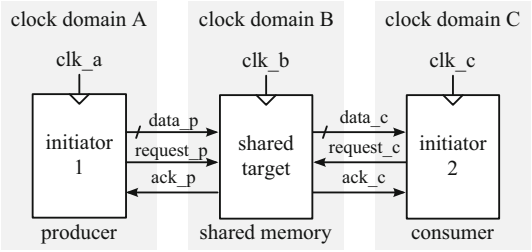


Fig. 3.22 Example SOC with three building blocks

3.2 Resource Sharing Between Building Blocks

It is common in a large SOC for a set of building blocks to require access to the same resource. For example, many of the building blocks in the SOC block diagram in Fig. 1.5 need write and read access to the external memory. The memory controller, shown at the top of Fig. 1.5, arbitrates between the write and read requests from these building blocks, by deciding the order in which they are applied to the off-chip SDRAM. The SDRAM is a *shared resource* in this SOC. The memory controller is the *arbiter* for this shared resource.

Figure 3.22 shows an Initiator 1 and an Initiator 2 that both require the services of a shared target. This shared target can however only accept and execute a single request at a time. Because the requests come to this target via separate ports, this target has to decide the order in which it services the requests from these two initiators. This process is called *arbitration*. Common resource arbitration algorithms include static, first-in/first-out, shortest job first, priority-based, and round-robin arbitration. We do not explore these specific algorithms here further, but instead analyze the possible effects that using a GALS design style has on the arbitration process.

In a GALS SOC, the request signals from initiators in other clock domains first need to be synchronized to the clock domain of the arbiter. This synchronization process prevents meta-stability problems, but introduces variable latencies in the communication of both requests to the shared target (refer to Sect. 3.1.5). The synchronized requests are subsequently combined with the requests originating from the arbiter's own clock domain and serviced in the order determined by the arbitration algorithm. Depending on the arbitration algorithm used, the arrival times of the

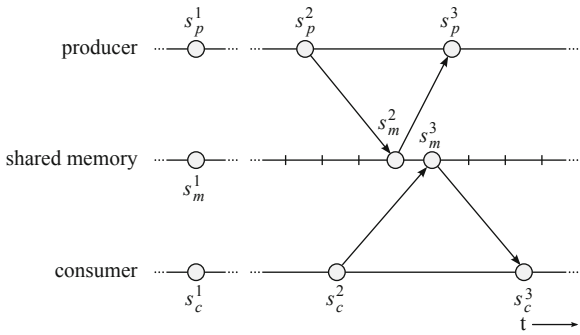


Fig. 3.23 "write before read" scenario

requests at the inputs of the arbiter may have an impact on the order in which these requests are subsequently handled. A difference in this order may in turn influence the SOC execution. Figures 3.23 and 3.24 illustrates this phenomenon for the two initiators and their shared target; Initiator 1 is a producer of data and Initiator 2 is a consumer of data. Both initiators communicate via separate ports to a shared memory. This shared memory can only accept and execute a single request at a time. In Fig. 3.23, the producer is the first initiator to start a write request, which is soon followed by a read request of the consumer. The producer's request is the first request to arrive at the shared memory and is therefore executed first. Afterwards the request of the consumer is executed by the shared memory. Another possible request sequence is shown in Fig. 3.24, but with a different transaction sequence. This time, due to different latencies on the communication path between the producer and the shared memory, and between the consumer and the shared memory, the read request of the consumer arrives before the write request of the producer. The read request of the consumer is therefore executed before the write request of the producer. The response message to the consumer may be different from what it was in the scenario in Fig. 3.23, because for example the consumer requested data from the shared memory before the producer was able to write it. This is reflected in the state of the shared memory, when the read request of the consumer comes in. In Fig. 3.23, this state

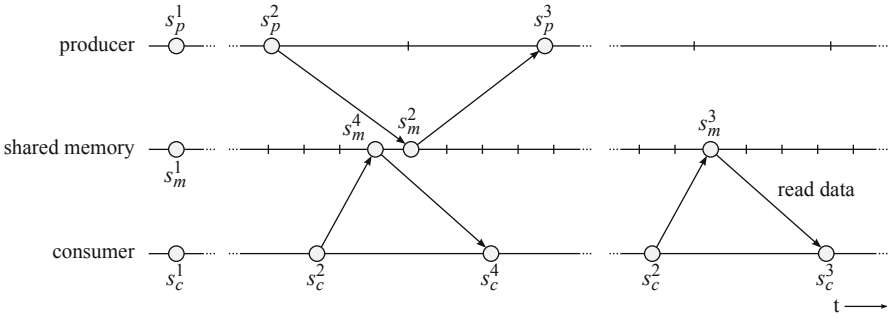


Fig. 3.24 "read before write, and then reread" scenario

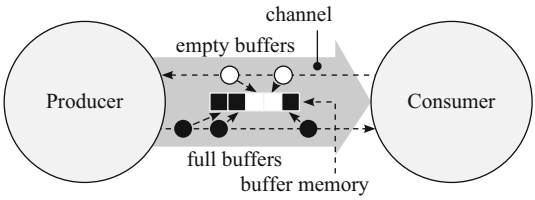


Fig. 3.25 C-HEAP overview, based on [3]

is state s_m^2, reflecting that the producer has written its data. In Fig. 3.24, this state is however still state s_m^1 prior to the arrival of the request from the consumer. The shared memory handles the read request of the consumer by transitioning to state s_m^4. This is a different state than the state s_m^3 where the shared memory transitioned to when it handled the read request of the consumer in Fig. 3.23. This state difference is caused by the fact that the producer has not (yet) written its data at this point. In this case, the consumer can read functionally incorrect or undefined data that may lead to undefined and potentially-harmful behavior of the consumer.

It is common for SOC design teams to take measures in the SOC implementation, to minimize the impact that these differences in request arrival times have on the order in which the requests are subsequently serviced, and on the execution of the SOC as a whole. One possible measure is at the abstraction level of the requests themselves. The team can use a static arbitration schedule to force the arbiter to always let the producer write its data, before letting the consumer read it. The use of such a static schedule may however cause an under-utilization of the shared resource. This happens, for example, when the arbiter follows its static arbitration schedule and consequently has to wait a long time before it receives a request from the scheduled initiator. While the arbiter waits, it is not allowed to service unrelated requests from other initiators to improve the resource utilization. In practice, static arbitration schedules are therefore hardly used in modern SOCs. We discuss this and other measures to make the execution of an SOC more deterministic in Sect. 9.2.

A second and more commonly-used measure to solve the impact of different request arrival orders on the SOC execution is the use of a communication protocol at an abstraction level above the level of individual requests. These protocols relax the scheduling requirements on the arbiter, by ensuring that the application dependencies between the initiators are met through their use of this higher-level communication protocol. Many algorithms exist for scalable data communication between multiple, asynchronous modules [8]. Space restrictions do not allow us to describe these in detail. Instead, we illustrate below how one of these higher-level communication protocol prevents undefined SOC behavior as a consequence of different request handling orders. We use a multi-processor communication protocol called the C-HEAP communication protocol [3, 13] in this illustration, because we use an implementation of this protocol in our case study in Chap. 8.

The C-HEAP communication protocol can be used between two tasks that execute on separate, asynchronous SOC modules. These two tasks share a unidirectional *channel* to communicate blocks of data, called *tokens* (refer to Fig. 3.25). The channel

has a fixed *capacity*, consisting of a number of token buffers. These buffers are initially empty. The protocol provides separate primitives for the synchronization and the data transportation between the tasks. The Producer on the channel first has to claim space on the channel using the `claim_space` primitive (refer to Listing 3.1). The producer either blocks on this primitive when no empty buffers are available, or it is granted an empty buffer to use. The producer can subsequently fill this buffer with a token, and release the filled buffer on the channel using the `release_data` primitive. Likewise, the Consumer on the channel first has to claim data from the channel using the `claim_data` primitive (refer to Listing 3.2). The Consumer either blocks on this primitive when no filled buffers are available, or it is granted a filled buffer to use. The consumer can subsequently empty this buffer and release it to the channel using the `release_space` primitive.

The four C-HEAP primitives combined prevent that the Producer overwrites a full buffer, that the Consumer reads an empty buffer, and that the capacity of the channel is exceeded. The protocol helps to provide these guarantees independent from the actual request orders to the shared channel. Under certain timing conditions, the operation of this protocol may however still fail, as we investigated in [6].

Although the use of a higher-level abstraction protocol prevents undefined behavior to occur as a result of the different request handling orders, there is a temporal difference, i.e., in the amount of time it takes the tasks to perform their functions. This is because the producer and consumer may poll more or less often, depending on how their memory accesses are interleaved. This difference has to be taken into account by the SOC design team, for example, by analyzing the worst-case scenario for producers and consumers to access shared resources, and by subsequently ensuring that the SOC applications meet the customer requirements under worst-case conditions [12].

3.3 Complicating Factors for Debugging

In this section, we identify three additional factors that complicate the task of debugging the silicon implementation of an SOC with multiple, interacting building blocks using the debug process described in Sect. 1.3.

Listing 3.1 Producer pseudo code, based on [3]

```
1 void producer{} {
2   Token *token = claim_space(channel);
3   store(token);
4   release_data(token);
5 }
```

Listing 3.2 Consumer pseudo code, based on [3]

```
1 void consumer{} {
2   Token *token = claim_data(channel);
3   load(token);
4   release_space(token);
5 }
```

3.3.1 Non-determinism at the Clock-Cycle, Handshake, and Transaction Level

In a synchronous SOC, we can determine the expected state and outputs of a building block at a particular point in time using Eqs. 2.4 and 2.6. This is no longer possible in a GALS SOC for three reasons. First, the internal clock signals are typically generated using on-chip phase-locked loops (PLLs), DLLs, and crystal oscillators that do not have a defined start-up state (refer to Sect. 2.2.2). In other words, the start-up phase of each clock is typically undefined. Second, each clock is subject to physical phenomena, such as *clock jitter* and *clock drift*, that affect the location of their active edges in time. These phenomena are neither predictable nor easily controllable. Third, the SOC may use DVFS techniques to dynamically manage the power and performance operating points of each building block, thereby changing the clock period of each building block and the propagation delays through its logic gates.

These variations may cause the communication handshakes between the building blocks to take place at different points in time and/or have a different duration (refer to Sect. 3.2) from one debug experiment to another. This temporal change may in turn influence the request order. A different request order may cause the building blocks to exhibit different behavior during different executions. As a result, it is no longer possible to precisely predict what the correct global state of the SOC is at a particular absolute point in time. SOC design teams rely on appropriate arbitration algorithms, high-level communication protocols and worst-case analysis, to ensure that no module exhibits undefined behavior because of these variations, and the SOC meets its customer requirements specification.

Although these variations no longer cause undefined behavior, they do still cause problems for traditional silicon debug methods, because the expected state and outputs are no longer known for a particular, absolute point in time. This makes it very difficult to properly compare the behaviors of two SOC implementations with each other for debug, as is done in the debug process in Fig. 1.12. We use the example state trace in Fig. 3.26 to illustrate this debug problem.

Figure 3.26 shows an example state trace of an SOC comprising two building blocks that communicate with each other through a two-phase handshake protocol. We use explicit states as opposed to super-states in this example for clarity. We call the state of each building block its *local state* and the combination of the local states of individual building blocks the *global state* of the SOC. The time lines in Fig. 3.26 are dashed to indicate that the individual clocks of the initiator and target are subject to undefined start-up phases, clock jitter and drift, and DVFS techniques. Consequently,

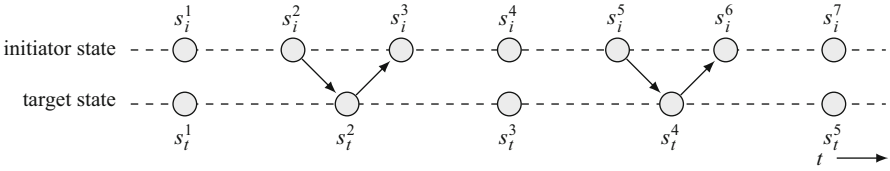


Fig. 3.26 One possible execution of an example SOC

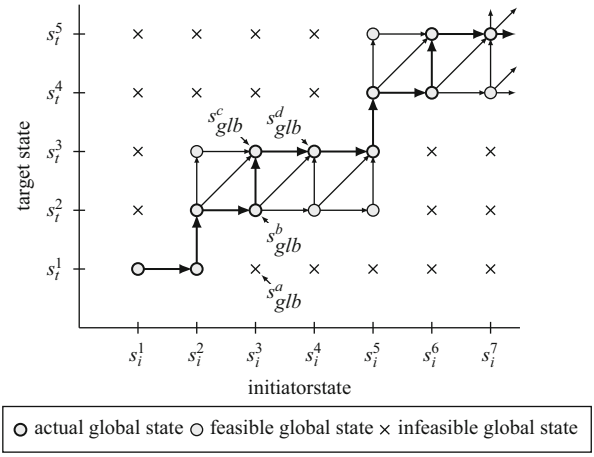


Fig. 3.27 Infeasible global state s^a_{glb} in the state space for the state trace in Fig. 3.26

the individual initiator and target states can move freely along the horizontal time axis, only subject to the local state dependencies, and to the dependencies between the states of the two blocks that are imposed by the handshake protocol. The resulting *global state space* for this example SOC and execution is shown in Fig. 3.27, with the state of the initiator on the horizontal axis, and the state of the target on the vertical axis. The initiator starts off in state s_i^1 and the target in state s_t^1 on the left-hand side of Fig. 3.26. This corresponds to global state (s_i^1, s_t^1) in Fig. 3.27. The initiator then starts a handshake with the target when it enters state s_i^2. The initiator cannot proceed to its next state, s_i^3, until the target accepts the data associated with the handshake in state s_t^2. Therefore it is functionally infeasible for the SOC to be, for example, in the global state $s^a_{glb} = (s_i^3, s_t^1)$ in Fig. 3.27. We have marked this and all other *infeasible global states* with crosses in Fig. 3.27. The remaining states are the *feasible global states*. Figure 3.27 clearly shows that the handshake protocol between the initiator and the target imposes restrictions on the global states that are functionally feasible during the execution of this SOC.

Depending on the exact position of the active edges of the initiator and target clocks, which are subject to the effects mentioned before, the SOC takes a particular path through this feasible global state space, from one feasible state to another, in the direction of the arrows. An example execution path of the SOC is indicated by the thick arrows and states in Fig. 3.27. We refer to each possible execution path of

the SOC through its feasible global state space as a *feasible state path*, and to the actual path that the SOC takes as the *actual state path*. We call the global states that are part of the actual state path *actual global states*. We have indicated an actual state path by the thick arrows and states in Fig. 3.27. On this particular, actual state path, the target transitions from the global state $s^b_{glb} = (s^3_i, s^2_t)$ first to state s^3_t and global state $s_{glb}c = (s^3_i, s^3_t)$, before the initiator transitions to state s^4_i and global state $s^d_{glb} = (s^4_i, s^3_t)$. The trace in Fig. 3.26 shows these two state transitions as simultaneous transitions. This difference is possible, because the events triggering these transitions are events local to these building blocks, and are therefore *logically concurrent* [5]. Logical concurrency means that either building block may transition first or both blocks may transition at the same time. The specific option out of these three that is taken during a debug experiment depends on the relative position of the active edges of their respective clocks and may therefore vary from one debug experiment to another.

Please note that the handshake protocol explicitly orders the state transitions of the initiator and target during communication handshakes. The initiator and the target need to transition to their next states in the order imposed by the handshake protocol. The initiator and target wait in one of their *stall states*, until the other building block makes the state transition that is required for the handshake to proceed. We use this property in our debug approach, described in more detail in Chap. 4.

Based on these observations, we define complicating factor CF-7 for debugging.

CF-7: Non-determinism at Clock-Cycle, Handshake, and Transaction Levels *The behavior of a building block, which interacts asynchronously with another building block, may no longer be deterministic at the clock-cycle, handshake, and transaction levels, due to the unpredictable timing of the active edges of their respective clocks and the unpredictable amount of time that is required to resolve possible meta-stability on their handshake control signals.*

We need a temporal abstraction function to address this complicating factor, similar to the structural, data, and behavioral abstraction functions that we need to address the problem of the state comparison, identified with complicating factor CF-5 in Sect. 2.2.3. This abstraction function raises the temporal execution control during debug to the abstraction level of for example handshakes, messages, transactions, and even software-defined tokens, to filter out the non-determinism in the SOC execution at the lowest abstraction level(s). We discuss this temporal abstraction as part of our debug approach in Chap. 4.

3.3.2 Uncertain Errors

We consider again the differences between the scenarios in Figs. 3.23 and 3.24 to illustrate another consequence of the differences that can take place in the execution of an SOC from one debug experiment to another. In this figure, we show the effect of a difference in request handling order using two example, feasible execution paths

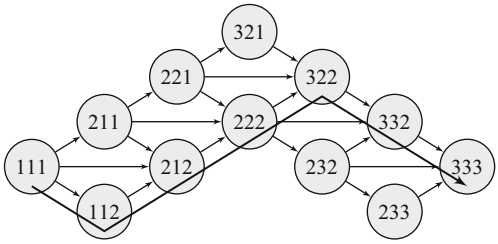


Fig. 3.28 Feasible state paths for the "write before read" scenarioglo

for the corresponding SOC. Figures 3.28 and 3.29 show all feasible state paths for the example scenarios in respectively Figs. 3.23 and 3.24. The label "*ijk*" in each node in these graphs corresponds to a global SOC state (s_p^i, s_m^j, s_c^k). We have left out the infeasible global states for clarity. Note how the handshake protocol between the three building blocks clearly imposes restrictions on the state progress of the individual building blocks, and therefore restricts the feasible global states and the transitions between them.

There are still many feasible state paths in Figs. 3.28 and 3.29. When the implementation of an SOC has an error and this error is only activated on a subset of the feasible state paths of the SOC, then our ability to detect and analyze this error is dependent on the probability that one of these paths is actually taken when a debug experiment is performed. This probability may be very low, e.g., if it depends on a certain phase difference between the clocks of the building blocks, or on metastability of the handshake control signals (refer to Figs. 3.20 and 3.21). This problem is illustrated in Fig. 3.29, where the SOC contains an error that is only activated when the SOC transitions from global state (s_p^2, s_m^4, s_c^2) to (s_p^2, s_m^2, s_c^4), indicated with the nodes with labels "242" and "224". Depending on the *actual state path* that the SOC takes during a debug experiment, we therefore may or may not observe this error. For example, the SOC does not make this state transition when it follows the state path in Fig. 3.28. It does make this state transition for the state path in Fig. 3.29. It may be difficult or even impossible to force the SOC to take the erroneous state transition during a debug experiment using its I/O pins (refer to complicating factor CF-2). When a debug engineer uses the debug process described in Sect. 1.3, it may therefore take many debug experiments to first have the SOC take a state path that activates the error, and to subsequently spatially and temporally localize this error. We call this type of error an *uncertain error*, as it is uncertain whether the SOC takes a feasible state path that activates this error in a debug experiment or not. In contrast, a *certain error* is an error that is activated on every feasible state path of the SOC. An example of a certain error is an error that occurs on the state transition from (s_i^2, s_t^1) to (s_i^2, s_t^2) in Fig. 3.27, because this transition has to occur during every SOC execution.

Based on these observations, we define complicating factor CF-8 for debugging.

CF-8: Uncertain Errors *Non-determinism can cause the execution of the SOC to follow one of many feasible state paths. The activation of an uncertain error depends*

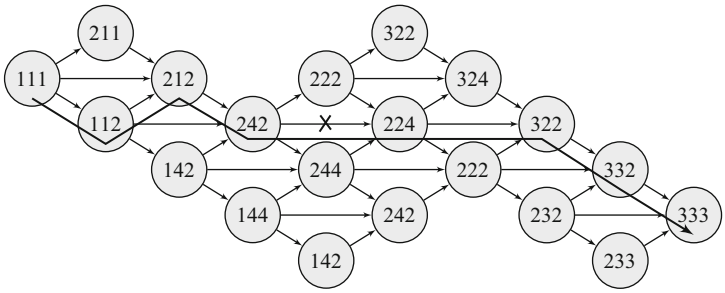


Fig. 3.29 Feasible state paths for the "read before write" scenario

Table 3.3 Overview of global state sampling techniques applied to a GALS SOC

State sampling technique	Locally-consistent	Globally-consistent	Non-intrusive	Physically feasible	No data loss	Use with DVFS
Synchronous and mesochronous	−	−	+	+	−	−
Periodic (LCM)	+	+	+	−	+	−
Periodic (GCD)	+	+	+	+	−	−
Asynchronous	+	+	−	+	+	+
GALS	+	−	+	+	+	+

+ pro, − con

on the actual state path that an SOC takes during a debug experiment, causing this type of error to only be observable in some debug experiments but not in others.

3.3.3 No Instantaneous, Distributed, Global Actions

Debugging requires observing and therefore sampling the state of the SOC building blocks for subsequent analysis. To debug an SOC, we may need to inspect and therefore sample the *global state* of the SOC at a single point on its actual state path. This state sampling process can be thought of as adding a debug building block to the SOC, which needs to be clocked and receive debug data from each functional building block in the SOC. As shown in Table 3.1, we have several options for the clock relation between the clock of this debug building block and every functional clock in the SOC. Table 3.3 provides an overview of the pros and cons of applying different global state sampling techniques to a GALS SOC, corresponding to each option.

In a *synchronous or mesochronous global state sampling technique*, we sample the states of all building blocks using a single debug clock. As explained in Sect. 3.1, we have to align the active edges of this debug clock with the active edges of the functional clock to ensure we sample the data in a building block without metastability. Sampling data in a building block outside of the setup-and-hold interval

around the active edges of its functional clock may cause us to sample an *inconsistent local state* (refer to Sect. 2.2.3). When this happens, we lose debug data and also obtain an inconsistent global state. It is possible to use a single debug clock in a *synchronous* or *mesochronous* SOC. It is however not possible to create a debug clock that is synchronous or mesochronous to all asynchronous clocks in a GALS SOC.

In a *periodic sampling technique*, we choose the debug clock such that it shares a (potentially-different) periodic master clock with the clock of each building block. We can then safely sample the state of each building block on the common edges between the debug clock and the clock of the building block. This leaves however two options for the debug clock itself. The first option is to use a debug clock with a frequency, which is a common multiple of the clock frequencies of the building blocks. Unfortunately modern SOCs use many clocks with diverse frequencies, yielding even a LCM clock frequency that can easily reach into the GHz range. It may therefore not be physically feasible to implement this clock. The second option is to use a debug clock with a frequency, which is a common divisor of the clock frequencies of the building blocks. Using this clock frequency however leads to a loss of debug data through the sub-sampling of the states of the building blocks, even when using the greatest common divisor (GCD) of the clock frequencies. Either option is also very difficult to combine with DVFS techniques.

In an *asynchronous sampling technique*, we perform a handshake between each functional building block and the debug building block to sample its state. This handshake can influence the behavior of the functional building block through meta-stability on the handshake signals, even when the debug building block is always ready to accept the data from that building block (refer to Sect. 3.3.2).

In a *CALS sampling technique*, we sample with multiple debug clocks. There is a separate debug clock for each clock domain, which is synchronous to that clock domain. This ensures that we sample *consistent local states*. We still however need to cross clock domain boundaries with a control signal that triggers the sampling operation in the different clock domains. This trigger signal can be safely communicated across clock domain boundaries using a handshake protocol. However, as crossing a clock domain boundary between asynchronous clock domains using a handshake introduces a variable latency, states are sampled in the different clock domains at different points in time. This may cause us to sample an *inconsistent global state*. In the state space shown in Fig. 3.30, we may for example sample the state of the target as s_t^3 when the SOC is in global state $s_{glb}^e = (s_i 5, s_t^3)$, but only later sample the state of the initiator as s_i^6 when the SOC has already transitioned to global state $s_{glb}^f = (s_i^6, s_t^4)$. The combination of these two *locally-consistent states* yields an infeasible, and therefore inconsistent, global state $s_{glb}^g = (s_i^6, s_t^3)$.

The main reasons for these state sampling limitations are the finite speed with which a debug event can be distributed over the entire SOC to trigger a global action, and the possible need to synchronize an incoming debug event signal to the local clock to avoid meta-stability in the local state. This intrinsic delay, combined with a variable synchronization delay, prevents us from executing an instantaneous, distributed, global action. We therefore define complicating factor CF-9 for debugging.

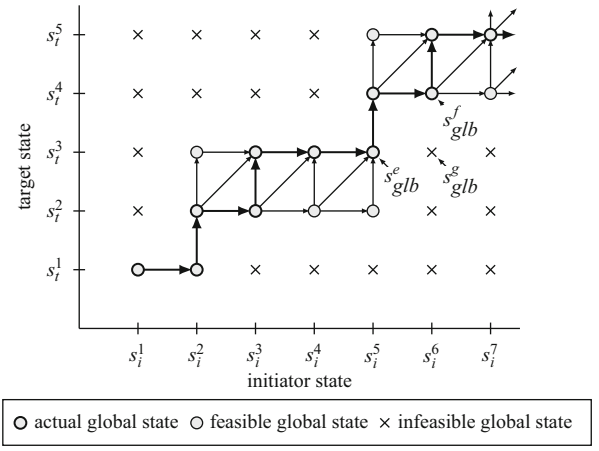


Fig. 3.30 Actual global states s^e_{glb} and s^f_{glb} and sampled global state s^g_{glb} in the state space for the state trace in Fig. 3.26

CF-9: No Instantaneous, Distributed, Global Actions *We cannot instantaneously execute a global action across all building blocks in a GALS SOC.*

A first consequence of complicating factor CF-9 is that we cannot instantaneously stop all on-chip clocks to control the execution of the SOC. A second consequence is that there is therefore no practical state sampling technique for a GALS SOC that can produce locally- and/or globally-inconsistent states, without influencing the functional execution of the SOC and without losing state data.

3.4 Summary

In this chapter, we analyzed SOCs consisting of multiple, interacting building blocks. We studied in detail the design steps that SOC design teams take in the implementation of a GALS SOC to ensure the correct transfer of data between building blocks. We then investigated the request arbitration for shared resources. Based on our analysis, we defined three additional factors that complicate debugging these SOCs using the debug process described in Sect. 1.3. One factor involves the unknown clock relation between the clocks of the building blocks, causing individual handshakes to take a variable amount of time. This variation in turn causes that the behavior of the individual building blocks, and consequently of the SOC, may no longer be deterministic at the clock-cycle, handshake, and transaction levels. The second factor is the occurrence of uncertain errors that may only be activated during certain debug experiments and not during others. The third factor expresses limitations on the techniques that we can use to stop the execution of a GALS SOC and sample its global state.

References

1. ARM Limited. *AMBA AXI Protocol Specification*, June 2003.
2. Clifford E. Cummings. Simulation and synthesis techniques for asynchronous fifo design, 2002.
3. David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley Publishing, 2007.
4. A.A.J. De Lange and I.C. Kang. Modeling and real-time simulation of complex media processing systems with parallel distributed computing. In *Proc. International Multi-Conference on Applied Informatics*, pages 818–824, 2003.
5. A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2008.
6. Erik Larsson, Bart Vermeulen, and Kees Goossens. Distributed Architecture for Checking Global Properties during Post Silicon Debug. In *Proc. European Test Symposium*, 5 2010.
7. George H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.
8. John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
9. David G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.
10. Edward F Moore. Gedanken-experiments on sequential machines. 34:129–153, 1956.
11. Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical Design of Globally-Asynchronous Locally-Synchronous Systems. In *Proc. International Symposium on Asynchronous Circuits and Systems*, page 52, Washington, DC, USA, 2000. IEEE Computer Society Press.
12. A.T. Nelson, A. Hansson, H. Corporaal, and K.G.W. Goossens. Conservative application-level performance analysis through simulation of mpsocs. In *Proc. 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, Scottsdale, USA, October 2010.
13. 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. *Design Automation for Embedded Systems*, 7(3): 233–270, 2002.
14. NXP Semiconductors. *CoReUse 5.0 Device Transaction Level (DTL) Protocol Specification. Version 5.0*, December 2009.
15. OCP International Partnership. *Open Core Protocol Specification. v3.0*, 2009.

Part III
The CSAR Debug Approach

Chapter 4
CSAR Debug Overview

Abstract In this chapter we introduce our communication-centric, scan-based, abstraction-based, run/stop-based (CSAR) debug approach and its associated infrastructure. This approach and its infrastructure meets the post-silicon debug requirements, identified in Chap. 1, and reduces the effects of the complicating factors for debugging, identified in Chaps. 2 and 3. We first give an introduction of the CSAR debug approach and its on-chip and off-chip debug infrastructure in Sect. 4.1, and show how it is used. In Sect. 4.2, we detail our rationale behind the CSAR debug approach, by describing in detail how this approach addresses the debug requirements and complicating factors for debugging. The CSAR debug approach imposes requirements on the on-chip and off-chip debug infrastructure. We review and formulate these requirements in Sect. 4.3. We conclude this chapter with a summary in Sect. 4.4.

4.1 Introduction

The CSAR debug approach and infrastructure help the spatial and temporal localization of the root cause of a failure, by combining four key aspects to form a consistent solution: (1) communication-centric debug, (2) scan-based debug, (3) abstraction-based debug, and (4) run/stop-based debug. In particular, we instrument the SOC to take control over the handshake control signals that are used in the communication protocols between the SOC building blocks. This communication control can be performed selectively (per communication link, at different points in time) to implement spatial and temporal scoping. It can also be performed at several levels of temporal abstraction, for example, at the handshake, message, and transaction levels. With this communication-centric execution control functionality, we can (re)create a partial or total transaction order in the SOC. We add a fast and scalable EDI on-chip, to approximate instantaneous debug event delivery and coordinate the actions between these instruments on multiple communication links. By not allowing the communication handshake protocol to complete, we furthermore ensure that each SOC building block remains in a stall state. We can subsequently sample its state with its local clock and obtain a *locally-consistent state*. The stall states of all building blocks combined constitute a globally-consistent SOC state. By using communication-centric execution control, we can therefore extract an SOC state from the silicon implementation

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6_4,

that is both locally and globally consistent. Structural, data, and behavioral abstractions can subsequently be performed on this extracted state by off-chip debugger software to allow this state to be compared to the state of other SOC implementations. We have furthermore architected the required on-chip CSAR infrastructure to be as non-intrusive and as efficient to implement and use as possible.

4.1.1 Communication-Centric Debug

The CSAR debug approach is a communication-centric debug approach, as we determine where the SOC is in its execution by monitoring the communication on the on-chip communication interconnect between the SOC modules. We furthermore control the on-chip communication by preventing the completion of specific handshakes that are used in the communication protocols between initiators and targets. Through this communication control, we can effectively stall the SOC at different points in its execution. We can also use this communication control to force a particular request order to take place on the input of shared resources. We call this feature *guided (re)play*. This feature helps us guide the SOC execution along a specific feasible state path, thereby increasing the probability of activating an (uncertain) error. Communication-centric debug requires however the support of on-chip modules. Figure 4.1 shows an example SOC in *functional mode*. This example SOC uses a NoC as its communication architecture.

The implementation of this SOC has been extended with support for communication-centric debug. We added communication monitors and PSIs, an EDI to connect them, an IEEE Std. 1149.1 TAP and associated controller [1], and a set of serially-connected configuration registers, constituting the debug control and status interconnect (DCSI). The TAP and the DCSI are used to configure the other CSAR modules. The monitors observe transactions, messages, and data elements on the communication links between the modules. These monitors generate debug events on the detection of pre-configured communication attributes. The monitors communicate their debug events using a dedicated and scalable EDI to the other monitors and the PSIs. This EDI can be configured to only forward events from one source to a subset of the monitors and PSIs. Note that Fig. 4.1 only shows the conceptual connectivity of the EDI. For a more accurate depiction, please refer to Fig. 5.1 in Sect. 5.1. Each PSI controls the flow of requests and responses between an initiator and a target, by preventing communication handshakes to complete after a particular debug event has been received via the EDI. Our PSIs thereby force the building blocks to remain in their stall states. A PSI contains a configuration register to allow the stop condition and its granularity to be configured via the TAP and the DCSI. Our DTL PSI supports stalling the communication at different levels of abstraction, specifically the granularities of DTL write and read transactions, DTL write requests, DTL read requests, DTL read response messages, and DTL command, DTL write, and DTL read data elements (refer to Sect. 3.1.4). Similar levels of granularity can be identified for other communication protocols, such as

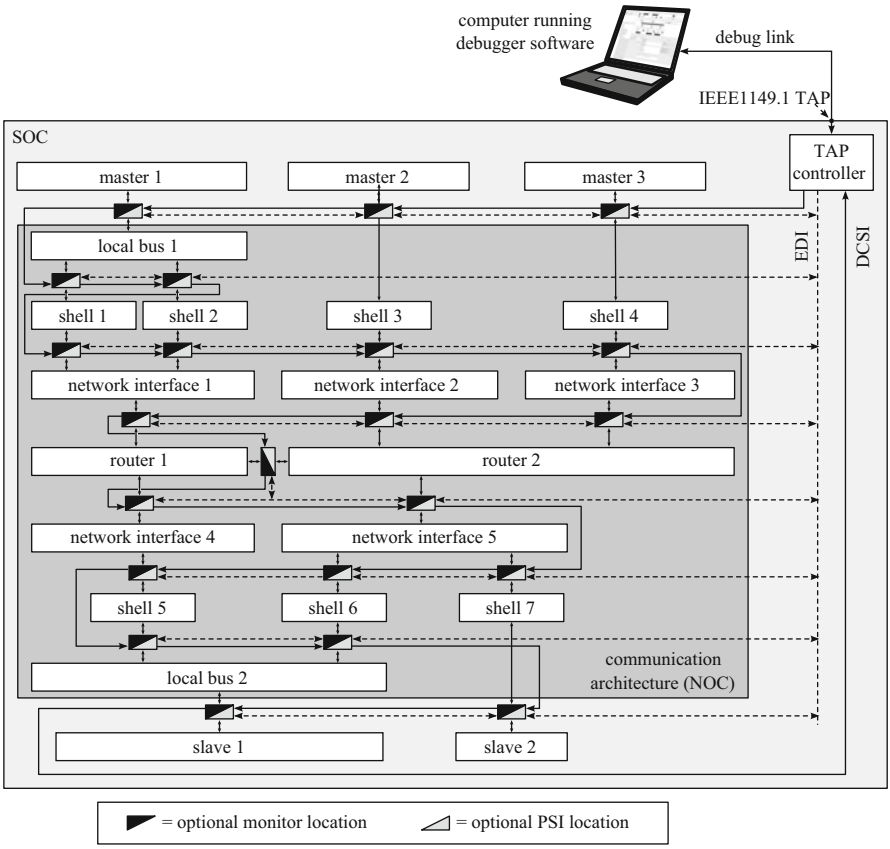


Fig. 4.1 An SOC in functional mode, in which we control the execution using the CSAR monitors and PSIs

the AXI protocol and OCP. This configuration register furthermore controls the PSI to selectively allow one or more communication handshakes on the corresponding communication link to complete at the configured granularity.

This on-chip functionality supports our guided (re)play process. This guided replay process can involve the *single-* or *multi-stepping* of the communication on a single communication link, or the *barrier-stepping* of the communication on multiple communication links simultaneously, both at the required communication granularities. Single-stepping the communication on one communication link at a time forces a unique transaction order for the SOC.

Consider as an example of single-stepping the SOC in Fig. 3.22. We can include PSIs on the communication links between the producer and the shared memory and between the consumer and the shared memory. With these PSIs we can single-step both communication links independently. This allows us to, for example, force the consumer to read from the shared memory before the producer has written to it. This way we force the SOC to take a state path from the ones shown in Fig. 3.29, instead of from the ones shown in Fig. 3.28.

With single-stepping, the order that we force upon the communication links has to be known in advance to correspond to the functional dependencies between the transactions on the different communication links in the SOC. Barrier-stepping instead allows a specific set of communication links to step simultaneously. It allows the debug engineer to be unaware and independent from the exact order of the steps of the links in this set. One barrier step requires that at least a minimum number of communication links make a step. The dependencies between the steps on the communication links in the set can then resolve themselves autonomously.

The PSI configuration register can also be used to query the current status of the communication link, e.g., whether the communication has been stopped and whether any communication handshakes are pending. The CSAR debugger software runs off-chip on a computer, which is connected to the TAP of the SOC via a debug link. This software has access to the DCSI via the TAP and can configure the on-chip monitors, PSIs, and EDI, by selecting a special, user-defined TAP instruction in the TAP controller.

4.1.2 Scan-Based Debug

The CSAR debug approach is a scan-based debug approach, as we reuse the scan chains that have been inserted in an SOC for manufacturing test also for debug state observability and controllability. These scan chains ensure high defect coverage by allowing the state of the internal flip-flops and embedded memories to be accessed from the SOC I/O pins during manufacturing test. Once the SOC has been stopped for debugging in a debug experiment, we access these scan chains via the TAP, because the I/O pins that are used during manufacturing test will be used by the SOC environment during functional operation. Through these scan chains, it is possible to extract the complete, scannable state of the SOC, and/or upload the same or a modified SOC state. This latter option helps in guiding the SOC execution along a particular feasible state path, thereby increasing the probability of activating an (uncertain) error.

Scan-based debug requires the support of on-chip modules (refer to Fig. 4.2). Figure 4.2 shows the same SOC as Fig. 4.1, except that Fig. 4.2 shows the SOC in its *debug scan mode*, whereas Fig. 4.1 shows the SOC in its *functional mode*.

A three-step process has to be used to correctly extract the functional SOC state from the scan chains. As the first step, we functionally stall the execution of the SOC using our communication-centric debug infrastructure. The use of this infrastructure ensure that the SOC execution is stopped, while the states in all modules are locally and globally consistent. We use the stall states that exist in the building blocks of the SOC modules to stop their respective executions. We can stall them there by not allowing the communication handshakes between the SOC modules to complete in a stall state. As the second step, we stop all on-chip clocks by accessing the on-chip clock and reset generation unit (CRGU) from the TAP. We have extended this CRGU for this purpose. We need to stop the functional clocks, because we do not want

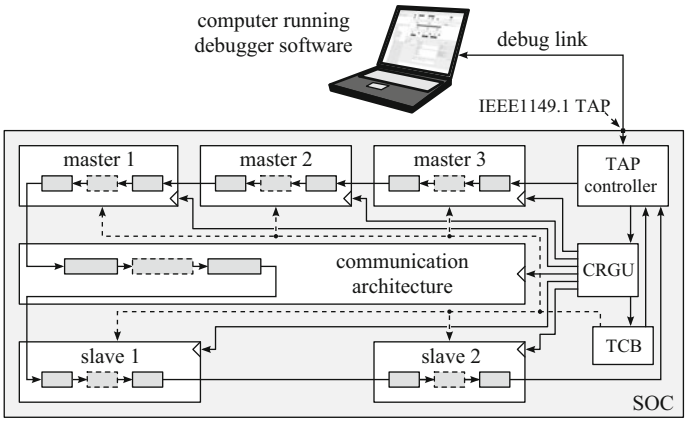


Fig. 4.2 An SOC in debug scan mode, in which we activate the scan chains and clock control

the scan chain activation signal to cause meta-stability at the inputs of the flip-flops in the different clock domains of the SOC. As the third step, we activate the scan chains after all the functional clocks have been stopped, by configuring the on-chip test control block (TCB) under TAP control. This activation now does not cause any accidental meta-stability in the flip-flops and embedded memories, because all functional clocks have been stopped before.

The TCB is a configuration register that sets the test control signals for manufacturing test. We reuse and extend this register to activate the debug scan mode. The TCB and CRGU are both part of the DCSI. Once the scan chains have been activated, we configure all scan-chains in one long serial register, which is accessible as a user-defined data register of the TAP controller. Care is taken that we use so-called *anti-skew flip-flops* at the output of the last scan chain in each clock domain. These flip-flops are clocked on the falling edge of the functional clock. Their use prevents race conditions between clock domains in shift mode, that may otherwise corrupt part of the state of the scan chains.

The off-chip debugger software has access to this long serial register, by selecting another special, user-defined TAP instruction in the TAP controller. While the external debugger accesses this long serial register, the TAP controller controls the CRGU to pass the TAP's test clock (TCK) clock signal to all clock domains in the SOC. This ensures that each section of this long serial register shifts synchronously (or mesochronously) with the TCK signal.

There is a third SOC mode, called the *debug normal mode*. This mode is the same as the *debug scan mode*, with the one exception that the scan chains are not activated from the TCB. This mode is used to access the states of embedded memories that are not part of the scan chains. The external debugger software can load write or read commands for these embedded memories in the scan chains in debug scan mode, and apply these commands to the embedded memories in debug normal mode. It does this by applying TCK clock pulses to the on-chip clock domains in the debug

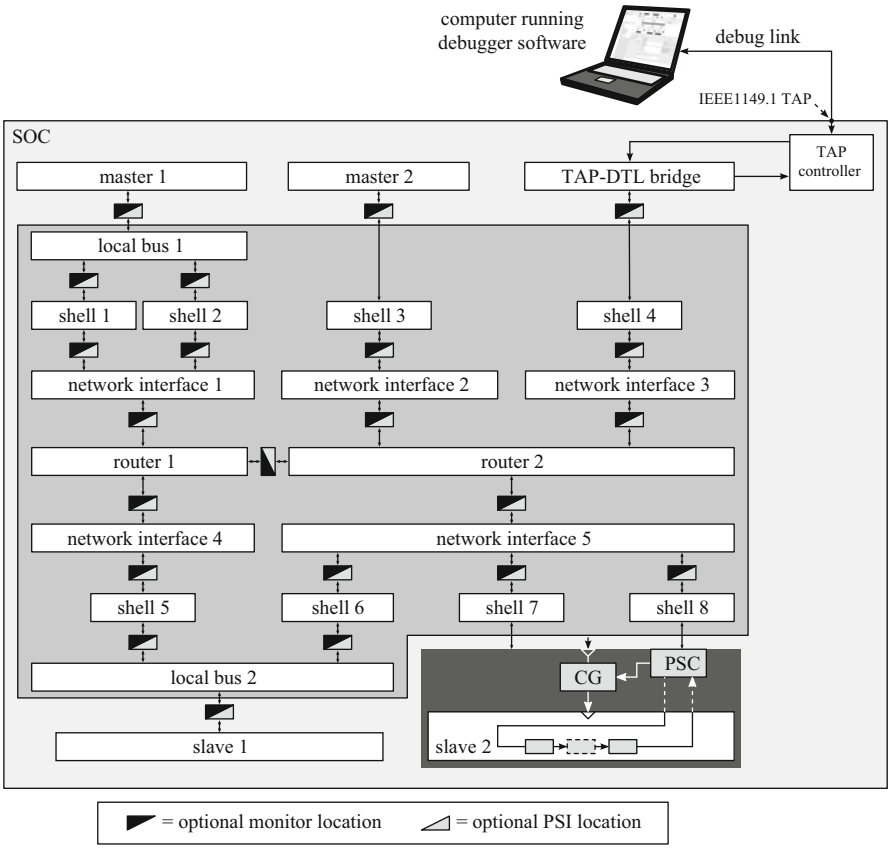


Fig. 4.3 An SOC in hybrid mode, in which we control the execution using the CSAR monitors and PSIs, and access the scan chains of a subset of SOC modules

normal mode and capturing their responses in the scan chains. These responses are subsequently extracted from these scan chains via the TAP in debug scan mode.

Finally, we provide a fourth, *hybrid SOC mode*, shown in Fig. 4.3. With this hybrid mode, we reduce the intrusiveness of our scan-based debug approach, while trading in some of its internal observability. We extend the test wrapper around the SOC modules with a protocol-specific controller (PSC) and a local clock gate. We can configure this PSC using a write operation on a dedicated, functional communication port (called the functional test port), to place the module in functional mode (which is the default), in debug scan mode, or in debug normal mode. When the module is configured in the debug scan mode, we can subsequently extract the state of the module via this functional test port on the PSC using successive read operations. We can also upload a new state in the module by performing successive write operations on this functional test port.

We enable access to this functional test port from the computer running the debugger software through the TAP, by including a special bridge module that acts as a master on the on-chip communication interconnect. This bridge module translates specific TAP accesses into write and read commands on the communication interconnect. We can access the PSC in the test wrapper around each SOC module by using its associated addresses in these write and read commands.

To use this more restrictive but less intrusive state observation option for a particular module, we first program its PSIs to stop all communication. The rest of the SOC can continue to execute, if its execution does not depend on this module. We subsequently reconfigure its PSC to switch the module to debug scan mode. At that same time, the test wrapper will continue to stop all communication to and from this module. The PSIs themselves becomes part of the module-level scan chains. The state of the module can then be extracted and/or modified via the communication interconnect. Afterwards, we reconfigure the PSC to switch back to functional mode, and if we have not done so already via the scan chains, instruct the PSI of the module to allow the communication to and from this module to resume.

4.1.3 Abstraction-Based Debug

The CSAR debug approach is an abstraction-based debug approach for three reasons: (1) the monitors observe the on-chip execution at different levels of temporal abstraction, (2) the PSIs control the communication at different levels of temporal abstraction, and (3) the off-chip debugger software can apply structural, data, and behavioral abstraction techniques to the state data retrieved from the scan chains. Figure 4.4 provides an overview of the abstraction levels that are supported by the CSAR debug approach. Abstraction is applied in steps going from the bottom of this figure, with a low abstraction level, to the top of this figure, with a high abstraction level. We show *structural*, *data*, and *behavioral abstraction* on the left-hand side, and *temporal abstraction* on the right-hand side of Fig. 4.4.

In each abstraction step, we combine multiple pieces of state information at one abstraction level to create state information at another, higher abstraction level. For example, the lowest abstraction level is represented by the content of the scan chains. This content comprises a sequence of bits that contain no structural information. We combine this state information with the hierarchical names of the flip-flops from the implementation refinement process in Fig. 1.4, to assign a state to each scannable flip-flop in the gate-level netlist of the SOC. We subsequently combine flip-flops that originate from the same multi-bit RTL signal in one RTL register, and combine related registers to one RTL memory instance.

One abstraction level above the states of registers and memories are the states of the individual building blocks. We obtain a locally-consistent state when we sample the state of a building block using its own clock signal. Note that these abstraction levels, indicated with ① in Fig. 4.4, only use structural information of the SOC. All modules together constitute the original, structural hierarchy of the SOC.

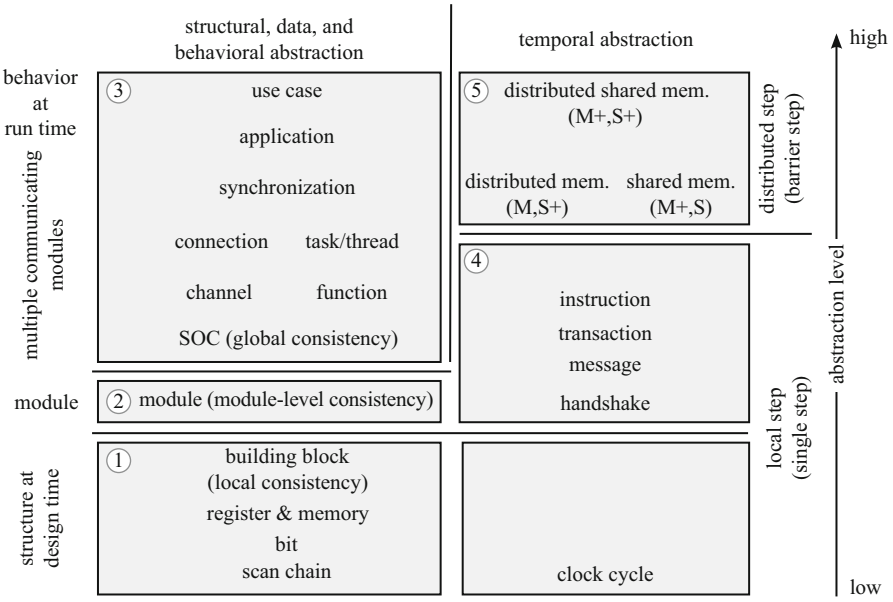


Fig. 4.4 CSAR debug abstractions, adapted from [3]

For the next abstraction level, we make an important step by interpreting one or more structural building blocks as a functional module. We stall the building blocks within a module in their stall states when we stop the handshake between the building blocks. When we subsequently sample the individual states using their own local clock, we obtain a consistent state for the complete module. We furthermore use behavioral information that allows us to functionally interpret the state of a module (indicated as ② in Fig. 4.4). For example, a module that implements a first-in first-out (FIFO) queue may be implemented using a memory and two registers to store respectively the FIFO data and the read and write pointers. With the abstraction from the structural level to the behavioral level of this FIFO queue, we interpret the values of these read and write pointers to, for example, only compare the valid entries in the memory between the silicon SOC implementation and its reference. We thereby avoid comparing the undefined substate in these two implementations. At this abstraction level, we can subsequently use a conventional computation-centric debug approach to debug the internal behavior of each module in isolation.

The higher abstraction levels from channel to use case, indicated with ③ in Fig. 4.4, abstract on the one hand from the functionality of the hardware to the functionality of the software, and on the other hand from the static view at design-time view to the dynamic view at run-time. In other words, these abstraction levels no longer show the structural SOC modules, but instead show their logical function, i.e., how these modules have been programmed. For the computation inside the SOC, we can observe how the structural processors execute logical *functions* as part of the execution of one or more *software threads* and one or more *software tasks*.

For the communication inside the SOC, we abstract from the structural modules that the communication interconnect is comprised of, to the logical *communication channels* and *connections*. The execution of the threads, tasks, channels, and connections are synchronized as part of a complete software *application*. The set of software applications that run on the SOC finally constitute the active *use case*.

On the right-hand side of Fig. 4.4, we use *temporal abstraction* to reduce the number and the temporal locations of our observations to those that are of interest to temporally localize a fault. The lowest temporal abstraction level is represented by individual clock cycles. We can observe the execution of a building block or its interface with the interconnect at every clock cycle, or we can use our monitors and PSIs to temporally abstract to communication *handshakes* and observe this interface only in clock cycles before or after information is actually transferred. The transfer of two or more related communication data elements corresponds to the transfer of one message, while the transfer of one or more related messages correspond to the transfer of one transaction (refer to ④ in Fig. 4.4 and Sect. 3.1.3). Above the transaction-level we identify the communication from one master to possibly multiple, distributed slaves (M,S+) and from multiple masters to a single, shared slave (M+,S). At the highest level of temporal abstraction, we identify communication between multiple masters and multiple, distributed, and shared slaves (indicated as ⑤ and (M+,S+) in Fig. 4.4).

4.1.4 Run/Stop-Based Debug

The CSAR debug approach is a run/stop-based debug approach, because we first use the monitors and PSIs to stop the execution of the SOC at a particular point of interest in its execution. We subsequently use the state controllability, provided by the scan chains, and the execution control provided by the monitors and the PSI, to guide the execution of the SOC further. We introduced in Sect. 1.3 a post-silicon debug process. We need to modify this debug process to handle the effects of the complicating factors, identified in Chaps. 2 and 3. Figure 4.5 shows our update of this post-silicon debug process.

This updated process is identical to the process described in Sect. 1.3 with the exception of Step 4. Instead of simply executing the silicon implementation and the reference, we need to take additional steps to be able to observe their respective behaviors for subsequent comparison in Step 5. Figure 4.6 shows the details of the CSAR inner loop that we execute as Step 4 in the updated post-silicon debug process. The CSAR inner loop uses the steps described below.

- Step 4.1: Set an appropriate breakpoint within the spatial and temporal scope of the silicon implementation and the reference, based on the scope decided on in Step 3 and/or Step 8, shown in Fig. 4.5.
- Step 4.2: Functionally reset the silicon implementation and the reference to correlate their states in time (refer to Sect. 2.2.2).

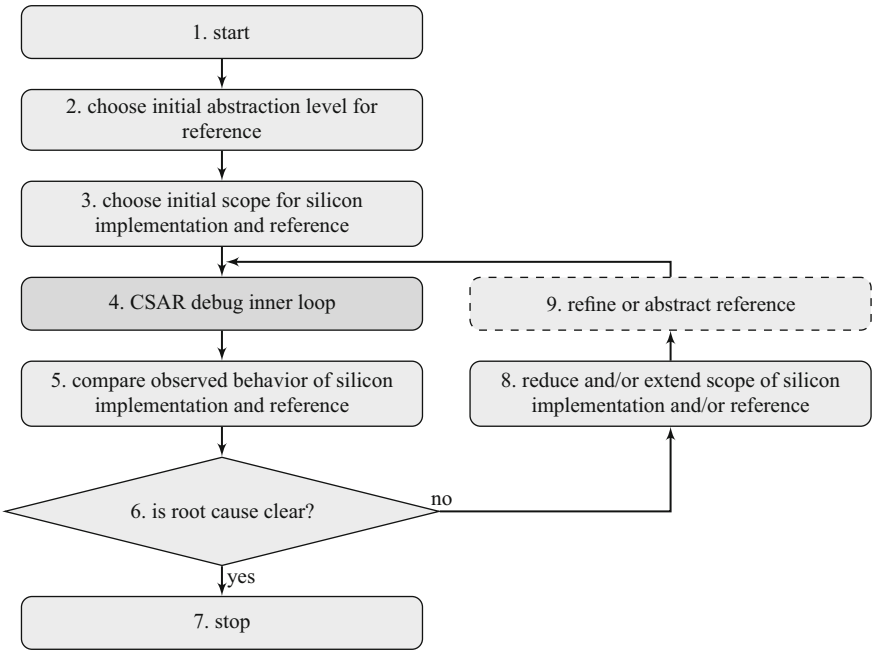


Fig. 4.5 Extended post-silicon debug process overview, adapted from [3]

- Step 4.3: Check whether the programmed breakpoint has been reached and the execution of the communication and computation inside the silicon implementation and the reference have stopped.
- Step 4.4: Use scan chain access to extract the state from the silicon implementation, as described in Sect. 4.1.2, and the reference. In case the reference is executing in a simulation environment, the debug engineer has the choice of either using the same state access mechanism as used for the silicon implementation, or using state access functionality of the simulation environment.
- Step 4.5: Decide whether the failure may be caused by an uncertain error. Proceed to Step 4.6 if this is likely, for example, because the failure only occurs intermittently, and not in all runs. Proceed to Step 4.7 if this is not the case, because the failure occurs during all executions.
- Step 4.6: Use single- and/or barrier-steps to advance the silicon implementation and the reference in a deterministic way, controlling among other things the request handling order of transactions by shared resources (refer to Sect. 3.2).
- Step 4.7: Decide whether the silicon implementation and the reference have reached the end of the temporal scope chosen in Step 3 or Step 8. Proceed to Step 4.8 if they have not, or to Step 5 in the process in Fig. 4.5 if they have.
- Step 4.8: Reconfigure the breakpoint condition to obtain the states of the silicon implementation and the reference at a later point in the debug experiment.
- Step 4.9: Resume the execution of the silicon implementation and the reference.

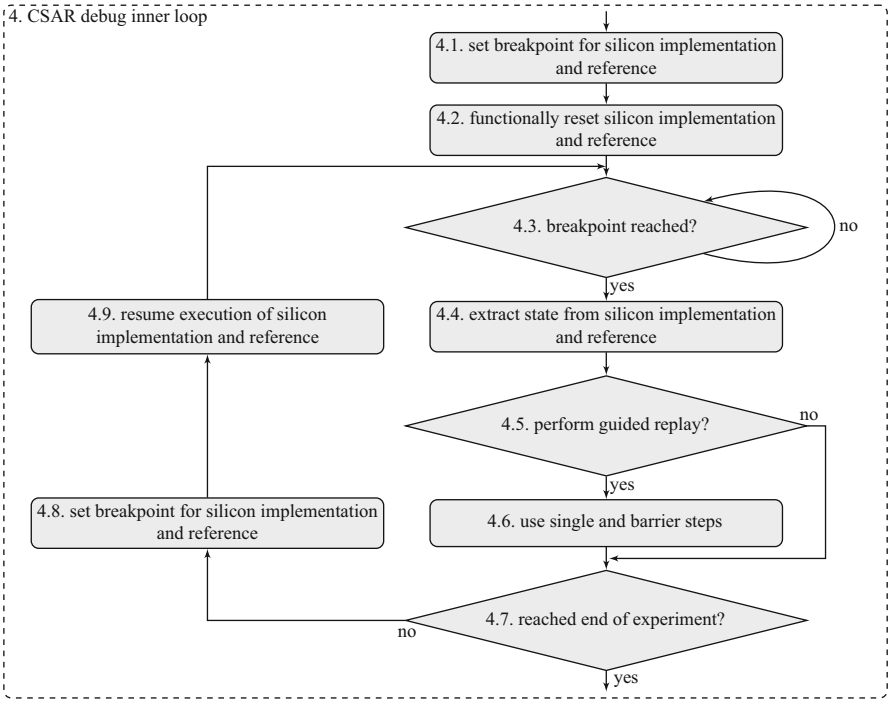


Fig. 4.6 Extended post-silicon debug process inner loop, adapted from [3]

The process shown in Figs. 4.5 and 4.6 can be used to force the execution of a silicon implementation of an SOC along a particular, feasible state path, provided (1) that the execution control takes place at the handshake abstraction level or above, (2) that this execution control is executed from reset, and (3) that this execution control is used to step the SOC one communication link at a time. Otherwise, stepping at the clock-cycle abstraction level or multiple communication links at a time, yields the problems of potentially sampling meta-stable state, not knowing the exact order in which the links took a step, and potentially extracting a feasible, global state, but not the actual global state.

Being able to deterministically step a silicon implementation of a SOC at the handshake abstraction level or above from reset, and to extract a globally-consistent state after each step mitigates the complicating factors for debugging identified in Chaps. 2 and 3. It generates state space data that can be used with other verification and validation methods that perform state-by-state and/or trace-by-trace comparisons, such as *(bi)simulation* [2] and *emulation*, to determine behavioral equivalence between two implementations.

4.2 CSAR Debug Analysis

We describe in this section our rationale behind the CSAR debug approach and show how it addresses the debug requirements and complicating factors for debugging identified in Chaps. 1, 2, and 3.

We decided that the CSAR debug approach has to address debug requirement DR-1, by providing the debug engineer with full internal state observability. We identified however, with complicating factor CF-1 on page CF-1, that there are physical limitations to the amount of state data that can be transported off-chip in real-time. Complicating factor CF-3 further limits our options, as the silicon cost involved in storing the state data on-chip will make it very difficult for the resulting SOC product to be competitive on the market. This is also related to debug requirement DR-4. Consequently, the CSAR debug approach is a *run/stop-based debug approach*, in which the SOC execution has to be stopped to allow a debug engineer to inspect the internal state for error localization. Because the SOC execution is stopped and its state no longer changing, we can use any SOC interface to extract this state, including a low-speed and low-cost interface, such as the IEEE Std. 1149.1 TAP.

Having to stop the execution of the SOC to provide state access, brings with it the consequences of complicating factor CF-9. We cannot instantaneously stop all clock domains in a GALS SOC and expect to find a globally-consistent state. If we try this, then we may introduce meta-stability in the state of one or more building blocks, and thereby possibly corrupting this state. This would severely limit the usefulness of the full internal observability. Instead, the distribution of the stop event has to be allowed to take time for the proper synchronization of the stop request in each on-chip clock domain. We reduce the impact of the delay incurred by this event distribution on the states of the SOC modules by first functionally stalling the execution of all modules. We have seen in Sect. 3.1.3 that the SOC modules use handshake protocols to communicate with each other. These communication protocols require that these modules have at least one stall state in their building blocks, in which they wait until the module on the other end of the communication link indicates that they can proceed. In the CSAR debug approach, we take control over the handshake signals involved in these communication protocols, to force both parties involved in a communication to remain in their stall state(s), thereby functionally stalling their execution. By controlling the handshakes on every communication link in the SOC, we can functionally stall the execution of the SOC. This ensures that we sample locally and globally-consistent states of the modules afterwards, even when the synchronization of the clock stop event takes more time in one clock domain than in another.

We furthermore need on-chip monitors to determine when to stall the SOC execution, and a fast and scalable EDI to distribute the stop signal to our PSI. Overall, the focus of the CSAR debug approach on the on-chip communication makes it a *communication-centric debug approach*.

Once the SOC execution has been functionally stalled and the functional clocks have been stopped, we have time to extract the state in its building blocks. For this,

Table 4.1 Coverage of debug requirements and complicating factors by the CSAR debug approach

Debug requirements and complicating factors for debugging		Page	C	S	A	R/S
DR-1	Internal observability	19		+		+
DR-2	Execution control	19	+	+	+	+
DR-3	Non intrusiveness	19	+/−	+/−	+	−
DR-4	Efficient implementation and use	19	+	+	+	+
CF-1	Limited spatial observability and controllability	34		+		
CF-2	Limited temporal observability and controllability	34	+	+	+	+
CF-3	Implementation cost	35	−	−	+/−	−
CF-4	Undefined substate and outputs	35		+	+	
CF-5	State comparison	37	+	+	+	
CF-6	Transient errors	37	+		+	
CF-7	Non determinism at clock cycle, handshake, and transaction level	61	+		+	
CF-8	Uncertain errors	63	+	+		+
CF-9	No instantaneous, distributed, global actions	65	+		+	

C communication-centric, *S* scan-based, *A* abstraction-based, *R/S* run/stop-based

we reuse the manufacturing test scan chains, the TAP, and its controller, because these provide full state observability at a minimum additional silicon area cost. We can subsequently apply structural, data, and behavioral abstraction techniques to the state that we have extracted, to address complicating factors CF-4 and CF-4. This use of abstraction makes the CSAR approach an *abstraction-based debug approach*.

The CSAR debug approach furthermore addresses debug requirement DR-2 by providing a debug engineer with (1) communication-centric control through the configuration of the on-chip PSIs and (2) state controllability via the scan chains. The use of temporal abstraction helps address the non-determinism, identified as complicating factor CF-7. Using the on-chip communication monitors, the EDI, and the PSIs, we can stop the execution of the SOC at multiple points in time. We can even take complete control over the on-chip communication, by using our guided (re)play functionality. This functionality improves the temporal observability (refer to complicating factor CF-2) and increases the probability of detecting the effect of a transient error on the SOC state, before this effect disappears (refer to complicating factor CF-6). Additionally, this functionality provides improved temporal and spatial controllability, which improves our options to detect uncertain errors (refer to complicating factor CF-8). As we see later in this book, we have taken care to ensure that the supporting debug infrastructure is both non-intrusive (debug requirement DR-3) and efficient in its implementation and use, where possible (debug requirement DR-4).

The coverage of the debug requirements and complicating factors for debugging by our CSAR debug approach and its infrastructure is summarized in Table 4.1. In this table, a positive contribution is indicated with an "+", and a negative contribution with an "−". A cell is left empty, when an aspect of our approach does not make a contribution to that requirement or complicating factor. We can see that overall the four main aspects of the CSAR debug approach effectively address both the four debug requirements and the nine complicating factors.

There are however also five limitations to the debug functionality provided by the CSAR debug approach and its infrastructure, that cause it to make a negative contribution on some points.

A first limitation of our approach is related to complicating factor CF-9. With the CSAR debug infrastructure, it is still not possible to instantaneously stop the SOC execution. We do try to distribute the stop debug event using our EDI as quickly as possible. This however still takes a certain amount of time. If the debug event has to be communicated across clock domain boundaries, this amount of time may also vary from one debug experiment to another (due to complicating factor CF-7). We abstract away from the difference in duration at the clock cycle level by increasing the granularity at which we stop the communication, i.e., to the data element, message, and transaction levels. This does however not prevent in a worst-case that the debug event arrives at a PSI just before the start of one of these elements in one debug experiment, and just after the start of one of these elements in another. In our debug approach, the variation between debug experiments therefore exhibits itself less frequently, but at a large temporal granularity, where it can more easily be recognized and possibly handled.

A second limitation of the CSAR debug approach is that our control of the handshake control signals stops the communication between two SOC building blocks, but it does not necessarily stall all computation inside the individual building blocks. We explained the operation of the handshake protocols in Chap. 3 using super-states to indicate that some local computation may still occur, while a block waits for the building block on the other side of the communication link to indicate that it can exit its super-state. The effect of this local computation during these stalled handshakes cause us to extract one of the states in such a super-state in otherwise identical debug experiments. This is a local variation that can be handled by combining our communication-centric approach with a computation-centric approach. We come back to this phenomena in Chap. 9.

A third limitation, related to debug requirement DR-3, is that we only have full internal observability after we have stopped the execution of the SOC and co-opt the scan chains. Stopping the SOC execution is very intrusive. Resuming the execution of the SOC afterwards at exactly the same point as it was stopped is not possible, as we cannot ensure that the clock relations at the point that we resume the execution of the SOC are exactly the same as they were when we stopped its execution. Errors that would have been activated if we had not stopped the SOC execution, may therefore no longer be activated. The CSAR debug infrastructure therefore lends itself best for either stopping the execution of the SOC after the error has been activated, or for setting up the correct conditions for an error to be activated after the SOC execution is resumed, by for example setting a specific global SOC state.

A fourth limitation, related to debug requirement DR-4 and complicating factor CF-3, is that our on-chip debug infrastructure still costs silicon area, even though we have implemented our on-chip debug infrastructure as efficiently as possible. We provide more detail of its area cost in Chap. 8. A large contributor to this area cost are the communication monitors, because they have to scale with the size of the data elements on the on-chip communication links and they have to be configurable, to

allow the generation of debug events at sufficient points in time during the execution of the SOC and thereby improve the temporal observability and controllability (refer to CF-2).

A fifth limitation is that our debug infrastructure can only support a debug engineer in detecting an error that (1) manifests itself in the SOC state, and (2) remains present in the SOC state until a next communication breakpoint. This excludes errors that do not corrupt the state of a flip-flop or embedded memory, such as an excessive power consumption or electromagnetic emission.

4.3 CSAR Debug Infrastructure Requirements

The CSAR debug approach imposes requirements on the on-chip and off-chip debug infrastructure. In this section, we review and formulate these requirements.

4.3.1 Communication-Centric Debug

A communication-centric debug approach imposes three high-level debug infrastructure requirements on the silicon implementation of an SOC and its off-chip debugger software: (1) non-intrusive monitoring of the on-chip communication, (2) controlling the communication at multiple levels of temporal abstraction, and (3) a fast and scalable distribution of debug events. We detail these requirements below.

4.3.1.1 Monitoring the Communication

The on-chip monitors need to be able to effectively and efficiently divide the execution of the SOC into progressively-smaller execution steps, to facilitate the iterative spatial and temporal localization of the root cause of a failure. To this end, these monitors generate one or more debug events when the SOC reaches one or more specific points in its execution, or when it completes an execution step at a certain level of behavioral or temporal abstraction [3].

Optionally, a monitor has to be able to calculate a checksum value based on the communication (meta)data it observes. Communication checksums are useful, for example to help diagnose cross-talk or other signal integrity issues. These issues may cause a checksum at one location to differ from the checksum calculated at another location along the same communication path, or from a checksum value calculated off-line using an abstract executable model. The comparison of checksum values enables the isolation of a suspect section on a communication path, thereby speeding up the debug process. A checksum value can also help to (1) distinguish between the actual state path taken by the (2) and other feasible state paths, and (3) detect the activation of an error on the actual state path.

These features help guide a debug engineer when debugging. In particular, the evidence of the existence of a transient error may be preserved in a checksum value until this value can be inspected by the debug engineer. Consider as an example the two feasible state paths that are indicated in Figs. 3.28 and 3.29 with the thick lines. The monitors on the communication links between the producer and the shared memory, and between the consumer and the shared memory will calculate different checksum values for both paths, because the consumer reads from the shared memory only once in Fig. 3.28, and reads for the shared memory twice in Fig. 3.29. We can determine the number of times the consumer actually reads from the shared memory in the silicon SOC implementation when a failure occurs, by comparing the checksum values after we stop the SOC to a reference implementation. This knowledge can subsequently be used during the guided replay process to, for example, make the consumer poll the shared memory that number of times, thereby increasing the chances of reproducing the failure.

The communication monitors need to be connected to the EDI to be able to correlate global debug events with local communication events (refer to Fig. 4.1). Each monitor has to be configurable as the required debug event to control the execution of the SOC on to localize a fault is not known in advance. These observations lead us to formulate debug infrastructure requirements IR-1 to IR-4 for monitoring the on-chip communication.

IR-1: Communication Monitoring *The debug infrastructure has to include monitors that non-intrusively observe the communication inside the SOC and generate debug events when specific communication elements are detected.*

IR-2: Event Generation *The communication monitor has to be able to receive debug events to correlate them with local communication conditions for the generation of another debug event.*

IR-3: Checksum Generation *The communication monitor has to be able to calculate a checksum value for the communication (meta)data it observes, to help debug transient and uncertain errors.*

IR-4: Debug Access to the Monitors *The debug monitor has to be configurable from the off-chip debugger software to change the communication condition it generates a communication debug event and/or a checksum value for.*

4.3.1.2 Controlling the Communication

The communication between two building blocks in a GALS SOC makes use of a handshake-based protocol to avoid meta-stability problems (refer to Sect. 3.1.3). In a communication-centric debug approach, we control the communication handshakes between the building blocks. Specifically, we need control over the validation and acceptance of communication elements. The PSI therefore needs to be able to prevent the completion of specific handshakes between building blocks either unconditionally, or after it has received a debug event from the EDI. It thereby stalls

the hardware execution thread in these building blocks. Given the general structure of SOC communication protocols, a PSI has to be able to control the communication at three different granularities: (1) elements, (2) messages, and (3) transactions.

Next to stalling the communication at these three granularities, we need to be able to selectively allow the communication between certain building blocks to resume at each of these granularities, while the communication between other building blocks remain stalled. This may be done per link when single-stepping, or per set of links when barrier-stepping.

The debug engineer needs access to the status of each communication link during these control operations. The access mechanism via the DCSI may not be fast enough to track every change in the status of the communication link in real-time. Therefore the PSI itself has to maintain a record of the relevant status changes, so that it can be queried by the DCSI. Additionally, PSIs have to be connected to the EDI to be able to respond to global debug events with appropriate local communication debug actions, and to generate a debug event when the status of the local communication link changes.

These observations lead us to formulate debug infrastructure requirements IR-1 to IR-4 for controlling the communication.

IR-5: Communication Control *The debug infrastructure has to include PSIs to control the communication between the on-chip building blocks.*

IR-6: Communication Control Granularity *The PSI has to support the granularity of elements, messages, and transactions for stalling and stepping the communication.*

IR-7: Communication Stop Condition *The PSI has to support stalling the communication on a link, either unconditionally or when a debug event is received from the EDI.*

IR-8: Communication Link Status *The PSI has to allow the status of the communication link to be queried via the DCSI.*

IR-9: Debug Access to the Protocol-Specific Instruments *The PSI has to be configurable from the off-chip debugger software with respect to the granularity at which the communication is stalled and stepped and the stop condition.*

4.3.1.3 Distributing the Debug Events

The EDI shown in Fig. 4.1 is responsible for globally distributing debug events. It therefore has to be connected to all monitors and PSI. The EDI has to be as fast as possible to approximate an instantaneous communication of debug events as close as possible. The implementation of this EDI has to be scalable to adapt to the specific number of monitors and PSI in a GALS SOC. In addition, the EDI implementation has to be able to cross multiple clock and power domain boundaries. These observations lead us to formulate debug infrastructure requirements IR-10 to IR-14 for the EDI.

IR-10: Fast Event Distribution *The EDI has to distribute debug events as quickly as possible.*

IR-11: Scalable Event Distribution *The EDI has to be scalable at design time in the number of event sources and event destinations and in the number of simultaneous debug events it can support.*

IR-12: Layout-Friendly Event Distribution *The EDI has to be layout friendly to simplify its inclusion in a GALS SOC implementation, which includes the ability for the EDI to cross clock and power domain boundaries.*

IR-13: Control over the Event Forwarding *The EDI has to support the configurable routing of debug events from an event source to a subset of event destinations to reduce its wiring cost.*

IR-14: Debug Access to the Event Distribution Interconnect *The event distribution interconnect has to be configurable from the off-chip debugger software.*

4.3.2 Scan-Based Debug

A scan-based debug approach imposes debug infrastructure requirements IR-15 to IR-18 on the silicon implementation of the SOC and its off-chip debugger software.

IR-15: Debug Access to the State of the Flip-Flops *The debug infrastructure has to provide access to the content of the manufacturing test scan chains from the off-chip debugger software.*

IR-16: Debug Access to the State of the Memories *The debug infrastructure has to provide access to the content of the embedded memories from the off-chip debugger software.*

IR-17: Debug Control over the SOC and Module Operating Mode *The debug infrastructure has to allow switching the operating mode of the SOC and its modules from functional mode to the debug scan and debug normal modes, and back, from the off-chip debugger software.*

IR-18: Debug Control over the On-Chip Clocks *The debug infrastructure has to provide control over the on-chip clocks to allow functional clocks to be turned off and on, and switched to the TCK signal of the TAP for the controlled shifting of the scan chains for debug.*

4.3.3 Abstraction-Based Debug

An abstraction-based debug approach imposes debug infrastructure requirements IR-15 to IR-22 for structural, data, behavioral and temporal abstraction on the silicon implementation of the SOC and its off-chip debugger software.

IR-19: Structural Abstraction *The debug infrastructure has to support the structural abstraction from the state data retrieved from the scan chains in the SOC to the state of registers, memories, building blocks, and modules.*

IR-20: Data Abstraction *The debug infrastructure has to support the data abstraction to substitute multi-bit, binary values with higher-level data values, such as enumerated types, states, hardware and software FIFOs, and software data types.*

IR-21: Behavioral Abstraction *The debug infrastructure has to support behavioral abstraction of the state of individual building blocks, via the state of modules, to the state of the software applications in an SOC use case.*

IR-22: Temporal Abstraction *The debug infrastructure has to support temporal abstraction from the clock-cycle level to the handshake, message, and transaction levels in hardware, and abstractions used in software communication (e.g. C-HEAP tokens).*

IR-23: SOC Environment Abstraction *The debug infrastructure has to support the abstraction from the specific access implementation of an SOC environment (refer to Fig. 1.11 on p. 17), and provide instead a generic interface.*

4.3.4 Run/Stop-Based Debug

A run/stop-based debug approach imposes debug infrastructure requirements IR-24 and IR-25 on the silicon implementation of the SOC and its off-chip debugger software.

IR-24: Debug Control over the Functional Reset *The debug infrastructure has to provide access to the functional reset from the off-chip debugger software.*

IR-25: Automation of the Debug Experiments *The debug infrastructure has to provide access to all on-chip debug modules from the TAP for configuration (pseudo-static), status (sampled), and control (possibly dynamic) from the off-chip debugger software.*

4.3.5 Debug Requirements Overview

Table 4.2 presents for each debug infrastructure requirement an overview of the sections in this book in which this requirement is addressed. In Chap. 5, we describe how these debug infrastructure requirements are supported by the on-chip CSAR debug hardware architecture. We detail in Chap. 6 how this support can be automatically added to an SOC. In Chap. 7 we discuss the implementation of off-chip debugger software that can subsequently take advantage of the on-chip debug hardware to debug the SOC using the CSAR debug approach.

Table 4.2 Overview of CSAR infrastructure requirements

Infrastructure requirement	Functionality	Page	On-chip hardware	Off-chip software
IR-1	Communication monitoring	84	5.3	–
IR-2	Event generation	84	5.3	–
IR-3	Checksum generation	84	5.3	–
IR-4	Debug access to the monitors	85	5.2, 5.3	7.2
IR-5	Communication control	85	5.4	–
IR-6	Communication control granularity	85	5.4	–
IR-7	Communication stop condition	85	5.4	–
IR-8	Communication link status	86	5.4	–
IR-9	Debug access to the protocol-specific instruments	86	5.2, 5.4	7.2
IR-10	Fast event distribution	86	5.5	–
IR-11	Scalable event distribution	86	5.5	–
IR-12	Layout-friendly event distribution	86	5.5	–
IR-13	Control over the event forwarding	86	5.5	–
IR-14	Debug access to the event distribution interconnect	86	5.2, 5.5	7.2
IR-15	Debug access to the state of the flip-flops	87	5.2, 5.7	7.2
IR-16	Debug access to the state of the memories	87	5.2, 5.7	7.2
IR-17	Debug control over the operating mode	87	5.2	7.2
IR-18	Debug control over the on-chip clocks	87	5.2, 5.8	7.2
IR-19	Structural abstraction	87	–	7.3, 7.5
IR-20	Data abstraction	87	–	7.3, 7.5
IR-21	Behavioral abstraction	87	–	7.3, 7.5
IR-22	Temporal abstraction	88	–	7.3, 7.5
IR-23	SOC environment abstraction	88	–	7.2
IR-24	Debug control over the functional reset	88	5.2, 5.8	7.2
IR-25	Automation of the debug experiments	88	–	7.4

4.4 Summary

In this chapter, we analyzed the debug requirements and complicating factors for debugging a GALS SOC, as identified in Chaps. 1–3, and introduced the CSAR debug approach. We subsequently described its communication-based, scan-based, abstraction-based, and run/stop-based aspects in more detail, and showed how these aspects help to address the identified debug requirements and complicating factors. During this analysis, we identified 25 debug infrastructure requirements that need to be met to support the debugging of a GALS SOC using the CSAR debug approach and off-chip debugger software.

References

1. IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture - IEEE Std 1149.1-2001*. IEEE Press, 2013.
2. R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.
3. Bart Vermeulen and Kees Goossens. Debugging Multi-Core Systems on Chip. In George Kornaros, editor, *Multi-Core Embedded Systems*, chapter 5, pages 153–198. CRC Press/Taylor & Francis Group, 2010.

Chapter 5
On-Chip Debug Architecture

Abstract In this chapter we present the on-chip CSAR debug architecture and its debug components. We start with an overview of the on-chip debug architecture in Sect. 5.1. In Sect. 5.2, we describe the DCSI that we use to control the CSAR on-chip debug functionality. We present details on the CSAR communication monitor implementation in Sect. 5.3, the PSI implementation in Sect. 5.4 and the EDI implementation in Sect. 5.5. We use a debug wrapper to group a module with its CSAR hardware components. This wrapper is described in Sect. 5.6. Our test wrapper is detailed in Sect. 5.7. The CSAR clock and reset generation module that provides debug clock and reset control is described in Sect. 5.8. We conclude this chapter with a summary in Sect. 5.9.

5.1 Overview

Any module-based SOC with modern interconnects can be made to support the CSAR debug approach. Figure 5.1 shows an example, generic top-level block diagram of such an SOC, including the hardware components it needs to support the CSAR debug approach. Please note that for clarity most block diagrams do not explicitly show the clock and reset signals. Each debug component uses the clock and reset signal of the module in which it is instantiated, unless explicitly stated otherwise.

This top-level module integrates the functional modules of the SOC and our CSAR hardware components. At the top of the block diagram we see three modules, e.g., processors, that act as masters on the on-chip communication interconnect, called "master 1", "master 2", and "master 3". Below these modules we see the communication interconnect. A NoC is used in this SOC, comprising buses, shells, network interfaces (NIs), and router(s) [5]. At the bottom of the block diagram we see two modules, e.g., memories, that act as slaves to the on-chip communication interconnect, called "slave 1" and "slave 2".

Support for the CSAR debug approach is implemented by adding a number of CSAR hardware components to this top-level module. A *communication monitor* can be connected to the communication link between any two modules to *observe the communication between* them. The implementation of these monitors is described in more detail in Sect. 5.3. A *PSI* can be inserted in the data communication link between two modules, to *control the communication between* the modules in response to events generated by other CSAR hardware components. The implementation of

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip*, Embedded Systems, DOI 10.1007/978-3-319-06242-6_5,

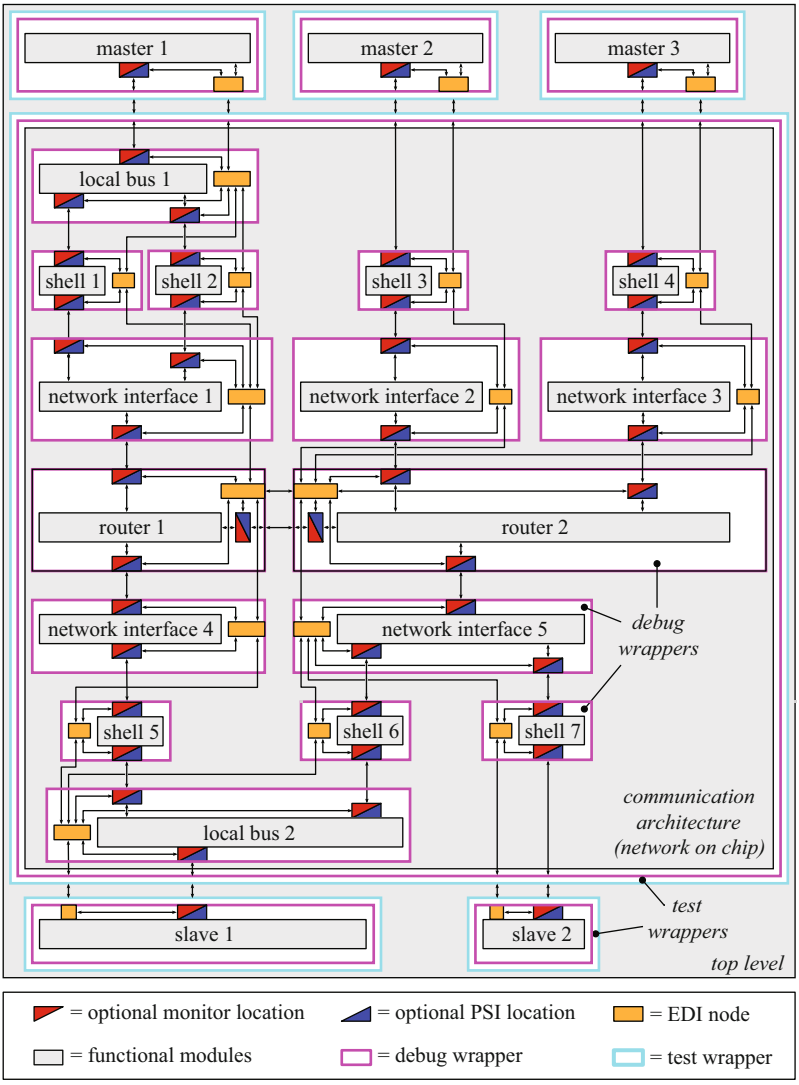


Fig. 5.1 Overview of the on-chip CSAR debug architecture

these PSIs is described in more detail in Sect. 5.4. A fast and scalable *EDI* distributes the debug events from the event sources to the event destinations using a multi-hop broadcast network of EDI nodes. Figure 5.1 shows how this EDI connects to all functional modules, monitors, and PSIs in the top-level module. The implementation of the EDI is explained in more detail in Sect. 5.5. The debug components around a module are grouped using a so-called *debug wrapper* to simplify subsequent design steps. The resulting wrapped module can be synthesized and scan-inserted. Example debug wrappers are shown in Fig. 5.1 in white.

Note how Fig. 5.1 shows the possibility of implementing two debug monitors and two PSIs on a single connection between two modules, one on the initiator side and one on the target side. This is redundant and therefore unnecessary, as both monitors and both PSIs respectively observe and control the same communication link. Using a single monitor and/or PSI is sufficient. Figure 5.1 is intended to show the freedom a design team has in the grouping of the CSAR hardware components with either module. The instantiation of a monitor and/or PSI inside a debug wrapper is optional for all modules. The instantiation of an EDI node is mandatory inside the debug wrapper around the building blocks of the communication interconnect, and optional for the functional modules. We lose the scalability of the EDI when the implementation of the EDI node is made optional for the debug wrappers around the building blocks in the communication interconnect.

Figure 5.1 shows that the NoC itself can also be considered as a module and subsequently wrapped for debugging. The implementation of debug wrappers is explained in more detail in Sect. 5.6. Each module is wrapped with a *test wrapper*, to provide access to its internal scan chains for manufacturing test and for debugging. Details on the implementation of the test wrapper are presented in Sect. 5.7. Test wrappers are shown in Fig. 5.1 as the light blue collar around the modules. Note how in Fig. 5.1 the building blocks inside the NoC have been wrapped for debugging but not for test. The NoC groups these wrapped building blocks and subsequently becomes itself a module, which is wrapped for debugging, synthesized, scan-inserted and subsequently wrapped for manufacturing test. This is just one possible approach. Alternative groupings and debug wrapper orderings are also valid, as long as the outer-most wrapper is a test wrapper. This outer-most test wrapper is required to support manufacturing test. The *top-level module* groups all testable modules. This top-level module is instantiated in a *chip-level module*. An example chip-level module is shown in Fig. 5.2. For clarity, Fig. 5.2 no longer shows the functional interconnect between the wrapped modules, but treats it as a normal module. Figure 5.2 provides a high-level overview of the DCSI. This DCSI consists of an IEEE Std 1149.1 *TAP* [6] and its *associated controller*, a chain of TPRs through the modules, a chip-specific *CRGU*, and a *global TCB*. The off-chip debugger software has access to the configuration and status of all on-chip debug components via the TPRs in the DCSI. Through the DCSIs, the off-chip debugger software can control the execution of the SOC and access the functional state of the SOC at interesting points during its execution. Details on the implementation of the TPRs and the DCSI are provided in Sect. 5.2. The CRGU generates the on-chip clock and reset signals in both functional and debug scan mode, and allows the off-chip debugger software to control them through the TAP. Details on the operation and implementation of the CRGU are provided in Sect. 5.8. The global TCB controls the operating modes of the top-level module. More details on the global TCB are provided in Sect. 5.2.3. In *functional mode*, the DCSI can access the TPRs embedded in the CSAR debug components inside each module. In *debug scan mode*, these TPRs become part of the manufacturing-test scan chains. The TPRs are made scannable in the test and debug modes to increase the testability of the SOC. From a manufacturing-test point of view, the TPRs are just functional logic that needs to be tested for manufacturing defects. This mode-dependent access method for the

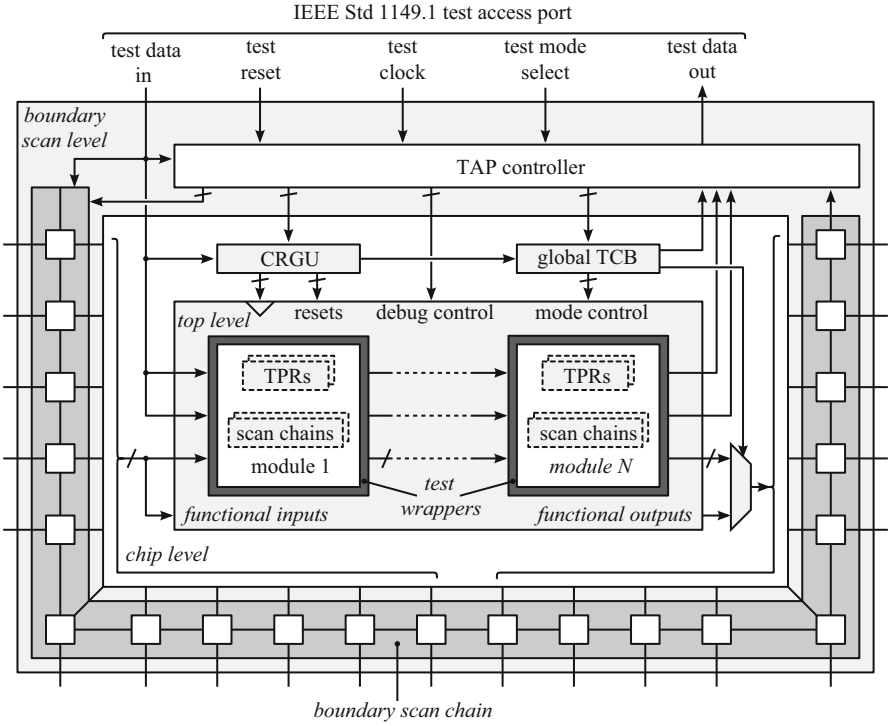


Fig. 5.2 Overview of the CSAR debug control and status interconnect

TPRs does require some consideration in the off-chip debugger software. We discuss this in Sect. 7.2.4, when we describe how we handle this. The module-level scan chains are all concatenated at the top-level. In *manufacturing test mode*, their inputs are connected to a set of input pins of the SOC and their outputs are multiplexed onto a set of output pins of the SOC. The boundary-scan chain that surrounds the chip-level module allows observability and controllability of the state of the I/Os of the chip-level module to facilitate board-level manufacturing test. Its function is extensively documented in [6] and is therefore not detailed here further.

5.2 Debug Control and Status Interconnect

The off-chip debugger software, shown in Fig. 1.11 and described in Chap. 7, can access the configuration and status of all CSAR hardware components through the DCSI. This access is independent from and transparent to the functional execution of the SOC to satisfy debug requirement DR-3. It has furthermore been designed to meet debug infrastructure requirements IR-4, IR-9, IR-14, IR-15, IR-16, IR-17, IR-18, and IR-24, identified in Chap. 4.

The DCSI consists of (1) the TAP and its associated controller, (2) two TCBs, and (3) one or more TPRs. We describe the TAP controller in Sect. 5.2.1. We then detail the architecture of a generic TCB and present the global TCB as a specific instantiation of this architecture in respectively Sects. 5.2.2 and 5.2.3. Another instantiation of this architecture is discussed in Sect. 5.8.2 as part of the CRGU. We present the architecture of a generic TPR in Sect. 5.2.4. Custom TPR instantiations are used in all CSAR debug components to configure, control, and query their functionality. A specific TPR instantiation is described in Sect. 5.2.5, when we detail an example TAP–DTL bridge, which allows access to the functional communication interconnect through the TAP.

5.2.1 Test Access Port and Associated Controller

The CSAR debug architecture is controlled by the off-chip debugger software via the IEEE Std 1149.1 TAP [6]. Using the TAP has four distinct benefits over using other ports. First, it is a low-cost interface, as it is already implemented for board-level manufacturing test and thereby removes the need to add dedicated debug I/O pins, which increase silicon area and packaging costs. Second, it is well-supported by commercial, off-the-shelf (COTS) test and debug equipment. Third, its use is non-intrusive in the sense that there is no need to share one of the functional ports on the SOC to gain access to the on-chip debug architecture, thereby avoiding influencing the execution of the SOC. Fourth, it is very often also available in the product environment.

The TAP is essentially a serial port, which connects to an on-chip controller. This controller allows access to on-chip serial registers. The IEEE Std 1149.1 allows this controller to be extended with user-defined serial registers to support additional test and debug functionality. We use this extensibility to control and query our debug components, and thereby control and observe the functional execution of the SOC. The CSAR TAP controller interfaces with a number of other hardware components, as shown in Fig. 5.2.

Figure 5.3 shows a block diagram of the TAP controller used in the case study in Chap. 8. Its TAP is shown on the left-hand side of Fig. 5.3. On the right-hand side of Fig. 5.3, we see the control signals to and status signals from the other CSAR debug components. These components are the boundary scan chain, the TCBs, the TPRs, the CRGU, and the test wrappers. We refer to the names of these signals in the remaining sections in this chapter, when we detail the behavior of the associated CSAR components. We then also show the timing relations between these signals and the state of the TAP controller's FSM, and the resulting behavior of each CSAR debug component.

The IEEE Std 1149.1 TAP is a dedicated, five-pin test and debug interface, comprising (1) a TCK input pin, (2) a test data input (TDI) pin, (3) a test data output (TDO) pin, (4) a test mode select (TMS) input pin, and (5) an optional test reset (TRSTN) input pin. The TCK and TMS signals control a standardized, 16-state FSM. The STG

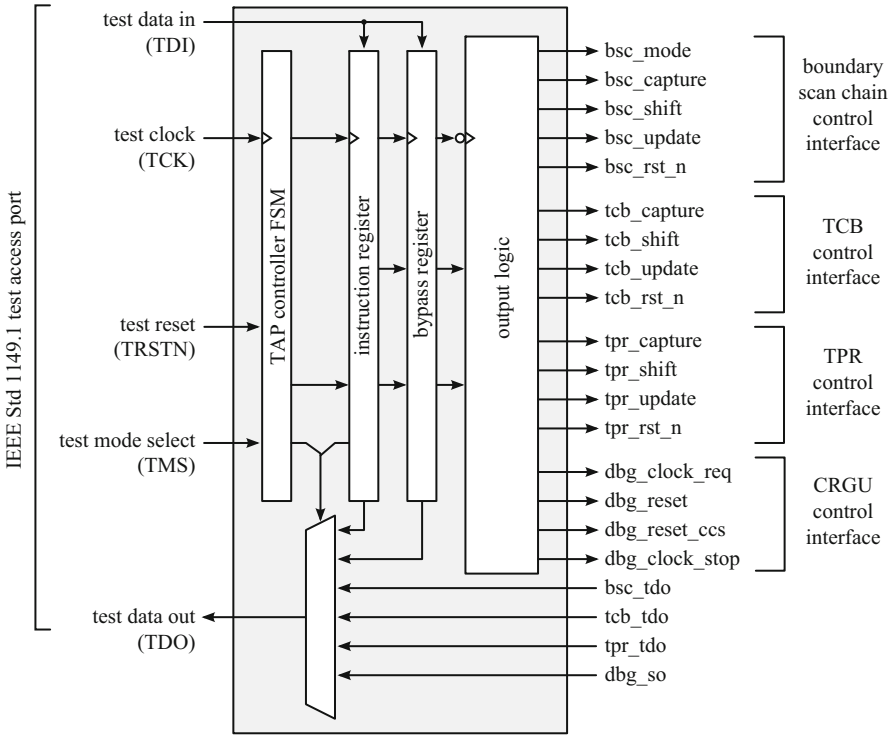


Fig. 5.3 Block diagram of a CSAR TAP controller

of this FSM is shown in Fig. 5.4, and based on [6]. The label near each directed edge refers to the value of the TMS signal. The FSM transitions to the associated next state on the *rising edge* of the TCK signal when the TMS signal has this value. The TAP controller contains an instruction register. This instruction register can be serially loaded with a new TAP instruction via the TDI pin in the "Shift-IR" (SIR) state of the FSM. During this operation, the previous content of the instruction register is unloaded via the TDO pin. When the new instruction is activated in the "Update-IR" (UIR) state, the TAP controller selects the associated serial data register for access via the TDI and TDO pins in the "Shift-DR" (SDR) state of the FSM. This access mechanism is fully specified in the IEEE Std 1149.1 [6].

In this book, we refer to the instruction that has been activated in the instruction register of the TAP controller as the *active TAP instruction*. By adding specific user-defined instructions and associated data registers, we enable control over the on-chip debug functionality. Table 5.1 lists the mandatory IEEE Std 1149.1 and user-defined CSAR instructions that are implemented by the CSAR TAP controller to support the CSAR debug approach.

This table also provides a short description of the purpose of each instruction and a reference to a section in this book or another source, which contains more details.

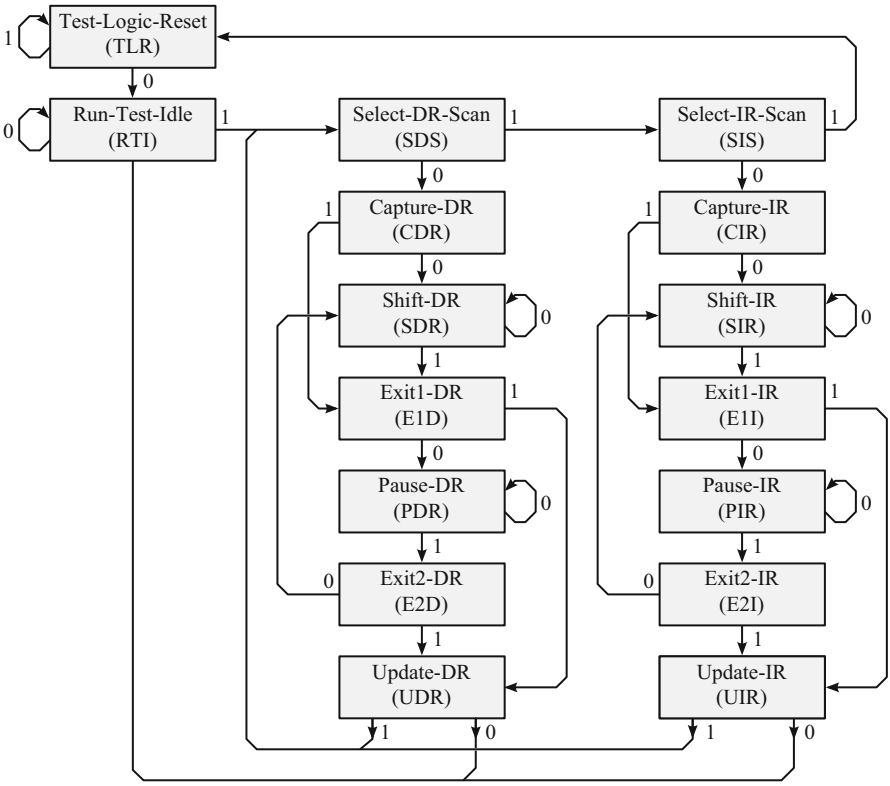


Fig. 5.4 State machine of the IEEE Std 1149.1 TAP FSM, based on [6]

Table 5.1 Example CSAR TAP controller instructions

Name	Purpose	Reference/Section(s)
BYPASS[a]	Access the bypass register	[6]
EXTEST[a]	Access the boundary scan register	[6]
PRELOAD[a]	Access the boundary scan register	[6]
SAMPLE[a]	Access the boundary scan register	[6]
PROGRAM_TCBS	Access the TCBs	5.2.2, 5.2.3, and 5.8.2
PROGRAM_TPRS	Access the TPRs	5.2.4, 5.3, 5.4, and 5.5
DBG_SCAN	Access the debug scan chain	5.8
DBG_RESET_CCS	Reset the clock control slices	5.8
DBG_CLOCK_STOP	Stop the functional clocks	5.8
DBG_RESET	Assert reset	5.8

[a] Mandatory IEEE Std 1149.1 instruction

It is important to note that the CSAR TAP controller (de)asserts all its control output signals on a falling edge on the TCK signal (see the inversion on the clock input of the output logic in Fig. 5.3). The other CSAR components sample these control signals on (the detection of) a rising edge on the TCK signal. In response

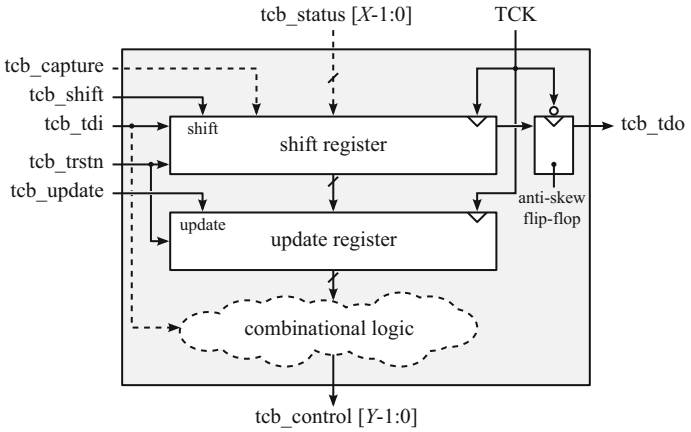


Fig. 5.5 Block diagram of the generic CSAR test control block

to the TAP control signals, the CSAR components update their data output signals also on (the detection of) a falling edge on the TCK signal. The TAP controller and possibly successor components in a serial chain sample these data outputs on (the detection of) a rising edge on the TCK signal. The TAP controller subsequently updates its TDO output on a falling edge on the TCK signal.

Because we can externally control the clock period of the TCK signal and thereby the time between the rising and the falling edges of the TCK signal, we can ensure that all clock domains in the GALS SOC have enough time to react to each clock edge on the TCK signal before another edge is issued. This guarantees the correct sampling of the control and data signals between debug components in different clock domains by design and makes the DCSI scalable and layout-friendly.

5.2.2 Test Control Block

There are two TCBs in the CSAR debug architecture; a TCB inside the CRGU and the global TCB. These TCBs are concatenated into a TCB chain and connected to the TAP controller as a user-defined data register (refer to Fig. 5.2). A TCB can be serially loaded from the TAP with a new mode without affecting its current mode. This new mode can be activated instantaneously after it has been completely loaded. Figure 5.5 shows a block diagram of the generic TCB architecture. The states of the shift and update registers are asynchronously reset on the assertion of the "tcb_trstn" signal. At the chip-level, this signal is directly connected to the TRSTN of the TAP. These registers need to be asynchronously reset, because it is necessary to reset the operating mode of the SOC even when the TCK clock signal is not active.

Data has to be loaded into the shift register to change the outputs of a TCB. This data is afterwards copied to the update register and output via the combinational

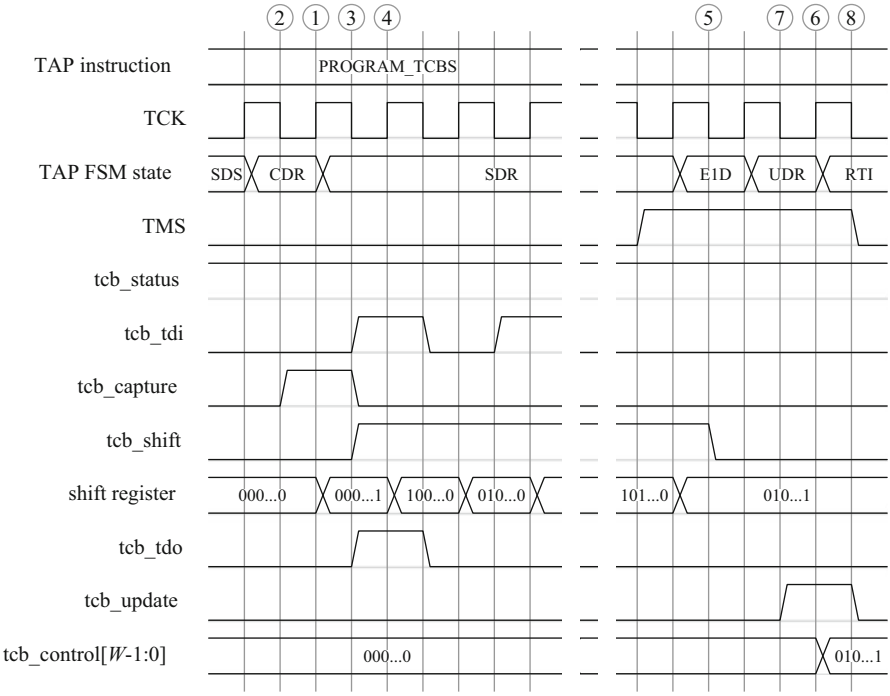


Fig. 5.6 Timing diagram for configuring and querying a TCB under TAP control

logic. The use of a separate update register allows the content of the shift register to be modified without changing the control outputs of the TCB. We call this behavior *silent shifting*.

Figure 5.6 shows a timing diagram of loading a bit pattern into the TCB under control of the TAP controller. A pre-condition for controlling a TCB is that the `PROGRAM_TCBS` instruction is active in the TAP controller. When the "tcb_capture" signal is asserted, the TCB captures the values of the status signals of the surrounding logic "tcb_status" into (part of) its shift register on a rising edge on the TCK signal (①). In the example shown in Fig. 5.6, the value of the single-bit "tcb_status" signal is captured in the least significant bit (LSB) of the shift register. The "tcb_capture" signal is only asserted in the "Capture-DR" (CDR) state of the TAP controller (②) and deasserted in all other states (e.g., ③, ⑤, and ⑦). This capture functionality is optional for a TCB. When the "tcb_shift" signal is asserted, a new configuration bit is serially loaded into the shift register via the "tcb_tdi" input on every rising edge on the TCK signal, while on every falling edge of the TCK signal, one bit of the previous configuration is unloaded via the anti-skew flip-flop and the "tcb_tdo" output (④). This flip-flop prevents timing problems when the TCB chain has to cover long distances on the chip.The "tcb_shift" signal is only asserted in the SDR state (e.g., ③) and is deasserted in all other states (e.g., ⑤, ⑦, and ⑧). The "tcb_tdi" input of the first TCB in a TCB chain is connected to the chip-level

TDI pin, as shown in Fig. 5.2. The other TCBs have their "tcb_tdi" input connected to the "tcb_tdo" output of the TCB preceding them in the TCB chain.

When the "tcb_update" signal is asserted, the content of the shift register is copied to the update register on a rising edge on the TCK signal (⑥). The "tcb_update" signal is only asserted when the TAP FSM is in the "Update-DR" (UDR) state (⑦) and deasserted in all other states (e.g., ②, ③, and ⑧). Because of the use of the two distinct states SDR and UDR, the content of the update register is only updated when the content of the shift register is held stable (⑥).

The TCB control outputs "tcb_control" are derived from the content of the update register and optionally from the "tcb_tdi" input signal using combinational logic. This optional combinational logic permits the forwarding of the "tcb_tdi" signal as a test control signal, as shown in Fig. 5.5. This is needed during manufacturing test.

5.2.3 Global Mode Control

The global TCB controls the operating mode of the SOC. We distinguish three main operating modes: (1) the functional mode, (2) the manufacturing test mode, and (3) the debug mode. Table 5.2 shows the register fields of the global TCB with a short description.

Table 5.3 lists the specific (sub)modes of the SOC and the associated values of the fields in the global TCB. We discuss these (sub)modes when we discuss the test wrapper in Sect. 5.7.

The global TCB contains a "disable_tpr_reset" field to disable the reset of the TPRs in the SOC (described in the next section). When set, this prevents the TPRs from being reset when the SOC is functionally reset. This functionality is needed to allow the configuration of the TPRs to persist when we functionally reset the SOC in step 4.2 of the post-silicon debug process shown in Fig. 4.6.

Note that in this book we use register offsets in the descriptions of the TCB and TPR fields. These offsets increment from right to left, i.e., the field with the lowest offset is located closest to the serial output and therefore its value needs to be loaded into the TCB or TPR first.

5.2.4 Test Point Register

The configuration, control, and status query of individual CSAR components is handled by an embedded TPR. These TPRs are concatenated in a TPR chain and connected to the TAP controller as a user-defined data register (refer to Fig. 5.2). Figure 5.7 shows a block diagram of the generic TPR architecture, as used in each CSAR component. A TPR oversamples the TCK signal using the local clock, instead of clocking its internal registers directly with the TCK signal. We take this approach to remove the need to introduce a TCK clock domain in every debug component and

Table 5.2 Output signals of and fields in the global TCB

Name	Width	Offset	Description
test_io	1	9	Test I/O enable 1: Enabled 0: Disabled
mem_se	2	7	Memory scan enable 1-: 1 01: TDI 00: 0
se	2	5	Logic scan enable 1-: 1 01: TDI 00: 0
intest_en	1	4	Internal or external test enable 1: Internal test 0: External test
dbg_tap_access_en	1	3	Scan chain access 1: Serial 0: Parallel
mem_test_en	1	2	Memory test enable 1: Enabled 0: Disabled
logic_test_en	1	1	Logic test enable 1: Enabled 0: Disabled
disable_tpr_reset	1	0	TPR reset disable 1: Reset disabled 0: Reset enabled

module that uses a TPR. Otherwise, this additional clock domain would complicate the design process by requiring that the paths between the functional and TCK clock domains are correctly identified and constrained during synthesis, static timing analysis, and layout. Instead, treating the TCK signal as a data signal and oversampling it avoids these complications, while simplifying the implementation and verification of the interactions between the TPR and its surrounding logic.

A drawback of this approach however is that the frequency of the TCK signal cannot be higher than half the frequency of the slowest, local clock in order to adhere to the sampling theorem [10]. This requirement can be met by externally lowering the TCK clock frequency at the cost of a slower access time. We take this design decision because it simplifies the instrumentation of a design with CSAR components, while the resulting, longer TPR access times are usually not a problem during debug.

The generic TPR architecture in Fig. 5.7 is a highly-customizable, adapted version of the TCB architecture. A TPR consists of up to three registers; (1) a control register, (2) a status register, and (3) an update register. Either the control register, the status register, or both have to be implemented. The control register permits the same control functionality as a TCB, while the optional update register allows *silent shifting*. The status register permits the same observation of the surrounding logic as a TCB.

Table 5.3 Test wrapper (sub)modes

TCB signals							Operating (sub)mode
test_io	mem_se	se	intest_en	dbg_tap_access_en	mem_test_en	logic_test_en	
							Functional modes
0	–	–	–	–	0	0	
0	–	–	–	0	–	0	
							Manufacturing test modes
0	0	–	1	0	1	0	⋆ Memory test normal mode
0	1	–	1	0	1	0	⋆ Memory test shift mode
1	–	0	1	0	0	1	⋆ Logic test normal mode
1	–	1	1	0	0	1	⋆ Logic test shift mode
1	0	0	1	0	1	1	⋆ Logic & memory test normal mode
1	1	1	1	0	1	1	⋆ Logic & memory test shift mode
1	–	0	0	0	0	1	⋆ Module-external test normal mode
1	–	1	0	0	0	1	⋆ Module-external test shift mode
							Debug modes
0	0	–	–	1	1	0	⋆ Memory debug normal mode
0	1	–	–	1	1	0	⋆ Memory debug shift mode
1	–	0	–	1	0	1	⋆ Logic debug normal mode
1	–	1	–	1	0	1	⋆ Logic debug shift mode
1	0	0	–	1	1	1	⋆ Logic & memory debug normal mode
1	1	1	–	1	1	1	⋆ Logic & memory debug shift mode

– don't care value

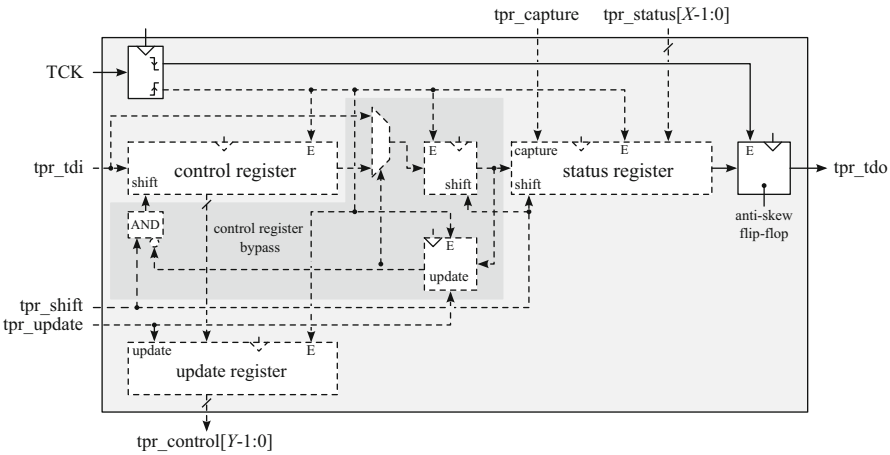


Fig. 5.7 Block diagram of the generic CSAR TPR

The TPR also contains an anti-skew flip-flop to register the serial output of the status register before it is output on the "tpr_tdo" output. The content of this anti-skew flip-flop is updated when a falling edge is detected on the TCK signal.

Next to the many similarities, there are also five important differences between the TCB and the TPR architectures that justify having two separate components. First, the registers inside the TCB are directly clocked by the TCK signal, while the registers inside the TPR are clocked by the local clock of the CSAR component that the TPR is embedded in. For the TPR this is done for the reasons stated before. For a TCB this is done because the timing of its interactions with the rest of the SOC are less timing critical as its control output signals are mainly used to statically configure behavior, such as the operating mode of the SOC. The introduction of the TCK clock domain at the chip-level is not a problem either, as a chip-level designer normally already has to deal with multiple clocks used by the modules.

Second, the TCBs control the operating mode and clock mode of the SOC, including its test and debug mode. Therefore, the TCBs always need to be accessible via the DCSI and cannot be part of the scan-inserted logic. During manufacturing test, the TCBs are tested either implicitly or using a dedicated TCB test. In contrast, standard scan insertion can be performed on all TPRs during synthesis and scan insertion of its parent module. In functional mode, the TPRs are accessible functionally via the DCSI. In manufacturing test mode and in logic debug shift mode, the states of all TPRs are fully observable and controllable via the scan chains.

Third, the optional control register bypass, shown in the middle of Fig. 5.7, permits the control register to be completely bypassed when loading and unloading the TPR, thereby preventing any modification of the content of the control and the update registers. Removing the control register from the serial path between the serial input and output once the TPR is configured, has the advantage of shortening the access time to the status information in this TPR and the other TPRs in the TPR chain.

Fourth, the TCBs have to contain an update register, because we cannot allow the (sub)mode control signals to the test wrappers to toggle in an uncontrolled way. The update register is an optional part of the TPR. It may be possibly to remove the update register from a large number of the TPR control outputs to reduce the amount of silicon area required for its implementation. In that case, we can implement a single update register bit to prevent the propagation of the shift register outputs to the logic around the TPR during its (re)configuration.

Fifth, to facilitate the layout process of the SOC, the TCB is a self-contained module at the chip-level and includes all combinational logic for its control outputs. In contrast, a TPR is embedded in a CSAR component. The cost of a TPR implementation, together with any required combinational logic on its outputs, can therefore be minimized by combining them with the logic of that component.

It can be noted that the functions of the control and the status registers can be combined in a single register to save silicon area, and that the update of the update register can be disabled in the bypass mode instead. However, the resulting double functionality of the bits in the control/status register complicates the design of the off-chip debugger software, so we choose not to use this option.

Figure 5.8 shows an example timing diagram of the configuration and status query of a TPR under TAP control. A pre-condition for controlling a TPR is that the PROGRAM_TPRS instruction is active (refer to Table 5.1). The behavior of a TPR is similar to the behavior of a TCB. When the "tpr_capture" signal is asserted, the TPR captures the values of the "tpr_status" signals from the surrounding logic into its status register on the detection of a rising edge on the TCK signal (①). The TAP controller only asserts the "tpr_capture" signal in the CDR state of the TAP controller (②), and deasserts it in all other states (e.g., ③, ⑤, and ⑦). This capture functionality is optional for a TPR. When the "tpr_shift" signal is asserted and the control register bypass is deactivated, a new configuration bit is serially loaded into the control and status registers via the "tpr_tdi" input every time a rising edge is detected on the TCK signal, while on the detection of every falling edge on the TCK signal, the previous configuration and captured status bits are unloaded via the anti-skew flip-flop and the "tpr_tdo" output (④). This serial (un)load operation does not affect the content of the control register when the control register bypass is activated. The "tpr_shift" signal is asserted when the TAP FSM is in the SDR state of the TAP controller (e.g., ③) and deasserted in all other states (e.g., ②, ⑤, and ⑦). The "tpr_tdi" input of the first TPR in a TPR chain is connected to the chip-level TDI pin, as shown in Fig. 5.2. The other TPRs have their "tpr_tdi" input connected to the "tpr_tdo" output of the TPR preceding them in the TPR chain. When the "tpr_update" signal is asserted, the content of the control register is copied to the update register on the detection of a rising edge on the TCK signal (⑥). The TAP controller asserts the "tpr_update" signal in the UDR state of the TAP controller (⑦) and deasserts it in all other states (e.g., ②, ③, and ⑧). The control outputs of the TPRs are taken from the update register, if implemented, or directly from the control register otherwise.

In the remainder of this chapter, we introduce customized variations of this generic TPR architecture to suit the specific requirements for the debug components in the CSAR hardware architecture. When specifying the register fields of these TPRs, we

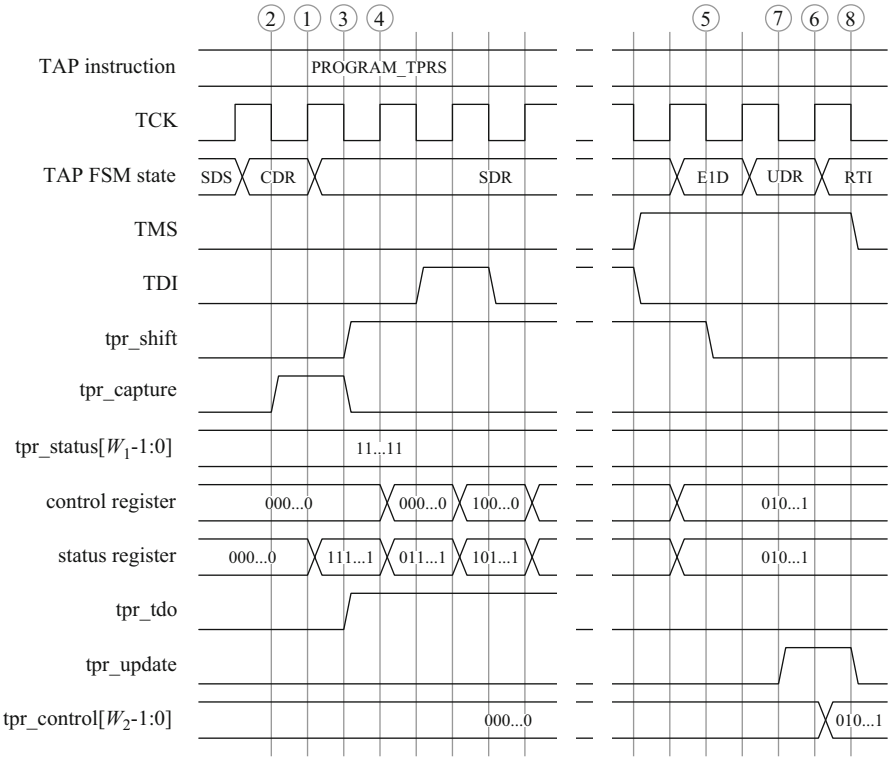


Fig. 5.8 Timing diagram for configuring and querying a TPR under TAP control

use type "U" to indicate a register field that has both control and update functionality, type "C" for a register field with control but no update functionality, and type "S" for a register field in the status register.

5.2.5 TAP–DTL Bridge

The external debug equipment can use the TAP, its associated controller, and a protocol-specific communication bridge to access the on-chip communication interconnect and perform functional read and write operations. Figure 5.9 shows a block diagram of the CSAR TAP–DTL bridge that provides access to an on-chip DTL bus from the TAP controller. Similar components can be designed for other communication protocols. The TAP–DTL bridge has a TPR interface on the left-hand side and a DTL initiator interface on the right-hand side. Table 5.4 shows the parameterized fields in the TAP–DTL bridge TPR.

Access is provided from the TAP to the TPR inside the TAP–DTL bridge by activating the PROGRAM_TPRS instruction in the TAP controller. This TPR can

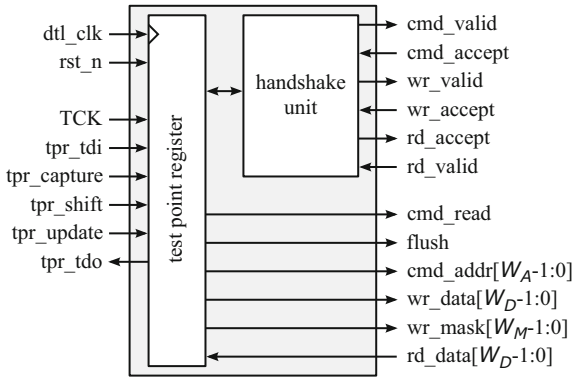


Fig. 5.9 Block diagram of a TAP–DTL bridge

subsequently be loaded with a DTL read or write command. The TPR can also be queried to detect whether these commands have been accepted by the on-chip communication interconnect and to obtain the data read by a previously-issued read command. The handshake unit uses the update register of the TPR to implement the DTL handshake logic. The "cmd_valid", "wr_valid", and "rd_accept" fields can be set using an update operation on the TPR. These bits are subsequently cleared when the corresponding handshake takes place. The status of these request and valid flags can be queried by capturing them in the status register of the TPR in the CDR state of the TAP controller. Note that the TPR in the TAP–DTL bridge does not allow specifying the "cmd_block_size" and "wr_last" DTL control signals. These signals are always set to respectively `0` and `1`, to indicate a single data element transfer.

5.3 Monitoring the Communication

5.3.1 Overview of a DTL Monitor

A *communication monitor* [3, 12] is attached to a communication link between an initiator and a target module to support a communication-centric debug approach. Figure 5.1 shows an overview of the possible monitor locations in an example SOC. In the remainder of this section, we describe how to implement hardware support for monitoring the communication inside an SOC. We focus on monitoring the DTL communication interface, because we use this type of interface in our case study in Chap. 8. Its concepts can be equally well applied to communication interconnects that use any modern handshake-based communication protocol, e.g., AXI [1] or OCP [9]. An AXI-based communication monitor, which supports similar functionality to that of the monitor we describe in this section, is for instance described in [11].

Figure 5.10 shows the architecture of a DTL monitor. This monitor implementation has been designed to meet debug infrastructure requirements IR-1, IR-2, IR-3, and IR-4, identified in Chap. 4.

Table 5.4 Fields in the TAP–DTL bridge TPR

Name	Width	Type	Offset	Description
cmd_addr	W_A	C	$2W_D + W_M + 9$	Command address
wr_data	W_D	C	$W_D + W_M + 9$	Write data
wr_mask	W_M	C	$W_D + 9$	Write mask
cmd_valid	1	C	$W_D + 8$	Command valid
cmd_read	1	C	$W_D + 7$	Read command
wr_valid	1	C	$W_D + 6$	Write valid
flush	1	C	$W_D + 5$	Command flush
rd_accept	1	C	$W_D + 4$	Read accept
bypass_en	1	U	$W_D + 3$	Control bypass enable `1`: Enabled `0`: Disabled
rd_data	W_D	S	3	Read data
cmd_status	1	S	2	Command handshake status `1`: Pending `0`: Completed
wr_status	1	S	1	Write handshake status `1`: Pending `0`: Completed
rd_status	1	S	0	Read handshake status `1`: Pending `0`: Completed

W_A address width, W_D data width, W_M mask width

This monitor consists of a DTL front end, three data matcher modules, an event encoder, and an event sequencer. The purpose of the front end is to abstract from the signals and signaling conventions used in the DTL communication protocol [8]. Each data matcher module is configurable to perform a data comparison operation and a checksum calculation. The event encoder compacts the information from the data matchers for use by the event sequencer. The event sequencer implements a configurable FSM to allow the detection of specific event sequences. These events may come from the communication on the DTL bus and from the EDI. The sequencer generates an event on one or more EDI outputs when it detects a pre-configured sequence. The data matchers and the event sequencer have an embedded TPR to configure their operation and to query their status using the DCSI.

5.3.2 DTL Front End

The DTL front end monitors the communication on the DTL interface and extracts four pieces of information: (1) a flag indicating whether the current command on the interface is a read or write command, (2) the associated command address, (3) any associated write data, and (4) any associated read data. The DTL protocol allows the initiator and target to provide a byte mask along with respectively the write and read data. The DTL front end applies this mask to the data values, before passing them on the data matchers, together with a signal that indicates when the handshake for that

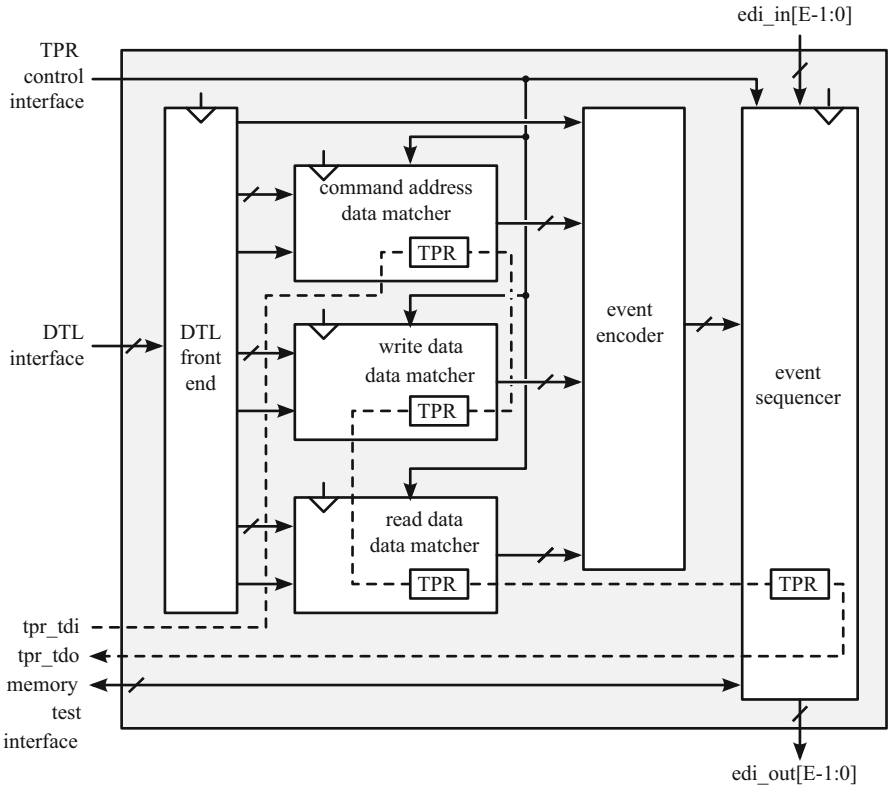


Fig. 5.10 Block diagram of a DTL monitor

data takes place. The data matchers only compare data that is actually transferred. The flag is passed to the event encoder using a pipeline register. This pipeline register is needed to keep the flag in sync with the results of the data matchers.

A number of communication protocols, including the AXI and DTL protocols, rely on the target to calculate the addresses belonging to the individual data elements in a block transfer. The DTL front end takes care of this address calculation for the DTL monitor.

5.3.3 Data Matcher

Figure 5.11 shows a block diagram of the data matcher module. A data matcher can perform either (1) an unsigned in-range test, where the range is determined by two configurable values, called "value_one" and "value_two", or (2) a masked equality test, where the reference value is determined by "value_one" and the mask by "value_two". Both values are provided by the TPR inside the data matcher. Matches

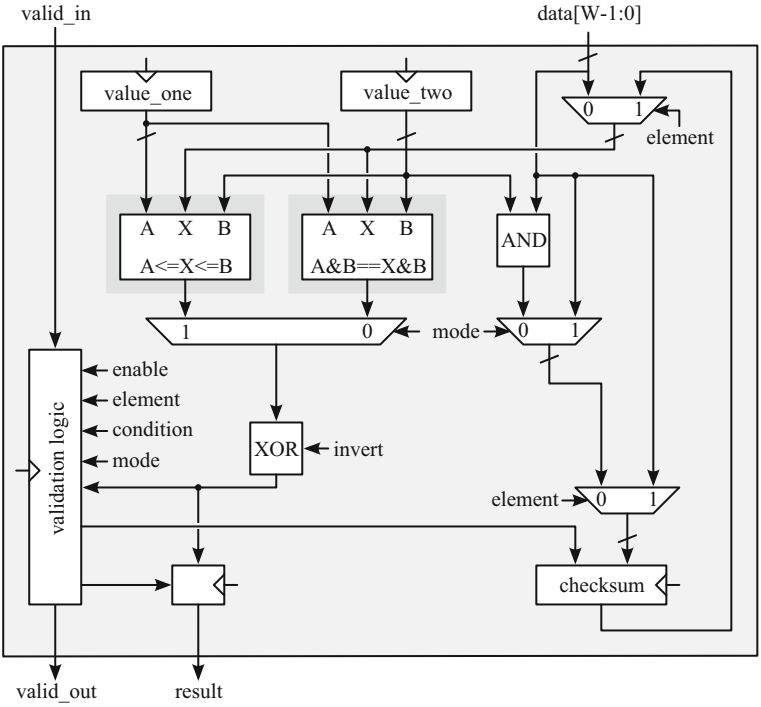


Fig. 5.11 Block diagram of a data matcher

can be inverted using the "invert" configuration bit in the TPR to provide either an unsigned out-of-range test or a masked inequality test. The matching operation can also be disabled by clearing the "enable" configuration bit in the TPR. The checksum logic calculates a compact signature of either the masked or unmasked incoming data, depending on the "mode" control signal from the TPR. The polynomial used in the checksum calculations is chosen at design time. The checksum value is updated on each match. The data value that is used in the tests depends on the "element" configuration bit in the TPR. The data value is either the incoming data value or the checksum value. We show examples of both in our case study in Chap. 8.

Table 5.5 shows the register fields in the data matcher TPR. The data matcher and its TPR are parameterizable in the width of the incoming data signal. Figure 5.11 does not show this embedded TPR for clarity. The TPR inside the data matcher consists of a single control register with bypass functionality (refer to Fig. 5.7). The implementation of this TPR has been optimized by merging the functionality of this control register with the functionality of the data matcher logic. When the "tpr_shift" signal is deasserted on the control register, the register fields inside the TPR implement the functionality as shown in Fig. 5.11. This is the case when the TPR is not accessed using the TAP controller and when this TPR has been placed in bypass mode. When the "tpr_shift" signal is asserted, all registers together behave as one TPR control register and allow loading and unloading their content under TAP control.

Table 5.5 Fields in the data matcher TPR

Name	Width	Type	Offset	Description
mode	1	C	$3W + 5$	Mode of operation `1`: Unsigned range test `0`: Masked equality test
value_one	W	C	$2W + 5$	Mode = `1`: upper bound (inclusive) Mode = `0`: data value
value_two	W	C	$W + 5$	Mode = `1`: lower bound (inclusive) Mode = `0`: mask
enable	1	C	$W + 4$	Data matcher enable `1`: Enabled `0`: Disabled
invert	1	C	$W + 3$	Invert match result `1`: Enabled `0`: Disabled
element	1	C	$W + 2$	Data value to compare `1`: Checksum value `0`: Incoming data
condition	1	C	$W + 1$	Checksum update condition `1`: On valid `0`: On match
checksum	W	C	1	Checksum value
bypass_en	1	U	0	Control bypass enable `1`: Enabled `0`: Disabled

W width of data value to match

Note that all registers in the data matcher are enabled using the "valid_in" input signal and the configuration bits in the TPR, to ensure that the data matcher only operates on valid data. This input signal is also output via a pipeline register to indicate the validity of the signal on its result output.

5.3.4 Event Encoder

The event encoder is used to combine the signals from the DTL front end and from the three data matchers into a more compact representation. This compaction is possible, because not all signal combinations are of interest to the event sequencer. For example, the result of a data matcher is only of interest when its corresponding valid output signal is asserted. The result can be ignored when this valid output signal is not asserted. The event encoder compacts the input combinations that to not convey any information together, and assigns unique values to all other combinations.

5.3.5 Event Sequencer

A block diagram of the CSAR event sequencer is shown in Fig. 5.12. This event sequencer is based on a synchronous, random accessible memory (SRAM) to allow

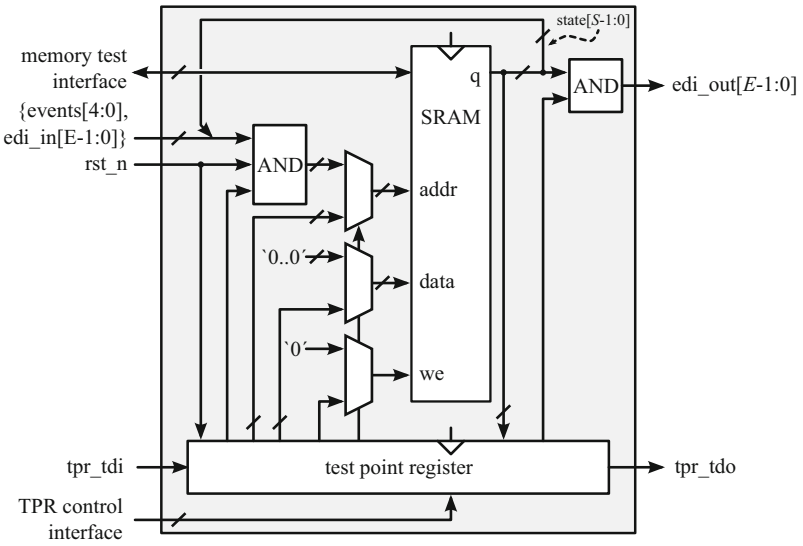


Fig. 5.12 Block diagram of an event sequencer

full programmability of the event sequence(s) for which an event is generated on the EDI. The address input of this SRAM is a vector, which consists of the output of the event encoder, the EDI input vector, and the current state of the sequencer. This address indexes an entry in the SRAM. This entry contains the values for the EDI output and the next state for the sequencer. The number of state bits S of the event sequencer is parameterizable. The total size of the SRAM needed in the event sequencer inside the CSAR DTL monitor is given by Eq. 5.1.

$$\text{RAM size} = 2^{(6+E+S)} \text{ words of } (E + S) \text{ bits each} \tag{5.1}$$

A TPR is used to configure the event sequencer. Table 5.6 summarizes the fields in this TPR. We can use the "reset_n" field in the TPR to force the address on the input of the SRAM to zero, thereby resetting the event sequencer to its initial state. This reset signal is asserted when the event sequencer is functionally reset and has to be deasserted before the event sequencer can response to incoming events. We can use the "control" field in the TPR to enable the control of the SRAM from the TPR, thereby allowing read and write operations to take place on the SRAM under control of the off-chip debugger software. When the "control" signal is asserted, the SRAM evaluates the control inputs from the TPR, and outputs the data stored at the address specified by the "tpr_addr" field. The SRAM output signal "q" is captured in the TPR when the "tpr_capture" signal is asserted (as explained in Sect. 5.2.4). The "tpr_we" control signal is automatically deasserted during load and unload operation of the TPR. This prevents accidental writes to the SRAM during TPR accesses. Only when the data in the control register of the TPR is stable, i.e., after the "tpr_shift" signal is deasserted, is the value of the "tpr_we" field in the TPR applied to the SRAM.

Table 5.6 Fields in the event sequencer TPR

Name	Width	Type	Offset	Description
reset_n	1	U	$W_A + 2W_D + 4$	Sequencer reset 0: Asserted 1: Deasserted
control	1	U	$W_A + 2W_D + 3$	Sequencer mode 0: Functional mode 1: Program mode
tpr_we	1	C	$W_A + 2W_D + 2$	Write enable 1: Enabled 0: Disabled
tpr_addr	W_A	C	$2W_D + 2$	Address
tpr_data	W_D	C	$W_D + 2$	Data to write
output_en	1	U	$W_D + 1$	Output enable 1: Enabled 0: Disabled
bypass_en	1	U	W_D	Control bypass enable 1: Enabled 0: Disabled
tpr_q	W_D	S	0	Data read

W_A address width, W_D data width

The EDI output can be disabled from the TPR to prevent the accidental generation of events on the EDI while the event sequencer is configured. The event sequencer also has a memory test interface (shown in the top-left of Fig. 5.12) to allow access to this memory during manufacturing test. This interface is described in more detail in Sect. 5.7.2.

5.4 Controlling the Communication

5.4.1 Overview of a DTL Protocol Specific Instrument

A PSI is inserted in a communication link between an initiator and a target module to support our communication-centric debug approach. Figure 5.1 shows an overview of the possible PSI locations in an example SOC. We describe in the remainder of this section the implementation of a PSI for the DTL protocol to illustrate the communication control functionality provided by a PSI. The communication control concepts however can be equally well applied to other communication protocols, e.g., AXI [1] or OCP [9].

A high-level block diagram of a DTL PSI is shown in Fig. 5.13. This PSI implementation has been designed to meet debug infrastructure requirements IR-5, IR-6, IR-7, IR-8, and IR-9, identified in Chap. 4. This particular PSI is used in our case study in Chap. 8 and consists of a PSI mask module, a PSI TPR, and an event generator, which are explained in the sections below.

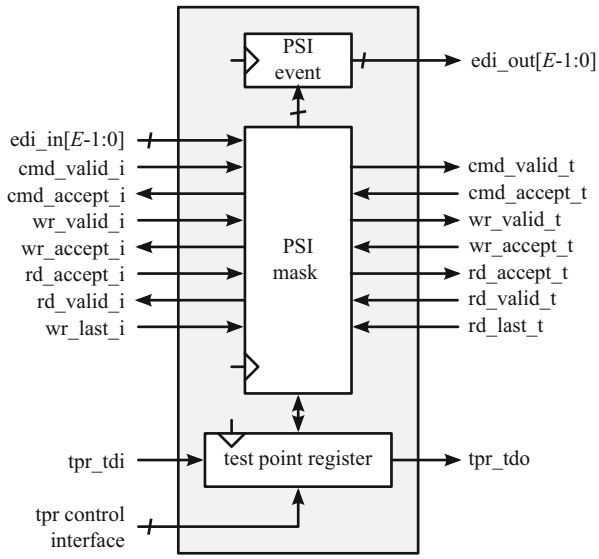


Fig. 5.13 Block diagram of a DTL PSI

5.4.2 DTL PSI Mask

The DTL PSI mask module stops the communication on the link by gating the handshake control signals used in the DTL protocol. For this purpose, the mask module consists of two counters, two flags, and two FSMs. The counters, "pending_writes" and "pending_reads", keep track of the number of respectively pending write transactions on the request channel and pending read transactions on the response channel. Each counter is incremented when its associated command is accepted on the communication link, and decremented when the last element in the associated block of data elements is transferred. The flags, "write_in_progress" and "read_in_progress", indicate whether respectively a write transaction and/or a read transaction is currently in progress on the communication link. These flags are set when respectively a new write and new read command is accepted, and cleared when respectively a new write data element and a new read data element is transferred with respectively the "wr_last" signal and the "rd_last" signal asserted. There is a request FSM for the request channel and a response FSM for the response channel. The STG of both FSMs is shown in Fig. 5.14. Each FSM has four states. Table 5.7 shows the states in which the command, write, and read handshakes of the DTL protocol are gated. Each FSM enters the RUN state upon reset. From there, the state transitions depend on the events on the DTL communication link, on the EDI, and the configuration of the TPR. We use the conditions described in Table 5.8 to explain the conditions for the state transitions in Fig. 5.14 below. The request FSM transitions from the RUN state to the WAIT state when Condition b_{req}, as defined in Eq. 5.2, is valid.

$$b_{req} = (edi + stop_{req}) \cdot enable_{req} \cdot (trans_{req} \cdot pending\_writes + message_{req} \cdot write\_in\_progress) \quad (5.2)$$

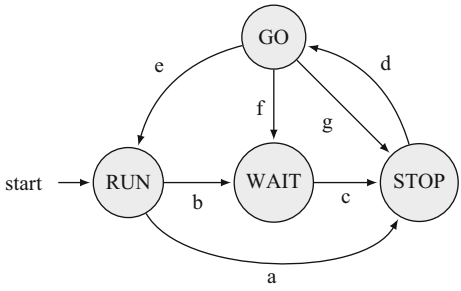


Fig. 5.14 STG of the FSMs used in the DTL PSI

Table 5.7 Handshake gating in the FSM states of the DTL PSI

State	Command	Write	Read
RUN	No	No	No
WAIT	Yes	No	No
STOP	Yes	Yes	Yes
GO	No	No	No

This transition takes place when (1) either an event arrives from the EDI or the user has specified the DTL PSI to stop unconditionally, and (2) the user has enabled the PSI, and (3) either the user has specified transaction-level stopping and there are still write commands pending or the user has specified message-level stopping and there is still a write transaction in progress. Similarly, Eq. 5.3 defines this condition for the response FSM.

$$b_{resp} = (edi + stop_{resp}) \cdot enable_{resp} \cdot (trans_{resp} \cdot pending\_reads + message_{resp} \cdot read\_in\_progress) \quad (5.3)$$

The command handshake is gated in the WAIT state, while the write handshakes are allowed to continue until Condition c_{req} in Eq. 5.4 is valid.

$$c_{req} = (trans_{req} \cdot \overline{pending\_writes}) + (message_{req} \cdot \overline{write\_in\_progress}) \quad (5.4)$$

This condition is true, when (1) the user has specified transaction-level stopping and there are no write transactions pending, or (2) the user has specified message-level stopping and there is no write transaction in progress. Similarly, Eq. 5.5 defines this condition for the response FSM.

$$c_{resp} = (trans_{resp} \cdot \overline{pending\_reads}) + (message_{resp} \cdot \overline{read\_in\_progress}) \quad (5.5)$$

From the RUN state, the request FSM transitions directly to the STOP state, when Condition a_{req}, as defined in Eq. 5.6, is valid.

$$a_{req} = (edi + stop_{req}) \cdot enable_{req} \cdot (c_{req} + element_{req}) \quad (5.6)$$

This is the case when (1) either an event arrives from the EDI or the user has specified the DTL PSI to stop unconditionally, and (2) the user has enabled the PSI, and

Table 5.8 Conditions used in the description of the DTL PSI mask module

Condition	Description
cmd	Command handshake completes on the communication link
wr	Write handshake completes on the communication link
rd	Read handshake completes on the communication link
edi	An event arrives on the EDI
$elem_{req}$	The stop granularity of the request channel is set to element level
msg_{req}	The stop granularity of the request channel is set to message level
$trans_{req}$	The stop granularity of the request channel is set to transaction level
$stop_{req}$	The stop condition is set to unconditional for the request channel
$enable_{req}$	The PSI is enabled for the request channel
$continue_{req}$	The user has issued a continue request on the request channel
$elem_{resp}$	The stop granularity of the response channel is set to element level
msg_{resp}	The stop granularity of the response channel is set to message level
$trans_{resp}$	The stop granularity of the response channel is set to transaction level
$stop_{resp}$	The stop condition is set to unconditional for the response channel
$enable_{resp}$	The PSI is enabled for the response channel
$continue_{resp}$	The user has issued a continue request on the response channel

(3) either the user has specified transaction-level stopping and there are no write commands pending or the user has specified message-level stopping and there is no write command in progress or the user has specified element-level stopping. Similarly, Eq. 5.7 defines this condition for the response FSM.

$$a_{resp} = (edi + stop_{resp}) \cdot enable_{resp} \cdot (c_{resp} + element_{resp}) \tag{5.7}$$

The request FSM gates the command and the write handshakes in its STOP state. Similarly, the response FSM gates the read handshakes in its STOP state.

The request FSM transitions from the STOP state to the GO state when the user issues a continue request for this channel. This condition is defined in Eq. 5.8.

$$d_{req} = continue_{req} \tag{5.8}$$

Similarly, Eq. 5.9 defines this condition for the response FSM.

$$d_{resp} = continue_{resp} \tag{5.9}$$

None of the handshakes are gated in the GO state. As a consequence, three transitions are possible from this state. First, the request FSM transitions to the RUN state when Condition e_{req}, as defined in Eq. 5.10, is valid.

$$e_{req} = (element_{req} \cdot wr + cmd) \cdot \overline{a_{req}} \cdot \overline{b_{req}} \tag{5.10}$$

This is the case when (1) either the user has specified element-level stopping for the request channel and a write handshake takes place on the communication link, or a command handshake take places on the communication link, and (2) Condition a_{req} to move from the RUN state to the STOP state is false, and (3) Condition b_{req} to

move from the RUN state to the WAIT state is false. Similarly, Eq. 5.11 defines this condition for the response FSM.

$$e_{resp} = element_{resp} \cdot rd \cdot \overline{a_{resp}} \cdot \overline{b_{resp}} \tag{5.11}$$

Second, the request FSM transitions from the GO state to the WAIT state when Condition f_{req}, as defined in Eq. 5.12, is valid.

$$f_{req} = (element_{req} \cdot wr + cmd) \cdot b_{req} \tag{5.12}$$

This is the case when (1) either the user has specified element-level stopping for the request channel and a write handshake takes place on the communication link, or a command handshake take places on the communication link, and (2) Condition b_{req} to move from the RUN state to the WAIT state is valid. Similarly, Eq. 5.11 defines this condition for the response FSM.

$$f_{resp} = element_{resp} \cdot rd \cdot b_{resp} \tag{5.13}$$

Third, the request FSM transitions from the GO state to the STOP state when Condition g_{req}, as defined in Eq. 5.14, is valid.

$$g_{req} = (element_{req} \cdot wr + cmd) \cdot a_{req} \tag{5.14}$$

This is the case when (1) either the user has specified element-level stopping for the request channel and a write handshake takes place on the communication link, or a command handshake take places on the communication link, and (2) Condition a_{req} to move from the RUN state to the STOP state is true. Similarly, Eq. 5.15 defines this condition for the response FSM.

$$g_{resp} = element_{resp} \cdot rd \cdot a_{resp} \tag{5.15}$$

Please note that Conditions f_{req} and g_{req} are mutual exclusive, because Conditions a_{req} and b_{req} are. Similarly, Conditions f_{resp} and g_{resp} are mutual exclusive.

5.4.3 DTL PSI Test Point Register

We can use the DTL PSI TPR shown in Fig. 5.13 to configure the DTL PSI. Table 5.9 shows the control and status register fields of this TPR.

The request and response channels have the same debug functionality, but each channel has its own dedicated set of register fields. Note how the fields in Table 5.9 have been ordered by type to match the generic TPR architecture shown in Fig. 5.7.

The PSI TPR deactivates the req_continue and resp_continue output signals during reprogramming operations to prevent the accidental generation of continue requests to the request and response FSMs. The PSI TPR captures eight status signals from the DTL PSI mask module (see Table 5.9). The debugger software, described in Chap. 7, can use this information to present an overview of the status of all DTL request and response channels.

Table 5.9 Fields in the DTL PSI TPR

Name	Type	Offset	Description
req_stop_en	C	$E + 18$	Request channel stop enable 1: Enabled 0: Disabled
req_stop_granularity	C	$E + 16$	Request channel stop granularity 1–: Transaction-level 01: Message-level 00: Element-level
req_stop_condition	C	$E + 15$	Request channel stop condition 1: Stop unconditionally 0: Stop on any event from the EDI
req_continue	C	$E + 14$	Request channel continue execution 1: Issue continuation request 0: Reset continuation request
resp_stop_en	C	$E + 13$	Response channel stop enable 1: Enabled 0: Disabled
resp_stop_granularity	C	$E + 11$	Response channel stop granularity 1–: Transaction-level 01: Message-level 00: Element-level
resp_stop_condition	C	$E + 10$	Response channel stop condition 1: Stop unconditionally 0: Stop on any event from the EDI
resp_continue	C	$E + 9$	Response channel continue execution 1: Issue continuation request 0: Reset continuation request
edi_mask	C	9	EDI output mask bit = 1: Enabled bit = 0: Disabled
bypass_en	U	8	Control register bypass 1: Enabled 0: Disabled
write_stop	S	7	Request channel write group stop status 1: Write group has been stopped 0: Write group has not been stopped
command_stop	S	6	Request channel command group stop status 1: Command group has been stopped 0: Command group has not been stopped
read_stop	S	5	Response channel read group stop status 1: Read group has been stopped 0: Read group has not been stopped
write_pending	S	4	Flag indicating whether communication is pending on the write group 1: Pending write request from initiator 0: No pending write requests
command_pending	S	3	Flag indicating whether communication is pending on the command channel 1: Pending command request from initiator 0: No pending command requests

Table 5.9 (continued)

Name	Type	Offset	Description
read_pending	S	2	Flag indicating whether communication is pending on the response channel 1: Pending read response from target 0: No pending read responses
req_stop_left	S	1	Flag indicating whether the communication has resumed on the request channel 1: Communication has resumed 0: Communication has not resumed
resp_stop_left	S	0	Flag indicating whether the communication has resumed on the response channel 1: Communication has resumed 0: Communication has not resumed

5.4.4 DTL PSI Event Generator

An edge generator in the PSI is connected to the same eight status signals of the DTL PSI mask module as the TPR. Its output is connected to the EDI (refer to Fig. 5.13). This generator outputs an event on the EDI layers, specified by the "edi_mask" field in the TPR, when any of these status signals is asserted.

5.4.5 Example DTL PSI Timing Diagram

Figure 5.15 shows an example timing diagram of stopping and subsequently continuing the communication at message-level granularity by a DTL PSI on a DTL response channel. The PSI TPR has been programmed to stop the communication on the response channel when an event comes in on the EDI at the level of messages by clearing the "resp_stop_condition" field and setting the "resp_stop_granularity" field to "01" in the PSI TPR.

When the DTL PSI detects an event on its EDI input, the response FSM transitions to the WAIT state because a previously-initiated read transaction is still in progress (①). In this state, the response FSM gates the command handshake and sends an event out on Layer 2 of the EDI using the "edi_mask" field and the "edi_out" output. On the first read data element handshake visible in Fig. 5.15, the target indicates that this is not the last read data element in the message by deasserting its "rd_last_t" signal (②). On the second read data element handshake, the target indicates that this is the last read data element in the message by asserting its "rd_last_t" signal (③). In response, the response FSM transitions to the STOP state and blocks all handshakes on the DTL response channel. Again, an event is sent on Layer 2 of the EDI.

While the communication on the response channel is stopped by the PSI, valid data elements may be offered by the initiator or the target through the assertion of the "cmd_valid_i" and "rd_valid_t" signals. The fact that the transfer of a data element is

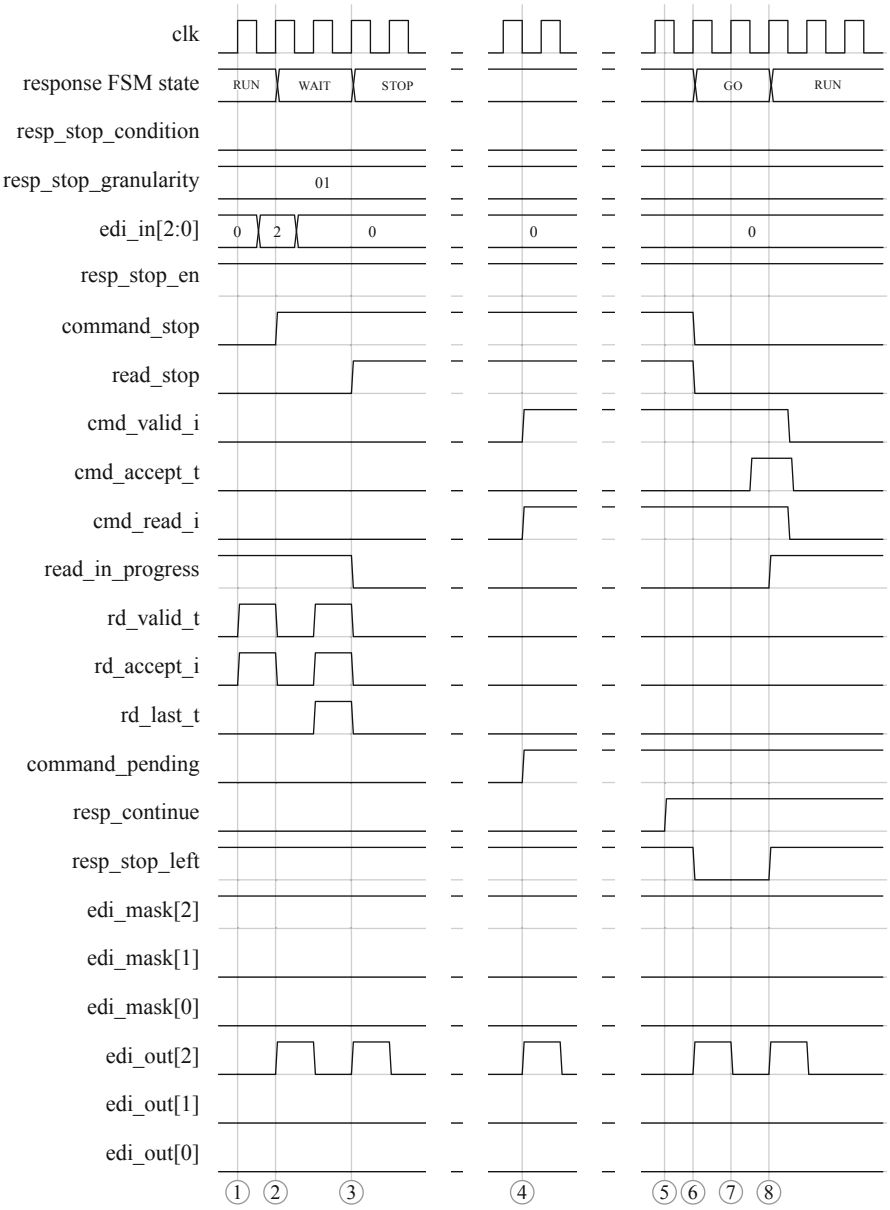


Fig. 5.15 Response channel stopping and continuing at message granularity

pending on the response channel at the input of the PSI is observable by the assertion of the "command_pending" and "read_pending" status signals (④). This condition also causes an event to be generated on the EDI.

When the "resp_continue" bit in the PSI TPR is toggled from "0" to "1" (⑤), the response FSM transitions to the GO state (⑥), in which the "resp_stop_left" status signal is deasserted (⑦). This condition also causes an event to be generated on the EDI. The pending command data element handshake can subsequently complete (⑧). The completion of this handshake causes the response FSM to transition to the RUN state, where it reasserted the "resp_stop_left" status signal. This indicates that communication has taken place on the response channel after the continuation request and another event is sent on Layer 2 of the EDI to signal this.

5.5 Event Distribution Interconnect

5.5.1 Overview

The EDI is a scalable, high-speed communication interconnect, which propagates events from event sources to a configurable subset of event destinations (see Fig. 5.1). The EDI implementation has been designed to meet debug infrastructure requirements IR-10, IR-11, IR-12, IR-13, and IR-14, identified in Chap. 4.

The EDI consists of a multi-hop broadcast network of EDI nodes (Sect. 5.5.2) and EDI CDC modules to cross clock domain boundaries (Sect. 5.5.3). External events can be injected onto the EDI using the TAP and the TAP–EDI bridge (Sect. 5.5.4). This bridge also allows internal EDI events to be observed by the off-chip debugger software. The topology of the EDI mirrors the topology of the on-chip communication interconnect. We designed a parameterizable implementation of the EDI node. The value of this parameter for a particular EDI node depends on the number of neighboring nodes that it has. An EDI node can selectively forward an event it receives from a local event source or neighbor EDI node to the event destinations and other EDI nodes in its neighborhood. Inside a clock domain, a debug event incurs only a single clock cycle delay for every EDI node that it crosses to reach its event destination(s).

When event sources and destinations are in different clock domains, we have to use an EDI CDC module to safely communicate the number of events generated in the source clock domain to the destination clock domain. A debug event therefore incurs a low but potentially variable number of clock cycles delay when it crosses a clock domain boundary (refer to Sect. 3.1.3).

Congestion on the EDI is prevented by the use of a parameterizable number of independent EDI layers and the selective forwarding of events. Each layer is able to distribute a debug event to any number of event destinations. With E EDI layers, we can distribute at least E events in parallel and possibly more, depending on the sharing of EDI nodes between different event routes. Figure 5.1 includes an overview of an EDI in an example SOC, together with its event sources and destinations. The monitors and PSIs introduced in Sects. 5.3 and 5.4 act as both event sources and event destinations for the EDI.

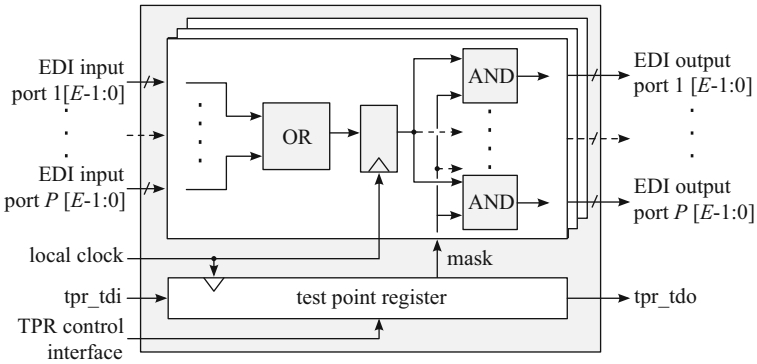


Fig. 5.16 Block diagram of an EDI node

5.5.2 EDI Node

Figure 5.16 shows a block diagram of the CSAR EDI node. An EDI node is basically a single pipeline register with a configurable output mask. This pipeline register ensures that the debug events can be distributed across a single clock domain without violating the timing constraints (refer to Sect. 3.1.2). The embedded TPR provides a mask to allow the selective forwarding of the events to be configured using the DCSI.

The EDI node is parameterizable in the number of EDI ports and the number of EDI layers. The number of ports on an EDI node matches the number of functional ports on the associated, functional communication building block (e.g., a bus, a router, or a network interface), and therefore varies from node to node. The number of EDI layers depends on the number of events that the EDI has to be able to communicate in parallel without event collisions. This number is the same for all nodes. As it may be expensive to add an additional layer to the entire EDI to support the propagation of an additional debug event, each EDI node has the ability per layer to forward a debug event to only a pre-configured subset of output ports. This potentially allows multiple debug events to be routed within the same EDI layer without collisions, thereby saving the cost of an additional layer at the expense of a larger EDI node. This masking functionality also comes with the advantage that no masking is needed at the event destinations, as the masking functionality in the preceding EDI node in the event path takes care of this. The EDI node forwards incoming debug events on an input port i to a subset of output ports J, depending on the binary mask that is provided by the embedded TPR. Table 5.10 shows the fields in this TPR.

The debug engineer is responsible for programming the mask values in the EDI nodes such that the propagation paths from the event sources to the event destinations are properly set up. The off-chip debugger software provides support for this, which we present in Chap. 7.

Table 5.10 Fields in the EDI node TPR

Name	Width	Type	Offset	Description
mask	$P \times E$	C	1	Event propagation enable per port and per EDI layer 1: Enable propagation to this port on this layer 0: Disable propagation to this port on this layer
bypass_en	1	U	0	Control register bypass 1: Enabled 0: Disabled

P number of EDI ports, E number of EDI layers

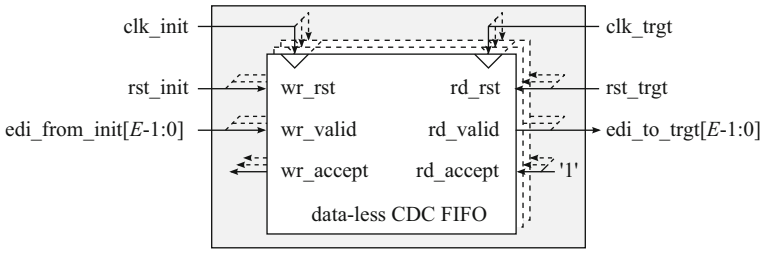


Fig. 5.17 Block diagram of an EDI clock domain crossing module

5.5.3 EDI Clock Domain Crossing Module

We cannot use an EDI node to cross clock domain boundaries because its interface is synchronous. Instead, we use the CSAR EDI CDC module to communicate debug events across a clock domain boundary. Figure 5.17 shows a block diagram of the EDI CDC module.

The EDI CDC module is based on the CDC FIFO in [4]. The EDI CDC module however does not have internal RAM modules. It only reuses the counters and their associated logic to reproduce the same number of events on its target output port as it received on its initiator input port. The number of bits in these counters has to be chosen at design time. Any additional events will be dropped when these counters reach their maximum value. Figure 5.18 shows an example timing diagram for a four-layer EDI CDC module.

We see that multiple debug events are generated on the four EDI layers over the course of six clock cycles (① through ⑥) on the initiator EDI port "edi_from_init". The sampling of these EDI events by the EDI CDC module is indicated by the black dots on the waveforms of the "edi_from_init" signals. For example, we see events being sampled on Layer 3 at times ②, ③, ④, and ⑥. Each of these events are reproduced on the corresponding layer at the target output port "edi_to_trgt". For example, the four input events on Layer 3 are reproduced at times ①, ②, ③, and ④. Figure 5.18 shows that the number of events per layer is preserved, i.e., no debug event is lost nor duplicated, even though their exact timing relation is not. The target therefore sees the exact same number of events per layer as were generated by the initiator.

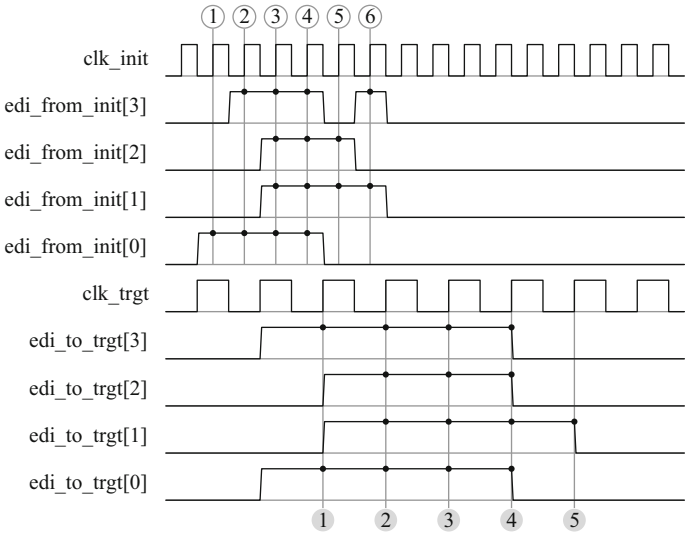


Fig. 5.18 Timing diagram of the operation of a four-layer EDI CDC module

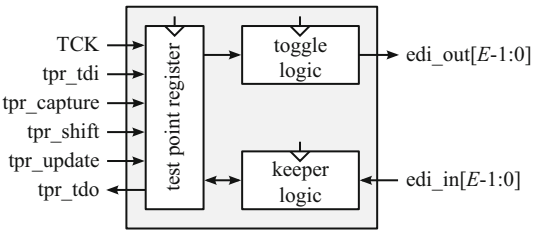


Fig. 5.19 Block diagram of a TAP–EDI bridge module

5.5.4 TAP–EDI Bridge

The off-chip debugger software may also wish to inject a debug event onto the on-chip EDI. For example, the execution inside the SOC may need to be stopped when the debug engineer presses a "stop" button in the graphical user interface (GUI) of the off-chip debugger software. We instantiate a TAP–EDI bridge component in the top-level module of the SOC, to be able to generate a debug event on the EDI under control of the TAP controller. This bridge also acts as a possible destination for the internal debug events distributed via the EDI, so that the off-chip debugger software can inform the debug engineer about them.

Figure 5.19 shows a block diagram of the TAP–EDI bridge module. This block diagram shows a TPR interface on the left-hand side of the bridge and EDI input and output ports on its right-hand side. Table 5.11 shows the parameterizable fields in the TAP–EDI bridge TPR.

Access to the TPR inside the TAP–EDI bridge from the TAP is provided by activating the PROGRAM_TPRS instruction in the TAP controller. This TPR can

Table 5.11 Fields in the TAP–EDI bridge TPR

Name	Width	Type	Offset	Description
edi_out	E	U	$E+1$	EDI output
bypass_en	1	U	E	Control bypass enable `1`: Enabled `0`: Disabled
edi_in	E	S	0	EDI input

E number of EDI layers

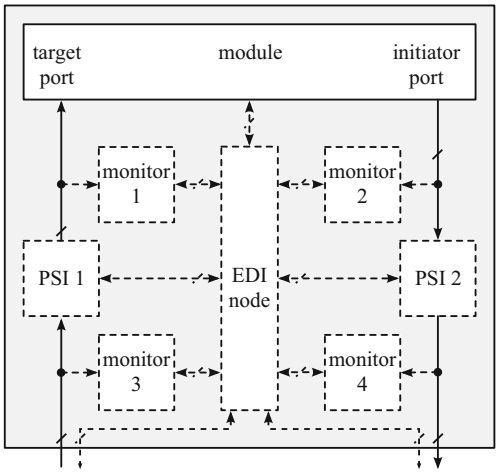


Fig. 5.20 Block diagram of an example debug wrapper

subsequently be loaded with an EDI event for each individual layer and be queried whether events were previously detected on each layer. Inside the bridge, logic surrounds the TPR update register to implement the EDI toggle logic. The "edi_out" signals are set via an update operation of the TPR and are cleared in the next clock cycle. This causes a single event to be injected on the associated EDI layer(s). The keeper logic tracks whether events have occurred on each EDI layer. This status information can be captured in the status register of the TPR and currently comprises a single bit per layer, which is set when at least one event has previously occurred on that layer. It is possible to replace this bit with a counter per layer, which counts the exact number of events that occurred on the corresponding layer. We chose not to do this. The keeper logic is reset when its status information is captured in the TPR status register.

5.6 Debug Wrapper

A debug wrapper groups a module with its communication monitors, PSIs, and EDI support for communication-centric debug. Figure 5.20 show the block diagram of an example debug wrapper around a module, such as a processor or a memory.

This module has an initiator and a target communication port. We have several options to connect monitors and PSIs as indicated in Fig. 5.20 by the dashed blocks. The PSIs are inserted in the communication link between the communication port on the module and the corresponding port on the debug wrapper. We identify four options to include a monitor on a communication link in which a PSI has been inserted: (a) on the *initiator* side, e.g., monitors 2 and 3, (b) on the *target* side, e.g., monitors 1 and 4, (c) on the side of the module, e.g., monitors 1 and 2, and (d) on the side of the *debug wrapper*, e.g., monitors 3 and 4. The EDI node is instantiated when needed. It routes the event signals from the EDI outside the wrapper to and from the monitors and the PSIs in the debug wrapper. It is also possible to hook up module-specific event signals, such as processor breakpoints, to the EDI node as well. This permits the seamless integration of our *communication-centric debug approach* with the traditional *computation-centric debug approach*, since for example processor breakpoint events can be routed to the PSIs as well, to also stop the ongoing transactions, or to stop the other modules.

5.7 Test Wrapper

5.7.1 Overview

Each module is wrapped in a *test wrapper* to provide access to its scan chains for manufacturing test and for scan-based debugging. Figure 5.21 shows a block diagram of our test wrapper. This generic test wrapper has been designed to meet debug infrastructure requirements IR-15 and IR-16 that are identified in Chap. 4. The implementation of a test wrapper therefore uses seven components: logic scan chains in the scan-inserted module (①), optional memory scan chains in test wrappers around the memories embedded in the module (②), a primary input/output (PIO) unit (③), a SIM (④), a local clock gate (⑤), a protocol-specific controller (⑥), and anti-skew flip-flops on the parallel and serial scan outputs (⑥). The logic scan chains (①) are inserted by commercial EDA tools during synthesis. In the description below, we use the value T for the number of parallel scan chains that were inserted in the module. The anti-skew flip-flops (⑦) on the scan outputs serve the same purpose as the anti-skew flip-flops in the TCBs and TPRs. The other components are described in more detail below.

5.7.2 Memory Test Wrapper

Designers need to implement special *memory test wrappers* around the embedded RAMs to allow access to their state during manufacturing test and during scan-based debug. Each wrapper allows serial access to the ports of the memory to execute functional read and write operations on the memory. During debug, we use these

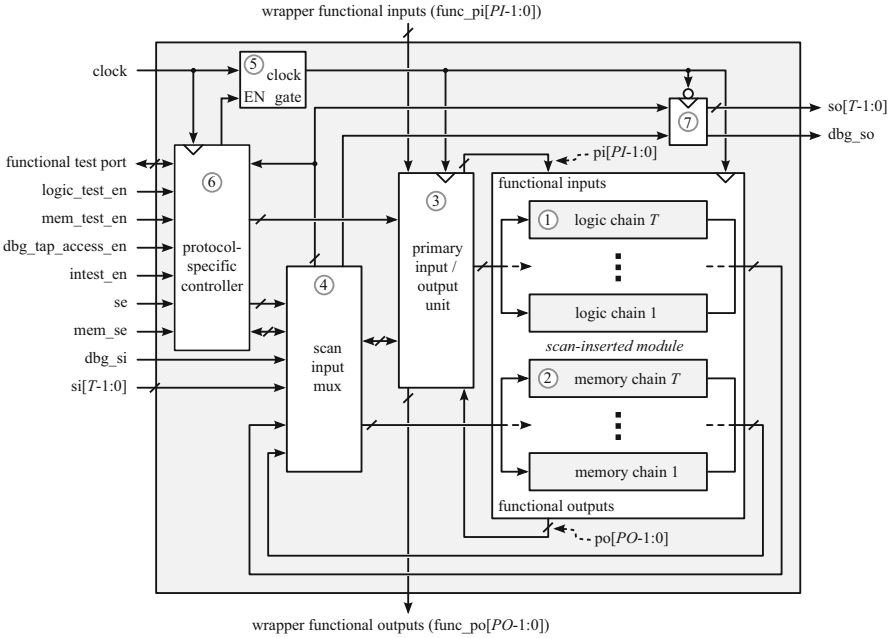


Fig. 5.21 Block diagram of a generic test wrapper

read operations to dump the content of the memory and these write operations to modify the memory content. The memory test wrapper is controlled from the global TCB (Sect. 5.2.3) and the protocol-specific controller inside the test wrapper.

Figure 5.22 shows an example block diagram of a memory test wrapper around a single-port memory. The wrapper is transparent in *functional mode*, passing the functional I/Os of the wrapper unmodified to the memory. During manufacturing test and during scan-based debug, the functional inputs of the memory are driven from the shift register inside the wrapper. This shift register can be loaded with a valid combination of address, data, and write-enable control signals to execute a functional write or read operation. Access to the shift register is enabled during manufacturing test and during scan-based debug, by asserting the "mem_se" signal from the global TCB. When the content of the shift register is accessed, the write-enable signal of the memory is automatically deasserted to prevent accidental write operations on the memory. At the same time, the data output of the memory wrapper is tied off to prevent test or debug data from the memory to propagate to the surrounding logic.

The address, data, and write-enable values that are loaded in the shift register are only applied to the inputs of the memory after the shift register has been loaded with a valid combination and the "mem_se" signal is deasserted. The data output of the memory is captured in the shift register when the "mem_se" signal is deasserted. Once captured, it can be unloaded for inspection. We use this memory dump functionality in the case study in Chap. 8. Table 5.12 shows the fields in the shift register, their widths, offsets, and a short description.

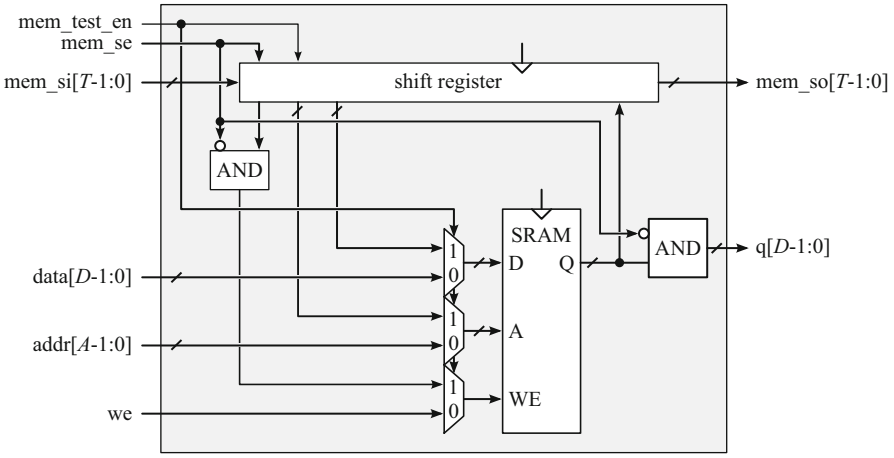


Fig. 5.22 Block diagram of a single-port SRAM test wrapper

Table 5.12 Fields in the shift register of the memory test wrapper

Name	Width	Offset	Description
we	1	$W_A + 2W_D$	Write enable `1`: Write operation `0`: Read operation
address	W_A	$2W_D$	Address input
data	W_D	W_D	Memory data input
q	W_D	0	Memory data output

W_D data width, W_A address width

The memory test wrapper for a single-clock, dual-port SRAM module is similar to the test wrapper for a single-port SRAM module. The shift register is however twice as long to also include the control and data fields for the second memory port.

5.7.3 *Primary Input/Output Unit*

The PIO unit wraps the inputs and outputs of the module, thereby providing control over its inputs and observability of its outputs in *manufacturing test mode* and in *debug mode*. Figure 5.26 shows the I/Os of the PIO unit. The PIO unit has an internal register "pio_r", which contains an input section with a flip-flop for each wrapped functional input and an output section with a flip-flop for each wrapped functional output of the module. The PIO unit has four operating modes:

1. *Transparent mode*: The functional I/Os of the test wrapper are directly connected to the functional I/Os of the module. The PIO register keeps its current state in this mode (refer to Fig. 5.25).

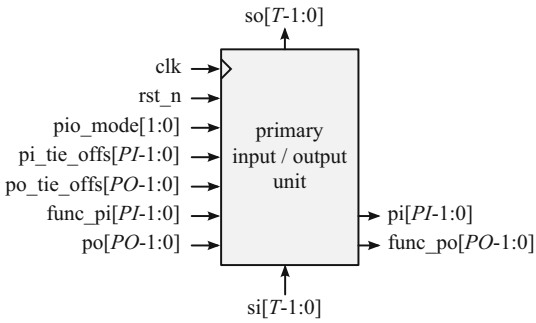


Fig. 5.23 External test mode of the PIO unit (pio_mode=11)

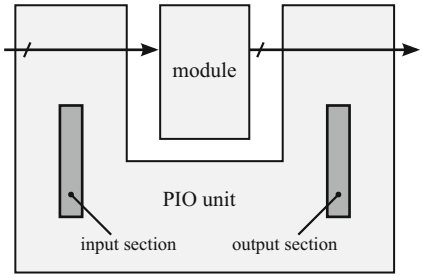


Fig. 5.24 Internal test mode of the PIO unit (pio_mode=10)

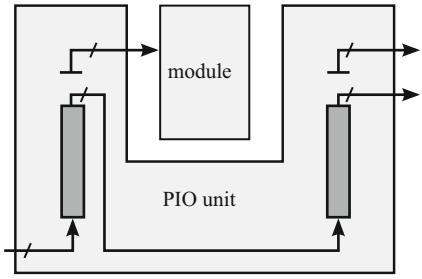


Fig. 5.25 Transparent mode of the PIO unit (pio_mode=00)

2. *Shift mode*: The functional inputs of the module and the functional outputs of the test wrapper are tied off to respectively the values "pi_tie_offs" and "po_tie_offs". The PIO register is loaded with T bits in each clock cycle from the parallel scan input "si", and is unloaded T bits in each clock cycle via the parallel scan output "so" (refer to Fig. 5.27).
3. *Internal test mode*: The functional inputs of the module are driven from the input section of the PIO register. The functional outputs of the module are captured in the output section of the PIO register. The functional outputs of the test wrapper are tied off to the value "po_tie_offs" (refer to Fig. 5.24).
4. *External test mode*: The functional inputs of the module are tied off to the value "pi_tie_offs". The functional inputs of the PIO unit are captured in the input section of the PIO register. The functional outputs of the test wrapper are driven from the output section of the PIO register (refer to Fig. 5.23).

The parallel scan output "so" is in all modes connected to the T most significant bits (MSBs) of the PIO register. The value of "pi_tie_offs" is specified at design time and

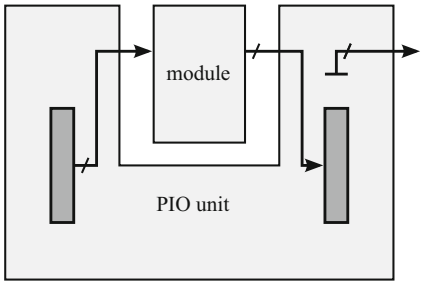


Fig. 5.26 PIO unit in the generic test wrapper

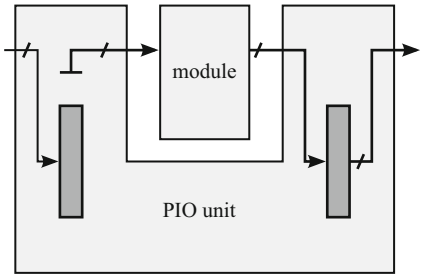


Fig. 5.27 Shift mode of the PIO unit (pio_mode=`01`)

chosen such that assigning this value to the functional inputs of the module causes no action to be taken inside the module. Likewise, the value of "po_tie_offs" is specified at design time and chosen such that assigning this value to the functional outputs of the test wrapper causes no action to be taken outside the module. We show in Sect. A.1 how these tie-off values can be specified for each individual module. The PIO unit is controlled from the protocol-specific controller, as explained in Sect. 5.7.6.

5.7.4 Scan Input Multiplexer

The SIM selects which state-holding elements inside the module and test wrapper are included in the scan path and which data input and output are used to access this scan path. This selection is based on the "input_select" control signal from the protocol-specific controller (refer to Sect. 5.7.6). Based on the "chain_config" control input from the same controller, the SIM either includes the logic chains, the memory chains, or both in the scan path in series with the controller itself and the PIO unit. The register in the PIO unit is always included in the scan path to make sure we can always control the inputs of the scan-inserted module and the outputs of the test wrapper and set them to safe functional values during test and debug operations. Figure 5.28 shows the I/Os of the SIM and Table 5.14 specifies its functionality (Table 5.14).

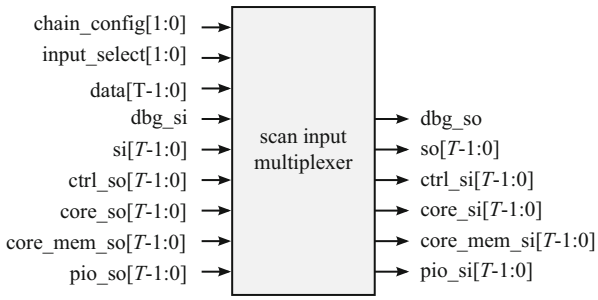


Fig. 5.28 Scan input multiplexer in the generic test wrapper

In *functional mode*, the SIM selects the data input "data" as input for the scan path (refer to the top row in Table 5.14). This input is connected to the data input of the functional test port. The output of the scan path is the "so" port on the SIM. This port is connected to the data output of the functional test port via the controller (refer to Fig. 5.21). This allows access to the scan path from the functional test port using the controller (refer to Sect. 5.7.6).

In *manufacturing test mode*, the SIM selects the parallel scan input "si" as the input of the scan path (refer to the second row in Table 5.14). The output of the scan path is the parallel scan output "so". At the chip-level, we use these parallel scan inputs and parallel scan outputs to chain the test wrappers between the scan input pins and scan output pins of the SOC (refer to Fig. 5.2).

In *debug mode*, the SIM selects the serial scan input "dbg_si" as the input of the scan path (refer to the third row in Table 5.14). The output of the scan path is the serial scan output "dbg_so". At the top-level, we chain the test wrappers together using this serial scan input and scan output. The resulting serial chain is connected to the TAP controller as a user-defined data register of the TAP controller at the chip-level, as shown in Fig. 5.2. The SIM takes care of concatenating all parallel scan chains into a single-bit scan path.

5.7.5 Local Clock Gate

Accessing the manufacturing test scan chains inside the module from the functional test port imposes two constraints on the on-chip communication interconnect. First, for every write operation to the scan chains, a simultaneous read operation has to occur to not lose the information that was previously contained in the scan chains. Communication interconnects normally do not support this type of simultaneous write-and-read operation. Second, once the manufacturing test scan chains inside a module are activated, the content of these scan chains shift every clock cycle.

The communication interconnect may however not be able to guarantee that new input data arrives at the inputs of the scan chains on time, or that the state data that is available on the outputs of the scan chains is read in time via the functional test port. This may cause invalid data to be shifted into the scan chains or valid state data from the scan chains to be lost.

Table 5.13 Functionality of the scan input multiplexer

input_select[1:0]	chain_config[1:0]	Chain configuration
00	00	→ PIO →
00	01	→ PIO → memory chains →
00	10	→ PIO → logic chains →
00	11	→ PIO → logic chains → memory chains →
01	00	→ Controller → PIO →
01	01	→ Controller → PIO → memory chains →
01	10	→ Controller → PIO → logic chains →
01	11	→ Controller → PIO → logic chains → memory chains →
10	00	→ Controller → PIO →
10	01	→ Controller → PIO → memory chains →
10	10	→ Controller → PIO → logic chains →
10	11	→ Controller → PIO → logic chains → memory chains →
11	--	Reserved for future use

– don't care value

Table 5.14 Input/output functionality of the scan input multiplexer

input_select	Chain input	Chain output
00	data[a]	so
01	si	so
10	dbg_si	dbg_so
11	Reserved for future use	

[a] On writes, the data available on the DTL write port. On reads, zeros

We have chosen to let the controller in the test wrapper control the module clock using a local clock gate. The controller gates the module clock when new data does not arrive on time or when the state data is not read on time. This approach effectively prevents shifting in invalid data or dropping valid state data, and therefore does not impose additional constraints on the on-chip communication interconnect.

5.7.6 Protocol-Specific Controller

The protocol-specific controller implements the module-level state access (refer to Sect. 4.1.2). A data register in this controller, called "psc_r", drives the control signals to the SIM, PIO unit, and the module itself. In the SOC functional mode, the operating mode of the module can be fully controlled by performing a write operation on this register using the functional test port on the test wrapper. This includes activating and accessing the manufacturing test scan chains inside the module. The functional ports of the module are tied-off using the PIO unit when the scan chains are activated to ensure that no accidental functional transactions to or from the module occur during state access. In the manufacturing test and the scan-based debug modes, this data register is bypassed and the control signals to the SIM, PIO unit, and the module are

connected directly to the control outputs of the global TCB (refer to Sect. 5.2.3). The internal control signals that are generated by the protocol-specific controller in each of these modes are listed in Table 5.15.

The controller supports (1) read operations to unload the content of the scan path inside the module, and (2) write operations to load new data into this scan path. Figure 5.29 shows the address map for the functional test port on the test wrapper, where W_D is the data width of the functional test port.

The "psc_r" register is mapped to address `0x0` on this port. The signals that are controlled from this control register are listed in Table 5.15. The top of the address range of this functional test port, A_{FF}, is given by Eq. 5.16, in which the number of scannable flip-flops in the module is represented with FF.

$$A_{FF} = \left\lceil \frac{FF}{T} \right\rceil \text{words} \tag{5.16}$$

Note that we need to ensure at design time that the value of T is chosen smaller than or equal to the data width W_D of the functional test port to ensure we can extract all T state bits in parallel using the functional test port. Read operations on this test port extract the state of the module for transport across the functional communication interconnect. Write operations to this test port load a new state into the module.

A debug engineer can use the steps below to access and optionally modify the state of all flip-flops in the module (refer to Listing 5.1):

1. Configure the test wrapper for module-level state access using a write operation to the control register "psc_r" at address `0x0` of the functional test port. Programming this register with the value `0xF7` switches the module to logic & memory test shift mode (lines 1 and 2).
2. Perform A_{FF} read operations in the address range "0x1-A_{FF}" to unload the complete flip-flop state of the module (lines 3 to 5).
3. Optionally perform A_{FF} write operations to the address range "0x1-A_{FF}" to load a complete state into the module (lines 6 to 8).
4. Reconfigure the test wrapper to functional mode using a write operation to the control register. Programming this register with the value `0x0` switches the module to functional mode (line 9).

The protocol-specific controller is fully scannable and is included in the scan path during manufacturing test to ensure the highest possible fault coverage.

5.8 Clock and Reset Control

5.8.1 Overview

The CRGU has three main functions: (1) generate the on-chip clocks with the required frequencies, (2) generate on-chip reset signals for a controlled initialization of each clock domain, and (3) allow clock and reset control for manufacturing test and debug.

Table 5.15 Signals generated by the protocol-specific controller depending on the (sub)mode of the test wrapper

Operating (sub)mode	Controller-generated internal signals							
	clock_gate_en	int_mem_test_en	int_intest_en	int_se	int_mem_se	mode[1:0]	input_select[1:0]	chain_config[1:0]
Functional mode	[a]	psc_r[7]	psc_r[6]	psc_r[5]	psc_r[4]	psc_r[3:2]	00	psc_r[1:0]
Manufacturing test modes								
⋆ Memory test normal mode	1	1	1	0	0	10	01	01
⋆ Memory test shift mode	1	1	1	0	1	01	01	01
⋆ Logic test normal mode	1	0	1	0	0	10	01	10
⋆ Logic test shift mode	1	0	1	1	0	01	01	10
⋆ Logic & memory test normal mode	1	1	1	0	0	10	01	11
⋆ Logic & memory test shift mode	1	1	1	1	1	01	01	11
⋆ Module-external test normal mode	1	0	0	0	0	11	01	00
⋆ Module-external test shift mode	1	0	0	1	0	01	01	00
Debug modes								
⋆ Memory debug normal mode	1	1	1	0	0	10	10	01
⋆ Memory debug shift mode	1	1	1	0	1	01	10	01
⋆ Logic debug normal mode	1	0	1	0	0	10	10	10
⋆ Logic debug shift mode	1	0	1	1	0	01	10	10
⋆ Logic & memory debug normal mode	1	1	1	0	0	10	10	11
⋆ Logic & memory debug shift mode	1	1	1	1	1	01	10	11

[a] Enabled on access from the functional test port or when "psc_r(6)" is cleared, disabled otherwise

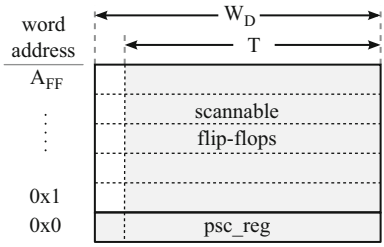


Fig. 5.29 Memory address map of the CSAR PSC

Figure 5.30 shows an overview of the CRGU as used in the on-chip CSAR hardware debug architecture. The CRGU implementation has been designed to meet debug infrastructure requirements IR-18 and IR-24 that are identified in Chap. 4. This CRGU is consequently parameterizable in the number of clock signals C.

One or more PLLs [2] generate the on-chip clock signals "in_clk" with the required frequencies from an external clock source "ext_clk". Each generated clock signal is routed to a clock control slice (CCS) and a reset generation unit (RGU). In addition to the clock signals "in_clk", the CCSes and RGUs receive the combination of the "locked" status signals from the PLLs, the asynchronous and external reset signal "ext_rst_an", and the "dbg_reset" control signal from the TAP controller (refer to Fig. 5.3). The dependency on these signals ensures the assertion of the reset of all CCSes and, via the RGUs, of all clock domains in the SOC when the PLLs are not yet locked to the external clock source, when the external reset is asserted, or when the TAP controller asserts its debug reset control output. The operation of the CCSes is controllable and observable from the CRGU TCB.

Listing 5.1 Example State Dump Code

```
1 address = PRODUCER_TEST_PORT;
2 *address = 0xF7;
3 for (int i=1; i<=A_FF; i++) {
4    old_state[i-1] = *(address+i);
5 }
6 for (int i=1;  i<=A_FF; i++) {
7    *(address+i) = new_state[i-1];
8 }
9 *address = 0x00;
```

Note that we choose not to reset the PLLs on the assertion of the "dbg_reset" control signal to prevent that they lose their lock with the external clock source "ext_clk" (refer to Fig. 5.30). The PLLs are only reset by the assertion of the external reset control signal "ext_rst_an". If the PLLs are reset by the debug reset signal "dbg_reset", then they lose their lock with the external clock source. Each time the PLLs are reset, it takes an unpredictable amount of time before all PLLs have reacquired their lock. This introduces a timing uncertainty that may unnecessarily limit the reproducibility of certain debug experiments (refer to complicating factor CF-7 on page 61).

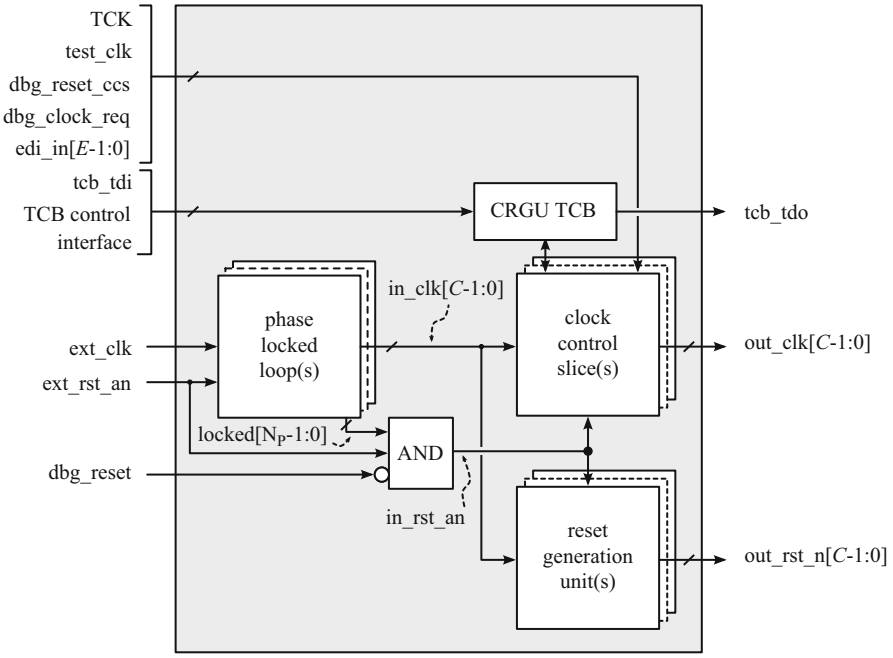


Fig. 5.30 Block diagram of a clock and reset generation unit

Table 5.16 Fields in the CRGU TCB

Name	Width	Offset	Description
clock_mode	2	3*C*	Clock mode `11`: Debug mode `10`: Test mode `0-`: Functional mode
dbg_clock_req_en	*C*	2*C*	Request enable per clock `1`: Enable `0`: Disable
dbg_clock_stop_en	*C*	*C*	Stop enable per clock `1`: Enable `0`: Disable
dbg_clock_status	*C*	0	Debug status per clock `1`: Clock has stopped `0`: Clock is running

C number of clocks, – don't care value

5.8.2 CRGU Test Control Block

The CRGU TCB is part of the same serial chain as the global TCB at the chip-level, and can be accessed from the TAP (refer to Fig. 5.2). Table 5.16 shows its register fields. The control outputs of this TCB are set to zero on reset. We explain the use of these fields when we describe the functionality of a CCS below.

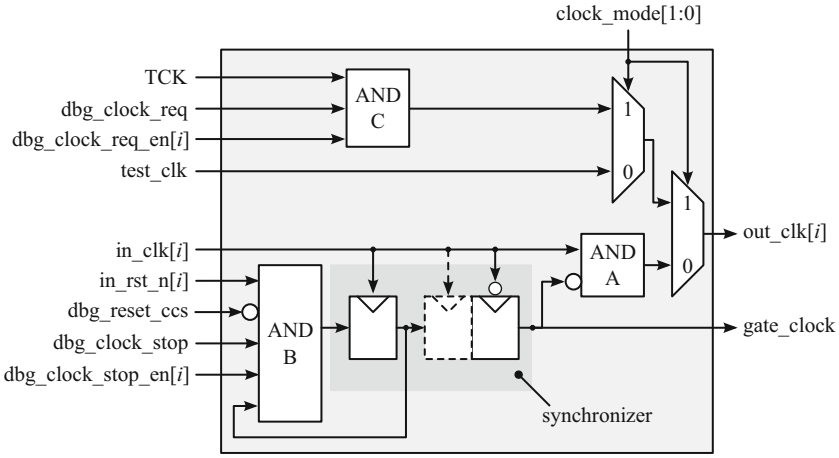


Fig. 5.31 Block diagram of a clock control slice

5.8.3 Clock Control Slices

A CCS has three functions: (1) to conditionally gate the functional output clock, (2) to switch to an external test clock, and (3) to switch to a debug scan clock, which is conditionally enabled from the TAP controller and the CRGU TCB. A CCS is configured from and observed by the CRGU TCB, and is dynamically controlled by the TAP controller. Figure 5.31 shows a block diagram of a single CCS.

The lower half of Fig. 5.31 is used in *functional mode*. The input clock "in_clk[i]" passes through AND-gate A and out on the functional clock output "out_clk[i]" when the LSB of the "clock_mode" control signal in the CRGU TCB is set to "0". This AND-gate is controlled from the output of a synchronizer. This synchronizer is parameterizable in the number of flip-flops that are used to synchronize the incoming, asynchronous control signals to the functional clock "in_clk[i]". The synchronizer has to contain at least two flip-flops to reduce the chance of meta-stability on the control input of AND-gate A [7]. The number of flip-flops in the synchronizer can be increased to further reduce the chance of meta-stability on the control input of AND-gate A at the cost of an increased delay in gating the functional clock signal. A falling-edge triggered flip-flop is used as the last flip-flop in the synchronizer to prevent gating the functional clock when it is high. This can otherwise introduce glitches on the clock output signal "out_clk[i]" and corrupt (part of) the state in the corresponding clock domain.

The output clock "out_clk[i]" is gated when an event arrives on its "dbg_clock_stop" input, provided that the control inputs on AND-gate B in front of the synchronizer are set appropriately. Figure 5.32 shows the effect of an event on the "dbg_clock_stop" input on the output clock signal "out_clk[i]", when both reset input signals "in_rst_an" and "dbg_reset_ccs" are deasserted, and stopping of this particular functional clock is enabled by setting the "dbg_clock_stop_en[i]" field in

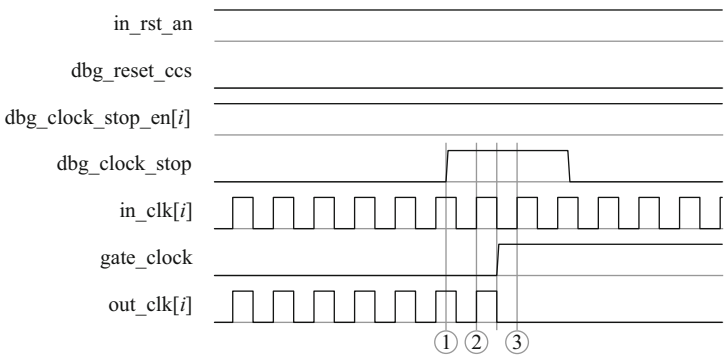


Fig. 5.32 Timing diagram of clock stopping under TAP control

the CRGU TCB. The "dbg_clock_stop_en" field in the TCB allows a configurable subset of the clock domains to be stopped when the "dbg_clock_stop" input is asserted. The "dbg_clock_stop" control signal can be asserted using the TAP controller by first activating the DBG_CLOCK_STOP instruction (refer to Table 5.1) and subsequently entering the SDR state of the TAP controller. The TAP controller asserts the "dbg_clock_stop" control signal as long as it is in the SDR state. It deasserts this control signal in all other states. The timing diagram in Fig. 5.32 is of a CCS with a single rising-edge triggered flip-flop and a single falling-edge triggered flip-flop as its synchronizer.

The "gate_clock" status signal can be captured in the CRGU TCB to determine from the TAP whether a particular clock has been stopped or not. The CRGU TCB continues to operate when all on-chip clocks have been stopped, because the TCBs operates on the TCK clock signal that is never stopped.

It takes a rising edge and a falling edge on the "in_clk[i]" clock signal to propagate the asserted "dbg_clock_stop" signal (①) through the synchronizer (②) and subsequently gate the output clock signal "out_clk[i]" (③). A feedback loop inside the CCS continues to gate the functional clock afterwards.

The upper half of Fig. 5.31 is used in the *manufacturing test mode* and in the *debug mode*. The "clock_mode" control signal in the CRGU TCB has to be set to "01" for the manufacturing test mode. This causes the external test clock "test_clk" to pass through each CCS to each clock domain. This mode allows complete external clock control from an automated test equipment (ATE) to achieve high fault coverage.

The "clock_mode" control signal in the CRGU TCB has to be set to "11" for the debug mode. This causes the TCK signal of the TAP to be conditionally passed to the output clock "out_clk[i]" under control of the TAP controller. This mode is illustrated in Fig. 5.33. Figure 5.33 assumes that the output clock "out_clk[i]" was stopped earlier, e.g., using the sequence shown in Fig. 5.32.

The TCK signal is passed to the clock output "clk_out[i]" of the CCS, when both the "dbg_clock_req_en[i]" field in the CRGU TCB and the "dbg_clock_req" control signal from the TAP controller are asserted (①). The "dbg_clock_req_en" configuration signals allow a configurable subset of clock signals to be activated during a debug scan operation.

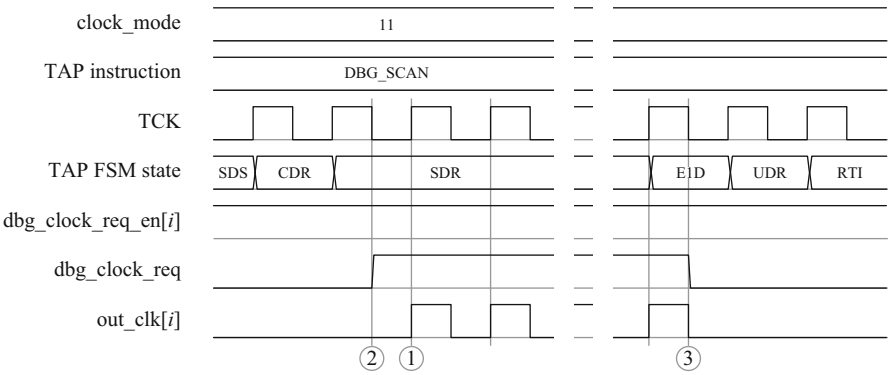


Fig. 5.33 Timing diagram of applying a clock under TAP control

The "dbg_clock_req" signal of the TAP controller (refer to Fig. 5.3) is asserted on the negative edge on the TCK signal when the DBG_SCAN instruction is the active TAP instruction and the TAP controller is in the SDR state (②). This causes the next pulse on the TCK signal to pass through the CCS to the associated clock domain (①). One clock pulse is issued to the associated clock domain for each TCK cycle that the TAP controller stays in the SDR state. The content of all flip-flops in this domain can be shifted out by keeping the TAP controller in this state for as many cycles as there are flip-flops in the associated clock domain. When the TAP controller leaves the SDR state, the "dbg_clock_req" control signal is deasserted on the first falling edge on the TCK signal (③). The reason for asserting and deasserting the "dbg_clock_req" signal on a falling edge on the TCK signal is to prevent the accidental introduction of glitches on the output of AND-gate C and thereby possibly corrupting (part of) the state of the associated clock domain.

The state of the synchronizer is synchronously reset when either the clock domain reset "in_rst_n[*i*]" is asserted, when the "dbg_reset_ccs" control signal from the TAP controller is asserted, or when the "dbg_clock_stop_en[*i*]" field in the CRGU TCB is deasserted. The "dbg_reset_ccs" signal allows the clock gate to be deactivated without having to completely reset the SOC. We use this reset functionality in Chap. 8 to resume the execution of the SOC after its state has been extracted and a possible new state has been uploaded. Deasserting the "dbg_clock_stop_en[i]" signal by reprogramming the TCB also resumes the functional execution inside the SOC. Figure 5.34 shows the reset process of a CCS under TAP control to allow the functional execution to resume in the associated clock domain.

Figure 5.34 shows that, for a CCS with a single rising-edge triggered flip-flop and a single falling-edge triggered flip-flop as its synchronizer, it takes one rising edge and one falling edge (①) on the ungated input clock "in_clk[*i*]" after the assertion of the "dbg_reset_ccs" control signal by the TAP controller (②) to reactivate the functional output clock "out_clk[*i*]" (③). The TAP controller asserts its "dbg_reset_ccs" control signal (①) in the SDR state when the DBG_RESET_CCS instruction is the active TAP instruction. The "dbg_reset_ccs" control signal is deasserted in all other states (e.g., ④).

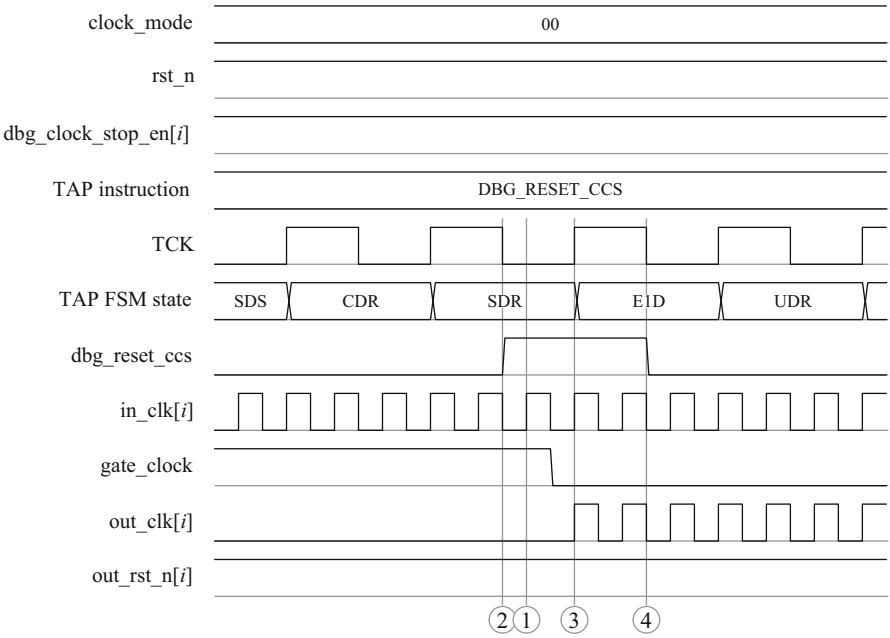
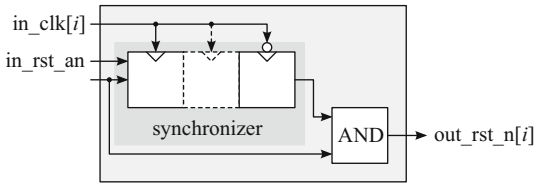


Fig. 5.34 Timing diagram of resetting the clock control slices under TAP control

Fig. 5.35 Block diagram of a reset generation unit



5.8.4 *Reset Generation Unit*

Each RGU generates a reset signal "out_rst_n[i]" for the corresponding clock domain i with clock signal "out_clk[i]". An RGU ensures the proper initialization of the state of the digital logic in the corresponding clock domain. Figure 5.35 show a block diagram of a single RGU.

The "out_rst_n[i]" output signal is asserted asynchronously as soon as the asynchronous reset input signal "in_rst_an" is asserted. The output is however synchronously deasserted on the falling edge of the functional clock, after the deassertion of the input reset signal has completely propagated through the synchronizer. Similar to the CCS, the RGU contains a parameterizable synchronizer to reduce the chance of meta-stability on the reset output signal "out_rst_n[i]" when the reset input signal "in_rst_an" is deasserted. This input can be deasserted using the "dbg_reset" control signal from the TAP controller (refer to Fig. 5.3).

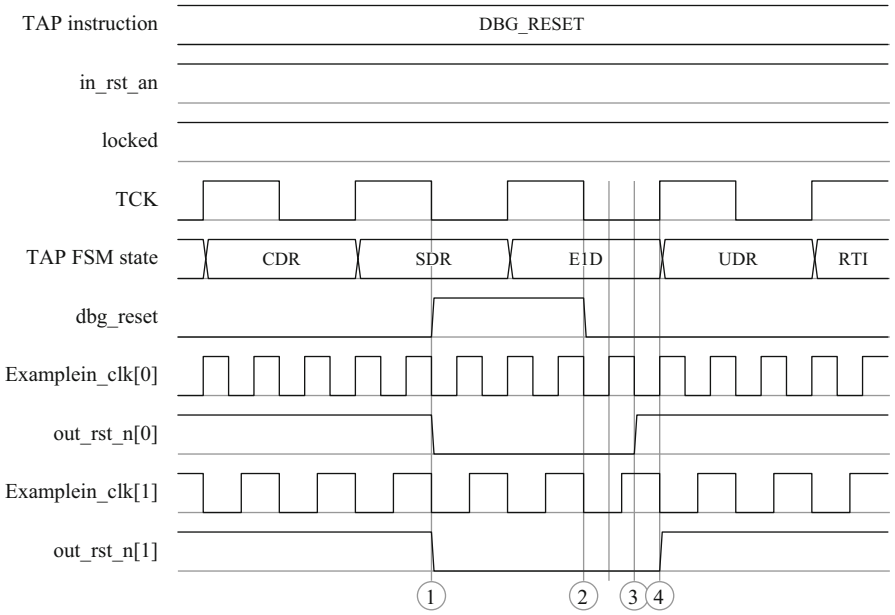


Fig. 5.36 Timing diagram of a functional reset under TAP control

Figure 5.36 illustrates the resulting reset operation in response to a functional reset request from the TAP controller for two clock domains with different functional frequencies.

The timing diagram in Fig. 5.36 is of two RGU with a single rising-edge triggered flip-flop and a single falling-edge triggered flip-flop each as their internal synchronizer.

When the DBG_RESET instruction is the active TAP instruction, the "dbg_reset" control signal is asserted in the SDR state of the TAP controller (①) and deasserted in all other states (e.g.,②). In response to the assertion and deassertion of the "dbg_reset" control signal by the TAP controller, the two example reset signals "out_rst_n[0]" and "out_rst_n[1]" are asynchronously asserted as soon as the "dbg_reset" signal is asserted (①). They are synchronously deasserted only after a rising and a subsequent falling edge on their respective functional input clocks "in_clk[0]" (③) and "in_clk[1]" (④) have occurred.

5.9 Summary

In this chapter we presented an overview of the on-chip CSAR hardware debug architecture, including implementation details on its debug components. We reused the IEEE Std. 1149.1 TAP, its associated controller, two TCBs, and multiple TPRs as our DCSI. We then described in detail our new debug components to support the

CSAR debug approach, comprising communication monitors, PSIs, and the event distribution interconnect. We showed how these components can be configured, controlled, and queried using off-chip debugger software through our DCSI. We reused existing design-for-test (DfT) and DfD methods to integrate our debug components in an SOC design that supports both manufacturing test and debug state access via the embedded scan chains. In addition, we added new functionality to the test wrapper around each module to allow the functional extraction of the state of a module via a DTL test port on the test wrapper.

References

1. ARM Limited. *AMBA AXI Protocol Specification*, June 2003.
2. Dean Banerjee. *PLL performance: simulation and design; 4th ed.* Dog Ear Publishing, Indianapolis, IN, 2006.
3. 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*, pages 1–10, Antibes, France, October 2006. IEEE Press.
4. Clifford E. Cummings. Simulation and synthesis techniques for asynchronous fifo design, 2002.
5. Andreas Hansson and Kees Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 99–108, New York, NY, USA, 2009. ACM.
6. IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture-IEEE Std 1149. 1-2001*. IEEE Press, 2013.
7. David J. Kinniment and J.V. Woods. Synchronisation and arbitration circuits in digital systems. *Proceedings of the IEE*, 123:961–966, 1976.
8. NXP Semiconductors. *CoReUse 5.0 Device Transaction Level (DTL) Protocol Specification. Version 5.0*, December 2009.
9. OCP International Partnership. *O*pen Core Protocol Specification. v3.0, 2009.
10. C.E. Shannon. Communication in the presence of noise. *Proc. Institute of Radio Engineers*, 37(1):10–21, January 1949.
11. Bart Vermeulen and Sjaak Bakker. Debug architecture for the En-II system chip. *IET Computers & Digital Techniques*, 1(6):678–684, 11 2007.
12. Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. International Symposium on Computer Architecture*, 2004.

Chapter 6
Design-for-Debug Flow

Abstract In this chapter we describe a *DfD flow* and associated tools, which automate the generation and instantiation of our debug components in an SOC. We start with an overview of our DfD flow in Sect. 6.1. In Sect. 6.2 we describe the generic DfD tool architecture and the third-party functionality we reused. We subsequently introduce custom tools in our DfD flow. Each tool is explained separately using (1) an overview of its input and output files, (2) details on its configuration, (3) a description of the DfD algorithms used, and (4) an example execution run. Each example execution run is taken from the case study described in Chap. 8. We conclude this chapter with a summary in Sect. 6.6.

6.1 Overview

The CSAR on-chip debug architecture consists of a large number of interacting components. To support the CSAR debug approach, the SOC design team has to correctly add these components at design time to the SOC implementation. The effort to implement support for the CSAR debug approach can be a burden for the often already scarce design resources in an SOC design team. In this chapter we therefore describe a *DfD flow* and associated tools, which automate the generation and instantiation of our debug components in an SOC. Each step in this flow is configurable. By automating this flow, the benefits of obtaining a highly-debugable SOC can outweigh the effort required to implement this debug support.

Please note that our DfD flow is a proof-of-concept flow. A key feature of this flow and difference with existing commercial, off-the-shelf (COTS) tools is the generation of the debug components and the subsequent consolidation of all information about the debug support in the resulting SOC in a single *boundary-scan-level DfD configuration file* at the end of the flow. This information is necessary for the off-chip debugger software described in Chap. 7. Variations on and alternatives to (steps in) this DfD flow are possible, provided that they still contribute to this DfD configuration file.

Figure 6.1 provides an overview of the steps required to add support for the CSAR debug approach to an SOC. These steps are:

1. The modules used in the SOC are either manually created or automatically generated based on a user specification (refer to Fig. 1.4). In this step, the designer can instrument the module with both computation- and communication-centric debug modules. This step is described in Sect. 6.3.

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6_6,

2. Each module is subsequently prepared for communication-centric debug by wrapping it in a debug wrapper (refer to Sect. 5.6). Each debug wrapper can include communication monitors and PSIs on the external communication port(s) of the module. An EDI node is also instantiated inside the debug wrapper when needed.
3. Each module is synthesized to a gate-level implementation. During this process, we use a COTS synthesis tool to translate the RTL description inside the hardware description language (HDL) files into logic gates using a specific process technology library. During this step, all resulting flip-flops in the gate-level implementation are replaced with scannable flip-flops and chained together in parallel logic scan chains to support manufacturing test. The synthesis process is controlled for each module using a *synthesis configuration file* and outputs a gate-level implementation of each module. The order of the flip-flops in the logic scan chains is written to a *logic scan chain configuration file*. We do not describe this synthesis step further in this book as it uses common COTS synthesis tools and DfT techniques.
4. Each *module* is subsequently prepared for manufacturing test and debug state access, by wrapping it in a test wrapper (refer to Sect. 5.7). The test wrapper generation process is controlled using a *test wrapper configuration file*.
5. All wrapped modules are integrated in a single, top-level module (refer to Sect. 5.1).
6. The resulting top-level module is then integrated in a single, chip-level module, together with the global TCB and a chip-specific CRGU (refer to Sect. 5.1).
7. The final step in our DfD flow is the integration of the chip-level module in a single, boundary-scan-level module. This step wraps the functional I/O interface of the chip-level module with a boundary-scan chain and adds an IEEE Std 1149.1 TAP controller (refer to Sect. 5.1).

The resulting boundary-scan-level module can subsequently be mapped to either silicon or to an FPGA platform. We do not describe these steps further as they use known COTS layout tools.

Because of the similarities between the functionality of the tools, required for steps 2, 4, 5, 6 and 7, we describe only step 2 in detail in Sect. 6.4. Details on steps 4 to 7 are provided in Appendix A.

6.2 DfD Tool Architecture

6.2.1 Overview

The tools used in the DfD flow in Fig. 6.1 share the same functionality. Typically a DfD tool starts by reading its configuration file. It then reads one or more HDL files of existing components and optionally a DfD configuration file for each component. These components are subsequently instantiated in a new, parent module. The implementation of this parent module depends on these components and the content of

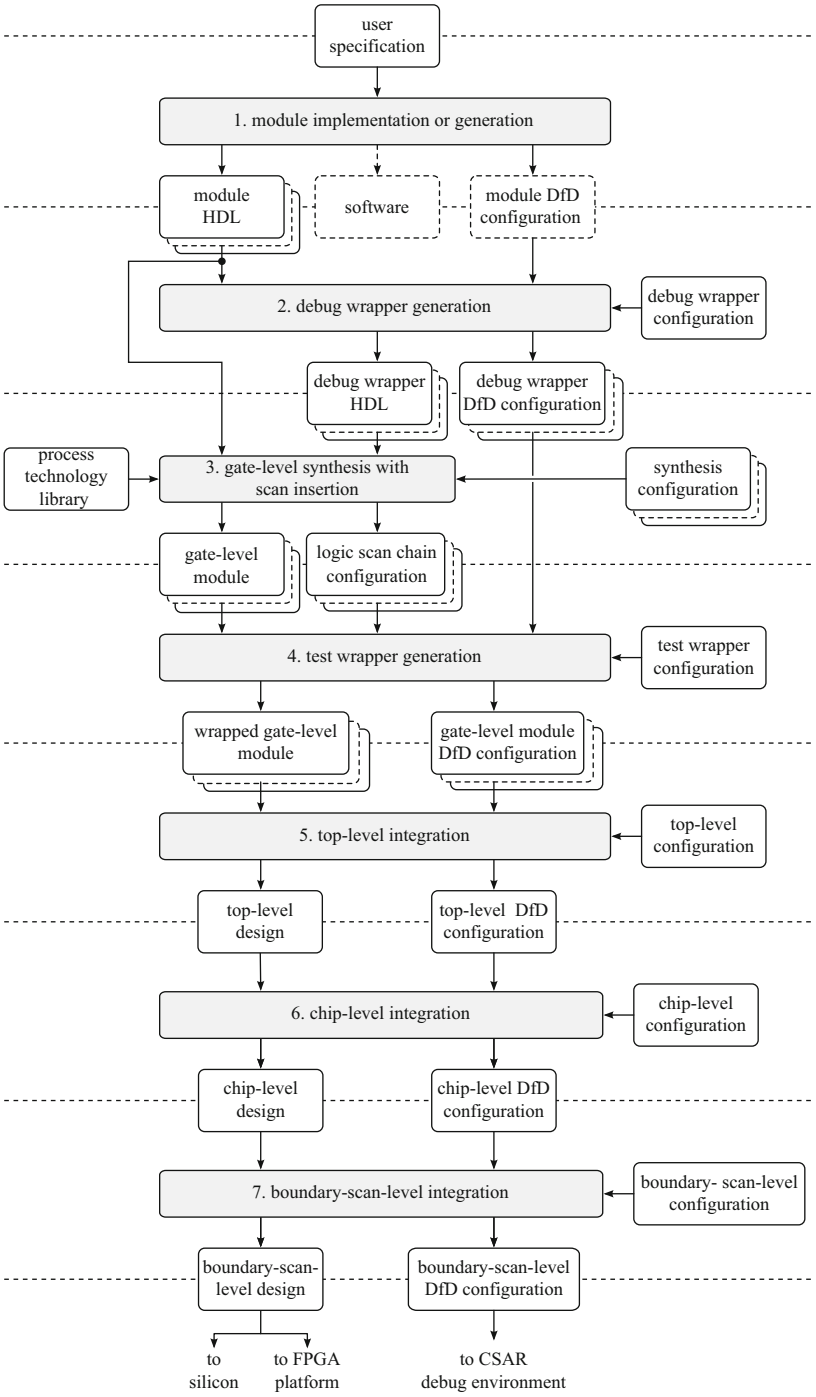


Fig. 6.1 Generic CSAR design-for-debug flow

the configuration file. Once the tool has implemented the parent module, it writes this module to a set of HDL files, together with its DfD configuration file. Because of these functional similarities between the tools, each tool can use the same source code template. We describe this template in Sect. 6.2.2, the module configuration and software classes in respectively Sects. 6.2.3 and 6.2.4, and the dependency of our tools on external libraries in Sect. 6.2.5.

6.2.2 Tool Architecture

Listing 6.1 shows the source code template for the main function of each CSAR DfD tool. The input to each tool is a set of command line arguments. Each tool starts on line 2 with the construction of an appropriate logger object. This logger is used by the other functions in the tool to log information, warning, and error messages to an output device. The FlowLogger class logs these messages on the command console. We can use other loggers to, for example, log messages in a GUI or in a file.

A command line parser object is constructed on line 3, which is subsequently used on line 4 to parse the options provided by the user on the command line. Next, an appropriate builder object is constructed for each command line argument on line 7, using the argument as the name of the configuration file and the logger object. This builder implements the IBuilder interface that is shown in Fig. 6.2 and only

Listing 6.1 Pseudo-code template for a CSAR DfD tool

```
1  public static void main(String[] args) {
2    ILogger logger = new FlowLogger();
3    CmdLineParser parser = new CmdLineParser();
4    parser.parse(args);
5    ...
6    for (String configurationFileName: parser.getRemainingArgs()) {
7      IBuilderbuilder = new ChipLevelBuilder(configurationFileName,logger);
8      builder.build();
9    }
10 }
```

contains the `build` function. This function is called by the tool on line 8. Please note that for reasons of brevity Listing 6.1 does not show the error handling that takes place in each tool. Appropriate messages are generated when errors occur during the construction of the logger, the parser, or the builder, the parsing of the command line arguments, or the execution of the `build` function.

Figure 6.2 shows the dependencies between the IBuilder interface, the AbstractBuilder abstract class, and as an example, the ChipLevelBuilder class. Functionality that is shared between the builders used in the different DfD tools is implemented in the AbstractBuilder abstract class. This includes reading an HDL file, writing an HDL architecture, component, configuration, and entity, and creating the required directory structure. This last function is for example used when the HDL files need to be placed in a specific directory structure, required by company design flows

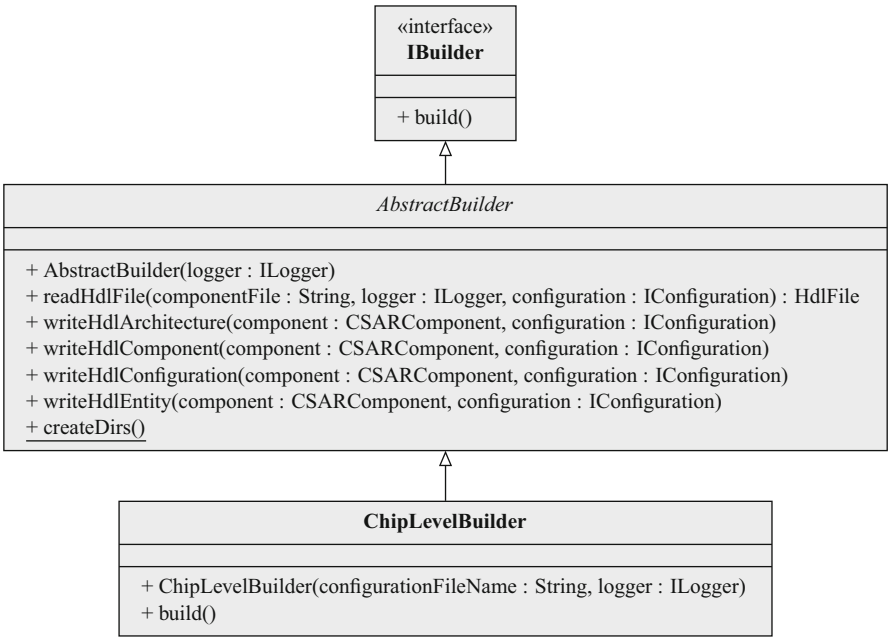


Fig. 6.2 The IBuilder interface, the AbstractBuilder abstract class, and the ChipLevelBuilder class

(e.g., CoReUse [9]). This function is a static class function as its operation does not use any specific state information from the individual builders. The ChipLevelBuilder class implements a tool-specific `build` function. We discuss the implementation of this function for each tool in Sect. 6.4 and Sect. A.1 to A.4.

6.2.3 CSAR Configuration Classes

The `build` function of each dedicated builder typically starts by reading the tool configuration file. Its file name is passed as an argument to the constructor of this builder. Figure 6.3 shows the dependencies between the IConfiguration interface, the AbstractConfiguration abstract class, and as an example, the ChipLevelConfiguration class. The builder uses these classes to manage the configuration of the corresponding DfD tool.

An IConfiguration object[1] provides access to the specified name for the parent module, the output directory, and the output file name stored inside a tool configuration object. It furthermore indicates whether a CoReUse directory structure should be used and provides a `makeAbsolute` function to transform a relative file name

[1] We refer in this book to an object that implements the I<x> interface as an I<x> object.

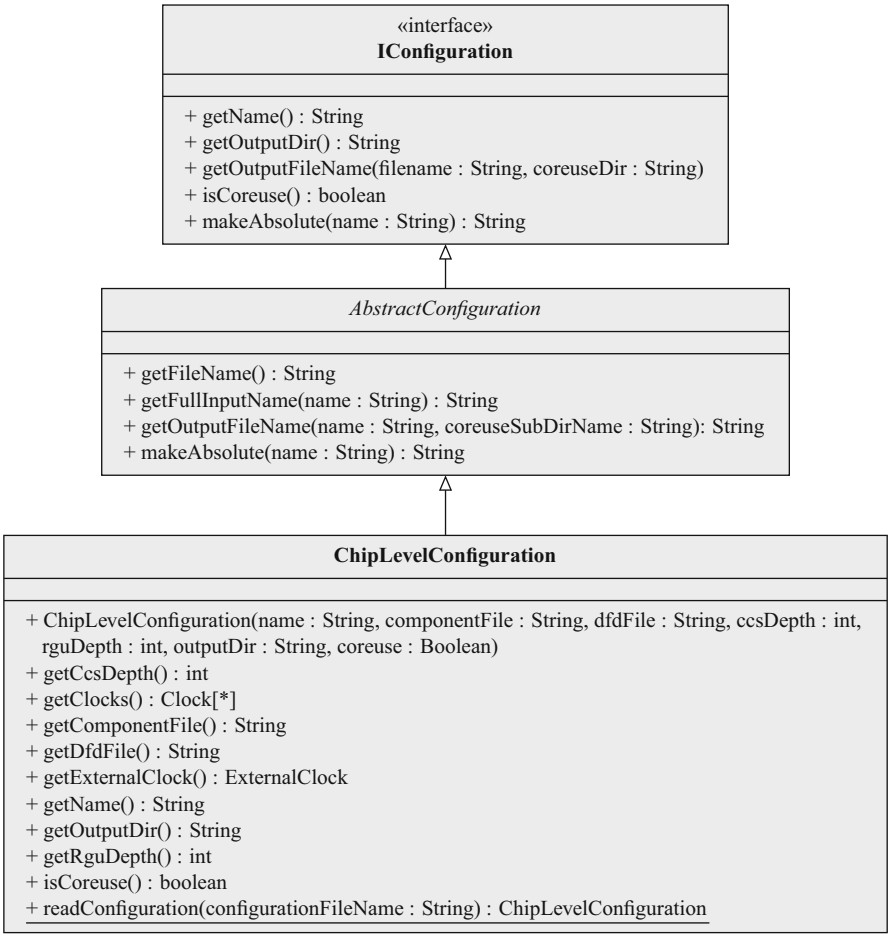


Fig. 6.3 The IConfiguration interface, the AbstractConfiguration abstract class, and the ChipLevelConfiguration class

into an absolute file name. Functionality that is shared between the configurations of each DfD tool is implemented in the AbstractConfiguration abstract class. This includes retrieving the file name, the full name of the configuration file, the output file name, and the implementation of the `makeAbsolute` function.

The ChipLevelConfiguration class implements the functions that relate to tool-specific configuration options. For the configuration of the chip-level integration tool (refer to Sect. A.3.2), this includes retrieving the pipeline depth of the synchronizers in the CRGU and RGU, details about the user-specified clocks and the external clock (stored in respectively Clock and ExternalClock class objects that we do not detail further), the names of the top-level component file and its optional DfD configuration

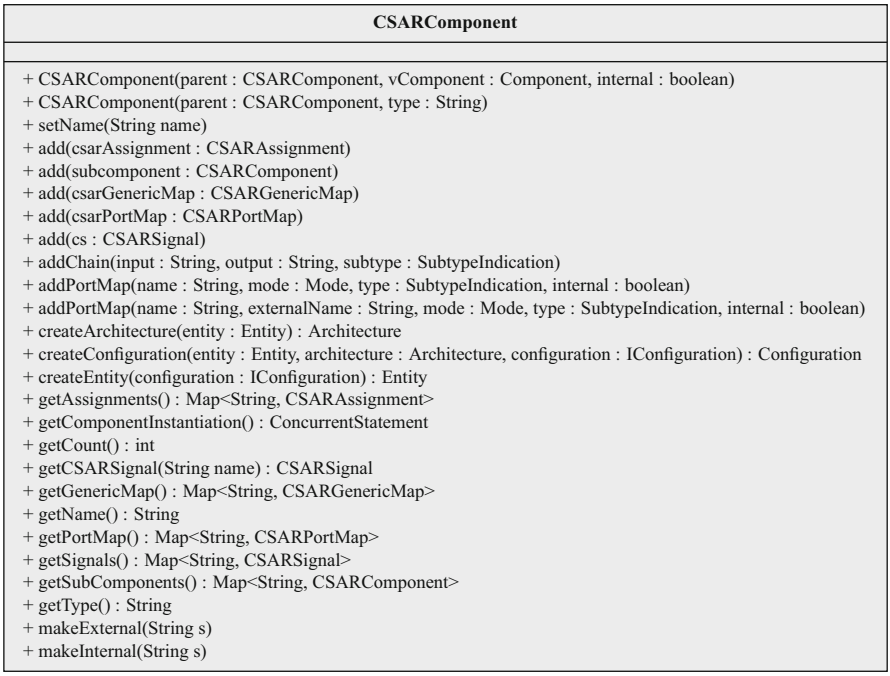


Fig. 6.4 The CSARComponent class

file to read, the name of the chip-level module, the output directory name, and whether a CoReUse directory structure is requested.

A builder parses a configuration file by calling the static `readConfiguration` function of the appropriate configuration object, and passing the configuration file name as the argument. This function returns an object of the appropriate configuration class. Afterwards, the builder can query this object for the values of options related to the construction of the parent module.

Note that the (DTDs) of all tool and DfD configuration files are provided in Appendix A.5.

6.2.4 CSAR Software Classes

A detailed description of how each parent module is built, is provided in Sect. 6.4 and in Sect. A.1 to A.4. The `build` function in each builder first creates the parent module in memory, using the CSARComponent class shown in Fig. 6.4. A builder then writes this parent module to HDL files using the `writeHdl*` functions of the AbstractBuilder abstract class. The CSARComponent class contains five main function groups:

1. Constructors: A builder use the CSARComponent object constructors to either construct a module based on an existing component or by instantiation in a parent module. The name of the component can be modified with the function `setName`. The former constructor creates a parent module, which instantiates the existing component. The signals on the interface of the existing component are either brought out to the interface of the parent or are registered as internal, depending on the value of the Boolean argument `internal`. The latter constructor creates a new module and registers it with a unique instantiation name with the parent. This instantiation name is the combination of a unique CSARComponent object identifier and the type of the component.
2. Subcomponent addition functions: A builder uses the `add*` functions to add (a) assignment statements, component instantiations, and signals to the architecture of the component, (b) generics and ports to its interface, and (c) a generic map and port map to its instantiation in a parent module. The `addChain` function adds a serial chain to the module, by concatenating the component instantiations in the module based on the specification of the input signal and output signal on each. Adding a component instantiation to a module causes the signals on its interface to be registered as signals of the parent module. In addition, these signals may become part of the interface of the parent module as well, provided that the `internal` flag of these signals are set to `false`.
3. Interface modification functions: A builder uses the `make*` functions to change the internal flag of a signal that indicates whether it is an internal signal or part of the module's I/O interface.
4. Query functions: A builder can query a CSARComponent object using its `get*` functions.
5. Creation functions: A builder creates the HDL architecture, component, configuration, and entity objects of the constructed parent module using the `create*` functions. These functions return objects that are defined by the vMAGIC library (refer to Sect. 6.2.5).

6.2.5 External Libraries

All our DfD tools are written in Java and leverage a number of external libraries to reduce the effort involved in creating the DfD flow shown in Fig. 6.1. These external libraries include:

- The ANTLR tool [10] to read the logic scan chain configuration files in (DEF) format [1].
- The JArgs command line option parsing suite for Java [12] to parse command line options.
- The Java Regular Expression library [4] to perform regular expression matching.

- The vMAGIC library [11] to read and write VHDL files. The tools currently work for the VHSIC hardware description language (VHDL) [8]. The concepts used in these tools can however be easily extended to also support the Verilog HDL [7].
- The XStream library [2] to read and write tool and DfD configuration files in the (XML) format.

6.3 Module Implementation or Generation

Each module in the SOC has to be either manually implemented or automatically generated based on a user specification. During this step, the designer has the option to instrument the module with both computation- and communication-centric debug instruments. For example, the processor and memory modules used in the case study described in Chap. 8 have been implemented manually at RTL. The processor was instrumented to output a computation debug event whenever the processor stalls. More elaborate instrumentation can be included based on an analysis of the module. Embedded memories are made accessible by design for manufacturing test and debug, by either implementing them using flip-flops or by wrapping them with a dedicated memory test wrapper (refer to Sect. 5.7.2).

In our case study in Chap. 8, we use a NoC as the communication interconnect between the processors, the memories, and other peripherals. The HDL files of this NoC are generated and instrumented with CSAR debug components using its own automated generation flow [3, 6]. Although we focus in the remainder of this section on the generation of this NoC, the concepts we use can however also be applied to other flows that generate a custom communication interconnect for a particular SOC.

Figure 6.5 provides an overview of the NoC generation flow [5]. In step 1, this flow creates a specific network topology, given (a) an *architecture specification* containing a description of the modules to be connected to the NoC and its interface ports with their associated communication protocol, and (b) a *communication specification* containing a set of communication use cases (i.e. sets of concurrent applications). The resources of this network are dimensioned in step 2 to handle the communication requirements for all use cases. The resulting network is instrumented for debugging in step 3. The flow instantiates a debug wrapper around each building block of the NoC with an EDI node. This EDI node is connected to the EDI nodes in the debug wrappers around neighboring NoC building blocks. This ensures that the EDI has the same topology as the NoC. The outputs of the NoC flow are generated in step 4. These outputs are (1) *HDL files* that implement the NoC with fully-specified connections between masters, local buses, monitors, network interfaces, PSIs, routers, slaves, and EDI nodes, (2) *use case configuration software* to configure the NoC at run time, and (3) a *NoC DfD configuration file* containing a description of the EDI in the resulting NoC architecture, and a functional description of the NoC, when it is configured with the use case software.

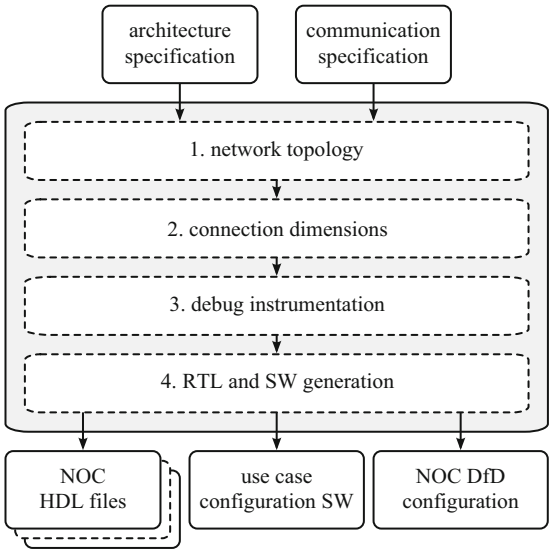


Fig. 6.5 Example communication interconnect generation flow [5]

6.4 Debug Wrapper Generation

6.4.1 Tool Overview

We provide a tool to generate a custom debug wrapper for modules (refer to Fig. 6.1). This tool reads the *debug wrapper configuration file* that is explained in Sect. 6.4.2. Based on this configuration, it reads in the component definition of the module and optionally its DfD configuration file that specifies the debug support embedded in the module. It outputs the HDL files of the debug wrapper and a *debug wrapper DfD configuration file* with information about the debug support inside the wrapper and the module.

6.4.2 Debug Wrapper Configuration

Listing 6.2 shows an example debug wrapper configuration file, which we use to generate the debug wrapper for a processor tile in our case study in Chap. 8.

Listing 6.2 Example configuration file for debug wrapper generation

```
<debugWrapperConfiguration>
  <name>rdt_dlx_tile_producer_debug_wrapper</name>
  <componentFile>../../../../noc/src/vhdl/data/rdt_dlx_lib/rdt_dlx_tile/INTERFACE/ ⟵
rdt_dlx_tile_cmp_pkg.p.vhdl</componentFile>
  <dfdFile>rdt_dlx_tile.dfd</dfdFile>
  <generics>
    <generic>
      <name>MBLOCK_SIZE_BITS</name>
      <value>5</value>
    </generic>
    <generic>
      <name> SBLOCK_SIZE_BITS</name>
      <value>5</value>
    </generic>
    <generic>
      <name>MMASK_WIDTH</name>
      <value>4</value>
    </generic>
    <generic>
      <name>SMASK_WIDTH</name>
      <value>4</value>
    </generic>
    <generic>
      <name>MMEM_ADDR_WIDTH</name>
      <value>10</value>
    </generic>
    <generic>
      <name>SMEM_ADDR_WIDTH</name>
      <value>14</value>
    </generic>
    <generic>
      <name>NR_CHAINS</name>
      <value>8</value>
    </generic>
  </generics>
  <port>
    < signal>(.*)cmd_valid(.*)</signal>
    <clock>clk</clock>
    <reset>rst_n</reset>
    <psi>
      <counterWidth>5</counterWidth>
    </psi>
    <monitor>
      <type>initiator</type>
      <stateBits>2</stateBits>
      <poly>04C11D87</poly>
    </monitor>
  </port>
  <scanWidth>8</scanWidth>
  <edi>
    <layers>6</layers>
    <signal>stalled</signal>
  </edi>
  <outputDir>../../rdt_dlx_tile_producer_debug_wrapper/</outputDir>
  <coreuse>true</coreuse>
</debugWrapperConfiguration>
```

The <debugWrapperConfiguration> element indicates the start of the configuration data for the debug wrapper generation process. The <name> element specifies the name of the debug wrapper. The <componentFile> element specifies the HDL file that contains the component declaration of the module to wrap for debugging. If the specified file name does not contain an absolute directory path, then this file name is taken relative to the directory that contains the configuration file. This allows for an easy grouping of the configuration file with the component HDL file. The <dfdFile> element specifies the optional DfD configuration file for the module.

It is currently necessary to specify the values for all generic parameters on the interface of the module that is to be wrapped. This decision was made to prevent having to generate a completely-parametrized version of the debug wrapper, which introduces many corner cases that need to be properly handled and verified, complicating the DfD flow and increasing the chance of introducing faults in its implementation. Values for the generic parameters are therefore specified using the <generics> element. Lines 6 to 9 show how the "MBLOCK_SIZE_BITS" parameter is set to the value "5" for our processor tile.

The insertion of PSIs and communication monitors is controlled by the inclusion of the <port> element with <psi> and <monitor> subelements. The clock and reset signals for these debug components are specified on lines 37 to 38. For example, lines 35 to 47 specify that all ports that have a signal that matches the regular expression "(.*)cmd_valid(.*)" should be instrumented with a PSI (lines 39 to 41) and with a monitor on the initiator side of that PSI (line 43). Other valid options for specifying the location of a monitor are "target", "block", "wrapper", and "both" (refer to Sect. 5.6). For the PSI we also specify on line 40 the number of bits to use in the internal counters that keep track of the number of pending write and read transactions (refer to Sect. 5.4.2). For the DTL monitor, we further specify on lines 44 and 45 the number of state bits in its event sequencer and the polynomial to use for the checksum calculations inside its data matchers (refer to Sect. 5.3).

The number of scan chains to use for the test wrapper around the memories inside the DTL monitors is specified on line 48 (refer to Sect. 5.7.2). The number of EDI layers is specified using the <edi> element and the <layers> subelement. We can use optional <signal> subelements to specify module signals that should be routed to and/or from the EDI. Currently, each specified signal has to be a single-bit signal. For example, we specify an EDI with six layers on line 50 and we connect the signal "stalled" from the processor to all six layers of the EDI on line 51 (refer to Sect. 6.3).

The directory for the generated files is specified using the <outputDir> element. When this element specifies a relative path, then the output directory is taken relative to the directory containing the configuration file. When no output directory is specified, then the HDL files are generated in the directory containing the configuration file. The <coreuse> element specifies whether a CoReUse directory structure has to be generated in the output directory. In a CoReUse directory structure [9], the HDL output files are written in an `INTERFACE` and an `RTL` subdirectory. The generated

DfD configuration file is written in a `CSARDE` subdirectory. This option is by default turned off, causing all output files to be written in the output directory without subdirectories.

Listing 6.3 shows the complete DTD of the configuration file for the debug wrapper generation process.

Listing 6.3 DTD for the debug wrapper configuration file

```
1  < !ELEMENT debugWrapperConfiguration (name,componentFile,dfdFile?,generics?, ↩
   nrChains?,port∗,scanWidth?,edi?,outputDir?,coreuse?)>,
2  < !ELEMENT name (#PCDATA)>
3  < !ELEMENT componentFile (#PCDATA)>
4  < !ELEMENT dfdFile (#PCDATA)>
5  < !ELEMENT generics (generic+)>
6  < !ELEMENT generic (name,value)>
7  < !ELEMENT value (#PCDATA)>
8  < !ELEMENT nrChains (#PCDATA)>
9  < !ELEMENT port (signal,clock,reset,psi∗,monitor∗)>
10 < !ELEMENT clock (#PCDATA)>
11 < !ELEMENT reset (#PCDATA)>
12 < !ELEMENT psi (counterWidth)>
13 < !ELEMENT counterWidth (#PCDATA)>
14 < !ELEMENT monitor (type,stateBits,poly)>
15 < !ELEMENT type (#PCDATA)>
16 < !ELEMENT stateBits (#PCDATA)>
17 < !ELEMENT poly (#PCDATA)>
18 < !ELEMENT edi (layers,signal∗)>
19 < !ELEMENT layers (#PCDATA)>
20 < !ELEMENT signal (#PCDATA)>
21 < !ELEMENT scanWidth (#PCDATA)>
22 < !ELEMENT outputDir (#PCDATA)>
23 < !ELEMENT coreuse (#PCDATA)>
```

6.4.3 Debug Wrapper Generation Process

Listing 6.4 shows the `build` function of the DebugWrapperBuilder class in pseudo code. This function generates a debug wrapper from a debug wrapper configuration file.

Listing 6.4 Pseudo-code for the debug wrapper generation process

```
1  public class DebugWrapperBuilder extends AbstractBuilder {
2    ...
3    public void build() {
4
5      hdlFile = readHdlFile(toolConfiguration, logger);
6
7      hdlWrappedComponent = selectComponent(hdlFile, logger);
8
9      processGenerics(hdlWrappedComponent, toolConfiguration);
10
11     ios = collectIOs(hdlWrappedComponent);
12     ios.groupPorts(configuration.getPorts(), logger);
13
14     ipBlockDfD = readIpBlockDfDConfiguration(toolConfiguration, ↩
   hdlWrappedComponent, logger);
15
16     csDebugWrapper = new CSARComponent(null, toolConfiguration.getName());
17
18     csIpBlock = createIpBlock(csDebugWrapper, hdlWrappedComponent, ipBlockDfD, ↩
   toolConfiguration);
19
20     createInstances(csDebugWrapper, csIpBlock, ios, toolConfiguration, logger);
21
22     debugWrapperDfD = new DFDDebugWrapper(toolConfiguration.getName());
23     update(csIpBlock, ipBlockDfD, csDebugWrapper, debugWrapperDfD, ↩
   toolConfiguration);
24
25     hdlEntity = csDebugWrapper.createEntity(toolConfiguration);
26     hdlComponent = new Component(hdlEntity);
27     hdlArchitecture = csDebugWrapper.createArchitecture(hdlEntity);
28     hdlConfiguration = csDebugWrapper.createConfiguration(hdlEntity, hdlArchitecture, ↩
   toolConfiguration);
29
30     writeHdlEntity(hdlEntity, toolConfiguration, logger);
31     writeHdlComponent(hdlComponent, toolConfiguration, logger);
32     writeHdlArchitecture(hdlArchitecture, hdlEntity, csDebugWrapper, toolConfiguration, ↩
    logger);
33     writeHdlConfiguration(hdlConfiguration, toolConfiguration, logger);
34
35     writeDfDConfiguration(debugWrapperDfD, toolConfiguration, logger);
36   }
37   ...
38 }
```

The DebugWrapperBuilder class is derived on line 1 from the implementation of the AbstractBuilder abstract class, described in Sect. 6.2.2. Note that all builders read their configuration file as part of their constructor function, so when the `build` function is called, the builder has already read this file.

As the first step in the `build` function, the builder reads the HDL component file that is specified in this configuration on line 5. The `selectComponent` function on line 7 selects one component declaration out of the possible set of component declarations present in the specified HDL file. This operation is implemented as a

separate function to allow another class to be derived from this class and provide a different implementation for the component selection. The current implementation takes the first component declaration found in the HDL file. A derived class may for example ask the user which component to use by showing a GUI with a list of available candidate components.

The generics used in the interface of the selected component are replaced on line 9 by their values, as specified in the debug wrapper configuration. The data structure *ios* subsequently collects the I/Os of the selected component on line 11. The `groupPorts` function on line 12 implements the identification of the communication ports on the module, using the information provided by the <port> element(s) in the configuration file. This operation is also implemented as a separate function to allow another class to be derived from this class and provide an implementation that identifies communication ports that use a different communication protocols. The current implementation groups all signals that belong to the same DTL port through regular expression matching. For example, the regular expression `(.*)cmd_valid(.*)`, given in the <signal> subelement of the <port> element on line 36 in Listing 6.2, is used to find all DTL command valid signals on the component. For each matching signal name, the prefix and suffix strings found using the "(.*)" expressions, are subsequently used to find the other signals that belong to the same DTL port. For one of the processor modules that is used in the case study in Chap. 8, this regular expression matches among others its `dtl_cmd_valid_tile1_m` signal name with a prefix of `dtl_` and a suffix of `_tile1_m`. The `groupPorts` function subsequently finds all DTL signals described in Table 3.2 with the same prefix and suffix. It subsequently groups and stores all these signals as a single DTL port in the `ios` data structure.

An optional DfD configuration file of the component is read and stored in a DfD configuration data structure on line 14. A new DfD configuration data structure is created when no DfD configuration file was specified. The name of this new DfD configuration is based on the selected component. On line 16, the builder creates a CSARComponent object in memory to represent the new debug wrapper. Similarly, it creates an object in memory on line 18 to represent the selected HDL component.

The builder uses the `createInstances` function on line 20 to fill in the implementation details for the debug wrapper. It first copies all I/Os of the module to the interface of this entity declaration, with the exception of the I/Os that need to be connected to the EDI. For the example processor module, this latter group consists of the "stalled" signal, as specified on line 51 in Listing 6.2. It adds one or two DTL monitors and a PSI to each DTL port stored in the data structure `ios`. The builder instantiates an EDI node to connect all monitors, all PSIs, and the optional EDI signals of the module to the EDI ports on the debug wrapper. It subsequently concatenates the TPRs in the EDI node, the monitors, and the PSIs in a single TPR chain, in series with any TPRs inside the module itself. It connects this TPR chain to the TPR interface on the debug wrapper. Similarly, it concatenates the memory test wrappers inside the DTL monitors in a memory test chain, in series with any memory test wrappers in the module itself. It then connects this memory test chain to the memory test interface on the debug wrapper.

The DfD configuration of the debug wrapper is created on line 22 and updated with the information on the PSIs, monitors, EDI node, TPR chain, and memory chain on line 23. HDL objects are created on lines 25 to 28 and written to files on lines 30 to 33. The optional values of the <outputDir> and <coreuse> elements in the configuration file are taken into account during these write operations. The resulting DfD configuration for the debug wrapper is written to a new DfD configuration file on line 35.

6.4.4 Executing the Debug Wrapper Generation Process

Listing 6.5 shows an example execution run of the debug wrapper generation tool. Note how the DTL ports on the processor are automatically identified on lines 8 and 9 and the PSIs and monitors are included in the implementation of the debug wrapper on lines 10 and 11. A seven-port EDI node is instantiated in the debug wrapper on line 12 with ports for the two monitors, the two PSIs, the two communication ports on the debug wrapper, and the event signal from the processor. The "stalled" signal of the processor is subsequently hooked up to this EDI node on line 13. The HDL files and the DfD configuration file of the debug wrapper are written on lines 14 to 18.

6.5 Other Tools in the DfD Flow

The other tools in the DfD flow use the same design principles and general architecture as the debug wrapper generation tool. Because of these similarities, we do not detail these tools here further. Interested readers can refer to Appendix A for a continued discussion of these tools.

Listing 6.5 Command line example of the debug wrapper generation for a processor tile

```
1  DebugWrapperGenerator v1.0
2
3  * Reading '/home/csar/book/vhdl/rdt_dlx_lib/rdt_dlx_tile/CSARDE/rdt_dlx_tile_producer.dwc'.
4  * Reading '../../../../noc/src/vhdl/data/rdt_dlx_lib/rdt_dlx_tile/INTERFACE/rdt_dlx_tile_cmp_pkg.p.vhdl'.
5  * Component 'rdt_dlx_tile' selected.
6  * Reading 'rdt_dlx_tile.dfd'.
7  * Generating debug wrapper structures.
8    - Found '< x> _m' DTL port.
9    - Found '< x> _s' DTL port.
10   - Adding PSI and core—side monitor to '< x> _m' DTL port.
11   - Adding PSI and wrapper—side monitor to '< x> _s' DTL port.
12   - Adding 7—port EDI node.
13   - Hooking up IP signal 'stalled' to the EDI.
14 * Writing '../../rdt_dlx_tile_producer_debug_wrapper/INTERFACE/rdt_dlx_tile_producer_debug_wrapper.e.vhdl'.
15 * Writing '../../rdt_dlx_tile_producer_debug_wrapper/INTERFACE/⟷
   rdt_dlx_tile_producer_debug_wrapper_cmp_pkg.p.vhdl'.
16 * Writing '../../rdt_dlx_tile_producer_debug_wrapper/RTL/rdt_dlx_tile_producer_debug_wrapper_rtl.a.vhdl'.
17 * Writing '../../rdt_dlx_tile_producer_debug_wrapper/RTL/rdt_dlx_tile_producer_debug_wrapper_rtl_cfg.c.vhdl'.
18 * Writing '../../rdt_dlx_tile_producer_debug_wrapper/CSARDE/rdt_dlx_tile_producer_debug_wrapper.dfd'.
```

6.6 Summary

In this chapter we presented an overview of our CSAR DfD flow. We presented details on how the CSAR hardware modules can be automatically instantiated in a custom SOC under user control. Functional hardware and software designers need not be aware that our DfD flow is used. Our flow takes care of the insertion of the required debug modules for them. We furthermore showed how information on these modules is stored in a single boundary-scan-level DfD configuration file. In Chap. 7, we describe the off-chip debugger software that interacts with the CSAR on-chip debug hardware. It uses the information in this DfD configuration file to facilitate debugging the SOC at multiple abstraction levels.

References

1. Cadence. *LEF/DEF Language Reference - Product Version 5.7*. Cadence Design Systems, July 2011.
2. Codehaus. Xstream library v1.4.2, November 2011.
3. 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*, pages 1182–1187, Washington, DC, USA, March 2005. IEEE Computer Society Press.
4. M. Habibi. *Java regular expressions: taming the java.util.regex engine*. Real World Series. Apress, 2004.
5. Andreas Hansson. *A* Composable and Predictable On-Chip Interconnect. PhD thesis, Eindhoven University of Technology, 2009.
6. Andreas Hansson and Kees Goossens. Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases. In *Proc. International Symposium on Networks on Chip*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society Press.
7. IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language-IEEE Std 1364-2005*. April 2006.
8. IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual-IEEE Std 1076-2008*. January 2009.
9. NXP Semiconductors. *CoReUse 5.0 Standards Book*, 2009.
10. T. J. Parr and R.W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
11. Christopher Pohl, Carlos Paiz, and Mario Porrmann. vmagic - automatic code generation for vhdl. *International Journal of Reconfigurable Computing*, page 9, March 2009.
12. Steve Purcell. Jargs command line option parsing suite for java, v1.0, April 2005.

Chapter 7
Off-Chip Debugger Software

Abstract In this chapter, we present our off-chip debugger software called the CSAR debug environment (CSARDE). The goal of the CSARDE is to provide an effective, efficient and extensible debug environment to debug an SOC using the CSAR debug approach. In Sect. 7.1, we give an overview of the requirements for the CSARDE, its software architecture, and the main design patterns used in its implementation. We subsequently present details on its four main software components: (1) the *SOC manager*, (2) the *abstraction manager*, (3) the *scripting engine*, and (4) the *user interfaces* in Sects. 7.2–7.5. We conclude this chapter with a summary in Sect. 7.6.

7.1 Overview

In this chapter, we present our off-chip debugger software called the CSAR debug environment (CSARDE). The goal of the CSARDE is to provide an effective, efficient and extensible debug environment to debug an SOC using the CSAR debug approach. The effectiveness and efficiency of the CSARDE is achieved through the implementation of the CSAR abstraction techniques described in Sect. 4.1.3. Its extensibility is achieved by making the CSARDE both modular and scriptable, allowing a debug engineer to customize the debug functionality to the debug requirements of a particular SOC and use case. A debug engineer can use the functionality of the CSARDE to control the execution of the SOC and to interpret its state at different abstraction levels and in different SOC environments. The CSARDE supports the debug process by enabling the effective and efficient comparison of the behavior of a silicon SOC implementation with the behavior of its reference(s) under user control. In the remainder of this section, we provide an overview of the requirements for the CSARDE, its software architecture, and the main design patterns used in its implementation.

7.1.1 CSARDE Requirements

The CSARDE addresses the following eight requirements:

1. *SOC environment abstraction* to access a debugable SOC in different SOC environments at the different phases in the implementation refinement process (refer to Fig. 1.4). Example environments are a simulation environment, an

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,* Embedded Systems, DOI 10.1007/978-3-319-06242-6_7,

FPGA-based prototyping environment, and the SOC's product environment. Details on this abstraction are provided in Sect. 7.2.2.
2. *Customization of the CSARDE to a specific SOC* using a boundary-scan-level DfD configuration file, as generated by the DfD flow described in Chap. 6. Details on this requirement are provided in Sect. 7.2.3.
3. *Centralized control over the SOC mode* by managing the operating and clock mode of the SOC and controlling the transitions between the available modes. Details on these modes are provided in Sect. 7.2.4.
4. *State access* using the on-chip debug architecture described in Chap. 5 that provides both full state access and module-level state access. More details on this requirement are provided in Sects. 7.2.5 and 7.3.5.
5. *Abstracted control over and interpretation of the SOC state* by applying *structural*, *data*, and *behavioral abstraction techniques* to this state. The implementation and application of these techniques are discussed in respectively Sects. 7.3.2, 7.3.3, and 7.3.4.
6. *Abstracted control over the communication inside the SOC* using the communication monitors, the PSIs, and the EDI by applying *temporal abstraction techniques*. Details on the implementation and application of these techniques are described in Sect. 7.3.5.
7. *Graphical* and *command-line user interfaces (UIs)* for the inspection of the SOC state, and the interactive or scripted execution of debug commands. The required scripting engine is described in Sect. 7.4 and the available UIs are described in Sect. 7.5. Source code examples throughout this chapter use the scripting language of this engine.
8. *Extensible with new, SOC-specific debug functionality*, e.g., to support other SOC environments and abstraction techniques. Details on how to extend the existing CSARDE functionality are provided when applicable.

7.1.2 Software Architecture

We chose to implement the CSARDE in Java, because Java applications run without modifications on the majority of the compute platforms in use today. The two other main reasons for choosing Java as the implementation language are the features of the Java language and the large number of existing Java libraries that the CSARDE can readily reused. For example, the CSARDE is built on top of the Eclipse 4 application platform, which offers support for dependency injection, OSGi services, and a modeled GUI [5]. These features enable the extensibility of the CSARDE.

The Eclipse platform supports the implementation of a modular application, with a loose coupling between its software components. This modular style simplifies the implementation of the CSARDE. Figure 7.1 shows a high-level overview of the software architecture of the CSARDE. This architecture consists of four main software components: (1) the *SOC manager*, (2) the *abstraction manager*, (3) the *scripting engine*, and (4) the *(graphical) user interfaces*. Before we describe these

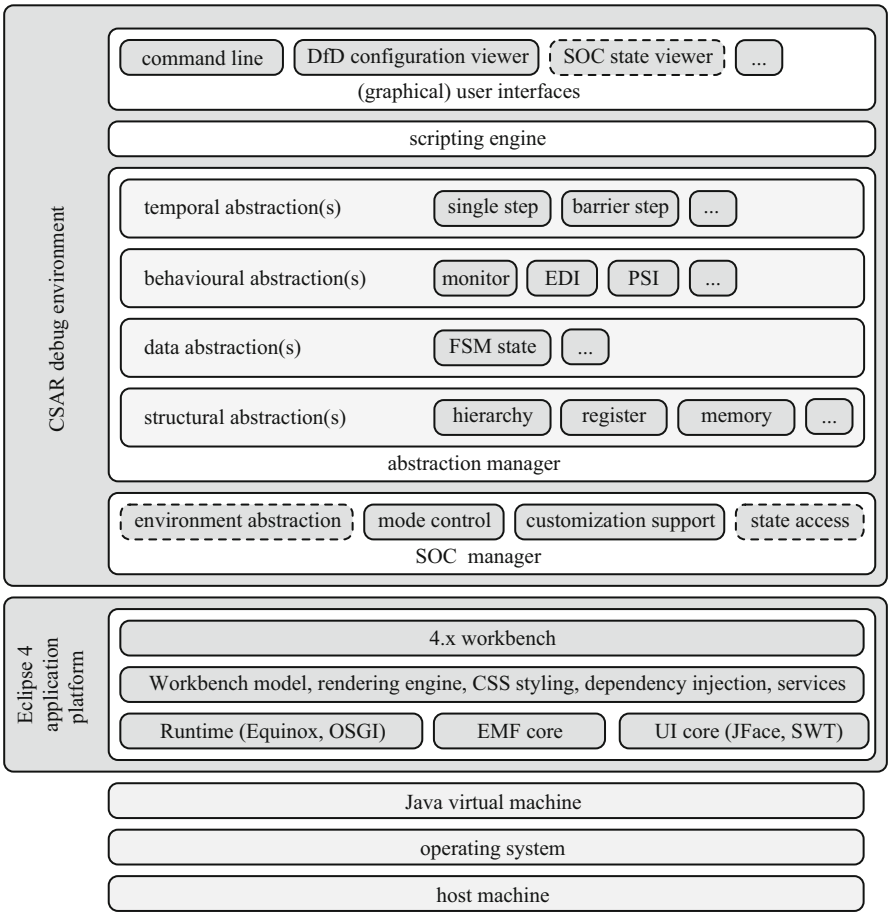


Fig. 7.1 CSARDE software architecture (dashed boxes are currently manual)

four components in more detail in subsequent sections, we first introduce the main design concepts that are used in the implementation of the CSARDE.

7.1.3 CSARDE Design Concepts

7.1.3.1 Interfaces

The CSARDE adopts the software design concept of "programming against an interface instead of against an implementation" [4]. This concept decouples a user[1]

[1] With the term *user* we refer to both the debug engineer as well as the software components of the CSARDE.

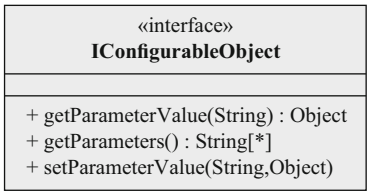


Fig. 7.2 The IConfigurableObject interface

of a function from a specific implementation of that function. This increases the modularity of the CSARDE, as it permits the substitution of one implementation of a function with another implementation of that same function without affecting the rest of the system. The application of this concept is visible in the source code of the CSARDE by the extensive use of *Java interfaces* to interact with most class objects. The names of all interfaces in the CSARDE start with the letter "I" to promote the use of this design concept. We use interfaces among others to abstract (1) from SOC environment objects using the IEnvironment interface, (2) from abstraction objects using the IAbstraction interface, and (3) from the SOC and abstraction managers using respectively the ISocManager and IAbstractionManager interfaces. We discuss these interfaces later in this chapter.

7.1.3.2 Configurability

A user can configure many of the class objects used in the CSARDE. The IConfigurableObject interface shown in Fig. 7.2 provides this configurability in a generic way. This interface uses the feature of the Java language that every class is derived from the Java Object class. This allows the value of each parameter of an IConfigurableObject object to be stored as a Java Object object using a unique Java String identifier. Each parameter can subsequently be retrieved from the object using the object's `getParameterValue` function and this unique identifier. The IConfigurableObject interface is used among others to configure the SOC environment(s) and abstraction objects.

7.1.3.3 Support for Multiple Threads

The CSARDE uses multiple execution threads to implement its functionality. A typical scenario uses one execution thread to execute the main (graphical) UI loop to interact with the user, while one or more background threads perform computational-intensive tasks or interact with an SOC environment. Partitioning the execution over multiple threads has the advantage of keeping the UIs responsive for the user. A disadvantage however is that these multiple execution threads may need to access the same resources, creating the possibility for a race condition. We use the IExecutable interface shown in Fig. 7.3 to be able to perform atomic operations on the CSARDE data structures.

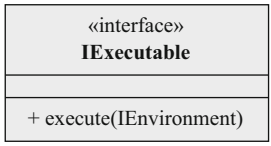


Fig. 7.3 The IExecutable interface

An IExecutable object can be passed as an argument to a synchronized Java method of an object to perform an atomic operation on the data stored in this object. We show a more concrete example of the usage of this interface in Sect. 7.2.

7.1.3.4 Re-Using Mature Software Design Patterns

The implementation of the CSARDE furthermore uses six mature software design patterns [1]. First, the *composite pattern* lets a user treat individual objects and compositions of objects in the same way. We use this pattern for example to manage the boundary-scan-level DfD configuration in Sect. 7.2.3, the module hierarchy in Sect. 7.3.2, and the state of an SOC in Sect. 7.3.3. Second, the *decorator pattern* dynamically attaches additional functionality to an object. We use decorators to augment the functionality of the base CSARDE objects. For example, the data and behavioral abstraction techniques decorate the base CSARDE objects with additional functions and state in Sects. 7.3.4 and 7.3.5. Third, the *dependency injection pattern* allows the selection of an object to be made at run-time rather than at compile time. This pattern is for example used by users to gain access to the SOC and abstraction managers (refer to Sects. 7.2 and 7.3). Fourth, the *factory pattern* defines an interface for creating an object, but lets the factory object that implements this interface decide which object to actually create. This pattern is for example used in Sect. 7.2.1 to allow users to create an SOC environment based on their debug requirements. Fifth, the *observer pattern* defines a one-to-many dependency between objects, so that when one object changes its state, all its dependents are automatically notified. This pattern is used for example in Sect. 7.5 to inform the interested graphical views about changes in the SOC state, so that these views can update their state representation. Sixth, the *service locator pattern* generalizes the process to obtain a particular OSGi service. This pattern is used for example to find the available environment factories in Sect. 7.2.1 and the available abstraction factories in Sect. 7.3.1.

7.2 The SOC Manager

7.2.1 Overview

The *SOC manager* is responsible for managing all interactions of a user with one or more SOC environments, as stated by debug infrastructure requirements IR-4, IR-9,

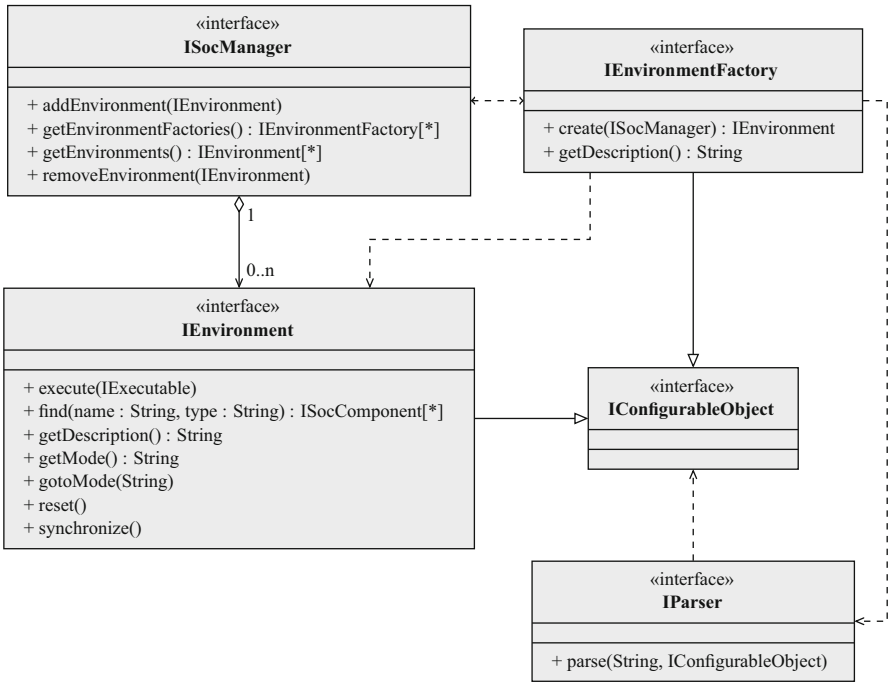


Fig. 7.4 The ISocManager interface and its dependencies

IR-14, IR-15, IR-16, IR-17, IR-18, IR-23 and IR-24. The SOC manager provides (1) SOC environment abstraction, (2) SOC customization support, (3) SOC mode control, and (4) SOC state access. Each of these features is described in more detail below.

7.2.2 SOC Environment Abstraction

The SOC environments that are used to debug the silicon implementation may have different intrinsic interfaces. For example, a simulation environment may allow a user to interact with an SOC simulation model using a TCP/IP[2] connection, while a physical SOC environment may only allow a user to interact with the silicon implementation of the SOC using a TAP connection. The CSARDE uses *SOC environment abstraction* to abstract from these different intrinsic interfaces and to provide instead a common debug interface. Figure 7.4 shows an overview of the interfaces involved in this abstraction technique.

[2] transmission control protocol over internet protocol

To implement this abstraction, the CSARDE requires that each type of SOC environment is supported by an associated Java class, which implements the IEnvironment interface. This interface effectively hides the specific access details for the corresponding SOC environment. The CSARDE currently provides a FileEnvironment class, which writes the TAP instructions and data that should be communicated with the SOC to a computer file. The actual application of these TAP instructions and data to the SOC is still a manual activity, as indicated by the dashed box in Fig. 7.1. As examples of other IEnvironment classes, the CSARDE could use a TcpIpEnvironment class to interact for example with an SOC in a simulation environment using a TCP/IP connection, or a UsbEnvironment class to interact with an SOC in a physical setup using a USB/TAP[3] adapter.

It is important that users only access and control the SOC in the corresponding environment using an object of the associated Java class, as its interface serializes all accesses and supports the execution of atomic operations on the SOC environment through the `execute` function and the IExecutable interface. These features prevent races conditions between different CSARDE software threads. Details on the `execute` function are given in Sects. 7.2.5 and 7.3.

The `find` function can be used to find modules in the SOC by name and type, and is explained in Sect. 7.3.2. The `getMode`, `gotoMode` and `reset` functions of the IEnvironment interface provide operating mode control and are discussed in Sect. 7.2.4. The `synchronize` function provides standardized access to the SOC state and is explained in Sect. 7.2.5.

A user uses the SOC manager to register an IEnvironment object with the CSARDE. A user requests a reference to the SOC manager using *dependency injection*. Instead of creating an IEnvironment object manually, a user first queries the SOC manager for a list of registered environment factories using its `getEnvironmentFactories` function. A user then chooses an appropriate environment factory from this list and configure it depending on the specific debug requirements. For example, the FileEnvironment class is supported by a FileEnvironmentFactory class. To customize an environment factory for a particular SOC, a user sets the value of their `url` parameter to the uniform resource locator (URL) of the boundary-scan-level DfD configuration file of the corresponding SOC, and the value of their `parser` parameter to an IParser object, which is able to parse the format of this file. Additionally, environment factories may have configuration parameters specific to each SOC environment.

The configured factory creates and returns an IEnvironment object when the user calls its `create` function. In this process, the factory parses the configured file using the `parse` function of the configured parser. This parser stores information from the configured file as the values of specific parameters of the newly-created IEnvironment object. The IEnvironment interface extends the IConfigurableObject interface to provide this functionality. For example, the CSARDE provides a default parser for the boundary-scan-level DfD configuration file generated in Chap. 6. This parser

[3] universal serial bus to test access port

stores the information in this file as the value of the `configuration` parameter of the IEnvironment object. We discuss this information in Sect. 7.2.3. The value of the `parser` parameter of the FileEnvironmentFactory class is set by default to a reference to this parser.

The factory subsequently registers the newly-created IEnvironment object with the SOC manager using this manager's `addEnvironment` function. The SOC manager maintains a list of all created IEnvironment objects, to allow the user to query them and to ensure that the system resources that are allocated by these environments can be properly deallocated when the CSARDE closes down.

A debug engineer can add support for a new SOC environment to the CSARDE using the following steps:

1. Implement the support for the SOC environment in a Java class, which implements the IEnvironment interface.
2. Implement an IEnvironmentFactory object in a Java class, which creates the new IEnvironment object with the optional help of a configured IParser object.
3. Register the environment factory as an OSGi service [5] with the CSARDE, so that the SOC manager detects it when it enumerates the available SOC environment factory services.
4. An IEnvironmentFactory object can support the creation of a configurable SOC environment. To enable this, create an IParser object and set the value of the `parser` parameter of the environment factory to a reference to this parser object. The URL of the file that contains the configuration parameters for this SOC environment needs to be configured as the value of the `url` parameter on the environment factory before calling its `create` function. The factory calls the `parse` function of the configured IParser object with this URL and a reference to the newly-created IEnvironment object.

7.2.3 SOC Customization Support

The environment factories in the CSARDE can use the configuration parser to customize the behavior of an IEnvironment object based on the information in a configuration file. Subsequent calls to functions of the IEnvironment object can use this information to change their behavior and adapt to the particular SOC and its environment. For example, the default parser translates the information in the boundary-scan-level DfD configuration file into an internal data structure. The root object of this structure is stored as the value of the `configuration` parameter of the IEnvironment object. Additionally, this default parser associates each flip-flop in this DfD configuration data structure with a FlipflopState object (refer to Fig. 7.5). These associated IState objects play a key role when we access the SOC state in Sect. 7.2.5. A FlipflopState object allows the IEnvironment object to associate a single character as the state of a flip-flop in the SOC. It furthermore implements

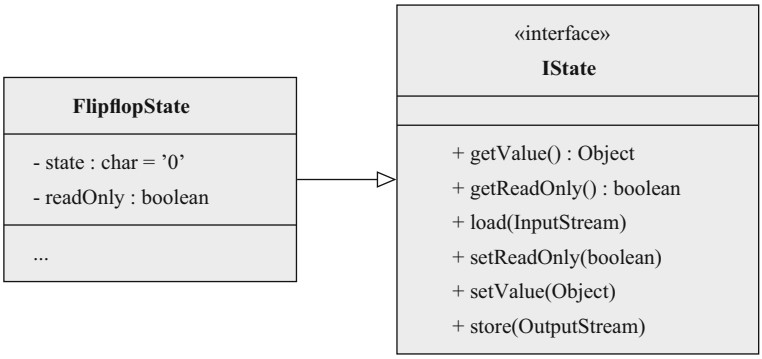


Fig. 7.5 The FlipflopState state and its implementation of the IState interface

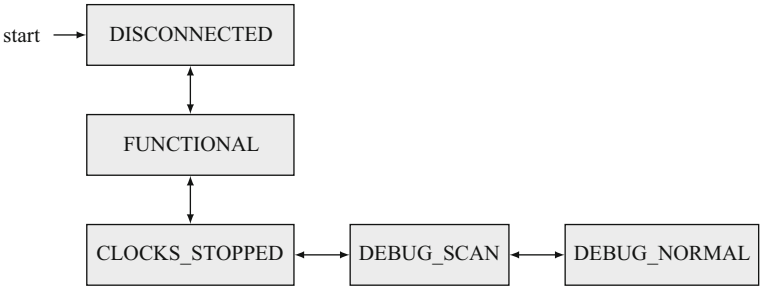


Fig. 7.6 SOC modes

the IState interface. A user can query the state of a flip-flop in the DfD configuration data structure using the `getValue` function of the FlipflopState object and any other IState object. A user can modify its state using the `setValue` function (unless it has previously been declared *read-only* with a call to the `setReadOnly` function). Reading a state from a Java InputStream object and writing a state to a Java OutputStream object are supported through the `load` and `store` functions.

7.2.4 SOC Mode Model

The global TCB and the CRGU TCB control the operating and clock mode of the SOC (refer to Sects. 5.2.3 and 5.8.2). The SOC manager controls these TCBs to simplify the mode selection for a user and to ensure that the configurations of both TCBs are always consistent. The SOC manager presents a simplified model with five SOC modes to its users with defined transitions between them, as shown in Fig. 7.6.

A user changes the SOC mode with the `gotoMode` function of the IEnvironment interface. A user can query the current mode of the SOC using the `getMode` function. Below we describe the five SOC modes and the available transitions between them.

The TAP instructions we use are taken from Table 5.1 and the modes of the global TCB and CRGU TCB are taken from respectively Sects. 5.2.3 and 5.8.2.

- DISCONNECTED: This is the initial mode of an SOC after its IEnvironment object has been created. This mode indicates that the SOC is not under control of the CSARDE. A user can start a debug experiment with a call to the `gotoMode` function with the argument `functional`. The IEnvironment object will try to establish a connection with the SOC inside its environment. The SOC mode changes to the FUNCTIONAL mode when a connection has been established. It stays in the DISCONNECTED mode when a connection cannot be established. For example, the FileEnvironment class implement the transition from the DISCONNECTED mode to the FUNCTIONAL mode by programming the TCBs inside the SOC using the DBG_PROGRAM_TCB TAP instruction. This class furthermore sets the "disable_tpr_reset" field in the global TCB during this operation to disable the functional reset of the TPRs.
- FUNCTIONAL: The TPRs are accessible in this mode using the `synchronize` function of the IEnvironment object (refer to Sect. 7.2.5). In the FUNCTIONAL mode, a user can reset the functional logic of the SOC by calling the `reset` function of the IEnvironment interface object. For example, the FileEnvironment class implements this functionality by activating the DBG_RESET TAP instruction and performing a data shift for one TCK cycle. The TPRs are not reset by this operation as this class set the "disable_tpr_reset" field in the global TCB in the transition from the DISCONNECTED to the FUNCTIONAL mode.
  A user can also control the communication inside the SOC by configuring the TPRs inside the on-chip communication monitors, the PSIs, and the EDI. The TPRs can also be queried to determine whether for example the SOC communication has stopped. A user can then transition to the CLOCKS_STOPPED mode. For example, the FileEnvironment class implements this transition by configuring the CRGU TCB to allow the functional clocks to be stopped and subsequently using the DBG_CLOCK_STOP TAP instruction to stop these clocks. A user can also stop the debug experiment by returning to the DISCONNECTED mode.
- CLOCKS_STOPPED: All functional clocks are stopped in this mode. A user can return from this mode to the FUNCTIONAL mode or go to the DEBUG_SCAN mode. For example, the FileEnvironment class implements the transition to the FUNCTIONAL mode by programming the CRGU TCB to disable clock stopping and subsequently using the DBG_RESET_CCS TAP instruction to reset all CCSes. It implements the transition to the DEBUG_SCAN mode by programming the global TCB with the "logic and memory debug shift" mode.
- DEBUG_SCAN: The logic and memory scan chains inside the test wrappers are accessible in this mode using the `synchronize` function of the IEnvironment object (refer to Sect. 7.2.5). A user can go from this mode to the DEBUG_SCAN mode, or return to the CLOCKS_STOPPED mode. For example, the FileEnvironment class implements these transitions by programming the global TCB with respectively the "logic and memory debug normal" node and the "functional" mode.

Table 7.1 Accessible state in each SOC mode

Mode	TPR state	Flip-flops	Wrapped memories
DISCONNECTED	–	–	–
FUNCTIONAL	Yes	–	–
CLOCKS_STOPPED	–	–	–
DEBUG_SCAN	Yes	Yes	–
DEBUG_NORMAL	–	–	Yes[a]

[a]Subject to the conditions set in the DEBUG_SCAN mode in the memory test wrappers

- DEBUG_NORMAL: The content of the memories inside the memory test wrappers can be accessed in this mode. When the user has previously placed a write or read operation in one or multiple memory test wrappers, these operations can be executed by using the `execute` function of the IEnvironment object. We discuss this functionality in the Sects. 7.2.5 and 7.3.5.

The current SOC mode is important for when we access the SOC state, as we explain next.

7.2.5 SOC State Access

Some SOC environments allow direct access to the SOC state, while other environments cannot due to intrinsic observability and controllability constraints. To be able to interact with both types of environments in the same way, the SOC manager maintains a data structure with a copy of the state of every accessible SOC flip-flop. The default DfD configuration parser parses the DfD configuration file, associates a FlipflopState object with every accessible flip-flop that is described in this file, and stores these objects in a data structure. This data structure is accessible as the value of the `configuration` parameter of the IEnvironment object. A user can find flip-flops in this data structure using the IEnvironment 's `find` function and subsequently modify their *associated state* directly using the functions of the IState interface. The `synchronize` function of the IEnvironment interface is responsible for exchanging the states in this data structure with the actual state of the SOC in its SOC environment.

With the CSAR hardware architecture, the access to the actual SOC state depends on the current SOC mode. The TPRs and the functional state are only accessible in some of these modes. Table 7.1 summarizes in which mode each (sub)state is accessible.

Note that the access to the TPRs in the DEBUG_SCAN mode takes place via the logic scan chains, while this access takes place via the chip-level TPR chain in the FUNCTIONAL mode (refer to Fig. 5.2).

Because the access to the SOC state is mode-dependent, an IEnvironment object has to implement its synchronization function differently in each mode, as described below. Note how the implementations for the FUNCTIONAL and DEBUG_SCAN modes

are the same, except for the search criterion for the DfD configuration data structure and the TAP instruction used.

- The implementation for the FUNCTIONAL mode should perform a depth-first search of the DfD configuration data structure, to find all flip-flops that are part of a control or a status register of a TPRs. It subsequently should use the character values of the FlipflopState objects associated with these flip-flops to update the state of the corresponding flip-flops in the SOC. The state of each flip-flop in the actual SOC can then be retrieved and stored in its corresponding FlipflopState object. This latter update can be prevented by setting the corresponding FlipflopState object to read-only.
  An IEnvironment class can implement this operation by (1) concatenating the state values of all associated FlipflopState objects found into a string of bits, (2) activating the DBG_PROGRAM_TPR TAP instruction in the SOC, (3) shifting the constructed string of bits into the SOC via the TAP, and (4) updating the associated FlipflopState objects using the string of bits that is returned via the TAP during this shift operation. This is currently still a manual step for the FileEnvironment class, as indicated by the dashed box in Fig. 7.1.
- The implementation for the DEBUG_SCAN mode should perform a depth-first search of the DfD configuration data structure, to find all flip-flops that are part of a logic scan chain, a memory scan chain, or a PIO unit in a test wrapper. It subsequently should use the character values of the FlipflopState objects associated with these flip-flops to update the state of the corresponding flip-flops in the SOC. The state of each flip-flop in the actual SOC can then be retrieved and stored in its corresponding FlipflopState. This latter update can be prevented by setting the corresponding FlipflopState object to read-only.
  For example, an IEnvironment class can implement these operations by (1) concatenating the state values of all associated FlipflopState objects found into a string of bits, (2) activating the DBG_SCAN TAP instruction in the SOC, (3) shifting the constructed string of bits into the SOC via the TAP, and (4) updating the associated FlipflopState objects using the string of bits that is returned via the TAP during this shift operation. This is currently still a manual step for the FileEnvironment class.
- The implementation performs no operation for the other modes.

7.3 The Abstraction Manager

7.3.1 Overview

The *abstraction manager* manages the abstraction techniques that a user may want to apply to the SOC state or to the control of its execution, as required by debug infrastructure requirements IR-19, IR-20, IR-21, and IR-22, as identified in Chap. 4.

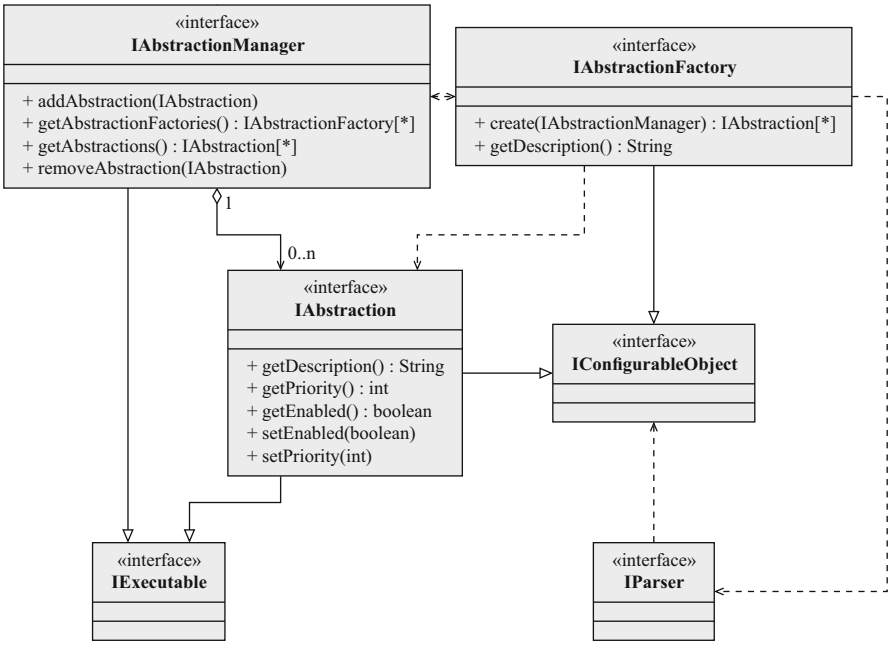


Fig. 7.7 The IAbstractionManager interface and its dependencies

The abstraction techniques discussed in this section are implemented by modifying the parameter values of a particular IEnvironment object. For example, they can be applied to the DfD configuration information that is stored as the value of the `configuration` parameter of FileEnvironment class objects, or a data structure derived from this information. Figure 7.7 shows an overview of the interfaces used by the abstraction manager. Note the many similarities with the interfaces used by the SOC manager in Fig. 7.4 on p. 169. Because of these similarities, we do not detail the creation and registration process of an IAbstraction object further. Instead we focus on the IAbstraction interface and classes that implement this interface.

Each abstraction technique is encapsulated in an IAbstraction class. This interface extends the IExecutable interface and as such, can be passed as an argument to the `execute` function of an IEnvironment object to atomically apply a specific abstraction technique to the parameters of this IEnvironment object. In the remainder of this section this process becomes clearer, when we provide an overview of the structural, data, behavioral, and temporal abstraction techniques that all use this extension mechanism.

Next to the `execute` function, the IAbstraction interface also provides the `getDescription` function to return a textual description of the abstraction class. This function may help identify a particular IAbstraction object in a list of such objects. Abstraction techniques can be enabled and disabled using the `setEnabled`

function, and given a numerical priority using the `setPriority` function. The enabled status and priority of an IAbstraction object can be queried using respectively the `getEnabled` and `getPriority` function.

The enabled status and priority of each IAbstraction object are important for when they are applied in a batch to a particular IEnvironment object. The IAbstractionManager interface also extends the IExecutable interface to enable this batch processing. A user can pass the abstraction manager as an argument to the `execute` function of an IEnvironment object. The effect of this is that all registered and enabled IAbstraction objects are atomically applied to the IEnvironment object, in sequence from lowest to highest priority. The advantage of this approach is that we can apply all registered abstraction techniques with one function call to an IEnvironment object. A possible disadvantage is that other software execution threads may experience a longer latency in accessing the same IEnvironment object, as these threads block on an access to this object for as long as the `execute` function applies the abstraction techniques registered with the abstraction manager. This should however not be a problem given that these abstraction technique need only be applied once, immediately after the creation of the IEnvironment object. We show an example of this process for our case study in Sect. 8.5.

A debug engineer can add SOC-specific abstraction techniques to the CSARDE. This process uses similar steps as described to add support for a new SOC environment in Sect. 7.2.2, and is therefore not detailed further. The abstraction techniques described in the remainder of this section have all been implemented and added to the CSARDE using this extension mechanism.

7.3.2 Structural Abstraction

7.3.2.1 Overview

Structural abstraction allows the SOC state to be presented to a user as a hierarchy of hardware modules, thereby abstracting away from the specific method used to access the SOC state. In the CSAR debug approach, we abstract away from the state access via the scan chains and the TAP. Instead, a user can refer to flip-flops, registers, and modules in a hierarchical module data structure using their names and relative positions in the pre-silicon module hierarchy. To achieve this goal, our structural abstraction technique comprises three steps: (1) hierarchy reconstruction, (2) register reconstruction, and (3) memory reconstruction. These techniques are SOC-independent and only require a list of hierarchical flip-flop names as input. This list is output by the synthesis tool and included in the boundary-scan-level DfD configuration file by the flow described in Chap. 6. Our structural abstraction techniques can therefore be applied to any SOC implementation that utilizes the CSAR debug architecture and DfD flow.

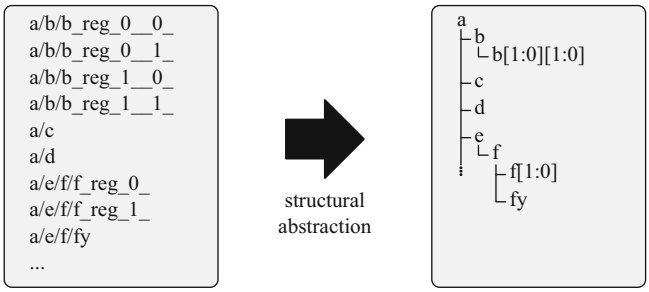


Fig. 7.8 Example usage of structural abstraction in the CSARDE

7.3.2.2 Hierarchy Reconstruction

Our first structural abstraction technique is called *hierarchy reconstruction*. This step relies on the synthesis tool to provide a list of all scan-accessible flip-flops, together with the order in which they appear in the scan chains. The synthesis tool inserts markers at the hierarchical boundaries of the original modules when it outputs the complete, hierarchical names of each flip-flop as part of this list. Our hierarchy reconstruction technique uses these hierarchy markers to reconstruct the functional module hierarchy. Figure 7.8 visualizes how this hierarchy (right-hand side) is reconstructed from an input list of flip-flops names (left-hand side). In this example, the synthesis tool inserted the character "/" as a marker between the names of a parent and a child in the original module hierarchy to construct the complete, hierarchical names of the flip-flops in the synthesized netlist. We reconstruct the original module hierarchy by reversing this process. The character used as a marker can be configured using the `separator` parameter of the corresponding abstraction factory.

We use Flipflop and Module classes, shown in Fig. 7.9, to store the result of this reconstruction step. These classes implement the ISocComponent interface. The ISocComponent interface uses the `composite pattern` to allow a user to interact with a Flipflop object in the same way as with a Module object. A user that wishes to iterate over the resulting module hierarchy data structure therefore can do this by only using the functionality provided by the ISocComponent interface.

A Flipflop and a Module object both hold a reference to its parent ISocComponent object. A user can obtain this parent reference using the `getParent` function. The ISocComponent interface allows both Flipflop and Module objects to be queried for the references of their child ISocComponent objects. These parent and optional child references effectively allow the modeling of the complete module hierarchy of an SOC. The `getChildren` function is implemented differently for the Flipflop and Module classes. A Flipflop object is a leaf node in the module hierarchy data structure and does not have any child objects. It therefore always returns an empty list when it is queried for its children. A Module object has child modules and/or flip-flops. It maintains a list of these children, which can be queried using the `getChildren` function.

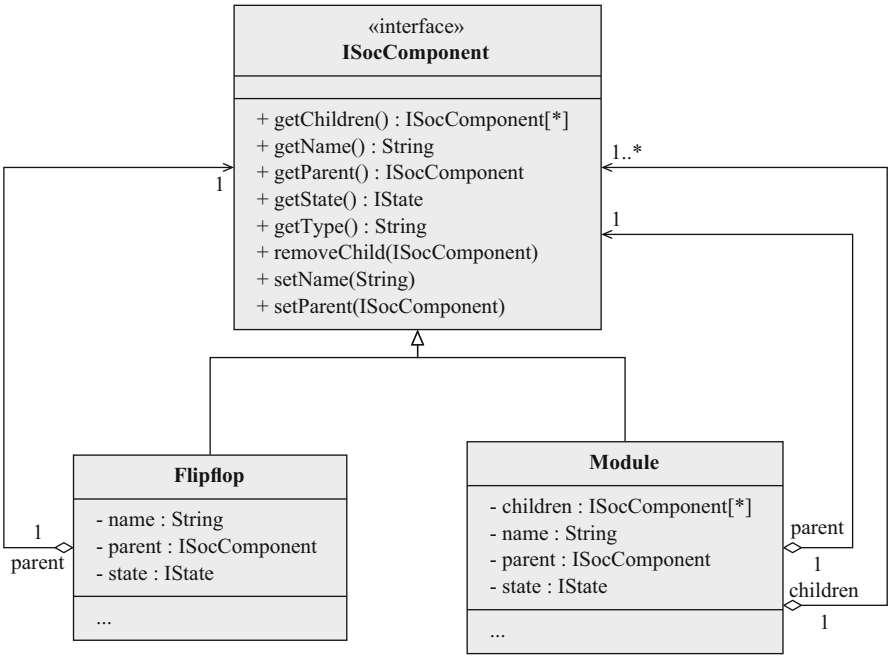


Fig. 7.9 The ISocComponent interface and the Flipflop and Module classes

The hierarchy reconstruction step represents all hardware modules in the SOC as Module objects in a hierarchical data structure, and makes each Flipflop object a child of its original, parent module. Our implementation performs these tasks by first splitting the complete, hierarchical name of each flip-flop, listed in the DfD configuration file, at the hierarchical markers inserted by the synthesis tool. Subsequently we create a Module object to represent each parent module.

As an example, all names in the list shown on the left-hand side of Fig. 7.8 belong to flip-flops that shared a common parent module named "a" in the original module hierarchy. We therefore create a Module object with the name "a", remove this name and the marker from the names of all flip-flops, and make all flip-flops children of this newly-created Module object. This process is recursively applied until all flip-flops are placed in their original hierarchy of parent modules.

The ISocComponent object at the root of this tree is stored as the value of the `root` parameter of the SOC environment object. Other abstraction techniques can subsequently retrieve the value of this parameter and apply their abstraction to this module hierarchy data structure. The CSAR data, behavioral, and temporal abstraction techniques, described respectively in Sects. 7.3.3, 7.3.4, and 7.3.5, all operate on this hierarchical module data structure.

Note that this module hierarchy data structure is orthogonal to the DfD configuration data structure that is used to access the SOC state (refer to Sect. 7.2.5).

7.3.2.3 Register Reconstruction

Our second structural abstraction technique is called *register reconstruction*. During the synthesis process, the synthesis tool replaces each multi-bit register with a set of flip-flops, one for each register bit. We regroup these individual flip-flops into a single register to allow a user to operate on the original multi-bit data value as a single unit. We implement this step by grouping the flip-flops that are part of the same register in a Register object. The implementation of the Register class is nearly identical to the implementation of the Module class. The only difference is that each child of a Register object has a unique index. This index corresponds to the bit position of the flip-flop in the original register. The state of a Register object consists of the set of FlipflopState objects of its children and is maintained by a RegisterState object. This RegisterState class implements the IState interface.

We support customizable register reconstruction through the use of pattern matching. A user can configure the *registerPattern* parameter of the IAbstraction object that implements the structural abstraction technique, with for example the regular expression "`(\S+)_(\d+)_`"[4]. This regular expression can match the names of flip-flops that originate from the same register; the first part "`\S+`" matches the name of the original register and the second part "`\d+`" matches the bit index of the flip-flop in this register. For the example list shown in Fig. 7.8, this identifies the first two flip-flops as flip-flops that are related to the same register "`b_reg_0_`", with respectively bit indices 0 and 1. Consequently, we create a Register object with the name "`b_reg_0_`" and add this object as a new child to the parent ISocComponent object of these two flip-flops, i.e., the Module object with the full name "a/b". The two flip-flops are added as children to this Register object, at respectively positions 0 and 1, and are subsequently removed from their original parent.

7.3.2.4 Memory Reconstruction

Our third structural abstraction technique is called *memory reconstruction*. This technique is used for memories that are implemented during synthesis using a multi-dimensional array of flip-flops. We describe support for the memories that instead are wrapped in a memory test wrapper in Sect. 7.3.5.

We implement the memory reconstruction step as a step after the register reconstruction. This step finds the Register objects that originate from the same memory and group them using a Memory object. The state of a Memory object consists of the set of RegisterState objects of its children and is maintained by a MemoryState object. The MemoryState class implements the IState interface.

The same as for the register reconstruction step, we support customizable memory reconstruction through the use of pattern matching. A user can configure

[4] We use the Java regular expression library [2] to provide the regular expression matching functionality.

the `memoryPattern` parameter of the IAbstraction object that implements the structural abstraction with for example the regular expression "`(\S+)_(\d+)_`". Because the memory reconstruction is performed after the register reconstruction step, this pattern matches those registers that correspond to the words in the original memory. For the example list shown in Fig. 7.8, this identifies two registers called "`b_reg_0_[1:0]`" and "`b_reg_1_[1:0]`" as words in the same memory, called "`b_reg`". The regular expression matches the original memory name and the index of the register in this memory. A Memory object is created with this name and added to the parent of these two registers as a new child, i.e., to the Module object with the full name "a". The two registers are added to the list of children of the Memory object, at respectively positions 01, and are subsequently removed from their original parent.

7.3.3 Data Abstraction

The structural abstraction step regroups flip-flops in the hierarchical module data structure, and reconstructs registers and memories. The state of one such group of flip-flops is still represented as a list of (a list of) flip-flop states, in the ModuleState, RegisterState, and MemoryState objects. Data abstraction raises the abstraction level of the state of these groups of flip-flops, to help bridge the difference in abstraction levels between a silicon SOC implementation and its references (refer to Sect. 2.2.3).

We support data abstraction by providing a set of IAbstraction objects that implement our data abstraction techniques. Each object analyzes the hierarchical module data structure and replaces the IState object of specific ISocComponent objects with an alternative IState object. This alternative object carries out a translation from the data type at the higher level of abstraction to the data representation at the bit level when its `setValue` function is called. It performs the reverse translation when its value is retrieved using the `getValue` function, provided that the state is consistent (refer to Sect. 2.2.3). The return value is "`null`" when the state of the IState object is inconsistent. A user can check whether an IState object is consistent with the `isConsistent` function.

As an example of this data abstraction technique, consider the 3-bit state register of the 5-state FSM described in Sect. 2.2.3. After the structural abstraction techniques described in Sect. 7.3.2 have been applied, this state register is represented in the hierarchical module data structure by a Register object with an associated RegisterState object as its state. A data abstraction technique object can replace this RegisterState object with an alternate FsmState object, which also implements the IState interface (refer to Fig. 7.10). Instead of returning a list containing three FlipflopState objects when its `getValue` function is called, this new class instead returns a Java String

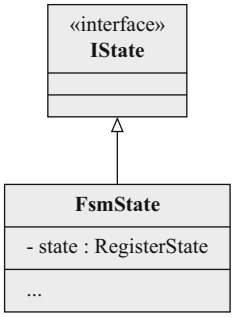


Fig. 7.10 An FsmState object and its dependency on the IState interface

object with the abstracted value "S1", "S2", "S3", "S4", "S5", or "xx"[5]. This data abstraction technique enables the interaction with this register as for example shown in the code fragment in Listing 7.1. In this code fragment, we first find the Register object on line 2 using the `find` function of the IEnvironment object, its full name (stored in the variable "$FSM_REGISTER"), and its type ("register"). We query on line 2 the abstracted data value of the register using the `getValue` function of the Register object. In our example, this returns the abstract value "S1". We subsequently program this register with the abstract value "S3" on line 4, query this value on line 5, and check whether its data value is consistent on line 7.

An added advantage of using data abstraction is that a user is not able to set an inconsistent value using this abstracted data interface. This is demonstrated on

Listing 7.1 Example of data abstraction

```
1  % set reg [ $env find $ {FSM_REGISTER} register ]
2  % puts [ $reg getValue ]
3  S1
4  % $reg setValue S3
5  % puts [ $reg getValue ]
6  S3
7  % puts [ $reg isConsistent ]
8  true
9  % $reg setValue HELLO
10 ! ERROR − invalid state value "HELLO"
11 % $env synchronize
12 % puts [ $reg isConsistent ]
13 false
14 % puts [ $reg getValue ]
15 xx
```

lines 9 and 10. The SOC may still contain an inconsistent (sub)state due to, e.g., state sampling artifacts (refer to complicating factor CF-9). These inconsistent (sub)states can show up in the register state after applying the `synchronize` function of the

[5] Note that with this technique it is trivial to return even more descriptive state names, for example the original FSM VHDL or Verilog state names.

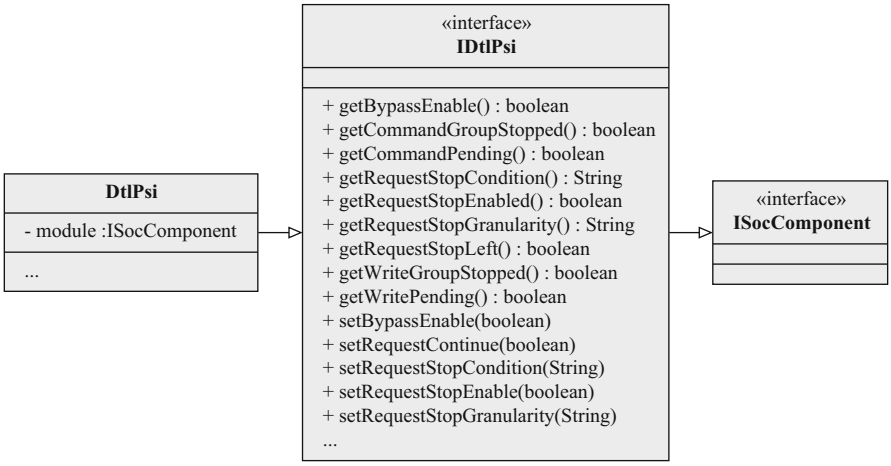


Fig. 7.11 The DtlPsi class, and the IDtlPsi and ISocComponent interfaces

SOC environment (line 11). The return value from this register is "xx" when the states of the three flip-flops combined are inconsistent (lines 12–15).

7.3.4 Behavioral Abstraction

Behavioral abstraction includes the function of a module in the interpretation and modification of its state. We support behavioral abstraction by providing a set of IAbstraction objects that implement our behavioral abstraction techniques. Each IAbstraction object analyzes the hierarchical module data structure and locates the Module objects that correspond to a particular type of hardware module in the SOC. This localization process uses knowledge of the hardware implementation of these modules. This knowledge is coded in an IAbstraction object and/or stored in a configuration file. Once the Module object corresponding to a specific type of hardware module is found, this IAbstraction object subsequently replaces the Module object in the hierarchical module data structure with a more specialized ISocComponent object. In addition to implementing the ISocComponent interface, this object provides functions that allow a user to more easily interact with the corresponding hardware module through a higher-level interface.

Consider as an example of this behavioral abstraction technique the DTL PSI hardware module described in Sect. 5.4. We provide the IDtlPsi interface in Fig. 7.11 to a user to make the interaction with this hardware module easier.

These operations correspond to the functionality described in Table 5.9. Note that we only show the functions for the request channel of a DTL PSI for brevity. Similar operations are available for the response channel. An associated DtlPsiAbstraction object analyzes the hierarchical module data structure of the IEnvironment object,

and replaces all Module objects that correspond to an on-chip DTL PSI with a DtlPsi object. As shown in Fig. 7.11, the DtlPsi class implements the ISocComponent interface through the IDtlPsi interface, and can therefore replace the original Module object in this data structure. In addition, the DtlPsi object provides additional functions through its IDtlPsi interface to facilitate the user interaction with on-chip DTL PSI hardware modules.

The code fragment in Listing 7.2 illustrates how we can easily interact with a DTL PSI inside the SOC using this IDtlPsi interface. We first find on line 1 all IDtlPsi objects in the hierarchical module data structure, using the `find` function of the SOC environment object and the type specifier "`dtlPsi`" (refer to Fig. 7.4). On line 2, we select the first object from the list that is returned. This yields a reference to an IDtlPsi object. On lines 3–6, we subsequently use this reference to configure the associated DTL PSI to stop unconditionally at transaction level. We call the `synchronize` function of the IEnvironment object on line 7, to exchange this configuration with the SOC in its environment (refer to Sect. 7.2.5). On lines 8 and 9, we poll the DTL PSI until the communication on its DTL command group has stopped.

In addition to allowing the fields in the DTL PSI TPR to be configured and queried through a high-level interface, the IDtlPsi object can also manage the read-only status of the control flip-flops in the TPR. This is to ensure that a synchronization operation only updates the status bits in the TPR, instead of all TPR bits.

Other behavioral abstraction objects have been implemented to perform similar replacement operations on the hierarchical module data structure for the other CSAR hardware module, and to insert specialized Java objects for configuring and querying the DTL monitors and the EDI nodes. A user can however extend this functionality and support more on-chip hardware modules, by adding custom abstraction techniques using the procedure described in Sect. 7.3.1.

Listing 7.2 Example usage of behavioral abstraction in the CSARDE for a DTL PSI

```
1 % set psis [$env find * dtlPsi ]
2 % set psi [ lindex $psis 0 ]
3 % $psi setRequestStopEnable 1
4 % $psi setRequestStopGranularity transaction
5 % $psi setRequestStopCondition unconditional
6 % $psi setBypassEnable 1
7 % $env synchronize
8 % set stopped [ $psi getCommandGroupStopped ]
9 % while {$stopped == 0 } { $env synchronize; set stopped [$psi ↩
  getCommandGroupStopped ] }
```

7.3.5 *Temporal Abstraction*

Temporal abstraction provides control over the execution of the SOC at different abstraction levels. It leverages the high-level interfaces that the previous abstraction techniques provide. We use this execution control in particular for two operations: (1)

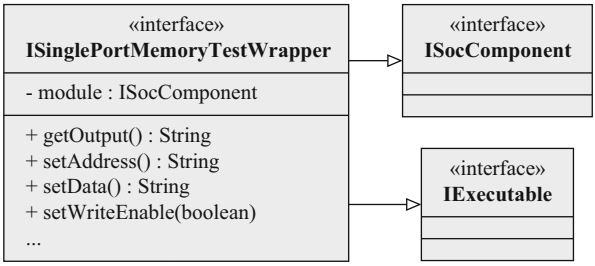


Fig. 7.12 The ISinglePortMemoryTestWrapper and ISocComponent interfaces

reading from and writing to the embedded memories via their memory test wrapper, and (2) stepping the on-chip communication at a higher abstraction level than clock cycles.

7.3.5.1 Read and Write Access on the Wrapped Embedded Memories

A memory test wrapper requires a two-step procedure to perform a write operation and a three-step procedure to perform a read operation on the wrapped memory. We have defined the ISinglePortMemoryTestWrapper interface that is shown in Fig. 7.12, to make the interaction with a memory test wrapper and its single-port memory inside the SOC easier for a user. A similar interface exists for accessing the memory test wrapper around a dual-port memory.

The ISinglePortMemoryTestWrapper interface extends the ISocComponent interface. Objects that implement this interface can therefore be inserted in the hierarchical module data structure by an IAbstraction object. This interface provides additional functions to configure a read or write operation on the wrapped memory and retrieve the result of a read operation. The interface furthermore extends the IExecutable interface to allow an ISinglePortMemoryTestWrapper object to be passed as an argument to the `execute` function of an IEnvironment object to perform an atomic write or read operation on the memory in the SOC.

The code fragment in Listing 7.3 shows an example of how the first 256 data words of an embedded memory can be read using an object with this interface. We first define the `readMemory` function on lines 1–6. We subsequently find all memory test wrappers around single-port memories on line 8 and select the first object from the returned list on line 9. This yields a reference to an ISinglePortMemoryTestWrapper object, which we subsequently pass in a loop, together with a reference to the IEnvironment object and the loop counter, to the `readMemory` function. We display the result of this function, together with the loop counter on line 11.

The `readMemory` function leverages the high-level ISinglePortMemoryTestWrapper interface to configure a read operation on lines 2 and 3. It then passes the ISinglePortMemoryTestWrapper object as an argument to the `execute` function of the IEnvironment object to start an atomic read operation on the SOC environment. Recall that all accesses to the `execute` function of an IEnvironment object are explicitly serialized, allowing only a single software thread to execute this function at every point in time.

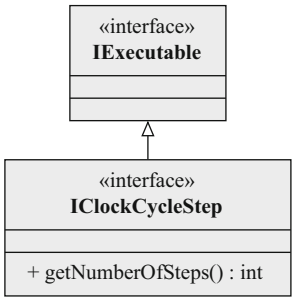


Fig. 7.13 The IExecutable and IClockCycleStep interfaces

The ISinglePortMemoryTestWrapper object performs the read operation when the IEnvironment object subsequently calls its `execute` function. An example implementation of the `execute` function of a Java class, which implements the ISinglePortMemoryTestWrapper interface is shown in Listing 7.4. It is important to note that the execution of the SOC can only be stepped at the clock-cycle level under control of the CSARDE in the DEBUG_NORMAL mode (refer to Sect. 7.2.4). An ISinglePortMemoryTestWrapper object therefore has to check the SOC mode, as shown on lines 2–5. The memory test wrapper inside the SOC can only be accessed and configured when the SOC is in the DEBUG_SCAN mode (line 6). The mode subsequently has to be changed to the DEBUG_NORMAL mode on line 7 to allow the execution to be stepped at the clock cycle level.

The ISinglePortMemoryTestWrapper object subsequently uses an IClockCycleStep object to control the execution of the SOC at the clock-cycle level (refer to Fig. 7.13). The IClockCycleStep interface extends the IExecutable interface to allow an IClockCycleStep object to be passed as an argument to the `execute` function of an IEnvironment object.

When an IEnvironment object receives an IClockCycleStep object as the argument of its `execute` function, it can query the number of clock cycles that the execution needs to be stepped using the `getNumberOfSteps` function and subsequently step the SOC in its environment for the configured number of clock cycles.

Listing 7.3 Example code fragment for the interaction with the test wrapper around a single-port memory

```
1  proc readMemory { env wrapper address } {
2    $wrapper setAddress $address
3    $wrapper setWriteEnable 0
4    $env execute $wrapper
5    return [ $wrapper getOutput ]
6  }
7
8  set memoryTestWrappers [ $env find * singlePortMemoryTestWrapper ]
9  set memoryTestWrapper [ lindex $memoryTestWrappers 0 ]
10 for {set i 0} {$i < 256} {incr i} {
11   puts "$i : [ readMemory $env $memoryTestWrapper $i ]"
12 }
```

By configuring an IClockCycleStep object with the required number of clock cycles and passing it as the argument to this `execute` function, the SOC environment can step the SOC in its environment for the configured number of clock cycles. The ISinglePortMemoryTestWrapper object uses an IClockCycleStep object to step the execution of the SOC at the clock-cycle level with two clock cycles for a read operation and with one clock cycle for a write operation (lines 9–14). The SOC mode is changed back to the DEBUG_SCAN mode after the execution of the SOC has been stepped one or two clock cycles (line 15). When the ISinglePortMemoryTestWrapper object is configured to perform a read operation, an additional synchronization step is executed to retrieve the data captured in the memory test wrapper from the SOC (lines 16–18).

Please note that for brevity the code in Listing 7.4 does not show the protection of the content of unrelated flip-flops, by saving their read-only flag, setting this flag before a synchronization operation, and restoring this flags afterwards. This can and probably should be included in practical cases.

Listing 7.4 Example implementation of the `execute` function of the ISinglePortMemoryTestWrapper interface

```
1  public void execute(IEnvironment env) {
2    if (! env.getMode().equals("debug_scan")) {
3      System.err.println("! ERROR − The SOC environment is not in the 'debug_scan'↩
   mode");
4      return;
5    }
6    env.synchronize();
7    env.gotoMode("debug_normal");
8    IClockCycleStep step;
9    if (getWriteEnable() == false) {
10     step = new ClockCycleStep(2);
11   } else {
12     step = new ClockCycleStep(1);
13   }
14   env.execute(step);
15   env.gotoMode("debug_scan");
16   if (getWriteEnable() == false)) {
17     env.synchronize();
18   }
19 }
```

7.3.5.2 Stepping the On-Chip Communication

In the CSAR debug approach, we use the communication monitors and PSIs to stop the execution of the SOC at a particular point in the internal communication. After an inspection of the SOC state, we resume or step through the on-chip communication at different granularity levels of the communication. We can use each individual PSI to control the communication on a communication link at the granularity of data elements, messages, and transactions. For a distributed application, we may need to

control multiple PSIs in parallel to force the SOC to take a particular path through it feasible state space to activate a dormant error and thereby causes a failure.

We can control multiple PSIs in parallel from the CSARDE. Consider as an example the code fragment in Listing 7.5. Note that this script assumes an interactive link with an SOC, which a future TcpIpEnvironment or UsbEnvironment can provide. We search the hierarchical module data structure on line 2 for all modules of type "dtlPsi". We subsequently enable each PSI that was found, and configure it to stop unconditionally at transaction level (lines 3–9). We synchronize the content of our hierarchical module data structure with the actual SOC implementation on line 10. By executing this code fragment, all communication in the SOC is stopped at the transaction level by the on-chip DTL PSIs.

We can then determine from the CSARDE whether all communication on the request channels has indeed been stopped by checking whether all DTL PSI show that their command and write groups have been stopped (lines 11–19). Once all communication has stopped, the on-chip clocks can be stopped without producing a globally-inconsistent state (line 21). We can afterwards extract the state of the flip-flops in the SOC state by entering the DEBUG_SCAN mode and synchronizing the content of the flip-flops (respectively line 22 and 23). Note that it is also possible at this stage to use the on-chip memory test wrappers to extract the state of the memories they wrap, using the procedure described earlier in this chapter. We will however not show this here for brevity.

Once the SOC state has been extracted, we can restore this state in the SOC (line 25) and return the SOC to its FUNCTIONAL mode (lines 27 and 28). Because we restored the state of the SOC, in which its DTL PSIs are stopping all communication, we effectively returned the SOC to the state it was in before we entered the CLOCKS_STOPPED mode on line 21.

As such, we can now use the continue functionality of each DTL PSI to step the communication on all links with one transaction (lines 29 and 38) and check whether all DTL PSIs have left their stop state (lines 38–45). Note that we also use the opportunity to clear the continue request for each DTL PSI at the same time (line 41). At this point, we can continue controlling the execution of the SOC, for example by returning to line 11 and check whether all DTL PSIs have stopped the communication on their respective communication links again. We however do not do this here. We describe instead a more elaborate debug scenario to control the communication in Sect. 8.4.2, in which we use the communication control illustrated in this section to replay a very specific ordering of transactions across the entire SOC to repeatedly reproduce the exact same communication order.

Listing 7.5 Example usage of the DTL PSIs for on-chip communication control

```
1  ...
2  # $env has previously been set to a reference of an IEnvironment object
3  set psis [ $env find * dtlPsi ]
4  foreach psi $psis {
5    $psi setRequestStopEnable 1
6    $psi setRequestStopCondition unconditional
7    $psi setRequestStopGranularity transaction
8    $psi setBypassEnable 1
9  }
10 $env synchronize
11 # Wait for all PSIs to stop their communication
12 foreach psi $psis {
13   while { [ $psi getWriteGroupStopped ] == 0 } {
14     $env synchronize
15   }
16   while { [ $psi getCommandGroupStopped ] == 0 } {
17     $env synchronize
18   }
19 }
20 # Extract SOC state
21 $env gotoMode clocks_stopped
22 $env gotoMode debug_scan
23 $env synchronize
24 # Restore SOC state
25 $env synchronize
26 # Return to functional mode
27 $env gotoMode clocks_stopped
28 $env gotoMode functional
29 # Issue continue request to all PSIs
30 foreach psi $psis {
31   $psi setBypassEnable 0
32 }
33 $env synchronize
34 foreach psi $psi {
35   $psi setRequestContinue 1
36   $psi setBypassEnable 1
37 }
38 $env synchronize
39 # Wait for all communication to resume
40 foreach psi $psis {
41   $psi setRequestContinue 0
42   while { [ $psi getRequestStopLeft ] == 0 } {
43     $env synchronize
44   }
45 }
46 ...
```

Note however that we could have easily restricted ourselves on lines 29–38 to issuing the continue request to a subset of the DTL PSIs instead of all of them, thereby only allowing the communication to continue on a subset of the communication links. In addition, we could have used *barrier stepping*, by using the

`requestBarrierStep` function shown in Listing 7.6 instead of the loop on lines 39–45 of Listing 7.5. A similar function can be defined for the response channel. In this `requestBarrierStep` function, we issue continue requests to all objects on the "`psiList`" list on lines 2–9. Note that this list can be a subset from the "`psis`" list in Listing 7.5. We subsequently initialize the counter "`left`" and enter a loop to count the number of DTL PSIs that have actually left their stopped state. The loop and function ends when this number reaches the minimum number of DTL PSIs "`min`" that is passed as an argument to this function. Note that the `requestBarrierStep` function waits for all DTL PSIs to leave their stop state, when the argument "`min`" is equal to the length of the list "`psiList`".

7.4 The Scripting Engine

The CSARDE includes a scripting engine to enable a user to easily interact with the SOC and abstraction managers, the boundary-scan-level DfD configuration information, the abstracted SOC module hierarchy, and other CSARDE objects using

Listing 7.6 Barrier step function to step the request channel of an arbitrary subset of communication links of a certain size

```
1  proc requestBarrierStep { env psis min } {
2      foreach psi $psis {
3          $psi setBypassEnable 0
4      }
5      $env synchronize
6      foreach psi $psis {
7          $psi setBypassEnable 1
8          $psi setRequestContinue 1
9      }
10     $env synchronize
11     set left 0
12     while { $min > $left } {
13         set left 0
14         foreach psi $psis {
15             $psi setRequestContinue 0
16             if { [ $psi getRequestStopLeft ] == 1 } {
17                 incr left
18             }
19         }
20     }
21 }
```

a command line interface. All code fragments in this chapter, with the exception of Listing 7.4, use the language of this scripting engine. This engine also allows the user to execute command scripts, thereby automating debug experiments, as required by debug infrastructure requirement IR-25, and supporting for example the debug flow shown in Fig. 4.5 on p. 78.

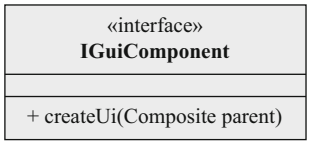


Fig. 7.14 The IGuiComponent interface

The language used by the scripting engine in the CSARDE is based on the (Tcl) [3]. We have chosen Tcl because it is used in most EDA tools and therefore familiar to hardware designers. We use the built-in support of Java to allow users to inspect CSARDE objects at run-time, and to call their methods. We currently have scripts that are executed by the scripting engine, but as the environment abstraction is still work in progress, these scripts do not affect the SOC directly yet. The CSARDE should be extended with TcpIpEnvironment and UsbEnvironment classes to provide this functionality, next to the existing FileEnvironment class.

7.5 The User Interfaces

The CSARDE UIs provide the debug engineer with a graphical and a textual interface to control the CSAR observability and controllability features of the SOC in different environment(s). Each ISocComponent object may implement the IGui-Component interface for this purpose (refer to Fig. 7.14). This interface provides the `createUi` function, which is used by the CSARDE to create a custom *view* for the corresponding object to inspect and/or edits its state. This view may implement structural, data, behavioral, and/or temporal abstraction to make it easier for a user to interact with the corresponding object. This satisfies the debug infrastructure requirements IR-19, IR-20, IR-21, and IR-22 from the UI point of view. We provide implementations of the `createUi` function for all CSAR debug modules described in Chap. 5. A user can extend the CSARDE by providing their own implementations.

Figure 7.15 shows an example GUI of the CSARDE, with a number of *views* on the states of different (types of) modules in an SOC. Note that this GUI is still a work in progress, as indicated by the dashed box in Fig. 7.1.

The CSARDE shows the registered SOC environments in its top-left view (indicated with ①). A user has to select an SOC environment in this view to be able to interact with it. Currently SOC environment can be created using the scripting engine. In future, SOC environments could for example also be created by pressing a button in the main toolbar (indicated with ⑨). This can then start a *wizard* process, which guides the user through this creation process.

The other views in the CSAR GUI can provide various, abstracted representations of (parts of) the SOC state and means to edit this state. The synchronize button in this main toolbar can be used to synchronize the SOC state in the CSARDE with the state of the SOC in its environment, as it calls the `synchronize` method on the currently-selected SOC environment.

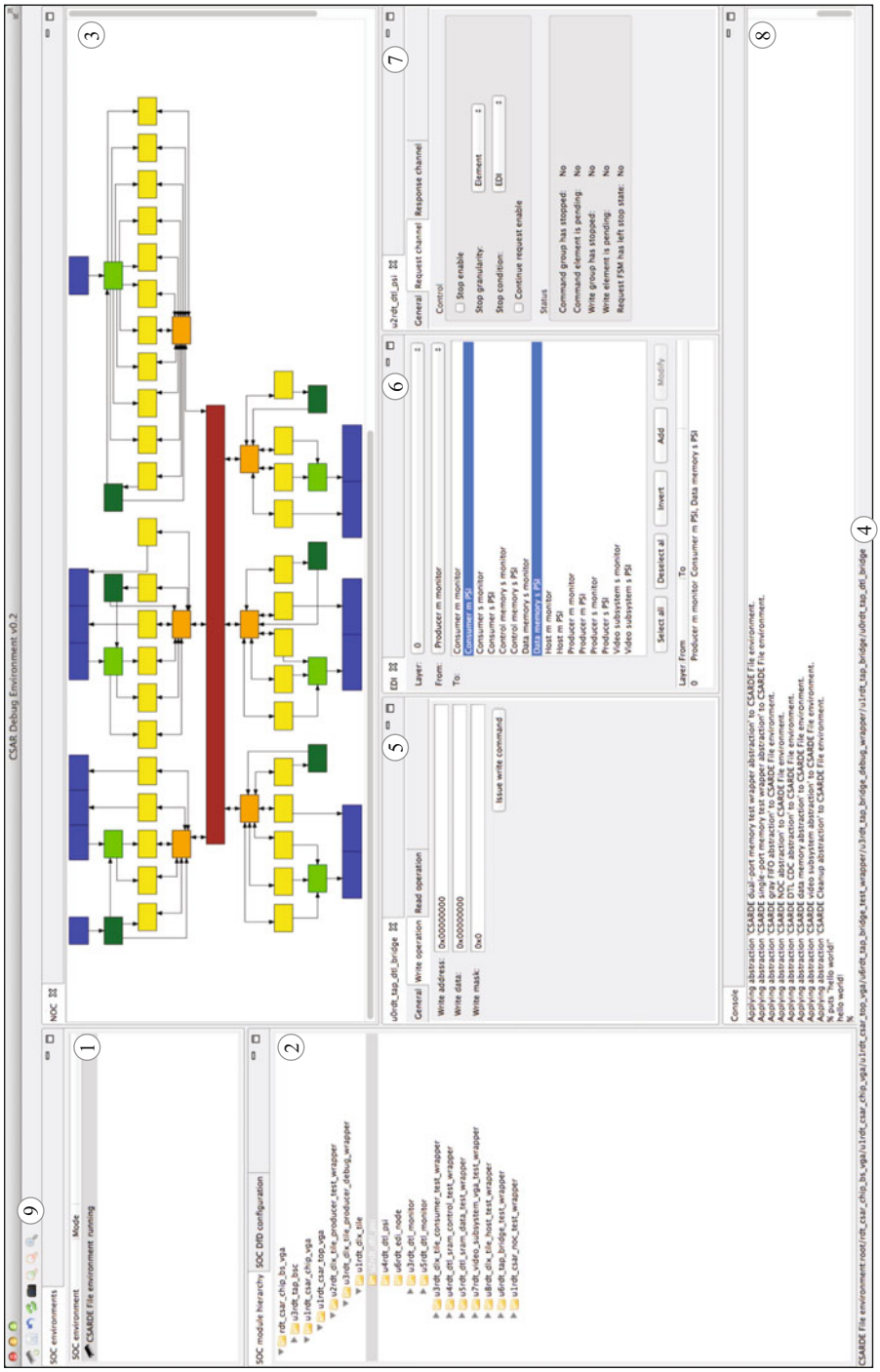


Fig. 7.15 CSARDE user interfaces

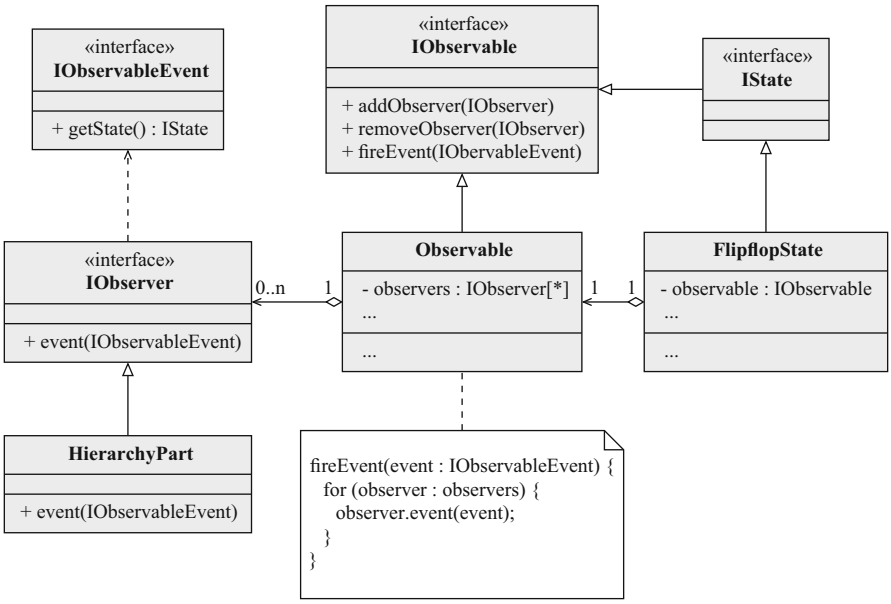


Fig. 7.16 Example usage of the observer pattern in the CSARDE

The hierarchical module data structure and DfD configuration data structure of the selected SOC environment are accessible in the view indicated with ②. A user can double-click on a module in this tree view to open a dedicated view, which permits the easier interpretation and modification of the corresponding module. These views use the `createUi` function of the IGuiComponent interface to create this dedicated view. The CSARDE provides a default view for Module objects.

An example of a view of the topology of the NoC and its building blocks is indicated with ③. In future, the view could allow a user to for example hover with the mouse pointer over each block to see the full hierarchical name of that block in the status bar at the bottom left of the GUI (indicated with ④). A user could then perhaps also double-click on one of these building blocks to go directly to the dedicated view that permits the interpretation and modification of the state of the corresponding module.

An example GUI for the TAP bridge module is indicated with ⑤. A user can fill in text fields to specify a write address, write data, and a write mask to execute a write operation, or specify a read address to execute a read operation. An example GUI to program the EDI is indicated with ⑥. A user can specify the EDI layer, the source of an EDI event, and the destinations of that EDI event. The CSARDE should then take care of configuring the EDI to create these paths in the EDI. This GUI could generate an error when the requested path cannot be realized due to restrictions in the SOC hardware. The GUI for a DTL PSI block is indicated with ⑦. A user can change the configuration of a PSI and use the synchronize method of the

currently-selected SOC environment to change the configuration of the SOC in the corresponding environment.

The command line interface is indicated with ⑧. The debug engineer can use Tcl commands and execute scripts from this interface using the scripting engine.

We use the *observer pattern*, shown in Fig. 7.16, to allow the views in the GUI to be informed of changes in the SOC state.

Each IState object implements the IObservable interface to support this pattern. Views can register themselves with an IObservable object using the `addObserver` function to be notified when the state of the corresponding object is changed. The view has to implement the IObserver interface. An IState object calls the `event` function on every registered observer when its state is changed, e.g., when a user modifies it from the command line interface or from one of the graphical views. Each observer can subsequently update its representation of this state.

7.6 Summary

In this chapter we provided an overview of the (CSARDE), its architecture, and its four key components: (1) the SOC manager, (2) the abstraction manager, (3) the scripting engine, and (4) the UIs. We showed how this user-friendly and extensible software meets the debug infrastructure requirements, identified in Chap. 4. We use the CSARDE in our case study of the CSAR SOC in Chap. 8.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
2. M. Habibi. *Java regular expressions: taming the java.util.regex engine*. Real World Series. Apress, 2004.
3. J.K. Ousterhout and K. Jones. *Tcl and the Tk Toolkit*. Pearson Education, 2009.
4. Bill Venners. Design principles from design patterns, 2005.
5. Lars Vogel. *Eclipse 4 Application Development: The complete guide to Eclipse 4 RCP development*. Lars Vogel, June 2012.

Part IV
Case Studies

Chapter 8
Case Studies

Abstract In this chapter we first evaluate the CSAR debug approach and infrastructure in Sect. 8.1. We subsequently use an illustrative GALS SOC model to evaluate specifically the communication-centric and abstraction-based aspects of the CSAR debug approach in more detail. We describe the application that runs on this SOC, its hardware architecture, and its internal clock domains in Sect. 8.2. We apply our DfD flow to the RTL implementation of this SOC in Sect. 8.3 and report on its effectiveness and efficiency. We also describe how we customize our CSARDE for use with this particular SOC. Even though this SOC is intentionally kept small for illustration purposes, we show in Sect. 8.4 that the factors identified in Chaps. 2 and 3 already complicate the debugging of an SOC this size. We also discuss in Sect. 8.4 how the CSAR debug approach and infrastructure reduces or eliminates the impact of each factor. In Sect. 8.5, we introduce examples of three different error types in the implementation of this SOC. These error types are (1) a *permanent, certain error*, (2) a *transient, certain error*, and (3) a *transient, uncertain error*. We subsequently describe how we can use the CSAR debug approach and infrastructure to localize these three example errors. We conclude this chapter with a summary in Sect. 8.6.

8.1 CSAR DfD in Industrial SOCs

We review in this section six industrial SOCs that contain parts of the CSAR debug functionality. We contributed to the inclusion of the debug functionality in the silicon implementation of these SOCs, through a close cooperation with the respective SOC design teams. We describe the lessons we learned from the implementation of this functionality in each SOC, the validation of this functionality on the SOC's silicon implementation, and the use of this functionality to debug errors in the first silicon implementation of these SOCs. The implementation and subsequent evaluation of this debug functionality served as proof-points and learning opportunities for the CSAR debug functionality described in this book. Table 8.1 lists the characteristics and the debug functionality of these SOCs, in relation to the CSAR debug approach.

Overall, the debug functionality mentioned in Table 8.1 has itself been successfully validated on all SOCs. In several cases, the included debug support was also successfully used post-silicon to diagnose unexpected failures and locate the root cause of erroneous behavior in both prototype silicon and in embedded software

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip*, Embedded Systems, DOI 10.1007/978-3-319-06242-6_8,

Table 8.1 Characteristics and debug functionality of six industrial SOCs and the CSAR SOC

	CPA	PNX8525	CODEC	Xetal-II	EN-II	CRISP-RFD	CSAR-SOC
Year	1999	2000	2002	2005	2007	2011	2013
Application Domain	Audio/Video	Audio/Video	Audio/Video	Video	General Purpose	Streaming DSP	Education
Process Technology	350 nm	180 nm	180 nm	90 nm	65 nm	90 nm	N/A
Clock Domains	30	92	32	9	32	1	9
Gate Count	$1{,}4 \cdot 10^6$	$8{,}0 \cdot 10^6$	$2{,}6 \cdot 10^6$	$1{,}8 \cdot 10^6$	$10 \cdot 10^6$	$7{,}4 \cdot 10^6$	$4{,}3 \cdot 10^6$
Flip-flop Count	$1{,}0 \cdot 10^5$	$1{,}9 \cdot 10^5$	$1{,}6 \cdot 10^5$	$5{,}0 \cdot 10^4$	$2{,}5 \cdot 10^5$	$2{,}7 \cdot 10^5$	$4{,}1 \cdot 10^5$
Die Size	169 mm^2	108 mm^2	102 mm^2	74 mm^2	45 mm^2	43.8 mm^2	N/A
TAP-based debug control	◇	●	●	●	●	○	●
Communication monitors	○	○	○	○	●	○	●
Event distribution	◇	◇	◇	○	◇	○	●
Execution control							
⋆ Chip-level clocks	◇	●	●	●	●	○	●
⋆ Module-level clocks	◇	○	○	○	○	●	●
⋆ Communication-centric	○	○	○	○	○	○	●
Debug state access							
⋆ Functional	◇	◇	◇	○	◇	●	●
⋆ Chip-level scan	●	●	●	●	●	○	●
⋆ Per module scan	○	○	○	○	○	●	●
Debug Area	4 %	0.3 %	0.3 %	0.1 %	2 %	¡1.0 %	15.91 %
More details	[10, 14]	[5, 13]	[17]	[1]	[16]	[2, 18]	this chapter

N/A = not applicable, ● = CSAR functionality implemented, ◇ = alternative functionality implemented, ○ = not implemented

applications. Details on these debug cases and the lessons learned are described in detail below.

8.1.1 Co-Processor Array SOC

The Co-Processor Array SOC SOC is an audio/video SOC developed by Philips Research to target future TV applications [10]. Its silicon implementation became available in 1999. It features a set of high-speed audio and video processors, which communicate data through a programmable network. Its 30 internal clock signals are all derived from the same 16 MHz external oscillator clock and are periodic (refer to Sect. 3.1.1).

The CPA SOC implementation was extended for debug at design time with (1) a set of hierarchically-interconnected matchers and brakes, to monitor the state of pre-selected internal signals and buses, (2) scan-based state access via the chip's TAP and using a debug scan chain wrapper around each functional module, (3) clock control functionality, to either stop the local clock domain immediately or stop all clock domains synchronously when the matcher and brake modules detect a pre-programmed sequence of debug events occurs, and to pass the TCK of the chip's TAP to the flip-flops in each clock domain for debug state access, (4) low-priority write and read functionality from the chip's TAP to and from the off-chip SDRAM, and (5) special TAP controller instructions to control this debug functionality [14].

We furthermore used the internally-developed integrated circuit debug environment (InCiDE) software [12] to access the CPA SOC. The InCiDE software used SOC environment abstraction in the form of a standardized C API, to interact with the CPA SOC, either in a simulation environment or, via proprietary TAP hardware, on its prototype evaluation board. The InCiDE is also scriptable in Tcl. This scripting functionality was used to implement the support for programming the on-chip DfD hardware.

The scan-based state access was successfully used to diagnose a video synchronization problem [14]. This problem occurred after the chip had processed approximately 50–100 input video frames. For no apparent reason, the image would disappear from the TV monitor connected to the video output of the chip. Before tape-out, the entire, timing-back-annotated SOC netlist had been simulated, but only for the first five video frames, due to the very large simulation run times. None of these video frames showed any error, leading to the incorrect conclusion that the design was error-free. Nevertheless, the silicon failed at bring-up. After taking repeated state dumps during the initialization sequence of the CPA SOC, and examining in detail the transactions on the on-chip bus, it was discovered that the internal boot read-only memory (ROM) contained a programming error, which causes the CPA SOC to loose synchronization with the input stream after some time. Once the root cause of this problem was localized, it was possible to use the debug functionality to stop the chip during its boot sequence and upload a corrected boot program in the external SDRAM. When the SOC was subsequently booted from this external

SDRAM, the output image remained stable and correct. With the ability to circumvent this initial problem, the design team could verify and demonstrate all major video processing capabilities of the chip without further problems, and in fact show that the remainder of the chip was error-free.

The CPA SOC use case proved that scan-based state access can be successfully used when debugging an SOC with periodic clocks. The CSAR debug approach leverages both this state access and clock control functionality. This use case however also brought an implementation issue to our attention in the debug scan wrapper around each functional module. This issue became apparent during pre-silicon verification. The implementation of the CPA TAP instructions couples the selection of the debug scan mode in each module to the activation of the TAP's TCK clock in each clock domain during debug scan operations. As a consequence, the timing paths between the core-level scan enable/disable signal and the local TCK clock had to be carefully matched, to prevent introducing additional normal or shift clock cycles in a clock domain. In later SOCs this problem was solved by explicitly moving the SOC mode control out of the TAP controller and into a separate building block, the TCB. The CSAR debug hardware architecture uses this second implementation for this reason.

This use case also highlighted that the configuration of the InCiDE software, by manually programming the driver modules for the on-chip DfD functionality, was rather cumbersome, as it had to be done at the abstraction level of individual bits, and preferably in C++, if the DfD module required a lot of communication with the InCiDE. The InCiDE also used an internal model of the testbench, and simulated the transfer of data between building blocks. This internal simulation engine turned out not to be used at all and was therefore abandoned for successive SOCs.

8.1.2 PNX8525 and CODEC SOCs

The target consumer products for the PNX8525 [5] and CODEC SOCs of Philips Semiconductors are digital televisions, digital video recorders, and setup boxes. Their silicon became available in respectively 2000 and 2002. Both SOCs contain more, modified, and extended debug functionality compared to the CPA SOC [17, 15]. Instead of using the matcher and brake modules, the PNX8525 and CODEC SOCs use an on-chip, real-time trace architecture to monitor key internal signals and to propagate these signals as debug events to a set of I/O pins and/or to a series of four debug counters. These counters are configurable from the TAP to signal the on-chip clock generation unit (CGU) to stop a subset of the on-chip clocks on the occurrence of a particular sequence of counted debug events. The state of each SOC can afterwards be extracted via the same TAP. Both chips also added TAP-based functional reset control, a feature that the CPA SOC does not have. This reset functionality allows the debug engineer to automate the debug process, by providing control over the on-chip functional reset signal from the InCiDE software. This reset

functionality is used in the CSAR debug approach and therefore also supported by the CSAR debug infrastructure (refer to Step 4.2 in Fig. 4.6).

The implemented debug functionality was put to good use by the design team of the CODEC SOC during first silicon bring-up and the early development of several embedded software applications [15]. A particular use case from which we learned a lot was the analysis of an audio problem. During a functional system test, several device samples had problems playing correct audio at their nominal supply voltage. No problems however occurred for these device samples when playing audio at a higher supply voltage. The manufacturing test of these samples passed on the ATE and did not show this voltage-dependent behavior. The debug process was started on the system test board, by first simplifying the audio test bench as much as possible. A breakpoint sequence was defined that allowed the state of a good chip to be compared with a failing one. After making state dumps of the content of several embedded memories over a large number of runs on the audio test bench, each time refining the breakpoint sequence, significant differences were identified between the state of the passing chip versus the failing chip. The root cause of these differences was traced to a single flip-flop containing an incorrect logic value at one particular clock cycle. Looking at the logic driving the input of this flip-flop, it was clear that it was driven by a minimal-sized buffer using a long tri-stateable bus wire, which also had a bus-keeper connected to it. Apparently under some operating conditions, this buffer was not able to drive the input of this flip-flop high against the bus-keeper before the start of the next clock cycle. These conditions did not showed up during pre-silicon timing verification or the initial structural test on the ATE. Once this was discovered this buffer was replaced with a larger version for inclusion in the planned silicon revision. The design team later estimated that the on-chip debug support saved them up to four times in debugging time, and overall up to eight times in debugging effort compared to the unsupported case.

For the PNX8525 and CODEC SOCs, we extended the functionality of the InCiDE software, by linking in a shared C++ library that reconstructed the hierarchical view of the design based on the scan chain information from the EDA tools. This hierarchical data structure could subsequently be viewed by designers in a primitive design browser window. The underlying data model contained a scan chain data structure, mirroring the scan chain data structure in the design. The hierarchical flip-flop, register, and module C++ objects contained pointers into this data structure. This new data structure, called the debug chain database (DCD), allowed a user to program these flip-flop, register, and module objects using method calls. It did not support higher-level data types yet, as values still needed to be specified in either bits or bit strings. The underlying C++ library would however take care of distributing the individual bits correctly over the scan chain data structure. A synchronization operation would then exchange the content of the scan chain data structure in the InCiDE with the content of the scan chains in the simulation model or physical SOC. This synchronization mechanism is still the basis for the SOC environment abstraction in the CSARDE (refer to Sect. 7.2.2).

Overall, five valuable lessons were learned from these two SOCs for the CSAR debug approach. First, these two SOCs proved that it is possible to debug GALS SOCs

with a scan-based debug approach. However, at the same time, we also learned that it is difficult to deterministically stop a GALS SOC. This in particular caused a problem when trying to diagnose system-test-only failures. We intended to use the same state dumping process as used to diagnose the audio problem. However, the boot sequence of the PNX8525 and CODEC SOCs proved to be non-deterministic at the clock cycle level. Each SOC reads its boot code from an external ROM using an inter-integrated circuit (I^2C) interface. We were not able to accurately generate a debug trigger each time an instruction from the external ROM was executed, because the boot and the I^2C interface modules share the same bus with other on-chip modules, which interfere with our programmed breakpoint.

Second, it proved to be difficult for the design team to identify significant differences between extracted states, as many bits in these extracted states would differ between debug experiments, both for a single SOC implementation and for different SOC implementations at the same breakpoint. Part of these differences can be explained by the imprecise breakpoint generation. Another part of these differences is caused by the undefined substate that exists in an SOC. The CSAR debug approach addresses both aspects by controlling the on-chip communication, and presenting the SOC state, at higher levels of spatial and temporal abstraction.

Third, the proposed debug functionality transferred well between different SOCs. After we made an initial implementation in the PNX8525 SOC, the design team of the CODEC SOC were able to implement the same functionality on their own, with remote consultancy only. The implementation however still took quite some effort to implement and to get working correctly before silicon. This is why in the CSAR debug approach, we have automated the insertion of the CSAR debug functionality in an SOC design with our DfD flow in Chap. 6.

Fourth, the real-time trace architecture in these SOCs was implemented using combinational gates only. As a consequence, a very large set of very long propagation paths would show up during the pre-silicon timing analysis. At the time, the only way to remedy this problem was to waive all timing constraints on these paths in this architecture. The CSAR debug architecture instead uses the EDI for debug event distribution across a GALS SOC, which is a more scalable and layout-friendly solution.

Fifth, the data model in InCiDE, in which the scan chains are the main data containers that are referenced by the flip-flop, register, and module C++ objects using internal pointers, works very well. However, in order to minimize scan chain access times for manufacturing test, DfT engineers prefer to implement configuration bits in hardware. These configuration bits control the bypass of complete test wrappers, of internal logic scan chains, and/or of internal memory scan chains. These configuration bits are part of either other scan chains or a TCB. To leverage this functionality for debug as well, and to speed up, for example, the debug access to the test wrappers around embedded memories, our C++ library would have to be extended to manage all these different, possible scan chain configurations. This created abstraction and control problems for the InCiDE software, as on the one hand the scan chain configuration was dependent on the content of particular flip-flops, while the programming of the content of these flip-flops was dependent on the active scan chain configuration.

To break this circular dependency, we therefore opt for the CSARDE to store all data with the individual Flipflop objects inside an IEnvironment object. When the debug engineer asks to perform a synchronization operation on a SOC environment, the active scan chain configuration should instead be derived top-down from this functional data structure (refer to Sect. 7.2.5). The advantage of this approach over the previous bottom-up approach, is that we can use similar behavioral abstraction techniques as we allow in the CSARDE to configure and query the other debug modules, e.g., the PSIs and monitors, for the test functionality inside the SOC. For example, a TestWrapper object could be implemented that returns a list of Flipflop objects between its debug scan input and debug scan output, depending on its current state. When it subsequently receives data from the SOC after a synchronization operation on the corresponding IEnvironment interface, it can use this data together with its current state to update the content of its internal flip-flops and subsequently its internal state. On a next query, its response will then adapt, if necessary, to any change in its internal scan chain configuration.

As indicated in Sect. 7.2.5, this functionality is currently still a work in progress, in particular because of the flexibility allowed in the implementation of the test wrappers in an SOC. This makes the implementation of a TestWrapper class with the functionality described above cumbersome. We however see no reason why this behavioral abstraction cannot be implemented in the near future. Once implemented, it should support different on-chip scan chain configurations transparently, and allow a debug engineer to gain quicker access to a specific part of the SOC state more efficiently and to focus on interpreting (differences in) the state of functional modules, without having to worry about scan chain configurations.

8.1.3 Xetal-II SOC

The Xetal-II SOC is a single-instruction, multiple-data (SIMD) processor targeted at advanced video processing application [1]. Its silicon became available in 2005. Its debug functionality consists of both silicon and basic software debug features. The Xetal-II global control processor (GCP) executes the software application. A breakpoint register can be configured from the TAP to a value for the program counter of the internal GCP. When the program counter of the GCP reaches this value, the on-chip clocks are stopped using the on-chip CGU. The InCiDE software can poll this internal breakpoint register via the TAP and determine whether this breakpoint has occurred or not. The mode of the entire Xetal-II SOC can subsequently be switched to debug scan mode via the TAP, by programming the on-chip TCB and an on-chip clock control block (CCB). The complete chip state can be extracted via the TAP and scan chains afterwards. Either the extracted or an alternative state can be uploaded at the same time. The mode of the Xetal-II SOC can then be switched back to functional mode to resume its functional operation.

The Xetal-II SOC use case demonstrates that scan-based silicon debug functionality can integrate seamlessly with traditional computation-centric debug approaches.

It allows the basic debugging of a software application that is executed on custom processors. To enable this, the InCiDE software extracts the program code and the current value of the program counter after a breakpoint occurs, and visualizes an abstracted processor state in the context of its software application. The debug engineer can subsequently step the execution of the processor at the temporal abstraction levels of individual clock cycles, instructions, or functions to validate the correctness of both the Xetal-II hardware and its software application.

8.1.4 En-II SOC

The En-II SOC is a technology demonstrator and design methodologies pipe cleaner for advanced GALS SOCs with DVFS functionality in nanometer complementary metal oxide semiconductor (CMOS) process technologies. Its silicon became available in 2007. The debug infrastructure implemented on the En-II SOC consists of the best-practices in debug functionality from the prior SOCs with several improvements [16]. The En-II SOC includes among others additional AXI monitors that observe the on-chip communication. These monitors can generate a debug event when a pre-configured transaction is detected on one of the on-chip buses. These monitors are also able to compact the transaction attributes into a checksum value. Cross-breakpoint support is provided between the embedded processors, the AXI bus monitors, the CGU, the on-chip trace buffer, and the trace output interface. It furthermore offers run/stop control over the embedded processors, using their built-in debug support, and the on-chip functional clocks, using a CCB in the on-chip CGU. This clock control allows all functional clocks to be stopped on a breakpoint in any of these SOC modules. Afterwards, the complete state of the chip can be accessed via the TAP and the scan chains. Real-time trace observability is implemented for the execution of the embedded processors, and their real-time trace information can be captured both in an on-chip trace buffer and by external data acquisition equipment. Special environment sensors allow the monitoring of the maximum process frequency, voltage drop, and local on-die temperature variations.

The En-II SOC use case demonstrates not only that the scan-based silicon debug functionality integrates seamlessly with traditional computation-centric debug approaches, but that it can also be used to control and read out on-chip process and environment sensors. We reused the InCiDE software to extract temperature and voltage levels at 14 points across the SOC die, and to visualize these graphically, while the SOC was functionally executing. This functionality was made possible by the use of on-chip TPRs in these sensors. This functionality was subsequently successfully used to validate the DVFS functionality of the En-II SOC. The En-II SOC use case also demonstrated the value of bus monitors. Compact checksum values were generated for long software application runs. A quick determination can be made about the correctness of the execution of the silicon implementation and its software application, by comparing the checksum values at the end of each run with the checksum values calculated using a reference.

We also learned three lessons with respect to the implementation of the CSAR debug functionality from the En-II SOC use case. First, the En-II SOC proved that the scan-based and run/stop-based debug functionality of the CSAR debug approach can be successfully used on a GALS SOC with DVFS.

Second, we discovered in the silicon implementation that our bus monitors did not take the byte mask field of the AXI bus into account. This can cause the checksum algorithm to use invalid data when not all bytes of a data word on the bus are fully specified. This bug has since been corrected. For example, the DTL front end in the CSAR communication monitor implements this masking.

Third, we discovered the difficulty of keeping the debug information that is used by the InCiDE software consistent with the SOC implementation throughout the complete implementation refinement process. We initially measured a debug scan chain length in the silicon implementation that did not match the available gate-level netlist implementation of the same SOC, nor the information we could extract from this netlist using EDA tools. This difference, once detected, could be corrected by locating the right set of files in the SOC design database and running the appropriate EDA tools to extract the required debug information. However, this use case does show the sensitivity of the CSAR debug approach to the correctness of the debug configuration data that is provided to it from the implementation refinement process. This is why we very closely align the generation and modification of debug configuration data with the actual implementation refinement process, as shown in Chap. 6.

8.1.4.1 CRISP-RFD

The Reconfigurable Fabric Die (RFD) SOC was developed by the partners in the European FP7-CRISP project. This SOC forms an essential part of a scalable platform template, called the General Stream Processor (GSP), which targets a wide range of future streaming DSP applications [2]. Its silicon became available in 2011. The RFD SOC was the first SOC activity that we participated in, that experimented with on-line dependability enhancement [11, 18]. The processor cores embedded in the RFD SOC were wrapped at design time in a special dependability wrapper as part of this enhancement. A dependability manager was included on-chip to generate test patterns and evaluate test responses. This dependability manager can access the scan chains of individual modules across the NoC, via a functional test port between the dependability wrapper and the NoC. Simulation-based evaluation demonstrated the ability to apply test patterns and obtain test responses from a subset of modules, while the other modules in the SOC remained in functional mode.

The functionality of the module wrappers in the RFD SOC has been incorporated in the CSAR debug approach. It has been described as the hybrid mode in Sect. 4.1.2 and implemented as part of the test wrapper in Sect. 5.7.

Compared to the functionality of the RFD SOC, the CSAR test wrapper does not use a fixed finite state machine to control the application of test patterns and extraction of test responses. Instead, all test control signals have been made available in the

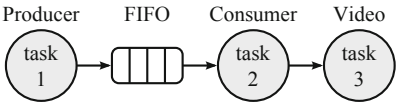


Fig. 8.1 Example application task graph

PSC register inside each test wrapper, to permit a state extraction without a prior test pattern application. This alternative operating mode of the test wrapper is necessary when we need to extract the state of a module for debugging, while minimizing the impact on the execution of the SOC.

8.1.4.2 CSAR SOC

The previous SOCs were used to develop and successfully validate most of the CSAR debug functionality presented in this book, with the exception of communication-centric SOC execution control. To evaluate the effectiveness of this aspect of the CSAR debug approach, we developed an illustrative GALS SOC model, called the CSAR SOC. The details of this SOC are discussed in the remaining sections in this chapter. We focus our discussion primarily on the temporal execution behavior of the CSAR SOC, to evaluate the effectiveness of the communication-centric aspect of the CSAR debug approach. We will however also present results of the abstraction-based, scan-based, and run/stop-based debug functionality that we implemented in this SOC.

8.2 CSAR SOC Overview

In this section, we introduce an illustrative GALS SOC, called the CSAR SOC model, and describe its software and hardware implementation. We use this SOC in the remainder of this chapter to help quantify on the one hand the impact of the complicating factors on debugging this SOC, and on the other hand the benefits and costs of the CSAR debug approach and infrastructure. We describe the application that runs on this SOC in Sect. 8.2.1, the SOC hardware architecture in Sect. 8.2.2, and its internal clock domains in Sect. 8.2.3.

8.2.1 CSAR SOC Application

The CSAR SOC executes an illustrative, multi-processor application. The *task graph* [9] of this application is shown in Fig. 8.1. This graph consists of three *tasks*: (1) a Producer task, (2) a Consumer task, and (3) a Video task. The Producer task generates data and communicates this data as *tokens* to the Consumer task using a FIFO. Each token contains eight 32-bit data words. The FIFO can hold up to 32

tokens. We use a FIFO between these two tasks to decouple their execution within the bounds imposed by the FIFO size.

The Producer task executes an infinite loop. In each loop iteration, the Producer task generates 256 tokens using a fixed data algorithm, which is agreed with the Consumer task. It writes each token in the FIFO. The Producer task blocks when the FIFO is full. The Producer updates a token counter in memory after each token, and a loop counter in memory at the end of each loop iteration. The Consumer task also executes an infinite loop. In each loop iteration, the Consumer task reads 256 tokens from the FIFO and checks whether the data content of the tokens matches the data algorithm agreed with the Producer task. The Consumer task blocks when the FIFO is empty.

The Consumer task maintains a separate status flag for each received token in each loop iteration. After all 256 tokens have been read, the Consumer task communicates these status flags to the Video task through memory mapped input/output (MMIO) operations. This Video task controls a VGA display. The VGA output image is a matrix of 16×16 rectangles, with one rectangle for each token generated in one loop iteration.

The Video task determines the color of each rectangle based on the values stored in 16 32-bit internal data registers. Each data register corresponds to one row in the video output image, and is divided in 16 pairs of data bits. Each pair controls the output color of a rectangle in the corresponding row in the video output image. The binary value "00" corresponds to a white rectangle, "01" to a blue rectangle, "10" to a green rectangle, and "11" to a red rectangle. The Consumer sets each pair of data bits to the binary value "10" (i.e., green) when the corresponding status flag is cleared, i.e., when the corresponding token was correctly received. It sets a pair to the binary value "11" (i.e., red) when its corresponding status flag is set, i.e., when the corresponding token contained erroneous data. The VGA display therefore shows a matrix of 16×16 green rectangles when the Consumer task received all tokens correctly, and one or more red rectangles for those tokens that were incorrectly received by the Consumer task.

8.2.2 CSAR SOC Hardware Architecture

Figure 8.2 shows a block diagram of the top-level of the CSAR SOC, including the mapping of the tasks and FIFO to their hardware modules.

This top-level contains eight modules. The CSAR SOC uses a NoC [7] as its communication interconnect. We generate an RTL implementation for this NoC using the flow described in Sect. 6.3. The communication specification for this NoC is provided in Sect. B.1. The solid arrows that connect the modules in Fig. 8.2 represent DTL communication links. Each arrow points from the initiator to the target module. The communication interconnect includes not only the NoC module, but also DTL CDC modules. These DTL CDC modules permit all eight modules to operate at independent clock frequencies, thereby creating a GALS SOC. The

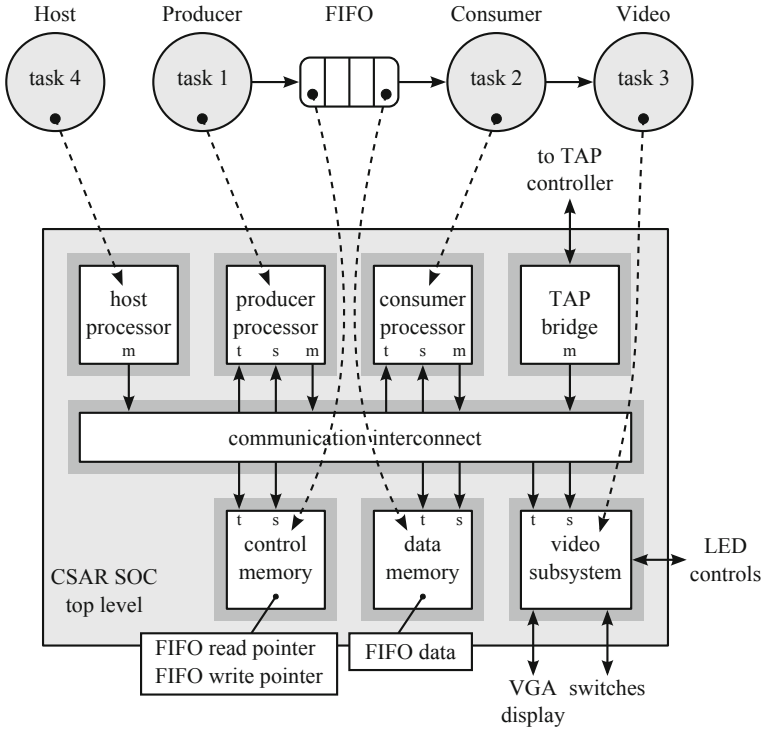


Fig. 8.2 Example, mapped application task graph

implementation of a CDC module is based on [3]. The darker borders around each module indicate the resulting eight clock domains. Note that we have not drawn the clock and reset signals of these clock domains in Fig. 8.2 for clarity.

The CSAR SOC contains four modules that are masters on the communication interconnect. These modules are the host processor, the producer processor, the consumer processor, and the TAP bridge. These four modules are used respectively to execute an additional Host task that initializes the communication interconnect on start up, to execute the Producer task, to execute the Consumer task, and to provide debug access to the on-chip communication interconnect from a TAP.

The processors are 32-bit reduced instruction set computer (RISC) processors [8]. These processors have separate instruction and data memories, and no cache memories for simplicity. All processors have a local, read-only instruction memory of 4096 32-bit data words, totaling 131,072 bits. The producer and consumer processors have a local data memory of 256 32-bit data words, totaling 8192 bits, while the host processor has a local data memory of 2048 32-bit data words, totaling 65,536 bits. Out of these 2048 data words, 658 data words are implemented as data ROM. This ROM stores the configuration data for the NoC. The data memory of each processor is mapped to the top of its data address range. Processor data accesses outside of this

range are handled via the master DTL port on each processor (indicated with "m" in Fig. 8.2). The producer and consumer processors have a slave DTL port, which allows other modules to access their local data and instruction memories using the communication interconnect.

The Producer and Consumer tasks in this application communicate with each other via a shared FIFO. A detailed description of the source code of both tasks is given in Sects. B.2 and B.3. We store the pointer information of this FIFO in the control memory and the FIFO's data elements in the data memory. It is not uncommon to separate the FIFO control information from the FIFO data, as this allows the sizes of these memories and their access times to be optimized for each application.

The host processor executes an additional Host task. The source code of this task is automatically generated as part of the NoC implementation flow and is executed from an embedded ROM inside the host processor. The host processor therefore does not have a slave DTL port. The TAP bridge module contains both a TAP-DTL bridge and a TAP-EDI bridge (refer to Sects. 5.2.5 and 5.5.4). These bridges connect respectively to a DTL interface and to an EDI interface on the communication interconnect.

The SOC contains three modules that are slaves on the on-chip communication interconnect. These modules are a control memory, a data memory, and a video subsystem. They are shown at the bottom of Fig. 8.2. Both memories contain 256 32-bit data words, totaling 8192 bits. This data is accessible via DTL slave ports (indicated with "s" in Fig. 8.2). The video subsystem includes a VGA controller, an external VGA interface, and a register file with 17 32-bit registers. The bottom 16 registers control the colors of the rectangles in the VGA output image, while the top register captures the values of eight user inputs and controls the state of eight light emitting diodes (LEDs). This register file is programmable via the DTL slave port of the video subsystem.

The producer and consumer processors, the memories, and the video subsystem have DTL test ports (indicated with "t" in Fig. 8.2), to control their test wrappers (refer to Sect. 5.7). The host processor, the communication interconnect, and the TAP bridge module do not have a test port, because they cannot be used in functional mode and in debug mode at the same time.

8.2.3 CSAR SOC Clock Domains

The hardware architecture of the CSAR SOC allows us to operate the eight modules at different clock frequencies. We have defined five sets of clock frequencies, called clock relations, to study the effects that using multiple clock frequencies has on the execution behavior of the SOC. Table 8.2 shows the details of these five clock relations. The specific clock periods in Table 8.2 were chosen within four constraints. First, the clock period of the video subsystem is fixed at 40 ns because this is required by its external VGA interface. Second, to allow conducting future experiments with the CSAR SOC implementation mapped on an FPGA board, we derive all internal clocks in this setup from one external oscillator clock signal with a fixed period

Table 8.2 Clock periods per module for five example clock relations

	Clock relation				
Module	A	B	C	D	E
Producer	110 ns	1010 ns	110 ns	1010 ns	110 ns
Consumer	110 ns	1010 ns	110 ns	110 ns	1010 ns
Control memory	110 ns	110 ns	1010 ns	110 ns	110 ns
Data memory	110 ns	110 ns	1010 ns	110 ns	110 ns
NoC and Host	20 ns	20 ns	20 ns	20 ns	20 ns
Video subsystem	40 ns	40 ns	40 ns	40 ns	40 ns
LCM	440 ns	44,440 ns	44,440 ns	44,440 ns	44,440 ns
GCD	10 ns	10 ns	10 ns	10 ns	10 ns

of 10 ns. The clock periods of all clock domains are therefore chosen to be multiples of the 10 ns period of this base clock. Third, the multiplication factors were chosen prime so that the resulting clock signals result in a relatively high LCM period of 44,440 ns. We use this common clock period later in this chapter to show that we lose valuable debug data when we use this period for a sampling clock. Fourth, the clock of the NoC and the host processor use a clock period of 20 ns to run much faster than the other functional modules. This helps to minimize their influence on the execution behavior of the Producer and Consumer tasks.

Within these constraints, we vary the clock periods of the modules in these five clock relations relative to each other. In clock relation A, all master and slave modules with the exception of the video subsystem use the same clock period of 110 ns. Clock relation A is the baseline against which we measure the behavior of the SOC for the other clock relations. In clock relation B, both the producer and the consumer processors run almost an order of magnitude slower than the rest of the SOC. This makes the high-level application handshake between the Producer and Consumer tasks on the one hand and the FIFO on the other hand dominant. In clock relation C, the producer and consumer processors run much faster than the memories. This makes the individual, low-level DTL handshake between the modules in the SOC dominant, as these processors stall to wait for responses from the memories. In clock relation D, the producer processor runs almost an order of magnitude slower than the rest of the SOC. The Consumer task stalls to wait for the Producer task to write data in the FIFO. This situation is reversed in clock relation E, where the consumer processor runs almost an order of magnitude slower than the producer processor. For this clock relation, the Producer task stalls to wait for the Consumer task to read data from the FIFO.

We necessarily restrict ourselves to these five clock relations, because it is not possible to perform an exhaustive validation for all possible clock relations (refer to Sect. 1.2.3). Note that the execution of our example application for these five clock relations results in the same output image on the VGA display in the absence of errors in the implementation. This is possible due to the use of both the low-level handshake mechanism on the DTL communication links and the high-level handshake mechanism via the FIFO, which decouple the functional outcome of the execution from the specific clock relation used.

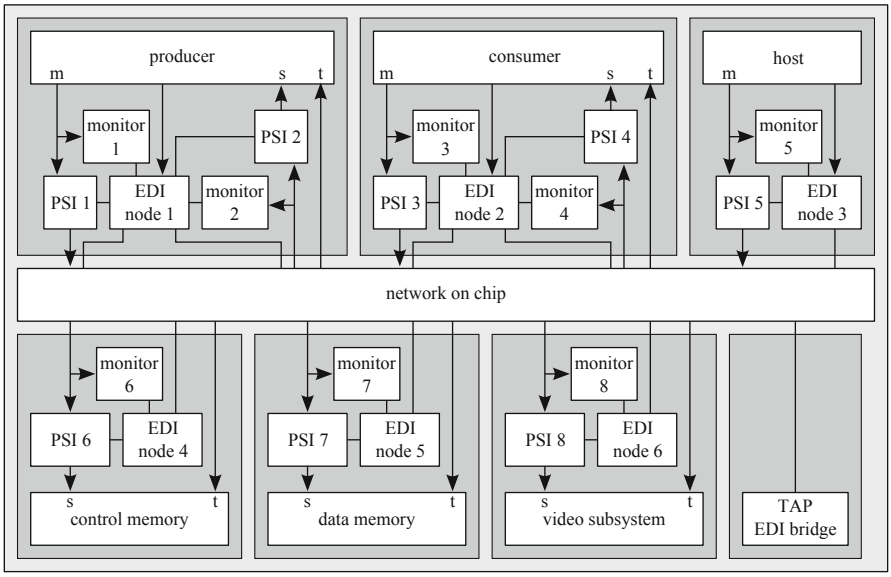


Fig. 8.3 High-level overview of the CSAR SOC DfD instrumentation

8.3 Application of the CSAR DfD Flow

8.3.1 Overview

We applied the CSAR DfD flow to the modules of our example SOC. A high-level overview of the resulting debug instrumentation is shown in Fig. 8.3.

We use the debug wrapper generation tool to instrument the producer and consumer processors with DTL PSIs on their master and slave ports. At the same time, we add DTL monitors to the initiator sides of these PSIs. These monitors allow us to monitor the communication to and from these processors. We furthermore instrument the host processor with a single DTL PSI and an initiator-side monitor on its master port. The control and data memories, and the video subsystem are instrumented with DTL PSIs and initiator-side monitors on their slave ports. We connect these CSAR debug modules together using EDI nodes, with an EDI node in the debug wrapper around each module.

The generated NoC contains the remainder of the EDI (not shown). This EDI has the same topology as the NoC and connects to the EDI nodes in the debug wrappers around the other modules. We do not instrument the DTL ports of the NoC, because the master and slave DTL ports of the other modules are already instrumented. We subsequently use the top-level integration tool to connect all modules together in the top-level module shown in Fig. 8.2. We then use the chip-level integration tool to combine this top-level module with the CSAR global TCB and a custom CRGU in

Table 8.3 DfD tool configuration files versus generated DfD configuration files and VHDL files, per component, in lines of code

Type	Metric	Input	Output	Increase
XML configuration files	XML start tags	782	50,765	6391.68 %
VHDL RTL files	Lines of code	24,575	39,422	60.42 %

a chip-level module. We subsequently use the boundary-scan-level integration tool to combine this chip-level module with a custom TAP controller and boundary scan chain in a boundary-scan-level module. This completes our implementation of the CSAR SOC.

8.3.2 DfD and Tool Configuration Effort

We analyzed the amount of effort that we spent on the configuration of the CSAR DfD flow, and on the generation and instantiation of the CSAR hardware modules for the CSAR SOC. Table 8.3 tries to quantify this effort in relation to the resulting output of the flow.

We wrote XML configuration files for the debug tools in our DfD flow and XML architecture and communication specification files for the NoC design flow [6]. The DfD flow outputs a DfD boundary-scan-level configuration file that contains a detailed description of the on-chip debug architecture. This file is also an XML file. We counted the number of XML start tags in these files to obtain a metric for their complexity. We see in Table 8.3 that the DfD flow adds a substantial amount of information to the information we provide as input. In particular, the DfD flow adds access and name details for all the 40,590 scannable, functional flip-flops in the CSAR SOC to the output file, for use by the CSARDE.

We also counted the number of source code lines for each VHDL RTL input and output file of the flow using the program `cloc` [4]. We only counted lines with actual source code, i.e., the comment and empty lines were excluded. We see in Table 8.3 that the DfD flow generated 14,847 lines of VHDL RTL source code. The larger part of this generated code is related to the NoC and its test wrapper (55 %). This is because the NoC flow generates complete VHDL RTL implementation files of the NoC, and because the VHDL RTL description of the test wrapper around this generated NoC is large due to the large number of DTL interfaces that are wrapped for manufacturing test in this wrapper.

8.3.3 DfD Flow Execution Time

We measured the execution time of the individual steps in the CSAR DfD flow. Table 8.4 summarizes our results.

Table 8.4 Amount of time spent on each step in the CSAR DfD flow

Step	Real time (s)	User + kernel time (s)
1. Module generation	1.80	0.79
2. Debug wrapper generation	6.55	7.10
3. Gate-level synthesis	2674.58	2569.45
4. Test wrapper generation	8.95	11.85
5. Top-level integration	8.70	11.96
6. Chip-level integration	5.94	7.44
7. Boundary-scan-level integration	6.03	7.59
Total time	2710.75 45 min 10 s	2615.39 43 min 35 s

These steps were executed on a computer with an eight-core Intel Xeon E5-2667 processor with the Red Hat Enterprise Linux 5 OS and running at 2.9 GHz. The execution times were measured with the Linux command `time`. The second column lists the wall clock time, while the third column lists the time that each tool spent in user or kernel mode combined[1].

Please note that the time reported for the module generation step only includes the time required to generate the custom NoC implementation for this SOC. We do not include the time required to design the processors, memories, video subsystem, and TAP bridge. This seems a valid method to calculate the productivity when using a design reuse strategy. These functional modules are already available in such a strategy, because they have been pre-designed and used in an earlier SOC, or they have been bought externally. Table 8.4 shows that the gate-level synthesis of the modules dominates the overall execution time of the CSAR DfD flow. The ratio between the execution time of the other steps and the total execution time of the flow is 1.33 %.

8.3.4 CSARDE Configuration

We configured the CSARDE described in Chap. 7 for the CSAR SOC using the boundary-scan-level generation DfD configuration file that is generated by the DfD flow. We did not need to take any additional actions to support the CSAR debug hardware modules described in Chap. 5, as these modules are by default supported by the CSARDE, and configured using the information in the DfD configuration file.

The CSAR SOC does however contain other modules, i.e., the processors, the control and data memories, and the video subsystem. As an example extension of the CSARDE, it is possible to add support for these modules to the CSARDE using the extension mechanisms described in Chap. 7. For the processors, someone can

[1] The number reported in the second column of Table 8.4 can be less than the number reported in the third column, because the tools were executed on a computer with eight cores. The OS may therefore take advantage of these compute resources by multi-threading the execution of each tool.

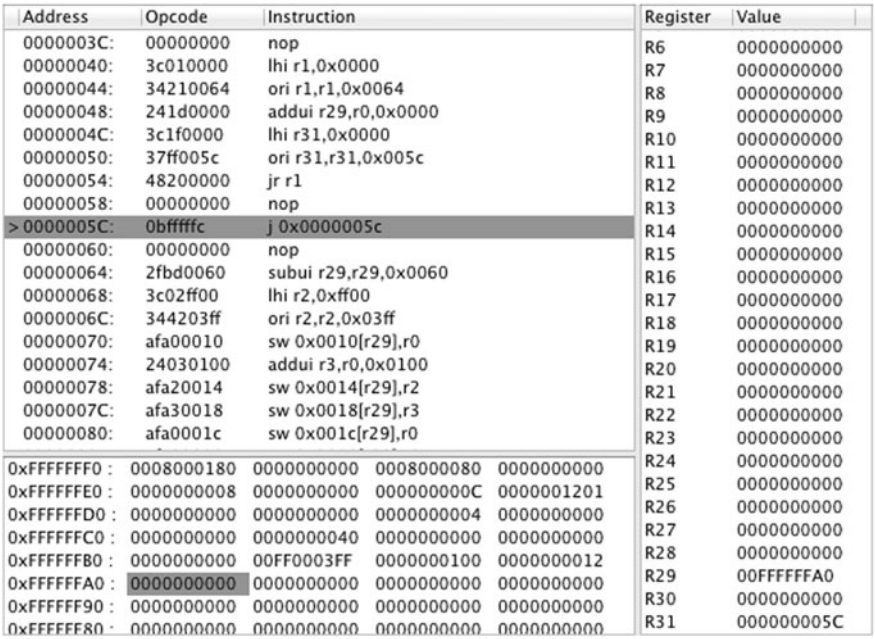


Fig. 8.4 Example custom GUI for the RISC processors in the CSAR SOC

provide a traditional instruction view of the source code, together with a table of register values and of data memory content (refer to Fig. 8.4).

This view could disassemble the content of the processor's instruction memory and present it as a listing of assembly instructions in the top-left corner of the view (indicated with ①). This subview can also show a horizontal bar to indicate the current value of the program counter. This allows a user to quickly inspect where the processor is in the execution of this code. The content of the 32 processor registers can be shown on the right side of the view (indicated with ②). The content of the local data memory can be shown on the bottom side of the view (indicated with ③). This subview could furthermore highlight the location of the stack pointer, to allow a user to easily examine the content of the processor stack.

Similarly, a view could be provided with two tab folders for the video subsystem (refer to Fig. 8.5). A user could then inspect the state of the video subsystem in two representations; as a video output image, or as a table with its 17 registers and their values. The view shown in Fig. 8.5 shows an example of the video output image. For example, a user could change the content of a register in the video subsystem by clicking on the corresponding rectangle in this image, thereby cycling through the four possible color values.

For both the control and data memory in the CSAR SOC, a view can be provided that is similar to the subview for the content of the data memory in the processor view (indicated as ③ in Fig. 8.4).

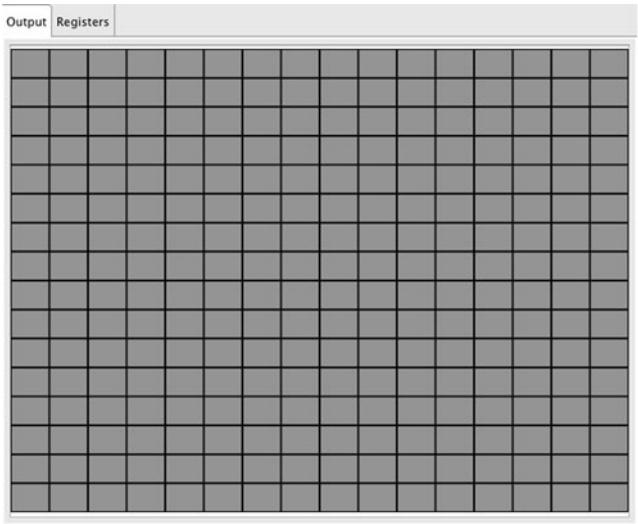


Fig. 8.5 Example custom GUI for the video subsystem in the CSAR SOC

8.4 Evaluating the Complicating Factors for Debugging

The CSAR SOC is intentionally kept small for illustration purposes. In this section, we show however that the task of debugging this SOC is already complicated by the nine factors identified in Chaps. 2 and 3. An analysis of how each factor complicates the debugging of this particular SOC is provided below. In addition, we show how the CSAR debug approach and infrastructure help to reduce or eliminate the impact of each factor.

8.4.1 Silicon Area Cost

After gate-level synthesis, we measured the characteristic of the CSAR SOC and the silicon area required for its functional modules and its on-chip CSAR hardware modules. The key characteristics of the CSAR SOC are:

- 8 logic modules
- 5 read-write memory modules
- 3 read-only memory modules
- 40,590 scannable, functional flip-flops
- 414,272 ROM bits
- 77,248 RAM bits
- 9 clock domains[2]
- 14 input pins
- 38 output pins

[2] including the video output clock

Table 8.5 CSAR SOC silicon area results

Name	Silicon area
Functional modules	59.56 %
Scan insertion of functional logic	17.08 %
Communication monitors	9.52 %
Test wrappers	7.45 %
EDI	4.66 %
PSI	0.99 %
Other DfD logic	0.74 %
Total	100.00 %
NAND2-equivalent gates	431,141

Table 8.5 shows the silicon area cost of this SOC, grouped by function, and sorted by area.

The functional modules contribute the most to the total silicon area. The second largest area contributor is the *scan insertion* step, i.e., the step in which all flip-flops in the design are replaced by scannable flip-flops. Our DTL communication monitors have the third largest contribution with 9.52 %. The reason why these monitors make a large contribution is because each monitor contains three 32-bit data matchers and a sizable memory for the event sequencer. The test wrappers around the nine modules have the fourth largest contribution. The fifth largest contributor is the EDI with 4.66 %.

We see overall that the ratio between the silicon area required for the CSAR DfD logic and the total silicon area of this SOC is 15.91 % and for manufacturing test is 24.53 %. These are relatively large percentages compared to other publications, e.g., [13]. This is due to the modules in the CSAR SOC being kept very simple for illustration purposes. For larger, more realistic SOCs we anticipate that the amount of silicon area required for our CSAR DfD modules to be similar to the amounts published in literature, i.e., below 5 % (refer to Table 8.1).

8.4.2 CSAR SOC Observability and Controllability

8.4.2.1 Intrinsic Observability and Controllability

We evaluated the intrinsic observability and controllability of the CSAR SOC. The physical, boundary-scan-level interface of the CSAR SOC consists of 14 input and 38 output pins. These 14 input pins are used for an external clock input signal, an asynchronous external reset input signal, four TAP input signals, and eight general-purpose user input signals. The external clock and reset inputs are functionally required by the application. The remaining 12 input pins can in principle be reused as debug I/O pins without affecting the functional behavior of the application. The 38 output pins are used for a video output clock signal, 4 video control signals, 24 video data signals, 8 LED control signals, and a TAP output signal. None of these outputs are strictly needed by the application, so all 38 pins can in principle be reused as debug I/O pins as well. This gives us a total of 50 pins that can be reused as debug I/O pins.

When we limit ourselves to debug observability with no debug controllability, we can use these 50 pins as debug output pins and try to stream the state of the SOC out via these pins. There are however 40,590 functional flip-flops and 77,248 RAM bits to observe. When we take a conservative clock period of 110 ns from Table 8.2 for all clock domains[3], then this SOC still generates approximately 1.1 Tb/s, which amounts to 21.4 Gb/s per available debug pin. These operating speeds are beyond the current technological capabilities for the I/O pins on mobile devices (refer to Fig. 1.2).

We furthermore only have the external clock input and asynchronous external reset input to control the execution of the application. When we assert the reset input, the application starts from its initial, reset state. Even though we can subsequently make the application run slower or faster by respectively decreasing or increasing the frequency of the external clock signal, we cannot start the execution of the application at any other point than its initial state, nor influence the order in which the transactions are generated by the master modules and processed by the slave modules inside the SOC.

We conclude that we have intrinsic spatial and temporal observability and controllability limitations when we try to debug the execution of this SOC (refer to complicating factors CF-1 and CF-2).

8.4.2.2 Spatial Observability and Controllability Using the CSAR Debug Infrastructure

With the CSAR debug infrastructure, we have three means to access and control the SOC state. First, we can use the TAP-DTL bridge module to functionally read and write any SOC state that is accessible from the on-chip communication interconnect while the SOC continues its execution. We used this access means to change the content of the register file in the video subsystem to display a different output than the normal, error-free output. As an example, please compare the normal video output in Fig. 8.6 to the modified video output shown in Fig. 8.7.

We can obtain this output image by first stopping the Consumer task using the DTL PSI on its master DTL port, and then writing 16 specific 32-bit data words to the register file in the video subsystem, to form the pattern shown in Fig. 8.7.

Note that we have to stop the Consumer task first to prevent it from overwriting the register file content with new status flags. Because the simulation to write the entire register file in the video subsystem would take a large amount of simulation time, we opted to only verify the access to a single register. In a silicon debug scenario, speed would not be an issue. Through this experiment, we were able to prove the accessibility of the video register file in the functional mode of the SOC using the TAP-DTL bridge.

Second, we can stop the execution of a single module by using the DTL PSIs on its DTL ports and switch its test wrapper to functional debug mode. This allows

[3] Note: the video subsystem, the host processor, and the NoC use clock periods that are lower, respectively 40 ns, 20 ns, and 20 ns.

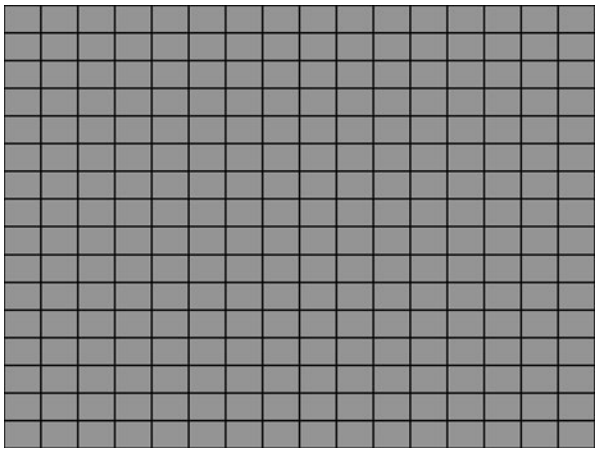


Fig. 8.6 Error-free output of the video subsystem

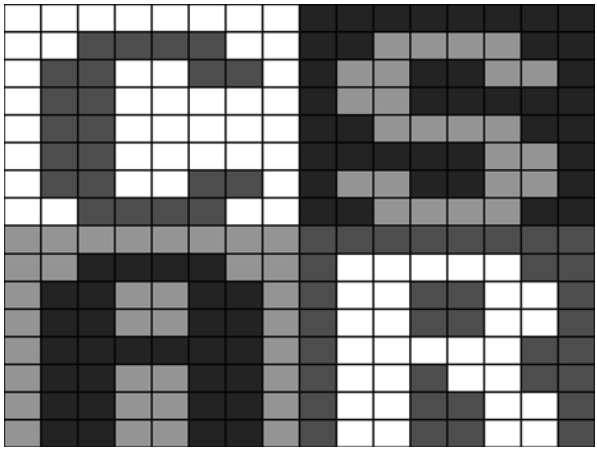


Fig. 8.7 First modified output of the video subsystem

us to access the entire state of that module from the CSARDE through its DTL test port. As an example, we used this access mechanism to reset the content of the register file in the video subsystem to display a different output than the normal, error-free output. The video subsystem in the CSAR SOC contains 1204 flip-flops. The top of the address space of its DTL test port of the video subsystem A_{FF} is therefore 38 (refer to Eq. 5.16 on p. 133). After stopping all DTL communication to the video subsystem, we switched its test wrapper to functional debug mode by writing to address 0x0 on its DTL test port. We subsequently used a single 32-bit read operation from address 0x4 to verify retrieving 32 bits of content from the scan chains in the video subsystem. Each read operation also automatically shifted logic-0's in the scan chains of the module. Figure 8.8 shows what the final video output image will look like after all 38 data words would have been read. The color of all rectangles in this image have been reset to white.

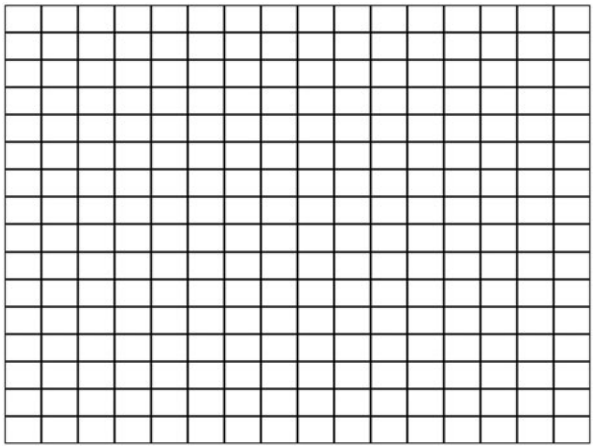


Fig. 8.8 Second modified output of the video subsystem

Third, we can functionally stop the execution of the entire SOC, by using the DTL PSIs on all DTL ports, and afterwards change the SOC mode to the DEBUG_SCAN mode (via the CLOCKS_STOPPED mode). A subsequent synchronize operation can retrieve the entire state of the SOC. We show this execution control functionality in our use cases in Sect. 8.5.

8.4.2.3 Temporal Observability and Controllability Using the CSAR Debug Infrastructure

With the CSAR debug infrastructure, we can exercise control over the generation and delivery of DTL transactions, messages, and data elements across the on-chip communication interconnect. As a first demonstration, we used this functionality to guide the execution of the SOC by taking complete control over the communication inside the chip.

To create a reference for our experiment, we used the CSAR debug infrastructure to have all DTL monitors calculate a checksum value for all the data elements on their DTL interfaces during one loop iteration. We then executed the application once for each of the five clock relations in Table 8.2. We expect the resulting checksum values to be different for each clock relation, as the execution of the SOC is influenced by the changes in the relative relations between the clock domains.

We then performed our *guided replay* experiment, in which we first allow the Host task to completely configure the NoC, before the Producer and the Consumer tasks are allowed to take turns to respectively writing one token in the FIFO and reading one token from the FIFO. After 256 tokens, we let the Consumer task write its status information to the Video task. We expect that the calculated checksum values will be exactly the same for all five clock relations, provided that we guide the execution and that we thereby indeed control the order in which all transactions are generated and delivered.

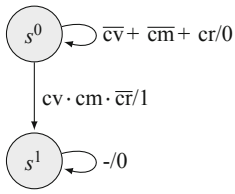


Fig. 8.9 Event sequencer STG to detect a specific DTL command address

To validate this hypothesis, we need to be able to examine the checksum values as calculated by the DTL monitors. For this we however need to stop the communication inside the SOC, otherwise the DTL monitors continue their checksum calculations for another loop iteration. To stop all functional communication, we configured PSIs 1 and 3 on the master DTL ports of the producer and consumer processors to stop the transactions on an EDI event. This event is generated when the Producer task updates the loop counter at the end of a loop iteration. All the communication in the SOC effectively stops when we stop the transactions on these DTL ports. To generate this event, we programmed Monitor 1 in Fig. 8.3 to detect a write transaction from the Producer task to the control memory at address 0x0000000C, which indicates an update of the loop counter in the control memory. We configured the command data matcher in Monitor 1 to match on the address 0x0000000C and the event sequencer to implement the STG shown in Fig. 8.9.

In this STG, the label "cv" refers to a command valid condition, the label "cm" to a command match condition, and the label "cr" to a read command.

The event sequencer enters state s_0 on reset. It stays there for as long as there is no value command, the command data matcher does not match, or there is no write transaction. As soon as there is a valid and matching write command, the event sequencer transitions to state s_1, while it generates an event on the EDI. The event sequencer then stays in state s_1 until it is reset.

We furthermore configured the EDI to transport this EDI event via EDI node 1, to PSI 1 to stop all transactions on the master DTL port of the producer processor on an EDI event. We also transported the EDI event through the EDI in the NoC to EDI node 2 and PSI 3 (refer to Fig. 8.3). PSIs 1 and 3 were configured to stop the transactions on their DTL communication link on an event from the EDI. After we configured the CSAR debug modules, we functionally reset the SOC to start its execution. We subsequently polled PSIs 1 and 3 to determine when all the transactions inside the SOC have been stopped. When this was the case, we accesses the checksum registers in all DTL monitors. We repeated this experiment for all five clock relations to obtain the checksum values to test this hypothesis.

We use the checksum values obtained using clock relation A as a reference and report in Table 8.6 their Hamming distances with the checksum values obtained for the four other clock relations. These results have been obtained through simulation with a manually-created testbench, based on the CSARDE script in Listing C.4 in Sect. C.3. We needed to create such a testbench by hand, because applying the output of this script to the SOC model and the synchronization operation are still manual tasks (refer to Sects. 7.2.2 and 7.2.5). Moreover, the scanning of the TPR

Table 8.6 Hamming distances between the checksum values

Checksum	A ↔ B	A ↔ C	A ↔ D	A ↔ E
Producer command group	11	11	10	17
Producer write group	0	0	0	0
Producer read group	10	13	19	9
Consumer command group	13	0	15	0
Consumer write group	0	0	0	0
Consumer read group	14	14	15	16
Control memory command group	1	2	1	2
Control memory write group	17	14	12	12
Control memory read group	13	17	16	9
Data memory command group	4	4	4	2
Data memory write group	0	0	0	0
Data memory read group	0	0	0	0
Video command group	0	0	0	0
Video write group	0	0	0	0
Video read group	0	0	0	0
Host command group	0	0	0	0
Host write group	0	0	0	0
Host read group	15	15	15	0

chain and scan chains take an excessive amount of simulation time. We therefore conducted all our experiments by directly controlling the control outputs of our TPRs, and observing their status inputs and the SOC state, using the simulator scripting facilities.

The second through fifth columns in Table 8.6 shows that we obtain unique sets of checksum values for each clock relation when we do not control the execution of the SOC. In this scenario, the relation between the on-chip clocks determines in which order the transactions are executed. Note how some checksum values do not change when the clock relation is changed, while other checksum values do. This is because each clock relation has a different interleaving of the transactions of the Producer and the Consumer tasks. As a consequence, the Producer or the Consumer task may need to poll the control memory more or fewer times to determine that the FIFO is respectively not full or not empty. This difference affects the DTL command groups that are shown to have different values in Table 8.6 and not the other groups. Also note that the configuration of the network by the Host task using the host command and write group is not subject to variation. The network status, which is obtained using the host read group is however subject to variation.

We demonstrated the temporal observability and controllability that the CSAR debug infrastructure provides by guiding the execution of the SOC, independent from the specific clock relation used. In this scenario, we control the execution of the SOC via its on-chip PSIs. We first allow the Host task to completely configure the communication interconnect, while the DTL transactions of the Producer and Consumer are blocked. We subsequently allow the Producer task to initialize the FIFO pointers and the Consumer task to read these pointers. We then step through the generation of a complete token, by allowing the Producer task to write one token of 16 data words to the data memory and update the FIFO read and write pointers and

the token counter in the control memory. Afterwards, we allow the Consumer task to query the FIFO write pointer and read the written token from the data memory.

After repeating this procedure for 256 tokens, we allow the Consumer task to write 16 32-bit data words, with 256 status flags to the Video task. Next to controlling the PSIs to guide the SOC execution along this very specific, feasible execution path, we also use the DTL monitors to calculate a checksum value for all transactions that occur on their DTL interfaces. The CSARDE script that is needed for this is provided in Listing C.5 in Sect. C.3. We re-execute our manually-created testbenches for all five clock relations listed in Table 8.2, and obtained checksum values from all DTL monitors at the end of the first loop iteration.

When we control the SOC execution as described using the on-chip PSIs, we enforce a particular execution scenario, and control the exact order in which the transactions are executed. This results in the exact same transaction order to take place each time we execute the SOC, and in identical checksum values at the end of every debug experiment. This proves the hypothesis and allows us to very precisely guide the execution of an SOC along a feasible execution path that activates an error for subsequent analysis.

8.4.3 Undefined Substate and State Comparison

We analyzed the undefined substate of the CSAR SOC during its execution and the impact of the undefined state on our ability to compare the state between SOC implementations at different abstraction levels. To perform this analysis, we used a functional simulation testbench with a gate-level implementation of the CSAR SOC. We used clock relation A in this experiment and defined a debug sample clock with the LCM of the clock periods of the module clocks as its period (refer to Table 8.2). On each rising edge of this debug sample clock, we sampled the states of all functional flip-flops and RAM bits and wrote them to a file. We subsequently processed the resulting files and counted the number of undefined bits in each file. The result is shown in Fig. 8.10. This figure shows on the horizontal axis the simulation time at which the testbench generated each file, and on the vertical axis the percentage of flip-flop and memory bits that were defined, i.e., had a logic-0 or logic-1 value.

Figure 8.10 shows that no part of the SOC state is instantly reset when the external reset input signal is asserted at $t = 0$ ns. We can understand this because we do not have functional state that can be asynchronously reset. Instead all functional flip-flops have to be synchronously reset, which requires the presence of a clock signal. The internal clock signals are at logic-0 while the PLLs are not yet locked to the external clock input signal.

The first time that we see that part of the SOC state has been initialized is at $t = 313885$ ns. From then on, up to 77.75 % of all flip-flops is initialized as the application executes. The flip-flops that are not initialized are flip-flops that are part of several small memory buffers, such as the FIFOs in the NoC. These buffers have associated status registers that indicate whether the buffer contains any valid data

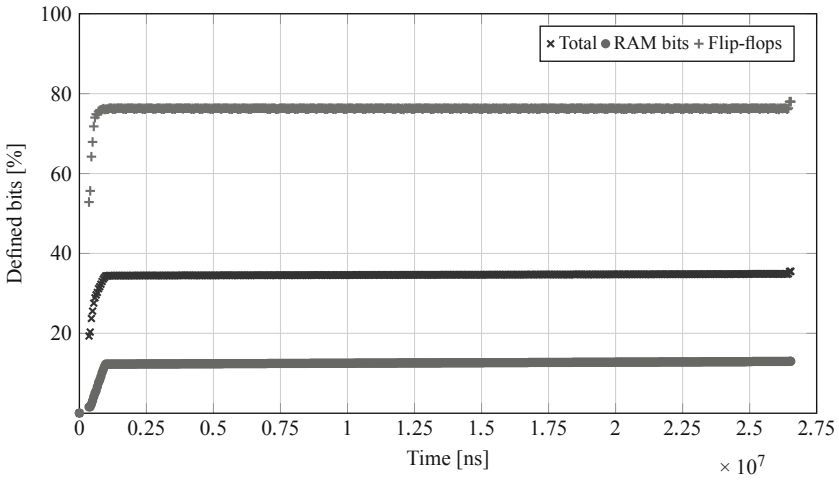


Fig. 8.10 Undefined substate as measured during CSAR SOC execution

and if so, where this data is located in the associated buffer. It is therefore not necessary to initialize the state of the buffer itself, as long as these status registers are properly initialized. The total percentage of initialized flip-flop bits therefore saturates below 100.00 %, as these buffers do not need to be completely initialized during the execution of the application.

The small increment at the end of the application is caused by the Consumer task communicating the status flags to the Video task. This operation causes additional FIFO buffers in the NoC between the consumer processor and the video subsystem to be used and subsequently initialized with valid data.

The percentage of RAM bits that are reset when the asynchronous external reset signal is asserted is also 0 %. We expect this because none of the embedded memories have asynchronous reset capability. When the producer processor comes out of reset, it initializes the FIFO read and write pointers, and the token and loop counters in the control memory, and subsequently writes tokens in the data memory. The number of initialized memory bits saturates at 12.55 %, even when the data memory is completely filled with valid tokens. This is because we also have (1) the other RAM bits in the control memory that are not used by this application, and (2) the data RAM bits of the processors. Most RAM bits in the control memory remain unused and therefore uninitialized. The data memories of the processors are used to implement a stack to support the nesting of program functions. The actual size of this stack and therefore the number of memory bits that are initialized in this stack, depend on the number of (nested) function calls and the size of the arguments of these functions. As visible in the source code of the Producer and Consumer tasks in Sects. B.2 and B.3, neither task uses any functions besides a `main` function. A very large part of the data memories of both processors therefore remains uninitialized.

Figure 8.10 shows that the application executes without problems, while 77,230 state bits remain uninitialized. In a silicon implementation, these state bits will take a random value at power-up, i.e., either logic-0 or logic-1. When we subsequently compare the states of two silicon implementations at the exact same point in their execution, this comparison will yield approximately half this amount, i.e., 38,615 state bits, in validly-mismatching state bits. Clearly we need abstraction methods to abstract away from these uninitialized state bits, to only compare the bits that have been initialized in the context of the application, and find the cause for any differences in the values of initialized bits. We show the effect of applying our abstraction techniques in Sect. 8.4.5.

Another reason to need abstraction methods is to bridge the difference in abstraction level between a silicon implementation of our SOC and for example its VHDL RTL implementation. We have seen that the silicon implementation of the CSAR SOC has in total 532,110 state bits. The RTL simulation instead has 1659 RTL register signals. Of these signals, 1205 signals are either multi-bit register signals or two-dimensional memory signals. Additionally, 572 of these multi-bit register signals have an enumerated type with more than two possible values.

Abstraction techniques therefore need to be applied to be able to compare the values of these 1205 RTL signals that represent multi-bit registers and two-dimensional memories, with the values of the $532,110 - 1659 + 1205 = 531,656$ state bits that are extracted by our CSAR infrastructure. The CSARDE offers support for the necessary abstraction techniques, explained in Sects. 4.1.3 and 7.3 to perform this comparison.

8.4.4 Transient Errors

We looked at transient errors that may occur within the CSAR SOC. One example of a transient output error is when one or more tokens that are exchanged between the Producer and the Consumer tasks contains an error. Without internal SOC observability, such an error may cause glitches in the output image sequence on the VGA display connected to the Video task. We come back to transient errors in Sects. 8.5.3 and 8.5.4, in which we use the CSAR debug approach and infrastructure to localize respectively a *transient, certain error* and a *transient, uncertain error* in an implementation of the CSAR SOC.

8.4.5 Non-determinism at Clock-Cycle, Handshake, and Transaction Levels

We already saw evidence of non-determinism at transaction levels in Sect. 8.4.2, when we examined the effects that different clock relations have on the checksum values calculated during the execution of the CSAR SOC. In this section, we conduct

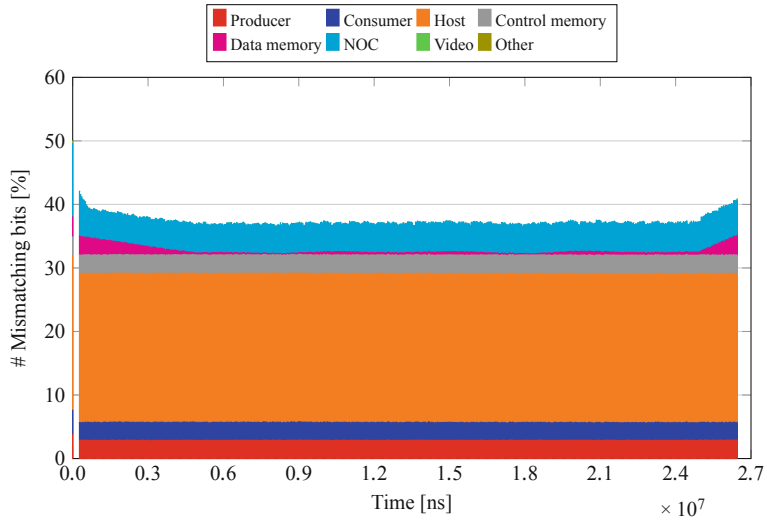


Fig. 8.11 Mismatching bits at each NoC clock cycle

two additional experiments to demonstrate the non-determinism at the clock-cycle and transaction abstraction levels, and the usefulness of the CSAR debug approach.

8.4.5.1 Non-Determinism at Clock-Cycle Level

We conducted an experiment to examine the effects that undefined bits and different internal clock relations have on the state of a GALS SOC at absolute points in time. We simulated the execution of the CSAR SOC twice, once for clock relation A and once for clock relation D. During each execution, we sampled the global state of the CSAR SOC on the common periodic master clock. This master clock has a clock period of 44,440 ns (refer to Table 8.2). At each clock cycle of this master clock, we compared the values of all state bits, i.e., of all flip-flops and embedded memories, between the two executions. The state values we found were either logic-0, logic-1, or undefined. We counted a match when both state bits were defined and equal, and a mismatch when both state bits were defined and unequal. We counted the cases where either state bit was undefined as a half mismatch. The reason for counting these cases as half mismatches, is because undefined state bits do not exist in a silicon implementation. Instead, the corresponding flip-flop or memory bit will have a random value, i.e., either a logic-0 or a logic-1 value with equal probability. The chance that the value of another state bit matches this random value is therefore 50 %, even when the value of this other state bit is also undefined. Figure 8.11 shows the results that we obtained for this experiment.

Figure 8.11 plots the number of counted mismatches for each clock cycle of the periodic master clock. This figure shows that on average around 35 % of the state

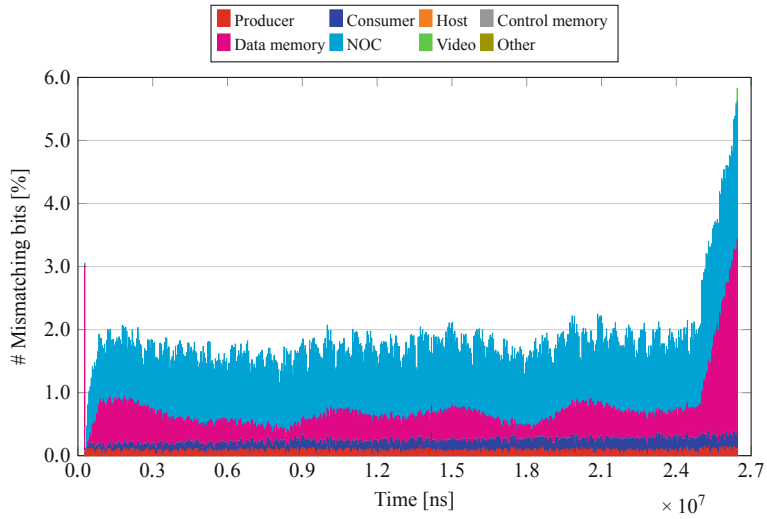


Fig. 8.12 Mismatching bits at each NoC clock cycle, after structural, data, and behavioral abstraction

bits do not match. Note that there is no error in the SOC implementation, so these mismatches are purely caused by the undefined state bits in the SOC and the use of different clock relations. Traditional silicon debug methods classify these 35 % of state bits as non-deterministic state noise and subsequently apply statistical filtering to remove them from consideration. We instead propose to apply structural, data, and behavioral abstraction to the two states extracted during the two experiments first, and then performing the comparison again.

In this state abstraction, we can apply knowledge that we have of the SOC implementation and the application that runs on top of its hardware. For example, we see that a large portion of the mismatching bits are located in the Host. Closer inspection tells us that these bits are part of the data memory that the Host does not use during its execution. Similar, the data memories in the producer and consumer processors are not extensively used in this application, nor is the control memory. The control memory only has to hold the two FIFO pointers. The state of the remainder of the control memory stays uninitialized. When we apply this knowledge to discard the comparison results for these bits in these memories, then we obtain Fig. 8.12.

Figure 8.12 still shows the mismatching bits in the CSAR SOC per clock cycle of the periodic master clock, but after the application of our structural, data, and behavioral abstraction techniques. In these techniques, the aforementioned undefined memory bits are considered to match based on our knowledge of the SOC and its application. As can be seen in Fig. 8.12, applying these abstraction techniques significantly reduces the number of mismatching bits, and yields on average 2.0 % of bits, whose values are not the same in the two SOC executions at the same absolute point in time. It is interesting to see the increase in mismatching bits at the right hand

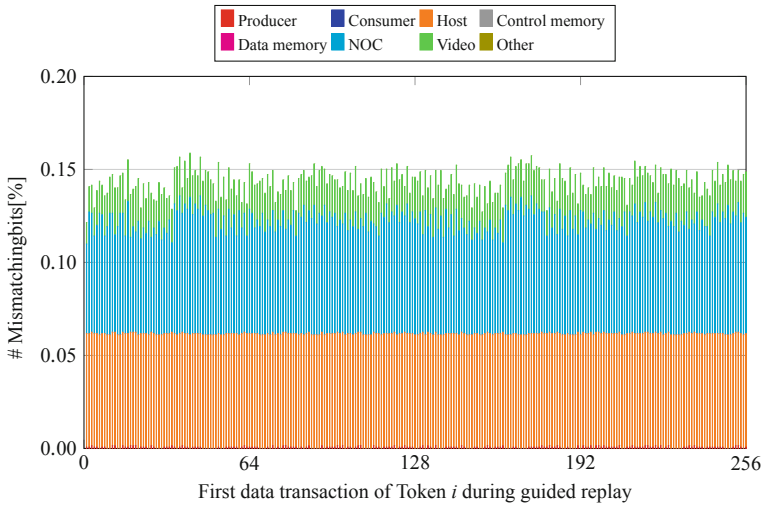


Fig. 8.13 Mismatching bits at the start of the first data transaction of each token in our guided replay process

side of the graph. This is caused by one execution already having finished the first loop iteration and starting a new iteration, while the other execution is still in its first loop iteration.

The remaining mismatches are caused by the difference in clock relations that are used for the two executions. We can try to filter the effect of the resulting temporal difference by applying temporal abstraction. In this abstraction, we no longer look at the state of the SOC in every clock cycle of the periodic master clock, but instead, we can look at the state of the SOC after every transaction in the system. To illustrate this, we again use the guided replay process described earlier, and enforce an absolute order in the two SOC executions. We sample the state of the entire SOC whenever the producer is ready to write the next token into the data memory. As the Producer writes 256 tokens in each loop iteration, we obtain per execution 256 samples of the SOC state. Figure 8.13 shows the number of bits that mismatch between these samples for the executions with the two different clock relations.

Figure 8.13 no longer uses the clock cycle of the periodic master clock on the horizontal axis. Instead it uses the transactions that write the first data word of each token into the data memory. When we compare Fig. 8.13 with Fig. 8.12, we see that we have made another significant step in reducing the mismatches among the state bits. This significant step is made possible by us taking control over the execution of the SOC in both cases, and sampling the SOC state at the same transaction for both. This removes the clock-cycle non-determinism that occurs in the execution of the CSAR SOC when a different clock relation is used.

We can explain the remaining mismatches in Fig. 8.13. We see mismatching bits in the host processor. After the host processor has configured the NoC, it no longer interacts with the other SOC modules. This causes it to perform internal computations

that are no longer synchronized with the rest of the SOC, due to the absence of any interactions of the host processor with the on-chip communication interconnect.

Another large part of the mismatching bits are in the NoC. Even though we have stopped the communication inside the SOC, the NoC still continues to perform internal computations. We see the effect of these computations back in a small amount of mismatching state. The same holds for the mismatching bits in the video subsystem. Even though the communication in the SOC has been stopped, the video subsystem continues to output a VGA image. For this, it needs to update internal counters to, among others, generate the right control signals for the externally connected VGA monitor. Again, we see these internally updating counters back in a small amount of mismatching state bits.

Overall, this experiment does clearly show that we can do better in explaining mismatches between SOC state bits, by applying structural, data, behavioral, and temporal abstraction techniques, as we do with the CSAR debug approach. Having these techniques at our disposal when we debug actual erroneous SOC implementations allows us to focus on those mismatches that are caused by the error we need to locate, and not because of the low-level non-determinism in the execution of a GALS SOC.

8.4.5.2 Non-Determinism at Transaction Level

We furthermore examined the impact of a small analog delay in the locking time of the on-chip PLLs on the timing of the transactions of the producer processor. Figure 5.30 shows that the CRGU ensures that the internal clock domains are held in functional reset, while the on-chip PLLs are not yet locked to the external clock input signal. In each execution of a GALS SOC, it may however take these PLLs a different amount of time before they successfully lock to this external clock signal. We examine the impact even a small amount of naturally-occurring variation in this lock time can have on the execution of the CSAR SOC.

We focus in this experiment on when the write transactions of the Producer task occur, relative to the deassertion of the external reset signal. In particular, we study the effect of a variation in the lock delay on when the first command handshake for each token occurs. We use the notation $t(i, \Delta_{lock})$ to represent the amount of time between the deassertion of the asynchronous, external reset signal and the point in time at which this handshake occurs for the i^{th} token, while forcing the Plls to take Δ_{lock} nanoseconds longer to lock their output clock signals to the external clock input signal. We furthermore use the notation $x(i, \Delta_{lock})$ to represent the time difference between the point in time of the i^{th} write transaction of the Producer task occurs in the presence of this PLL lock delay of Δ_{lock} and without. Equations 8.1 and 8.2 hold under the hypothesis that a delay in the lock time of the PLLs causes an identical delay in the point in time that the write transactions of the Producer task occur.

$$t(i, \Delta_{lock}) = t(i, 0) + \Delta_{lock} \tag{8.1}$$

$$x(i, \Delta_{lock}) = t(i, \Delta_{lock}) - t(i, 0) = \Delta_{lock} \tag{8.2}$$

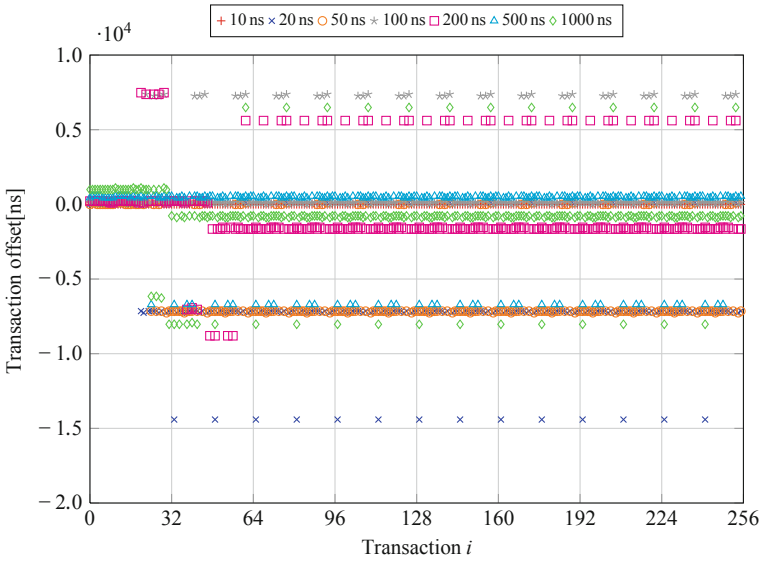


Fig. 8.14 Producer transaction offsets as a function of a small analog lock delay for clock relation A

Figures 8.14 and 8.15 show the producer transaction offsets $x(i, \Delta_{lock})$ that we measured in a simulation with the gate-level implementation of the CSAR SOC. We used clock relations A and D in these simulations. We plot the producer transaction offsets as a function of the transaction number i between 1 and 256, and for a number of small analog lock time delays (10 ns, ... , 1000 ns).

Clearly the measured producer transaction offset values $x(i, \Delta lock)$ are not equal to the PLL lock delay Δ_{lock}. This means that the hypothesis stated earlier is invalid. We see that some transactions occur at the same point in time ($x(i, \Delta_{lock}) = 0$), while others occur earlier or later. We conclude that the execution of this SOC is non-deterministic at the clock-cycle, handshake, and transaction levels, due to small variations that can also occur naturally.

The CSAR infrastructure improves on this situation by making its breakpoints dependent on the actual occurrence of the communication handshakes and transactions. This allows a debug engineer to stop the SOC execution much closer to the same point in the communication between the modules in the chip, than when clock cycle counters are used.

8.4.6 Uncertain Errors

We looked at uncertain errors that may occur within the CSAR SOC. One example of an uncertain error is when the conditions under which the SOC has to operate, cause a momentary voltage drop on its on-chip power grid. This drop may in turn cause

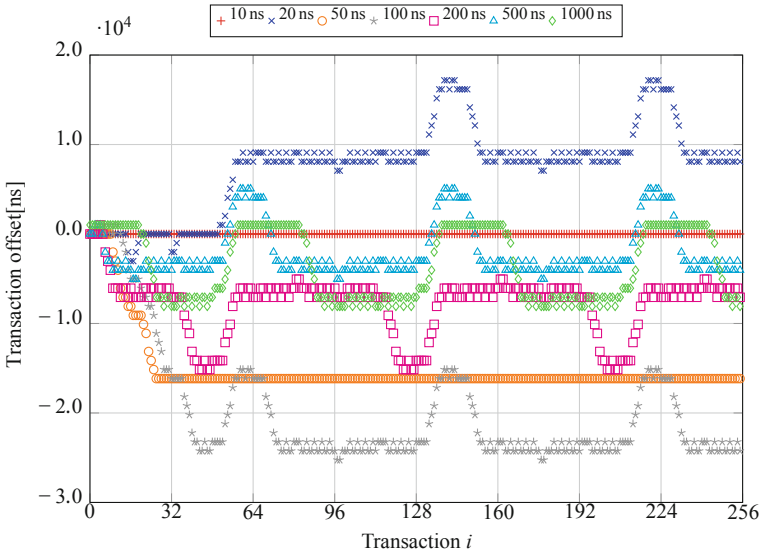


Fig. 8.15 Producer transaction offsets as a function of a small analog lock delay for clock relation D

the propagation delay of a subset of its logic gates to become slower than normal. When this occurs for gates on a critical path in the design, then that path may no longer meet its setup timing constraint (refer to Eq. 3.1). This in turn can cause an inconsistent state in the SOC. We come back to this type of error in Sect. 8.5.4, in which we use the CSAR debug approach and infrastructure to localize a transient, uncertain error.

8.4.7 No Instantaneous, Distributed, Global Actions

We identified possible state sampling artifacts as a result of the complicating factor CF-9 for GALS SOCs. The implementation of the CSAR SOC allows all eight modules to operate at their own clock frequency. As such, these eight clock domains may run completely asynchronous from each other. The CDC modules that are used on the DTL communication links between the modules ensure that the functional data is communicated without the risk of metastability. When there is a LCM period for all clock signals, then this period may be so large that it causes intermediate state data to be unobservable.

For example, the LCM period for clock relation A is 440 ns. Both the producer and consumer processor execute four clock cycles during this period. In clock relation C, they even execute 404 clock cycles during the LCM period of 44,440 ns. Clearly, even when there is a LCM clock period, using it may lead to the loss of state observability. When the clock frequencies are dynamically changing during the execution of the

SOC, e.g., as part of a DVFS technique to reduce the SOC's power consumption, then it will be extremely difficult, if not impossible, to choose a good single debug sample clock.

The CSAR debug approach and infrastructure work around this problem by first functionally stopping the communication in the CSAR SOC using the PSIs to ensure that we obtain a global state that is both locally- and globally-consistent.

8.5 Use Cases

8.5.1 Overview

In this section we evaluate the application of the CSAR debug approach and infrastructure to help localize the cause of three different types of error in the gate-level implementation of the CSAR SOC. These errors are (1) a *permanent, certain error*, (2) a *transient, certain error* , and (3) a *transient, uncertain error* .

8.5.2 Debugging a Permanent, Certain Error

An error in an SOC implementation may be both *permanent* and *certain* in nature. When it is, then any execution of the SOC will activate the error. We can therefore choose to guide the execution of the SOC, as described in Sect. 8.4.2, to localize its cause.

To demonstrate this using the CSAR SOC, we introduced a permanent, certain error in the program ROM of the Host task. This error resembles the error described in [14]. At the start of each debug experiment, we programmed the DTL monitors in the SOC to calculate checksum values based on every data element on their DTL interface. We then stepped through the execution of the SOC, and after each step, extracted the checksum values for all DTL monitors. Listing 8.1 shows the post-processed result of applying this *guided replay* to the erroneous implementation.

Listing 8.1 Hamming distances between and the values of the calculated checksum values

```
1  ...
2  101: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3  102: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4  103: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
5       * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 FB5E1FED 650B6107 0
6       * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 FB5E1FED 650B6100 0
7  104: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0
8       * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 F96E5051 3644AD37 0
9       * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 F96E5051 3285B083 0
10 105: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0
11 ...
```

This experiment was performed in simulation. We used the CSARDE script in Listing C.5 in Sect. C.3 to manually prepare the simulation testbench for this experiment.

On line 2 we first see a transaction count, followed by the Hamming distances between the checksum values calculated by the DTL monitors in the erroneous SOC implementation and those calculated in a golden reference. The order in which these distances are reported is the same as in Table 8.6. We clearly see that up to transaction 102, the executions of both SOCs result in identical checksum values. Only in transaction 103, we see that the checksum for the Host write group starts to deviate. On lines 5 and 6 we show the specific checksum values retrieved. Note how the last digit of the checksum values for the DTL write group of the host processor differs. Note also that the checksum value for the Host command group is correct and stays correct. The addresses used by the Host processor therefore is not affected by the error. We conclude that the Host processor writes incorrect data on its DTL interface at that point. Traditional computation-centric debug approaches can subsequently be used to trace the cause of this error, which in our case was in the Host's instruction ROM.

Note additionally how all checksum values other than the ones from the Host processor are still at their initial value 0x0. This is because up to this point in our guided replay process, we have not allowed any transaction to take place on the associated DTL interfaces.

8.5.3 Debugging a Transient, Certain Error

To demonstrate an example transient and certain error, we included a DTL error injector module in the CSAR SOC implementation, between the NoC and the data memory. This module is configured from the global TCB to inject an error in the write data of a particular DTL transaction of the producer processor. Throughout this example, we configured this error injector module to invert the LSB of data word 4 of Token 17[4]. The effect of this error is that in the first loop iteration, this token will be received by the Consumer task with an error in its data. The Consumer task will signal this to the Video task, which will output a red rectangle on the VGA display to indicate this. In the next loop iteration, no errors are injected causing the Consumer to receive all tokens with correct data and the Video task to subsequently output rectangles that are all green (refer to Fig. 8.6). The only observable effect of this error on the outside of the SOC therefore is a momentary glitch in the output image, specifically in the color of the rectangle that corresponds to Token 17. This error is *certain*, because it occurs during every execution of the CSAR SOC, and *transient*, because the erroneous state is replaced with a correct state in the next loop iteration.

Having observed this momentary glitch in the output image, we can investigate its cause by stopping all SOC communication at the end of the first loop iteration. Once all communication has been stopped, we switch the SOC mode to debug scan mode and access the entire SOC state to print the content of the register file of the video subsystem and the data memory. We reused the script in Listing C.4 in Sect. C.3 to

[4] Note that we count data word and token indices from 0.

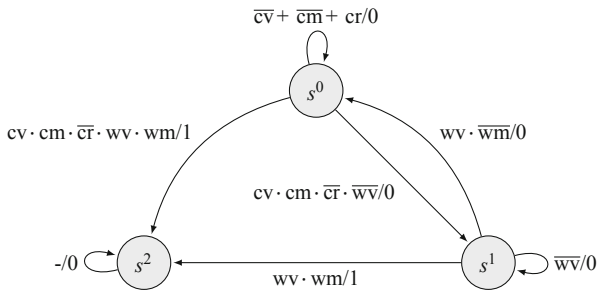


Fig. 8.16 Event sequencer STG for the detection of a specific DTL command address and write data

manually prepare another testbench. In this testbench, we stop all communication inside the SOC at the end of the first loop iteration and extend the simulator script to print the state we are interested in at the end, based on the code from Listing C.6. Listing 8.2 shows the content of the register file of the video subsystem after the first loop iteration.

Listing 8.2 Error localization in the video subsystem register file

```
0x00: 0xAAAAAAAA 0xBAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x10: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x20: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x30: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
```

Note that the injected error is visible in the value of a bit pair in the second word in this register file. This leads us to the conclusion that Token 17 contained an error. It does however not yet indicate what caused this error. Listing 8.3 shows the content of the data memory at the same point in time. Note that this data memory no longer contains any information regarding Token 17, as it is now filled with the data words of Tokens 224–255.

Listing 8.3 Data memory content after one loop iteration

```
0x000: 0x0000F000 0x0000F001 0x0000F002 0x0000F003 ←T:224,W:0-3
0x010: 0x0000F004 0x0000F005 0x0000F006 0x0000F007       .
0x020: 0x0000F008 0x0000F009 0x0000F00A 0x0000F00B       .
0x030: 0x0000F00C 0x0000F00D 0x0000F00E 0x0000F00F       .
0x040: 0x0000F100 0x0000F101 0x0000F102 0x0000F103 ←T:225,W:0-3
. . .
0x3F0: 0x0000FF0C 0x0000FF0D 0x0000FF0E 0x0000FF0F ←T:255,W:12-15
```

To examine which part of the token data path from the Producer Task to the Video task contains the error, we need to rerun the debug experiment with a breakpoint at an earlier point in time. We choose to stop the communication inside the SOC when the Producer task writes a data value of 17 into the token count variable in the control memory. We therefore configure Monitor 1 to have its command data matcher match the value 0x00000008, i.e., the memory address of the token count variable for the Producer task, and to have its write data matcher match the value 0x00000011, i.e., Token number 17. We configure its event sequencer to the STG shown in Fig. 8.16.

The STG in Fig. 8.16 is more complex than the STG in Fig. 8.9, because we now need to match both the command and the write groups of the DTL interface. This

match can take place in one of two ways. The first way is when the handshakes for both groups take place at the same time. The event sequencer transitions in that case from state s^0 to s^2, while generating an event on the EDI. The condition for this transition is that we have a valid command ("cv"), a command match ("cm"), a write command ("$\overline{cr}$"), a valid write ("wv"), and a write match ("wm"). When the event sequencer is in state s^2, it stays there. A second way for this match to take place is when first the handshake on the command group takes place, and only afterwards the handshake on the write group takes place[5]. When a matching command handshake takes place, the event sequence transitions from state s^0 to state s^1. The condition for this transition is that we have a valid command, a command match, a write command, and no valid write.

Once in state s^1, the event sequencer has to transition to state s^2 when a valid write and a write match take place. It simultaneously has to generate an event on the EDI. If the write handshake uses the wrong data, i.e., when there is a valid write, but no write match, the event sequencer has to transition back to state s^0. The event sequencer waits in state s^1 until it sees the write handshake associated with the command handshake that causes it to transition to the state s^1. The event sequencer waits in state s^0 when neither match condition place. We again configure PSIs 1 and 3 to stop the communication on an event on the EDI.

We subsequently reset the SOC functionally to restart its execution. Listing 8.4 shows the extracted data memory content after the PSIs have stopped the on-chip communication and we have extracted the SOC state in the debug scan SOC mode.

Listing 8.4 Error localization in the data memory

```
1 0x000: 0x00001000 0x00001001 0x00001002 0x00001003 ← T:16,W:0-3
2 0x010: 0x00001004 0x00001005 0x00001006 0x00001007      .
3 0x020: 0x00001008 0x00001009 0x0000100A 0x0000100B      .
4 0x030: 0x0000100C 0x0000100D 0x0000100E 0x0000100F      .
5 0x040: 0x00001100 0x00001101 0x00001102 0x00001103 ← T:17,W:0-3
6 0x050: 0x00001105 0x00001105 0x00001106 0x00001107
7 0x060: 0x00001108 0x00001109 0x0000110A 0x0000110B      .
8 ...
9 0x3F0: 0x00000F0C 0x00000F0D 0x00000F0E 0x00000F0F ← T:15,W:12-15
```

Listing 8.4 shows that the LSB of data word 4 of Token 17 is incorrectly received by the data memory. We can therefore conclude that the cause of this error is located on the path from the Producer task to the data memory. We do however not yet know which part on this path is responsible for this error.

To examine this path in more detail, we once again have to rerun our debug experiment. We use the script shown in Listing C.8 to manually create another testbench. This time we configure Monitor 1 to generate an event on the EDI when the Producer task writes the correct data value for data word 4 of Token 17 to the correct location in the data memory. This involves configuring the monitor to generate a debug event on a transaction with a command address of 0x08000050 and a write

[5] Note that the DTL protocol does not permit the write handshake to occur before the associated command handshake.

data value of 0x00001104. If the DTL monitor never generates an event, then we know that the Producer task and the producer processor are responsible for this error. If the DTL monitor does generate an event, and thereby stops the communication inside the SOC, we at least know that they do not contain this error. We furthermore configured PSI 7 on the DTL slave port of the data memory to stop on an event on the EDI. This allows us to keep the data on this particular transaction inside the NoC, to determine whether the NoC contains an error or whether the problem is located at the data memory. We rerun the debug experiment and again extract the SOC state after the on-chip communication has been stopped. Listing 8.5 shows the content of the CDC module on the output of the NoC towards the data memory. Note how the write data is still correct in this module.

Listing 8.5 Correct data in the NoC output CDC towards the data memory

```
1  Command FIFO:                Write FIFO:
2  0x0C:   0x000800004C         0x0C:  0x1f00001103
3  0x08:   0x0008000048         0x08:  0x1f00001102
4  0x04:   0x010800018C         0x04:  0x1f00001101
5  0x00:   0x0008000050         0x00:  0x1f00001104   ← correctwritedata
6  Write   pointer:  0x1        Write  pointer:  0x7
7  Read    pointer:  0x0        Read   pointer:  0x6
```

Listing 8.6 shows the content of the data memory at this same point in time.

Listing 8.6 Data memory content before data word 4 of Token 17 is written into the data memory

```
1  0x000:  0x00001000 0x00001001 0x00001002 0x00001003   ← T:16,W:0-3
2  0x010:  0x00001004 0x00001005 0x00001006 0x00001007     .
3  0x020:  0x00001008 0x00001009 0x0000100A 0x0000100B     .
4  0x030:  0x0000100C 0x0000100D 0x0000100E 0x0000100F     .
5  0x040:  0x00001100 0x00001101 0x00001102 0x00001103   ← T:17,W:0-3
6  0x050:  0x00000104 0x00000105 0x00000106 0x00000107   ← T:1,W:4-7
7  0x060:  0x00000108 0x00000109 0x0000010A 0x0000010B     .
8  0x070:  0x0000010C 0x0000010D 0x0000010E 0x0000010F     .
9  0x080:  0x00000200 0x00000201 0x00000202 0x00000203   ← T:2,W:0-3
10 . . .
11 0x3F0:  0x00000F0C 0x00000F0D 0x00000F0E 0x00000F0F   ← T:15,W:12-15
```

It is clear from this content that the data memory has not yet received this data word. We subsequently single-step the write data transaction on the DTL slave port of the data memory using the local PSI, PSI 7. We afterwards inspect the content of the data memory and observe that an incorrect data word has been written in the data memory. We again obtain the incorrect data memory content that is shown in Listing 8.4. We therefore conclude that the DTL communication link between the NoC and the data memory is probably responsible for the observed error.

To validate this hypothesis, we correct the content of the data memory. We subsequently disable the PSIs on the Consumer and the data memory, and reconfigure the producer DTL PSI to stop on the update of the loop counter (as before). We then reset the event sequencer inside the Producer's DTL monitor, and issue continuation requests to the Producer, Consumer, and Data memory PSIs. We extract the content

of the video subsystem after all communication in the SOC has stopped again. Listing 8.7 shows that at the end of the first loop iteration, the video subsystem register file now does contain the correct values.

Listing 8.7 Correct data values in the video subsystem register file

```
0x00: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x10: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x20: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
0x30: 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA
```

We therefore conclude that the failure was caused by an incorrect DTL interface between the CDC module on the NoC output towards the data memory and the data memory itself. Now that it has been localized in the gate-level implementation, details on this failure can be forwarded to the failure analysis team to further stress test this particular interface with special test patterns.

8.5.4 Debugging a Transient, Uncertain Error

The error in an SOC implementation may also be *transient* and *uncertain* in nature. This is for example the case when the previously-described, transient, certain error occurs only in some of the debug experiments, but not in all of them. We can recreate this type of error by randomizing the occurrence of the previously-described, transient and certain error. The net effect of the randomness in the occurrence of this error is that we observe that sometimes the video output image glitches and other times it does not. Investigating this error more closely, we can observe that the register file of the video subsystem sometimes contains an error in its second register and other times it does not. Because this error is not visible in every debug experiment, we conclude that the error is an *uncertain error*. We may therefore have to execute step 8 in Fig. 4.5 multiple times to activate this error and observe its effect on the state of the SOC.

However, after we have observed this error in at least one debug experiment, in the incorrect content of the register file of the video subsystem, we are still able to backtrack this error to the CDC module on the output of the NoC towards the data memory, as we did in Sect. 8.5.3. We observe that for each experiment this module, this module always contains the correct data. By also single-stepping in every debug experiment the delivery of this data element to the data memory and observing that the data memory content is sometimes incorrect afterwards, we can conclude that sometimes the DTL interface between the NoC and the data memory injects an error in the data. We forward this conclusion to the failure analysis team, as we did with the transient, certain error.

Even though the randomness in the occurrence of the error is inconvenient, the required debug process can be automated using the CSARDE and its scripting engine. This frees up valuable debug resources in the SOC validation team to work on other issues, while the CSARDE performs this backtracking procedure.

8.6 Summary

In this chapter, we introduced an illustrative GALS SOC and showed that the factors that we identified in Chaps. 2 and 3 already complicate debugging this SOC. We demonstrated how the CSAR debug approach and infrastructure effectively addresses these factors and can be used to localize the cause of permanent and transient, and certain and uncertain errors. Our approach can help localize the root cause of transient, uncertain errors and *transient*, even though this type of error is difficult to reproduce. In our post-silicon debug process, we calculate checksum values on key internal processes and automate the debug experimentation loop. This permits a significant number of runs to be executed without human interaction. As soon as the error is reproduced once, a debug engineer can take the next step in reducing the spatial and temporal scope of the debug experiment to further localize the root cause of the error.

References

1. A.A. Abbo, R.P. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, B. Vermeulen, and M. Heijligers. Xetal-ii: A 107 gops, 600 mw massively parallel processor for video scene analysis. *IEEE* Journal of Solid-State Circuits, 43(1):192–201, 1 2008.
2. Tapani Ahonen, Timon D. ter Braak, Stephen T. Burgess, Richard Geissler, Paul M. Heysters, Heikki Hurskainen, Hans G. Kerkhoff, Andre B. J. Kokkeler, Jari Nurmi, Jussi Raasakka, Gerard K. Rauwerda, Gerard J.M. Smit, Kim Sunesen, Henk van Zonneveld, Bart Vermeulen, and Xiao Zhang. Crisp: Cutting edge reconfigurable ics for stream processing. In Joao M. P. Cardoso and Michael Hubner, editors, *Reconfigurable Computing - From FPGAs to Hardware/Software Codesign*, chapter 9, pages 211–238. Springer Science, 2011.
3. Clifford E. Cummings. Simulation and synthesis techniques for asynchronous fifo design, 2002.
4. A. Danial. Cloc—count lines of code, 2012.
5. 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, 18(5):21–31, 2001.
6. Andreas Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, Eindhoven University of Technology, 2009.
7. Andreas Hansson and Kees Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 99–108, New York, NY, USA, 2009. ACM.
8. John Hennessy and David Patterson. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann, 2003.
9. 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. *Design Automation for Embedded Systems*, 7(3): 233–270, 2002.
10. Marino T. J. Strik, Adwin H. Timmer, Jef L. Van Meerbergen, and Gert jan Van Rootselaar. Heterogeneous multiprocessor for the management of real-time video and graphics streams. Solid-state circuits. In *IEEE Journal of Solid-State Circuits*, volume 35, pages 1722–1731, 2000.

11. T. D. ter Braak, S. T. Burgess, H. Hurskainen, H. G. Kerkhoff, B. Vermeulen, and X. Zhang. On-Line Dependability Enhancement of Multiprocessor SoCs by Resource Management. In *Proc. International Symposium on Systems on Chip*, pages 103–110. IEEE Circuits & Systems Society, 9 2010.
12. G.J. Van Rootselaar and B. Vermeulen. Silicon debug: scan chains alone are not enough. In *Proc. IEEE International Test Conference*, pages 892–902, 1999.
13. Bart Vermeulen. Functional Debug Techniques for Embedded Systems. *IEEE* Design and Test of Computers, 25(3):208–215, 2008.
14. B. Vermeulen and G.J. van Rootselaar. Silicon debug of a co-processor array for video applications. In *Proc. High-Level Design Validation and Test Workshop*, pages 47–52, 2000.
15. Bart Vermeulen and Hervé Vincent. DfD-Assisted System Test Analysis for Manufacturing Test Program Improvements. In *Informal Proc. IEEE Workshop on Silicon Debug and Diagnosis*, Ajaccio, Corsica, France, 5 2004.
16. Bart Vermeulen and Sjaak Bakker. Debug architecture for the En-II system chip. *IET Computers & Digital Techniques*, 1(6):678–684, 11 2007.
17. B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and debug strategy of the pnx8525 nexperia^{TM} digital video platform system chip. In *Proc. IEEE International Test Conference*, pages 121–130, 2001.
18. Xiao Zhang, Hans G. Kerkhoff, and Bart Vermeulen. On-Chip Scan-Based Test Strategy for a Dependable Many-Core Processor Using a NoC as a Test Access Mechanism. In *Proc. Euromicro Symposium on Digital System Design*, pages 531–537, Los Alamitos, 9 2010. IEEE Computer Society.

Part V
Related Work, Conclusion, and Future Work

Chapter 9
Related Work

Abstract In this chapter we look at work that relates to the CSAR debug approach and its infrastructure. We discuss related work to increase the internal observability of a silicon SOC implementation in Sect. 9.1, and related work to increase a debug engineer's control over the execution of a silicon SOC implementation in Sect. 9.2. We give an overview of related debug standardization in Sect. 9.3. In Sect. 9.4, we describe related debug tools for implementing DfD support in an SOC implementation, for accessing an SOC in its debug environment using a standardized API, and for using the on-chip debug support to facilitate the subsequent debugging of a silicon SOC implementation by a debug engineer. We furthermore present an overview of debug algorithms in literature in Sect. 9.5.

9.1 Internal Observability and Controllability

In Sect. 9.1.1, we first look at the internal observability that is provided by physical and optical debug methods. We subsequently discuss the internal state observability and controllability that is intrinsically available for functional reasons in Sect. 9.1.2. In Sect. 9.1.3, we review related work on DfD to increase the internal state observability and controllability.

9.1.1 Intrinsic Physical and Optical Observability

A debug engineer can use traditional *physical* and *optical debug methods*, such as emission microscopy (EMMI) [38], laser voltage probing (LVP) [76], time-resolved photo-emission [72] (also known as picosecond imaging circuit analysis (PICA) [44]), and wafer probing [13], to obtain internal observability of an IC. These methods have three advantages. First, these methods are available without necessarily having to prepare the SOC implementation pre-silicon for their post-silicon use (debug requirement DR-4). Second, they are *non-intrusive*, provided that removing the package and preparing the silicon die for this type of internal observability causes no behavioral side effects (debug requirement DR-3). Third, these methods provide internal observability at the lowest abstraction level, i.e., at the voltage levels on wires between transistors (debug requirement DR-1). These methods are therefore

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6_9,

often used to confirm the root cause of a failure once it has been identified, or as a last resort to find the root cause of a failure when all other methods fail.

These methods however also have three disadvantages when we try to apply them to SOCs that are fabricated in nanometer process technologies. First, the initial set of candidate transistors and wires to probe to find the root cause of a failure is very large. Second, these transistors and wires are small and may be hard to access due to the metal layers above and/or below them (complicating factor CF-1 and Fig. 1.2). Silicon back-side probing methods [28, 40] help to reduce this problem. Third, the preparation of the silicon die for each observation takes a long time and is often (very) expensive (debug requirement DR-4).

Overall, these methods tend to be very slow and expensive in their use. These methods can therefore benefit from a logical debug method, which helps localize the root cause of a failure. This logical debug method first narrows down the probable location of the root cause of a failure to a small set of candidate locations using higher abstraction levels, such as the gate level or RTL. These physical methods can subsequently be used to further reduce this candidate set to a single location at the physical level, and thereby confirm the root cause of the failure. As demonstrated in Sect. 8.5, the CSAR debug approach and infrastructure can serve as such a front-end debug method that provides specific guidance at the logical level, on when and where to examine the SOC execution with these physical and optical debug methods.

9.1.2 Intrinsic Functional Observability and Controllability

A debug engineer can reuse any functional path that is part of the SOC to access (part of) its state. For example, an embedded processor can be (re)programmed to perform read and write operations that query the state of other SOC modules via the functional communication interconnect. It can subsequently send this state data out of the SOC via a functional interface. This includes, for example, the common practice to add "printf" or other logging statements to the source code of a program that executes on an embedded processor. This program queries the state of the processor and/or other SOC modules, and send this data out via, e.g., an universal asynchronous receiver/transmitter (UART) or another functional port. When a functional interface gives direct access to the on-chip interconnect, then this interface can be used for debugging as well. An example is the CRISP-RFD chip, presented in Sect. 8.1. This SOC has a multi-channel port (MCP), from which the on-chip communication interconnect and the SOC modules connected to it can be accessed. This MCP can therefore be used for functional debugging.

The advantage of reusing the functional access paths for internal observability and controllability is that these methods can be used without having to prepare the SOC implementation pre-silicon for their post-silicon use (debug requirement DR-4). These methods however also have two disadvantages. First, the spatial scope of this approach is limited to those SOC modules that can be reached via this interconnected, and to the parts of the state that they expose via their MMIO registers (complicating

factor CF-1). Typically this state is much smaller than the complete state of a module. Second, the reuse of existing access paths and resources that are part of the SOC implementation for functional reasons is intrusive (debug requirement DR-3). Its use for debug can change the SOC execution and thereby cause errors to disappear, or become transient or uncertain. This makes them more difficult to localize.

In contrast, the CSAR debug approach does not rely on intrinsic functional observability, but instead adds DfD hardware to guarantee that the entire state is observable and controllable in the debug scan mode. The required DfD hardware to support the CSAR debug approach comes at a cost of a pre-silicon debug effort and a silicon area cost, but enables us to overcome the spatial scope limitations and the intrusiveness of these related methods. An example of this improved resolution was the localization of the error in Sect. 8.5.3, on the interface between the NoC CDC and the data memory. Without our on-chip hardware, the resolution of our error localization would have been limited to the entire path from the producer processor to the data memory.

A related approach is the work on global snapshot algorithms for distributed systems, by among others Chandy and Lamport [20] and Miller and Choi [66]. With their algorithms, it is possible to obtain *consistent global states* in a distributed system by using in-band communication between the processes. Two conditions are subsequently stated in [57] for obtaining a consistent global state in those systems: (1) no message that is recorded as having been sent is every lost, and (2) all messages that are recorded as having been received are also recorded as having be sent. The first condition requires that no messages are lost during the process to capture the global system state. The second condition requires that no additional messages are created during this process. We have analyzed in detail in [95] the correspondence between their formalism of a distributed systems with the ones used in the CSAR debug approach (refer to Chaps. 2 and 3). We conclude from this analysis that the states that we extract from a GALS SOC using our CSAR infrastructure are indeed by their definition locally and globally consistent. This consistency is guaranteed by the use of the on-chip PSIs that ensure the unique, unambiguous localization of the messages that are exchanged between the on-chip building blocks on either the initiator or the target. A key difference between their methods and ours however is that we use an adjacent channel, the EDI, to communicate the state recording action. This EDI is as fast or faster than the communication interconnect used by them. This causes the feasible states obtained using the CSAR debug approach to be temporally closer to the actual state path of the system than with their in-band approach. A disadvantage of our approach is the silicon area cost associated with this EDI. We discuss this in Chap. 8.

9.1.3 DfD for Internal Observability and Controllability

Because of the limitations of the observability methods described in the previous sections, SOC design teams add debug support hardware [36, 41, 42, 52, 59, 64] to their SOC implementations at design time. This debug support is subsequently used

post-silicon, as explained in Sect. 1.3. We structure our discussion on the related work in this domain using the following categories: (1) debug event sources, (2) trace-based observability, (3) scan-based observability and controllability, (4) debug event distribution mechanisms, and (5) debug configuration mechanism. We discuss related DfD for SOC execution control in Sect. 9.2.3.

9.1.3.1 Debug Event Sources

SOC design teams add debug event sources to initiate a debug observability and/or controllability action inside a chip. A large variety of on-chip debug event sources have been proposed in the past, at different abstraction levels. At the higher abstraction levels, we see processor monitors [7, 68], property and protocol checkers [1, 17, 27, 53, 70, 79, 86], and communication monitors [29, 87, 100]. Near the physical abstraction level, we see temperature, voltage, and process monitors [30, 80, 81]. These monitors and checkers are often configurable to detect different conditions and can sometimes also derive new (meta)data from the observed SOC data, for example, by filtering or compressing this information [22, 23, 101]. These monitors increase the internal observability (debug requirement DR-1) and help overcome the I/O bandwidth limitations by their on-chip preprocessing of SOC data (complicating factor CF-1). These monitors do have a silicon area cost (complicating factor CF-3). Often they are (highly) configurable to allow their use not only for debugging, but also for performance analysis and optimization purposes. The outputs of these monitors and checkers are typically single-bit, flag signals that are distributed on-chip to start debug actions. The meta data can be kept local for extraction at a later point or communicated via the functional interconnect.

The CSAR debug approach relies on on-chip monitors to observe the communication between building blocks and to generate a debug event on the detection of a sequence of interest. In addition, the CSAR monitors calculate checksum values as an abstract representation for the observed communication sequence and to help detect and locate *transient errors*. The debug events that are generated by the CSAR monitors are distributed via a dedicated interconnect.

9.1.3.2 Trace-Based Debug Observability

Trace-based debug observability provides real-time internal observability. Real-time observability is important when trying to understand the sequence of internal events that caused a bug to be activated. Example trace-based debug solutions include Altera's SignalTap [5], Xilinx's ChipScope [99], ARM's CoreSight Trace [9], and MIPS/FS2's PDTrace [87] architectures. These architectures all support a non-intrusive, real-time trace debug approach (debug requirement DR-3). Trace data can be directly output on device pins using either a parallel or a high-speed serial interface. Due to pin and bandwidth constraints, modern SOCs may however also capture trace data in an internal trace buffer under control of debug events [2, 19].

The set of signals to capture and output can be hard-wired or configurable. This depends on the trade-off made by the SOC debug team, between the amount of internal observability required and the associated hardware cost. Typical hardware signals that are selected are (internally-generated) clock, reset, and handshake signals, and the state registers of deeply-embedded FSMs [2].

The increase in SOC complexity, and in particular the use of different clock and power domains, require these techniques to use multiple and larger trace buffers. Methods to compress debug trace data have been developed to address the consequences of this trend [6, 11, 25]. In addition, methods have been proposed that analyze the SOC implementation pre-silicon and identify a subset of signals which, after being captured on-chip, can be expanded using debug software and knowledge of the SOC implementation, to provide state information on more internal signals than were originally traced [12, 21, 54, 55, 62].

Re-use of the functional interconnect to transport debug trace data has also been proposed [91]. When tracing via the functional interconnect, the functional application may be impacted by this debug activity, which is not desirable (debug requirement DR-3). Sample on the Fly [52] is another real-time trace method used for CPUs. In this method, a debug engineer instructs on-chip debug functionality to copy the (sub)state of a processor periodically in a set of dedicated registers. The debug engineer can subsequently read out the contents of these registers without affecting the execution of the processor.

The key advantage of trace-based debug approaches is their ability to acquire data in real-time. Another advantage is that the state of the selected internal signals can be observed over a long time interval, helping to track down timing-relating problems between signals.

Trace-based debug however also has four disadvantages. First, only a small number of SOC pins can be reused and dedicated to debug while the SOC is used functionally. A second disadvantage is the restriction on the number of internal signals that can be observed in real-time. Third, trace-based methods do not provide controllability, which prevent these methods from supporting the localization of uncertain errors (complicating factor CF-8). Fourth, it is difficult to combine trace-based debug with a GALS SOC, because when debug trace data has to cross a clock domain boundary, it has to be synchronized to the clock of the receiving clock domain, which introduce a variable latency (complicating factor CF-7). This resynchronization may cause the timing information to be lost. Work has been performed on embedding timestamps in the debug trace data, however this additional information increases the debug data volume further.

The CSAR debug approach does not use trace-based debug. Instead we set ourselves a goal to provide 100 % internal observability and controllability (refer to Sect. 4.2). Whether trace-based debug is better than scan-based debug is open for debate, and depends to a large extend on the type of (unknown) error that has to be located. There is however nothing preventing an SOC design team to support both. These approaches have been successfully combined in a number of processors and SOCs, such as the PNX8525 [97], the CODEC SOC [96], and the En-II SOC [93] presented in Sect. 8.1. Combining trace-based and scan-based debug infrastructures

has the additional advantage of being able to share on-chip debug functionality, such as the debug event generators and the EDI, to yield a more efficient implementation (debug requirement DR-4 and complicating factor CF-3).

9.1.3.3 Scan-Based Debug Observability

Scan-based state dumping has been in use for large micro-processor chips since the mid 1990 s [36, 41, 52, 61]. In [61] a strategy is described to locate design errors in the SuperSPARC-II microprocessor, a single clock domain IC, by creating scan dumps. A clock controller design is presented in [36] that allows precise control over the processor clock in debug mode. The clock can be halted either by a program breakpoint, or through external control. Stopping the system at an interesting point in time allows us to extract a much larger volume of state data from the SOC for subsequent analysis than can be achieved through trace-based methods. The SOC needs to be stopped, which is intrusive and non-trivial for GALS SOCs (debug requirement DR-3 and complicating factor CF-9).

The CSAR debug approach extends the use of scan-based debug to GALS SOCs. We use this infrastructure to extract the entire SOC state after its execution has been stopped. We take special care by using the monitors, the PSIs, and EDI, to ensure that the extracted state is both locally and globally consistent. We demonstrated the CSAR debug functionality for seven SoCs in Sect. 8.1.

9.1.3.4 Debug Event Distribution Mechanisms

There are two options for GALS SoCs to globally distribute debug events: (1) re-using the existing functional communication interconnect, and (2) using a dedicated debug event interconnect. The authors of [91] reuse the functional communication interconnect, such as the on-chip buses or NoC, as an EDI. The advantage of this reuse is that the functionality to route events from a specific event source to a (subset of) event destinations already exists without significant additional hardware cost (debug requirement DR-4). A disadvantage of re-using the functional communication interconnect for debug is that the additional debug data volume has to be included as part of the functional requirements for the communication interconnect. Without a permanent reservation for this data volume on the interconnect, inserting the debug events at run-time may cause a debug event packet to take the place of a functional packet, thereby potentially causing the execution of the SOC to change, and violating debug requirement DR-3. This intrusiveness can be avoided by making a permanent bandwidth reservation, if the communication interconnect supports this functionality. However, permanently reserving this bandwidth may be expensive in terms of silicon area, when this bandwidth is subsequently only used during debugging and is not available during normal operation (complicating factor CF-3). The packetization and depacketization that may be needed to transport data across the functional communication interconnect also adds a delay to the delivery of debug events to their destinations (complicating factor CF-2).

A dedicated EDI [2, 9, 51, 64, 87, 88, 92] prevents this intrusiveness. An advantage of adding this additional hardware is that this allows an optimization of its design, improving the response time of our infrastructure to debug events, and making it shorter than in these related works (complicating factor CF-2). The ability to react quicker to debug events allows us to capture and extract state information that is temporally closer to the actual state of the system and improve the temporal resolution of our error localization. This may help find *transient errors* (complicating factor CF-6).

We have used these advantages in the design of our EDI. Our EDI is a multi-hop broadcast network, which makes it scalable and fast. Its architecture mirrors the architectures of modern SoCs, to allow it to approximate instantaneous debug event distribution, compared to the delays obtained when using the functional communication [10]. We use this EDI because it is the faster of the two event distribution techniques, it is scalable, and it re-uses the functional communication topology, while it does not affect the functional communication.

When functional communication data causes a debug communication event, then this event crosses the EDI as fast as or faster than the functional communication data crosses the functional communication interconnect [94]. We use this property to conditionally keep this data within the functional communication interconnect during debug. We can then step-by-step analyze the acceptance of this data and subsequent processing by the target module at different abstraction levels, using the PSI between the interconnect and the slave. Our dedicated EDI does have an associated silicon area cost (complicating factor CF-3).

9.1.3.5 Debug Configuration Mechanisms

The previously-mentioned debug functionality can be either controlled from a dedicated initiator on the functional interconnect, directly from a TAP, or both. Examples of the first are ARM's debug access port (DAP) [9] and FS2's multi-core embedded debug (MED) system [59]. Using the functional interconnect to access the state of modules does not allow the state of the functional interconnect to be debugged. Although less intrusive than reusing a functional port or a functional resource, these approaches are still more intrusive than a dedicated observation and control architecture.

The CSAR debug infrastructure provides a combination of both. Our TAP-DTL bridge reuses the functional interconnect, but consumes functional resources, e.g. bandwidth on the communication interconnect. The CSAR debug infrastructure furthermore modifies the TCB and TPR structures, standardized by the Philips/NXP core test action group (CTAG) and debug action group (DAG). Details on these structures have previously been published in [85]. We modified the basic TPR structure to use the functional clock of the building block in which the TPR is instantiated. We oversample the TCK clock signal, as opposed to using the TCK clock signal directly as the original implementation does. Using the functional clock signal to clock the logic in our TPRs simplifies their integration in an SOC, by not requiring that an additional TCK clock domain is created and verified (refer to Sect. 5.2.4).

9.2 Execution Control

In this section, we review related work that improves the debug control over the SOC execution. In Sect. 9.2.1, we review deterministic SOC architectures that do not exhibit the low-level non-determinism identified in Chap. 3. In Sect. 9.2.2, we discuss related work that tries to reproduce or *replay* the SOC execution, by first recording internal events that lead to a failure, and subsequently replaying these events in the same order, to try and reproduce the same SOC behavior and consequently the failure. In Sect. 9.2.3, we discuss DfD techniques to control the SOC execution.

9.2.1 Deterministic Architectures

The authors of [39] propose synchro-tokens to make the synchronizations between all parties in a GALS system deterministic. This makes the execution of this system also deterministic, as the resulting system will exhibit a single, global trace for every run, independently of whether it is being debugged or not. We call this trace the "deterministic trace" [94]. This trace is defined by the (software) synchronization points between the parties. A result of this determinism is that all errors become *certain*, as an error now either deterministically occurs during this deterministic trace or not. We refer this deterministic execution as *deterministic play* [94].

Synchronous design approaches and languages, such as TT-NOC and TT-SOC [74] and LUSTRE, ESTEREL, and SCADE [15, 16] result in systems that are deterministic by communicating and synchronizing the operation between distributed components at a very coarse time grain. This coarse grain allows ample time for resolution of any lower-level non-determinism when it occurs (complicating factor CF-7). These approaches use a 100 % static schedule, based on worst-case execution times of the tasks in the system. This schedule uses static interleaving of the requests at the inputs of the shared resources. In the end, there is always still a probability of meta-stability. It is possible to make the effect that this meta-stability has on the function of the system as small as possible, by increasing the size of the time grain allowed to resolve this meta-stability.

An advantage of these methods is that they address the problem of uncertain errors (complicating factor CF-8). Because the SOC executes deterministically, there is only one absolute order in the execution of the SOC. An error in the SOC implementation is either on the state path that the SOC deterministically follows, consequently activating and causing a failure to be observable, or an error is not on this state path, does not get activated, and does not cause a failure. Uncertain errors (complicating factor CF-8) can therefore not occur.

A general drawback of these methods is that they reduce the performance of the SOC by essentially statically scheduling the entire execution of the SOC. This performance is lost, because a shared resource may be left idle to ensure a deterministic execution, while under other circumstances, this resource could have been used to

service requests (refer to Sect. 3.2). The fact that these methods are not commonly-used on-chip or off-chip and are only used for safety-critical applications, shows that their costs are really only affordable in safety-critical applications, and are not suited for use in the consumer electronics domain (debug requirement DR-4).

These methods also do not solve the internal observability and execution control problems (debug requirements DR-1 and DR-2). Once a failure is observed on the outside of a deterministic SOC, still other debug observability and controllability methods, as described in Sect. 9.1, are needed to both spatially and temporally localize the root cause of this failure (complicating factors CF-1 and CF-2).

9.2.2 Deterministic Replay

The authors of [35, 60, 71, 84] describe systems to reproducibly replay parallel program executions by recording the interaction between different computation threads. The authors of [18, 33] aim for the same by focusing on the interactions using shared variables. The authors of [98] tackle the problem of debugging multiple processors using simulation with instruction-set simulators. The interconnect or silicon debug are not considered.

Deterministic replay [32, 83] and instant replay [60] methods have been proposed to try to cause an uncertain error to occur in more debug experiments. To achieve this, these methods prescribe that we first record all sources of non-determinism in the system that we are debugging during a debug experiment in which the uncertain error manifests itself. This recording has to take place at the abstraction level at which these sources cause the execution of the system to diverge from one debug experiment to another. During a subsequent replay operation, these sources need to be controlled to such a degree, that they behave as recorded and consequently recreate the circumstances under which the fault gets (re)activated and the failure reoccurs. This record-and-replay process greatly facilitates the root cause analysis of an uncertain error. These methods have been successfully applied to debug software systems [60]. It is possible to successfully apply these methods to software systems, as these systems typically have a small number of divergence points in their execution. The non-determinism in the execution that occurs at these points can therefore be more easily recorded and replayed by focusing on the synchronization between the internal processes. Fortunately, the synchronization frequency between these processes is small, which keeps the data volume to be recorded and replayed still manageable.

The authors of the pervasive debugging [37] method model the system to be debugging at a sufficient level of detail, such that its non-deterministic execution becomes deterministic again. It may be possible to model (source)-synchronous systems to this level of detail. We anticipate however that it is unfeasible to model GALS SoCs at this level of detail, as explained in Chap. 3. The authors of relative debugging [3] use an alternative version of the system to be debugged as a reference. This version is typically a sequential version of the actual system. Such an alternative system is affected by the same complicating factors as the pervasive debugging method.

A disadvantage of these methods is that for GALS SoCs, we need to record or model the non-determinism that is caused by the variable latency in the communication between the asynchronous clock domains. An SOC can easily contain tens to hundreds of building blocks, and these building blocks connect asynchronously to an on-chip communication interconnect [34]. The resulting data volume and rate to be recorded increases rapidly, when these building blocks and the on-chip communication interconnect run at higher clock frequencies. We have seen for trace-based debug methods in Sect. 9.1.3, that to record this data non-intrusively on-chip is expensive in terms of silicon area. It may also be either very expensive or impossible to stream this data non-intrusively off-chip. In addition, these solutions are often intrusive, in that they incur a performance overhead that may prevent the activation of an error (debug requirement DR-3).

The CSAR debug approach offers instead its guided (re)play process as a related method, to try to replay the events that led to the activation of an uncertain error and enable its localization. We focus on actively controlling the concurrency inside the GALS SOC, while these prior works focus on the post-hoc replay and analysis of parallel programs. As described in Sect. 4.2, we execute our guided replay process from the off-chip debugger software using the on-chip monitors, PSIs, and EDI. It therefore cannot be executed at speed and may therefore not be suited to reproduce certain, timing-critical errors (complicating factor CF-2).

9.2.3 DfD for Execution Control

A debug engineer traditionally observes the execution of the computation inside the building blocks to debug an SOC. We call this a *computation-centric debug approach*. These approaches allow a debug engineer to inspect the state of a building block when its computation has reached a certain point. These approaches have been studied extensively in the past and are offered commercially as configurable parts of a processor module [9, 59, 64]. The advantages of computation-centric debug approaches are that they allow very fine-grain execution control of individual building blocks and modules (complicating factor CF-2). Detailed knowledge of the functionality and implementation of a processor module also allows the debug support to be custom-made and the visualization to be done at an abstraction level familiar to the software programmers (debug requirement DR-4).

Using only a computation-centric debug approach does however have three drawbacks. First, it may not be trivial to generally extend and define the concept of computation breakpoints to other SOC modules, such as hardware accelerators and dedicated I/O peripherals. This resulting lack of debug support in these other modules reduces the spatial resolution that can be obtained during the debug process (debug requirement DR-1). We discussed this in Sect. 8.5, where we concluded that without direct controllability over the execution of the NoC or of the embedded memories, the resolution of our error localization would have been limited to the

entire path from the producer processor to the data memory. Second, a computation-centric debug approach does not directly allow debugging the communication that takes place on the communication interconnect between the building blocks. This interconnect is however a key, and nowadays, very active component in the overall SOC architecture. The interconnect may contain faults as well due to the increase in the complexity of the communication between the SOC modules (complicating factor CF-7).

The CSAR debug approach therefore complements the existing *computation-centric debug approach* with a *communication-centric debug approach* to effectively address the debugging of the complex communication and the on-chip communication interconnect in a modern SOC. The infrastructures required for these two debug approaches can be combined to make more efficient use of the required silicon area (debug requirement DR-4).

9.3 Debug Standardization

Well-defined industry standards enable the interoperability between SOC components produced by different internal and external parties. Standards provide a basis upon which either the producer or the consumer of an SOC module can rely, to ensure that their component will work together with conformant parts of others, when they are integrated in an SOC. A number of standardization activities in the area of debug and diagnosis bring module providers, semiconductor manufacturers, and vendors of debug tools together. In particular, the activities on the IEEE Std 1149.1 [48], on the IEEE Std 1149.7 [46], in the MIPI Debug Working Group [67], on the IEEE P1687 draft standard [47], on the IEEE-ISTO Nexus 5001 standard [50], and in the OCP-IP Debug working group [75], are relevant for the work in this book.

At the center of these standardization activities resides the IEEE Std 1149.1 TAP that is traditionally used for test and debug. The IEEE Std 1149.7 standardizes a successor to this TAP, enabling additional capabilities for present-day and future test and debug needs, while providing full backward compatibility. The MIPI Debug WG leverages the IEEE Std 1149.7 interface and adds high bandwidth, unidirectional trace interfaces for smart phones and other networked devices. Recently a project authorization request (PAR) was submitted for an IEEE P1149.10 draft standard to "define a high speed test access port for delivery of test data, a packet format for describing the test payload and a distribution architecture for converting the test data to/from on-chip test structures" [49]. This initiative complements the existing family of IEEE 1149 standards that have standardized test and debug access interfaces across the industry. The IEEE P1687 draft standard focuses on standardizing the access to on-chip instruments via the existing TAP, and is going into the IEEE ballot process at the time of writing this book. The IEEE-ISTO Nexus 5001 standard extends the TAP by defining a message-based interface between on-silicon instrumentation and debug tools for the automotive industry. The OCP-IP Debug WG focuses on standardizing the critical set of on-chip debug functions and how these functions interface to SOC-level debug and analysis tools.

The CSAR debug infrastructure described in this book reuses the IEEE 1149.1 TAP and extends on the concepts of the IEEE Std 1500 test wrapper [45] for scan-based silicon debug. The test wrapper presented in Sect. 5.7 is however different from the one described in IEEE Std 1500, because of the functional access requirements to the scan chains in the hybrid mode (refer to Fig. 4.3). This hybrid mode is not covered by the IEEE Std 1500 standard. The IEEE P1687 standard proposal was not mature enough yet to be used as a basis for our DCSI, presented in Sect. 5.2. We expect however that the CSAR test wrappers and DCSI can be adapted in future to be able to leverage industry support for the IEEE 1500 and P1687 standards, in terms of DfD and debugger tools.

9.4 Debug Tool Support

In this section, we review existing DfD tool support for including debug modules in an SOC implementation (Sect. 9.4.1), existing standardized APIs to interface with different SOC environments from one or multiple debugger tools (Sect. 9.4.2), and off-chip debugger tools (Sect. 9.4.3).

9.4.1 DfD Tool Support

The number of commercially-available tools to add debug support to an SOC implementations is very small for three reasons: (1) most companies do not automate the insertion of their debug modules and leave the insertion of debug support in the implementation of their SoCs as a manual task for their SOC design team, (2) they reuse the debug support that is provided to them by the provider of their functional SOC modules, or (3) they do automate the insertion but do so with mostly proprietary tools. One noted exception were the DAFCA ClearBlue tools [2].

There are tools that provide support for adding debug modules to an FPGA-based implementation of an SOC [5, 99]. The implementations of the debug modules that are inserted by these tools are however often optimized for the type of resources found in FPGAs, and not for the standard cell designs of industrial SoCs. Additionally, the uncertainty regarding the liability aspects of including these modules in an industrial SOC also tend to prevent their wider use. The CSAR DfD flow, described in Chap. 6 and Appendix A, automates the inclusion of the CSAR DfD modules in any GALS SOC implementation. We have described its implementation in detail, to allow it to be used as a starting point for others.

9.4.2 Debug Application Programmer's Interfaces

Debug engineers need to be able to combine the use of the best debugger tools available, because of the many possible combinations of components in their SoCs.

Each debugger tool can provide the appropriate abstraction level for the state and execution control of the corresponding component to its user. A user debugging the software application executing on an embedded processor is likely to be interested in observability and breakpoint control in relation to the application's source code, whereas a user debugging the configuration of the on-chip network might be more interested in transaction-level observability or single-stepping the communication on individual communication links at the message or transaction level. Providing each debugger tool with a dedicated debug link to the SOC implementation is however an inefficient solution, as it requires that more valuable device pins and silicon area are dedicated to debug and because it is difficult to synchronize the actions of the different debugger tools in this way.

Debugger tools from potentially different vendors need to be able to share the debug link to the same SOC environment and coordinate their actions. A good overview of the requirements for such a debug link and tool interface is provided in [63]. As with the on-chip debug infrastructure, true industry-wide consolidation still has to take place to standardize the definition of an open debugger interface. Candidates for an industry-wide multi-core debug API include Lauterbach's TRACE32 API [58], ARM RealView APIs [8], and the GNU gdb Remote Serial Protocol [31]. Another candidate is the multi-core debug (MCD) API, which was demonstrated to work with three different types of SOC environment: a simulation environment, an automotive electronic control unit (ECU), and an application board [63]. This API was developed as part of the European FP6-SPRINT project. This project also resulted in a retargetable debugger, which uses information from a standardized IP-XACT description to adapt to a new SOC [56].

The CSARDE uses its boundary-scan-level DfD configuration file in much the same way as this retargetable SPRINT debugger uses its IP-XACT description. The CSARDE also uses a fairly high-level debug API to abstract from the specifics of the SOC environments, the SOC implementation, and how to access the SOC implementation in the SOC environment. We anticipate that the information we store in our boundary-scan-level DfD configuration file can be easily incorporated in an IP-XACT description of the SOC. Additionally, we think that the requirements that we impose on an SOC environment with our IEnvironment interface can be fairly easily supported by the existing debug APIs. These integration steps are however left as future work. We come back to them in Sect. 10.2.

9.4.3 Debugger Tools

The available debugger tools split predominantly into tools that facilitate the debugging of software applications and those that facilitate the debugging of hardware implementations.

9.4.3.1 Software Debugger Tools

Many tools already exist that permit the monitoring and debugging of the computation inside an SOC, especially for a single processor. This is a very mature area. Most, if not all, processor vendors include a proprietary software debugger tool with their processor module. With this debugger, the debug engineer can set breakpoints and watchpoints in the source code that executes on the processor, inspect the content of local and remote memories and MMIO peripherals, and inspect stack traces. The efficient and effective debugging of multiple (programmable) modules on an SOC is however still a fairly new research field. Debugging the individual modules by themselves is insufficient for debugging a GALS SOC, because the complexity of a GALS SOC increasingly resides in the interactions between these modules [35, 65]. Therefore, the debugger tool has to consider the interaction between the computational threads in the SOC as well. Example debugger tools that support the debugging of parallel programs include the DDT debugger [4], the NVIDIA NSight debugger [73], and the TotalView debugger [82]. These debuggers support the debugging of parallel computations on a multi-processor system on chip (MPSOC). They however provide very little support to debug the communication inside such an MPSOC. A specialized debugger that does make the distinction between inter-process communication and intra-process computation is described in [14]. The CSARDE complements these computation-centric, parallel debuggers, by providing debug observability and controllability over the concurrent and pipelined communication that takes place over the on-chip communication interconnect.

9.4.3.2 Hardware Debugger Tools

Hardware description languages such as Verilog and VHDL define parallel hardware. Traditional hardware simulators however do not make a distinction between the communication between the modeled building blocks and the computation inside these building blocks. This is in particularly visible when debugging for example SystemC models, where the communication behavior between building blocks often has to be debugged at the level of the C source code that implements the higher-level communication protocol and data elements. These simulators furthermore do not allow the easy correlation of the execution of a building block at multiple levels of abstraction in parallel. It is also cumbersome to introduce debug data into these environments, because the abstraction level, at which this data was obtained, is often different from the level at which these simulators simulate the building blocks, therefore requiring that the support for these data format changes by semi-automated tools is created by the SOC design and/or verification and validation team.

Recently, simulators have started to offer support for transaction-level modeling and visualization [90]. EDA tools are introduced to back annotate the debug data, that was obtained from a silicon implementation of one or more building blocks using state-dump, real-time trace, or functional access techniques, to pre-silicon HDL source code [43, 89]. From there, more advanced data analysis algorithms can

now be used to try to localize the root cause of a failure. Debug tools from FPGA vendors, such as Altera's SignalTap [5] and Xilinx's ChipScope [99] tools do offer internal observability inside an FPGA. Their use for debugging GALS SoCs so far has been limited to their associated hardware not being included in silicon SOC implementations.

9.5 Debug Algorithms

Zeller [102] introduces the *delta debugging algorithm* to automatically find faults in software applications, that cause these software applications to fail for a certain input sequence. This algorithm uses a divide-and-conquer strategy to identify the minimal portion of the input sequence that causes this application to fail. In this algorithm, the application is run repeatedly. Each time the output of the software application is observed and classified as correct, faulty, or undetermined. Each time, the input sequence is modified to identify the essential input that triggers the failure. This work has been extended in [69], to take advantage of the hierarchical structure of the input sequence.

These algorithms resemble the temporal scoping used in the CSAR debug process, shown in Fig. 4.5. We also reduce and extend the temporal scope after each debug experiment, based on a comparison between the behaviors of the silicon SOC implementation and a reference implementation. These two approaches differ in the scope and abstraction level of their observation. Zeller's algorithms focuses only on the output of the software application and the application level, whereas we also use internal observability to localize the first occurrence of an error spatially, and perform the comparison between the silicon implementation and reference implementations at potentially, multiple abstraction levels (refer to Fig. 4.4).

The goal of *latch divergence analysis* [24] is also to automate the localization of the root cause of an error. This analysis involves the execution of the silicon implementation of a processor a number of times. This execution can for example be performed on an ATE. The authors repeatedly program the on-chip clock stop circuitry to stop the CPU clock at each clock cycle in its execution. They execute the processor under passing and under failing circumstances. They then group the obtained state dumps by circumstances and by clock cycle. They ascribe the fact that some flip-flops do not have the same value at the same clock cycle under the same circumstances to latch divergence noise. The values of these flip-flops are discarded in the remainder of the process. This process yields two filtered substates per clock cycle, one taken under passing circumstances and one taken under failing circumstances. The flip-flops that have a different value in these two filtered substates are considered to be on the error propagation path, so the combinational logic in front of the data input of these flip-flops are subsequently analyzed in more detail to try to find the root cause of the error. The clock cycle, at which the number of flip-flops with mismatching values in these two substates increases significantly, is typically temporally very close to the first activation of the associated fault. The authors successfully demonstrated the automation of their localization process on a number of processors.

A disadvantage of this method is however, that the filtering process does not distinguish between state deviations that are caused by errors and state deviations that are caused by the non-deterministic execution of a GALS SOC at the clock cycle level (complicating factor CF-7). When this approach is applied to a GALS SOC, it is therefore possible that this analysis incorrectly filters out state deviations that are caused by an error, causing the debug process to take longer than strictly necessary.

In contrast, we analyze the different causes for state deviations in this book, and try to address these causes directly, for example, by combining our guided replay process with appropriate structural, data, and behavioral abstraction techniques. We show in Chap. 8 that the state dumps obtained from different experiments show a smaller degree of *latch divergence* when these techniques are used.

Another debug algorithm is the *Bug Positioning System (BPS)* [26]. This work is related to the Bug Localization Graph (BLoG) [78] and instruction footprint recording and analysis (IFRA) [77] debug methods. The authors of [26] use signature-based analysis to identify a candidate group of internal signals, related to an error in the implementation. The signals that are compacted into these signatures are selected at design time. The authors provide little details on the selection criteria, except that they exclude data signals, leaving the control signals, and that these signals are high in the module hierarchy. Experiments are performed either once or multiple times, depending on whether sufficient I/O bandwidth is available to stream the signatures periodically off-chip or not (complication factors CF-1 and CF-2). The authors do include the effects of variable and random communication latencies in their experiments, but do not provide details on the relation between the internal clocks in their OpenSparc T2 system. It is therefore difficult to assess whether they have experienced the non-determinism at clock-cycle, handshake, and transaction level as we have identified in Chap. 3. The authors furthermore apply a "common mode rejection" filtering step, to "mitigate the noise present in large designs". This approach appears to be similar to the latch divergence analysis and would therefore also not address the root cause of (some of) this state noise, i.e., the non-deterministic execution and undefined substate in their SOC.

The CSAR debug approach tries to directly address the causes for non-determinism inside a GALS SOC, using both the guided replay process and the abstraction techniques. As shown in Sect. 8.4.2, these techniques are very effective in filtering out internal state noise, without unnecessarily losing important state information that can prove instrumental in locating the root cause of a failure.

References

1. Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *Proc. International Conference on Computer Aided Verification*, pages 538–542, London, UK, 2000. Springer-Verlag.
2. Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. Design Automation Conference*, pages 7–12, New York, NY, USA, 2006. Association for Computing Machinery, Inc.

3. D.A. Abramson and R. Sosic. Relative Debugging Using Multiple Program Versions. In *Proc. International Symposium on Languages for Intensional Programming*, 1995.
4. Allinea Softeware Inc. *Allinea DDT 4.1: High-impact debugging from novice to master*, 2013.
5. Altera Corporation. *Quartus II Handbook Version 13.0 - Design Debugging Using the SignalTap II Logic Analyzer*, 2013.
6. Ehab Anis and Nicola Nicolici. On Using Lossless Compression of Debug Data in Embedded Logic Analysis. In *Proc. IEEE International Test Conference*, 2007.
7. ARM Limited. *ARM920T (Rev1) Technical Reference Manual*, 2001.
8. ARM Limited. *ARM RealView ESL API v2.0 Developer's Guide*, 2007.
9. ARM Limited. *CoreSight SoC Technical Reference Manual Revision r2p0*, 2012.
10. Arnaldo Azevedo, Bart Vermeulen, and Kees Goossens. Architecture and design flow for a debug event distribution interconnect. In *Proc. International Conference on Computer Design*, pages 439–444, 2012.
11. K. Basu and P. Mishra. Efficient trace data compression using statistically selected dictionary. In *Proc. IEEE VLSI Test Symposium*, 2011.
12. Kanad Basu and Prabhat Mishra. RATS: Restoration-aware trace signal selection for post-silicon validation. *IEEE* Transactions on Very Large Scale Integration (VLSI) Systems, 2012.
13. C. Beddoe-Stephens. Semiconductor wafer probing. *Test and Measurement World Magazine*, pages 33–35, November 1982.
14. Michael Bedy, Steve Carr, Xianlong Huang, and Ching-Kuang Shene. A visualization system for multithreaded programming. In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pages 1–5, 2000.
15. Gérard Berry. Scade: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer Netherlands, 2007.
16. Gerard Berry. Synchronous design and verification of critical embedded systems using SCADE and esterel. *Lecture Notes in Computer Science*, 2008.
17. Marc Boule and Zeljko Zilic. *Generating Hardware Assertion Checkers For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer-Verlag, 2008.
18. R.H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, March 1991.
19. Cell Broadband Engine Debugging for Unknown Events. *IEEE* Design and Test of Computers, 24(5):486–493.
20. K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
21. Debapriya Chatterjee, Calvin McCarter, and Valeria Bertacco. Simulation-based signal selection for state restoration in silicon debug. In *Proc. International Conference on Computer-Aided Design*, 2011.
22. Călin Ciordaş. *M*onitoring-Aware Network-on-Chip Design. PhD thesis, Eindhoven University of Technology, 2008.
23. 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.
24. Peter Dahlgren, Paul Dickinson, and Ishwar Parulkar. Latch Divergency In Microprocessor Failure Analysis. In *Proc. IEEE International Test Conference*, pages 755–763, 2003.
25. Ehab Anis Daoud and Nicola Nicolici. Real-time lossless compression for silicon debug. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(9):1387–1400, 2009.
26. Andrew DeOrio, Daya Khudia, and Valeria Bertacco. Post-silicon bug diagnosis with inconsistent executions. In *Proc. International Conference on Computer-Aided Design*, 2011.
27. Andrew DeOrio, Jialin Li, and Valeria Bertacco. Bridging pre- and post-silicon debugging with BiPeD. In *Proc. International Conference on Computer-Aided Design*, 2012.

28. Travis M Eiles, GL Woods, and V Rao. Optical probing of vlsi ic's from the silicon backside. In *Proc. International Symposium for Testing and Failure Analysis*, pages 27–33, 1999.
29. Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. Mpsocs run-time monitoring through networks-on-chip. In *Proc. Design, Automation, and Test in Europe conference*, pages 558–561, 2009.
30. Robert L. Franch, Phillip Restle, James K. Norman, William V. Huott, Joshua Friedrich, R. Dixon, Steve Weitzel, K. van Goor, and G. Salem. On-chip timing uncertainty measurements on ibm microprocessors. In *Proc. IEEE International Test Conference*, 2007.
31. Bill Gatliff. Embedding with GNU: The gdb Remote Serial Protocol. *Embedded Systems Programming*, pages 108–113, 1999.
32. 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.
33. Aaron J. Goldberg and John L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE* Transactions on Parallel and Distributed Systems, 4(1):28–40, January 1993.
34. 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 Academic Publishers, 2004.
35. Siegfried Grabner, Dieter Kranzmüller, and Jens Volkert. Debugging of concurrent processes. In *Proc. International Parallel and Distributed Processing Symposium*, pages 547–554, January 1995.
36. Hong Hao and Rick Avra. Structured Design-for-Debug - The SuperSPARC^{TM} II Methodology and Implementation. In *Proc. IEEE International Test Conference*, pages 175–183, Washington, DC, USA, 1995. IEEE Computer Society Press.
37. Timothy L. Harris. Dependable software needs pervasive debugging. In *Proc. Workshop on ACM Special Interest Group on Operating Systems*, pages 38–43, New York, NY, USA, 2002. Association for Computing Machinery, Inc.
38. 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. International Symposium for Testing and Failure Analysis*, 1990.
39. Matthew W. Heath, Wayne P. Burleson, and Ian G. Harris. Synchro-tokens: A deterministic GALS methodology for chip-level debug and test. *IEEE* Transactions on Computers, 54(12):1532–1546, December 2005.
40. H. K. Heinrich, N. Pakdaman, D. Kent, and L. Cropp. Backside optical detection of internal gate delays in flip-chip mounted silicon vlsi circuits. In *Proc. Lasers and Electro-Optics Society Annual Meeting*, pages 559–560, 1990.
41. Kalon Holdbrook, Sunil Joshi, Samir Mitra, Joe Petolino, Renu Raman, and Michelle Wong. microSPARC: A Case Study of Scan-Based Debug. In *Proc. IEEE International Test Conference*, pages 70–75, Washington, DC, USA, 1994. IEEE Computer Society Press.
42. A.B.T. Hopkins and K.D. McDonald-Maier. Debug support for complex systems on-chip: A review. *IEE Proceedings Computers and Digital Techniques*, 153(4):197–207, July 2006.
43. Yu-Chin Hsu, Furshing Tsai, Wells Jong, and Ying-Tsai Chang. Visibility enhancement for silicon debug. In *Proc. Design Automation Conference*, 2006.
44. 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 International Test Conference*, page 883, Washington, DC, USA, 1999. IEEE Computer Society Press.
45. IEEE. *IEEE Standard for Embedded Core Test - IEEE Std 1500-2005*. IEEE Press, 2005.
46. IEEE. *IEEE Standard for Reduced- Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture-IEEE Std 1149.7-2009*. IEEE Press, 2010.

47. IEEE. IEEE Draft Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device-IEEE P1687, 2013.
48. IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture-IEEE Std 1149.1-2001*. IEEE Press, 2013.
49. IEEE. IEEE P1149.10 - High Speed Test Access Port and On-Chip Distribution Architecture Working Group, 2014.
50. IEEE-ISTO. IEEE-ISTO 5001-2012 - The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface.
51. IPextreme. *Multicore debug and performance optimization IP for complex SoCs*, 2008.
52. Don Douglas Josephson, Steve Poehhnan, and Vincent Govan. Debug methodology for the McKinley processor. In *Proc. IEEE International Test Conference*, pages 451–460, Washington, DC, USA, 2001. IEEE Computer Society Press.
53. Mohammad Reza Kakoee, M.H, Saeed Safari, and Zainalabedin Navabi. On-chip verification of nocs using assertion processors. *Proc. Euromicro Symposium on Digital System Design*, pages 535–538, 2007.
54. H.F. Ko and N. Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 2009.
55. H.F. Ko and N. Nicolici. Automated trace signals selection using the RTL descriptions. In *Proc. IEEE International Test Conference*, 2011.
56. Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based on ip-xact standards. In *Proc. Design, Automation, and Test in Europe conference*, 2008.
57. A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems. Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.
58. Lauterbach GmbH. *Lauterbach APIs*.
59. Rick Leatherman and Neal Stollon. An Embedded Debugging Architecture for SoCs. *IEEE* Potentials, 24(1):12–16, February 2005.
60. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.
61. M. Levittt, S. Nori, S. Narayanan, GP Grewal, L. Youngs, A. Jones, G. Billus, and S. Paramanandam., debuggability , and manufacturability of the ultrasparc-i microprocessor. In *Proc. IEEE International Test Conference*, 1995.
62. X. Lui and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proc. Design, Automation, and Test in Europe conference*, 2009.
63. Albrecht Mayer. A seamless tool access architecture from esl to end product. In *Inf. Proc. System, Software, SoC and Silicon Debug Conference*, 2009.
64. Albrecht Mayer, Harry Siebert, and Klaus D. McDonald-Maier. Boosting Debugging Support for Complex Systems on Chip. *Computer*, 40(4):76–81, 2007.
65. Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, 1989.
66. B.P. Miller and J.-D. Choi. Breakpoints and halting in distributed programs. In *Proc. International Conference on Distributed Computing Systems*, pages 316–323, June 1988.
67. MIPI Alliance Debug Working Group.
68. MIPS Technologies, http://www.mips.com. *EJTAG Specification, Document Number: MD00047, Revision 02.53*, 1 2001.
69. Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proc. International Conference on Software Engineering*, 2006.
70. José Augusto Miranda Nacif, Flávio Miana de Paula, Harry Foster, Claudionor José Nunes Coelho, Jr, and Antônio Otávio Fernandes. The chip is ready. am i done? on-chip verification using assertion processors. In *Proc. IFIP International Conference on Very Large Scale Integration*, 2003.

71. Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE* Micro, 26(1):100–109, 2006.
72. N. Nataraj, T. Lundquist, and Ketan Shah . Fault localization using time resolved photon emission and stil waveforms. In *Proc. IEEE International Test Conference*, volume 1, pages 254–263, September 2003.
73. NVIDIA Corporation. *NVIDIA Parallel Nsight - Power of GPU Computing Simplicity of Visual Studio*, 2011.
74. Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. The time-triggered system-on-a-chip architecture. In *Proc. IEEE International Symposium on Industrial Electronics*, pages 1941–1947. IEEE, 2008.
75. OCP International Partnership - Debug Working Group.
76. M. Paniccia, T. Eiles, V. R. M. Rao, and Wai Mun Yee. Novel optical probing technique for flip chip packaged microprocessors. In *Proc. IEEE International Test Conference*, pages 740–747, Washington, DC, USA, October 1998.
77. Sung-Boem Park, Ted Hong, and Subhasish Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 2009.
78. Sung-Boem Park, Anne Bracy, Hong Wang, and Subhasish Mitra. BLoG: post-silicon bug localization in processors using bug localization graphs. In *Proc. Design Automation Conference*, pages 368–373, New York, NY, USA, 2010. ACM.
79. M. Pellauer, M. Lis, D. Baltus, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proc. International conference on Formal Methods and Models for Co-Design*, 11–14 July 2005.
80. Rex Petersen, Pankaj Pant, Pablo Lopez, Aaron Barton, Jim Ignowski, and Doug Josephson. Voltage transient detection and induction for debug and test. In *Proc. IEEE International Test Conference*, 2009.
81. V. Petrescu, M. Pelgrom, H. Veendrick, P. Pavithran, and J. Wieling. A Signal Integrity Self Test (SIST) concept for the debug of nanometer CMOS ICs. In *Proc. International Solid State Circuits Conference*, pages 544–545, San Francisco, CA, USA, January 2006.
82. Rogue Wave Software, Inc. *TotalView Debugger*, 2013.
83. Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.
84. Smruti R. Sarangi, Brian Greskamp, and Josep Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *Proc. International Conference on Dependable Systems and Networks*, pages 301–312, Washington, DC, USA, 2006. IEEE Computer Society Press.
85. Geert Seuren and Tom Waayers. Extending the digital core-based test methodology to support mixed-signal. In *Proc. IEEE International Test Conference*, pages 281–289. IEEE, 2004.
86. F.C. Sica, Jr. Coelho, C.N., J.A.M. Nacif, H. Foster, and A.O. Fernandes. Exception handling in microprocessors using assertion libraries. In *Proc. Symposium Integrated Circuits and Systems Design*, 7–11 Sept. 2004.
87. N. Stollon. *On-Chip Instrumentation: Design and Debug for Systems on Chip*. Springer-Verlag, 2010.
88. Synopsys, Inc. *Identify - Simulator-like Visibility into Hardware Debug*, 2011.
89. Synopsys, Inc. *Verdi*^{3} *- Automated Debug System*, 2012.
90. B. Tabbara and K. Hashmi. Transaction-Level Modelling and Debug of SoCs. In *Proc. IP SOC Conference*, 2004.
91. Shan Tang and Qiang Xu. In-band cross-trigger event transmission for transaction-based debug. In *Proc. Design, Automation, and Test in Europe conference*, pages 414–419, New York, NY, USA, 2008. ACM.
92. Temento Systems. *DiaLite Platform*, 2007.
93. Bart Vermeulen and Sjaak Bakker. Debug architecture for the En-II system chip. *IET Computers & Digital Techniques*, 1(6):678–684, 11 2007.

94. Bart Vermeulen and Kees Goossens. Debugging Multi-Core Systems on Chip. In George Kornaros, editor, *Multi-Core Embedded Systems*, chapter 5, pages 153–198. CRC Press/Taylor & Francis Group, 2010.
95. Bart Vermeulen and Kees Goossens. Obtaining consistent global state dumps to interactively debug systems on chip with multiple clocks. In *Proc. High-Level Design Validation and Test Workshop*, 6 2010.
96. Bart Vermeulen and Hervé Vincent. DfD-Assisted System Test Analysis for Manufacturing Test Program Improvements. In *Informal Proc. IEEE Workshop on Silicon Debug and Diagnosis*, Ajaccio, Corsica, France, 5 2004.
97. B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and debug strategy of the pnx8525 nexperia™ digital video platform system chip. In *Proc. IEEE International Test Conference*, pages 121–130, 2001.
98. Andreas Wieferink, Tim Kogel, Rainer Leupers, Heinrich Eyr, Achim Nohl, and Andreas Hoffmann. A generic tool-set for SoC multiprocessor debugging and synchronization. In *Proc. International Conference on Application-Specific Systems, Architectures, and Processors*, pages 161–171, June 2003.
99. Xilinx, Inc. *ChipScope Pro Software and Cores—User Guide*, 2012.
100. Xilinx, Inc. *LogiCORE IP ChipScope AXI Monitor (v3.03.a)*, 4 2012.
101. Joon-Sung Yang and N.A. Touba. Enhancing silicon debug via periodic monitoring. In *Proc. International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 125–133, October 2008.
102. Andreas Zeller. *Why programs fail-a guide to systematic debugging*. Elsevier, 2006.

Chapter 10
Conclusion and Future Work

Abstract In this chapter, we summarize our contributions to the post-silicon SOC debug domain in Sect. 10.1. We furthermore discuss directions for future work in Sect. 10.2.

10.1 Conclusions

Consumer demands and cost benefits drive the semiconductor industry towards the integration of more and more functionality in an SOC. As the functionality and implementation of these SOCs grow in complexity and the product life cycles become shorter, SOC design teams have difficulty in guaranteeing that all errors in SOC software and/or hardware are detected and removed before a silicon SOC implementation has to be manufactured. Some SOC errors therefore only show up when validating the silicon SOC implementation. At present, the industry spends on average more than 50 % of the total project design time or cost on post-silicon validation and debugging. At this stage it is still very difficult and time consuming to debug failures, because we intrinsically lack the required internal observability in and execution control over these SOC for the proper localization of their root causes. The delays caused by these difficulties in turn lead to higher development cost, slipping deadlines, and a potential loss of customers.

In this book, we address the problem of debugging a faulty silicon SOC implementation. For this, we first captured the commonly-used post-silicon debug practices into one generic post-silicon debug process and derived four key debug requirements from this process: (1) internal observability, (2) execution control, (3) non-intrusive debug, and (4) efficient implementation and use. It is difficult to meet these requirements when we try to apply this process to the silicon implementation of a GALS SOC. We subsequently analyzed in detail what the reasons for these difficulties are. In this analysis, we identified nine factors that complicate the debugging of a silicon implementation of a GALS SOC.

In Chap. 4, we therefore proposed the CSAR debug approach. This approach combines four key aspects to address the identified debug requirements and complicating factors for a GALS SOC: (1) communication-centric debug, (2) scan-based debug, (3) abstraction-based debug, and (4) run/stop-based debug. This approach reuses many of the best debug practices in the semiconductor industry and is compatible with most other, alternative methods that we looked at in Chap. 9, in particularly with

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6_10,

computation-centric debug and trace-based debug. This reuse not with standing, we do contribute the following eight key innovations to the debug field:

1. We are the first to use on-chip PSIs to provide debug communication control. Other debug methods restrict themselves to controlling either the on-chip clocks or the computation inside the SOC modules themselves. With these PSIs, we can control the handshake signals that are part of most on-chip SOC communication protocols in use today.
2. We subsequently use this functionality to control the transfer of communication data elements, messages, and transactions across the on-chip communication interconnect, and can thereby create either a partial or a total transaction order during the execution of the SOC. The ability to (re)create a particular transaction order was shown to be helpful in Chap. 8 in the localization of, in particular, *uncertain errors*.
3. We also identified that stall states have to exist within the SOC building blocks to support modern SOC communication protocols and their handshake control signals. These stall states are important when trying to extract a locally-consistent state from a building block. We are guaranteed to extract a locally-consistent state by forcing the individual building blocks into (one of) their stall states by controlling the handshake control signals.
4. We use the combination of Contributions 2 and 3 to create a globally-consistent state in the system, by stalling the execution of all building blocks in their stall states.
5. We use a fast and scalable EDI, which connects on-chip monitors and PSIs. These monitors generate debug events when they detect a communication sequence of interest. The EDI communicates these events to the other monitors and PSIs. When an enabled PSI received a debug event, it stops the communication on the associated communication link at a pre-configured granularity. The concerted stopping of communication on the communication links between the SOC building blocks permits the extraction of globally-consistent states from the SOC. The speed of the EDI furthermore permits the extraction of states that are temporally closer to the actual state path of the SOC during its execution than can be obtained with other methods.
6. As far as we know, we are the first to show the applicability of scan-based silicon debug for GALS SOCs. For this, we take particular care to ensure not only the local consistency, but also the global consistency of the state we extract from the SOC using Contributions 2, 3 and 4.
7. We introduce the guided replay process for GALS SOCs. In this process, we replay the communication on the communication links between the SOC building blocks. Other methods either make the execution completely deterministic, record higher-level interface interactions and replay them, or use computation-centric execution control.
8. We propose the abstraction from the scan-chain contents of a silicon SOC implementation all the way up to the application level, supporting multiple, distributed

masters and multiple, distributed, shared slaves. We utilize structural, data, behavioral and temporal abstraction techniques in the process. Related methods do exist that perform limited back-annotation of simulation data to one or two abstraction levels, such as the RTL and the SystemC levels. These tools do not cover the range of abstraction levels that we do.

Our CSAR debug approach requires the implementation of on-chip debug support. We therefore provide a customizable, scalable, and layout-friendly, on-chip debug architecture to support our CSAR debug approach, and automate the inclusion of a customized CSAR debug architecture at design time in an SOC using a configurable DfD flow. This on-chip debug support is complemented by configurable, off-chip debugger software, called the CSARDE, to help a debug engineer control the on-chip debug functionality, and analyze and compare the state information extracted from the SOC with one or more of its references, in order to localize the root cause of an failure. We subsequently demonstrated the applicability of this work, by evaluating our CSAR debug approach using seven SOCs.

10.2 Future Work

The ITRS roadmap in Fig. 1.2 predicts that the SOCs that we use in consumer devices will continue to increase in complexity. These SOCs will contain (even) more programmable processors, a scalable communication infrastructure (e.g. a NoC), and a multitude of dedicated, hard-wired functions. Many software and hardware execution threads will be active simultaneously within these SOCs at run-time.

Our debug methods and tools will therefore have to cope with these more complex SOCs. More research still needs to be carried out to reach this goal. Specifically we see promising opportunities on the following topics:

- Raise the temporal abstraction further: The synchronization and communication between the threads executing on the SOC building blocks takes place at many levels of abstraction. In our current infrastructure, we provide control over this synchronization and communication from the clock level up to the level of transactions. We have however also seen that, e.g., C-HEAP tokens are communicated on a channel at a higher abstraction level. Each token may consist of multiple transactions. Therefore stopping at the transaction level with the current CSAR debug infrastructure does not guarantee that a C-HEAP token is either completely at the producer side or at the consumer side. It may actually not have been communicated yet, been partially communicated, or been fully communicated. This ambiguity can be resolved by raising the temporal abstraction level to C-HEAP tokens. Similarly, there may also be other abstraction levels in the implementation of the SOC hardware and software as well, that we do not use yet. It may therefore be possible to find more points in time, at each abstraction level, where it is safe to sample the state of the interacting building blocks. This ensures that the data that is sampled from these building blocks is locally consistent. Determining these

additional points however requires knowledge of the SOC implementation. More work needs to be done to investigate whether this knowledge can be leveraged to facilitate debugging.

- Complete and standardize the access to the SOC environments: As mentioned in Sects. 7.2.2 and 7.2.5, there are still two manual steps in the SOC environment abstraction and state access that should be automated. Furthermore, the CSARDE currently uses its own IEnvironment interface to interact in a standardized way with different SOC environments. As described in Sect. 9.4.2, this API is not unique. Many commercially-available and proprietary tools define their own interface to one or more SOC environments, at different abstraction levels, despite efforts that have been undertaken in the past to harmonize and standardize them. Perhaps when the physical access mechanism to the on-chip test and debug instruments is sufficiently standardized, for example through the IEEE P1687 draft standard, will it be possible to leverage an industry-wide API and focus all debug engineering resources on the localization of the root causes of failures, instead of on the infrastructure required to do so.
- Investigate the possible synergy between the traditional computation-centric debug approaches and the communication-centric debug approach presented in this book further: The CSAR debug approach is a communication-centric debug approach as it takes control over the on-chip communication in a GALS SOC. Traditional computation-centric debug approach take control over the on-chip communication. It will be interesting to investigate the combination of both approaches, as this combination is expected to provide full control over the SOC execution. This will improve the options to temporally locate transient and uncertain errors, in particular when complemented with the 100 % state observability provided by a scan-based debug approach.
- Investigate the possible synergy between trace-based and scan-based debug approaches further: The CSAR debug approach uses scan-based debug to provide 100 % state observability. As explained in Sect. 4.2, this makes the CSAR debug approach necessarily a run/stop-based debug approach and prevents it from providing information on internal signals for a longer period of time in the same debug experiment. Trace-based debug techniques provide the opposite, i.e., they provide information on internal signals, but because of bandwidth and silicon area constraints cannot do so for all signals in the SOC. It is a challenging problem to find out how to best combine the advantages of both approaches while staying within the I/O bandwidth and silicon area constraints. The first steps in this direction look promising, e.g., as described in [1].
- Bridge the gap between pre-silicon and post-silicon verification and validation methods and tools: There are a few examples of the reuse of pre-silicon verification and validation methods and tools, including the inclusion of assertions and property checkers in the silicon SOC implementation and their subsequent use during post-silicon debug [2], and the use of formal methods to help interpret the data extracted from the silicon SOC implementation [3–5]. The reuse of these methods and tools provide a structured means to reduce the effort and time to localize the root cause of a problem in the silicon SOC implementation. In addition,

the errors that are still only discovered by post-silicon verification and validation methods and tools should be studied to help identify and close possible gaps in the pre-silicon verification and validation methods and tools.
- Bridge the gap to failure analysis and physical and optical observability methods and tools: There is currently no agreed format to take the results from the CSARDE to these other tools. We currently also use the flip-flops in the SOC implementation as the leaves in our CSARDE data structure. This does not provide us observability below the abstraction level of individual gates. It may however be possible to refine the information used in the CSARDE to make it more useful for these other debug methods.

In general, there are numerous examples throughout the semiconductor industry where the inclusion and subsequent use of debug functionality have contributed significantly to a reduction in TTM. Most of them are however not documented in public literature. The lack of information on successful DfD use cases and the (perceived) ad-hoc nature of most debug processes causes most people in the industry to regard debugging more of an art than an exact science. It is consequently taking a lot of time for (research in) structured debug methods and tools to become widely recognized as key instruments in the reduction of TTM and improving the quality of the end product.

A trend is however emerging in the industry to try to change the debug process and make it become more of a science. Tool vendors are starting to provide support to the semiconductor manufacturers for the inclusion of commonly-accepted DfD features in their chips, and have debugger tools that can correlate silicon data with both design data and application source code. We can expected to see these tools become more widely adopted and demand for them increasing, as the semiconductor industry has to increase its debug productivity to address the debug challenges of next-generation SOC chips within a profitable TTM.

The complexity of these next-generation SOCs will require a holistic approach to SOC debugging, thereby integrating hardware and software debug infrastructures, leveraging pre-silicon verification and validation methods, tools, and insights, as well as extensions beyond current debug capabilities. Semiconductor manufacturers, tool vendors, and academia need to co-operate to find structured, scientific solutions to address the debug challenges of these next-generation SOCs. It is therefore very good to see the first results of their co-operations, particular on debug standardization, slowly emerging. We expect the post-silicon SOC debug domain to become an interesting area of increased research and development activity in the years to come.

References

1. K. Basu, P. Mishra, and P. Patra. Efficient combination of trace and scan signals for post silicon validation and debug. In *Proc. IEEE International Test Conference*, 2011.
2. Marc Boule and Zeljko Zilic. *Generating Hardware Assertion Checkers For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer-Verlag, 2008.

3. Flavio M De Paula, Alan J Hu, and Amir Nahir. nutab-backspace: rewriting to normalize non-determinism in post-silicon debug traces. In *Computer Aided Verification*, pages 513–531. Springer-Verlag, 2012.
4. R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.
5. Yu-Shen Yang, Andreas Veneris, and Nicola Nicolici. Automating data analysis and acquisition setup in a silicon debug environment. *IEEE* Transactions on Very Large Scale Integration (VLSI) Systems, 20(6):1118–1131, 2012.

Appendix A
Design-for-Debug Flow (Continued)

Abstract In this appendix, we continue the description of the tools in our DfD flow that was started in Chap. 6 with the debug wrapper generation tool. Specifically, we describe here the test wrapper generation tool in Sect. A.1, the top-level integration tool in Sect. A.2, the chip-level integration tool in Sect. A.3, and the boundary-scan-level integration tool in Sect. A.4. We conclude this appendix in Sect. A.5 with the DTD of the DfD configuration files that are exchanged between these tools and an overview of the parameters for each CSAR hardware module that have to be chosen at design time.

A.1 Test Wrapper Generation

A.1.1 Tool Overview

The test wrapper generation tool shown in Fig. 6.1 reads the *test wrapper configuration file* that is described in Sect. A.1.2. Based on this configuration, it reads the component definition of the debug wrapper and two configuration files with information on respectively the flip-flops in the logic scan chains and the debug support in the debug wrapper and its embedded module. It outputs the HDL files of the test wrapper and a *test wrapper DfD configuration file* with information on the test and on the debug support in the test wrapper and the wrapper module.

A.1.2 Test Wrapper Configuration

Listing A.1 shows an example test wrapper configuration file, which we use to generate the test wrapper for the NoC in our case study in Chap. 8. The <testWrapperGeneration> element indicates the start of the configuration data for the test wrapper generation process. The <name> element specifies the name of the test wrapper. The <componentFile> element specifies the HDL file that contains the component declaration of the module to wrap for manufacturing test. The <dfdFile> element specifies the DfD file of the debug wrapper. The <scanDefFile> element specifies the name of the logic scan configuration file that contains information about the position of the flip-flops in the scan chains inside the debug wrapper and the

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6,

module. The current flow supports the DEF [1] format that is output by the RTL Compiler tool of Cadence [2]. It is not difficult to add support other formats from other EDA tools. The <scanWidth> element specifies the number of scan chains to use.

The builder by default wraps all inputs and outputs of the specified module using the PIO unit inside the test wrapper (refer to Sect. 5.7). A user can selectively overrule this default behavior for specific signals using the <unwrapped> element. The inputs and outputs of the module that should not be wrapped are specified with a regular expression in the <signal> element. The use of this element in the example configuration file in Listing A.1 excludes all clock signals from the wrapping process. We leave the clock signals unwrapped as the CRGU that is added at the chip-level provides all the clock control we need to support the CSAR debug approach. All control and data inputs and outputs of the TPR and memory chains found in the DfD configuration file of the module are automatically left unwrapped as well.

The builder uses a PIO unit to by default tie off all wrapped signals to the value "0". This value is expanded to the appropriate bit width for multi-bit signals. The

Listing A.1 Example configuration file for test wrapper generation

```
1  <testWrapperConfiguration>
2    <name>rdt_csar_noc_test_wrapper</name>
3    < componentFile>../INTERFACE/rdt_csar_noc_debug_wrapper_cmp_pkg.p.vhdl</↩
4  componentFile>
5    <dfdFile>rdt_csar_noc_debug_wrapper.dfd</dfdFile>
6    <scanDefFile>../CONSTRAINTS/rdt_csar_noc_debug_wrapper_scan.def</scanDefFile↩
7  >
8    <scanWidth>8</scanWidth>
9    <unwrapped>
10     <signal>.*clk.*</signal>
11   </unwrapped>
12   <tieOffs>
13     <tieOff>
14       <signal>.*rst_n</signal>
15       <value>1</value>
16     </tieOff>
17   </tieOffs>
18   <clock>tile1_clk</clock>
19   <reset>tile1_rst_n</reset>
20   <outputDir>../../rdt_csar_noc_test_wrapper/</outputDir>
     <coreuse>true</coreuse>
</testWrapperConfiguration>
```

<tieOffs> element allows the user to overrule this default behavior for selected signals. The user specifies alternative tie-off values for a set of specific module signals using a <tieOff> element with subelements <signal> and <value>. The builder uses these alternative tie-off values when instantiating the PIO unit in the test wrapper. For the example configuration file in Listing A.1, we specify on lines 12 and 13 that all reset signals have to be deasserted when they are tied-off. We do this to prevent an accidental reset of the state of the module when the scan chains are shifted.

Lines 16 and 17 specify the clock and reset signals to use for the CSAR components inside the test wrapper using respectively the <clock> and <reset> element.

The <outputDir> and <coreuse> elements specify the output directory structure for the generated HDL files of the test wrapper.

Listing A.2 shows the complete DTD of the configuration file for the test wrapper generation process.

Listing A.2 DTD for the test wrapper configuration file

```
<!ELEMENT testWrapperConfiguration (name,componentFile,dfdFile?,scanDefFile?,↩
scanWidth,unwrapped?,tieOffs?,clock,reset,dtlPort?,outputDir?,coreuse?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT componentFile (#PCDATA)>
<!ELEMENT dfdFile (#PCDATA)>
<!ELEMENT scanDefFile (#PCDATA)>
<!ELEMENT scanWidth (#PCDATA)>
<!ELEMENT unwrapped (signal+)>
<!ELEMENT tieOffs (tieOff+)>
<!ELEMENT tieOff (signal,value)>
<!ELEMENT signal (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT clock (#PCDATA)>
<!ELEMENT reset (#PCDATA)>
<!ELEMENT dtlPort (addressWidth,dataWidth,maskWidth,blockSizeBits,nrEdis)>
<!ELEMENT addressWidth (#PCDATA)>
<!ELEMENT dataWidth (#PCDATA)>
<!ELEMENT maskWidth (#PCDATA)>
<!ELEMENT blockSizeBits (#PCDATA)>
<!ELEMENT nrEdis (#PCDATA)>
<!ELEMENT outputDir (#PCDATA)>
<!ELEMENT coreuse (#PCDATA)>
```

A.1.3 Test Wrapper Generation Process

Listing A.3 shows the `build` function of the TestWrapperBuilder class in pseudo code. This function generates a test wrapper from a tool configuration file.

Listing A.3 Pseudo-code for the test wrapper generation process

```
public class TestWrapperBuilder extends AbstractBuilder {
  ...
  public void build() {
    hdlFile = readHdlFile(configuration, logger);
    coreDfD = readCoreDfdConfiguration(toolConfiguration, logger);
    hdlCore = selectComponent(hdlFile, logger);
    ios = collectIOs(hdlCore, logger);
    ios.setUnwrapped(toolConfiguration.getUnwrappedList(), logger);
    ios.setTieOffValues(toolConfiguration.getTieOffsList(), logger);
    csTestWrapper = new CSARComponent(null, configuration.getName());
    new PIOUnit(csTestWrapper, toolConfiguration, ios);
    new ScanInputMux(csTestWrapper, toolConfiguration);
    csDebugWrapper = createDebugWrapper(csTestWrapper, hdlCore, coreDfD,
configuration, ios);
```

```
14      new DtlWrapperController(csTestWrapper, toolConfiguration);
15      new OutputRegister(csTestWrapper, toolConfiguration);
16      new ClockGate(csTestWrapper, toolConfiguration);
17      mapPrimaryInputsToPioUnit(csTestWrapper, ios);
18      mapPioUnitToPrimaryOutputs(csTestWrapper, ios);
19      makeMemoryChain(coreDfD, csTestWrapper);
20      testWrapperDfD = createDfDConfiguration(csDebugWrapper, logger);
21      scanDef = buildScanDef(toolConfiguration, logger);
22      update(testWrapperDfD, scanDef, ios);
23      hdlEntity = testWrapper.createEntity(toolConfiguration);
24      hdlComponent = new Component(hdlEntity);
25      hdlArchitecture = testWrapper.createArchitecture(hdlEntity);
26      hdlConfiguration = testWrapper.createConfiguration(hdlEntity, hdlArchitecture,
    toolConfiguration);
27      writeHdlEntity(hdlEntity, toolConfiguration, logger);
28      writeHdlComponent(hdlComponent, toolConfiguration, logger);
29      writeHdlArchitecture(hdlArchitecture, hdlEntity, csTestWrapper, toolConfiguration,
    logger);
30      writeHdlConfiguration(hdlConfiguration, toolConfiguration, logger);
31      writeDfDConfiguration(testWrapperDfD, toolConfiguration, logger);
32    }
33    ...
34  }
```

The builder reads the HDL component file on line 4 and the optional DfD configuration file on line 5. It creates an empty core DfD configuration when no DfD configuration file was specified.

The `selectComponent` function on line 6 works in the same way as it did for the debug wrapper generation tool (refer to Sect. 6.4.3). The data structure `ios` on line 7 collects the I/Os of the selected component. This data structure stores information about each input and output signal, including its bit width, whether it is wrapped or not, and its tie-off value when wrapped. All signals on the module are by default wrapped and tied-off to "0". The builder overrules this default behavior on lines 8 and 9 for the signals specified with the <unwrapped> and <tieOffs> elements in the tool configuration file.

A test wrapper object is created in memory on line 10. The builder instantiates an appropriate PIO unit in this test wrapper on line 11. The implementation of this unit depends on the signals of the module that are wrapped, as specified in the `ios` data structure. Next, the scan input multiplexer is added to the implementation of the test wrapper on line 12. Its implementation depends on the number of scan chains that is specified with the <scanWidth> element in the tool configuration. Third, the module itself is instantiated in the test wrapper on line 13. The implementation of the instantiated module depends on the selected HDL component `hdlModule`. All inputs and outputs of the component are copied to this entity declaration. The control signals for the test wrapper, described in Sect. 5.7, are added to the I/O interface of the test wrapper to allow control over the test wrapper from the chip-level TCB. The builder finishes the implementation of the test wrapper by instantiating the DTL wrapper controller, the output register, and the clock gate on lines 14–16. The implementation of all three components depend on the scan width, the clock, and the reset signals specified in the tool configuration.

On line 17, the builder maps the primary inputs of the test wrapper to the generic primary inputs on the PIO unit, while on line 18, it maps the generic primary outputs of this unit to the primary outputs of the test wrapper. The builder subsequently creates a memory chain through the test wrapper and the module on line 19.

The builder creates a DfD configuration for the test wrapper on line 20. This configuration needs to be extended with the information on the logic scan chains inside the module. This information is optionally read on line 21 using the name of the logic scan chain configuration file in the tool configuration. The test wrapper DfD configuration is subsequently updated with this information on line 22.

The HDL objects for the test wrapper are created on lines 23 to 26 and written to files on lines 27–30. The optional values of the <outputDir> and <coreuse> elements in the configuration file are taken into account during these write operations. The resulting DfD configuration for the test wrapper is written to a new DfD configuration file on line 31.

A.1.4 Executing the Test Wrapper Generation Process

Listing A.4 shows an example execution run of the test wrapper generation tool, in this case for the test wrapper of the NoC in the CSAR SOC.

Listing A.4 Command line example of the test wrapper generation for a NoC

```
1  TestWrapperGenerator v1.0
2
3  * Reading '/home/csar/book/vhdl/rdt_csar_lib/rdt_csar_noc_debug_wrapper/CSARDE/↩
   rdt_csar_noc_debug_wrapper.twc'.
4  * Reading 'rdt_csar_noc_debug_wrapper.dfd'.
5  * Reading '../INTERFACE/rdt_csar_noc_debug_wrapper_cmp_pkg.p.vhdl'.
6  * Component 'rdt_csar_noc_debug_wrapper' selected.
7  * Generating test wrapper data structures.
8    – Leaving 'tile1_clk' unwrapped.
9    – Leaving 'tile2_clk' unwrapped.
10   – Leaving 'host_clk' unwrapped.
11   – Leaving 'noc_clk' unwrapped.
12   – Leaving 'ctrlmem_clk' unwrapped.
13   – Leaving 'datamem_clk' unwrapped.
14   – Leaving 'video_clk' unwrapped.
15   – Leaving 'mem_se' unwrapped.
16   – Leaving 'mem_si' unwrapped.
17   – Leaving 'mem_so' unwrapped.
18   – Leaving 'mem_test_en' unwrapped.
19   – Leaving 'se' unwrapped.
20   – Leaving 'test_ctrl_sel' unwrapped.
21   – Leaving 'tpr_capture' unwrapped.
22   – Leaving 'tpr_shift' unwrapped.
23   – Leaving 'tpr_tck' unwrapped.
24   – Leaving 'tpr_tdi' unwrapped.
25   – Leaving 'tpr_tdo' unwrapped.
```

```
26     - Leaving 'tpr_update' unwrapped.
27     - Leaving 'si' unwrapped.
28     - Leaving 'so' unwrapped.
29     - Setting 'tile1_rst_n' tie-off value to 1
30     - Setting 'tile2_rst_n' tie-off value to 1
31     - Setting 'host_rst_n' tie-off value to 1
32     - Setting 'noc_rst_n' tie-off value to 1
33   * Instantiating the PIO unit.
34   * Instantiating the scan input multiplexer.
35   * Instantiating the debug wrapper.
36   * Instantiating the DTL controller.
37   * Instantiating the output register.
38   * Instantiating the local clock gate.
39   * Reading '../CONSTRAINTS/rdt_csar_noc_debug_wrapper_scan.def'.
40   * Reading 'rdt_csar_noc_debug_wrapper.dfd'.
41   * Writing '../../rdt_csar_noc_test_wrapper/INTERFACE/rdt_csar_noc_test_wrapper.e.vhdl'.
42   * Writing '../../rdt_csar_noc_test_wrapper/INTERFACE/rdt_csar_noc_test_wrapper_cmp_pkg.↩
     p.vhdl'.
43   * Writing '../../rdt_csar_noc_test_wrapper/RTL/rdt_csar_noc_test_wrapper_rtl.a.vhdl'.
44   * Writing '../../rdt_csar_noc_test_wrapper/RTL/rdt_csar_noc_test_wrapper_rtl_cfg.c.vhdl'.
45   * Writing '../../rdt_csar_noc_test_wrapper/CSARDE/rdt_csar_noc_test_wrapper.dfd'.
```

Note how the signals of the module that need to be left unwrapped, as specified on lines 7–9 in Listing A.1, are identified on lines 8–28. The tie-off values for the reset signals are overruled on lines 29–32. The components inside the test wrapper are subsequently created and instantiated on lines 33–38. The logic scan chain configuration file is read on line 39, while the DfD configuration file of the module is read on line 40. The HDL files and the DfD configuration file for the test wrapper are written on lines 41–45.

A.2 Top-Level Integration

A.2.1 Tool Overview

The top-level integration tool instantiates and connects the wrapped modules and the communication interconnect in a top-level module (refer to Fig. 4.1). The test and debug data interfaces of all modules are concatenated to form top-level scan chains. The order in which these modules are concatenated is important information for the debugger software, as it determines where each flip-flop and TPR bit is located with respect to the input and output of these chains. We need a tool to automate this task and make sure this location information is correct by design (refer to Fig. 6.1). This tool reads the *top-level configuration file* that is described in Sect. A.2.2. Based on this tool configuration, it reads the component HDL file of the test wrappers to instantiate in the top-level. It also reads the DfD configuration information of these test wrappers. It outputs the HDL files of the *top-level module* and a *top-level DfD configuration file*, which consolidates the information of all the integrated test wrappers, together with the order of their concatenated scan chains.

A.2.2 Top-Level Configuration

Listing A.5 shows an example top-level configuration file, which we use to generate the top-level module for the CSAR SOC in our case study in Chap. 8.

Listing A.5 Example configuration file for top-level integration

```
<topLevelConfiguration>
  <name>rdt_csar_top_vga</name>
  <nrChains>8</nrChains>
  <testWrapper>
    <type>producer</type>
    <dir>../../../rdt_dlx_lib/rdt_dlx_tile_producer_test_wrapper</dir>
    <componentFile>INTERFACE/rdt_dlx_tile_producer_test_wrapper_cmp_pkg.p.vhdl↩
</componentFile>
    <dfdFile>CSARDE/rdt_dlx_tile_producer_test_wrapper.dfd</dfdFile>
  </testWrapper>
  <testWrapper>
    <type>consumer</type>
    <dir>../../../rdt_dlx_lib/rdt_dlx_tile_consumer_test_wrapper</dir>
    <componentFile>INTERFACE/rdt_dlx_tile_consumer_test_wrapper_cmp_pkg.p.vhdl
</componentFile>
    <dfdFile>CSARDE/rdt_dlx_tile_consumer_test_wrapper.dfd</dfdFile>
  </testWrapper>
  <testWrapper>
    <type>host</type>
    <dir>../../../rdt_dlx_lib/rdt_dlx_tile_host_test_wrapper</dir>
    <componentFile>INTERFACE/rdt_dlx_tile_host_test_wrapper_cmp_pkg.p.vhdl</↩
componentFile>
    <dfdFile>CSARDE/rdt_dlx_tile_host_test_wrapper.dfd</dfdFile>
  </testWrapper>
  <testWrapper>
    <type>sram</type>
    <dir>../../../rdt_dtl_lib/rdt_dtl_sram_test_wrapper</dir>
    <componentFile>INTERFACE/rdt_dtl_sram_test_wrapper_cmp_pkg.p.vhdl</↩
componentFile>
    <dfdFile>CSARDE/rdt_dtl_sram_test_wrapper.dfd</dfdFile>
  </testWrapper>
  <testWrapper>
    <type>noc</type>
    <dir>../../../rdt_csar_lib/rdt_csar_noc_test_wrapper</dir >
    <componentFile>INTERFACE/rdt_csar_noc_test_wrapper_cmp_pkg.p.vhdl</↩
componentFile>
    <dfdFile>CSARDE/rdt_csar_noc_test_wrapper.dfd</dfdFile>
  </testWrapper>
  <testWrapper>
    <type>video_subsystem_vga</type>
    <dir>../../../rdt_video_lib/rdt_video_subsystem_vga_test_wrapper </dir >
    <componentFile>INTERFACE/rdt_video_subsystem_vga_test_wrapper_cmp_pkg.p.↩
vhdl</componentFile>
    <dfdFile>CSARDE/rdt_video_subsystem_vga_test_wrapper.dfd </dfdFile>
  </testWrapper>
  <testWrapper>
```

```
41      <type>tap_bridge</type>
42      <dir>../../../rdt_debug_lib/rdt_tap_bridge_test_wrapper</dir>
43      <componentFile>INTERFACE/rdttap_bridge_test_wrapper_cmp_pkg.p.vhdl</componentFile>
44      <dfdFile>CSARDE/rdt_tap_bridge_test_wrapper.dfd</dfdFile>
45    </testWrapper>
46    <outputDir>../</outputDir>
47    <coreuse>true</coreuse>
48  </topLevelConfiguration>
```

The <topLevelConfiguration> element indicates the start of the configuration data for the top-level integration process. The <name> element specifies the name of the top-level module. The <nrChains> specifies the number of parallel scan chains in the test wrappers. The list of <testWrapper> elements that follows, establishes a relation between the building types used in the specification of the communication interconnect (refer to Sect. 6.3) and the HDL component and DfD configuration files of the test wrapper of that module. The <outputDir> and <coreuse> elements control the generation of the output files, in the same way as they did for the debug and test wrapper generation tools.

Listing A.6 shows the complete DTD of the configuration file for the top-level integration process.

Listing A.6 DTD for the top-level configuration file

```
1  <!ELEMENT topLevelConfiguration (name,nrChains,testWrapper+,outputDir?,coreuse?)>
2  <!ELEMENT name (#PCDATA)>
3  <!ELEMENT nrChains (#PCDATA)>
4  <!ELEMENT testWrapper (type,dir,componentFile,dfdFile)>
5  <!ELEMENT type (#PCDATA)>
6  <!ELEMENT dir (#PCDATA)>
7  <!ELEMENT componentFile (#PCDATA)>
8  <!ELEMENT dfdFile (#PCDATA)>
9  <!ELEMENT outputDir (#PCDATA)>
10 <!ELEMENT coreuse (#PCDATA)>
```

A.2.3 Top-Level Integration Process

Listing A.7 shows the `build` function of the TopLevelBuilder class in pseudo code. This function generates the top-level module based on the tool configuration file.

Listing A.7 Pseudo-code for the top-level integration process

```
1  public class TopLevelBuilder extends AbstractBuilder {
2     ...
3     public void build() {
4        readTestWrapperData(hdlComponents, testWrapperDfDs);
5        csTopLevel = new CSARComponent(null, toolConfiguration.getName());
6        nocTestWrapper = findTestWrapperWithCommunicationArchitecture(↩
   testWrapperDfDs);
7        hdlComponent = hdlComponents.get(nocTestWrapper.getType());
8        instantiateNOC(csTopLevel, hdlComponent);
9        nocDfD = testWrapperDfDs.get(nocTestWrapper.getType());
10       nocArchitectureSpec = nocDfD.getDebugWrapper().getIpBlock().getArchitecture();
11       addIPs(csTopLevel, nocArchitectureSpec, hdlComponents);
12       createChains(csTopLevel);
13       hdlEntity = csTopLevel.createEntity(toolConfiguration);
14       hdlComponent = new Component(hdlEntity);
15       hdlArchitecture = csTopLevel.createArchitecture(hdlEntity);
16       hdlConfiguration = csTopLevel.createConfiguration(hdlEntity, hdlArchitecture, ↩
   toolConfiguration);
17       topLevelDfD = createTopLevelDfD(csTopLevel, testWrapperDfDs);
18       writeHdlEntity(hdlEntity, toolConfiguration, logger);
19       writeHdlComponent(hdlComponent, toolConfiguration, logger);
20       writeHdlArchitecture(hdlArchitecture, hdlEntity, csTopLevel, toolConfiguration, ↩
   logger);
21       writeHdlConfiguration(hdlConfiguration, toolConfiguration, logger);
22       writeDfDConfiguration(topLevelDfD, toolConfiguration, logger);
23    }
24    ...
25 }
```

The builder reads the HDL component and DfD configuration files of all test wrappers specified in the tool configuration file on line 4. This information is stored by name in two associative arrays, `hdlComponents` and `testWrapperDfDs`.

On line 5, the builder creates a CSARComponent object for the top-level module in memory. It examines on line 6 the DfD configuration of each test wrapper to find the test wrapper that contains the communication interconnect. The current implementation only supports the use of a NoC, which is generated with the flow described in Sect. 6.3. One can however use the same concept to support other communication interconnects. When the NoC is found, its HDL component definition is obtained on line 7 and used on line 8 to instantiate a CSARComponent object for the NoC in the top-level module. The information on this communication interconnect is also used on lines 9–11 to connect the other modules to the communication interconnect. This action is performed by first looking up the type identifier of each module, finding the associated HDL component, and adding a CSARComponent object for this module in the top-level module. Connections between the instantiated component and the NoC are made by matching their signal names. The set of signals that cannot be matched becomes the I/O interface of the top-level module. The scan chains inside each test wrapper are concatenated in the top-level module on line 12. The same happens for the TPR chain.

The HDL objects for the top-level module are created on lines 13–16. A DfD configuration for the top-level module is created on line 17 and includes the DfD configuration of all instantiated test wrappers and the top-level test and debug connectivity between them. The generated top-level module is written to HDL files on lines 18–21. The top-level DfD configuration file is written on line 22.

A.2.4 Executing the Top-Level Integration Process

Listing A.8 shows an example execution run of the top-level integration tool.

Listing A.8 Command line example of the top-level integration process

```
1  Top−level integrator v1.0
2
3  * Reading '/home/csar/book/vhdl/rdt_csar_lib/rdt_csar_top_vga/CSARDE/rdt_csar_top_vga.↩
   tlc'.
4  * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_producer_test_wrapper/INTERFACE/↩
   rdt_dlx_tile_producer_test_wrapper_cmp_pkg.p.vhdl'.
5  * Component 'rdt_dlx_tile_producer_test_wrapper' selected.
6  * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_producer_test_wrapper/CSARDE/↩
   rdt_dlx_tile_producer_test_wrapper.dfd'.
7  * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_consumer_test_wrapper/INTERFACE/↩
   rdt_dlx_tile_consumer_test_wrapper_cmp_pkg.p.vhdl'.
8  * Component 'rdt_dlx_tile_consumer_test_wrapper' selected.
9  * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_consumer_test_wrapper/CSARDE/↩
   rdt_dlx_tile_consumer_test_wrapper.dfd'.
10 * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_host_test_wrapper/INTERFACE/↩
   rdt_dlx_tile_host_test_wrapper_cmp_pkg.p.vhdl'.
11 * Component 'rdt_dlx_tile_host_test_wrapper' selected.
12 * Reading '../../../rdt_dlx_lib/rdt_dlx_tile_host_test_wrapper/CSARDE/↩
   rdt_dlx_tile_host_test_wrapper.dfd'.
13 * Reading '../../../rdt_dtl_lib/rdt_dtl_sram_test_wrapper/INTERFACE/↩
   rdt_dtl_sram_test_wrapper_cmp_pkg.p.vhdl'.
14 * Component 'rdt_dtl_sram_test_wrapper' selected.
15 * Reading '../../../rdt_dtl_lib/rdt_dtl_sram_test_wrapper/CSARDE/rdt_dtl_sram_test_wrapper.↩
   dfd'.
16 * Reading '../../../rdt_csar_lib/rdt_csar_noc_test_wrapper/INTERFACE/↩
   rdt_csar_noc_test_wrapper_cmp_pkg.p.vhdl'.
17 * Component 'rdt_csar_noc_test_wrapper' selected.
18 * Reading '../../../rdt_csar_lib/rdt_csar_noc_test_wrapper/CSARDE/↩
   rdt_csar_noc_test_wrapper.dfd'.
19 * Reading '../../../rdt_video_lib/rdt_video_subsystem_vga_test_wrapper/INTERFACE/↩
   rdt_video_subsystem_vga_test_wrapper_cmp_pkg.p.vhdl'.
20 * Component 'rdt_video_subsystem_vga_test_wrapper' selected.
21 * Reading '../../../rdt_video_lib/rdt_video_subsystem_vga_test_wrapper/CSARDE/↩
   rdt_video_subsystem_vga_test_wrapper.dfd'.
22 * Reading '../../../rdt_debug_lib/rdt_tap_bridge_test_wrapper/INTERFACE/↩
   rdt_tap_bridge_test_wrapper_cmp_pkg.p.vhdl'.
23 * Component 'rdt_tap_bridge_test_wrapper' selected.
24 * Reading '../../../rdt_debug_lib/rdt_tap_bridge_test_wrapper/CSARDE/↩
   rdt_tap_bridge_test_wrapper.dfd'.
```

```
25    Mapping 'tile1' to 'rdt_dlx_tile_producer_test_wrapper'.
26    Mapping 'tile2' to 'rdt_dlx_tile_consumer_test_wrapper'.
27    Mapping 'ctrlmem' to 'rdt_dtl_sram_test_wrapper'.
28    Mapping 'datamem' to 'rdt_dtl_sram_test_wrapper'.
29    Mapping 'tap' to 'rdt_tap_bridge_test_wrapper'.
30    Mapping 'video' to 'rdt_video_subsystem_vga_test_wrapper'.
31    Mapping 'host' to 'rdt_dlx_tile_host_test_wrapper'.
32  * Writing '../INTERFACE/rdt_csar_top_vga.e.vhdl'.
33  * Writing '../INTERFACE/rdt_csar_top_vga_cmp_pkg.p.vhdl'.
34  * Writing '../RTL/rdt_csar_top_vga_rtl.a.vhdl'.
35  * Writing '../RTL/rdt_csar_top_vga_rtl_cfg.c.vhdl'.
36  * Writing '../CSARDE/rdt_csar_top_vga.dfd'.
```

Note how the HDL and DfD information of the individual test wrappers is all read on lines 4–24. The top-level module is subsequently created from lines 25–31 and written to files from line 32–35. Its DfD configuration file is written on line 36.

A.3 Chip-Level Integration

A.3.1 Tool Overview

The chip-level integration tool instantiates and connects the top-level and the global TCB and CRGU in a chip-level module (refer to Fig. 4.2. This tool reads the *chip-level configuration file* that is described in Sect. A.3.2. Based on this tool configuration, it reads the component HDL and DfD configuration files of the top-level module. It outputs the HDL files of the *chip-level module* and a *chip-level DfD configuration file*.

A.3.2 Chip-Level Configuration

Listing A.9 shows an example chip-level specification file, which we use to generate the chip-level module of the CSAR SOC in our case study in Chap. 8.

Listing A.9 Example configuration file for chip-level integration

```
1  <chipLevelConfiguration>
2    <name>rdt_csar_chip_vga</name>
3    <componentFile>../INTERFACE/rdt_csar_top_vga_cmp_pkg.p.vhdl</componentFile>
4    <dfdFile>rdt_csar_top_vga.dfd</dfdFile>
5    <externalClock>
6      <name>ext_clk</name>
7      <period>10</period>
8    </externalClock>
9    <clock>
10     <name>tile1_clk</name>
11     <reset>tile1_rst_n</reset>
```

```
    <period>110</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>tile2_clk</name>
    <reset>tile2_rst_n</reset>
    <period>110</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>host_clk</name>
    <reset>host_rst_n</reset>
    <period>20</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>noc_clk</name>
    <reset>noc_rst_n</reset>
    <period>20</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>datamem_clk</name>
    <reset>datamem_rst_n</reset>
    <period>110</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>ctrlmem_clk</name>
    <reset>ctrlmem_rst_n</reset>
    <period>110</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>tap_clk</name>
    <reset>tap_rst_n</reset>
    <period>20</period>
    <pll>1</pll>
  </clock>
  <clock>
    <name>video_clk</name>
    <reset>video_rst_n</reset>
    <period>40</period>
    <pll>2</pll>
  </clock>
  <clock>
    <name>video_out_clk</name>
    <period>40</period>
    <pll>2</pll>
  </clock>
  <ccsDepth>2</ccsDepth>
  <rguDepth>20</rguDepth>
  <outputDir>../../rdt_csar_chip_vga/</outputDir>
  <coreuse>true</coreuse>
</chipLevelConfiguration>
```

The <chipLevelConfiguration> element indicates the start of the configuration data for the chip-level integration process. The <name> element specifies the name of the chip-level module. The <componentFile> and <dfdFile> elements specify respectively the HDL component and the DfD configuration files of the top-level module.

The <externalClock> element specifies the chip-level input signal for the external clock source and its clock period in nanoseconds. This signal is connected to the external clock input of the CRGU instantiated in the chip-level module (refer to Sect. 5.8). The list of <clock> elements specify the clock signals of the top-level module, their associated reset signal, their clock periods, and the PLL inside the CRGU that generates this clock. In the example configuration file in Listing A.9, all but the last two clock signals of the top-level module are generated using a first PLL in the CRGU. A second PLL generates the clock signals for the video subsystem.

The depth of the synchronizer inside each CCS of the CRGU (refer to Fig. 5.31 on p. 137) is specified using the <ccsDepth> element. The depth of the synchronizer inside each RGU (refer to Fig. 5.31 on p. 137) is specified using the <rguDepth> element. The <outputDir> and <coreuse> elements control the generation of the output files. Listing A.10 shows the complete DTD of the configuration file for the chip-level integration process.

Listing A.10 DTD for the chip-level configuration file

```
1  <!ELEMENT chipLevelConfiguration (name,componentFile,dfdFile,externalClock,clock↩
   +,ccsDepth,rguDepth,outputDir?,coreuse?)>
2  <!ELEMENT name (#PCDATA)>
3  <!ELEMENT componentFile (#PCDATA)>
4  <!ELEMENT dfdFile (#PCDATA)>
5  <!ELEMENT externalClock (name,period)>
6  <!ELEMENT period (#PCDATA)>
7  <!ELEMENT clock (name,reset?,period,pll)>
8  <!ELEMENT reset (#PCDATA)>
9  <!ELEMENT pll (#PCDATA)>
10 <!ELEMENT ccsDepth (#PCDATA)>
11 <!ELEMENT rguDepth (#PCDATA)>
12 <!ELEMENT outputDir (#PCDATA)>
13 <!ELEMENT coreuse (#PCDATA)>
```

A.3.3 Chip-Level Integration Process

Listing A.11 shows the `build` function of the ChipLevelBuilder class in pseudo code. This function generates the chip-level module from the tool configuration file.

Listing A.11 Pseudo-code for the chip-level integration process

```
1  public class ChipLevelBuilder extends AbstractBuilder {
2    ...
3    public void build() {
4      hdlFile = readHdlFile(configuration.getComponentFile(), logger, configuration);
5      topLevelDfD = readTopLevelDfdConfiguration(toolConfiguration, logger);
6      hdlTopLevel = selectComponent(hdlFile, logger);
7      csChipLevel = new CSARComponent(null, toolConfiguration.getName());
8      new CSARComponent(csChipLevel, hdlTopLevel, false);
9      new GlobalTcb(csChipLevel);
10     new Crgu(csChipLevel, toolConfiguration.getClocks().size(), toolConfiguration.↩
   getCcsDepth(), toolConfiguration.getRguDepth());
11     internalizeSignals(csChipLevel);
12     connectClocksAndResets(csChipLevel, toolConfiguration);
13     csChipLevel.createChain("tcb_tdi", "tcb_tdo", StdLogic1164.STD_LOGIC);
14     hdlEntity = csChipLevel.createEntity(toolConfiguration);
15     hdlComponent = new Component(hdlEntity);
16     hdlArchitecture = csChipLevel.createArchitecture(hdlEntity);
17     hdlConfiguration = csChipLevel.createConfiguration(hdlEntity, hdlArchitecture,↩
   toolConfiguration);
18     chipLevelDfD = createChipLevelDfD(topLevelDfD);
19     update(chipLevelDfD, toolConfiguration);
20     writeHdlEntity(hdlEntity, toolConfiguration, logger);
21     writeHdlComponent(hdlComponent, toolConfiguration, logger);
22     writeHdlArchitecture(hdlArchitecture, hdlEntity, csChipLevel, toolConfiguration,↩
   logger);
23     writeHdlConfiguration(hdlConfiguration, toolConfiguration, logger);
24     writeDfDConfiguration(chipLevelDfD, toolConfiguration, logger);
25   }
26   ...
27 }
```

The builder reads the HDL component and DfD configuration files of the top-level module on lines 4 and 5. The `selectComponent` function on line 6 works in the same way as it did for the debug wrapper generation tool (refer to Sect. 6.4.3).

On line 7, the builder creates a CSARComponent object of the chip-level module in memory. It instantiates the top-level module in this chip-level module on line 8. The `false` value for the `internal` parameter of the CSARComponent object constructor causes all signals on the interface of the top-level module to be copied to the interface of the chip-level module. On line 9, the builder instantiates the global TCB in the chip-level module. This TCB is a standardized CSAR component, so its signals are automatically matched with and connected to the corresponding input and output signals of the top-level module. The builder instantiates a custom CRGU on line 10, based on the options specified in the tool configuration.

The builder subsequently internalizes all test and debug control signals between the top-level module, the global TCB, and the CRGU on line 11, thereby removing them from the interface of the chip-level module. The builder uses a static, internal list of signal names for this purpose. On line 12, the builder connects the clock and reset signals of the top-level module to the CRGU, as specified in the tool configuration. The builder creates the TCB chain inside the chip-level module on line 13.

The HDL objects for the chip-level module are created on lines 14–17. A DfD configuration for the chip-level module is created on line 18 and includes the DfD configuration of the top-level module. It is extended with the information on the connectivity between the top-level module, the global TCB, and the CRGU on line 19. The generated chip-level module is written to HDL files on lines 20–23. The DfD configuration is written to file on line 24.

A.3.4 Executing the Chip-Level Integration Process

Listing A.12 shows an example execution run of the chip-level integration tool.

Listing A.12 Command line example of the chip-level integration process

```
1  Chip−level integrator v1.0
2
3  * Parsing '/home/csar/book/vhdl/rdt_csar_lib/rdt_csar_top_vga/CSARDE/rdt_csar_top_vga.↩
   clc'.
4  * Reading '../INTERFACE/rdt_csar_top_vga_cmp_pkg.p.vhdl'.
5  * Component 'rdt_csar_top_vga' selected.
6  * Reading 'rdt_csar_top_vga.dfd'.
7  * Instantiating the global TCB.
8  * Instantiating a custom CRGU.
9  * Writing '../../rdt_csar_chip_vga/INTERFACE/rdt_csar_chip_vga.e.vhdl'.
10 * Writing '../../rdt_csar_chip_vga/INTERFACE/rdt_csar_chip_vga_cmp_pkg.p.vhdl'.
11 * Writing '../../rdt_csar_chip_vga/RTL/rdt_csar_chip_vga_rtl.a.vhdl'.
12 * Writing '../../rdt_csar_chip_vga/RTL/rdt_csar_chip_vga_rtl_cfg.c.vhdl'.
13 * Writing '../../rdt_csar_chip_vga/CSARDE/rdt_csar_chip_vga.dfd'.
```

Note how the global TCB and a custom CRGU are instantiated on lines 7 and 8.

A.4 Boundary-Scan Level Integration

A.4.1 Tool Overview

The boundary-scan-level integration tool integrates the chip-level module with a custom TAP controller and boundary-scan chain (refer to Fig. 5.2). This tool reads the boundary-scan-level configuration file that is described in Sect. A.4.2. Based on this configuration, it reads the HDL component and DfD configuration files of the chip-level module. It outputs the HDL files of the boundary-scan-level module and a boundary-scan-level DfD configuration file.

A.4.2 Bofundary-Scan-Level Configuration

Listing A.13 shows an example boundary-scan-level specification file, which we use to generate the boundary-scan-level module for the CSAR SOC in our case study in Chap. 8.

Listing A.13 Example configuration file for boundary-scan-level integration

```
<boundaryScanLevelConfiguration>
  <name>rdt_csar_chip_bs_vga</name>
  <componentFile>../../rdt_csar_chip_vga/INTERFACE/rdt_csar_chip_vga_cmp_pkg.p.↩
vhdl</componentFile>
  <dfdFile>rdt_csar_chip_vga.dfd</dfdFile>
  <tapController>
    <instruction>
      <name>PROGRAM_TCBS</name>
      <opcode>0100</opcode>
      <capture>tcb_capture</capture>
      <shift>tcb_shift</shift>
      <update>tcb_update</update>
      <output>tcb_tdo</output>
    </instruction>
    <instruction>
      <name>PROGRAM_TPRS</name>
      <opcode>0101</opcode>
      <capture>tpr_capture</capture>
      <shift>tpr_shift</shift>
      <update>tpr_update</update>
      <output>tpr_tdo</output>
    </instruction>
    <instruction>
      <name>DBG_SCAN</name>
      <opcode>0110</opcode>
      <shift>dbg_clock_req</shift>
      <output>dbg_so</output>
    </instruction>
    <instruction>
      <name>DBG_RESET_CCS</name>
      <opcode>0111</opcode>
      <shift>dbg_reset_ccs</shift>
      <output>bypass</output>
    </instruction>
    <instruction>
      <name>DBG_CLOCK_STOP</name>
      <opcode>1000</opcode>
      <shift>dbg_clock_stop</shift>
      <output>bypass</output>
    </instruction>
    <instruction>
      <name>DBG_RESET</name>
      <opcode>1001</opcode>
      <shift>dbg_reset</shift>
      <output>bypass</output>
    </instruction>
  </tapController>
  <unwrapped>
    <signal>ext_clk</signal>
  </unwrapped>
  <outputDir>../../rdt_csar_chip_bs_vga/</outputDir>
  <coreuse>true</coreuse>
</boundaryScanLevelConfiguration>
```

The <boundaryScanLevelConfiguration> element indicates the start of the configuration data for the boundary-scan-level integration process. The <name> element specifies the name of the boundary-scan-level module. The <componentFile> and <dfdFile> elements specify respectively the HDL component file and the DfD configuration file of the chip-level module.

The <tapController> element groups the <instruction> subelements that specify the instructions of the TAP controller in the boundary-scan-level module. Each <instruction> element specifies the name of a TAP instruction and its opcode. The instructions specified in Listing A.13 are the same as listed in Table 5.1 on p. 98. The mandatory BYPASS, EXTEST, PRELOAD, and SAMPLE instructions do not need to be defined in the configuration file, because they are mandatory and therefore automatically included during the instantiation of the TAP controller. The configuration file also specifies the signals, if any, that the TAP controller asserts in the CDR, SDR, and UDR states, using respectively the <capture>, <shift>, and <update> subelements. These signals correspond to the I/O interface of the TAP controller (refer to Fig. 5.3). Finally, it specifies for each instruction an output signal using the <output> subelement. This signal is connected to the TDO pin of the SOC in the SDR state, when the associated instruction is the active instruction. A special, reserved signal name "bypass" is available for the <output> subelement to specify the output of the bypass register inside the TAP controller as the data register between the TDI and TDO pins.

The <unwrapped> element and <signal> subelement permit specifying, using a regular expression, those chip-level signals that should not be included in the boundary scan chain. We specify on line 48 that the external clock input of our SOC should be left unwrapped, because not all backends to our DfD flow support the isolation of a clock input with a boundary-scan-chain cell. The <outputDir> and <coreuse> elements control the generation of the output files, as they do for all our tools.

Listing A.14 shows the complete DTD of the configuration file for the boundary-scan-level integration process.

Listing A.14 DTD for the boundary-scan-level configuration file

```
1  <!ELEMENT boundaryScanLevelConfiguration (name,componentFile,dfdFile, ↩
   tapController,unwrapped?,outputDir?,coreuse?)>
2  <!ELEMENT name (#PCDATA)>
3  <!ELEMENT componentFile (#PCDATA)>
4  <!ELEMENT dfdFile (#PCDATA)>
5  <!ELEMENT tapController (instruction+)>
6  <!ELEMENT instruction (name,opcode,capture?,shift,update?,output)>
7  <!ELEMENT opcode (#PCDATA)>
8  <!ELEMENT capture (#PCDATA)>
9  <!ELEMENT shift (#PCDATA)>
10 <!ELEMENT update (#PCDATA)>
11 <!ELEMENT output (#PCDATA)>
12 <!ELEMENT unwrapped (signal)+>
13 <!ELEMENT signal (#PCDATA)>
14 <!ELEMENT outputDir (#PCDATA)>
15 <!ELEMENT coreuse (#PCDATA)>
```

A.4.3 Boundary-Scan-Level Integration Process

Listing A.15 shows the `build` function of the BoundaryScanLevelBuilder class in pseudo code. This function generates the boundary-scan-level module from the tool configuration file.

Listing A.15 Pseudo-code for the boundary-scan-level integration process

```
1  public class BoundaryScanLevelBuilder extends AbstractBuilder {
2    ...
3    public void build() {
4      hdlFile = readHdlFile(toolConfiguration, logger);
5      chipLevelDfD = readChipLevelDfdConfiguration(toolConfiguration, logger);
6      hdlChipLevel = selectComponent(hdlFile, logger);
7      csBoundaryScanLevel = new CSARComponent(null, toolConfiguration.getName());
8      csChipLevel = new CSARComponent(csBoundaryScanLevel, hdlChipLevel, false);
9      new TapController(csBoundaryScanLevel, toolConfiguration);
10     isolated = collectSignalsToIsolate(csChipLevel);
11     width = calculateBoundaryScanWidth(isolated);
12     new BoundaryScanChain(csBoundaryScanLevel, width, isolated);
13     mapBoundaryScanChain(csBoundaryScanLevel, csChipLevel, isolated);
14     internalizeSignals(csBoundaryScanLevel);
15     connectChains(csBoundaryScanLevel);
16     hdlEntity = csBoundaryScanLevel.createEntity(toolConfiguration);
17     hdlComponent = new Component(hdlEntity);
18     hdlArchitecture = csBoundaryScanLevel.createArchitecture(hdlEntity);
19     hdlConfiguration = csBoundaryScanLevel.createConfiguration(hdlEntity, ↩
   hdlArchitecture, toolConfiguration);
20     boundaryScanLevelDfD = createBoundaryScanLevelDfD(chipLevelDfD);
21     update(boundaryScanLevelDfD, toolConfiguration, isolated);
22     writeHdlEntity(hdlEntity, toolConfiguration, logger);
23     writeHdlComponent(hdlComponent, toolConfiguration, logger);
24     writeHdlArchitecture(hdlArchitecture, hdlEntity, csBoundaryScanLevel,↩
   toolConfiguration, logger);
25     writeHdlConfiguration(hdlConfiguration, toolConfiguration, logger);
26     writeDfDConfiguration(boundaryScanLevelDfD, toolConfiguration, logger);
27   }
28   ...
29 }
```

The builder reads the HDL component and DfD configuration files of the chip-level module on lines 4 and 5. The `selectComponent` function on line 6 works in the same way as it did in the debug wrapper generation tool (refer to Sect. 6.4.3).

On line 7, the builder creates a CSARComponent object of the boundary-scan-level module in memory. It instantiates the chip-level module in this boundary-scan-level module on line 8. The `false` value for the `internal` parameter of the constructor causes all signals on the interface of the chip-level module to be copied to the interface of the boundary-scan-level module.

On line 9, the builder instantiates a new TAP controller in the boundary-scan-level module. This TAP controller is parametrized with the information from the tool

configuration. It has a standardized interface, so its signals are automatically matched with and connected to the corresponding input and output signals of the chip-level module. The builder first determines the signals on the chip-level interface that should be isolated and stores them in the data structure `isolated` on line 10, and then determines their combined bit width on line 11. It instantiates a custom boundary scan chain on line 12.

On line 13, the builder maps the primary inputs of the boundary-scan-level module that need to be isolated to the generic primary inputs on the boundary-scan chain and the generic primary outputs on this chain to the primary outputs of the boundary-scan-level module. The builder internalized the standardized signals between the TAP controller, the boundary scan chain, and the chip-level module on line 14, thereby removing those signals from the interface of the boundary-scan-level module.

The builder hooks up the TPR and TCB inputs, outputs, clocks, and resets on line 15, together with the serial input of the debug scan chain.

The HDL objects for the boundary-scan-level module are created on lines 16–19. A DfD configuration for the boundary-scan-level module is created on line 20, based on the DfD information of the chip-level module. It is extended on line 21 with information on the connectivity between the chip-level module, the TAP controller, and the boundary scan chain. The generated boundary-scan-level module is written to HDL files on lines 22–25. The DfD configuration file is written on line 26.

A.4.4 Executing the Boundary-Scan-Level Integration Process

Listing A.16 shows an example execution run of the boundary-scan-level integration tool.

Listing A.16 Command line example of the boundary-scan-level integration process

```
1  Boundary−scan−level integrator v1.0
2
3  * Reading '/home/csar/book/vhdl/rdt_csar_lib/rdt_csar_chip_vga/CSARDE/↩
   rdt_csar_chip_vga.bslc'.
4  * Reading '../../rdt_csar_chip_vga/INTERFACE/rdt_csar_chip_vga_cmp_pkg.p.vhdl'.
5  * Component 'rdt_csar_chip_vga' selected.
6  * Reading 'rdt_csar_chip_vga.dfd'.
7  * Instantiating the TAP controller.
8  * Generating a custom boundary scan chain.
9     − Excluding the 'ext_clk' signal.
10 * Instantiating the boundary scan chain.
11 * Writing '../../rdt_csar_chip_bs_vga/INTERFACE/rdt_csar_chip_bs_vga.e.vhdl'.
12 * Writing '../../rdt_csar_chip_bs_vga/INTERFACE/rdt_csar_chip_bs_vga_cmp_pkg.p.vhdl'.
13 * Writing '../../rdt_csar_chip_bs_vga/RTL/rdt_csar_chip_bs_vga_rtl.a.vhdl'.
14 * Writing '../../rdt_csar_chip_bs_vga/RTL/rdt_csar_chip_bs_vga_rtl_cfg.c.vhdl'.
15 * Writing '../../rdt_csar_chip_bs_vga/CSARDE/rdt_csar_chip_bs_vga.dfd'.
```

Note how the TAP controller and boundary scan chain are instantiated on respectively line 7 and 10. The external clock signal "ext_clk" is left unwrapped on line 9, as specified on line 48 in the configuration file in Listing A.13.

A.5 DfD Configuration Files

Listing A.17 shows the DTD of the DfD configuration files that are used throughout the DfD flow in Chap. 6 and in this appendix. From the <architecture> and <communication> elements down, this DfD reuses the DTDs, as used in [3, 4].

Listing A.17 Design-for-debug configuration DTD file

```
<!ELEMENT boundaryScanLevel (name,tapController,boundaryScanChain,chipLevel)>
<!ELEMENT name (CDATA)>
<!ELEMENT tapController (name,tapInstruction*)>
<!ELEMENT tapInstruction (name,opcode,capture?,shift,update?,output)>
<!ELEMENT opcode (CDATA)>
<!ELEMENT capture (CDATA)>
<!ELEMENT shift (CDATA)>
<!ELEMENT update (CDATA)>
<!ELEMENT output (CDATA)>
<!ELEMENT boundaryScanChain (name,flipflops+)>
<!ELEMENT chipLevel (name,tcbChain,topLevel)>
<!ELEMENT tcbChain (tcbComponent+)>
<!ELEMENT tcbComponent (name,type,tcbParameter*)>
<!ELEMENT type (CDATA)>
<!ELEMENT tcbParameter (name,value)>
<!ELEMENT value (CDATA)>
<!ELEMENT topLevel (name,testWrapper+)>
<!ELEMENT testWrapper (name,scanWidth,pio,debugWrapper)>
<!ELEMENT scanWidth (CDATA)>
<!ELEMENT pio (pioSignal+)>
<!ELEMENT pioSignal (name,tieOffValue)>
<!ELEMENT tieOffValue (CDATA)>
<!ELEMENT debugWrapper (name,tprChain?,logicChains?,memoryChain?,core)>
<!ELEMENT tprChain (input,output,capture?,shift,update?,tprComponent+)>
<!ELEMENT input (CDATA)>
<!ELEMENT tprComponent (name,type,tprParameter*)>
<!ELEMENT tprParameter (name,value)>
<!ELEMENT logicChains (logicChain+)>
<!ELEMENT logicChain (name,input,output,ff*)>
<!ELEMENT ff (name)>
<!ELEMENT memoryChain (input,output,testEnable,shiftEnable,memoryWrapper+)>
<!ELEMENT testEnable (CDATA)>
<!ELEMENT shiftEnable (CDATA)>
<!ELEMENT memoryWrapper (name,addressWidth,dataWidth,dualPort)>
<!ELEMENT addressWidth (CDATA)>
<!ELEMENT dataWidth (CDATA)>
<!ELEMENT dualPort (CDATA)>
<!ELEMENTcore(name,tprChain?,memoryChain?,architecture?,communication?)>

<!ELEMENTarchitecture(parameter*,clk*,ip*,connect*,constraint*)>
<!ATTLISTarchitectureidCDATA#REQUIRED>
<!ELEMENTparameterEMPTY>
<!ATTLISTparameteridCDATA#REQUIRED>
<!ATTLISTparametertypeCDATA#REQUIRED>
<!ATTLISTparametervalueCDATA#REQUIRED>
<!ATTLISTparameterunitCDATA#IMPLIED>
```

```
<!ELEMENTclkEMPTY>
<!ATTLISTclkidCDATA#REQUIRED>
<!ATTLISTclkperiodCDATA#REQUIRED>
<!ATTLISTclkdutyCDATA#IMPLIED>
<!ELEMENTip(parameter *,port+)>
<!ATTLISTipidCDATA#REQUIRED>
<!ATTLISTiptypeCDATA#REQUIRED>
<!ELEMENTport(parameter *)>
<!ATTLISTportidCDATA#REQUIRED>
<!ATTLISTporttype(Initiator|Target)#REQUIRED>
<!ATTLISTportprotocol(MMIO_DTL|FIFO_AE|FLIT_AE|EDI|TPR|CFG_AE)#REQUIRED>
<!ELEMENTconnect(endpoint,endpoint)
<!ELEMENTendpointEMPTY>
<!ATTLISTendpointipCDATA#REQUIRED>
<!ATTLISTendpointportCDATA#REQUIRED>
<!ELEMENTconstraintEMPTY>
<!ATTLISTconstraintidCDATA#REQUIRED>
<!ATTLISTconstraintportCDATA#REQUIRED>
<!ATTLISTconstraintniCDATA#IMPLIED>
<!ELEMENTcommunication(schedule?,application+,constraint*)>
<!ELEMENTschedule(event *)>
<!ELEMENTeventEMPTY>
<!ATTLISTeventusecaseCDATA#REQUIRED>
<!ATTLISTeventtype(setup|teardown)#REQUIRED>
<!ATTLISTeventtimeCDATA#REQUIRED>
<!ELEMENTapplication(connection+,tree *)>
<!ATTLISTapplicationidCDATA#REQUIRED>
<!ELEMENTconnection(initiator,target,read?,write?,parameter*)>
<!ATTLISTconnectionidCDATA#REQUIRED>
<!ATTLISTconnectionqos(BE |GT|GTN)#REQUIRED>
<!ELEMENTinitiatorEMPTY>
<!ATTLISTinitiatoripCDATA#REQUIRED>
<!ATTLISTinitiatorportCDATA#REQUIRED>
<!ELEMENTtargetEMPTY>
<!ATTLISTtargetipCDATA#REQUIRED>
<!ATTLISTtargetportCDATA#REQUIRED>
<!ELEMENTreadEMPTY>
<!ELEMENTwriteEMPTY>
<!ATTLISTreadbwCDATA#REQUIRED>
<!ATTLISTreadburstsizeCDATA#IMPLIED>
<!ATTLISTreadlatencyCDATA#IMPLIED>
<!ATTLISTwritebwCDATA#REQUIRED>
<!ATTLISTwriteburstsizeCDATA#IMPLIED>
<!ATTLISTwritelatencyCDATA#IMPLIED>
<!ELEMENTtree(channel+)>
<!ATTLISTtreequeue(merged|split)#REQUIRED>
<!ELEMENTchannelEMPTY>
<!ATTLISTchannelconnectionCDATA#REQUIRED>
<!ATTLISTchanneldirection(request|response)#REQUIRED>
<!ELEMENTconstraintEMPTY>
<!ATTLISTconstrainttype(allow|disallow)#REQUIRED>
<!ATTLISTconstraintapplCDATA#REQUIRED>
<!ATTLISTconstraintwithCDATA#REQUIRED>
```

This DTD contains a description of the order of the TCBs and TPRs in their respective chains connected to the TAP controller.

Finally, Table A.1 shows the possible TCB and TPR module types and their parameters. These parameters need to be chosen at implementation time.

Table A.1 CSAR debug hardware module types and their parameters

Module	Type	Parameter	Description
DTL monitor	dtlMonitor	addressWidth	Number of address bits
		dataWidth	Number of data bits
		nrEdis	Number of EDI layers
		stateBits	Number of state bits
		poly	Polynomial to use for checksum
DTL PSI	dtlPsi	counterWidth	Number of bits in pending transaction counter
		nrEdis	Number of EDI layers
EDI node	ediNode	nrEdis	Number of EDI layers
		nrPorts	Number of EDI ports
Memory wrapper	memoryWrapper	addressWidth	Number of address bits
		dataWidth	Number of data bits
		dualPort	Flag indicating a dual-port memory
TAP-DTL bridge	tapDtlBridge	addressWidth	Number of address bits
		dataWidth	Number of data bits
		maskWidth	Number of mask bits
TAP-EDI bridge	tapEdiBridge	nrEdis	Number of EDI layers
Global TCB	globalTcb	–	–
CRGU TCB	crguTcb	nrClocks	Number of functional clocks

References

1. Cadence. *LEF/DEF Language Reference - Product Version 5.7*. Cadence Design Systems, July 2011.
2. Cadence. *Using Encounter RTL Compiler - Product Version 11.1*. Cadence Design Systems, 2011.
3. 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*, pages 1182–1187, Washington, DC, USA, March 2005. IEEE Computer Society Press.
4. Andreas Hansson and Kees Goossens. Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases. In *Proc. International Symposium on Networks on Chip*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society Press.

Appendix B
CSAR SOC

Abstract In this appendix, we provide additional implementation details of the CSAR SOC used in our case study in Chap. 8. In particular, we describe the NoC specification in Sect. B.1, the source code of the Producer task in Sect. B.2, the source code of the Consumer task in Sect. B.3, and the parametrization of the CSAR debug modules in the CSAR SOC in Sect. B.4.

B.1 NoC Specification

Table B.1 provides the communication specification of the NoC inside the CSAR SOC. The resulting NoC block diagram is shown in Fig. B.1. The NoC core is integrated in a NoC module, which includes CDC modules on the DTL ports to the producer and consumer processors, the control and data memories, and the video subsystem. No CDC modules are needed on the interfaces between the NoC and the host processor, and between the NoC and the TAP bridge, because these three modules share the same clock signal. All connections guarantee their throughput.

Table B.1 NoC connections between the CSAR SOC modules

Nr	Master	Port	Slave	Port	Read BW [Mbit/s]	Write BW [Mbit/s]
1	Producer	m	Control memory	s	7.0	6.0
2	Producer	m	Data memory	s	7.0	42.0
3	Consumer	m	Control memory	s	7.0	4.0
4	Consumer	m	Data memory	s	23.0	4.0
5	Consumer	m	Video subsystem	s	1.0	1.0
6	Bridge	m	Producer	s	0.3	0.3
7	Bridge	m	Consumer	s	0.3	0.3
8	Bridge	m	Control memory	s	0.3	0.3
9	Bridge	m	Data memory	s	0.3	0.3
10	Bridge	m	Video subsystem	s	0.3	0.3
11	Bridge	m	Producer	t	0.3	0.3
12	Bridge	m	Consumer	t	0.3	0.3
13	Bridge	m	Control memory	t	0.3	0.3
14	Bridge	m	Data memory	t	0.3	0.3
15	Bridge	m	Video subsystem	t	0.3	0.3

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6,

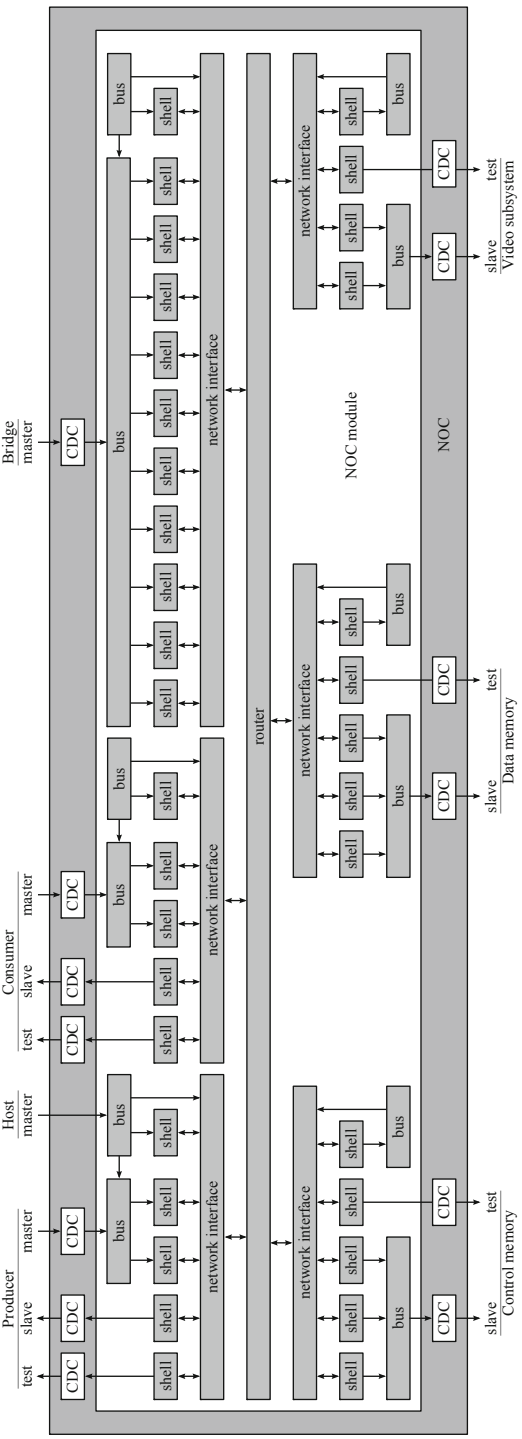


Fig. B.1 NoC module with generated NoC core and CDC modules

B.2 Producer Code

The producer processor in the CSAR SOC executes the Producer task. The pseudo code of this task is shown in Listing B.1.

Listing B.1 Pseudo code of the Producer task

```
int main()
{
  BUFFER_MASK = 0xFF0003FF;
  BUFFER_SIZE = 256;
  read_ptr_addr = 0x00000000;
  write_ptr_addr = 0x00000004;
  token_count_addr = 0x00000008;
  loop_count_addr = 0x0000000C;
  read_ptr = (1 <<27);
  write_ptr = (1 <<27);
  /*Initialize write and read pointers*/
  *write_ptr_addr = write_ptr;
  *read_ptr_addr = read_ptr;
  /*Loop forever */
  loop = 0;
  while (1) {
    /*Generate 256 tokens per loop iteration*/
    for (token=0; token< 256; token++) {
      /*Produce an interesting value*/
      value = ((~loop)<< 16) + (token<<8);
      /*Write data into FIFO*/
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      *write_ptr++ = value++;
      /*Store write pointer */
      write_ptr = write_ptr & BUFFER_MASK;
      *write_ptr_addr = write_ptr;
      /*Read back read pointer and wait until there is write space in the FIFO*/
      stall = 1;
      while (stall==1) {
        read_ptr = *read_ptr_addr;
        if (write_ptr >read_ptr) {
          space_left = BUFFER_SIZE - (write_ptr - read_ptr);
        } else {
          space_left = read_ptr - write_ptr;
        }
        if (space_left>16) {
          stall = 0;
        }
      }
```

```
54          /*Update token count*/
55          *token_count_addr = token;
56        }
57        /*Update loop count*/
58        loop++;
59        *loop_count_addr = loop;
60      }
61      return 0;
62  }
```

The Producer task initializes application constants on lines 3–10. These constants include the FIFO buffer size, the locations and values of the FIFO read and write pointers, and the locations of the token and loop counters. It subsequently initializes the FIFO read and write pointers on lines 12 and 13 to the start address of the data memory in the memory map of the producer processor. The Producer task then resets the loop counter (line 15) and enters an infinite loop on line 16.

In each loop iteration, the Producer task writes 256 tokens in the FIFO. Each token consists of 16 32-bit data words each, which are written on lines 22–37. The Producer task updates the FIFO write pointer on lines 39 and 40 after each token has been written. The FIFO may become full after writing a token, as the Consumer task may read tokens from the FIFO at a slower pace than the Producer task writes tokens into the FIFO. The Producer task therefore continuously polls the FIFO read pointer between lines 44–54, until there is space for another token in the FIFO. The Producer task subsequently increments the token counter in the control memory (line 56).

After all 256 tokens have been written, the Producer task increments the loop counter and writes it into the control memory (lines 59 and 60), before starting another loop iteration.

B.3 Consumer Code

The consumer processor in the CSAR SOC executes the Consumer task. The pseudo code of this task is shown in Listing B.2.

Listing B.2 Consumer pseudo code

```
1   int main()
2   {
3     BUFFER_MASK = 0xFF0003FF;
4     BUFFER_SIZE = 256;
5     read_ptr_addr = 0x00000000;
6     write_ptr_addr = 0x00000004;
7     ref = 0;
8     read_ptr = (1 << 27);
9     write_ptr = (1 << 27);
10    video_addr = 0x10000000;
11    /*Wait for the producer to initialize the read pointer*/
12    stall=1;
13    while (stall==1) {
14      read_ptr = *read_ptr_addr;
15      if (read_ptr == 0x08000000) {
```

```
16          stall=0;
17        }
18      }
19      /*Loop forever*/
20      loop = 0;
21      while (1){
22        /*Read 256 tokens per loop iteration*/
23        token = 0;
24        error = 2;
25        while (token<256) {
26          /*Read back write pointer and wait until there is data in the FIFO*/
27          stall = 1;
28          while (stall==1) {
29            write_ptr = *write_ptr_addr;
30            if (write_ptr > = read_ptr) {
31              data_left = write_ptr - read_ptr;
32            } else {
33              data_left = BUFFER_SIZE - (read_ptr - write_ptr);
34            }
35            if (data_left> =16) {
36              stall = 0;
37            }
38          }
39          /* Read data from FIFO */
40          value = *read_ptr++;
41          ref = ((~loop)<<16) + (token<<8);
42          if ( value != ref ) { error |= 1; }
43          value = *read_ptr++;
44          ref++;
45          if ( value != ref ) { error |= 1; }
46          value = *read_ptr++;
47          ref++;
48          if ( value != ref ) { error |= 1; }
49          value = *read_ptr++;
50          ref++;
51          if ( value != ref ) { error |= 1; }
52          value = *read ptr++;
53          ref++;
54          if ( value != ref ) { error |= 1; }
55          value = *read_ptr++;
56          ref++;
57          if ( value != ref ) { error |= 1; }
58          value = *read_ptr++;
59          ref++;
60          if ( value != ref ) { error |= 1; }
61          value = *read_ptr++;
62          ref++;
63          if ( value != ref ) { error |= 1; }
64          value = *read_ptr++;
65          ref++;
66          if ( value != ref ) { error |= 1; }
67          value = *read_ptr++;
68          ref++;
69          if ( value != ref ) { error |= 1; }
70          value = *read_ptr++;
```

```
71          ref++;
72          if ( value != ref ) { error |= 1; }
73          value = *read_ptr++;
74          ref++;
75          if ( value != ref ) { error |= 1; }
76          value = *read_ptr++;
77          ref++;
78          if ( value != ref ) { error |= 1; }
79          value = *read_ptr++;
80          ref++;
81          if ( value != ref ) { error |= 1; }
82          value = *read_ptr++;
83          ref++;
84          if ( value != ref ) { error |= 1; }
85          value = *read_ptr++;
86          ref++;
87          if ( value != ref ) { error |= 1; }
88          /*Store read pointer*/
89          read_ptr = read_ptr & BUFFER_MASK;
90          *read_ptr_addr = read_ptr;
91          if ((token & 0xF) == 0xF) {
72            results[token>> 4] = error;
93            error = 2;
94          } else {
95            error = (error << 2) + 2;
96          }
97          token++;
98        }
99        /*Write result to video controller*/
100       for (i=0; i< 16; i++) {
101         *(video_addr+i) = results[i];
102       }
103       loop++;
104     }
105     return 0;
106 }
```

The Consumer task initializes application constants on lines 3–10. These constants include the FIFO buffer size, and the locations and values of the FIFO read and write pointers. It subsequently polls the FIFO read pointer between lines 13–17 to wait until the Producer task has initialized this pointer. The Consumer task subsequently resets the loop counter (line 20) and enters an infinite loop on line 21.

In each loop iteration, the Consumer task tries to read 256 tokens from the FIFO. It initializes the token counter on line 23, and initializes the error flag with the value 2 on line 24. This value will cause the corresponding rectangle on the associated VGA display to be shown in green, i.e., indicating no error in the corresponding token. The FIFO may become empty, as the Producer task may write tokens from the FIFO at a slower pace than the Consumer task reads tokens from the FIFO. The Consumer task therefore continuously polls the FIFO write pointer between lines 28 and 38, until the FIFO contains data of another token. It subsequently reads all 16 data words of this token, and checks each data word for the expected data value, on

lines 40–87. For each data word read, the error flag is updated in case of an error, to cause the corresponding rectangle on the associated VGA display to be shown in red instead of green, to indicate an error in the corresponding token. The Consumer task updates the FIFO read pointer on lines 8990 after each token has been read. The Consumer task checks on lines 91–96 whether the error flag is full, i.e., contains the error information for 16 tokens. If so, then the current error flag value is stored in the array "results" and the error flag is reset. If not, then the error flag is shifted to make room for additional error information for the next token.

After all 256 tokens have been read and checked, the Consumer task writes the information in the "results" array to the video subsystem in the CSAR SOC on lines 100–102, effectively changing the VGA output of the CSAR SOC to reflect the collected error information. The Consumer task subsequently increments the loop counter, before starting another loop iteration.

B.4 Module Implementation Parameters

Tables B.2 and B.3 provides the parameters of the modules in the CSAR SOC.

Table B.2 Parameter values for the CSAR SOC monitor modules

Module	addressWidth	dataWidth	nrEdis	stateBits	poly
Monitors 1 and 6	10	32	3	2	0x04C11DB7
Monitor 2	14	32	3	2	0x04C11DB7
Monitors 3 and 8	10	32	3	1	0x04C11DB7
Monitor 4	14	32	3	1	0x04C11DB7
Monitor 5	13	32	3	2	0x04C11DB7
Monitor 7	10	32	3	5	0x04C11DB7

Table B.3 Parameter values for the other CSAR SOC debug modules

Module	Parameter	Value
PSIs 1–8	counterWidth	5
	nrEdis	3
TAP-DTL bridge	addressWidth	32
	dataWidth	32
	maskWidth	4
TAP-EDI bridge	nrEdis	3
CRGU	nrClocks	9
	ccsDepth	2
	rguDepth	20

Appendix C
CSARDE Grammars and Scripts

Abstract We document in this appendix the grammar of the language that is supported by the scripting engine in the CSARDE (Sect. C.1), the grammar of the domain specific language (DSL) to specify the STG for the event sequencer (Sect. C.2), and describe the scripts that were used in Chap. 8, to debug the erroneous implementations of the CSAR SOC (Sect. C.3).

C.1 CSARDE Tool Control Language Grammar

Listing C.1 shows the ANTLR4 grammar [1] of the CSARDE tool control language.

Listing C.1 Grammar for the CSARDE tool control language (Tcl)

```
grammar CsardeTcl;

prog: NEWLINE* body NEWLINE* EOF;
body: command (';'|NEWLINE+ command)*;
command: exprCommand
        | forCommand
        | foreachCommand
        | ifCommand
        | incrCommand
        | inputCommand
        | lindexCommand
        | llengthCommand
        | listCommand
        | procCommand
        | putsCommand
        | returnCommand
        | setCommand
        | sourceCommand
        | stringCommand
        | whileCommand
        | idCommand
        | variableCommand;
exprCommand: 'expr' expr;
forCommand: 'for' '{' body '}' '{' expr '}' '{' body '}'
        '{' NEWLINE* body NEWLINE* '}';
foreachCommand: 'foreach' ID VARIABLE
        '{' NEWLINE* body NEWLINE* '}';
ifCommand: 'if' '{' expr '}'
```

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6,

```
        '{' NEWLINE* body NEWLINE* '}'
        ( 'elseif' '{' expr '}' '{' NEWLINE* body NEWLINE* '}' )*
        ('else' '{' NEWLINE* body NEWLINE* '}')*;
incrCommand: 'incr' ID;
inputCommand: 'input' STRING;
lindexCommand: 'lindex' (VARIABLE|evaluation) expr;
llengthCommand: 'llength' (VARIABLE|evaluation);
listCommand: 'list' argument+;
procCommand: 'proc' ID '{' formalParameters* '}'
        '{' NEWLINE* body NEWLINE* '}';
putsCommand: 'puts' argument;
returnCommand: 'return' expr;
setCommand: 'set' ID argument;
sourceCommand: 'source' STRING;
stringCommand: 'string' arguments;
whileCommand: 'while' '{' expr '}'
        '{' NEWLINE* body NEWLINE* '}';
idCommand: ID arguments?;
variableCommand: VARIABLE arguments?;
formalParameters: formalParameter+;
formalParameter: ID;
expr: '!' expr
        | expr op=('*'|'/') expr
        | expr op=('+'|'-') expr
        | expr op=('!='|'=='|'>'|'>='|'<'|'<=') expr
        | expr op=('&&'|'||') expr
        | INT
        | VARIABLE
        | '(' expr ')'
        | evaluation;
arguments: argument+;
argument: evaluation
        | INT
        | HEX
        | ID
        | STRING
        | VARIABLE
        | '*'
        | OPTION;
evaluation: '[' command ']';

MUL: '*';
DIV: '/';
ADD: '+';
SUB: '-';
NOT: '!=';
EQ: '==';
GT: '>';
GE: '>=';
LT: '<';
LE: '<=';
AND: '&&';
OR: '||';
VARIABLE: '$' ID '{' | '$' ID;
```

```
ID: [a−zA−Z][_0−9a−zA−Z]∗;
HEX: ’0x’ [0−9a−hA−H]+;
INT: ’0’ | ’−’? [1−9][0−9]∗;
OPTION: ’−’ ID;
STRING: ’”’∗? ’”’;
COMMENT: ’#’ .∗? NEWLINE −>skip;
NEWLINE:\r’? ’\n’+;
WS: [ \t]+ −>skip;
```

C.2 CSARDE Event Sequencer Grammar

Listing C.2 shows the ANTLR4 grammar [1] used to program the CSAR event sequencer inside a CSAR DTL monitor.

Listing C.2 Grammar for the CSAR event sequencer

```
grammar CsardeEventSequencer;

sequence: transition+;
transition: STATE ’−>’ STATE condition? output? ’;’;
condition: ’when’ expr;
output:  ’output’ BITSTRING;
expr:  ’!’ expr
            | expr (’∗’ expr)+
            | expr (’+’ expr)+
            | ’(’ expr ’)’
            | CMD_VALID
            | CMD_MATCH
            | CMD_READ
            | WR_VALID
            | WR_MATCH
            | RD_VALID
            | RD_MATCH
            ;
STATE:  ’s’ INT;
INT:  ’0’ | ’− ’? [1− 9][0− 9]∗;
BITSTRING: ’\ ’’ [’0’|’1’]+ ’\ ’’;
CMD_VALID: ’cv’;
CMD_MATCH: ’cm’;
CMD_READ: ’cr’;
WR_VALID: ’wv’;
WR_MATCH: ’wm’;
RD_VALID: ’rv’;
RD_MATCH: ’rm’;
WS:  [ \ t\ r\ n]+ − >skip;
```

C.3 CSARDE Scripts

Listing C.3 shows the initialization code to configure the CSARDE for use with the CSAR SOC.

Listing C.3 Script to initialize the CSARDE for use with the CSAR SOC

```
# Query all SOC environment factories and select the first one
set envFactories [ $socManager getEnvironmentFactories ]
set envFactory [ lindex $envFactories 0 ]
# Configure the SOC environment factory
$envFactory setParameterValue url "./rdt_csar_chip_bs_vga.dfd"
# Create the SOC environment
set env [ $envFactory create $socManager ]
# Query all abstraction factories and create all available abstractions
set absFactories [ $abstractionManager getAbstractionFactories ]
foreach absFactory $absFactories {
    $absFactory create $abstractionManager
}
# Apply all abstractions to the SOC environment
$env execute $abstractionManager
# Enter the functional mode
$env gotoMode functional
# Function to set bypass enables of modules
proc  setBypassEnable { modules flag } {
  foreach  m $modules {
    $m setBypassEnable $flag
  }
}
# Function to print all checksum values
proc printChecksums { matcherList } {
  setBypassEnable $matcherList 0
  $env synchronize
  foreach m $matcherList {
    puts [ $m getChecksum ]
  }
  setBypassEnable $matcherList 1
  $env synchronize
}
# Create variables to easily refer to PSIs, data matchers, and event sequencers etc in the SOC
# (not shown for brevity ...)
```

Listing C.4 shows the CSARDE Tcl script to stop the execution of the CSAR SOC at the end of the first loop iteration, while simultaneously having the DTL monitors calculate checksum values for the transactions on their respective DTL communication links.

Listing C.4 Debug script to calculate communication checksum values

```
# Start debug experiment
source "initialization.tcl"
# Enable all data matchers and have them update their checksum values on valid transactions
foreach m $matchers {
  $m setEnable 1
  $m setMode range
  $m setCondition valid
}
# Configure producer master data matcher to generate match on command address 0x0000000C
```

```
$producerMasterCmdMatcher setValues 0x0000000C 0x0000000C
# Configure producer master event sequencer to generate an event on a write transaction to address 0x0000000C
$producerMasterEventSequencer setProgram $env "s0 -> s1 when cv*cm*!cr output '001';"
$producerMasterEventSequencer setResetN 1
$producerMasterEventSequencer setOutputEnable 1
# Configure producer master PSI to stop at transaction level on an EDI event
$producerMasterPsi setRequestStopEnable 1
$producerMasterPsi setRequestStopCondition edi
$producerMasterPsi setRequestStopGranularity transaction
# Configure consumer master data matcher to generate a match on command address 0x1000003C
$consumerMasterCmdMatcher setValues 0x1000003C 0x1000003C
# Configure consumer master event sequencer to generate an event on a write transaction to address 0x1000003C
$consumerMasterEventSequencer setProgram $env "s0 -> s1 when cv*cm*!cr output '001';"
$consumerMasterEventSequencer setResetN 1
$consumerMasterEventSequencer setOutputEnable 1
# Configure consumer master PSI to stop at transaction level on an EDI event
$consumerMasterPsi setRequestStopEnable 1
$consumerMasterPsi setRequestStopCondition edi
$consumerMasterPsi setRequestStopGranularity transaction
#CreateanEDIpropagationpathfromtheproducermastermonitortotheproducer master PSI
$edi createPath -layer 0 -from "producer m monitor" - to "producer m psi"
#CreateanEDIpropagationpathfromtheconsumermastermonitortotheconsumer master PSI
$edi createPath - layer 0 -from "consumer m monitor" - to "consumer m psi"
setBypassEnable $all 1
#ConfiguretheSOC
$env synchronize
#FunctionallyresettheSOC
$env reset
# Loop until both the producer and consumer master PSIs has stopped the communication
set producerStopped 0
set consumerStopped 0
while { $producerStopped != 1 && $consumerStopped != 1 } {
   $env synchronize
   set producerStopped [ $producerMasterPsi getCommandGroupStopped ]
   set consumerStopped [ $consumerMasterPsi getCommandGroupStopped ]
}
printChecksums $matchers
# - LocationA - RefertoListingD.6
#Enddebugexperiment
$env gotoMode disconnected
```

Listing C.5 guides the execution of the CSAR SOC, by precisely controlling the order in which the transactions are allowed to take place on the master DTL ports of the producer, consumer, and host processors. The `printChecksums` is re-used from Listing C.4.

Listing C.5 Debug script to guide the execution of the CSAR SOC

```
# Start debug experiment
source "initialization.tcl"
# Procedure to continue one particular PSI and its DTL request channel
proc continueCommunication { env psi } {
   # Request continue for request channel
   $psi setBypassEnable 0
   $env synchronize
   $psi setRequestContinue 1
   $psi setBypassEnable 1
   $env synchronize
   $psi setRequestContinue 0
   set pending [ $psi getRequestStopLeft ]
   # Wait for the request channel to leave the stop state
   while { $pending != 1 } {
      $env synchronize
      set pending [ $psi getRequestStopLeft ]
   }
}
# Procedure to wait for a pending command on the DTL request channel of a PSI
proc waitForCommandPending { env psi } {
   # Wait for a pending command on the request channel
   set pending [ $psi getCommandPending ]
   while { $pending != 1 } {
      $env synchronize
      set pending [ $psi getCommandPending ]
   }
}
proc nextCommand { env psi } {
   continueCommunication $env $psi
   waitForCommandPending $env $psi
}
# Enable all data matchers and have them update their checksum values on valid transactions
foreach m $matchers {
   $m setEnable 1
   $m setMode range
   $m setCondition valid
}
# Program the producer, consumer, and host master PSIs to stop unconditionally at transaction level
set psis [ list $producerMasterPsi $consumerMasterPsi $hostPsi ]
foreach psi $psis {
   $psi setRequestStopEnable 1
   $psi setRequestStopCondition unconditional
   $psi setRequestStopGranularity transaction
}
setBypassEnable $all 1
# Configure the SOC
$env synchronize
# Functionally reset the SOC
$env reset
# Loop until the producer master, consumer master, and host PSIs have commands pending
waitForCommandPending $env $producerMasterPsi
waitForCommandPending $env $consumerMasterPsi
```

```
waitForCommandPending $env $hostPsi
printChecksums $matchers
# Have the Host initialize the network
for {set i 0} {$i<218} {incr i} {
   nextCommand $env $hostPsi
   printChecksums $matchers
}
continueCommunication $env $hostPsi
printChecksums $matchers
# Producer initializes the write pointer
nextCommand $env $producerMasterPsi
printChecksums $matchers
# Producer initializes the read pointer
nextCommand $env $producerMasterPsi
printChecksums $matchers
# Consumer reads the initialized read pointer
nextCommand $env $consumerMasterPsi
printChecksums $matchers
# For all token
for {set i 0} {$i <256} {incr i} {
   for {set j 0} {$j <16 } {incr j} {
      # Producer writes token
      nextCommand $env $producerMasterPsi
      printChecksums $matchers
   }
   # Producer updates write pointer
   nextCommand $env $producerMasterPsi
   printChecksums $matchers
   # Producer reads read pointer
   nextCommand $env $producerMasterPsi
   printChecksums $matchers
   # Producer updates token count
   nextCommand $env $producerMasterPsi
   printChecksums $matchers
   # Consumer reads updated write pointer
   nextCommand $env $consumerMasterPsi
   printChecksums $matchers
   for {set j 0} {$j <16} {incr j} {
      # Consumer reads token
      nextCommand $env $consumerMasterPsi
      printChecksums $matchers
   }
   # Consumer updates read pointer
   nextCommand $env $consumerMasterPsi
   printChecksums $matchers
}
# Consumer writes its status flags to the Video task
 for {set i 0} {$i <16} {incr i} {
 nextCommand $env $consumerMasterPsi
 printChecksums $matchers
 }
 # End debug experiment
 $env gotoMode disconnected
```

Listing C.6 needs to be inserted at the end of Listing C.4 where it says "– Location A –" to print the content of the register file of the video subsystem and the content of the data memory.

Listing C.6 Additional code to print the content of the register file of the video subsystem and the content of the data memory

```
...
$env gotoMode clocks_stopped
$env gotoMode debug_scan
$env synchronize
# Print register file content in video subsystem
$videoSubsystem printRegisterFile
# Print state of data memory
$dataMemory printContent $env
...
```

Listing C.7 stops the execution of the CSAR SOC immediately after the Producer task has completely written Token 17 to the data memory. The content of the data memory is subsequently printed for inspection.

Listing C.7 Debug script to print the content of the data memory after Token 17 has been completely written by the Producer task

```
# Start debug experiment
source "initialization.tcl"
# Make list of relevant matchers to configure
set matchers [ list $producerMasterCmdMatcher $producerMasterWrMatcher ]
# Enable all relevant data matchers
foreach m $matchers {
  $m setEnable 1
  $m setMode range
}
# Configure producer master data matcher to generate match on command address 0x00000008
$producerMasterCmdMatcher setValues 0x00000008 0x00000008
# Configure producer master data matcher to generate match on write data 0x00000011
$producerMasterWrMatcher setValues 0x00000011 0x00000011
# Configure producer master event sequencer to generate an event on a write transaction to address 0x00000008 with write data 0x00000011
$producerMasterEventSequencer setProgram $env "s0 -> s0 when !cv+!cm+cr output '000'; s0 -> s1 when cv*cm*!cr*!wv output '000'; s0 -> s2 when cv*cm*!cr*wv*wm output '001'; s1 -> s1 when !wv output '000'; s1 -> s0 when wv*!wm output '000'; s1 -> s2 when wv*wm output '001'; s2 -> s2 output '000';"
$producerMasterEventSequencer setResetN 1
$producerMasterEventSequencer setOutputEnable 1
# Create an EDI propagation path from the producer master monitor to the producer master PSI and consumer master PSI
$edi createPath -layer 0 -from "producer m monitor" -to "producer m psi, consumer m psi"
# Configure producer master PSI to stop at transaction level on an EDI event
$producerMasterPsi setRequestStopEnable 1
$producerMasterPsi setRequestStopCondition edi
$producerMasterPsi setRequestStopGranularity transaction
# Configure consumer master PSI to stop at transaction level on an EDI event
$consumerMasterPsi setRequestStopEnable 1
$consumerMasterPsi setRequestStopCondition edi
```

```
$consumerMasterPsi setRequestStopGranularity transaction
setBypassEnable $all 1
# Configure the SOC
$env synchronize
# Functionally reset the SOC
$env reset
# Loop until the producer and consumer master PSIs have stopped the communication
set producerStopped 0
set consumerStopped 0
while { $producerStopped != 1 && $consumerStopped != 1 } {
   $env synchronize
   set producerStopped [ $producerMasterPsi getCommandGroupStopped ]
   set consumerStopped [ $consumerMasterPsi getCommandGroupStopped ]
}
# Extract SOC state
$env gotoMode clocks_stopped
$env gotoMode debug_scan
$env synchronize
# Print register file content in video subsystem
$videoSubsystem printRegisterFile
# Print state of data memory
$dataMemory printContent $env
# End debug experiment
$env gotoMode disconnected
```

Listing C.8 stops the execution of the CSAR SOC immediately after the Producer task has written data word 4 of Token 17 to the data memory. We then print the content of the data memory, and the content of the CDC module on the output of the NoC towards the data memory. The DTL communication link between the NoC and the data memory is then single-stepped, and the content of the data memory is printed again to observe the incorrect transfer of data word 4 to the data memory. Afterwards, we correct the content of the data memory and resume the execution of the SOC till the end of the first iteration loop. At that point, we stop the execution of the SOC again, and print the content of the register file in the video subsystem.

Listing C.8 Debug script to correct the effects of a previously-localized failure

```
# Start debug experiment
source "initialization.tcl"
# Make list of relevant matchers to configure
set matchers [ list $producerMasterCmdMatcher $producerMasterWrMatcher ]
# Enable all relevant data matchers
foreach m $matchers {
   $m setEnable 1
   $m setMode range
}
# Configure producer master data matcher to generate a match on write command address 0x08000050, write data 0x00001104
$producerMasterCmdMatcher setValues 0x08000050 0x08000050
$producerMasterWrMatcher setValues 0x00001104 0x00001104
# Configure producer master event sequencer to generate an event on a write transaction to address 0x08000050, data 0x00001104
```

```
14 $producerMasterEventSequencer setProgram $env "s0 -> s0 when !cv+!cm+cr output
   '000'; s0 -> s1 when cv*cm*!cr*!wv output '000'; s0 -> s2 when cv*cm*!cr*wv*wm↩
   output '001'; s1 -> s1 when !wv output '000'; s1 -> s0 when wv*!wm output '000'; s1 ↩
   -> s2 when wv*wm output '001'; s2 -> s2 output '000';"
15 $producerMasterEventSequencer setResetN 1
16 $producerMasterEventSequencer setOutputEnable 1
17 # Create an EDI propagation path from the producer master monitor to the producer↩
   master PSI, consumer master PSI, and data memory slave PSI
18 $edi createPath -layer 0 -from "producer m monitor" -to "producer m
19 psi, consumer m psi, data memory s psi"
20 # Configure producer master PSI to stop at transaction level on an EDI event
21 $producerMasterPsi setRequestStopEnable 1
22 $producerMasterPsi setRequestStopCondition edi
23 $producerMasterPsi setRequestStopGranularity transaction
24 # Configure consumer master PSI to stop at transaction level on an EDI event
25 $consumerMasterPsi setRequestStopEnable 1
26 $consumerMasterPsi setRequestStopCondition edi
27 $consumerMasterPsi setRequestStopGranularity transaction
28 # Configure data memory slave PSI to stop at transaction level on an EDI event
29 $datamemSlavePsi setRequestStopEnable 1
30 $datamemSlavePsi setRequestStopCondition edi
31 $datamemSlavePsi setRequestStopGranularity transaction
32 setBypassEnable $all 1
33 # Configure the SOC
34 $env synchronize
35 # Functionally reset the SOC
36 $env reset
37 # Loop until the producer master, the consumer master, and the data memory slave PSIs
   have stopped the communication
38 set producerStopped 0
39 set consumerStopped 0
40 set datamemStopped 0
41 while { $producerStopped != 1 && $consumerStopped != 1 && $datamemStopped != 1 }↩
   {
42   $env synchronize
43   set producerStopped [ $producerMasterPsi getCommandGroupStopped ]
44   set consumerStopped [ $consumerMasterPsi getCommandGroupStopped ]
45   set datamemStopped [ $datamemSlavePsi getCommandGroupStopped ]
46 }
47 # Extract SOC state
48 $env gotoMode clocks stopped
49 $env gotoMode debug scan
50 $env synchronize
51 # Print state of NOC CDC module
52 $nocCdcModule printContent -commandFIFO -writeFIFO
53 # Print state of data memory
54 $dataMemory printContent $env
55 # Restore SOC state
56 $env synchronize
57 # Single- step the write transaction pending at the data memory PSI
58 $env gotoMode clocks stopped
59 $env gotoMode functional
60 continueCommunication $env $datamemSlavePsi
61 waitForCommandPending $env $datamemSlavePsi
```

```
# Extract SOC state
$env gotoMode clocks stopped
$env gotoMode debug scan
$env synchronize
# Print the state of the data memory
$dataMemory printContent $env
# Correct the value in the data memory
writeMemory $env $dataMemory 0x00000050 0x00001104
# Configure producer master data matcher to generate match on command address 0↩
x0000000C, write data 0x00000001
$producerMasterCmdMatcher setValues 0x0000000C 0x0000000C
$producerMasterWrMatcher setValues 0x00000001 0x00000001
# Disable DTL PSIs at the data memory
$datamemSlavePsi setRequestStopEnable 0
# Reset producer master event sequencer
$producerMasterEventSequencer reset $env
# Restore SOC state
$env synchronize
# Continue producer and consumer
$env gotoMode clocks stopped
$env gotoMode functional
continueCommunication $env $producerMasterPsi
continueCommunication $env $consumerMasterPsi
# Loop until the producer and consumer master PSIs have stopped the communication
set producerStopped 0
set consumerStopped 0
while { $producerStopped != 1 && $consumerStopped != 1 } {
   $env synchronize
   set producerStopped [ $producerMasterPsi getCommandGroupStopped ]
   set consumerStopped [ $consumerMasterPsi getCommandGroupStopped ]
}
# Extract SOC state
$env gotoMode clocks stopped
$env gotoMode debug scan
$env synchronize
# Print register file content in video subsystem
$videoSubsystem printRegisterFile
# End debug experiment
$env gotoMode disconnected
```

References

1. T.J. Parr and R.W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

Glossary

C	Number of clocks
E	Number of EDI layers
FF	Number of flip-flops
I	FSM input
L	STG edge label set
N_{es}	Number of state bits in an event sequencer
N_{pll}	Number of PLLs
O	FSM output
P	Number of EDI ports
T	Clock period
T_{min}	Minimal clock period
W	Width
W_A	Address width
W_D	Data width
W_M	Mask width
$X(l)$	Input symbol for STG label l
$Y(l)$	Output symbol for STG label l
a	Number of possible input symbols
b	Number of possible output symols
d_i	Data element i
e_{ij}	STG edge from state s^i to state s^j
e_i^x	Event x in process p_i of a distributed system
$f_s(v)$	State corresponding to vertex v
$f_v(s)$	Vertex corresponding to state s
$f_{abstract}$	Abstraction function
i^n	FSM input symbol n
l	STG edge label
m_{ij}	Message from process p_i to prcess p_j
o^n	FSM output symbol n
p_n	Process n in a distributed system
$s(t)$	FSM state at time t
s	FSM state

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6,

s_i^X	Super-state X of building block i
s_i^j	State j of building block i
s^0	FSM initial state
s_{glb}^x	SOC global state x
t	Time
t_h	Flip-flop hold time
t_s	Flip-flop setup time
$t_{ae}(n)$	Time of active edge n
$u(t)$	Quantized time at time t
v, w	Vertices
$x(t)$	FSM input symbol at time t
x	FSM input symbol
$y(t)$	FSM output symbol at time t
y	FSM output symbol
Δt	Time interval
Δt_{c2q}	Flip-flop clock-to-output delay
Δt_{cd}	Clock distribution delay
Δt_{pd}	Data propagation delay
Δt_{sh}	Setup-and-hold interval
Δ_ϕ	Phase difference
$\delta(s, i)$	FSM next-state function for state s and input symbol i
$\mathcal{E}$	STG edge set
$\mathcal{G}$	state transition graph
$\mathcal{I}$	FSM input alphabet
$\lambda(s, i)$	FSM output function for state s and input symbol i
$\mathcal{M}$	FSM 6-tuple
$\mathcal{O}$	FSM output alphabet
ϕ	Clock phase
$\mathcal{S}$	Set of FSM states
$\mathcal{V}$	STG vertex set

Index

B. Vermeulen, K. Goossens, *Debugging Systems-on-Chip,*
Embedded Systems, DOI 10.1007/978-3-319-06242-6,